PROGRAMAÇÃO MODULAR

Programação Orientada por Objetos - Java

Roberto S. Bigonha

PROGRAMAÇÃO MODULAR

Programação Orientada por Objetos

Java

Roberto S. Bigonha

Belo Horizonte, MG

29 de Junho de 2021

Roberto S. Bigonha: PhD em Ciência da Computação pela Universidade da Califórnia, Los Angeles. Professor Emérito da Universidade Federal Minas Gerais. Membro da Sociedade Brasileira de Computação. Áreas de interesse: Linguagens de Programação, Programação Modular, Estruturas de Dados, Compiladores, Semântica Formal.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

B594	Bigonha, Roberto S. PROGRAMAÇÃO MODULAR: Programação Orientada por Objetos – Java / Roberto da Silva Bigonha — Belo Horizonte, MG, 2021 Bibliografia. ISBN 978-65-00-22238-8 1. Modularidade 2. Linguagem Java I. Título. CDD: 005.1 CDU: 004.41

Índice para catálogo sistemático:
1. Linguagens de Programação : Engenharia de Software

Sumário

Prefácio **vii**

Agradecimentos **viii**

1 Qualidade de Sofware **1**
1.1 Fatores externos de qualidade 2
1.2 Fatores internos de qualidade 6
1.3 Custo da manutenção 7
Conclusão 9

2 Tipos Abstratos de Dados **11**
2.1 Abstração 12
2.2 Especificação de tipo abstrato 17
2.3 Especificação algébrica 20
2.4 Implementação 21
2.5 Tipos parametrizados 25
Conclusão 30

3 Processo de Desenvolvimento **33**
3.1 Critérios de desenvolvimento 34
3.2 Desenvolvimento *top-down* 36
3.3 Desenvolvimento *bottom-up* 40
3.4 Programação orientada por objetos 40
Conclusão 42

4 Modularidade **45**
4.1 Unidade linguística 47
4.2 Baixa conectividade 48
4.3 Interface pequena 50
4.4 Interface explícita 50
4.5 Ocultação de informação 54
4.6 Unidade focal 55
4.7 Boas práticas 55
4.8 Bibliotecas 57
Conclusão 59

5 Estruturação de Módulos **61**
5.1 Módulos-função 62
5.2 Módulos-abstração 64
5.3 Módulos-tipo 64
5.4 Módulos-paramétricos 65
5.5 Módulos-mistifório 66
5.6 Coesão interna de métodos 67
5.7 Coesão interna de classes 70
5.8 Coesão interna de módulos 71
5.9 Acoplamento de módulos 73
Conclusão 80

6 Camadas de Sofware **83**
6.1 Modelo de quatro camadas 85
6.2 Organização de camadas 89
Conclusão 90

7 Programação por Contrato **93**
7.1 Direitos e obrigações 95
7.2 Exceção disciplinada 96

7.3 Subcontratação 98
Conclusão 99

8 Reúso de Componentes 101
8.1 Reúso de componente aberto 102
8.2 Reúso de componente original 103
8.3 Reúso de componente adaptável 105
8.4 Adaptação de dados e algoritmos 106
Conclusão 109

9 Princípios de Projeto Modular 111
9.1 Encapsulação de constantes 112
9.2 Respeito à natureza das classes 115
9.3 Programação para interface 126
9.4 Inversão da dependência 127
9.5 Resiliência 129
9.6 Substituição de Liskov 135
9.7 Segregação de interface 141
9.8 Responsabilidade única 145
Conclusão 147

10 Padrões de Projeto 149
10.1 Singleton 151
10.2 Factory Method 153
10.3 Abstract Factory 157
10.4 Prototype 160
10.5 Builder 162
10.6 Proxy 164
10.7 Adapter 167
10.8 Bridge 172
10.9 Composite 174

10.10 Decorator 177
10.11 Façade 180
10.12 Flyweight 184
10.13 Memento 187
10.14 State 190
10.15 Chain-of-Responsability 192
10.16 Command 195
10.17 Observer 198
10.18 Strategy 201
10.19 Template Method 204
10.20 Iterator 206
10.21 Mediator 208
10.22 Interpreter 212
10.23 Visitor 219
Conclusão 224

11 Estilo de Programação **227**
11.1 Efeito colateral em funções 227
11.2 Objetos como máquinas de estado 232
11.3 Escolha de abstrações 245
11.4 Acesso ao estado concreto 254
11.5 Nomeação de classes, objetos e métodos 255
Conclusão 256

12 Considerações Finais **257**

Bibliografia **260**

Índice Remissivo **268**

Prefácio

A série de livros Programação Modular destina-se a alunos de cursos de graduação da área de Computação, como ciência da computação, análise de sistemas, matemática computacional, engenharia de computação e sistemas de informação. As técnicas apresentadas são voltadas para o desenvolvimento de programas de grande porte e complexos.

Neste volume, Programação Orientada por Objetos com Java, é apresentado o contexto no qual os conceitos e técnicas aqui discutidos estão inseridos, a motivação e recursos necessários para o desenvolvimento de software modular, o impacto da modularidade no custo de manutenção de programas e em outros fatores de qualidade de software. Discute-se também o papel das metodologias de desenvolvimento de software, suas vantagens e desvantagens. Os conceitos e técnicas da programação modular são apresentados, discutidos e avaliados. Particular atenção é dada a estilo e padrões de programação. Práticas condenadas são expostas, e princípios e critérios de projeto de programas modulares de boa qualidade são enunciados e fundamentados. Conclui-se este volume com a apresentação de um conjunto de mandamentos da programação de boa qualidade, definidos segundo os princípios que balizaram este trabalho.

Adotou-se a linguagem de programação Java como meio de comunicação e de implementação dos exemplos apresentados para dar clareza aos conceitos e ideias aqui defendidos. A escolha de Java certamente é arbitrária, outras linguagens com recursos para programação modular existem e poderiam ter sido usadas.

Agradecimentos

Agradeço aos diversos colegas que fizeram a leitura dos primeiros manuscritos, apontaram erros, indicaram as correções e também contribuíram com exemplos. Particularmente, agradeço a Mariza Andrade da Silva Bigonha o trabalho de revisão do texto.

Roberto S. Bigonha

Capítulo 1

Qualidade de Sofware

Desenvolvimento de software não é mais uma arte, transformou-se em uma atividade de engenharia. Na arte, a criação resulta do talento do artista, sem a necessidade de se observar qualquer método científico ou sistemático. Já o engenheiro, que embora também tenha a mesma oportunidade de criação do artista, deve pautar-se pelo embasamento científico, aplicar métodos testados e produzir resultados economicamente viáveis.

A grande diferença entre a arte de construção de software e a engenharia de software é que esta emprega de forma sistemática metodologias, processos, ferramentas e métodos. Uma semelhança detectável seria o compromisso de ambas com a elegância das soluções, boa qualidade dos resultados e a beleza da obra.

Não é surpresa que um dos principais objetivos da Engenharia de Software seja a produção de software de boa qualidade. Embora o conceito de qualidade de software dependa do interesse do observador, existe um conjunto de fatores de qualidade reconhecidos por importantes autores. Alguns desses fatores focalizam-se na qualidade da programação dentro de um módulo. São fatores ligados à denominada Programaçao em Ponto Pequeno. Outros referem-se à chamada Programação em Ponto Grande, que trata da comunicação entre módulos, do controle de visibilidade e encapsulação, e dos processos de se organizar módulos para construir

grandes sistemas de software.

Os fatores de qualidade podem ser classificados em *Fatores Externos* ou *Fatores Internos*. Os fatores externos são em geral do interesse direto dos usuários do software, que podem ser o gerente de uma equipe de desenvolvimento, um especialista em desenvolvimento de software ou o usuário final. Os fatores internos de qualidade, por outro lado, estão associados à organização interna dos programas. Estes fatores são normalmente de interesse exclusivo dos especialistas em Engenharia de Software, que têm acesso ao texto fonte dos programas.

Do ponto de vista do usuário, são os fatores externos que interessam, mas os internos têm impacto direto nos fatores externos. É sobre os fatores internos é que se deve atuar para se atingir os fatores externos e, assim, melhorar a qualidade final do produto.

1.1 Fatores externos de qualidade

Os principais fatores externos de qualidade são Correção, Robustez, Extensibilidade, Reusabilidade, Eficiência, Compatibilidade, Facilidade de Uso, Portabilidade, Integridade e Verificabilidade. Outros fatores podem ser identificados, mas a lista acima tem grande aceitação entre autores importantes, e certamente fatores como correção, robustez, extensibilidade, reusabilidade e eficiência têm importância destacada.

Correção

Correção é a propriedade de um produto de software executar exatamente sua função, conforme definida pelos requisitos e especificação. Correção garante o funcionamento no que foi previsto e é esperado por quem contratou o desenvolvimento do produto.

Correção é um fator de primordial importância, embora geralmente muito difícil ser demonstrado.

Correção somente pode ser verificada ou exigida se houver uma especificação do que se quer. A especificação dos requisitos de um sistema, expressa por meio de uma notação formal, é o ideal para se garantir precisão na comunicação das ideias e regras. É fato que o desenvolvimento de sofware baseado em métodos formais de especificação é bastante raro no mercado, talvez devido às dificuldades adicionais que o rigor matemático pode impor no processo de desenvolvimento e também pela falta de tradição do mercado em exigir tamanho nível de qualidade. Entretanto, especificação é fundamental. Seja ela formal ou não. Sem especificação é impossível garantir a correção do software desenvolvido, ou melhor, sem especificação de requisitos todo software pode ser considerado correto!

Robustez

Um software de boa qualidade não deve apenas ser correto, mas deve funcionar em situações não previstas na sua especificação. Certamente todas as ações de um software devem ter sido previstas em sua implementação.

O programador cuidadoso verifica sempre as condições de funcionamento do programa em pontos importantes de forma a garantir o seu bom comportamento. Nas situações de erro ou de condições não previstas, pode-se programar ações alternativas ou simplesmente encerrar a execução. Sempre que possível uma solução alternativa e consistente com a especificação dá ao programa robustez, melhorando sua qualidade.

Robustez, portanto, é a propriedade de um software funcionar mesmo em condições não definidas em sua especificação de requisi-

tos. Robustez garante funcionamento em certos aspectos que não foram previstos, complementando sua correção.

Uma característica comum a software robusto é sua capacidade de degradação suave, isto é, o provimento de um comportamento aceitável mesmo em situações não previstas em sua especificação. Software tolerante a falhas são exemplos de softwares robustos.

Eficiência

Eficiência é o bom uso dos recursos de hardware, tais como processador, memória, dispositivos de comunicação. No embate correção versus eficiência, que pode surgir no decorrer de um projeto, correção deve merecer prioridade, haja vista que software eficiente e incorreto tem seu uso seriamente comprometido, porque frequentemente o desejo não é produzir erros em alta velocidade.

Extensibilidade

Extensibilidade é uma medida da facilidade com que o software pode ser adaptado para atender a mudanças na sua especificação. Extensibilidade é propriedade essencial em Programação em Ponto Grande. Está relacionado com a qualidade da organização dos módulos de um programa. Esse fator de qualidade tem impacto direto no custo de manutenção do software e deve ser perseguido com vigor pelo engenheiro de software.

Princípios que facilitam extensibilidade são simplicidade do projeto e descentralização, via construção de módulos autônomos. A simplicidade facilita o processo de adaptação, e a descentralização tende a confinar as alterações em poucos módulos.

Reusabilidade

Reusabilidade é a propriedade de um software ser usável em novas aplicações. A ideia de reusabilidade deve ser uma preocupação inerente ao processo de desenvolvimento, porque introduzir reusabilidade *a posteriori* pode ser um processo caro e inseguro.

Ressalte-se que reusabilidade é chave para a construção de sistemas mais confiáveis e de menor custo. O aumento de confiabilidade decorre do fato de software reusado já ter, em geral, sido exaustivamente testado em seus usos anteriores, e o menor de custo vem da redução do esforço de desenvolvimento.

Compatibilidade

Compatibilidade é uma medida da facilidade com que um software pode ser combinado com outros. Compatibilidade implica em projeto homogêneo e padronização. Exemplos bem sucedidos são a manipulação de arquivos UNIX, que aceita o mecanismo de linha de montagem (*pipe*), o uso de estrutura de dados única como as expressões simbólicas do Lisp e a interface padronizada do Smalltalk.

Facilidade de uso

Facilidade de uso inclui preparação de dados de entrada, interpretação de resultados, recuperação de erros de uso e interface amigável.

Portabilidade

Portabilidade é uma medida da facilidade de transporte de um software para diferentes ambientes de programação. Em geral, o transporte de um software de um ambiente para outro demanda

reprogramação. A medida de esforço desse transporte está ligada ao grau de portabilidade de software.

Integridade

Integridade é a capacidade de um software de proteger seus componentes contra acesso ou modificação não autorizada, de forma que se possa confiar no produto.

Verificabilidade

É uma medida da facilidade de se preparar procedimentos de aceitação, dados para testes, ou provar propriedades de seu funcionamento durante a fase de *verificação* e de *validação*. Verificação visa assegurar que cada fase do projeto do software esteja implementando o software corretamente, enquanto que validação trata de garantir que o produto correto esteja sendo implementado.

Os procedimentos de teste consistem em executar o programa com diferentes entradas de dados para exercitar seus caminhos internos e verificar o seu comportamento e procurando revelar erros. Embora testes não assegurem ausências de erros, sua realização é indispensável ao suporte dos processos de verificação e validação.

1.2 Fatores internos de qualidade

Os fatores internos de qualidade são Modularidade, Legibilidade e Manutenibilidade.

Modularidade

Modularidade é a arma para domar a complexidade de grandes sistemas. Sem os recursos do particionamento de um sistema em

módulos é praticamente impossível garantir sua correção e extensibilidade. Mecanismos para encapsulação e de controle estrito de visibilidade dos constituintes de programas são essenciais para manter o projeto e a programação sob controle. Manutenção de um software não-modular pode ser muito cara e até mesmo inviável. O foco principal deste livro é o estudo das principais técnicas e mecanismos de modularização de grandes sistemas de software.

Legibilidade

Legibilidade é um fator essencial para garantir a extensibilidade de um software. Para realizar qualquer alteração em um software, é preciso entendê-lo de forma a avaliar e acomodar o impacto das modificações. Até mesmo durante o desenvolvimento de um programa de pequeno porte, legibilidade é importante, porque uma vez escrita a primeira versão de um programa, ele deverá ser lido diversas vezes pelo seu programador antes de declará-lo finalizado.

Manutenibilidade

Manutenibilidade trata da capacidade de um software ser objeto de manutenção, que é o termo atribuído às mudanças feitas em um sistema depois que ele foi colocado em operação. Todo software está sujeito a processo de manutenção, que pode ser necessária por diversos motivos.

1.3 Custo da manutenção

Define-se manutenção como sendo a atividade de modificação de um software para atender mudanças no mundo externo ou para remover erros, que não deviam estar lá.

No fim da década de setenta, Lintz e Swanson [30] fizeram um levantamento em 487 instalações de software de diversos tipos, e.g., centros científicos, centros de processamento de dados, centros de informática do Departamento de Defesa Americano. Esse levantamento, considerado muito representativo, mediu onde estava o custo no desenvolvimento de software e revelou a seguinte distribuição dos custos de manutenção:

- mudanças na especificação: 41.8%
- mudanças no formato dos dados: 17.4%
- consertos de emergências: 12.4%
- depuração: 9.0%
- mudanças no hardware: 6.2%
- atualização da documentação: 5.5%
- melhoria na eficiência: 4%
- outras: 3.4%

Observe que, de acordo com a distribuição apresentada, 41,8% do custo referem-se a mudanças na especificação, 17,4%, no formato dos dados. Esses dois pontos podem ser vistos como mudanças na especificação, porque o formato dos dados, de alguma forma, pode fazer parte da especificação. Assim, pode-se observar que cerca de 60% do custo de manutenção são devidos a mudanças na especificação, que são situações impossíveis de serem previstas, pois são externas ao software.

Consertos de emergência constituem 12% do custo de manutenção. Esse tipo de manutenção em geral carece dos cuidados necessários, e acabam introduzindo outros erros, encarecendo todo o processo.

O custo em depuração, durante a fase de manutenção, apenas 9% do custo de manutenção, é baixo provavelmente porque somente faz-se depuração de programa nessa fase se erros forem detectados.

Os riscos de introduzir erros no processo de depuração são altos, e normalmente deseja-se evitá-los.

O levantamento de Lintz e Swanson sugere que durante a fase de manutenção investe-se muito pouco na atualização da documentação. Isto provavelmente decorre da falta de integração da documentação ao software implementado. O ideal seria poder atualizar um sistema tanto pelo lado do programa implementado, como pela sua especificação, e que o impacto das alterações fosse devidamente incorporado onde pertinente.

Os resultados obtidos por Lintz e Swanson demonstram que, naquela época, cerca de 70% do custo de um software é devido à manutenção. Isto não é um resultado muito surpreendente porque o tempo de manutenção normalmente é muito maior que o tempo de desenvolvimento do software, haja vista que, em geral, o tempo de uso de um software é sempre muito maior que o tempo de seu desenvolvimento. Assim, todo investimento feito para reduzir custos de manutenção tem impacto direto na redução do custo do produto.

Conclusão

O desenvolvimento de sistemas de grande porte necessita de técnicas de organização de software que privilegiem extensibilidade e reusabilidade e que suportem o desenvolvimento sistemático de software de forma a garantir correção e robustez.

São os fatores externos que interessam a todos, mas só podem ser atingidos por meio dos fatores internos, como Modularidade, Legibilidade e Manutenibilidade. Os recursos mais modernos para se atingir alto grau de modularidade e baixo custo de manutenção são programação com tipos abstratos de dados, programação ori-

entada por objetos e programação orientada por aspectos.

Exercícios

1. Avalie o impacto (positivo ou negativo) de cada um dos fatores externos de qualidade no custo de manutenção de software.
2. Mostre como fatores internos de qualidade podem contribuir para melhorar os fatores externos de qualidade de *correção* e *extensibilidade*.
3. Qual deve ser a principal preocupação de programador para produzir software com elevado grau de reusabilidade?
4. Cite um fator externo de qualidade de software em ambiente com alto grau de paralelismo.

Notas bibliográficas

Qualidade de software tem sido objeto de preocupação de vários autores desde os primórdios da Computação. Um clássico digno de menção é o trabalho de C.A.R. Hoare de 1972 [25].

Bertrand Meyer [41] fez uma excelente conexão entre fatores de qualidade de software e os recursos oferecidos pela Programação Orientada por Objetos. Uma boa parte das ideias aqui defendidas foi inspirada no trabalho de B. Meyer.

O conceito de programação em ponto grande e de programação em ponto pequeno foi estabelecido por Franklin DeRemer em 1975 [13].

Capítulo 2

Tipos Abstratos de Dados

Uma fase do projeto de um sistema de software compreende a atividade de identificar as entidades do mundo real da aplicação e definir seu mapeamento em elementos que vão compor o programa.

O projeto, portanto, é um exercício intelectual que tem o objetivo de transpor a **lacuna semântica**, ilustrada na Fig. 2.1, existente entre os objetos do mundo real, identificados pela fase de análise, e os objetos a ser implementados no computador por meio da linguagem de programação escolhida. O resultado desse exercício são a definição das estruturas de dados envolvidas e a formulação das soluções algorítmicas para suas operações.

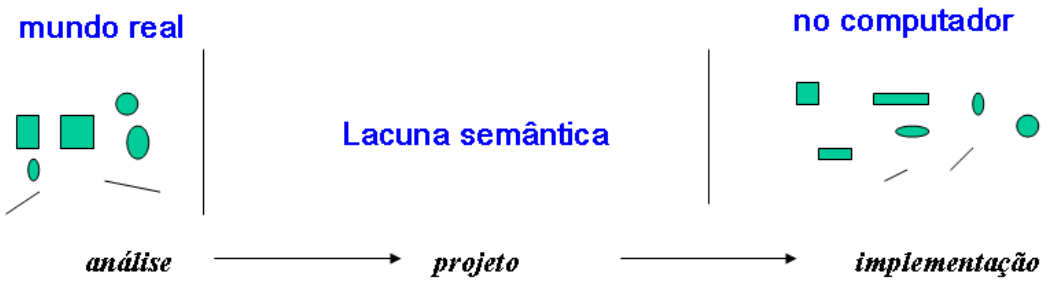

Figura 2.1 Lacuna Semântica

A dificuldade dessa tarefa é diretamente proporcional à largura da lacuna semântica e inversamente proporcional às facilidades de expressão disponibilizadas pela linguagem de programação.

2.1 Abstração

Uma abstração é um ato de destacar as características de um objeto que o distinguem de outros tipos de objetos e que proveem, segundo o ponto de vista do observador, limites conceituais claros e bem definidos. Ela nasce da percepção de similaridades entre objetos e da decisão de se focar a atenção nessas similaridades e de ignorar as diferenças.

Uma abstração dá a visão externa do ente observado e separa seu comportamento de sua implementação, sendo assim um recurso muito poderoso para se dominar a complexidade de grandes sistemas. Abstração é um modo de pensar no qual o projetista concentra-se na ideia e não nas suas manifestações específicas. Abstrair-se significa concentrar-se no que o programa faz e não em como ele o faz e, por isso, é o elemento básico de construção de módulos de boa qualidade.

O resultado da abstração certamente depende dos objetivos de quem se abstrai. Por exemplo, a abstração da entidade *pessoa* depende de seu uso no sistema: uma *pessoa* sob a perspectiva de um médico é um ente que possui cérebro, coração, pressão sanguínea, pulmão, etc, enquanto que, do ponto de vista de um empregador, a mesma pessoa é vista com uma entidade dotada de atributos como nome, cpf, estado civil, endereço, dependentes, escolaridade e salário.

Uma vez identificada, a abstração deve ser implementada como um módulo de forma a poder ser usada onde for necessária no programa. Para isso, suas propriedades como um objeto do mundo real devem ser mapeadas em elementos do programa, transpondo a mencionada lacuna semântica.

Um dos segredos para enfrentar com sucesso a complexidade do esforço para reduzir a lacuna semântica em cada caso é a criação

de objetos de computação que mais se aproximem dos objetos do mundo real encontrados pela análise e a existência de meios para manipulá-los adequadamente. Quando a correspondência entre esses dois mundos, o mundo da aplicação e o da computação, for *um-para-um*, a construção dos objetos de computação pode tornar-se bastante simples.

Historicamente, a linguagem de programação PL/I representa uma tentativa, no início da década de 60, de reunir Fortran, Cobol e Algol 60 em uma única notação com o propósito de se ter uma linguagem com mecanismos para realizar todas as abstrações de estruturas de controle de fluxo desejadas e oferecer os principais tipos de dados que fossem necessários a qualquer aplicação.

A linguagem de simulação Simscript, também criada na década de 60, é rica em construções voltadas para a expressão de modelos de simulação e tratamento de eventos. Nessa linguagem, pode-se usar comandos como *insira a entidade x na fila q* ou então *calcule o tamanho médio da fila q*, o que torna a expressão dos comandos de simulação de processos bastante direta.

Essas linguagens tiveram seu período de sucesso na primeira metade da década de 60, mas seus usos revelaram rapidamente a inviabilidade de se incorporar em uma única linguagem de programação todos os recursos desejáveis, e que uma melhor solução seria trabalhar com linguagens extensíveis, haja vista a dificuldade de se prever todas as necessidades de abstração.

Inicialmente, considerou-se mais importante a extensibilidade de estruturas de controle de fluxo de execução. As linguagens deveriam oferecer recursos para o programador criar os comandos mais apropriados a cada aplicação. Nesse sentido, foram inventados pré-processadores de linguagens, com recursos para estender as estruturas de controle de fluxo de execução da linguagem base.

Em geral, os pré-processadores permitiam a definição de novas estruturas de controle por meio de definição de *macros*, e, numa operação de pré-processamento do programa-fonte, as macros usadas seriam expandidas, segundo as definições contidas no próprio texto do programa-fonte, para transformá-las em textos da linguagem base pura, que seriam então submetidos à compilação.

O resultado dessa corrente de pensamento, do ponto de vista de Engenharia de Software, foi desastroso: programas ficaram ainda mais ilegíveis, porque cada programador poderia usar estruturas de controle particulares, por ele definidas no próprio código, e, geralmente, somente por ele conhecidas e sem qualquer documentação, tornando muito difícil o entendimento do programa por terceiros.

A experiência permitiu, antes do fim da década de 60, a compreensão de que é o conjunto de tipos de dados de uma linguagem de propósito geral que deve ser extensível e não sua estrutura de controle de fluxo. Chegou-se ao entendimento que abstração de dados tem um papel muito mais importante no projeto de software que abstrações de estrutura de controle, as quais devem ser as mais simples. Para reduzir a chamada lacuna semântica, extensibilidade de dados é mais efetiva que a de estrutura de controle.

Essa nova visão levou à invenção do conceito de *tipos abstratos de dados*, que valoriza a criação de tipos apropriados para cada aplicação, como exemplificado a seguir

Para ilustar a importância de se ter os módulos ou abstrações apropriados para realizar uma dada implementação, considere o seguinte trecho de programa, escrito em uma linguagem muito limitada, que possui um único tipo de dados, o tipo básico `boolean`, com as operações & (conjunção), | (disjunção) e `not` (negação), e que se propõe a ler os 16 bits de dois inteiros, somá-los e imprimir o resultado.

```
1  programa LacunaGrande;
2  var x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,
3      y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,
4      t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,
5      c : boolean;
6  begin
7    c := false;
8    read(x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,
9                                                    x14,x15);
10   read(y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,
11                                                   y14,y15);
12
13   t15 := (not x15)&(not y15)& c  | (not x15)&(y15)&(not c)
14          | x15&(not y15)&(not c) | (x15)&(y15)&c;
15   c   := (x15 & y15) | (x15 & c) | (y15 & c);
16   t14 := (not x14)&(not y14)&c | (not x14)&(y14)&(not c)
17          | x14&(not y14)&(not c) | (x14)&(y14)& c;
18   c   := (x14 & y14) | (x14 & c) | (y14 & c);
19   t13 := (not x13)&(not y13)& c  | (not x13)&(y13)&(not c)
20          | x13&(not y13)&(not c) | (x13)&(y13)& c;
21   c   := (x13 & y13) | (x13 & c) | (y13 & c);
22   comandos para calcular t12, t11, t10, ..., t3 e t2
23   t1  := (not x1)&(not y1)& c    | not x1)&y1&(not c)
24          | x1& (not y1)& (not c) | x1 & (y1) &  c;
25   c   := (x1 & y1) | (x1 & c)    |  (y1 & c);
26   t0  := (not x0)&(not y0)& (c)  | (not x0)& y0&(not c)
27          | x0&(not y0)& (not c)  | x0&y0& c;
28
29   println(t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,
30                                                  t14,t15);
31 end.
```

Compare o código acima com o do seguinte programa, escrito na linguagem Pascal, dotada do tipo `integer`, sendo capaz de operar com aritmética de inteiros de 16-bits, e que resolve o mesmo problema de ler e somar dois valores inteiros e imprimir o resultado.

```
1 programa LacunaPequena;
2 var x, y, t : integer;
3 begin
4    read(x);
5    read(y);
6    t := x + y;
7    write(t)
8 end.
```

A falta do tipo de dados apropriado tornou a programação de `LacunaGrande` mais difícil e prejudicou enormemente a legibilidade do código produzido. E com o uso do tipo de dado adequado, é muito fácil deduzir a semântica do programa `LacunaPequena` por simples inspeção do código implementado.

Sempre que os tipos de dados necessários estiverem disponíveis, a expressão do programa fica simplificada. Como não é razoável uma linguagem de programação prover todos os tipos de dados necessários a qualquer aplicação, principalmente por que a lista de tipos pode ser incontrolavelmente grande, a melhor solução é prover um conjunto básico de tipos de dados acompanhado de mecanismos que permitam ao programador criar os novos tipos que sejam necessários em cada aplicação.

O mecanismo usado para criar novos tipos de dados denomina-se *Tipo Abstrato de Dados* (TAD), que é uma das peças fundamentais para reduzir lacunas semânticas pelo provimento das abstrações adequadas a cada caso. Um TAD é um componente de programa que implementa um novo tipo de dados, encapsulando a estrutura de dados usada para representar os estados das variáveis do tipo sendo definido, e a torna conhecida fora do componente apenas pelas operações que se podem realizar sobre os seus elementos, sem que se permita acesso direto à sua representação nem se indique externamente como essas operações foram codificadas.

O termo *abstrato* enfatiza que essas estruturas de dados não podem ser acessíveis ao usuário do tipo, i.e., as estruturas de dados que formam a representação de um TAD não fazem parte da sua visão externa. E encapsulação é o termo usado para descrever o ato de reunir em um mesmo local diversos elementos do programa e exercer sobre eles controle de acesso e de visibilidade.

2.2 Especificação de tipo abstrato

Um tipo abstrato de dados compreende a declaração de uma ou mais estruturas de dados e operações que contribuem para implementá-lo, e deve ser caracterizado por um único e relevante contrato público, definido por uma interface clara e explícita.

Em linguagens orientadas por objetos a estrutura mais importante para implementar um novo tipo é a classe, que pode usar ou encapsular outras classes, mas deve implementar apenas um tipo. As demais estruturas nela contidas devem ser de uso apenas local ou parte de sua interface.

Os métodos de um tipo abstrato de dados devem operar sobre seus campos de forma relevante e significativa, e somente os métodos do seu contrato podem ser públicos, sendo todos os demais métodos e os campos declarados privados. E não deve haver nem mesmo visibilidade `protected` de campos e métodos para não liberar acesso a partir de subclasses.

Tipos abstratos de dados são um recurso eficiente para reduzir a chamada *lacuna semântica*, que é a distância entre os conceitos que seres humanos conhecem e os conceitos que os computadores manipulam. Quanto maior for essa lacuna, mais difícil é a fase de projeto e implementação de um programa. Quanto mais rica em recursos para expressar as necessidades de uma aplicação for a

linguagem de programação usada, mais facilmente realizam-se as abstrações necessárias. Simula 67, Modula 2, C++, Eiffel, Java e Phyton são exemplos de linguagens projetadas para reduzir essa lacuna semântica.

Uma vez identificados os objetos da aplicação, deve-se descrever o comportamento de cada um, o qual consiste em caracterizar suas operações, isto é, determinar o seu tipo. Detalhes da representação dos objetos são de interesse exclusivo de sua implementação e, por isto, não devem constar de sua especificação, cuja função principal é destacar e definir sua interface.

Como cada conjunto de operações define um novo tipo de dados, tipos abstratos de dados apresentam-se com um mecanismo apropriado para descrever o comportamento dos objetos.

A descrição de um tipo abstrato de dados compreende a definição de sua interface, declaração de restrições e exigências impostas no uso de cada uma das operações e a especificação de seu funcionamento. A descrição deve ser completa, precisa, não-ambígua, independer de representação física e preferencialmente formal.

Uma descrição informal, baseada em linguagem natural, de um tipo abstrato de dados é muitas vezes aceitável, desde que seja completa, rigorosa e precisa. Entretanto, essas propriedades são difíceis de ser obtidas por meio de linguagens naturais, que são inerentemente imprecisas, pois, no mínimo, permitem construções com mais de um significado.

Um exemplo de ambiguidade inerente a linguagens naturais é oferecido pela frase "*Navegar é preciso; viver não é preciso*", extraída da poesia *Navegar é Preciso* de Fernando Pessoa. O sentido dessa frase depende da interpretação de *exatidão* ou de *necessidade* que se poderia dar a palavra *preciso*. A ambiguidade só é resolvida pelo contexto, que pode ser descrito por meio de

palavras adicionais, ou pela circunstância em que própria frase foi dita pela primeira vez em uma outra língua[1].

A necessidade de prover contexto para resolver ambiguidades frequentemente torna a especificação em linguagem natural verbosa e demasiadamente longa, dificultando seu uso como base de raciocínio dedutivo, em contraste com notações matemáticas, que geralmente são precisas e compactas.

Uma notação para especificação de tipos abstratos de dados é usar a própria linguagem de programação, que pode ser vista com uma linguagem formal, porque um programa nela escrito tem um único e preciso significado, definido pelo compilador da linguagem e pelo computador usado para executar o programa. Execuções repetidas do programa para diversas entradas exibem seu significado.

Entretanto, linguagens de programação não são o tipo de notação formal mais apropriado para especificação de tipos abstratos de dados, porque elas são destinadas a descrever a representação de objetos e como eles são manipulados, e não para descrever uma abstração de objetos.

O que se precisa é uma notação que encoraje o uso de abstrações para descrever as ideias em vez de detalhar sua representação. Além disso, o raciocínio dedutivo a partir do código de um programa também é difícil de ser conduzido, pois os detalhes da representação e a presença de *loop*s que dificultam o processo.

Conclui-se que uma notação mais rigorosa e de fácil uso é indispensável para garantir a produção de software de boa qualidade. Uma notação recomendada para modelar os objetos que um sistema manipula é fornecida pela teoria de tipos abstratos de dados.

[1] A frase original "*Navigare necesse; vivere non est necesse*"é de autoria de Pompeu, general romano, 106-48 AC, dirigida aos marinheiros que se recusavam viajar durante a guerra.

2.3 Especificação algébrica

Há muitas formas de se especificar um tipo abstrato de dados. A definição algébrica `PilhaDeInt` do tipo de pilhas de objetos do tipo `Integer` apresentada abaixo é uma delas:

```
1  type PilhaDeInt
2  sorts:
3     Integer, Boolean
4  operations:
5    vazia      : PilhaDeInt --> Boolean
6    crie       : Integer --> PilhaDeInt
7    empilhe    : (Integer,PilhaDeInt) --> PilhaDeInt
8    desempilhe : PilhaDeInt --> PilhaDeInt
9    topo       : PilhaDeInt --> Integer
10 preconditions:
11   pre crie(n : Integer)           : ( n > 0)
12   pre desempilhe(s : PilhaDeInt) : (not vazia(s))
13   pre topo(s : PilhaDeInt)        : (not vazia(s))
14 axioms:
15   for all n : Integer, x : Integer, s : PilhaDeInt:
16      vazia(crie(n)) and (n > 0)
17      not vazia(empilhe(x,s))
18      topo(empilhe(x,s)) = x
19      desempilhe(empilhe(x,s)) = s
20 end
```

A especificação algébrica de um TAD consiste em quatro partes: relação dos tipos (*sorts*) usados na especificação, a definição das assinaturas das operações do TAD, as pré-condições exigidas pelas operações em relação a seus parâmetros e a definição, por meio de axiomas, da semântica das operações, que, no exemplo acima, define que `crie` aloca uma pilha vazia, somente pode-se desempilhar uma pilha não-vazia, o último elemento empilhado passa a ser o topo da pilha e que o elemento desempilhado é sempre o do topo.

Note que uma boa especificação nunca se refere a qualquer elemento de implementação, sendo a mais abstrata possível. O ideal é sempre prover uma especificação usando uma notação formal, baseada em matemática, mas se isto não for possível, uma especificação em linguagem natural ainda é preferível a não se ter especificação alguma. De qualquer forma, mesmo sem usar notação matemática, é importante que as quatro partes de descrição citadas acima sejam desenvolvidas e que se busque rigor e precisão, evitando ambiguidades e inconsistências.

2.4 Implementação

A assinatura de um TAD é formada pelo conjunto das assinaturas de suas operações, as quais devem informar os seus nomes, tipos dos valores retornados e nomes e tipos de seus parâmetros. Muitas linguagens de programação oferecem recursos para especificar a assinatura de tipos abstratos de dados.

Em Java, o mecanismo de interfaces permite especificar a visão externa de um TAD apresentando as assinaturas de suas operações, sem impor qualquer compromisso sobre como poderá o TAD ser implementado por uma ou mais classes. Por exemplo, pode-se definir a interface do TAD `PilhaDeInt` da seguinte forma:

```
1 public interface PilhaDeInt {
2     boolean vazia();
3     void empilha(int x) throws FaltaEspaço;
4     int desempilha() throws FaltaElemento;
5 }
6 public class FaltaEspaço extends Exception {}
7 public class FaltaElemento extends Exception {}
```

onde as classes `FaltaEspaço` e `FaltaElemento` são usadas para sinalizar as condições anormais das operações de empilhar e desem-

pilhar, respectivamente. Essas classes ajudam na implementação das pré-condições de uso e exigências impostas sobre as operações, que, nesse exemplo, estão listadas na cláusula **preconditions** da especificação algébrica apresentada na Seção 2.3.

Note, contudo, que as interfaces de Java apenas mostram a assinatura das operações. Cabe ao implementador incluir mais informações, por exemplo, na forma de comentários, para aproximar a declaração de uma interface Java da especificação algébrica da interface do tipo abstrato de dados que ela pretende representar.

Definidas as interfaces, os tipos abstratos devem ser implementados. Classes de Java são um mecanismo para implementá-los, pois permitem encapsular os detalhes de sua implementação, particularmente os relativos à sua representação.

Encapsulação em Java é o agrupamento de estruturas de dados e funções dentro de uma classe e o respectivo controle de acesso aos membros da classe. A estrutura de dados deve ser formada por campos privados e somente as operações, também chamadas de métodos, devem ser públicas. A parte pública de um tipo abstrato de dados é a definida em sua interface, que deve independer da sua representação, que é dada pelos campos privados.

Mais de uma classe pode implementar uma mesma interface. Isto ocorre quando se deseja prover mais de uma representação para um tipo de dados. Cada classe define a estrutura dos objetos a ser instanciados e o código de suas operações. Essa representação dos dados deve obrigatoriamente estar encapsulada.

A escolha da estrutura de dados da representação tem impacto na implementação das operações do tipo ou em seu desempenho, mas não deve afetar a interface do tipo implementado. Se isto for obedecido, diversas implementações de uma mesma interface podem conviver harmonicamente em um mesmo programa, onde

os detalhes de cada implementação podem ser abstraídos.

O mecanismo existente em praticamente todas linguagens de programação para criar estruturas de dados mais elaboradas é o de composição de estruturas menores, formando, recursivamente, coleções de dados homogêneos ou tuplas de elementos heterôgeneos.

Coleções de dados homogêneos são implementadas via arranjos, e tuplas de elementos heterogêneos, que são conhecidas como *fichas de dados*, são definidas pela reunião de grupos de entidades relacionadas, mas possivelmente de diferentes tipos.

Em linguagens tradicionais, o recurso linguístico para descrever composição de estruturas de diferentes tipos é denominado *record* ou *struct*. Nas linguagens mais modernas, fichas de dados são implementadas por meio de classes, como a do seguinte exemplo:

```
1 public class Aluno {
2     public String nome;
3     public String matrícula;
4     public int nota;
5     public int frequência;
6 }
```

Esse tipo de classe deve possuir somente declarações de campos públicos e nenhum método, exceto construtoras, se necessárias, e serve para compor a formação de estruturas de dados mais elaboradas. Seus campos são acessados individualmente para leitura ou escrita, podendo-se alternativamente ser declarados com visibilidade `private` e acessados via métodos `set` e `get`.

As classes fichas-de-dados geralmente são usadas apenas como estruturas dados fortemente vinculadas ao tipo abstrato de dados do qual fazem parte, devendo ser declaradas como estáticas e aninhadas nas classes que implementam o TAD, como ilustra a participação das classes `Info` e `Nodo` na seguinte definição do tipo `Lista`:

```
1  public class Lista {
2     public static class Info {public String nome, cpf;}
3     private static class Nodo {
4        public  Info valor;
5        public  Nodo próximo;
6        public  Nodo(Info valor) {this.valor = valor;}
7     }
8     private Nodo cabeça;
9     public Lista( ) {cabeça = null; }
10    public void insira(Info valor){... new Nodo(valor);...}
11    public Info retire(){Nodo x; ... ; return x.valor; }
12 }
```

A classe Java abaixo usa uma estrutura baseada em arranjos para implementar um tipo com a interface `PilhaDeInt`, na qual somente pode-se instalar inteiros.

```
1  public class PilhaSóDeInteiros implements PilhaDeInt{
2      private int[] s ;
3      private int max, topo = -1;
4      public PilhaS(int size) {
5         max = (size < 1 ? 1 : size - 1);
6         s = new int[max + 1];
7      }
8      public boolean vazia() { topo == -1; }
9      public void empilhe(int x) throws FaltaEspaço {
10        if (topo = max) throw new FaltaEspaço();
11        s[++topo] = x;
12     }
13     public int desempilhe() throws FaltaElemento  {
14        if (topo == -1) throw new FaltaElemento();
15        return s[topo--];
16     }
17 }
```

2.5 Tipos parametrizados

Muitas vezes, o tipo abstrato de dados definido por uma classe descreve o comportamento de objetos contêineres, que são objetos que armazenam uma coleção de outros objetos de um dado tipo.

Por exemplo, o tipo `PilhaDeInt` foi implementado por uma classe que define uma estrutura de dados linear do tipo pilha capaz de armazenar elementos do tipo `int` e o código das operações usuais para empilhar ou desempilhar itens desse tipo. Na prática, entretanto, é possível que uma mesma aplicação, ou uma outra, necessite de diversas pilhas de diferentes tipos de itens em cada uma. Uma solução possível de implementação é duplicar e adaptar a classe `PilhaSóDeInteiros` para cada novo tipo de itens que vier a ser necessário.

O problema dessa abordagem é o seu alto custo de manutenção decorrente da existência de código replicado: qualquer alteração na especificacão do tipo poderá ter impacto em todas as cópias. Além disto, o número de cópias a ser atualizadas pode não ser evidente pela simples inspeção do código, demandando uma análise mais profunda do programa.

Felizmente, uma solução de reúso de melhor qualidade surge no caso em que as operações sobre tipos contêineres não têm qualquer dependência com o tipo de objetos que manipulam.

Por exemplo, em vez de se especificar o tipo `PilhaDeInt`, que somente opera com itens inteiros, é mais apropriado definir o tipo abstrato `Pilha[T]`, de pilhas de objetos do tipo `T`, sendo `T` um tipo parâmetro a ser substituído conforme a necessidade de cada aplicação.

Em Java, no lugar de se definir a interace `PilhaDeInt` específica, pode-se ter a declaração de uma interface genérica ou parametrizada `Pilha<T>` com a seguinte assinatura:

```
1 public interface Pilha<T> {
2    void empilhe(T v) throws FaltaEspaço;
3    T desempilhe() throws FaltaElemento;
4    boolean vazia();
5 }
```

onde o tipo do objeto a ser empilhado ou desempilhado deve ser definido apenas no momento de instanciação de cada pilha. E o parâmetro de `empilhe` e o objeto retornado por `desempilhe` devem ser tratados como referência a `Object`. Assim, as operações sobre objetos do tipo `T` são as definidas na classe `Object`, atribuição de referências, passagem de parâmetros, retorno de função e comparação por igual ou diferente. A especificação algébrica da interface `Pilha<T>` pode ter a seguinte forma:

```
1  type Pilha[T]
2  sorts:
3      Integer, Boolean
4  operations:
5      vazia      : Pilha[T] --> Boolean
6      crie       : Integer --> Pilha[T]
7      empilhe    : (T,Pilha[T]) --> Pilha[T]
8      desempilhe : Pilha[T] --> Pilha[T]
9      topo       : Pilha[T] --> T
10 preconditions:
11     pre crie(n : Integer)        : ( n > 0)
12     pre desempilhe(s : Pilha[T]) : (not vazia(s))
13     pre topo(s : Pilha[T])       : (not vazia(s))
14 axioms:
15     for all n : Integer, x : T, s : Pilha[T]:
16        vazia(crie(n)) and (n > 0)
17        not vazia(empilhe(x,s))
18        topo(empilhe(x,s)) = x
19        desempilhe(empilhe(x,s)) = s
20 end
```

A implementação de interfaces Java genéricas é realizada por classes genéricas que são um mecanismo destinado à definição de tipos abstratos parametrizados, com um ou mais parâmetros do tipo *tipo*. Esses tipos, que são parâmetros, são definidos no momento em que a classe genérica for instanciada, gerando novos tipos de dados. As classes que implementarem essa interface têm a obrigação de assegurar o atendimento das condições definidas na especificação algébrica e de sinalizar situações de falha de atendimento via as exceções indicadas.

A classe genérica `PilhaSequencial<T>` abaixo provê uma implementação do tipo abstrato `Pilha<T>` por meio de uma estrutura de dados arranjo, cujo tamanho é definido pelo usuário por meio de sua função construtora.

```
1  public class PilhaSequencial<T> implements Pilha<T> {
2     private int ultimo, topo = -1;
3     private T[ ] itens;
4     public PilhaSequencial(int max) {
5        if (max < 1) max = 1;
6        itens = (T[ ]) new Object[max];
7        ultimo = max - 1;
8     }
9     public void empilhe(T v) throws FaltaEspaço {
10       if (topo == ultimo) throw FaltaEspaço();
11       itens[++topo] = v;
12    }
13    public T desempilhe() throws FaltaElemento {
14       if (vazia()) throw new FaltaElemento();
15       return itens[topo--];
16    }
17    public boolean vazia() {
18       return (topo == -1);
19    }
20 }
```

Observe que operações do TAD `PilhaSequencial` foram implementadas de forma a assegurar a satisfação das pré-condições definidas, que são *pilha não vazia* para `desempilhe` e *falta de espaço disponível na pilha*, no caso de `empilhe`.

Assim, qualquer aplicação pode usar tipo `Pilha<T>` e a sua implementação fornecida pela classe `PilhaSequencial<T>` para criar uma ou mais pilhas independentes, como ilustrado no exemplo abaixo, que cria uma pilha de `Integer` e outra de `Double`, compartilhando a mesma implementação de `PilhaSequencial<T>`.

```
1  public class TesteDePilhaSequencial {
2     public static void main(String[] args) {
3        Pilha<Integer> p = new PilhaSequencial<Integer>(20);
4        Pilha<Double>  q = new PilhaSequencial<Double>(10);
5        int a, x[ ]    = { 1, 2, 3, 4, 5, 6 };
6        double b, y[ ] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
7        for(int i : x) p.empilhe(i);
8        for(double i : y) q.empilhe(i);
9        System.out.println("Topo de q = " + q.desempilhe());
10       System.out.print("Pilha p:");
11       while(!p.vazia()) {
12          a = p.desempilhe();
13          System.out.print(" " + a);
14       }
15    }
16 }
```

No programa acima, `Pilha<T>` é instanciado duas vezes para formar dois tipos de dados, `Pilha<Integer>` e `Pilha<Double>`, inteiramente novos. Esses tipos formam estruturas homogêneas, verificáveis pelo compilador, no sentido em que os objetos contidos nas pilhas devem ser somente do tipo especificado nas declarações das pilhas, não sendo permitido mistura de tipos fora da hierarquia de cada um.

Essa separação da interface de sua implementação facilita a criação de uma segunda implementação de `Pilha<T>`, por meio da classe genérica `PilhaEncadeada<T>` abaixo, que diferencia-se da implementação `PilhaSequencial<T>` acima por usar uma estrutura de dados baseada em alocação encadeada, estando, neste caso, o tamanho das pilhas limitado pela memória disponível.

```
1  public class PilhaEncadeada<T> implements Pilha<T> {
2     private static class Elemento<T> {
3        Elemento<T> prox;
4        T info;
5        Elemento(T info) {this.info = info;}
6     }
7     private Elemento<T> topo;
8     public PilhaEncadeada() { }
9     public void empilhe(T v) throws FaltaEspaço {
10       Elemento<T> novo;
11       try {novo = new Elemento<T>(v); }
12       catch(Exception e) { throw new FaltaEspaço(); }
13       novo.prox = topo; topo = novo;
14    }
15    public T desempilhe() throws FaltaElemento {
16       T info;
17       if(vazia()) throw new FaltaElemento();
18       info = topo.info;
19       topo = topo.prox;
20       return info;
21    }
22    public boolean vazia() {return (topo == null);}
23 }
```

Parece razoável que o usuário tenha consciência do tipo objeto a ser criado, `PilhaSequencial` ou `PilhaEncadeada`, mas a partir daí, não deve haver quaisquer diferenças no seu uso.

Os recursos de classes e interfaces genéricas de Java proveem uma solução com alto grau de reusabilidade ao permitir a imple-

mentação de tipos abstratos de dados parametrizados ao mesmo tempo que assegura disciplina no uso dos tipos envolvidos.

O programa abaixo, que produz o mesmo resultado que o programa `TesteDePilhaSequencial`, usa o tipo `PilhaEncadeada` para implementar a pilha de elementos do tipo `Double`, sendo essa a única diferença entre esses códigos.

```
1  public class TesteDePilhas {
2     public static void main(String[] args) {
3        Pilha<Integer> p = new PilhaSequencial<Integer>(20);
4        Pilha<Double>  q = new PilhaEncadeada<Double>(10);
5        int a, x[ ]    = { 1, 2, 3, 4, 5, 6 };
6        double b, y[ ] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
7        for(int i : x) p.empilhe(i);
8        for(double i : y) q.empilhe(i);
9        System.out.println("Topo de q = " + q.desempilhe());
10       System.out.print("Pilha p:");
11       while(!p.vazia()) {
12          a = p.desempilhe();
13          System.out.print(" " + a);
14       }
15    }
16 }
```

Conclusão

A orientação por objetos tem foco na identificação e implementação dos objetos que compõem um sistema, e, nesse contexto, tipos abstratos de dados são uma ferramenta extremamente útil e indispensável.

Tipos abstratos de dados permitem a implementação dos seguintes clássicos conceitos da Engenharia de Software:

- Abstração: foco no relevante enquanto ignoram-se detalhes de implementação.
- Modularidade: divisão de um problema em subproblemas independentes.
- Encapsulação e controle de visibilidade: *boas cercas fazem bons vizinhos*.
- Separação de interesses: conceitos relacionados devem ser agrupados em módulos, evitando seu espalhamento ao longo do código.

Exercícios

1. Descreva a abstração *caixa de banco*, de forma independente de sua implementação, seja eletrônica ou humana.
2. Avalie importância de se impedir o acesso a representação de um tipo de dados.
3. Por que não é possível implementar tipos abstratos de dados na linguagem **C**.

Notas bibliográficas

Fortran é a primeira linguagem de programação de alto nível. Sua primeira versão surgiu em 1954. Outras versões, Fortran IV, Fortran 77 e Fortran 90, que incorporam alguns avanços da Área, ficaram mais conhecidas [24, 55].

A linguagem PL/I [26] foi criada pela IBM para ser a linguagem número 1, com todos os recursos necessários a qualquer aplicação. Essa abordagem terminou sendo condenada diante da preferência por linguagens dotadas de mecanismos apropriados para ser estendidas conforme a necessidade.

O conceito de tipos abstratos dados é devido a Barbara Liskov [32], que o incorporou no projeto da linguagem CLU [33].

Capítulo 3

Processo de Desenvolvimento

O processo clássico de desenvolvimento de software compreende as fases de Análise, Projeto e Implementação.

A fase de análise consiste no levantamento e definição de requisitos e da identificação dos elementos da aplicação existentes no mundo real que devem ser modelados no sistema de computação. Normalmente essa fase demanda do Engenheiro de Sofware conhecimento do domínio da aplicação e das funções que o software deve desempenhar. Seus resultados são a documentação das decisões tomadas, a especificação dos requisitos e a definição da funcionalidade desejada e das interfaces dos principais objetos ou elementos que compõem a aplicação.

A fase de projeto trata da definição da arquitetura do software, suas estruturas de dados principais, das representações de interfaces e da concepção dos algoritmos que implementam os detalhes procedimentais necessários ao funcionamento do software.

A fase de implementação trata da codificação do programa na linguagem de programação escolhida, realização de testes e de sua validação em relação aos requisitos definidos na fase de análise.

Um método de desenvolvimento de boa qualidade deve ajudar o projeto na identificação da melhor estrutura de um sistema e também deve funcionar como uma metodologia para dirigir o raciocínio na busca da melhor forma de quebrar a complexidade iner-

cial de todo sistema e identificar seus módulos.

Neste livro, o foco são a fase de projeto do sistema e sua implementação na linguagem Java.

3.1 Critérios de desenvolvimento

Os seguintes critérios ou normas ajudam na avaliação dos métodos de projeto de software em relação à modularidade: Decomposibilidade em Módulos, Composibilidade de Módulos, Inteligibilidade Modular, Continuidade do Método e Proteção Modular. Um método de projeto de software somente pode ser chamado de modular se satisfizer a esses critérios.

Pelo critério da decomposilidade, requer-se que o projeto possua meios para quebrar a complexidade inicial do sistema, que o método seja baseado na decomposição de problema em subproblemas. E a solução de cada subproblema identificado possa ser desenvolvida separadamente, e os módulos do sistema venham surgindo naturalmente em decorrência do processo.

Um método de projeto atende o critério da composibilidade se ele favorece a produção de componentes de software independentes e que podem ser combinados entre si para produzir novos sistemas. Em geral, métodos desse tipo dão prioridade ao desenvolvimento de componentes isolados para construção de novos sistemas, e não o desenvolvimento de componentes a partir de uma especificação. Esse critério é chave para se conseguir alto grau de reusabilidade de software.

O critério da inteligibilidade é satisfeito por projetos que produzem módulos que podem ser estudados e entendidos isoladamente. Boa encapsulação de dados e funcionamento que independa de contexto são características indispensáveis. Um contra-exemplo seria

um método que enseja a construção de módulos que somente funcionam se ativados em certa ordem.

O critério da continuidade é inspirado na noção de funções contínuas da Matemática. Em analogia com esse conceito diz-se que o método de projeto é contínuo se ele produz módulos tais que pequenas alterações na especificação do sistema afetam poucos módulos. Continuidade é chave para se atingir elevado grau de extensibilidade. Por exemplo, projetos que valorizam o uso criterioso de constantes simbólicas são mais contínuos que aqueles que usam constantes literais livremente. Considere, a título de contra-exemplo, um programa de controle de uma central telefônica que opera com números de telefone de 8 dígitos. Suponha que esses números sejam armazenados em vetores Java declarados da forma:

```
byte[ ] numero = new byte[8].
```

Provavelmente nesse programa, teremos milhares de ocorrências da constante 8, sendo muitas relativas ao comprimento de número de telefone, e outras não. O mesmo pode ser dito, por exemplo, em relação a todo literal cujo valor pode estar vinculado ao número de dígitos do telefone, como a constante 7, que pode ou não representar o comprimento do número menos uma unidade. Observe que, nesse programa, a pequena mudança nos requisitos, de passar o número de dígitos de 8 para 9, causará um esforço brutal de atualização do programa, pois todo o código, linha a linha, terá que ser inspecionado para que as alterações necessárias sejam consistentemente realizadas. Um método de projeto que adota essa prática para o manuseio de constantes não é contínuo.

O último critério, o da proteção modular, valoriza os métodos de projeto que facilitam a construção de módulos que confinam os efeitos de ocorrências anormais, ou então as propagam a um número mínimo de outros módulos. A ideia de minimizar a propagação

de erros, por exemplo, tornando seu tratamento mais modular e com isto reduzindo o custo de manutenção. Prevalece aqui o velho ditado que diz que *boas cercas fazem bons amigos.*

Há dois processos de desenvolvimento que se consagraram ao longo dos anos, o *Projeto Top-Down* e o *Projeto Bottom-Up*, e que ainda fundamentam os processos modernos da Engenharia de Software.

3.2 Desenvolvimento *top-down*

No processo de desenvolvimento de software denominado *Projeto Estruturado Top-Down*, inicia-se com a especificação do software, por meio da definição de sua função principal, que deve ser decomposta ou refinada em subfunções. Cada refinamento pode gerar uma sequência de comandos a ser executados em uma dada ordem ou um comando condicional ou então um comando de repetição, todos contendo subfunções. Esse processo deve se repetir continua e sucessivamente, refinando as subfunções, somente parando quando o nível de abstração da função refinada for diretamente implementável na linguagem de programação alvo.

Esse processo de decomposição da função do sistema forma uma árvore de subfunções, na qual as funções nas folhas da árvore são implementadas como sub-rotinas, cada uma representando uma abstração de comandos ou de expressões.

Esse método de projeto apresenta vantagens e desvantagens. A maior vantagem está na sua capacidade de disciplinar o pensamento do desenvolvedor de forma lógica, organizada e ordenada, oferecendo uma sistemática para quebrar a complexidade inicial do sistema. Essa capacidade conforma-se perfeitamente com a definição de metodologia como sendo *a arte de dirigir o espírito na*

busca da verdade.

Do lado negativo, observa-se que o método *top-down* não leva em consideração a natureza evolutiva do sistema de software. Se a solução do projeto está apoiada em uma árvore de refinamentos, e o software evolui, a árvore terá que ser mudada. E quanto mais próximo da raiz da árvore ocorrer a alteração, maior poderá ser o custo de manutenção e redocumentação do sistema.

Além disto, a ideia de que um sistema de software possa ser caracterizado por uma única função é questionável. Um desenvolvimento baseado somente na função do sistema produz tipos de módulos bastante limitados, pois são invariamente realizações de abstrações de expressões ou de comandos.

Por exemplo, pode-se definir a função principal de um Sistema Operacional como sendo a capacidade de *processar todas as requisições do usuário*, que é refinável em:

```
1 while tudo em ordem do
2    se houver pedido do usuário:
3       leia o pedido;
4       ponha pedido na fila de entrada;
5    retire um pedido r da fila;
6    processe o pedido r;
7 end
```

Entretanto, sistemas de software são mais bem descritos pelos vários serviços que oferecem. O Sistema Operacional, por exemplo, poderia ser descrito como sendo um componente básico que oferece os serviços de: gerência do processador, gerência de memória, entrada e saída e interpretação de comandos. E cada um desses serviços poderia ser refinado adequada e individualmente tendo como base uma estrutura de dados comuns, a qual, na abordagem **top-down** pura, é negligenciada.

Isto significa que a metodologia *top-down* não ajuda a projetar uma boa estrutura de dados para a aplicação. Os dados vão aparecendo à medida que se fizerem necessários e acabam espalhados ao longo da implementação. Por exemplo, se surge a necessidade de uma tabela, na qual dados são inseridos, somente mais tarde, poderá ou não surgir a necessidade por pesquisar a mesma tabela, ou então dela remover elementos. As demandas por eficiência de cada uma dessas operações podem ser conflitantes, levando a alterações complicadas em implementações já definidas.

Uma solução mais interessante poderia ser encontrada se estruturas de dados importantes fossem projetadas completa e isoladamente, por exemplo, como um tipo abstrato de dados, com todas as suas implicações e tudo confinado em um único local para que as modificações sejam efetivas, diretas e mais baratas.

O réuso de componentes de software também não é favorecido pelo método de desenvolvimento *top-down*. Isto ocorre porque a árvore de refinamento naturalmente fixa ordem e contexto de execução, contaminando a reusibilidade do sistema. A prefixação de ordem de execução, aliada à definição de estruturas de dados globais, encoraja a escrita de módulos que funcionam somente em determinados contextos e se ativados em uma ordem específica.

Muitas decisões de projeto determinam indevidamente a arquitetura do programa. Por exemplo, embora a decisão de se ter um aplicativo interativo ou não apenas trate de fixar uma característica do componente interface, a escolha de se ter um aplicativo não-interativo pode leva ao seguinte refinamento inicial:

```
1 Leia valores de entrada;
2 Compute os resultados;
3 Imprima resultados;
```

Enquanto que, no caso de se desejar uma solução interativa, o

refinamento mais apropriado poderia ser:

```
1  if novos dados fornecidos then
2        Leia dados;
3        Armazene dados;
4  elsif há pedidos sobre velhos dados then
5        Recupere informação;
6        Imprima resultado;
7  elsif resultados foram pedidos then
8        if informação está disponível then
9             Recupere resultados pedidos;
10            Imprima resultados;
11       end
12 end
```

Apesar de a interface do sistema ser apenas um de seus componentes, o método exige sua definição prematuramente, e as decisões quanto ao seu modo de operação ficam tão impregnadas na arquitetura da aplicação que qualquer alteração posterior fica inviabilizada.

Essa diversidade nos passos de refinamentos ocorre porque o método de projeto *top-down* propõe-se a responder à pergunta

O que o sistema faz por mim?

em vez de

Sobre o que o sistema faz o quê?

Dessa forma, indevidamente, transfere-se o foco de atenção da estrutura de dados para a função.

O processo *top-down* teve seus dias de glória, mas com a evolução dos mecanismos de abstração, a compreensão de que os tipos de dados manipulados são mais estáveis que as funções que se podem executar e que há uma necessidade cada vez maior por elevados graus de correção e reusabilidade, o desenvolvimento *bottom-up* e seus variantes tomaram seu lugar.

3.3 Desenvolvimento *bottom-up*

Foco de atenção centrado na função pode prejudicar a influência da estrutura de dados na arquitetura do software. Ademais estruturas de dados são mais estáveis do que funções e por isso deveriam ser privilegiadas no projeto.

O foco nos dados favorece Reusabilidade e Extensibilidade, entretanto foco de atenção inteiramente voltado para estrutura de dados também pode não ser a solução.

De fato, a experiência mostra que a melhor solução é o equilíbrio oferecido pelo conceito de *tipos abstratos de dados*, cujas implementações tratam das operações e dos dados sobre os quais elas operam.

Tipo abstrato de dados é possivelmente o conceito mais importante que apareceu depois do computador. Programação orientada por objetos é essencialmente programação com tipos abstratos de dados.

No projeto *bottom-up*, os elementos básicos de uma aplicação são identificados e implementados em primeiro lugar na forma de módulos. A seguir, entidades que usam módulos já implementados são desenvolvidas. Esse processo deve se repetir até que o módulo principal, onde tudo se inicia, seja implementado.

Cada módulo deve ser projetado e implementado sem que se tenha o conhecimento do contexto em que será usado, tornando-o mais independente e, portanto, mais reusável.

3.4 Programação orientada por objetos

Programação orientada por objetos é a construção de sistemas de software como uma coleção estruturada de implementações de tipos abstratos de dados. A menção a **tipo abstrato de dados** sig-

nifica que os módulos devem ser baseados nas abstrações de dados que representam os objetos da aplicação. O termo *coleção* sugere que a metodologia de desenvolvimento seja baseada na montagem *bottom-up* de classes da aplicação, a partir dos objetos identificados na fase de análise. A palavra *estruturada* prevê relacionamentos entre objetos, que podem ser do tipo cliente-servidor ou de hierarquia entre suas classes.

Naturalmente o primeiro passo no projeto de um sistema é descobrir os objetos que devem ser manipulados.

Todo sistema de software deve fornecer respostas a questões do mundo exterior por meio de um modelo operacional baseado na interpretação desse mundo em termos de estruturas armazenáveis no computador. Assim, os objetos que compõem o software devem ser uma representação dos objetos relevantes que constituem o mundo exterior.

Em princípio, para identificar os objetos a ser implementados, basta reconhecer sua presença na parte do mundo exterior que se pretende modelar. Na prática, o processo de descoberta dos objetos a ser implementados é muito mais complicado, e técnicas bem mais sofisticadas devem ser usadas. A análise orientada por objetos serve a esse propósito e tem a função de identificar os principais objetos que compõem um sistema. O resultado da análise é uma descrição dos objetos encontrados.

A diretriz básica do método orientado por objetos é buscar respostas para a pergunta norteadora "*Sobre o que o sistema faz o quê?* ", a qual coloca o foco de atenção nos objetos manipulados pelo sistema. O problema do desenvolvimento orientado por objetos torna-se assim uma questão de como encontrar os objetos, descrevê-los e implementá-los.

Todo sistema orientado por objetos é formado por uma coleção

de classes, que definem os tipos dos objetos que serão criados e manipulados. Os objetos são entidades que proveem uma parte do comportamento geral da aplicação e podem ser:

- objetos transientes de computação, que desaparecem com o fim da aplicação;
- objetos persistentes, que sobrevivem à aplicação, sendo automaticamente armazenados em Bancos de Dados;
- objetos de interface, que proveem a interface humano-computador da aplicação.

O programa principal fica reduzido à função de criar e iniciar os objetos principais da aplicação e disparar a computação. A partir daí, os demais objetos são criados, formando um grafo de objetos, no qual as arestas denotam relacionamentos de diversas naturezas entre os respectivos vértices.

Conclusão

A definição de metodologia como sendo *a arte de dirigir o espírito na busca da verdade* foi estabelecida, há cerca de 360 anos, por René Descartes (1586-1650) [12].

Construir software é uma tarefa muito complexa, que requer a aplicação de metodologias sólidas e bem estruturadas para garantir o sucesso do processo, que, independentemente da metodologia adotada, envolve várias atividades essenciais, como especificação, projeto, implementação, verificação, validação e manutenção.

As diversas metodologias hoje disponíveis, cujo conhecimento certamente são de extrema importância para todo desenvolvedor de software, não são abordadas no presente texto, o qual tem foco na organização e implementação de classes de forma a realçar seus aspectos de modularidade, extensibilidade e reusabilidade.

Exercícios

1. Pesquise o método de projeto denominado *Programação Ágil* e veja se ele pode ser classificado como *bottom-up* ou *top-down*.
2. Explique por que independência de contexto favorece reusabilidade.

Notas Bibliográficas

A noção do que é tipo abstrato de dados foi descoberta na década de 70. O primeiro artigo de importância sobre a matéria é de Barbara Liskov [32], o qual apresenta a linguagem CLU com recursos para encapsulação e criação de novos tipos de dados.

A noção de classes surgiu em 1965 com uma linguagem inicialmente chamada Simula e depois Simula 67 [10, 27]. Simula 67 introduziu os conceitos-chave da programação orientada por objetos, tais como objetos, classes, subsclasses, procedimentos virtuais, referências seguras e polimorfismo de inclusão. Mais tarde, em 1972, o americano Alan Kay descobriu a orientação por objeto e a revelou com a linguagem Smalltalk [20, 21], que foi desenvolvida no início da década de 70 pelo *Learning Research Group* do Palo Alto Research Center da Xerox. As principais ideias incorporadas em Smalltalk têm suas raízes em Simula e Lisp.

As metodologias de desenvolvimento de software modernas, que são derivadas dos processos aqui apresentados, podem ser encontradas nos bons textos de Engenharia de Software como o recente livro de Marco Túlio Valente [54], que aborda com propriedade diversos tipos de processos e modelos de software.

Capítulo 4

Modularidade

Mending Wall [**Robert Frost**]

... ...
Good fences make good neighbors.
... ...
Before I built a wall I'd ask to know
What I was walling in or walling out,
And to whom I was like to give offense.
... ...

Modularização é uma ideia que permeia os processos de construção de artefatos em todas as áreas do conhecimento, os quais, via de regra, combinam peças de uma coleção de componentes para produzir o efeito desejado. Essas peças são geralmente padronizadas para ser utilizadas em diferentes projetos. E esse processo de componentização facilita a montagem de artefatos de diversos níveis de complexidade, de arquitetura flexível e geralmente reduz o custo final de construção. Cada uma dessas peças corresponde ao que é chamado de módulo em desenvolvimento de software.

Módulo, nesse contexto, é um componente de software compilável separadamente, como uma `unit` do Turbo Pascal, um arquivo fonte em C e C++, um arquivo com uma ou mais classes

em Java e um pacote de ADA, sendo que todos têm o objetivo de implementar abstrações de diferentes tipos.

Da mesma forma que em outros ramos do conhecimento, módulos em programação são essencialmente componentes que executam uma ou mais funções e que podem ser construídos separadamente de outros para formar o artefato final.

Modularização é a única ferramenta disponível para quebrar a complexidade de grandes sistemas, facilitar a divisão de tarefas entre os membros da equipe de desenvolvimento e reduzir o custo do software, além dos benefícios como extensibilidade e reusabilidade decorrentes do processo. E esse objetivo somente é atingido a partir de módulos construídos conforme os critérios de qualidade e princípios como os apresentados neste capítulo.

Um conceito basilar no processo de produção de módulos é a *encapsulação*, que é o ato de compartimentalizar a estrutura de dados e definição do comportamento de uma abstração, ou seja, encapsulação é um mecanismo usado para agrupar e *esconder*, em um mesmo local, as estruturas internas e detalhes de implementação de um objeto, forçando que toda interação seja via canais de comunicação bem definidos, dessa forma separando a interface contratual do componente de sua implementação. Esconder apenas significa permitir o acesso às estruturas de dados encapsulada somente via operações definidas na interface do módulo.

O processo de definição de módulos deve ser cientificamente bem fundamentado e seguir princípios que orientem sua construção de forma a garantir uma estrutura modular de boa qualidade. Os primeiros princípios nessa linha são: *Unidade Linguística*, *Baixa Conectividade*, *Interface Pequena*, *Interface Explícita*, *Ocultação de Informação* e *Unidade Focal*.

4.1 Unidade linguística

Princípio da Unidade Linguística
A todo tipo de abstração realizável na linguagem deve corresponder um tipo de módulo.

Modularizar consiste em encapsular construções, como declarações, comandos e expressões, que ocorrem em um programa, construindo uma entidade com as funções e semântica desejadas.

Esse processo é facilitado se a todo tipo de abstração desejada existir formas de expressá-la como um módulo codificado na linguagem de programação em uso. A tabela a seguir associa tipos de módulos às entidades de programa que se pode desejar abstrair-se:

Tipo de Construção	Tipo de Módulo
Expressões	Função
Comandos	Procedimento
Declarações de variáveis	Abstração de dados
Classes	Tipo abstrato de dados
Classes genéricas	Tipo parametrizado

Toda linguagem de programação modular deve necessariamente prover o suporte linguístico para a realização desses tipos de módulos. Pois, sem o suporte linguístico adequado é muito difícil construir módulos que efetivamente ajudam a quebrar a complexidade de software de grande porte.

Java é uma linguagem que atende a esse requisito, pois funções e procedimentos são implementáveis por meio de *métodos*, abstração de dados e tipos abstratos usam classes, e tipos parametrizados, classes genéricas.

4.2 Baixa conectividade

Princípio da Baixa Conectividade
Todo módulo deve comunicar-se com um número mínimo de outros módulos.

Um projeto modular devem priorizar a criação de módulos de baixa conectividade. Isso significa que todo módulo deve comunicar-se com um número mínimo de outros módulos.

Para se atingir esse objetivo de baixo grau de conectividade, os seguintes critérios de projeto devem ser observados no processo de identificação e construção de módulos:

- Continuidade: módulos devem ser definidos de tal forma que alterações na especificação do problema tenham efeito apenas em poucos módulos, ou seja, pequenas mudanças na especificação devem provocar apenas pequenas alterações no código.
- Proteção: módulos devem confinar as ocorrências de erros, propagando seu efeito para poucos outros módulos.
- Inteligibilidade: módulos devem ser construídos privilegiando sua legibilidade por humano, e isso requer baixo grau de conectividade.

Para ilustrar o efeito do grau de conectividade de módulos no custo de manutenção de software de grande porte, considere um sistema proposto por G.J. Meyer de 100 lâmpadas, que podem ser conectadas de alguma forma e satisfazem as seguintes propriedades:

- Toda lâmpada que estiver ligada tem a probabilidade de 50% de desligar-se no próximo segundo.
- Toda lâmpada que estiver desligada e conectada diretamente a pelo menos uma lâmpada ligada tem a probabilidade de 50% de ligar-se no próximo segundo.
- Toda lâmpada desligada assim permanece enquanto estiver conectada diretamente a somente lâmpadas desligadas.

Suponha que cada lâmpada represente um módulo, que acender uma lâmpada equivale a dizer que a interface do módulo correspondente foi modificada e que apagar a lâmpada significa que o módulo correspondente foi devidamente consertado.

Suponha ainda que inicialmente todas as 100 lâmpadas estejam acesas, ou seja, todos os 100 módulos sofreram modificações em suas interfaces. Nesse contexto, deseja saber o tempo necessário para que o sistema se estabilize com todas as lâmpadas apagadas, considerando as seguintes três configurações das conexões entre elas:

- 100 lâmpadas totalmente desconectadas;
- 100 lâmpadas completamente conectadas;
- 100 lâmpadas conectadas em grupos de 10 e os grupos desconectados entre si.

Na primeira configuração, quando nenhuma lâmpada está conectada, a metade apaga-se no segundo seguinte. Como essas 50 lâmpadas não estão conectadas, elas nunca se reacenderão, e no próximo segundo, outras 25 lâmpadas apagam-se. Em 7 segundos, todas as lâmpadas estarão apagadas, conforme a sequência `(50, 25, 12, 6, 3, 2, 1)`. Portanto, o equilíbrio será atingido em 7 segundos.

Para caso de lâmpadas reunidas em 10 grupos de 10, estando todas conectadas em cada grupo, Meyers demonstrou que o equilíbrio é atingido em 20 minutos.

E, por fim, com 100 lâmpadas completamente conectadas, o equilíbrio, segundo Meyers, requer em 10^{22} anos para ser atingido.

Esses resultados mostram que conectividade tem relação direta com complexidade de um sistema e que, ao se estabelecer uma conectividade entre módulos, está-se diretamente aumentando seu custo de manutenção.

4.3 Interface pequena

Princípio da Interface Pequena
Se dois módulos não podem evitar de se comunicarem, então a comunicação entre eles deve ser a mínima possível.

O controle do volume e tipo de comunicação entre módulos é fundamental para se obter sistemas modulares. Um excelente contra-exemplo seria um sistema que tem um módulo descritor de dados que encapsula todas as estruturas de dados usadas em um programa e todos os demais módulos acessam as partes que lhes interessam desse módulo de dados.

A consequência nefasta desse modelo de organização é que qualquer alteração no módulo de dados implica na análise e depuração de todos os demais módulos, o que pode ser extremamente caro.

O fato é que a melhor forma de comunicação entre módulos deve ser apenas via parâmetros passados por valor.

4.4 Interface explícita

Princípio da Interface Explícita
Se dois módulos A e B se comunicam, então esse fato deve ser evidente pela simples inspeção do texto de A ou de B.

O efeito de toda alteração no código de um programa em decorrência do processo de manutenção tem que ser completamente identificado e devidamente absorvido por todos os módulos afetados. Se a propagação desse efeito não for facilmente visível, não há como garantir a correção do processo de manutenção.

A aplicação desse princípio está relacionado com os seguintes critérios:

- Composibilidade: composição só é possível se conexões forem claras.

- Continuidade: componentes afetados por mudanças devem ser óbvios e limitados.
- Inteligibilidade: como entender comportamento de um módulo que é influenciado por outro módulo de forma misteriosa?

Para ilustrar o valor de interfaces explícitas, considere os módulos `A` e `B` de uma biblioteca `BiblioX` definidos a seguir, onde o arquivo `A.java` tem o seguinte código:

```
1  package BiblioX;
2  public class A {
3     private int pri = 1;
4     protected int pro = 3;
5     public int op(int z){
6        X x = new X();
7        Y y = new Y();
8        pri = x.pac + y.op(z) + pri + pro;
9        return pri;
10    }
11 }
12
13 class X{
14    private int pri = 1;
15    int pac = 2;
16    ...
17 }
18
19 class Y{
20    private int pri = 1;
21    int op(int z) { return z+pri; };
22 }
```

Observe que dentro da biblioteca `BiblioX`, os tipos disponíveis são `A`, `X` e `Y`, e fora dessa biblioteca, somente `A`, cuja interface disponibiliza os seguintes campos e operações:

- operação `int op(int z)`

- campo `pro`, exportado para subclasses e outros módulos do pacote
- classes `X` e `Y`, exportadas para outros módulos da mesma biblioteca
- campo `X.pac`, exportado para outros módulos da biblioteca
- método `Y.op(int z)`

E o arquivo módulo `B.java` tem o seguinte código:

```
1  package BiblioX;
2  public class B  {
3     private int pri = 1;
4     protected int pro = 3;
5
6     private static class X {
7        private int pri = 1;
8        int pac = 2;
9        ...
10    }
11    private static class Y {
12       private int pri = 1;
13       int op(int z) {return z+pri;}
14    }
15
16    public int op(int z) {
17       X x = new X();
18       Y y = new Y();
19       pri = x.pac + y.op(z) + pri + pro;
20       return pri;
21     }
22 }
```

A interface do módulo `B`, por ser bem menor que a de `A`, é de melhor qualidade, pois contém apenas:

- operação `int op(int z)`

- campo `pro`, exportado para subclasses e outros módulos do pacote.

Essa qualidade superior é claramente demonstrada pelo módulo `C`, que usa as entidades exportadas por `A` e `B`:

```
1  package BiblioX;
2  public class C {
3      ...
4      public int f() {
5          A a = new A();
6          B b = new B();
7          X x = new X();
8          Y y = new Y();
9          return  x.pac + y.op(2) + a.op(3) + b.op(4);
10     }
11     ...
12 }
```

Uma simples inspeção do código acima mostra que `C` depende de `A` e `B`, que são módulos, e de `X` e `Y`, que não são módulos.

Por outro lado, a simples inspeção do módulo `C` não permite determinar em que arquivos X e Y estão definidos: o diretório `BiblioX` não possui arquivo X.java nem Y.java, pois essas classes estão dentro de A.java. Portanto, pode ser necessário inspecionar outros arquivos da biblioteca BiblioX para compreender a semântica de `C`. Nesse sentido, módulo B tem uma estrutura de melhor qualidade que a de A.

No caso de uso de `A` e `B` serem usados em módulo independente, definido em uma outra biblioteca, e.g., `BiblioY`, as interfaces de `A` e `B` são de qualidades similares, pois as classes `X` e `Y` e campos declarados `protected` estão fora de alcance. O programa `Main` a seguir mostra esses detalhes:

```
1  package BiblioY;
2  import BiblioX.*;
3  public class Main {
4     public static void main(String[] args) {
5        A a = new A();
6        B b = new B();
7        int x = a.op(10) + b.op(20);
8        System.out.println("x = " + x);
9     }
10 }
```

onde o uso de `X` e `Y` definidos no módulo `A` não é permitido.

4.5 Ocultação de informação

Princípio da Ocultação de Informação
Toda informação a respeito de um módulo deve ser privativa do módulo, exceto se for explicitamente declarada pública.

Todo módulo deve ser conhecido pela sua interface. Ocultação de informação facilita a manutenção da integridade do sistema a longo prazo e, principalmente, privilegia, os critérios de continuidade, composibilidade e inteligibilidade.

A aplicação desse princípio tem por base o papel da encapsulação, que pode ser vista com um recurso para definir os serviços que um módulo torna disponíveis para seus clientes e proteger os seus demais elementos de acessos indesejáveis.

No contexto de linguagens orientadas por objetos baseadas no conceito de classes que participam de hierarquias de tipos, módulos que são classes têm dois tipos de clientes: o externo, que tem acesso somente a seus componentes públicos, e o cliente descendente, que tem acessos aos componentes públicos e aos protegidos.

Essa ideia de módulos terem dupla interface: uma para os estranhos e outra para a família pode ter alguma relação com a vida real, mas do ponto de vista programação modular de boa qualidade deve ser evitada.

4.6 Unidade focal

Princípio da Unidade Focal

Todo módulo deve servir a um único propósito, estar focado em um aspecto específico do sistema, de tal forma a se ter apenas um motivo para ser alterado.

Módulos que implementam aspectos distintos de um sistema, como os módulos do tipo *mistifório*, discutidos na Seção 5.5, têm mais motivos de serem modificados em consequência das inevitáveis alterações na especificação de todo sistema. E isso tem consequência direta no grau de conectividade do sistema, pois toda vez que um módulo precisa ser alterado, todos os que dele dependem devem ser inspecionados para ver o efeito das alterações realizadas.

4.7 Boas práticas

Um módulo TAD deve implementar um único tipo de dados. Isso decorre da ideia de cada módulo deve implementar um único conceito, conforme aconselhado pelo Princípio da Unidade Focal.

Isso implica que todos os campos de qualquer classe devem, por *default*, ser privados. Somente a classe pública de um módulo deve poder ser referenciada fora dele.

E deve-se evitar ao máximo acesso a classes que não sejam públicas, mesmo dentro do próprio pacote. Para garantir essa organização, classes não-públicas devem ser implementadas como classes aninhadas da classe pública do módulo.

Em Java pode-se ter uma classe dentro de outra classe. Se a classe interna é pública, ela é válida fora de A. Se ela for privada, ela só vale dentro de A, contudo ela pode ter membros públicos, mas essa publicidade é só do lado de fora de sua classe e não do lado de fora da classe que a está envolvendo.

Para assegurar legibilidade e minimizar conectividade, deve-se observar as seguintes restrições ao declarar classes aninhadas:

- usar apenas um nível de aninhamento;
- declará-las estáticas para impedir referências a campos da classe envolvente;
- programá-las como se fossem classes de primeiro nível;
- declará-las privadas sempre que possível.

O seguimento dessas regras torna o módulo mais coeso, como ilustra o seguinte pequeno exemplo, que mostra um módulo `A` encapsulando completamente uma classe auxiliar `C`, enquanto exporta uma classe `B` a ele fortemente vinculada.

```
1  public class A {
2     public  static class B {
3        private int x;
4        private int y;
5        ...
6     }
7     private static class C {
8        public int z;
9        ...
10    }
11    ...
12 }
```

Observe que fora da classe `A`, pode-se escrever A.B, mas `A.C`, `A.B.x` e `A.B.y` não são permitidos.

4.8 Bibliotecas

A estrutura de um sistema orientado por objetos pode ser a da Fig. 6.2, onde camadas são usadas para disciplinar o fluxo de mensagens, conforme descrito no Capítulo 6.1, acervos agrupam bibliotecas afins e estas reúnem módulos, que são compostos por classes.

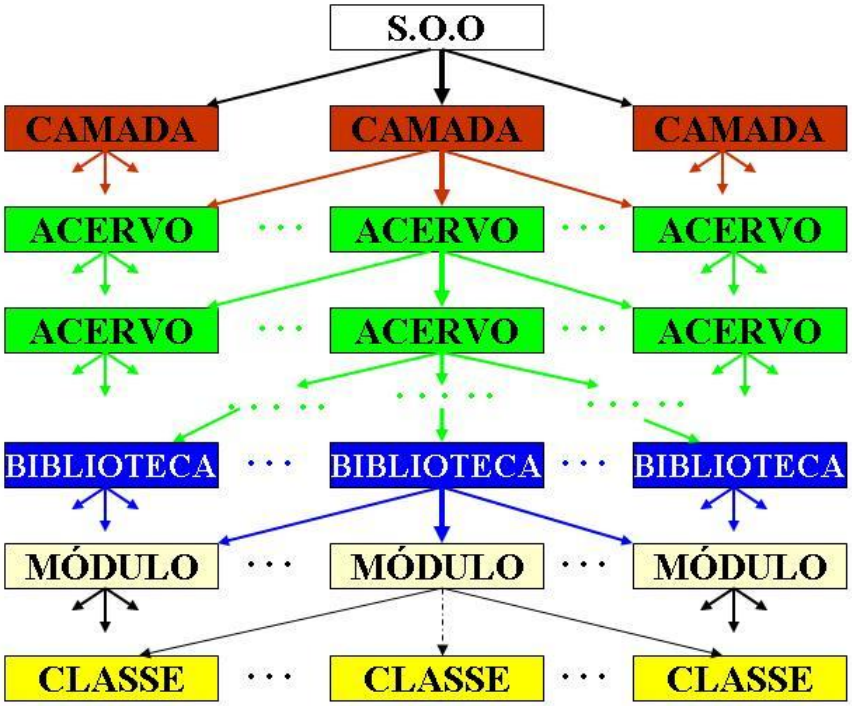

Figura 4.1 Organização de um Sistema OO

Essa hierarquia corresponde à arvore de diretórios do sistema de arquivos do ambiente de execução, onde os arquivos contêm as classes. Uma biblioteca, que em Java é chamada de pacote, é uma pasta contendo uma coletânea de arquivos, cada um contendo declarações de tipos, os quais podem ter no máximo a declaração de um tipo público, o qual pode ser usado localmente ou em outras bibliotecas. As demais classes ou interfaces não-públicas, declaradas em arquivos de um pacote são estritamente locais.

O exemplo a seguir ilustra a definição dos módulos `A` e `B`, declarados na biblioteca `Biblio1`, pertencente ao `Acervo1` da `Camada1`.

O arquivo `A.java` é o seguinte:

```
1  package Camada1.Acervo1.Biblio1;
2  public class A {
3     private static class X {
4        private int pri = 1;  int pac = 2; ...
5     }
6     private static class Y {
7       private int pri = 1;
8       int op(int z) {return z + pri;}
9     }
10    private int pri = 1;  protected int pro = 3;
11    public int op(int z) {
12        X x = new X();  Y y = new Y();
13        pri = x.pac + y.op(z) + pri + pro;
14        return pri;
15    }
16 }
```

E o arquivo `B.java` é o seguinte:

```
1  package Camada1.Acervo1.Biblio1;
2  public class B {
3     private static class X {
4        private int pri = 1; int pac = 2; ...
5     }
6     private static class Y {
7        private int pri = 1;
8        int op(int z){return z+pri;};
9     }
10    private int pri = 1; protected int pro = 3;
11    public int op(int z) {
12      X x = new X();  Y y = new Y();
13      pri = x.pac + y.op(z) + pri + pro;
14      return pri;
15   }
16 }
```

O módulo principal, declarado no arquivo `MainB` e que faz uso de `A` e `B`, é o seguinte:

```
1  package Camada1.Principal;
2  import Camada1.Acervo1.Biblio1.*;
3  public class MainB {
4     public static void main(String[] args) {
5        A a = new A();
6        B b = new B();
7        int x = a.op(10) + b.op(20);
8        System.out.println("x = " + x);
9     }
10 }
```

Conclusão

Modularizar é a atividade central no desenvolvimento de sistemas de grande porte, e módulos devem ser autônomos e independentes, de forma a minimizar sua conectividade e reduzir os custos de manutenção.

Exercícios

1. Tendo em vista os conceitos apresentados neste capítulo, em que circunstâncias campos de uma classe podem ter visibilidade pública?
2. Há algum outro tipo de módulos além dos citados na Seção 4.1?

Notas bibliográficas

Um excelente texto sobre modularidade é o livro de Bertrand Meyer [41], que apresenta grande parte dos conceitos aqui descritos.

O modelo de 100 lâmpadas foi proposto por Glenford J. Myers [42] para demonstrar o efeito explosivo do alto grau de conectividade no custo de manutenção.

Capítulo 5

Estruturação de Módulos

Módulo é uma porção de programa compilável separada ou independentemente, e que pode ser incorporado a outros programas como código compilado. Ressalva-se que compilação em separado é distinta de compilação independente, porque, na compilação em separado, se um módulo depender de outros, as interfaces destes devem ser providas para que compilação tenha efeito.

A porção de código que pode formar um módulo varia de linguagem para linguagem. Por exemplo, na linguagem C, um módulo pode ser formado por declarações de variáveis e/ou funções, que após sua compilação podem ser usadas em outros módulos.

Em Modula 2, a estrutura `module` permite encapsular definições de tipos, declarações de estruturas de dados, funções, procedimentos e definir os itens exportados.

Em Java, um módulo é formado pela declaração de uma classe pública e possivelmente outras classes não-públicas. Além dessas estruturas, módulos de Java podem conter diretivas de compilação, como cláusulas de importação de tipos, que ajudam na compilação em separado, e um especificador `package <biblioteca>`, que define a biblioteca a que o módulo pertence. Cada módulo Java tem uma interface para o mundo exterior, que é a de sua única classe pública, e uma interface para módulos de sua própria biblioteca, que são classes declaradas sem modificadores de visibilidade. Res-

trições adicionais de visibilidade de tipos exportados por módulos podem ser impostas por um mecanismo denominado `module`, introduzido pela Edição SE 14 da linguagem Java.

Um módulo pode ser formado por porções de programa de natureza diversa, relacionadas ou não entre si, e encapsuladas em uma unidade de compilação distinta, podendo ser classificado como Módulo-Função, Módulo-Abstração, Módulo-Tipo, Módulo-Parametrizado ou Módulo-Mistifório, de acordo com seu conteúdo.

5.1 Módulos-função

Módulos-função são aqueles que implementam e exportam um conjunto de funções ou métodos independentes, mas relacionadas com um problema específico.

Funções são um importante tipo de abstração que deve estar disponível em todo ambiente de programação. Boas funções normalmente não fazem acesso a dados externos que não sejam os que lhe forem passados via parâmetros nem causam efeito colateral no ambiente de execução.

Frequentemente, há funções fazem parte de conjunto de abstrações ligadas a domínios específicos de determinadas aplicações, como cálculos matemáticos relacionados, análises estatísticas e pacotes gráficos, sendo conveniente reuni-las em um único módulo conforme sua aplicação.

A rigor, cada uma dessas funções deveria ser um módulo individual. Entretanto, o agrupamento de funções fortemente relacionadas em um único módulo facilita seu gerenciamento, e a independência de uma em relação a outra reduz o possível impacto negativo do agrupamento na conectividade do sistema.

Módulos-função geralmente não contêm declarações de estrutu-

ras de dados de interesse comum e são implementados por meio das chamadas *classes empacotadoras*, que são classes cujo foco principal são declarações de métodos e não de campos. No contexto de módulos funcionais essas classes empacotadoras devem ser declaradas públicas, bem como as funções que empacotam, e devem ser vinculadas a bibliotecas específicas.

Exemplos ilustrativos de módulos funcionais são os que implementam funções de domínios específicos de certos campos do conhecimento como:

- Módulo `Trigonometria.java`:

```
1 package Matemática;
2 public class Trigonometria {
3    public static float sin(float x) { ... }
4    public static float cos(float x) { ... }
5    public static float tan(float x) { ... }
6    ...
7 }
```

- Módulo `Geometria.java`:

```
1 package Gráfico;
2 public class Geometria {
3    public static float quadrado(float lado) { ... }
4    public static float retângulo(float a, float b){...}
5    public static float círculo(float raio) { ... }
6    ...
7 }
```

- Módulo `Atuária.java`:

```
1 public class Atuária {
2    public static boolean aposentável(int idade, int t);
3    ...
4 }
```

5.2 Módulos-abstração

Na linguagem C, módulo abstração de dados é um arquivo contendo declarações de variáveis e funções. Em Java, é um arquivo contendo uma classe pública em que todos os seus campos são declarados `private static` e que contém um conjunto de funções públicas e estáticas para manipulá-los.

O ponto central nesse tipo de módulo é que se permite somente uma instância da abstração declarada, como demonstra o exemplo a seguir:

```
1 package abstração;
2 public class Buffer {
3    private static char[] entrada =  new char[80];
4    public static void leiaLinha() { ... }
5    public static char getChar() { ... }
6    public static boolean linhaVazia() { ... }
7 }
```

Em módulos que importam a biblioteca `abstração`, definida acima, os métodos `Buffer.leiaLinha()`, `Buffer.getChar()` e `Buffer.linhaVazia()` podem ser acionados, sem a necessidade de se criar objetos do tipo `Buffer`, valorizando o fato de o módulo ser uma abstração de dados única.

5.3 Módulos-tipo

Um módulo-tipo é um arquivo contendo uma ou mais estruturas de dados e operações que contribuem para implementar *um* tipo abstrato de dado, caracterizado por um único e relevante contrato público, definido por uma interface clara e explícita. Em linguagens orientadas por objetos, a estrutura mais importante para implementar um novo tipo de dado é a classe.

Esse tipo de módulo pode encapsular diversas classes, mas deve implementar apenas um tipo abstrato de dados. As demais estruturas nele contidas são de uso apenas local. Em Java, isso é parcialmente automático, pois somente uma classe pode ser pública por módulo, e assim do ponto de vista de módulos não-pertencentes à mesma biblioteca esse requisito está resolvido. Para fazê-lo valer dentro da biblioteca do módulo é recomendado encapsular as estruturas auxiliares dentro da classe pública, como exemplificado pela seguinte implementação do tipo `Lista`, que esconde a declaração da classe `Nodo`, mas exporta a ficha de dados `Lista.Info`:

```
1  public class Lista {
2     public static class Info {public String nome, cpf;}
3     private static class Nodo {
4        public  Info v;
5        public  Nodo próximo;
6        public  Nodo(Info v) {this.v = v;}
7     }
8     private Nodo cabeça;
9     public Lista( ) {cabeça = null; }
10    public void insira(Info v) {... new Nodo(v); ...}
11    public Info retire(){Nodo x; ... ; return x.v; }
12 }
```

5.4 Módulos-paramétricos

Módulos-paramétricos são aqueles que encapsulam declarações e operações associadas e relacionadas, realizando a implementação de um tipo contêiner com uma representação parametrizada pelo tipo de objetos por ele encapsulados. Esses módulos são em essência tipos abstratos de dados paramerizados e são geralmente implementados por meio de interfaces e classes genéricas.

5.5 Módulos-mistifório

Módulos que encapsulam uma miscelânea de construções, como declarações de dados, funções e procedimentos, a maioria sem qualquer relação entre si é um mistifório[1]. Esses módulos são resultados de classes mal-construídas, chamadas *depositárias*, *fantasmas* ou *polivalentes*.

As depositárias são classes públicas de poucas instâncias, seus objetos são depósitos de dados, seus campos não são relacionados, seus métodos são, em sua maioria, do tipo `set` e `get`, como ilustrado pela classe `Orange` a seguir:

```
1 public class Orange {
2    private float temperatura, juros;
3    public void setTemperatura(double t) {temperatura = t;}
4    public double getTemperatura() {return temperatura;}
5    public void setJuros(double a) {juros = a;}
6    public double getJuros() {return juros;}
7 }
```

As classes-fantasma são classes públicas que não possuem declarações de campos e seus métodos não são relacionados, i.e., seus métodos apenas operam sobre seus parâmetros, portanto sobre dados de terceiros. Diferenciam-se de classes que implementam módulos-função pelo fato de seus métodos não serem relacionados. A classe `Fantasma` abaixo é um exemplo a não ser seguido:

```
1 public class Fantasma {
2     public void matricula(String nome,Curso c) {...}
3     public double imposto(double salario) { ... }
4     public double temperatura(float t) { ...}
5 }
```

[1]Mistifório é uma confusão de coisas ou pessoas, mistura de elementos que se repelem.

Classes-polivalentes são classes que implementam mais de um contrato. Seus métodos e campos podem ser particionados em subcontratos, conforme sugere o seguinte exemplo:

```
1  public class Esquizo {
2      private int[] pilha;
3      public void empilhe(int x) {...}
4      public int desempilhe() {...}
5      public boolean pilhaVazia(int x) {...}
6
7      private int[] lista;
8      public Lista(...) {...}
9      public void insert(int x) {...}
10     public int remove() {...}
11     public boolean listaVazia(int x) {...}
12 }
```

Essa organização modular não é recomendada, pois ela cria uma situação em que a manutenção de um mesmo módulo pode ter que ser feita por motivos não-relacionados. Melhor seria se o módulo `Esquizo` acima fosse dividido em dois, cada um implementando um TAD distinto.

5.6 Coesão interna de métodos

O propósito de um método é o de implementar uma dada função ou procedimento, sendo a natureza dessa função o principal critério de medida de sua coesão interna, que deve ser computada em função do relacionamento entre seus elementos para atingir o objetivo final.

A coesão interna de um método é a de menor intensidade que caracterize alguma de suas ações, conforme a seguinte classificação em ordem crescente: Coincidente, Temporal, Lógica e Funcional.

1. Coesão coincidente de método: Um método tem essa coesão quando contém ações que não são relacionadas entre si, estando vinculadas a propósitos diversos. Em geral, esse tipo de método surge como resultado de uma modularização a posteriori com o objetivo de evitar duplicação de código, por exemplo, transformando uma sequência de código em uma rotina a ser chamada sempre que essa sequência for necessária, como ocorre no seguinte trecho de programa:

```
1  A x;
2  ...
3  a = b + c;
4  ...
5  d = x.getint();              |
6  System.out.print(h);         | código repetido
7  if (r = s) d++ else d--      |
8  ...
9  d = x.getint();              |
10 System.out.print(h);         | código repetido
11 if (r = s) d++ else d--      |
12 ...
```

O código repetido identificado acima pode ser encapsulado em um método, e.g., `M.m`, cujos parâmetros definem seu contexto de execução, sendo `M` a seguinte classe:

```
1 public class M {
2    public static int m(A x, int h, char r, char s) {
3       int d = x.getint();
4       System.out.print(h);
5       if (r = s) d++ else d--
6       return d;
7    }
8 }
```

O uso de `M.m` permite reescrever o código original como:

```
A x;
...
a = b + c;
...
d = M.m(x,h,r,s);
...
d = M.m(x,h,r,s);
...
```

Esse é o pior grau de coesão possível, devendo ser evitado! Os elementos encapsulados não têm relação alguma entre si.

2. Coesão temporal de método: Métodos têm coesão temporal quando contêm ações não-relacionadas, exceto no tempo e na ordem em que são executadas. Geralmente, são ações relacionadas com detalhes de implementação e não com o problema em foco e operam sobre diferentes objetos. Por exemplo, abrir/fechar todos os arquivos da aplicação. O principal defeito desse tipo de módulo é ter mais de um motivo para ser alterado, aumentando conectividade do sistema.

3. Coesão lógica de método: Um método tem esse nível de coesão interna quando implementa mais de uma função, dentre as quais uma deve ser escolhida pelo usuário via a passagem de um parâmetro. Um exemplo é o de um único método que pode fazer inserção, remoção e pesquisa de um valor em uma tabela, conforme um dos parâmetros que lhe for passado:

```
enum Operação {INSIRA, REMOVA, PESQUISE}
T tabela(Operação op, T x) {
    switch (op) f {
        case INSIRA: ...;
        case REMOVA: ...;
        case PESQUISE: ...;
    }
}
```

Melhor seria ter um método distinto para cada função. Métodos que compõem módulos-função deveriam poder ser acionados de forma independentes. Fundir diferentes funções em um único método prejudica essa independência e dificulta o processo de manutenção.

4. Coesão funcional de método: Um método têm coesão funcional quando todas as suas ações são no sentido de implementar uma única função.

5.7 Coesão interna de classes

O comportamento de uma classe é definido pelo conjunto dos comportamentos de seus métodos. Portanto, as qualidades ou defeitos de um método contaminam o perfil da classe que o contém. Consequentemente, a coesão interna de uma classe depende do grau de coesão de seus métodos, e, além disso, a coesão da classe é função da afinidade do relacionamento entre os seus métodos.

O grau de coesão interna de uma classe é o de menor intensidade em que a classe se enquadra na seguinte ordenação crescente da sua intensidade.

1. Coesão coincidente: a classe implementa mais de um contrato ou contém métodos não-relacionados, métodos de coesão coincidente ou declara dados não-relacionados.
2. Coesão lógica: a classe contém um ou mais métodos de coesão lógica.
3. Coesão temporal: a classe contém um ou mais métodos de coesão temporal ou grupos de métodos que precisam sempre ser executados em conjunto. Um exemplo é uma classe com métodos que se destinam a iniciar o ambiente de execução como na seguinte classe:

```
1 class Ambiente {
2       public void abreArquivodeDados() {...}
3       public void abreArquivodeLog() {...}
4 }
```

4. Coesão procedimental: a classe contém métodos cuja ordem de execução é semanticamente relevante. Ocorre, por exemplo, quando a iniciação do objeto é via métodos específicos e não via funções construtoras.
5. Coesão contratual: a classe implementa um único tipo de dados, todos os seus métodos têm coesão funcional e são relacionados com o objetivo da implementação do tipo e todos seus campos são privados.
6. Coesão funcional: implementa um conjunto de funções relacionadas com um tema específico e todas têm coesão funcional.

5.8 Coesão interna de módulos

Coesão interna de um módulo trata do relacionamento entre os seus elementos, os quais são declarações de campos, comandos, funções ou classes. A idéia central é que elementos fortemente relacionados deveriam ficar confinados juntos em módulos específicos, e aqueles não-relacionados em outros módulos. A obediência a essa clara separação de interesses pode ser medida pelo grau de coesão interna de cada módulo, a qual depende da coesão de seus constituintes, que, em linguagens orientadas a objetos pura, são essencialmente classes, cujos níveis de coesão dependem dos de seus métodos. Assim, a coesão interna de um módulo está diretamente relacionada com a das classes que o constituem e do grau de relacionamento existente entre elas.

Define-se que a coesão interna de um módulo é no máximo a

de seu constituinte que apresentar o pior grau de coesão. A determinação da coesão interna de um módulo deve iniciar-se com a avaliação da coesão dos métodos que compõem as classes nele encapsuladas, seguido da avaliação da coesão de cada uma dessas classes e depois da medida do relacionamento existente entre essas classes.

Glenford Myers, em 1975, classificou a coesão interna de módulos nos sete níveis, listados a seguir em ordem crescente após terem sido adaptados para o jargão das linguagens orientadas por objetos. São eles: Coincidente, Lógica, Temporal, Procedimental, Comunicacional, Informacional e Funcional.

A coesão de um módulo é por definição a de menor nível no qual ele se enquadrar na seguinte classificação:

1. Coesão coincidente: é o maior grau de coesão atribuível a módulos que contêm uma ou mais classes de coesão coincidente. Esse é o grau de coesão interna de módulos do tipo mistifório, que contêm elementos não-relacionados.
2. Coesão lógica: é o maior grau de coesão atribuível a módulos que contêm uma ou mais classes de coesão lógica.
3. Coesão temporal: é o maior grau de coesão atribuível a módulos que contêm uma ou mais classes de coesão temporal e/ou classes que se relacionam apenas pelo momento em que seus métodos são necessários.
4. Coesão procedimental: é o maior grau de coesão atribuível a módulos que contêm uma ou mais classes de coesão procedimental.
5. Coesão comunicacional: é o maior grau de coesão atribuível a módulos que contêm classes cujos métodos requerem ordens específicas de execução por serem dependentes de dados comuns.

6. Coesão informacional: é o maior grau de coesão atribuível a módulos que implementam um único tipo abstrato de dado via uma classe de coesão contratual. Módulos-abstração, módulos-tipo e módulos-paramétricos são candidatos a terem esse nível de coesão.
7. Coesão funcional: é o maior grau atribuível a módulos que implementam uma classe de coesão funcional. Módulos-função são os únicos candidatos a terem esse nível de coesão interna.

Bons módulos são os que têm coesão interna de grau *informacional* ou *funcional*. Módulos com os demais níveis de coesão interna deveriam ser evitados, porque tendem a elevar a conectividade do grafo de módulos e, consequentemente, elevam o custo de manutenção do software.

5.9 Acoplamento de módulos

Módulos de qualquer sistema geralmente devem ser interconectados, formando um grafo dirigido de conexões entre módulos-cliente e módulos-servidor. Identificam-se, nesse contexto, dois conceitos: a interconexão de módulos, ou conectividade, e a intensidade dessas interconexões, denominada acoplamento.

A interconexão entre módulos implementa uma relação de dependência entre cliente e servidor, havendo pouco espaço para sua reestruturação. Entretanto, como acoplamento existente entre dois módulos é uma medida intensidade da interconexão ou da dependência existente entre eles, é possível organizar uma arquitetura que minimize a complexidade desse relacionamento, valorizando a modularidade do sistema. A regra fundamental nesse processo é que quanto mais fraca a intensidade da interconexão entre os módulos, mais modular é o sistema.

A ideia de classificar o acoplamento de módulos foi proposto por Glenford Myers em 1975, em uma época em que tipos abstratos de dados eram conceitos incipientes, e as mais importantes abstrações eram essencialmente procedimentos e funções.

Atualmente, os principais tipos de módulos estão centrados na ideia de classes que implementam serviços, via seus métodos. Assim, o relacionamento entre métodos de classes distintas forma o relacionamento das respectivas classes. O acoplamento dessas classes trata, portanto, do inter-relacionamento de seus respectivos métodos.

A intensidade de um acoplamento está diretamente relacionado como a facilidade ou dificuldade de se avaliar o impacto de uma alteração em um módulo nos demais módulos a ele conectados.

Do ponto de vista do custo de manutenção, o ideal é que os módulos fossem todos desacoplados, mas isto não é realista, pois é a conexão que torna os módulos úteis. O que é necessário fazer é manter a intensidade dessas conexões com o menor valor possível, usando, em cada caso, o tipo mais adequado de acoplamento.

Glenford Myers classificou os tipos de acoplamento possíveis entre dois módulos, em ordem decrescente de intensidade, como: por Conteúdo, por Dado Comum, por Inclusão, por Dado Externo, por Controle, por Referência ou por Informação.

Acoplamento por conteúdo

O acoplamento por conteúdo ocorre quando um módulo tem acesso direto e sub-reptício a campos de outro módulo. Isso permite que um módulo modifique valores de campos de um outro módulo de uma forma que o projetista do sistema pode não ter previsto.

Por exemplo, define-se uma pilha como um tipo abstrato de dados, mas acidentalmente deixa-se um caminho de acesso direto à

sua representação, o qual pode ser usado sub-repticiamente, possivelmente quebrando o invariante das estruturas. Em linguagens como C++, que trabalham com apontadores, situações como essa são fáceis de ocorrer. Em Java, esse tipo de acoplamento pode ser evitado sempre declarando todos os campos como privados.

Acoplamento por dado comum

Módulos distintos têm acesso direto, previsto pelo projetista, a componentes da estrutura de objetos comuns, declarados globalmente, e que possuem uma interpretação rígida.

O problema com esse tipo de acoplamento é que modificações na estrutura do objeto comum podem ter impacto, difícil de ser percebido, em todos os módulos envolvidos, e a localização desse impacto pode ser trabalhosa.

Um exemplo desse tipo de acoplamento é o existente entre os módulos `A`, `B` e `C`, definidos a seguir, onde o módulo `C` exporta o vetor `v`, cujas posições têm interpretações muito específicas, por exemplo, que `v[3]` seja a idade de um trabalhador e `v[4]`, o seu tempo de serviço, e essa interpretação do conteúdo de `v` é usada pelos módulos `A` e `B`.

- Módulo `C`:

```
1 public class C {
2     public static int[] v = new int[20];
3 }
```

- Módulo `A`:

```
1 public class A {
2     ...
3     void f() { ... C.v[3] = z; ... }
4 }
```

- Módulo `B`:

```
1 public class B {
2     ...
3     void g() { ... y = C.v[4]; ... }
4 }
```

Qualquer alteração no módulo `C` que mude a interpretação do conteúdo de `v`, por exemplo, a inserção de uma informação nova em `v`, poderá ter um efeito desastroso nos módulos-clientes de `C`.

Acoplamento por inclusão

Este tipo de acoplamento é similar ao de dado comum e pode ocorrer em linguagens como C e C++, que permitem inclusão de código-fonte de um módulo em outro.

Nessa situação, todos os módulos que fazem a inclusão estão acoplados entre si e também estão com o do texto incluído, conforme mostra o seguinte exemplo, onde `B`, `C` e `D` representam módulos, e o arquivo `B.h` contém a definição da classe `A`:

```
1  #include B.h;
2  public class C {
3     private A a = new (...);
4     ...... a.f(..); ...
5  }
6
7  #include B.h;
8  public class D {
9     private A a = new (...);
10    ...... a.f(..); ...
11 }
```

Acoplamento por elemento externo

Este tipo de acoplamento é similar ao *por dado comum*, porém é de menor intensidade, pois, neste caso, módulos têm acesso a componentes individualmente identificados de uma área comum.

E esse acesso é feito independentemente da presença dos demais elementos da área comum, de forma que qualquer modificação no objeto comum não afeta necessariamente todos os módulos a ele interconectados, e a identificação de seu impacto é facilmente determinável. No exemplo a seguir, o módulo `M` define uma área de dados comum, da qual os campos `w` e `z` podem ser acessados, no caso, pelos módulos `A` e `B`.

Módulo `M`:

```
1 public class M {
2    public int w;
3    public float z;
4    public int q() {...}
5 }
```

Módulo `A`:

```
1 public class A {
2    M t; ... t.z = ...; ... t.w ...
3 }
```

Módulo `B`:

```
1 public class B {
2    M r; ... k = r.q(); ... r.w ...
3 }
```

Diferentemente do acoplamento por dado comum, novos campos podem ser inseridos na classe `M`, causando pouco impacto nos módulos `A` e `B`.

Acoplamento por controle

Esse é o acoplamento resultante da conexão de um módulo com um outro de coesão interna lógica. O módulo-cliente deve conhecer a implementação dos métodos de outro para passar-lhes parâmetros para controlar seu fluxo de execução, como no seguinte código:

```
1 public rotina(String comando) {
2    if (comando.equals("desenhe circulo") desenheCirculo();
3    else desenheRetangulo();
4 }
```

Esse tipo de acoplamento obriga que o método-servidor tenha que ser modificado sempre que surgir um novo comando.

Acoplamento por referência

Nesse tipo de acoplamento, a comunicação entre módulos é feita unicamente via chamada de métodos, mas usam-se parâmetros por referência, facilitando que os componentes acoplados modifiquem ou acessem dados um do outro, como ocorre com módulos `A` e `B`:

Módulo `A`:

```
1 public class A {
2    ...
3    public void muda(T x) {
4       x.valor = 100;
5    }
6 }
```

Módulo `B`:

```
1 public class B {
2    private A a;
3    private T c  = new T();
4    public void f() {
5       a.muda(c);
6    }
7 }
```

Módulo `T`:

```
1 public class T {
2    public int valor;
3 }
```

Acoplamento por informação

Acoplamento por informação ocorre quando a comunicação entre módulos é via chamadas de métodos com parâmetros passados por valor. Essa forma de comunicação garante independência de funcionamento dos módulos envolvidos, facilitando o processo de manutenção.

No programa a seguir, o método **`muda`** da classe `A` não tem como alterar quaisquer campos de `B`:

Módulo-servidor `A`:

```
1 public class A {
2    ...
3    public void muda(int x) {
4       x += 100;  .. x ...
5    }
6 }
```

Módulo-cliente `B` tem o seguinte código:

```
1 public class B {
2    private A a  = new A ();
3    private int c = 10;
4    public void f() {
5       a.muda(c);
6    }
7 }
```

Este é o tipo de acoplamento recomendado para se atingir elevado grau de modularidade e manter o custo de manutenção mais baixo.

Conclusão

A orientação por objetos tem foco na identificação e implementação dos objetos que compõem um sistema, e, nesse contexto, tipos abstratos de dados são uma ferramenta extremamente útil e indispensável.

Tipos abstratos de dados permitem a implementação dos seguintes clássicos conceitos da Engenharia de Software:

- Abstração: foco no relevante enquanto ignora detalhes de implementação.
- Modularidade: divisão de um problema em subproblemas independentes.
- Encapsulação e controle de visibilidade: *boas cercas fazem bons vizinhos*.
- Separação de interesses: Conceitos relacionados devem ser agrupados em módulos, evitando seu espalhamento ao longo do código.

Modularizar é a atividade central no desenvolvimento de sistemas de grande porte, e módulos devem ser autônomos e indepen-

dentes de forma a minimizar sua conectividade e reduzir os custos de manutenção.

Exercícios

1. Descreva a abstração *caixa de banco*, de forma independente de sua implementação, seja eletrônica ou humana.
2. Avalie importância de se impedir o acesso a representação de um tipo de dados.
3. Por que não é possível implementar tipos abstratos de dados na linguagem **C**?

Notas bibliográficas

Fortran é a primeira linguagem de programação de alto nível. Sua primeira versão surgiu em 1954. Outras versões, Fortran IV, Fortran 77 e Fortran 90, que incorporam alguns avanços da Área, ficaram mais conhecidas [24, 55].

A linguagem PL/I [26] foi criada pela IBM para ser a linguagem número 1, com todos os recursos necessários a qualquer aplicação. Essa abordagem terminou sendo condenada diante da preferência por linguagens dotadas de mecanismos apropriados para ser estendidas conforme a necessidade.

O conceito de tipos abstratos dados é devido a Barbara Liskov [32], que o incorporou no projeto da linguagem CLU [33].

Capítulo 6

Camadas de Sofware

Propagação do impacto devido a mudanças realizadas em um módulo para os demais módulos de um sistema é uma questão determinante do custo do processo de manutenção. Quanto maior o número de módulos afetados pela propagação de uma alteração no código mais custoso é esse processo.

Encapsulação e controle de visibilidade são recursos poderosos para reduzir o impacto de alterações no código de um módulo, mas quando essas alterações atingem a sua interface, a propagação de seu impacto para módulos que a utilizam é inevitável, provocando a necessidade de identificação desses módulos para verificar se foram ou não afetados pelas mudanças e tomar as providências necessárias.

Esse processo requer a identificação de todos os módulos do sistema que dependem direta ou indiretamente do módulo que foi alterado. A princípio todos os módulos de um sistema em manutenção deveriam ser inspecionados para ver se eles foram impactados pela interface que foi alterada.

Essa busca exaustiva em grandes sistemas, que sem dúvida é muito custosa, pode ser evitada agrupando os módulos conforme a direção do fluxo de suas mensagens. Com isso, o conjunto de módulos sujeitos a impactos de determinadas alterações pode ser de tamanho reduzido, facilitando o trabalho de manutenção. Con-

juntos de módulos com essas características são denominados *camadas de software*. Assim, camadas de software são uma forma de organização de módulos de uma aplicação visando melhor distribuir as suas funcionalidades e dependências.

Um sistema orientado por objetos em execução é um conjunto de objetos que trocam mensagens entre si, com o objetivo de solicitar a execução de serviços. Essas mensagens são essencialmente pedidos de serviços implementados via chamadas de métodos. Para simplificar os conceitos, apenas o pedido de um serviço é tratado como uma mensagem. As possíveis respostas a esses pedidos de serviço não são consideradas mensagens neste contexto.

O conceito de camada de software está vinculado diretamente a ideia de que a direção do fluxo dessas mensagens entre camadas deve ser uniderecional e pré-definido, e que somente pode haver fluxo de mensagens entre classes de certos tipos de camadas. Essas restrições reduzem a cardinalidade dos conjuntos de módulos que podem ser afetados por alterações realizadas em módulos de outros conjuntos, facilitando a determinação do impacto dessas alterações. Essa disciplina, que é imposta no fluxo de mensagens, contribui para aumentar os graus de *extensibilidade*, *manutenabilidade* e *portabilidade* do sistema. Por exemplo, se as mensagens, i.e., pedidos de serviços, fluem somente de uma camada A para uma outra B, então as classes de A não têm dependência alguma das classes de B, e, portanto, pode-se modificar as classes da camada A sem afetar classes da camada B, com diretos benefícios para os três fatores de qualidade citados.

Os modelos podem ser de três camadas, de quatro camadas ou de n camadas. Do ponto de vista do modelo, o número de camadas não é o ponto mais relevante: o que importa é a disciplina imposta na direção do fluxo de mensagem.

6.1 Modelo de quatro camadas

Em geral, as aplicações consistem em um conjunto de classes que formam a sua interface com o usuário, um outro conjunto de classes que implementam as regras de negócios, que são as classes relacionadas ao domínio da aplicação, e o conjunto das classes que estabelecem a conexão com o sistema de gerência do banco de dados. Há também um quarto conjunto de classes, que são as chamadas classes de sistema.

O modelo de camadas preconiza que conjuntos de classes, como esses acima identificados, sejam organizados conforme ilustrado na Fig. 6.1, onde a seta (→) indica a direção do fluxo de mensagens. Nesse modelo, há quatro camadas formadas segundo o seguinte critério:

- *Camada de Interface* com as classes que tratam da interface com o usuário;
- *Camada de Negócios* com as classes de negócios ;
- *Camada de Persistência* que reúne as classes que implementam a conexão com o sistema de banco de dados;
- *Camada de Sistema* com bibliotecas de uso geral e possivelmente dependente do ambiente de programação.

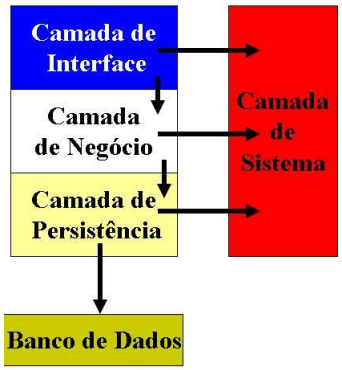

Figura 6.1 Modelo de Quatro Camadas

O fluxo de mensagens é tal que as classes da camada de interface somente enviam mensagens para as camadas de negócios e de sistemas. Por sua vez, classes da camada de negócio somente podem enviar mensagens para as camadas de persistência e de sistemas. O fluxo de mensagem dentro de uma camada é livre. Outras trocas de mensagens, além dessas descritas acima, violam o modelo.

Observe que o ponto fundamental do conceito de camadas é a descoberta de que as mensagens são unidirecionais, e, portanto, as dependências entre camadas também o são. Por exemplo, a troca da interface do usuário de uma aplicação, o que normalmente acontece, quando se transporta uma aplicação de uma plataforma a outra, não tem efeito nas classes da camada de negócios e nem nas da camada de persistência. Por outro lado, a troca do sistema de banco de dados afeta a implementação das classes de persistências, que podem ser atualizadas sem afetar a camada de negócios e muito menos a camada de interface. Se mensagens fluíssem em ambas as direções obviamente a dependência entre as classes da aplicação seria muito maior, encarecendo o processo de manutenção.

Camada de interface

A camada de interface contém um conjunto de classes que implementam a interação dos usuários com a aplicação, tipicamente, via o uso de uma interface gráfica de usuário (GUI).

À luz do modelo de camadas, todas as ações da aplicação devem ser iniciadas a partir da sua interface com o usuário por meio do envio de mensagens a objetos de classes que implementam as regras de negócio. Consequentemente todas as mensagens automaticamente fluem da camada de interface para a camada de negócio, e nunca no sentido contrário, e as mudanças na interface não afetam as classes da camada de negócio.

As classes de interface são construídas a partir de bibliotecas gráficas e classes utilitárias da camada de sistema, que oferecem os serviços solicitados. Nesse caso, o fluxo natural de mensagens também tem o sentido da camada de interface para a camada de sistema.

Ressalva-se o caso de bibliotecas que usam a técnica conhecida como rechamada (*callback*). Nesse caso, as mensagens enviadas aos objetos passados como parâmetro a métodos da biblioteca têm a direção voltada para camada de interface. Esse tipo de exceção tem sido aceita no modelo, mas certamente cria uma dependência indesejável em relação à plataforma, impactando negativamente importantes fatores de qualidade de software.

Ainda consistente com o modelo, em alguns casos, é permitido o fluxo de mensagens diretamente entre a camada de interface e a de persistência. A vantagem dessa abordagem é que se ganha flexibilidade e tempo de execução, mas pode-se perder extensibilidade.

Camada de negócios

A camada de negócio encapsula um conjunto de classes do domínio da aplicação ou do negócio. As classes de negócio são as classes identificadas durante a fase de análise, que encapsulam a funcionalidade da aplicação, sem se preocupar com interface de usuário, administração de dados e detalhes de sistema. As classes de negócio, em geral, não são afetadas quando portadas para outro ambiente de execução.

O fluxo de mensagens é no sentido da camada de interface para camada de negócio e desta para a camada de persistência. Assim como a camada de interface, pode haver o fluxo bidirecional de mensagens entre a camada de negócio e a de sistema.

Camada de persistência

A camada de persistência encapsula um conjunto de classes que proveem o acesso ao banco de dados da aplicação. Essa camada não é o Banco de Dados, mas apenas uma interface que empacota o acesso ao banco de dados. Ela torna a aplicação independente do sistema de gerência do banco de dados e de suas versões. As classes da camada de persistência em geral devem ser modificadas quando o sistema de arquivos ou banco de dados for modificado.

Observe que a camada de persistência permite que as classes de negócio tenha uma visão puramente orientada por objetos do banco de dados, mesmo quando o próprio banco de dados seja, por exemplo, apenas relacional. Os complexos processos de recuperação e gravação de objetos são abstraídos por meio das classes dessa camada.

Camada de sistema

A camada de sistemas encapsula serviços relacionados ao sistema operacional e às bibliotecas de uso geral e de vários tipos oferecidas pelo ambiente de execução. Exemplos são bibliotecas de tipos de dados universais, componentes que fornecem acesso a recursos de hardware, do Sistema Operacional ou a dispositivos de comunicação. Aí também estão bibliotecas de componentes de sistemas gráficos cuja implementação seja dependente de plataforma.

Em geral, a camada de sistema recebe mensagens de outras camadas e pode enviar mensagens de volta via o mecanismo de rechamada.

Classes da camada de sistema podem ser desenvolvidas sem quaisquer informações dos seus usuários e são aquelas que geralmente precisam ser modificadas se o ambiente de execução ou sis-

tema operacional for alterado.

6.2 Organização de camadas

Java não oferece recursos linguísticos para controlar e verificar a direção do fluxo de mensagens entre camadas. Essa tarefa é responsabilidade do programador, que deve usar os recursos disponíveis para produzir um conjunto de módulos bem organizados.

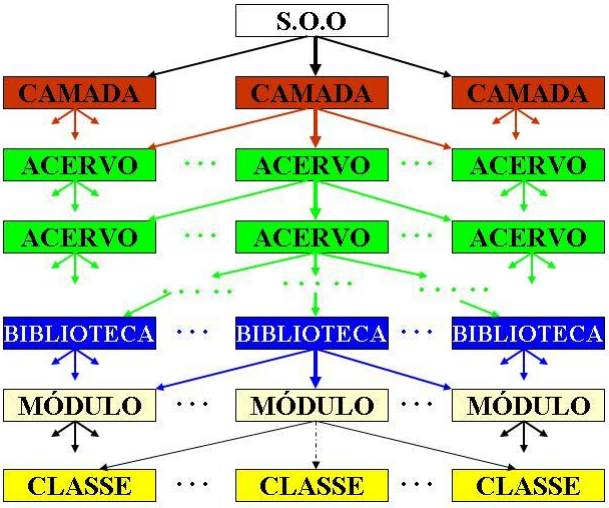

Figura 6.2 Organização de um Sistema OO

Os recursos disponíveis em Java para organizar camadas são o conceito de pacotes, que estão associados a diretórios do sistema de arquivo do ambiente de programação.

Considerando que um aplicativo Java é formado por um conjunto de arquivos com extensão `.java`, sendo que cada arquivo possui declarações de uma ou mais classes, para estruturar as diversas camadas, deve-se distribuir esses conjuntos de arquivos do aplicativo em uma hierarquia de diretórios.

Os arquivos do programa-fonte, que são os módulos da aplicação, devem ser particionados em pacotes Java, formando bibliotecas, as quais podem, por sua vez, ser agrupadas em diretórios para

constituir coleções de bibliotecas ou *acervos*. Acervos relacionados podem então ser recursivamente combinados para montar novas hierarquias, correspondendo às funcionalidades de interface, de negócio ou de persistência, conforme ilustrado na Fig. 6.2.

Uma camada, na prática, é um conjunto de classes que estão organizadas em arquivos, que são módulos. Os módulos estão organizados em diretórios, que são bibliotecas, as quais estão organizadas em diretórios, que são acervos, i.e., conjuntos de bibliotecas, e esses formam uma camada.

Assim, uma camada de software em Java é uma árvore de acervos, cujas folhas são bibliotecas, as quais contêm módulos. Pode-se dizer que uma camada é um diretório, que contém diretórios que são acervos de bibliotecas.

Conclusão

Propagação do impacto devido a mudanças realizadas em um módulo para os demais módulos de um sistema é uma questão determinante do custo do processo de manutenção. Quanto maior o número de módulos afetados pela propagação de uma alteração no código mais custoso é esse processo. O conceito de camadas, como um método de disciplinar o fluxo de mensagens em módulos, permite reduzir os custos de localizar o impacto das mudanças ocorridas em um módulo, reduzindo os custos de manutenção.

Exercícios

1. Avalie à luz do modelo de camadas o caso de classes pertencentes à camada de negócio estenderem classes da camada de interface e vice-versa.

Notas Bibliográficas

O Capítulo 4 do livro de Ambler Scott contém uma boa descrição do modelo de quatro camadas [51].

Capítulo 7

Programação por Contrato

Encapsulação, controle de visibilidade, separação de interesses e abstração de dados são a base para o desenvolvimento de programas modulares de boa qualidade.

Na programação orientada por objetos, tudo começa com a identificação dos objetos a ser implementados e que devem ser especificados com precisão, por exemplo, via especificações algébricas. Assim, programar esses objetos significa traduzir a sua especificação nos termos de definição de tipos abstratos de dados.

Classes de Java oferecem importantes mecanismos para implementação de tipos abstratos de dados, pois permitem a concretização dos preceitos do exercício efetivo da abstração, como encapsulação, controle de visibilidade e separação de interesses.

Entretanto, acima desses preceitos, há a questão da correção da implementação, que é ponto central na especificação de tipos abstratos de dados, a qual deve sempre enfatizar a definição de pré-condições e pós-condições para as operações do tipo.

Essas condições podem ser interpretados como o contrato de cada operação com seu usuário, estabelecendo as regras de chamadas e a garantia dos resultados produzidos.

Deve-se portanto incorporar no código da implementação das operações de um tipo abstrato as suas pré-condições e pós-condições, bem como os invariantes de sua representação que forem pertinen-

tes ao tipo, todos presentes na descrição do tipo a ser implementado, como ilustra a seguinte especificação algébrica:

```
type PilhaDeInt
sorts:
   Integer, Boolean
operations:
  vazia      : PilhaDeInt --> Boolean
  crie       : Integer --> PilhaDeInt
  empilhe    : (Integer,PilhaDeInt) --> PilhaDeInt
  desempilhe : PilhaDeInt --> PilhaDeInt
  topo       : PilhaDeInt --> Integer
preconditions:
  pre crie(n : Integer)              : ( n > 0)
  pre desempilhe(s : PilhaDeInt : (not vazia(s))
  pre topo(s : PilhaDeInt)      : (not vazia(s))
axioms:
  for all n : Integer, x : Integer, s : PilhaDeInt:
     vazia(crie(n)) and (n > 0)
     not vazia(empilhe(x,s) )
     topo(empilhe(x,s)) = x
     desempilhe(empilhe(x,s)) = s
end
```

Entende-se por *pré-condição* os requisitos a ser satisfeitos pelos valores dos parâmetros de cada operação e o estado corrente do objeto da classe que implementa o tipo no momento de sua chamada, e por *pós-condição*, os que devem ser satisfeitos pelo estado do objeto no fim da operação e o valor por ela retornado.

No exemplo acima, a pré-condição de `desempilhe(s)` é definida por `not vazia(s)`, e a pós-condição é que o valor desempilhado seja o que foi empilhado por último.

7.1 Direitos e obrigações

O relacionamento entre uma classe e seus clientes é uma relação de direitos e obrigações, e essa relação está vinculada a ideia de contrato. Há cláusulas de pré-condições e de pós-condições que devem ser satisfeitas por todo método exportado pela classe.

O contrato de uma classe é a coleção dos contratos das operações por ela exportadas. A semântica dos métodos exportados, i.e., suas pré-condições e pós-condições, faz parte do contrato da classe, que é o que o seu projetista promete que os seus métodos farão. Métodos privados da classe não fazem parte do contrato.

A regra básica do contrato é que o usuário deve comprometer-se a chamar o método com sua pré-condição satisfeita e, em contrapartida, o método chamado garante devolver um resultado que satisfaz a sua pós-condição e deixar o objeto associado em um estado coerente com seu invariante, ou então deve informar explicitamente seu insucesso. Isso exige que todo método-servidor siga as seguintes regras de conduta:

- Todo método deve iniciar-se verificando se suas pré-condições foram integralmente satisfeitas.
- Todo método deve lançar a exceção pertinente se alguma de suas pré-condições não for atendida.
- Uma chamada a um método de um objeto tem sucesso se o método chamado terminar sua execução em um estado que satisfaz o seu contrato.
- Uma chamada de um método fracassa quando ele não consegue cumprir seu contrato.
- Todo método ao fracassar deve levantar uma exceção indicadora do erro e transmiti-la ao seu método-cliente.

Em resumo, pré-condições definem as condições sob as quais

chamadas a um método são legítimas, e pós-condições definem as condições que o método deve garantir no seu retorno. Em consequência, o retorno normal de um método significa cumprimento do contrato, e falhas no seu cumprimento devem sempre causar o lançamento de exceções.

Princípio do Contrato

Todo método ou fracassa, relatando claramente esse fato, ou cumpre seu contrato.

A implementação desse princípio requer que o primeiro ato de um método-servidor seja o teste de sua pré-condição e o último seja o da pós-condição, como sugere o seguinte esquema de codificação:

```
1  public void g(...) throws Pré, Pós {
2     if (!pré-condição) throw new Pré(...);
3     "implementação das regras de negócio de g";
4     if (!pós-condição) {
5         "restabelece o invariante do objeto";
6         throw new Pós(...);
7     }
8  }
9  public class Pré extends Exception { ... }
10 public class Pós extends Exception { ... }
```

onde `Pré` e `Pós` são as exceções relacionadas com o contrato de `g`, e o retorno normal de um método significa cumprimento do contrato.

7.2 Exceção disciplinada

Se um método-cliente `f` chama o método-servidor `g`, e este recusa-se a cumprir sua parte no contrato porque sua pré-condição não foi atendida, levantando uma exceção do tipo `Pré`, ou declara-se incapaz de produzir a pós-condição especificada, e sinaliza sua

decisão levantando uma exceção como `Pós`, então `f` tem três opções de caminhos a seguir:

- método `f` captura a exceção levantada e a trata convenientemente em uma de suas cláusulas `catch` e decide como prosseguir no fluxo de execução diante da falha detectada, inclusive fazendo nova tentativa de execução do método-servidor;
- método `f` captura a exceção levantada, mas não a trata, preferindo delegar seu tratamento a uma outra parte do programa, via o lançamento de uma outra exceção em seu `catch`;
- método `f` nada faz a respeito. Nesse caso, ele está obrigado a listar a exceção recebida em sua cláusula `throws`, de forma a delegar o seu tratamento ao módulo-cliente de f.

Consistentemente, falhas no cumprimento do contrato devem sempre causar lançamento de exceções, e as regras do negócio de uma classe não devem fazer parte de código de tratamento de exceção, que deve estar sempre focado na recuperação do erro para continuidade da execução.

Tratamento de exceções, como o próprio nome sugere, tem caráter excepcional e deve ser feito obedecendo aos princípios da *Simplicidade do Tratamento* e do *Cumprimento do Contrato*.

A *Simplicidade do Tratamento* estabelece que todo processamento feito em resposta a uma exceção deve:

- ser o mais simples possível;
- ter como único objetivo restabelecer as condições seguras de execução de forma a permitir tentativas de re-execução do método que falhou;
- ser executado sem levantar novas exceções, a não ser para delegar o processamento da falha a outro tratador de exceções.

O *Cumprimento do Contrato* estabelece que somente há duas

respostas legítimas para uma exceção que ocorre durante a execução de um método de um objeto:

- Tentativa de Recuperação:
  mudam-se as condições que levaram à situação de exceção e reinicia-se a execução do método, geralmente no estado corrente do objeto.
- Confissão da Falha:
  restabelece-se o invariante do estado do objeto, considera-se terminada a execução da rotina, mesmo com o contrato não-satisfeito, e relata-se obrigatoriamente a falha do método ao seu cliente.

7.3 Subcontratação

Declarações de subclasses são um processo de estender os serviços de uma classe, criando assim subtipos. A presença de novos serviços na subclasse normalmente requer uma extensão do contrato herdado da superclasse. Nesse processo, é possível redefinir todo o contrato, mas não se pode violar o contrato herdado da superclasse.

Essa liberdade de redefinição do contrato está delimitada por duas regras fundamentais: enfraquecimento das pré-condições e fortalecimento das pós-condições dos métodos redefinidos. Entende-se, nesse contexto, que uma condição mais fraca é uma menos restritiva, e uma condição mais forte é uma mais restritiva.

Note que o fortalecimento de pré-condições tem impacto negativo nos antigos módulos-cliente, que podem passar a ter suas chamadas recusadas. Um exemplo dessa situação seria substituição de um caixa-humano, que autoriza retiradas até R$50,00 sem verificar existência de fundos, por um caixa-eletrônico mais rigoroso, que não autoriza esse tipo de saque. Claramente, nesse caso, há

uma violação de contrato, porque fortaleceu-se incorretamente a pré-condição do método que faz retiradas.

Por outro lado, pós-condições fortalecidas permitem que os resultados retornados possam continuar a ser usados no mesmo contexto de antes da definição da subclasse. Por exemplo, se um método garante que sempre retorna um inteiro no intervalo de 10 a 19, e, assim, o valor retornado pode ser usado, sem necessidade de novos testes, para indexar arranjos cujo tamanho seja maior ou igual a 20. E isso continua valendo, se o método redefinido na subclasse passar a retornar valores no intervalo de 12 a 18. Problemas podem surgir se o método redefinido passar a retornar valores entre 1 e 100, violando o contrato herdado pelo enfraquecimento de sua pós-condição.

Conclusão

O atributo correção deve prevalecer sempre sobre eficiência, porque somente faz sentido ter rapidez na produção de respostas corretas. Programação por contrato segue essa ideia pela introdução de código defensivo em locais apropriados do programa. Isso pode, em princípio, encarecer a execução, mas certamente aumenta a segurança de que o programa está executando corretamente e reduz o custo da depuração do código.

Exercícios

1. Na estrutura proposta neste capítulo, os testes de pré-condições e de pós-condições foram incorporados aos corpos dos métodos-servidor. Uma alternativa seria usar comandos `assert` de Java para esse fim. Compare essas duas soluções.

2. Avalie a possibilidade de se ter diretivas de compilação que

permitam desligar os testes da programação por contrato depois que o sistema é declarado concluído. Isso é uma boa ideia? Como isso se compara com a decisão de deixar os botes salva-vidas de um transatlântico no porto para reduzir o peso do navio durante a viagem?

Notas bibliográficas

Os conceitos de programação por contrato aqui apresentados são derivados dos definidos por Bertand Meyer, sob o nome *Design by Contract*, que está muito bem apresentado no livro *Object Oriented Software Construction* [41].

Capítulo 8

Reúso de Componentes

Reúso é uma prática altamente difundida nos diversos ramos da atividade humana e industrial. Por exemplo, quando projeta-se um novo carro, a maior parte dos componentes não são reinventados, mas apenas adaptados, se necessário, e combinados com os que implementam os avanços tecnológicos ou estéticos desejados.

Na área de software, reúso manifesta-se de diversos formas e com muita flexibilidade. Segundo Bertrand Meyer, reúso pode ocorrer nos seguintes níveis:

- Reúso de Programa-Fonte: a leitura de um programa-fonte permite o estudo de técnicas de implementação e fazer cópias e adaptação das soluções encontradas.
- Reúso de Pessoal: a rotatividade de especialistas entre as equipes de uma empresa de desenvolvimento de software permite transferir experiência de um setor para outro.
- Reúso de Projetos: a estrutura de um projeto bem sucedido é um bom guia para o desenvolvimento de projetos similares.
- Reúso de Programa Objeto: módulos que implementam serviços, os quais foram devidamente padronizados e encapsulados, são de fácil reúso. Isso ocorre com frequência em ambientes de desenvolvimento de software para nichos bastante específicos em que subproblemas repetitivos podem ser claramente identificados e modularizados.

- Reúso de Componentes de Software: componentes são usados diretamente em novas aplicações ou são adaptados para novas soluções de software com objetivo de reduzir custos e aumentar confiabilidade.

Réuso de componentes de software é o foco deste capítulo e, considerando o nível de adaptação que esses componentes são submetidos, manifesta-se de pelo menos três formas: reúso de componente aberto, reúso de componente original e reúso de componente adaptável.

8.1 Reúso de componente aberto

O *reúso de componente aberto* está relacionado com a ideia de que programar é uma atividade de repetição com pequenas modificações. Por exemplo, frequentemente, tem-se que ordenar e pesquisar em tabelas; ler, escrever, comparar e caminhar em estruturas de dados ou sincronizar processos, entre outras atividades.

Uma forma de criar novos componentes via esse tipo de reúso é obter uma cópia do fonte de um software similar ao do problema a ser resolvido e **realizar as modificações** que se fizerem necessárias para adaptá-lo ao novo contexto.

A literatura descreve muitos algoritmos que podem servir de modelo para codificar soluções para problemas específicos. Um livro de algoritmos e estruturas de dados é uma excelente fonte de exemplos de códigos a ser adaptados e incorporados em novas aplicações.

A vantagem desse tipo de reúso é que o padrão de programação que já vem definido, e a desvantagem é a necessidade de depuração e teste das modificações ou adaptações introduzidas.

8.2 Reúso de componente original

O *reúso de componente original* é focado no objetivo de reduzir custos de desenvolvimento e aumentar o nível de correção das aplicações via o uso, **sem qualquer modificação**, de componentes de software previamente testados e demonstrados corretos. Esses componentes são incorporados no sistema na forma de programa-fonte ou de programa-compilado.

Esse tipo de reúso é altamente praticado, e o exemplo mais popular são as bibliotecas presentes em quase todo ambiente de programação, que disponibilizam módulos na forma de funções, classes e pacotes, possivelmente parametrizados e genéricos, que podem ser usados diretamente sem necessidade de reinventá-los a partir do zero.

Para atingir alto grau de reúso, modularidade é a chave, mas outros recursos linguísticos também são necessários, porque as abstrações clássicas, como funções e procedimentos, são insuficientes para suportar um nível adequado de reusabilidade.

Observe que o Princípio da Unidade Linguística definido no Capítulo 4, que associa uma construção apropriada para realizar abstração a comandos e expressões (subrotina), a uma declaração de variáveis (objetos e operações), a família de declarações de variáveis (classes definindo tipos com suas operações) e a família de definições de tipos (classes genéricas + sobrecarga), é muito importante para produção de bons módulos, mas reúso demanda mais recursos linguísticos.

O componente a ser reusado deve ser incorporado no novo sistema em sua totalidade, i.e., com todas as operações e tipos a ele associados. Um exemplo clássico é o do reúso da rotina `pesquise()`, que implica no reúso de rotinas associadas para inserir ou apagar elemento da mesma tabela. Isso é resolvido pelo mecanismo de

classes, que permite encapsular toda a definição de um tipo em uma única construção, como ilustra o tipo `Tabela`:

```
1 class Tabela {
2     ...
3     boolean pesquise(...) { ... }
4     void insira (...) { ... }
5     void remova (...) { ... }
6 }
```

Há também o caso de o reúso de um componente original implicar na reutilização de outros componentes relacionados, que, por isso, precisam ser agrupados em uma única unidade linguística, e.g., em um pacote Java, pois, do contrário, o reúso de cada classe demandaria uma investigação do seu código para identificar e incluir os serviços por ela utilizados.

O espectro de aplicação de um componente original é valorizado pela possibilidade de variação nos tipos por ele manipulados, a qual pode ser obtida pelo uso de métodos e classes genéricas.

Um reúso importante é o que ocorre pelo mecanismo de composição de objetos, o qual é destinado à montagem da representação de novos objetos. Trata-se de reúso *caixa preta*, no qual, normalmente, os objetos que participam da composição são somente acessíveis via suas interfaces públicas.

Tratamento de exceção é também um mecanismo de reúso importante, porque aumenta a reusabilidade de um componente original, retirando do módulo servidor o tratamento do erro, tornando-o mais genérico e mais independente, portanto mais reusável, pois cada cliente pode definir seu próprio tratador da exceção. A classe `Pilha1` a seguir é mais reusável que `Pilha2`, porque a operação `empilhe` do primeiro delega aos seus módulos clientes o tratamento do erro encontrado, enquanto que o de `Pilha2` responde à violação de seu contrato de forma muito particular.

```
1 class Pilha1 {
2    private int topo = 0;
3    ...
4    public void empilhe(int v) throws ExMax {
5       if (topo == last) throw new ExMax();
6       item[++top] = v;
7    }
8    ...
9 }
```

```
1 class Pilha2 {
2    private int top0 = 0;
3    ...
4    public void empilhe(int v) {
5       if (topo == last) System.out.println("Erro!");
6       else item[++top] = v;
7    }
8    ...
9 }
```

8.3 Reúso de componente adaptável

O terceiro tipo de reúso, o *reúso de componente adaptável*, trata da incorporação em um novo sistema do componente a ser reusado **sem qualquer modificação** em seu código, e do uso de recursos de linguagens orientadas por objetos para ajustar seu comportamento ao novo contexto, sem modificá-lo diretamente, e, assim, preservando sua funcionalidade.

O principal mecanismo linguístico de suporte a este tipo de reúso é o de especialização de classes, por meio do qual pode-se construir subclasses para realizar as adaptações necessárias do componente a ser reusado sem alterá-lo diretamente. As adaptações realizadas

são extensões da representação interna dos objetos envolvidos e as consequentes redefinições das operações associadas.

Herança de classe, que permite definir implementação de uma classe em termos de outra, gera um reúso *caixa branca*, no qual partes internas da superclasse são visíveis nas subclasses, possibilitando suas redefinições. Entretanto, esse processo pode violar encapsulação, pois os elementos herdados, e que devem ser adaptados, devem ter nas subclasses visibilidade no mínimo `protected` e, consequentemente, mudanças na superclasse podem ter impacto em suas subclasses. Sua principal vantagem é a facilidade de adaptar componentes em tempo de compilação.

O processo de reúso de componente adaptável também implica em escrever código para realizar variações no comportamento dos componentes a ser reusados, sem modificar seu código original, de forma ajustá-los ao seu novo contexto de uso. Para isso, são usados recursos linguísticos, tais como: variação na estrutura de dados e algoritmo, independência da implementação e fatoração de elementos comuns.

8.4 Adaptação de dados e algoritmos

No processo de reúso de um componente adaptável, não se deve modificar o seu código, porque, se isso ocorrer, ter-se-ia na verdade reúso de componente aberto. Como visto na seção anterior, a especialização do componente via subclasses é a solução, a qual permite até mudar a representação do tipo de dados manipulados pelo componente. Por exemplo, dado um componente que implementa uma tabela sequencial, pode-se desejar alterar sua estrutura interna para listas encadeadas, arranjos, árvores binárias ou B-tree, e essa troca de representação deve ocorrer sem modificação direta

do código do componente sendo reusado, como exemplifica o reúso da classe `Tabela` pela classe `OutraTabela`, definidas a seguir:

```
1  class Elemento {public int val;}
2  class Corrente {
3     public Elemento corrente() {...}
4     public void próximo() {...}
5     ...
6  }
7  class Tabela {
8     "Aqui entra a definição da E.D. da tabela"
9     protected Corrente iniciePesquisa() {...}
10    protected boolean acabou() {...}
11    protected boolean encontrou(Corrente p,Elemento x){...}
12    public boolean pesquise(Elemento x) {
13       Corrente p;
14       p = iniciePesquisa();
15       while (!acabou() && !encontrou(p, x)) p.próximo();
16       return (!acabou());
17    }
18     ...
19    public void insere(Elemento x) {...}
20 }
```

Alteração na representação do componente tem impacto direto nas operações associadas. Os recursos linguísticos usados para esse fim são redefinição de membros de dados, redefinição de membros-função e criação de tipos dinâmicos, facilmente implementáveis por meio de definição de subclasses do componente em reúso.

Suponha que no reúso da classe `Tabela`, deseja-se trocar sua representação. Para isso, definem-se as subclasses `OutraTabela`, `OutroElemento` e `OutroCorrente`, nas quais as novas estruturas de dados são definidas e, por consequência, pode-se ter que redefinir as operações `insere`, `pesquise` e outras associadas, como `acabou`, `encontrou` e `iniciePesquisa`.

A classe **OutraTabela** a seguir implementa as redefinições que se fizeram necessárias nesse caso:

```
class OutraTabela extends Tabela {
   "Aqui entra definição da E.D. da nova tabela"
   protected Corrente iniciePesquisa() {...}
   protected boolean acabou() {...}
   protected boolean encontrou(Corrente p,Elemento x){...}
   public void insere(Elemento x) {...}
}
class OutroElemento extends Elemento {public String val;}
class OutroCorrente extends Corrente {...}
```

Observe que a operação **pesquise** declarada na classe **Tabela** não teve que ser redefinida na subclasse, pois sua adaptação ocorreu pelo processo de fatoração das operações locais **acabou**, **encontrou** e **iniciePesquisa**. E a aplicação a seguir usa o componente original e o adaptável, que convivem harmoniosamente, cada um com sua semântica.

```
class Aplicação {
   public static void main(String args) {
      Tabela t1 = new Tabela();
      OutraTabela t2 = new OutraTabela();
      Elemento x;
      OutroElemento y;
      ...
      if t1.pesquise(x) { ... };
      if t2.pesquise(y) { ... };
      ...
   }
}
```

Os mecanismos de ligação dinâmica, polimorfismo de referências, polivalência de funções e fatoração de operações garantem que os métodos corretos serão ativados em cada caso.

Conclusão

Réuso de componentes, principalmente o de componentes abertos e originais, são prática frequente no processo de desenvolvimento de software desde os primórdios da programação de computadores. É muito comum o reaproveitamento e adaptação de trechos de programas na implementação de novos componentes e o uso de componentes disponibilizados por bibliotecas.

Os recursos introduzidos pela orientação por objetos facilitam enormente os reúsos baseados na adaptação de componentes de software sem alterá-los diretamente, permitindo atingir alto grau de reusabilidade. É considerado boa prática de programação sempre buscar desenvolver componentes com características que facilitam futuro reúso, como valorizar encapsulação e genericidade das estruturas que representam os objetos.

A prática de reúso necessita de recursos linguísticos diversos para ser efetivada. Os principais recursos envolvidos diretamente nesse processo são classes, interfaces, pacotes, encapsulação, composição, herança, hierarquia de tipos, especialização, redefinição de métodos, declaração de atributos, vinculação dinâmica de métodos a mensagens, referências polimórficas, funções polissêmicas, funções polivalentes, tratamento de exceção e genericidade.

Em resumo, orientação por objetos contribui diretamente para tornar reúso factível em diversos níveis.

Exercícios

1. Por que a declaração de campos com visibilidade `protected` viola encapsulação?

2. Em que consiste fatoração de operações de uma classe? Qual é o recurso usado para atingir esse fim?

Notas bibliográficas

O livro *Object-Oriented Software Construction* de Bertrand Meyer [41] apresenta um excelente capítulo sobre reúso: vale a pena sua leitura.

Capítulo 9

Princípios de Projeto Modular

O processo de desenvolvimento de componentes de software modulares demanda a observância de um conjunto de princípios para a construção de bons módulos. Esses princípios disciplinam a organização de cada módulo por meio de ações e decisões que reduzem conectividade, estimula encapsulação da representação de objetos, limita tamanho de interfaces e reduz o impacto das alterações feitas um módulo nos demais componentes do sistema.

Os principais princípios para disciplinar a construção de bons módulos em ambientes orientados por objetos são os seguintes:

- Encapsulação de constantes: use sistematicamente constantes simbólicas no lugar de constantes literais.
- Respeito à natureza das classes: use composição e herança conforme seus propósitos.
- Programação para a interface: programação deve basear-se em interface e nunca na implementação de uma classe.
- Inversão da dependência: módulos de alto nível não devem depender de módulos de nível mais baixo.
- Resiliência ou aberto-fechado: entidades de software são implementadas para nunca serem modificadas.
- Substituição de Liskov: funções que usam objetos de uma classe devem ser capazes de usar objetos de suas subclasses.
- Segregação de interfaces: interfaces mais específicas são me-

lhores que interfaces de propósitos diversos.

- Responsabilidade Única: nunca deve haver mais de um motivo para uma classe ser alvo de alteração.

A estrita observação desses princípios pode reduzir a liberdade de ação do projetista, mas, com certeza, é compensada pela elevação do grau de réuso dos módulos produzidos, por facilitar o processo de depuração e consequente redução do custo de manutenção.

9.1 Encapsulação de constantes

Princípio da Encapsulação de Constantes
Use sistematicamente constantes simbólicas no lugar de constantes literais.

Inclusão de valores literais diretamente nos comandos de um programa dificulta a sua manutenção. Veja, por exemplo, impacto do uso do literal `9` para designar o número de dígitos dos telefones celulares em uma dada aplicação, e de `12` para representar o número de dígitos dos mesmos telefones incluindo o código DDD. Observe que, nesse programa, uma pequena mudança nos requisitos, por exemplo, expansão do número de dígitos de 9 para 10, causará um esforço brutal de atualização do programa, pois todo o código, linha a linha, terá que ser inspecionado para que as alterações necessárias sejam consistentemente realizadas, pois nem todo `9` ou `12` presentes no programa deverão ser atualizados.

Um método de projeto que adota a prática de usar valores literais diretamente em todo uso de constantes não é contínuo, pois pequenas alterações na sua especificação podem causar grande impacto. A recomendação é que somente deve-se usar constantes literais para definir constantes simbólicas, de forma a facilitar, ao longo da vida útil da aplicação, a troca de constantes literais.

Constantes simbólicas podem ser classificadas quanto ao tipo de acoplamento em que participam como:

- de sistema: constantes de interesse geral.
- de biblioteca: constantes de interesse de uma biblioteca.
- de módulo: constantes de interesse de um módulo.

Os valores atribuídos a constantes simbólicas de sistema podem ser dependentes de plataforma, mas essas constantes, que podem ter o escopo de todo o sistema, devem ser usadas de forma que alterações de seus valores não impliquem em manutenção dos módulos que as utilizam. Essas constantes simbólicas podem ser agrupadas em algum pacote, que reúne um conjunto de classes públicas empacotadoras, cada qual definindo um conjunto coeso de constantes simbólicas públicas relacionadas, como ilustrado pelo seguinte esquema de código do pacote `CS`, que exporta as constantes `CE1.C11`, ..., `CE1.C1n`, `CE2.C21`, ..., `CE2.C2m`:

- Arquivo `CE1.java`:

```
1 package CS;
2 public class CE1 {
3    public static final T C11 = valor;
4    ...
5    public static final T C1n = valor;
6 }
```

- Arquivo `CE2.java`:

```
1 package CS;
2 public class CE2 {
3    public static final T C21 = valor;
4    ...
5    public static final T C2m = valor;
6 }
```

Constantes simbólicas públicas de biblioteca podem ser definidas conforme o seguinte esquema de código:

- Arquivo `CP1.java`, que define as constantes públicas do pacote `BibliotecaA` de nomes `CP1.C11`, ... , `CP1.C1n`:

```
1 package BibliotecaA;
2 public class CP1 {
3     public static final T C11 = valor;
4     .....
5     public static final T C1n = valor;
6 }
```

- Arquivo `CP2.java`, que define as constantes públicas do pacote `BibliotecA` de nomes `CP2.C21`, ... , `CP2.C2m`:

```
1 package BibliotecaA;
2 public class CP2 {
3     public static final T C21 = valor;
4     .....
5     public static final T C2m = valor;
6 }
```

- Arquivo `CL1.java`, que define as constantes locais do pacote `BibliotecaA` de nomes `CL1.K11`, ..., `CL1.K1n`, `CL2.K21`, ..., `CL2.K2m`:

```
1 package BibliotecaA;
2 class CL1 {
3    public static final T K11 = valor;
4    ...
5    public static final T K1n = valor;
6 }
7 class CL2 {
8    public static final T K21 = valor;
9    ...
10   public static final T K2m = valor;
11 }
```

Declaração de constantes simbólicas públicas e privadas de módulos é ilustrada pelo seguinte trecho de código, onde `C1`, ..., `Cn` são constantes públicas, e `K1`, ... `Km` são constantes privativas de `M1`.

```
1 public class M1 {
2     public  static final T C1 = valor;
3     ...
4     public  static final T Cn = valor;
5     private static final T K1 = valor;
6     ...
7     private static final T Kn = valor;
8     ...
9 }
```

9.2 Respeito à natureza das classes

Princípio do Respeito à Natureza das Classes
Use composição e herança conforme seus propósitos mais nobres.

Esse princípio é, segundo Bertand Meyer, Gamma et alii, denominado **Preferência à Composição**, para enfatizar que muitas vezes agregação de estruturas por composição é preferível à herança no processo de criação de classes.

Estruturas de dados são formadas pela agregação de declarações de campos de diversos tipos, os quais podem ser primitivos, como `boolean`, `char`, `int`, `double` e `float`, ou nomes de classes, as quais definem outras estruturas. E esse processo de agregar estruturas baseia-se em dois mecanismos básicos: **composição** e **extensão**.

Composição trata da combinação de estruturas dados de um ou mais tipos para obter uma agregação específica. Um exemplo é a declaração `class X{int a; boolean[] b; char c;}`.

E agregação por extensão constrói uma nova estrutura por meio da justaposição de estruturas, via um mecanismo denominado herança, como a declaração `class B extends A{char c;}`, onde a classe `A` foi definida por `class A{int a; boolean[] b;}`, produzindo para `B` o mesmo leiaute da classe `X`.

Composição é um método de reúso no qual novas funções são obtidas por agregação de objetos. Cada componente de uma composição possui seu próprio conjunto de operações, que são usadas para definir as operações dos objetos gerados pela composição.

Objetos que formam uma composição são chamados de objetos contidos, e o objeto que os contém chama-se compositor, e sua respectiva classe, compositora. Os benefícios do mecanismo de composição, quando usado para formar estruturas de dados mais complexas, são pelo menos os seguintes:

- Objetos contidos podem ser acessados pela classe compositora somente por meio de suas interfaces.
- Composição é um reúso caixa-preta, pois detalhes internos dos objetos contidos não precisam de ser visíveis.
- Composição favorece boa encapsulação, separando interesses.
- Composição não depende de como os objetos contidos são implementados.
- A composição é dinamicamente definida, porque as referências aos objetos contidos são obtidas durante a execução e podem ter seus valores alterados a qualquer momento.

Extensão, por outro lado, realiza agregação de estruturas com base no conceito de herança, que pode ser simples ou múltipla, e que permite criar novas classes a partir de classes existentes. Nesse processo, as chamadas superclasses definem um conjunto de atributos ou campos, e a classe especializada, via herança, estende as estruturas herdadas acrescentando novos campos, fazendo assim

uma agregação dessas estruturas.

Herança apresenta os seguintes benefícios ao projeto de software modular:

- Facilita a geração de novas implementações, porque reúso é facilitado.
- Facilita modificação ou extensão das implementações que são reusadas.
- Permite reúso de funções polivalentes decorrente do polimorfismo inerente à estrutura hierárquica de tipo.
- Possibilita redefinição de comportamento.

Há, entretanto, algumas dificuldades vinculadas ao uso de herança para realizar combinação de objetos, como:

- Herança possibilita violar encapsulação, porque detalhes da superclasse podem estar expostos na subclasse.
- Subclasses podem ter que ser adaptadas quando altera-se sua superclasse.
- O uso de herança para agregar representações de objetos não é o seu propósito principal.

Esses dois mecanismos de agregação de estruturas de dados, a composição e a extensão, podem levar a programas que produzem os mesmos resultados, mas, do ponto de vista metodológico, eles não são equivalentes: O fato é que são as circunstâncias de cada aplicação e seus contextos de uso que determinam o mecanismo de agregação de estruturas que é mais apropriado em cada caso. Por exemplo, um fator relevante para fundamentar essa escolha é a natureza dos objetos e o papel desempenhado por eles dentro do sistema.

Para ilustrar essa afirmação, considere a implementação das classes `Segmento1` e `Segmento2` apresentadas a seguir.

Suponha a seguinte declaração da classe **Ponto**, a qual define um ponto de coordenadas cartesianas e disponibiliza pelo menos duas operações: a construtora **Ponto** e o método **distância**, que calcula a distância do ponto corrente a um ponto passado via parâmetro. As demais operações de **Ponto** foram omitidas para simplificar a apresentação do exemplo.

```
1  public class Ponto {
2    private double x1, y1;
3    public Ponto(double x1, double y1) {
4      this.x1 = x1;
5      this.y1 = y1;
6    }
7    public double distância(Ponto p) {
8      double dx = p.x1 - this.x1;
9      double dy = p.y1 - this.y1;
10     return Math.sqrt(dx*dx + dy*dy);
11   }
12   ...
13 }
```

Segundo a Geometria Analítica, um segmento de reta é definido por dois pontos. Assim, a representação de objetos que são segmentos deve conter os dois pares de coordenadas que os delimitam.

Como a classe **Ponto** definida acima já define um dos pares necessários à definição de um segmento e ainda tem uma função que calcula distância entre dois pontos, uma solução inspirada em reúso sugere definir a classe **Segmento1** como uma extensão da classe **Ponto** pela inclusão de um novo par de coordenadas e de operações pertinentes, como **comprimento** para informar a dimensão de um segmento.

Essa forma de implementação valoriza reúso e ilustra como combinar estruturas dados por meio do mecanismo de herança.

```
1  public class Segmento1 extends Ponto {
2    private double x2, y2;
3    public Segmento1(
4                  double x1,double y1,double x2,double y2){
5      super(x1,y1); this.x2 = x2; this.y2 = y2;
6    }
7    public double comprimento() {
8      Ponto p2 = new Ponto(this.x2, this.y2);
9      return super.distância(p2);
10   }
11   ...
12 }
```

Observe que a classe **Segmento1** usa a operação **distância** de **Ponto** para implementar o seu método **comprimento**.

O programa de teste de **Segmento1**, apresentado a seguir, imprime o valor `5.0`, que é computado pela operação **comprimento**, dados os pontos `(2,3)` e `(5,7)`, limites do segmento.

```
1  public class UsaSegmento1 {
2     public static void main(String[] args) {
3        Segmento1 s = new Segmento1(2,3,5,7);
4        System.out.println(s.comprimento());
5     }
6  }
```

O programa **UsaSegmento1** funciona a contento, mas essa implementação apresenta um problema conceitual devido ao uso pouco nobre do mecanismo de herança, cujo propósito principal é o de criar hierarquias de tipos, que compartilham propriedades e códigos de implementação, e concede alto grau de reúso aos componentes de software implementados, mas herança tem também uma função menos nobre, que é apenas dar suporte à prática de reúso de implementação. A classe **Segmento1** é um exemplo deste tipo de

suporte, pois essa classe, apesar de estender `Ponto`, não é conceitualmente um subtipo de `Ponto`, i.e., um segmento não é um ponto.

Embora o mecanismo de polimorfismo de linguagens orientadas por objetos admita que objetos do tipo `Segmento1` possam ser usados onde objetos do tipo `Ponto` forem esperados, essa prática deve ser evitada, devido ao erro conceitual acima mencionado.

Uma solução mais condizente com o espírito da Geometria Analítica é oferecida para classe `Segmento2`, apresentada a seguir, que implementa segmentos como objetos compostos por dois pontos `p1` e `p2` explicitamente declarados, usando o mecanismo composição, discutido no início desta seção.

```
1  public class Segmento2 {
2    private Ponto p1, p2;
3    public Segmento2(
4                  double x1,double y1,double x2,double y2){
5       p1 = new Ponto(x1,y1);
6       p2 = new Ponto(x2,y2);
7    }
8    public double comprimento() {
9       return p1.distância(p2);
10   }
11 }
```

O programa a seguir imprime exatamente o mesmo valor que o do programa `UsaSegmento1`.

```
1  public class UsaSegmento2 {
2     public static void main(String[] args) {
3        Segmento2 s = new Segmento2(2,3,5,7);
4        System.out.println(s.comprimento());
5     }
6  }
```

Em suma, os programas `Segmento1` e `Segmento2` apresentados mostram que o poderoso recurso da herança deve ser usado com parcimônia no sentido de valorizar conceitualmente as soluções de implementação. Nesse exemplo, tratar um segmento como se fosse um ponto nada tem de elegante, sendo melhor dar-lhe uma identidade própria.

Há ainda outras regras que, baseadas na natureza de cada objeto participante da aplicação, ajudam a dirigir o processo de escolha entre composicão e extensão para compor novas estruturas de dados. Nesse sentido, Peter Coad et alii sugerem que a natureza de cada objeto em uma aplicação seja necessariamente a de um dos seguintes tipos de entidades:

- **Papel**: o objeto representa um papel na aplicação, e.g., aluno e professor.
- **Transação**: o objeto descreve um tipo de atividade que pode ocorrer em determinado momento ou intervalo de tempo, e.g., compra, venda e pagamento.
- **Ente**: o objeto representa uma entidade, dispositivo ou coisa que tem estrutura permanente, e.g., pessoa, árvore, cadeira.

Todo objeto deve representar apenas uma das entidades descritas acima, e suas características devem ser passadas a seus descendentes. E todos os objetos devem sempre ser usados conforme as entidades que representam e nunca devem ter que ser convertidos de um tipo a outro.

Essa restrição aparentemente pode limitar a função dos objetos que podem ser criados via o mecanismo de herança, mas ela valoriza a organização modular.

Essencialmente, essa visão sobre a natureza dos objetos gera a recomendação de que herança deve ser usada com cuidado para que subclasses expressem sempre uma relação “é um tipo especial

de" e não a relação "pode exercer o papel de", porque herança deve preservar na subclasse a natureza ou tipo de entidade representada pela superclasse na aplicação.

Toda subclasse deve especializar ou um papel, ou uma transação ou um ente, para formar novos papéis, novas transações ou novos entes especializados, respectivamente. E isso deve ser feito sem se redefinir nem anular responsalidades da superclasse, ou seja, a especialização não pode mudar o tipo herdado, estando limitada a estendê-lo, haja vista que anular funcionalidade compromete subtipagem e pode gerar comportamentos incompatíveis entre classes e subclasses, frustando a expectativa dos módulos-cliente.

Para atingir esses resquisitos, deve-se observar as seguintes regras de uso saudável do mecanismo de especialização de classes:

- Especialização de Papel: a especialização de um papel tem que ser um papel. Por exemplo, `Professor` pode ser uma especialização de `Servidor` em uma universidade, mas não deve ser definido como uma especializacão de `Pessoa`, a qual normalmente é um *ente* e não um *papel*.
- Especialização de Transação: transações podem ser especializadas para definir transações mais específicas. Por exemplo, `Reserva` e `Venda` são tipos especializados da transação `EmissãoDePassagens`.
- Especialização de Entes (Dispositivo, Entidade ou Coisa): classes que denotam dispositivos, entidades ou coisas devem ser estendidas para gerar objetos de mesma natureza, exceto mais especializados. Por exemplo, `Caminhão` é uma especialização legítima do tipo `Veículo`, pois é um tipo especial de veículo.

Além disso, somente deve-se construir hierarquias com classes definidas no domínio da própria aplicação. Isso significa que não se deve estender, por exemplo, classes-utilitárias, porque herança

é um mecanismo de reúso caixa-branca, portanto a subclasse pode depender de detalhes de implementação de sua superclasse. A implementação de classes utilitárias pode ser dependente de plataforma, e, assim, mudanças de plataforma podem requerer mudanças em detalhes da classe utilitária, forçando manutenção da aplicação para atualizar subclasses de classes utilitárias.

As condições que autorizam o uso de herança são as seguintes:

C1: A subclasse deve expressar uma relação "é um tipo especial de" e não "pode exercer o papel de", sendo sempre um tipo especializado da superclasse.

C2: A subclasse deve especializar um papel, uma transação ou um ente, para produzir novos papéis, transações ou entes, respectivamente. Ou seja, herança não deve dar à subclasse uma natureza distinta da de sua superclasse.

C3: A subclasse deve estender, e não redefinir nem anular responsalidades da superclasse, i.e, a subclasse deve ser de fato um subtipo da superclasse.

C4: A subclasse não deve estender classes fora do seu domínio.

C5: A hierarquia não deve criar necessidade de converter objetos de uma subclasse em outra da mesma família.

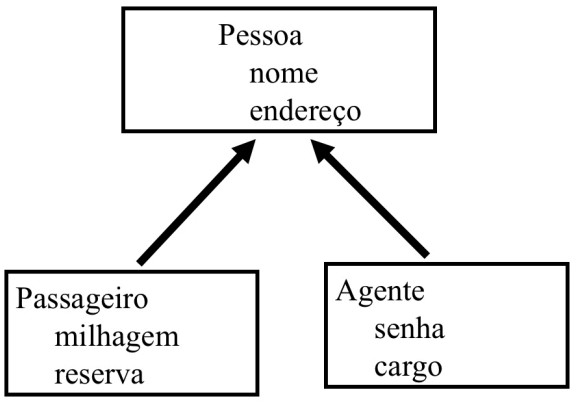

Figura 9.1 Hierarquia

No exemplo da Fig. 9.1, objetos do tipo `Passageiro` são formados pela combinação dos campos `nome`, `endereço`, `milhagem` e `reserva`, enquanto os do tipo `Agente` têm os campos `nome`, `endereço`, `senha` e `cargo`. Os campos `nome` e `endereço` de ambos são herdados de `Pessoa`. Essa hierarquia, entretanto, viola as regras **C1**, **C2** e **C5**, porque `Passageiro` e `Agente` não são tipos especiais de `Pessoa`, mas papéis que eles exercem dentro da aplicação. E essa violação pode trazer algumas dificuldades no caso de ser necessário redefinir os papéis de objetos em uma aplicação, como no caso de se desejar dar ao objeto `Passageiro` o papel de `Agente`, e desejar fazê-lo com um mínimo de impacto no sistema.

Uma solução para implementar essa alteração seria o uso de herança múltipla como mostrado na Fig. 9.2, a qual valoriza reúso e elimina necessidade de conversão de `Passageiro` em `Agente`, mas é demasiadamente rígida, e muitos ambientes de programação não operam com herança múltipla.

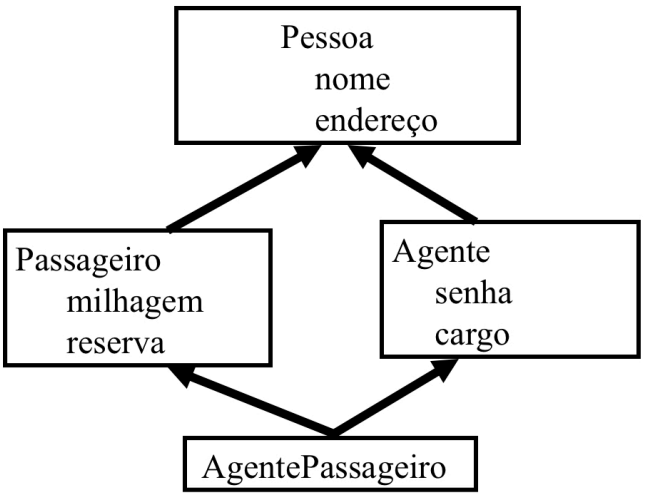

Figura 9.2 Agente Viajante

Claramente, a organização da Fig. 9.1 não é a melhor solução para o problema em foco, pois subclasses deviam especializar somente papel, transação ou ente para gerar novos papéis, transações

ou entes, respectivamente. E no caso, `Pessoa` é um ente, mas `Passageiro` e `Agente` são papéis.

Uma primeira solução, que produz um melhor resultado, seria usar composição, no lugar de herança, para construir `Passageiro` e `Agente`, como mostrado na Fig. 9.3.

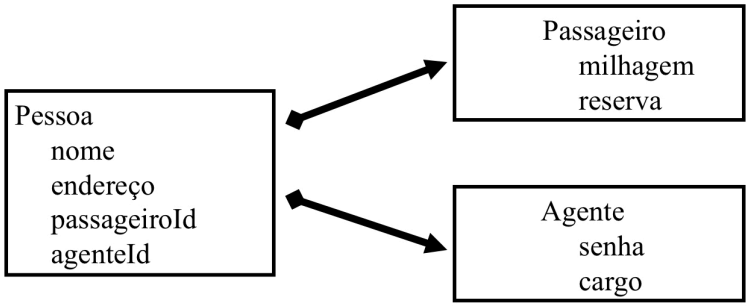

Figura 9.3 Uso de Composição

onde a classe `Pessoa` passa a incorporar os campos `passageiroId` e `agenteId` com a descrição de seus papéis.

Nessa solução, todas as quatro condições para o bom uso de herança estão satisfeitas, e o papel de `Pessoa` pode mudar dinamicamente durante a execução, alterando-se os valores dos campos `passageiroId` e `agenteId`.

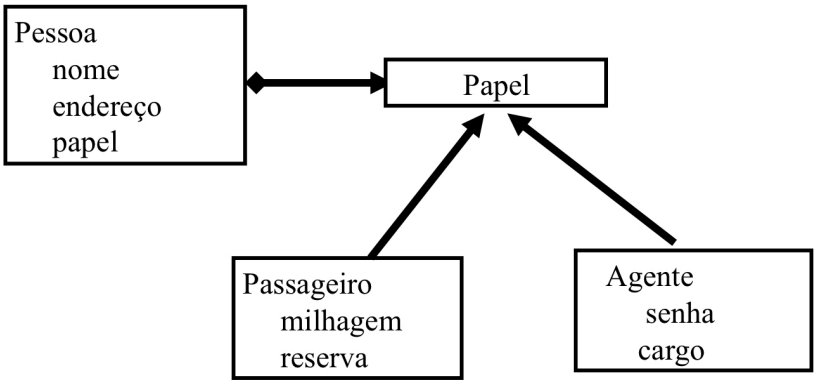

Figura 9.4 Composição + Herança

Uma outra solução, ainda melhor, é usar uma combinação dos mecanismos de composição e herança, conforme ilustrado na Fig. 9.4, onde `Passageiro` e `Agente` são tipos especiais do papel exercido

por `Pessoa`, como recomenda a condição **C1**. Nessa solução, novos papéis para `Pessoa` podem ser inventados e não se tem que converter um papel em outro.

A especialização de classes que representam transações é ilustrada na Fig. 9.5, onde `Reserva` e `Venda` são tipos especiais de `EmissãodePassagens`.

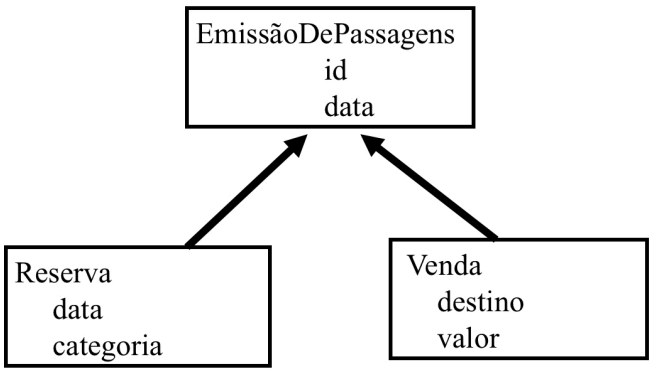

Figura 9.5 Herança OK

Conclui-se que composição e herança são importantes no processo de reúso de componentes de software, mas devem cooperar-se. Herança é muito útil para aumentar o grau de polivalência de métodos, mas deve ser usada com cuidado, obedecendo a regra de que uma subclasse sempre deve expressar a relação "é um tipo especial de", e não "pode exercer o papel de". Essa regra leva, em muitas situações, à preferência de composição à herança.

9.3 Programação para interface

Princípio da Programação para Interface
Programação deve basear-se em interface e nunca na implementação de classe.

Uma interface define o conjunto de métodos públicos que podem ser chamados por módulos-cliente. Esses métodos de interface são

um subconjunto dos métodos públicos de cada uma das classes que implementam a interface. Um objeto pode então ter muitas interfaces, cada uma correspondendo a um subconjunto dos métodos que sua classe torna públicos. E cada conjunto de métodos define um tipo de dados distinto, o qual é uma interface específica de um ou mais objetos, e um mesmo objeto pode ter diferentes tipos.

A ideia é que os objetos sejam conhecidos via suas interfaces, as quais devem expressar a propriedade de "ser um de". O foco na interface e não nas classes que a implementam permite o uso de notação uniforme para acesso aos serviços de um módulo, sem qualquer dependência em como eles são implementados.

Na programação voltada para interface, módulos-cliente não precisam preocupar-se com a classe específica do objeto que se está usando, e um objeto pode ser substituído por outro de diferente classe, aumentando a flexibilidade dos serviços oferecidos, conferindo um menor grau de acoplamento de classes, aumentando seu grau de reúso e facilitando composição, porque objetos contidos podem ser de quaisquer classes que implementem a interface.

Em suma, o princípio fundamental é que todo objeto deve ter seu tipo definido por uma interface pela qual seus usuários o conhecem, e o uso de um objeto deve sempre independer de sua implementação.

9.4 Inversão da dependência

Princípio da Inversão de Dependência

Abstrações não devem depender de detalhes. São os detalhes que devem depender das abstrações.

Suponha que a classe `R` disponibilize operações para fazer a leitura de caracteres de um dado arquivo e que as classes `W1` e `W2`

implementem arquivos de saída. A operação `copia`, definida a seguir, transfere o contéudo de um arquivo do tipo `R` para o arquivo de saída da família `W1` ou `W2`, conforme seja o dispositivo especificado em sua chamada.

```
1  public class A {
2     public void copia(char device) {
3       char c;
4       R r   = new R(...);
5       W1 w1 = new W1(...);
6       W2 w2 = new W2(...);
7       while ((c = r.read()) != EOF) {
8         if (device == 'X') w1.write(c); else w2.write(c);
9       }
10    }
11 }
```

No programa acima, `A` depende das classes `R`, `W1` e `W2`, e isso limita seu reúso para realizar cópias de outros tipos de arquivos. Isso acontece quando uma abstração depende de detalhes que, em princípio, não têm vínculo com sua semântica.

Uma solução alternativa, que eleva o nível de abstração, produz um programa que deixa de fora detalhes dos arquivos envolvidos:

```
1  public interface Reader {public char read();}
2  public interface Writer {public void write(char c);}
3  public class A {
4     public void copia(Reader r, Writer w) {
5        char c;
6        while ((c = r.read()) != EOF) {w.write(c);}
7     }
8  }
```

Nessa implementação, a classe `A` não depende das classes específicas que porventura implementem `Reader` ou `Writer`, como recomenda o princípio da inversão de dependência.

9.5 Resiliência

Princípio da Resiliência

Entidades de software devem ser implementadas de forma a nunca terem que ser modificadas, mas apenas estendidas.

Esse princípio, que é também conhecido por *Aberto-Fechado*, aplica-se à construção de módulos de grande capacidade de adaptação. O termo resiliência destaca que sua adaptação é elástica e não compromete a semântica original do módulo, que pode ser estendido de várias formas, enquanto mantém-se a integridade do módulo original.

Aplicação desse princípio permite obter maior grau de reúso e extensibilidade, podendo ser considerado uma prática central do projeto orientado por objetos.

Um módulo é dito aberto quando está disponível para ser estendido, pela expansão de seu conjunto de operações ou para receber novos campos em sua estrutura de dados, por meio do mecanismo de criar subclasses. Um bom módulo deve ser fechado para modificações, porém deve estar sempre aberto para incorporar novas extensões.

Módulo fechado é um módulo pronto para uso e armazenado na biblioteca de módulos, portanto disponível para uso por outros componentes de software.

Um módulo deveria ser fechado para modificações somente quando julga-se que já atingiu estabilidade, ou seja, quando tiver uma interface bem definida e estável. Mas, frequentemente, fecha-se um módulo porque não se pode esperar o levantamento de toda informação sobre o uso do módulo

O princípio Aberto-Fechado ou da Resiliência estabelece que se deve tentar sempre construir módulos que nunca tenham que ser modificados, mas que possam ser adaptados somente via extensões,

quando for necessário.

Provavelmente nem todo módulo de um sistema satisfaz esse princípio, mas deve-se minimizar o número desses módulos. As extensões típicas são inclusão de novos campos na estrutura de dados contida no módulo e de novos serviços. E os recursos necessários para exercer com sucesso esse princípio são abstração, polimorfismo, herança e interfaces.

Para ilustrar a aplicação do Princípio da Resiliência, considere uma aplicação com as classes **Peça**, **Estoque** e **Vendedor**.

```
1  public class Peça {
2     private String nome;
3     private double preçoBase;
4     public Peça(String nome, double preço) {
5        this.nome = nome; preçoBase = preço;
6     }
7     public void setPreço(double preço) {preçoBase = preço;}
8     public double getPreço() {return preçoBase;}
9  }
```

A classe **Estoque** fornece o preço total de uma lista de peças:

```
1  public class Estoque {
2     ...
3     public double preçoTotal(Peça[] peças) {
4        double total = 0.0;
5        for (int i = 0; i < peças.length; i++) {
6             total += peças[i].getPreço();
7        }
8        return total;
9     }
10    ...
11 }
```

E a classe **Vendedor** seleciona 10 peças a ser vendidas e calcula o custo final.

```
1  public class Vendedor {
2     public static void main(String arg[]) {
3        double custo;
4        Estoque estoque = new Estoque(...);
5        Peça[] peças = new Peça[10];
6        for (int i = 0; i <10 ; i++ ) {peças[i] = ...;}
7        custo = estoque.preçoTotal(peças);
8        System.out.println(custo);
9     }
10 }
```

Entretanto, se a empresa decidir cobrar um preço adicional por algumas peças, por exemplo, do tipo `Memória` e `PlacaMãe`, subclasses de `Peça`, a primeira solução poderia ser alterar o método `preçoTotal` da classe `Estoque` para o seguinte código:

```
1  public double preçoTotal(Peça[] peças) {
2     double total = 0.0;
3     for (int i = 0; i < peças.length; i++) {
4          if (peças[i] instanceof PlacaMãe)
5               total += (1.45* peças[i].getPreço());
6          else if (peças[i] instanceof Memória)
7                    total += (1.30* peças[i].getPreço());
8               else total += peças[i].getPreço();
9     }
10    return total;
11 }
```

Claramente, essa solução não atende o Princípio da Resiliência, porque o módulo `Estoque` terá que ser alterado toda vez que se decidir por uma nova política de preços de peças, pois dever-se-á sempre modificar novamente seu método `preçoTotal`.

E isso pode causar impactos indesejáveis na aplicação, porque outros módulos (vendedores) poderiam, simultaneamente, desejar diferentes, e possivelmente incompatíveis, políticas de preços.

O princípio da Resiliência visa evitar problemas desse tipo, recomendando o fechamento do módulo `Estoque`, que contém o método `preçoTotal`, para que ele possa ser usado, e cuidar que não tenha que ser modificado posteriormente.

Para realizar fechamento da classe `Estoque`, há pelo menos dois caminhos:

- Caminho I: incorporar a política de preço redefinindo o método `getPreço` de cada subclasse que define uma peça específica.
- Caminho II: criar uma classe `PolíticaDePreço` para acomodar diferentes políticas via subclasses.

Caminho I: fechando `Estoque`

Uma solução natural é incorporar a política de preço de uma peça na própria peça, via a definição de subclasse, como mostram as seguintes declarações das subclasses `PlacaMãe` e `Memória`:

```
1  public class PlacaMãe extends Peça {
2     public double getPreço() {
3        return 1.45 * super.getPreço();
4     }
5  }
6
7  public class Memória extends Peça {
8     public double getPreço() {
9         return 1.30 * super.getPreço();
10    }
11 }
```

Nesse caso, o método `preçoTotal`, reapresentado abaixo, em sua forma original definida na classe `Estoque` trata todas as peças de forma uniforme, e o mecanismo de polimorfismo acionará o `getPreço` pertinente a cada peça.

```
1 public double preçoTotal(Peça[] peças) {
2    double total = 0.0;
3    for (int i=0; i<peças.length; i++) {
4        total += peças[i].getPreço( );
5    }
6    return total;
7 }
```

Observe que `preçoTotal` e, consequentemente, `Estoque` não precisam mais ser alterados para processar diferentes políticas de preço de peças: somente as subclasses especiais de `Peças` são afetadas, demonstrando a aplicação do Princípio.

Caminho II: fechando `Estoque`

Uma solução mais radical e definitiva é reprojetar a classe `Peça` para incorporar uma política de preços a ser usada conforme a demanda, como no esquema a seguir:

```
1  public class Peça {
2     private String nome;
3     private double preçoBase;
4     private PolíticaDePreço política =
5                                     new PolíticaDePreço();
6     public Peça(String nome, double preço) {
7        this.nome = nome; preçoBase = preço;
8     }
9     public void setPolíticaDePreco(PolíticaDePreço p) {
10         política = p;
11    }
12    public void setPreço(double preço) {preçoBase=preço;}
13    public double getPreço() {
14         return políticaDePreço.getFator( ) * preçoBase;
15   }
16 }
```

A política de preços é definida por uma classe adicional denominada `PolíticaDePreço`:

```
1 public class PolíticaDePreço {
2    private double fator = 1.0;
3    public void setFator(double fator){this.fator = fator;}
4    public double getFator() {return fator;}
5 }
```

E a classe `Vendedor` passa a ter a seguinte implementação, onde cada peça pode ter uma política de preço especial:

```
1  public class Vendedor {
2     public static void main(String arg[]) {
3       double custo;
4       Estoque estoque = new Estoque(...);
5       PolíticaDePreço p = new PoliticaDePreço();
6       p.setFator(1.5);
7       Peça[] peças = new Peça[10];
8       for (int i = 0; i <10 ; i++ ) {
9         ...
10        peças[i] = ...;  // define uma das peças à venda
11        if ( ...) peças[i].setPolíticaDePreço(p);
12        ...
13      }
14      custo = estoque.preçoTotal(peças);
15      System.out.println(custo);
16    }
17 }
```

As mudanças na política de preço são uma decisão do vendedor, i.e., de objetos do tipo `Vendedor`, e não afetam as classes `Peça`, `Estoque` e `PolíticaDePreço`, que após fechadas não mais precisam ser alteradas. Os pontos de modificação foram transferidos para a classe `Vendedor`, exibindo claros ganhos em reusabilidade e extensibilidade.

Em resumo, a implementação de subclasses nunca deve demandar alterações nas respectivas superclasses, e a inclusão de novas classes em uma aplicação não deveria implicar na alteração de classes existentes, pois novas classes deveriam no máximo estender classes existentes, sem causar-lhes modificações, ou seja, classes devem ser projetadas para nunca serem modificadas.

9.6 Substituição de Liskov

Princípio da Substituição de Liskov

Funções que usam referências para objetos de uma classe devem ser capazes de referenciar objetos das subclasses dessa classe automaticamente.

Polimorfismo, ligação dinâmica, hierarquia de tipos são mecanismos que concedem grande aplicabilidade às operações das classes, mas para garantir que a semântica desejada seja obtida é preciso tomar alguns cuidados no processo de uso da herança. Para ilustrar esse argumento, considere o método `desenheForma`:

```
1 public void desenheForma(Forma f) {
2     ...
3     f.desenhe();
4     ...
5 }
```

No método acima, `Forma` pode ser uma classe ou uma interface que tenham a operação `desenhe`, e o método `desenheForma` deve funcionar também para qualquer descendente de `Forma`. Certamente, do ponto de vista sintático a construção está correta, pois conforma-se com as regras de uso de tipos e subtipos, mas pode acontecer que os resultados produzidos não sejam sempre o esperado.

O exemplo a seguir mostra uma dessas situações em que o problema causado é muito sutil, difícil de ser notado, mas que deve ser evitado. A pequena aplicação a seguir ilustra uma dessas situações.

Considere inicialmente a classe **Retângulo**:

```
1  public class Retângulo {
2      private double largura, altura;
3      public Retângulo(double w, double h) {
4          largura = w; altura = h;
5      }
6      public double getLargura() {return largura;}
7      public double getAltura() {return altura;}
8      public void setLargura(double w) {largura = w;}
9      public void setAltura(double h) {altura = h;}
10     public double area() {return largura * altura;}
11 }
```

Considere agora a classe **Quadrado**. Sabe-se que todo quadrado é um tipo especial de retângulo, portanto a classe **Quadrado** pode ser declarada como um subtipo de **Retângulo**, naturalmente derivável por meio do mecanismo de herança.

Note que, nesse exemplo, herança incorpora aos quadrados os atributos privados **largura** e **altura** herdados de **Retângulo** e as operações associadas a esses campos.

A presença desses atributos em quadrados é pouco usual, mas, em princípio, não é um problema, pois quadrados podem têm dimensões, mas deve-se assegurar a preservação do invariante de que nos objetos do tipo **Quadrado** a largura seja sempre igual a sua altura, e as operações associadas sejam definidas apropriadamente para preservar esse requisito.

Na classe **Quadrado**, definida a seguir, as operações **setLargura** e **setAltura** herdadas foram redefinidas conforme esperado, e as demais operações **Retângulo** continuam válidas.

```
1  public class Quadrado extends Retângulo {
2      public Quadrado(double w) {
3          super(w,w);
4      }
5      public void setLargura(double w) {
6         super.setLargura(w); super.setAltura(w);
7      }
8      public void setAltura(double h) {
9         super.setAltura(h); super.setLargura(h);
10     }
11 }
```

Considere a seguinte implementação da classe `Poliformismo`:

```
1  public class Polimorfismo {
2     public static void teste(Retângulo r) {
3        r.setLargura(4.0);
4        r.setAltura(8.0);
5        System.out print( "Área 4.0 X 8.0  =  " + r.area());
6     }
7  }
```

Tudo parece bem feito, as classes Retângulo e Quadrado parecem corretamente escritas e consistentes, mas considere o que ocorre com a seguinte classe usuária `Polimorfismo`:

```
1  public class TestePolimorfismo {
2     public static void main(String[] args) {
3        Retângulo r = new Retângulo(1.0, 1.0);
4        Quadrado  s = new Quadrado(1.0);
5        Polimorfismo.teste(r); // Funciona
6        Polimorfismo.teste(s); // Nao funciona!
7    }
8  }
```

Método `teste` da classe `Polimorfismo` deveria funcionar para todo objeto do tipo `Retângulo` e de seus subtipos, mas o programa `TestePolimorfismo` mostra que isso pode não acontecer imprimindo um resultado não-esperado:

```
Área 4.0 X 8.0 = 32.0
Área 4.0 X 8.0 = 64.0  ?????????
```

O problema é que `Polimorfismo.teste` considerou que alterando-se a altura de um retângulo, a sua largura não poderia ser alterada, porque os métodos `setAltura` e `getLargura` de `Retângulo` operam de forma independente, como normalmente espera-se dos retângulos. E a implementação de `Quadrado` tornou a suposição acima inválida, porque o efeito colateral dessa operação passou a afetar operações que não eram a ela relacionadas na superclasse.

Do ponto de vista matemático, quadrado é um rectângulo, mas isto não é sempre automaticamente verdade do ponto de vista da orientação por objetos. Um objeto `Quadrado` na implementação não é sempre um objeto Retângulo, porque seu comportamento não é consistente com o de um `Retângulo`. Sob esse ponto de vista, um objeto `Quadrado` não deve ser usado onde se espera um objeto `Retângulo`.

Objetos são caracterizados pelo conjunto de operações que disponibilizam, as quais definem seu comportamento, e hierarquia de classes somente deve ser usada quando a relação *é-um*, entre superclasse e subclasse, for relativa a comportamentos. No exemplo apresentado acima isso não foi observado, pois quadrados não se comportam exatamente como retângulos.

O princípio de Substituição de Liskov recomenda que se observe que a relação *é-um* seja sempre relativa a comportamento.

Solução que respeita Liskov

São as operações `setAltura` e `setLargura` que causam os problemas apontados acima, pois elas implementam comportamentos distintos em relação as suas contrapartidas na superclasse. Se removida das classes `Retângulo` e `Quadrado` o problema desaparece, como mostra o código a seguir:

```
1  public class Retângulo {
2      private double largura, altura;
3      public Retângulo(double w, double h) {
4         largura =  w; altura = h;
5      }
6      public double getLargura()  {return largura;}
7      public double getAltura() {return altura;}
8      public double area() {return largura * altura;}
9  }
10 public class Quadrado extends Retângulo {
11     public Quadrado(double w) {super(w,w);}
12 }
```

Os objetos `Retângulo` e `Quadrado` têm agora o mesmo comportamento, porque suas respectivas operações `área`, `getLargura` e `getAltura` têm exatamente as mesmas definições.

A nova classe usuária `Polimorfismo` passa a ter a seguinte implementação:

```
1  public class Polimorfismo {
2     public static void teste(Retângulo r) {
3        int w = r.getLargura(4.0);
4        int h = r.getAltura(8.0);
5        System.out print( "Área " + w + " X " + h + " =  "
6                                                + r.area());
7     }
8  }
```

E a classe principal deve ser alterada para criar os objetos nos tamanhos desejados e estando ciente que esses tamanhos não podem ser alterados.

```
1 public class TestePolimorfismo {
2    public static void main(String[] args) {
3       Retângulo r = new Retângulo(4.0, 8.0);
4       Quadrado  s = new Quadrado(4.0);
5       Polimorfismo.teste(r); // Funciona
6       Polimorfismo.teste(s); // funciona!
7    }
8 }
```

O resultado impresso de `TestePolimorfismo` é o seguinte:

```
Área 4.0 X 8.0 = 32
Área 4.0 X 4.0 = 16
```

Em resumo, para satisfazer o Princípio de Liskov, toda subclasse deve ter o comportamento que os clientes esperam da sua respectiva superclasse. Em termos de contrato, uma classe derivada é de fato uma substituta de sua superclasse se:

- Pré-condições de seus métodos redefinidos nunca são mais fortes que as dos respectivos métodos da classe-base.
- Pós-condições de seus métodos redefinidos nunca são mais fracas que as dos respectivos métodos da classe-base.
- O invariante da subclasse deve implicar no da superclasse e vice-versa, porque o invariante é fator integrante das pré e pós condições de todos métodos das respectivas classes.
- Um subtipo não pode ter mais restrições (em seu comportamento) que o tipo correspondente, porque subtipo deve poder ser usado no lugar do tipo.

Satisfação do Princípio de Liskov assegura que objetos da subclasse podem ser usados onde objetos da superclasse são esperados!

9.7 Segregação de interface

Princípio da Segregação de Interfaces
Interfaces mais específicas são melhores que interfaces de propósito geral.

A prática de reúso e alterações na funcionalidade de sistemas podem levar a criação de classes pouco coesas, caracterizadas por interfaces gordas ou poluídas, sugerindo que implementam contratos múltiplos. Réuso é muito importante, mas interfaces mais enxutas e objetivas também são altamente desejadas. Por isso, atenção ao Princípio da Segregação de Interface é recomendado.

Um exemplo de interface poluída nasce naturalmente durante a seguinte implementação de um sistema de segurança para portas de um prédio, modelado pela classe `Prédio`:

```
1 public class Prédio{
2     private Porta[] porta;
3     ...
4 }
```

que por simplicidade não explicita onde as portas estão localizadas.

Portas podem ser trancadas ou não e sabem quando estão abertas ou não, conforme a seguinte implementação:

```
1 public class Porta {
2     public void fecha() {...}
3     public void abre() {...}
4     public boolean portaAberta() {...}
5 }
```

Há também no sistema as portas de segurança, que são portas como as de cofre, que somente podem ser abertas em determinados horários.

Como portas de cofre são portas, o réuso da classe `Porta` parece adequado para facilitar a implementação de `PortaDeCofre`:

```
1 public classe PortaDeCofre extends Porta {
2    public void defineHorario() {...}
3    public boolean podeAbrir() {...}
4 }
```

Depois de tudo implementado, usuários manifestaram a necessidade de acrescentar ao sistema a capacidade de as portas de cofre dispararem alarmes se ficarem abertas além de um certo tempo.

Para atender essa demanda com um mínimo de custo e impacto no sistema já implementado, decidiu-se pelo uso da classe `Temporizador` já implementada, mas que não se pode ser alterada, pois ela está em uso no mesmo sistema:

```
1 public class Temporizador {
2    public void registra (int t, Monitorado c) {...}
3 }
4 public class Monitorado {
5    public disparaAlarme() {...}
6 }
```

A operação `registra` de `Temporizador` anota o objeto monitorado, o qual será automaticamente notificado, via chamada ao seu método `disparaAlarme`, após decorrido o tempo `t` de temporização. Assim, portas de cofre devem ser objetos monitorados, i.e., objetos do tipo `Monitorado`. A solução óbvia para isso é definir `PortaDeCofre` como uma extensão de `Monitorado`, mas isto não é possível porque `PortaDeCofre` já é uma subclasse de `Porta`. Para contornar essa restrição de Java, que somente opera com herança simples de classes, a solução é fazer a classe Porta estender `Monitorado`, conforme ilustra a Fig. 9.6.

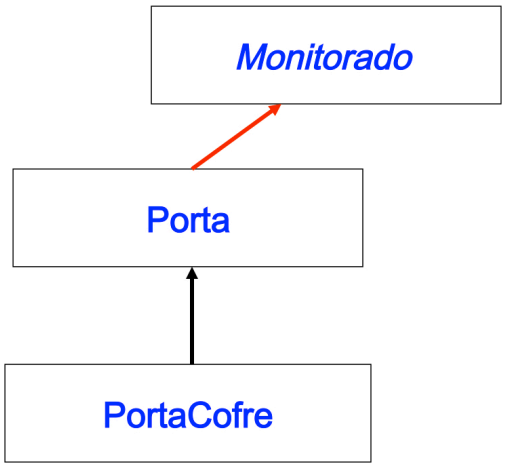

Figura 9.6 Hierarquia dos Monitorados

Essa hierarquia resolve o problema de tornar objetos de tipo `PortaDeCofre` um objeto `Monitorado`, mas gera consequências indesejadas, pois agora `Porta` passa a ter o método `disparaAlarme` em sua interface, porque pelo menos uma de suas subclasses precisa desse comportamento.

```
1  public class Porta extends Monitorado {
2     public void fecha() {...}
3     public void abre() {...}
4     public boolean portaAberta() {...}
5     // public void disparaAlarme() { ... }  // herdado
6  }
7  public classe PortaCofre extends Porta {
8     public void disparaAlarme() {...} // redefinição?
9     public void defineHorario() {...}
10    public boolean podeAbrir() {...}
11 }
```

Considerando que `disparaAlarme` nada têm a ver com a classe `Porta`, pode-se considerar sua presença nesta classe como um poluente.

Essa é a síndrome da **interface poluída**, que foi provocada pela restrição à herança múltipla, a qual poderia ter sido evitada,

com maior elegância, pelo uso da construção de interfaces no lugar de classe quando a classe `Temporizador` foi criada. Se no lugar de ter sido declarado como uma classe, `Monitorado` tivesse sido definido como a interface:

```
1 public interface Monitorado {
2     public disparaAlarme();
3 }
```

que pode ser usada sem qualquer impacto na classe `Temporizador`.

```
1 public class Temporizador {
2    public void registra (int t, Monitorado c) {
3        ...
4    }
5 }
```

E as classes `Porta` e `PortaDeCofre` passariam a ter as seguintes definições:

```
1 public class Porta {
2    public void fecha() {...}
3    public void abre() {...}
4    public boolean portaAberta() {...}
5 }
6 public class PortaDeCofre extends Porta
7                          implements Monitorado{
8    public void defineHorario() {...}
9    public boolean podeAbrir() {...}
10   public void disparaAlarme() {...}
11 }
```

cada qual com interfaces condizentes com seus papeis no sistema.

Certamente essa solução não se aplica se a classe `Monitorado` não poder ser alterada devido ao seu uso no sistema. Entretanto, o exemplo acima demonstra que, para favorecer reúso, o uso de interfaces como raízes de hieraquia de tipos é um recurso poderoso.

9.8 Responsabilidade única

Princípio da Responsabilidade Única
Nunca deve haver mais de um motivo para uma classe ser alterada.

Cada responsabilidade constitui-se em uma dimensão de mudanças. Se uma classe tiver mais de uma responsabilidade, então haverá mais de uma razão para alterá-la. Uma vez alterada, todos os seus usos devem ser conferidos para verificar se ainda estão em funcionamento. Quanto mais responsabilidades houver em uma classe, mais frágil, i.e., maior risco de parar de funcionar, terá o sistema.

Para ilustrar uma violação do Princípio da Responsabilidade Única, considere o projeto de uma nova classe `Polígono`, que deve oferecer operações para cálculo de suas propriedades, como `área`, e também operações para sua exibição gráfica, como `desenhe`. Resumidamente, a classe `Polígono` poderia ser:

```
1 package A;
2 import Gráficos.*;
3 import Geometria.*;
4 public class Polígono {
5    public void desenhe() {...}
6    public double área() {...}
7    ...
8 }
```

que depende dos módulos `Gráficos`, por exemplo, para implementar `desenhe`, e `Geometria`, para `área`.

O exemplo acima é muito simples, mas permite ver as implicações da organização adotada, que força aplicações geométricas a terem que ter o módulo `Gráficos` disponível, apesar de não precisar

dele, e aplicações gráficas a terem que conviver com o módulo `Geometria`.

A consequência dessas dependências desnecessárias é que mudanças no módulo `Gráficos` têm impacto nas aplicações geométricas que usam `Polígono`, e alterações no módulo `Geometria` podem afetar as aplicações gráficas, ambas para realizar retestes, recompilação, etc. Tudo isso porque a classe `Polígono` encapsula responsabilidades distintas e não necessariamente relacionadas.

A solução é acatar o Princípio da Responsabilidade Única, separando as responsabilidades pela divisão da classe `Polígono` em dois módulos:

- Módulo `PolígonoGráfico`:

```
1 package A;
2 import Gráficos.*;
3 public class PolígonoGráfico {
4    public void desenhe() {...}
5    ...
6 }
```

- Módulo `PolígonoGeométrico`:

```
1 package A;
2 import Geometria.*;
3 public class PolígonoGeométrico {
4    public double área() {...}
5    ...
6 }
```

Nessa solução, mudanças feitas na apresentação gráfica do polígono não afetam as aplicações geométricas e vice-versa, reduzindo a conectividade do sistema.

Conclusão

Para a construção de classes de boa qualidade, que facilitam reúso, garantam baixo custo de manutenção e tornam o sistema mais extensível, a observação dos seguintes princípios de projeto é imprescindível:

1. Encapsulação de constantes: use sistematicamente constantes simbólicas no lugar de constantes literais.
2. Preferência à finalidade principal: use composição e herança conforme seus propósitos.
3. Programação para a interface: programação deve basear-se em interface e nunca na implementação de uma classe.
4. Inversão da dependência: módulos de alto nível não devem depender de módulos de nível mais baixo.
5. Resiliência: entidades de software são implementadas para nunca serem modificadas, mas apenas estendidas.
6. Substituição de Liskov: funções que usam objetos de uma classe devem ser capazes de usar objetos de suas subclasses.
7. Segregação de interfaces: interfaces mais específicas são melhores que interfaces de propósitos diversos.
8. Única responsabilidade: nunca deve haver mais de um motivo para uma classe ser alterada.

Exercícios

1. Ocultação de informação é tema recorrente em muitos dos princípios de projetos de sistemas orientados por objetos. Há alguma situação em que informação deixada pública é a melhor solução?

Notas bibliográficas

Referências sobre princípios de projeto são o livro de Peter Coad et alii [8], que tem foco no desenvolvimento de melhores aplicativos por meio de estratégias de projeto com o baseado em interfaces e uso critérios de herança, o livro de Gamma et alii [19], que aborda em seus capítulos iniciais os mecanismos que produzem arquiteturas mais simples e mais fáceis de serem entendidas, e a principal referência sobre o tema é o clássico de Bertrand Meyer [41], onde a maior parte dos princípios aqui descritos estão detalhados e muito bem justificados.

Capítulo 10

Padrões de Projeto

Desenvolvimento de software é uma atividade complexa, que demanda muita criatividade, e a de projeto de sistemas orientados por objetos é ainda mais complexa e difícil, porque artefatos frequentemente devem ser construídos desde suas bases, e não basta construir algoritmos que funcionem, deve-se também focar na construção de software extensível e de fácil manutenção, o que dificulta ainda mais o processo.

Por outro lado, soluções de implementação são recorrentes, e, com frequência, mesmas estruturas de código são repetidas para produzir soluções em diferentes contextos. Por exemplo, quando tem-se que determinar a soma dos elementos de um vetor de inteiros, um código da forma

```
int[] a = {...}
int s = 0;
for (int i:a) s += a[i];
```

é o padrão de programação adotado com frequência por projetistas experientes.

A ideia é estender essa abordagem a sistemas orientados por objetos nos quais há diversos padrões de soluções que ajudam a produzir arquiteturas comprovadamente úteis para construir software mais flexível e mais extensível, de forma que o conhecimento e reúso desses padrões reduzem custos e aumentam confiabilidade.

As ideias de reúso de projetos de software foram inspiradas no trabalho científico de Christopher Alexander [1] para a área de Arquitetura, o qual estudou meios de melhorar o processo de projeto de edifícios e áreas urbanas pelo uso disciplinado de elementos já estabelecidos para o projeto de novas edificações. Por exemplo, arcos e colunas são uma estratégia comprovada para construir edificações seguras, e a instalação de portas e janelas seguem um padrão bem estabelecido, e não há necessidade de reinvenção.

Essas ideias foram levadas para a área de desenvolvimento de software em 1987, quando Beck e Cunnigham [4] usaram as ideias de Alexander para programação com Smalltalk. Em 1990, Erich Gamma e John Vlissides, participantes de uma sessão do workshop OOPSLA'90 dirigida por Bruce Anderson, denominada *Towards an Architecture Handbook*, decidiram montar um catálogo de padrões e, com a partipação de Richard Helm e Ralph Johnson, iniciaram a compilação desse catálogo, que redundou na publicação do livro *Design Patterns* [19] em 1995.

Padrões de Projeto, hoje muito populares, trazem benefícios como o estabelecimento de vocabulário comum entre desenvolvedores, a redução da complexidade do processo de projeto de software, o aumento da confiabilidade do software decorrente do uso de arquiteturas comprovadas e da experiência acumulada na Área e a elevação do grau de reutilização de software produzido.

Os elementos essenciais de um padrão de projeto de software são:

- Nome: usado para facilitar comunicação entre desenvolvedores.
- Motivação: definição de quando usar o padrão.
- Solução: descrição das partes que constituem o padrão, com clara indicação de relacionamentos, responsabilidades e colaborações.

- Consequências: detalhamento dos prós, contras, reusabilidade e extensibilidade.

Gamma et alii classificam os padrões de projeto de sofware orientado por objetos em três categorias: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais.

Padrões de criação descrevem técnicas comuns de se organizar classes e objetos em um sistema com facilidades para criar determinados objetos. Os padrões definidos são: Singleton, Abstract Factory, Factory Method, Prototype e Builder.

Padrões estruturais descrevem técnicas comuns de se organizar classes e objetos. Os padrões definidos são: Proxy, Adapter, Bridge, Composite, Decorator, Façade e Flyweight.

Padrões comportamentais definem estratégias para modelar como objetos colaboram uns com os outros em um subsistema. Os padrões definidos são: Memento, State, Chain-of-Responsability, Command, Observer, Strategy, Template Method, Iterator, Mediator, Interpreter e Visitor.

10.1 Singleton

Há situações em que somente deve haver um objeto responsável por gerenciar recursos compartilhados, como as conexões com um banco de dados. Nesses casos, é recomendado que se tenha apenas um objeto encarregado desse processo e, deve-se, de alguma forma, assegurar que esse objeto seja único.

A solução para essa questão é o padrão de criação `Singleton`, que é construído por meio de classe com um construtor privado único. O fato de o único construtor da classe ser `private` garante que objetos somente podem ser criados dentro da classe via um de seus métodos públicos estáticos, e, assim, a unicidade do objeto

criado pode ser garantida por programação.

O padrão **Singleton** segue o esquema do seguinte exemplo:

```
1  public final class Singular {
2     private static Singular único;
3     private Singular() {}
4     public static Singular instancie() {
5        if (único == null)
6            único = new Singular();
7        return único;
8     }
9     "Estrutura de dados privada da abstração"
10    "Operações definidas pelo requisito funcional"
11 }
```

onde o especificador **final** acima é desnecessário para a semântica do programa, pois o fato de a construtora ser **private** já torna a classe final, mas essa redundância é util para deixar claro que subclasses de **Singular** não são permitidas.

Note que somente o método **Singular.instancie** está autorizado a criar objetos da classe e foi programado para criar somente o primeiro e retornar o seu endereço nos demais pedidos de criação.

O programa **Teste** a seguir imprime **Funcionou**, demonstrando que apenas um objeto foi criado nas duas chamadas de **instancie**.

```
1  public class Teste {
2     public static void main(String[] a) {
3        PrintStream o = System.out;
4        Singleton s1   = Singular.instancie();
5        Singleton s2   = Singular.instancie();
6        if (s1 != s2) o.println("Eu, hein!");
7        else o.println("Funcionou");
8     }
9  }
```

Em resumo, para construir uma classe segundo o padrão de criação `Singleton`, deve-se criá-la de forma a que:

- tenha uma única construtora, que é declarada `private`;
- tenha um método estático com a função de criar um objeto em sua primeira chamada e retornar sempre esse mesmo objeto em chamadas subsequentes.

10.2 Factory Method

Considere o método `monteX` da classe `A` definida a seguir e que cria objetos de classes `B1`, ..., `Bn` e com eles constrói um da classe `X`.

```
1  public class A {
2     ...
3     public X monteX() {
4        B1 b1 = new B1(...);
5        B2 b2 = new B2(...);
6        ...
7        Bn bn = new Bn(...);
8        X x   = new X(b1, b2, ..., bn);
9        return x;
10    }
11 }
```

O programa abaixo usa `monteX` para montar dois objetos `X`:

```
1  public class Objetos {
2     public static void main(String[] args) {
3        A a  = new A(...);  X x1 = a.monteX();
4        A b  = new A(...);  X x2 = b.monteX();
5        ...
6     }
7  }
```

Suponha agora que se deseja montar o objeto `x2` com variações em sua composição, redefinindo a estrutura de alguns de seus componentes, por exemplo, pela definição das seguintes subclasses:

```
1 class Sb1 extends B1 {...}
2 class Sb2 extends B2 {...}
3 ...
4 class Sbn extends Bn {...}
```

as quais serão usadas no lugar de `B1`, ..., `Bn` em uma nova versão de `monteX`, redefinida em uma subclasse de `A`, com o seguinte código, onde os novos objetos são apropriadamente criados:

```
1  public class Sa extends A {
2      ...
3      public X monteX() {
4          B1 b1 = new Sb1(...);
5          B2 b2 = new Sb2(...);
6          ...
7          Bn bn = new Sbn(...);
8          X x   = new X(b1, b2, ..., bn);
9          return x;
10     }
11 }
```

Essa organização resolve o problema, mas a forma como a classe `A` foi estruturada não favorece extensibilidade, porque o impacto de adaptação de `monteX` é muito extenso: todo o seu código teve que ser replicado e alterado, apesar de ter exatamente a mesma estrutura, variando apenas as classes usadas para criar os novos componentes.

E, consequentemente, o programa que usa `monteX` também teve que ser alterado para incluir a classe `Sa` e passa a ter a seguinte codificação:

```
1 public class Objetos {
2    public static void main(String[] args) {
3       A a  = new A(...);  X x1 = a.monteX();
4       A b  = new Sa(...); X x2 = b.monteX();
5       ...
6    }
7 }
```

A dificuldade revelada por essa forma de estruturar classes decorre do método clássico inerente aos mecanismos da orientação por objetos, que estabelece que alterar o comportamento de uma classe via subclasse consiste na redefinição do comportamento de métodos herdados, e nunca de comandos herdados, pois, por exemplo, não há como trocar as ocorrências de `new B1(..)` que estão em `A` por `new Sb1(...)`, os quais devem ser usados em `Sa`, sem redefinir também `monteX`.

A solução para esse problema é dada pelo padrão de criação `Factory Method` que, no caso em discussão, recomenda substituir toda ocorrência do operador primitivo `new` de criação de objetos por chamadas de métodos especializados em fabricação dos respectivos objetos.

Esses métodos especializados na fabricação de objetos podem ou não ser redefinidos em subclasses de `A`, sem que qualquer alteração precise ser feita no método `monteX` dessa classe. O papel de um método de fabricação é poder ser livremente redefinido para criar outros tipos de objetos.

Uma implementação da classe `A` de acordo com o padrão de criação `Factory Method` é apresentada a seguir, onde cada uso do operador `new` está encapsulado dentro de um método de fabricação próprio, declarado público de forma a poder ser redefinido em subclasses descendentes de `A`.

```
class A {
   public B1 crieB1(...) {return new B1(...);}
   public B2 crieB2(...) {return new B2(...);}
   ...
   public Bn crieBn(...) {return new Bn(...);}
   public X  crieX(...)  {return new X(...);}
   ...
   public X monteX( ) {
      B1 b1 = crieB1(...);
      B2 b2 = crieB2(...);
      ...
      Bn bn = crieBn(...);
      X   x = crieX(b1, b2, ..., bn);
      return x;
   }
}
```

E caso deseje-se alterar os componentes de alguns dos objetos do tipo `X` montados por `monteX`, basta criar uma subclasse, e.g., `Sa`, com as redefinições desejadas para os métodos de fabricação, enquanto o método `monteX` da classe `A` permanece inalterado, como é demonstrado a seguir.

```
class Sa extends  A {
   public B1 crieB1(...) {return new Sb1(...);}
   public B2 crieB2(...) {return new Sb2(...);}
   ...
   public Bn crieBn(...) {return new Sbn(...);}
   public X  crieX(...)  {return new X(...);}
}
```

Assim, nesse ponto, pode-se ter um programa que monta dois tipos de objetos `X`, um é definido pela classe `A`, e outro, por `Sa`, e que simplesmente altere o tipo dos componentes a ser instalados por `monteX` em objetos do tipo `X`, preservando todo o restante do código.

O programa `Objetos` abaixo monta dois objetos do tipo `X`, sendo um com os componentes `B1`, ..., `Bn` e o outro com os componentes `Sb1`, ..., `Sbn`, convivendo harmonicamente no mesmo programa, sem precisar de alterar o método `monteX` da classe `A`.

```
1 public class Objetos {
2    public static void main(String[] args) {
3       A a = new A(...);  X x1 = a.monteX();
4       A b = new Sa(...); X x2 = b.monteX();
5       ...
6    }
7 }
```

A versão da classe `A` que usa `Factory Method` é mais reusável e extensível que a versão inicialmente proposta.

Em resumo, para construir uma classe segundo o padrão criação `Factory Method`, deve-se criá-la de forma a que:

- o operador de criação de objetos `new` seja somente usado dentro de métodos cuja única função seja a de fabricar um objeto;
- toda criação desses objetos seja sistematicamente feita via esses métodos de fabricação;
- métodos de fabricação possam ser livremente redefinidos pelo mecanismo de construção de subclasses.

10.3 Abstract Factory

Para tornar o processo de fabricação de objetos mais flexível, deve-se criar uma classe separada para reunir os métodos de fabricação. Esse tipo de classe segue o padrão de criação `Abstract Factory`, e seus objetos são fábricas de objetos, conforme a definição de seus métodos de fabricação. Dessa forma, a classe cliente de uma `Abstract Factory` pode decidir em tempo de execução qual fábrica de objetos a ser usada.

Suponha a seguinte definição de uma **Abstract Factory**:

```
class Fábrica {
   public B1 crieB1(...) {return new B1(...);}
   public B2 crieB2(...) {return new B2(...);}
   ...
   public Bn crieBn(...) {return new Bn(...);}
   public X  crieX(...)  {return new X(...);}
}
```

E considere a seguinte definição de uma classe **A** que possui um método para montar objetos do tipos **X**.

```
class A {
   ...
   public X monteX(Fábrica f) {
      B1 b1 = f.crieB1(...);
      B2 b2 = f.crieB2(...);
      ...
      Bn bn = f.crieBn(...);
      X   x = f.crieX(b1, b2, ..., bn);
      return x;
   }
}
```

Uma segunda fábrica pode ser definida por **OutraFábrica**, a qual cria objetos de classes **Sb1**, **Sb2**, ..., **Sbn**, que são especializações de **B1**, **B2**, ..., **Bn**, respectivamente.

```
class OutraFábrica extends Fábrica{
   public B1 crieB1(...) {return new Sb1(...);}
   public B2 crieB2(...) {return new Sb2(...);}
   ...
   public Bn crieBn(...) {return new Sbn(...);}
}
```

O programa **Objetos** cria as fábricas e faz a montagem de dois objetos do tipo **X**, usando para cada um a fábrica desejada.

```
1 public class Objetos {
2    public static void main(String[] args) {
3       Fábrica f1 = new Fábrica();
4       Fábrica f2 = new OutraFábrica();
5       A a = new A(...); X x1 = a.monteX(f1);
6       A b = new A(...); X x2 = b.monteX(f2);
7       ...
8    }
9 }
```

O uso do padrão **Abstract Factory** torna a implementação mais legível e mais flexível. A fábrica de componente foi terceirizada, sendo comunicada diretamente ao método que dela necessita, o método **monteX**. Novas fábricas dentro da mesma hierarquia podem ser facilmente incorporadas, sem qualquer impacto na implementação da classe **A**, e a estrutura do padrão garante que somente objetos de uma família podem ser criados.

Em resumo, para construir uma classe segundo o padrão criação **Abstract Factory**, deve-se criá-la de forma a que:

- não use diretamente o operador **new** para criar os componentes que manipula;
- o operador de criação de objetos **new** seja somente usado dentro de métodos cuja única função seja a de fabricar um objeto e que esses métodos de criação sejam encapsulados em classes separadas e especializadas em criação de objetos;
- os objetos dessas classes de criação de objetos sejam passadas como parâmetros a métodos da classe cliente;
- métodos de fabricação possam ser livremente redefinidos pela definição de novas classes via mecanismo de construção de subclasses.

10.4 Prototype

O padrão de criação `Prototype` parece com o `Factory Method`, mas ao invés de criar novos objetos, fazem-se clonagens de protótipos de objetos existentes, somente conhecidos durante a execução.

Inicialmente, cria-se um objeto que encapsula protótipos de vários objetos, dos quais cópias podem ser obtidas.

A declaracão de `A` a seguir modela o uso do padrão `Prototype`, que encapsula protótipos de objetos dos tipos `B1`, `B2`, ..., `Bn`, definidos pela sua construtora ou redefinidos a qualquer momento por métodos da forma `setB`$_i$`(b`$_i$`)`, e define o método `monteX`, para montagem de objetos a partir dos componentes encapsulados:

```
1  public class A {
2     private B1 b1; private B2 b2; ...; private Bn bn;
3     public A(B1 b1, B2 b2, ..., Bn bn) {
4         this.b1 = b1; this.b2 = b2; ...; this.bn = bn;
5     }
6     public void setB1(B1 b1) {this.b1 = b1;}
7     public void setB2(B2 b2) {this.b2 = b2;}
8     ...
9     public void setBn(Bn bn) {this.bn = bn;}
10    public B1 crieB1() {return b1.clone();}
11    public B2 crieB2() {return b2.clone();}
12    ...
13    public Bn crieBn() {return bn.clone();}
14    public X monteX() {
15       B1 b1 = crieB1(); B2 b2 = crieB2();...;
16       Bn bn = crieBn();
17       X  x  = crieX(); x.inicie(b1, b2, ..., bn);
18       return x;
19    }
20    ...
21 }
```

A título de ilustração, suponha que as seguintes subclasses sejam definidas:

```
1 public class Sb1 extends B1 {...}
2 public class Sb2 extends B2 {...}
3 ...
4 public class Sbn extends Bn {...}
```

as quais são usadas pelo programa principal `Objetos` para criar os protótipos dos objetos a ser clonados e fazer a montagem de dois objetos do tipo `X`, usando em cada um os objetos desejados.

```
1  public class Objetos {
2     public static void main(String[] args) {
3        B1 b11 = new B1();  B1 b12 = new Sb1();
4        B2 b21 = new B2();  B2 b22 = new Sb2();
5        ...
6        Bn bn1 = new Bn();  Bn bn2 = new Sbn();
7        A a = new A(b11, b21, ..., bn1);
8        X x1 = a.monteX();
9        "comandos tipo a.setBij(bi), para i:1..n, j:1..2"
10       X x2 = a.monteX();
11       ...
12    }
13 }
```

Em resumo, para construir uma classe segundo o padrão criação `Prototype`, deve-se criá-la de forma a que:

- encapsule referências a todos os componentes que são usados na montagem de objetos compostos;
- essas referências sejam iniciadas, via a construtora da classe que as contém, por referências a objetos criados externamente, que atuam como protótipos;
- seja possível redefinir a qualquer momento os protótipos vinculados a essas referências;

- seus métodos de fabricação de componentes apenas fazem clonagem de protótipos.

10.5 Builder

O padrão de criação **Builder** permite separar a construção de um objeto complexo de sua representação de forma que o mesmo processo de construção possa criar diferentes representações.

Esse padrão é semelhante ao **Abstract Factory**, porque ambos criam objetos de uma dada família, mas, enquanto o primeiro retorna objetos de uma família de classes relacionadas, **Builder** constrói um objeto complexo com base em dados a ele fornecidos.

O padrão **Builder** consiste nos seguintes elementos:

- uma classe diretora, que constrói o objeto desejado, utilizando a fábrica de componentes a ela associada;
- uma família de fabricantes dos componentes usados pela classe diretora.
- classes dos objetos construídos e de seus componentes.

Suponha a seguinte definição de uma **Fábrica**, que usa as classes de componentes **B1**, **B2**, ... **Bn** e **X**:

```
1 class Fábrica {
2    public B1 crieB1(...) {return new B1(...);}
3    public B2 crieB2(...) {return new B2(...);}
4    ...
5    public Bn crieBn(...) {return new Bn(...);}
6
7    public X  crieX(B1 b1, B2 b2, ..., Bn bn) {
8       return new X(b1, b2, ..., bn);}
9 }
```

Outras fábricas podem ser definidas como subclasse de **Fábrica**, como **OutraFábrica** abaixo, a qual cria objetos de classes **Sb1**, **Sb2**, ..., **Sbn**, que são especializações de **B1**, **B2**, ..., **Bn**, respectivamente.

```
1 class OutraFábrica extends Fábrica{
2    public B1 crieB1(...) {return new Sb1(...);}
3    public B2 crieB2(...) {return new Sb2(...);}
4    ...
5    public Bn crieBn(...) {return new Sbn(...);}
6 }
```

A classe diretora do padrão consiste na definição de uma classe, como a classe **A** apresentada a seguir, que possui o método **monteX** para montar objetos do tipo **X**. Essa definição de **A** difere da usada no exemplo de uso do padrão **Abstract Factory** pelo fato de, agora, a fábrica a ser usada pelo **Builder** ser a referenciada por objetos do tipo **A** e não mais a que era passada como parâmetro a método **monteX** no caso de **Abstract Factory**.

```
1  class A {
2     Fábrica f;
3     public A(Fábrica f) {this.f = f;}
4     public X monteX() {
5        B1 b1 = f.crieB1(...);
6        B2 b2 = f.crieB2(...);
7        ...
8        Bn bn = f.crieBn(...);
9        X   x = f.crieX(b1, b2, ..., bn);
10       return x;
11    }
12 }
```

O programa **Objetos** cria as fábricas necessárias e faz a montagem de dois objetos do tipo **X**, usando para cada um a fábrica desejada.

```
1 public class Objetos {
2    public static void main(String[] args) {
3       Fábrica f1 = new Fábrica();
4       Fábrica f2 = new OutraFábrica();
5       A a = new A(f1); X x1 = a.monteX();
6       A b = new A(f2); X x2 = b.monteX();
7       ...
8    }
9 }
```

Em resumo, para construir uma classe segundo o padrão criação `Builder`, deve-se criá-la de forma a que:

- os métodos de fabricação de componentes usados sejam implementados em classes especializadas em fabricação de objetos;
- tenha uma referência a um objeto de fabricação de componentes, que deve ser iniciado pela construtora da classe;
- toda criação de objetos seja feita via essa referência encapsulada na classe.

10.6 Proxy

O padrão estrutural `Proxy` permite a um objeto chamado `Proxy` (procurador) agir como um substituto de outro. Há muitas situações nas quais a disponibilidade de um substituto é conveniente, como a possibilidade de se exibir alguma imagem provisória enquanto as imagens reais são carregadas ou então um objeto móvel deve deixar um outro em seu lugar para responder mensagens a ele dirigidas durante sua ausência.

Há situações em que o módulo-cliente não quer referenciar um objeto diretamente, mas precisa interagir com ele indiretamente e, nesse caso, um objeto procurador pode agir como intermediário entre o cliente e o objeto.

Para ilustar o papel de um `Proxy`, suponha que um sistema em operação defina a seguinte interface `Tabela`, onde `r` e `c` designam linha e coluna, respectivamente.

```
1 public interface Tabela {
2    public int getElemento(int r, int c);
3    public void setElemento(int e, int r, int c);
4    public int númeroDeLinhas();
5 }
```

Essa interface é usada para construir a classe `TabelaPadrão`, definida a seguir, na qual não há previsão de sincronizar acesso às suas linhas:

```
1 public class TabelaPadrão implements Tabela {
2    private int[] t;
3    public TabelaPadrão(int[] t) {this.t = t;}
4    public int getElemento(int r, int c) {return t(r,c);}
5    public void setElemento(int e,int r,int c){t(r,c) = e;}
6    public int númeroDeLinhas() {return t.length;}
7 }
```

Suponha também que, após essa classe ter sido colocada em operação, deseje-se alterar sua implementação para introduzir mecanismos de sincronização de acesso em nível de linhas aos elementos da tabela.

Entretanto, essas alterações não podem afetar as implementações já disponibilizadas, pois tanto a interface `Tabela` quanto a classe `TabelaPadrão` são de uso disseminado na aplicação, e, consequentemente, qualquer alteração que lhes for feita poderá gerar um impacto substantivo de manutenção em todo o sistema.

A solução desse problema é o padrão estrutural `Proxy`, que, como na vida real, propõe substituir, automaticamente, o agente de uma ação por seu procurador para executar suas operações, que,

no exemplo em discussão, deve estar devidamente equipado com recursos para administrar acessos sincronizados a linhas de tabelas.

O mecanismo de *dynamic binding* das linguagens orientadas por objetos, o qual determina em tempo de execução a versão do método a executar, é de fato um processo de delegação a terceiros de tarefas a ser efetuadas. O padrão `Proxy` é uma aplicação desse mecanismo, como mostra a seguinte definição da classe `Outorgado`:

```
1  public class Outorgado implements Tabela {
2     Tabela outorgante;
3     Object[] cadeados;
4     public Outorgado(Tabela t) {
5       outorgante = t;
6       cadeados = new Object[t.númeroDeLinhas()];
7       for (int i = 0 ; i < cadeados.length ; i++)
8            cadeados[i] = new Object();
9     }
10    public int getElemento(int r, int c) {...}
11    public void setElemento(int e, int r, int c){...}
12    public int númeroDeLinhas() {return outorgante.length;}
13 }
```

As redefinições dos métodos `getElemento` e `setElemento` para equipá-los com controle de acesso sincronizado são as seguintes:

```
1  public int getElemento(int r, int c) {
2     synchronized(cadeados[r]) {
3        return outorgante.getElemento(r,c);
4     }
5  }
6  public void setElemento(int e, int r, int c) {
7     synchronized(cadeados[r]) {
8        outorgante.setElemento(e,r,c);
9        return;
10    }
11 }
```

Pode-se, então, usar um objeto `t` do tipo `TabelaPadrão`, no qual sincronismo de acesso às suas linhas não é definido, e, quando necessário, usar também um objeto `procurador`, criado por um comando como `Outorgado procurador = new Outorgado(t)`, e que, além de executar as tarefas de `t`, usa sincronismo para acessar elementos de cada linha.

Em resumo, para construir uma solução de programação segundo o padrão estrutural `Proxy`, deve-se:

- criar uma classe como uma implementação da interface da classe da qual ela deve atuar como sua substituta;
- encapsular nessa classe uma referência a um objeto da classe a ser substituída, a qual deve ser inicializada no momento de criação de objeto da classe substituta;
- implementar na classe substituta os métodos especificados na interface com a nova semântica desejada.

10.7 Adapter

Ao construir bibliotecas, muitas vezes não é possível prever todos os contextos que seus componentes serão usados. Assim, frequentemente classes de biblioteca não podem ser usadas, porque suas interfaces não são exatamente as requeridas por uma aplicação.

Considerando que classes de bibliotecas, em geral, não podem ser alteradas, uma solução é fazer uma adaptação, sem alterar classes da biblioteca, para expandir as interfaces desejadas. O padrão estrutural `Adapter` define uma forma de fazer esse tipo de adaptação.

Para exemplificar como uma adaptação pode ser feita, suponha que a biblioteca de um sistema disponibilize as classes `A` e `B`, e que

a classe `A` faça uso de serviços definidos em `B`, conforme modelado pelo seguinte esquema de programação:

```
1  public class A {  // classe da biblioteca
2     ...
3     public void f(B b) { ...; b.g(); ...}
4  }
5  public class B {  // classe da biblioteca
6     public void g() {...}
7  }
```

A aplicação define uma nova classe `C` e deseja usá-la como o tipo do parâmetro de `A.f`, o que não é permitido pelo compilador:

```
1  public class C {  // uma nova classe da aplicação
2     public void h() {...}
3  }
4  public class Aplicação {
5     public static void main(String[] args) {
6        A a = new A();
7        B b = new B();
8        C c = new C();
9        a.f(b);
10       a.f(c); // erro de compilação
11       ...
12    }
13 }
```

O padrão estrutural **`Adapter`** contorna essa dificuldade, definindo um modelo de programa que permite passar o objeto do tipo `C` para um método que espera um objeto do tipo `B`, como tentou-se fazer na linha 10 acima, e produzir os resultados desejados.

Ele funciona de forma análoga a adaptadores de tomadas elétricas, convertendo uma interface em outra. Esses adaptadores podem ser unidirecionais ou bidirecionais.

Adaptador unidirecional

Um adaptador unidirecional de `B` em `C` deve, de um lado, funcionar como um objeto do tipo `B`, e, do outro, produzir os efeitos descritos em `C`. Em outras palavras, o adaptador unidirecional deve ser uma subclasse de `B`, e seus métodos devem agir como os seus correspondentes na classe adaptada `C`, como em:

```
1 public class Adaptador extends B {
2    private C c;
3    public Adaptador(C c) {this.c = c;}
4    public void g() {c.h();}
5 }
```

E a aplicação deve ser adaptada com o uso do adaptador:

```
1 public class Aplicação {
2    public static void main(String[] args) {
3       A a = new A();
4       B b = new B();
5       C c = new C();
6       a.f(b);
7       a.f(new Adaptador(c)); // Em vez de a.f(c)
8       ...
9    }
10 }
```

Em resumo, para adaptar unidirecionalmente objetos de uma classe `C` para ser aceitos como objetos de outra classe `B`, deve-se:

- definir uma classe adaptadora que seja subclasse de `B`;
- encapsular nessa classe o objeto a ser adaptado;
- usar métodos do objeto a ser adaptado para reimplementar os métodos herdados;
- apresentar um objeto da classe adaptadora onde os objetos adaptados forem esperados.

Adaptador bidirecional

Um adaptador bidirecional aceita como entrada um dentre dois tipos de objetos a ser adaptados, por exemplo, ele deve permitir que se use um objeto de um tipo `B` onde um de `C` for esperado e vice-versa. Isso significa que o adaptador deve implementar ambas as interfaces de `B` e de `C`.

Para ilustrar a construção desse tipo de adaptador, considere a classe `A`, onde estão destacados os métodos `f1` e `f2`:

```
1 public class A {
2     ...
3     public void f1(Ib b) { ...; b.g(); ...}
4     public void f2(Ic c) { ...; c.h(); ...}
5 }
```

onde as interfaces `Ib` e `Ic`, que têm a seguinte especificação:

```
1 public interface Ib {
2     void g();
3 }
4 public interface Ic {
5     void h();
6 }
```

são usadas para definir `B` e `C`, as quais devem ser adaptadas para seus objetos poderem ser passados a `A.f2` e `A.f1`, respectivamente:

```
1 public class B implements Ib {
2     public void g() {...};
3 }
4 public class C implements Ic {
5     public void h() {...};
6 }
```

O adaptador a seguir encapsula referências aos objetos a ser adaptados, que são fornecidos pelas suas construtoras, e implementa os métodos especificados na interfaces `Ib` e `Ic`:

```
1 public class Adaptador implements Ib, Ic {
2    private Ib b;
3    private Ic c;
4    public Adaptador(Ib b) {this.b = b;}
5    public Adaptador(Ic c) {this.c = c;}
6    public void g() {c.h();}
7    public void h() {b.g();}
8 }
```

E a aplicação a seguir usa o adaptador quando necessário:

```
1 public class Aplicação {
2    public static void main(String[] args) {
3       A  a = new A();
4       Ib b = new B();
5       Ic c = new C();
6       a.f1(b);
7       a.f1(new Adaptador(c));
8       a.f2(c);
9       a.f2(new Adaptador(b));
10      ...
11   }
12 }
```

Em resumo, para adaptar os objetos das classes `B` e `C` para que um possa ser usado no lugar do outro, deve-se:

- definir uma classe adaptadora que implemente as interfaces das classes a ser adaptadas;
- encapsular nessa classe os objetos a ser adaptados, um de cada classe;
- usar métodos dos objetos a ser adaptados para reimplementar os respectivos métodos herdados;

- apresentar um objeto da classe adaptadora onde os objetos adaptados forem esperados.

10.8 Bridge

O padrão estrutural `Bridge` define uma forma de separar a interface de uma classe de sua implementação, de forma que a implementação possa ser alterada sem que isso afete o código das classes-cliente. A título de motivaçao, suponha que uma biblioteca defina uma interface `I` e duas classes `B` e `C`:

```
1  public interface I {
2      public f();
3  }
4  public class B implements I {
5      ...
6     public f() {...}
7  }
8  public class C implements I {
9      ...
10    public f() {...}
11 }
```

Herança é um recurso que permite agregar a uma classe estruturas de dados e operações definidas em outra, sendo assim uma forma de vincular a definição de uma classe à implementação de outra. Esse mecanismo é o aplicado no seguinte esquema de programa, onde a classe `A` incorpora atributos e operações de `B`.

```
1  class A extends B {
2      ...
3     public g() { ...; f(); ... }
4  }
```

E um usuário de `A` pode fazer indiretamente uso da operação `f` herdada de `B`, via operação `g` de `A`, da seguinte forma:

```
1 public void Aplicação {
2    public static void main(String[] args) {
3       A a = new A();
4       a.g();
5    }
6 }
```

Essa solução, a princípio, pode parecer satisfatória, mas tem um problema de extensibilidade, porque, caso deseje-se também incorporar à classe `A` atributos e operações definidas em outras classes da família da interface `I`, herança não pode mais ser usada, pois Java somente admite herança simples, a qual já foi consumida pela extensão de `B`.

A solução nesse caso é trocar herança por composição: em vez de estender `B`, a classe `A` deve definir privadamente uma referência a objeto do tipo definido pela interface `I`, e essa referência deve ser devidamente iniciada antes de qualquer uso de operações do objeto referenciado. Uma nova classe `A`, segundo esse padrão, pode ser definida da seguinte forma:

```
1 public class A {
2    I b;
3    public A(I b) {this.b = b;}
4    public setI(I b) {this.b = b;}
5    ...
6    public g() { ...; b.f(); ... }
7 }
```

a qual permite dinamicamente associar objetos do tipo `I` a objetos do tipo `A`.

E novo programa `Aplicação`, que agora vincula a `A` objetos das classes `B` e `C`, passa a ter o seguinte código:

```
1 public void Aplicação {
2    public static void main(String[] args) {
3       A a1 = new A(new B());
4       a1.g();  // opera segundo B.f()
5       A a2 = new A(new C());
6       a2.g();  // opera segundo C.f()
7    }
8 }
```

Em resumo, para desvincular abstrações de suas implementações, de forma a poder variá-las de forma independente, por meio do padrão `Bridge`, deve-se;

- estruturar a classe usuária das abstrações para incorporar, via composição, uma referência encapsulada a objeto da abstração que se deseja utilizar, a qual é caracterizada por uma interface comum;
- essa referência deve ser devidamente iniciada com os objetos desejados, podendo seu valor ser alterado dinamicamente.

10.9 Composite

Há estruturas de objetos em que seus componentes podem ser objetos atômicos ou então objetos compostos por outros componentes de mesma natureza, formando uma coleção hierárquica, como uma árvore que possui nodos, os quais podem ser folhas, i.e., objetos atômicos, ou nodos, que por sua vez podem ser compostos de outros nodos e folhas.

Como nessa organização há duas categorias de componentes, os primitivos e os compostos, as classes usuárias dessas estruturas devem ser capazes de diferenciar esses componentes a fim de dar-lhes o devido tratamento, mas o melhor seria que todos eles tivessem a mesma interface, liberando os clientes da tarefa de diferenciá-los.

Uma solução para isso é dar a todos os nodos da árvore, inclusive as folhas, a mesma estrutura, diferenciando nodos internos de folhas pela presença ou não de filhos.

O padrão estrutural `Composite` segue exatamente essa estruturação, definindo uma organização que permite tratar ambos os componentes do tipo folha e do tipo nodo de maneira uniforme. Isso torna a implementação mais extensível, conforme ilustrado pela seguinte classe `A`, que caracteriza objetos que têm um valor intrínseco e uma coleção, possivelmente vazia, de outros objetos do mesmo tipo `A`, de onde destaca-se a operacão `total`, a qual trata todos os componentes de `A` de maneira uniforme:

```
1  public class A {
2     private int valor;
3     private Vector<A> filhos;
4     public A(int valor) {
5        this.valor = valor;
6        filhos = new Vector<A>();
7     }
8     public void insira(A a) {filhos.addElement(a);}
9     public void remova(A a) {filhos.removeElement(a);}
10    public int valor() {return valor;}
11    public int total() {
12       int s = valor;
13       for (int i = 0; i < filhos.size(); i++)
14            s += filhos.elementAt(i).valor();
15       return s;
16    }
17 }
```

Suponha que os nodos das árvores possam ser objetos de qualquer tipo descendente de `A`, como `B`, `C` ou `D`, definidos a seguir, onde, para simplificar o exemplo, supõe-se que os métodos por eles herdados de `A` não são redefinidos:

```
1 class B extends A {public B(int valor) {super(valor);...}}
2 class C extends A {public C(int valor) {super(valor);...}}
3 class D extends A {public D(int valor) {super(valor);...}}
```

A seguinte aplicação monta uma árvore com nodos do tipo definido pela classe `A` acima definida:

```
1  public class Aplicação {
2     public static void main(String[] args) {
3        A a  = new A(100);
4        A b1 = new B(50); a.insira(b1);
5        A b2 = new B(50); a.insira(b2);
6        A c1 = new C(20); b1.insira(c1);
7        A c2 = new C(20); b1.insira(c2);
8        A d1 = new D(10); b2.insira(d1);
9        int total = a.total();
10       System.out.println("Soma total = " + total);
11    }
12 }
```

Em resumo, para criar uma estrutura contendo valores atômicos e objetos compostos por outros componentes de mesma natureza, formando uma coleção hierárquica, na qual todos os elementos são tratados uniformente, pode-se usar uma estrutura de programa definida pelo padrão `Composite`. Para isso, deve-se:

- criar uma classe que encapsule os atributos, os quais consistem em um valor atômico e uma coleção de outros objetos do mesmo tipo dessa classe;
- definir construtoras e métodos especializados para iniciar os atributos dos objetos dessa classe;
- definir operações para atuar uniformemente sobre todos os objetos da coleção encapsulada.

10.10 Decorator

Muitas vezes, deseja-se adicionar responsabilidades a objetos individualmente, estendendo o código de algumas de suas operações, sem afetar outros objetos da classe, de forma a poder usar, em um mesmo programa, as diversas versões de seus objetos. Por exemplo, dada uma classe `A` como a declarada a seguir

```
1 class A {
2     ...
3     public void f() {"corpo de f"}
4 }
```

deseja-se acrescentar ao método `f` uma nova ação `c1`, o que pode ser feito via o mecanismo de herança, como em

```
1 class A1 extends A {
2     ...
3     public void f() {super.f(); "código de c1"}
4 }
```

E se for desejado acrescentar à operação `A.f` uma outra ação `c2`, uma nova extensão de `A` resolve a questão:

```
1 class A2 extends A {
2     ...
3     public void f() {super.f(); "código de c2"}
4 }
```

produzindo o efeito desejado.

Dessa forma, em um mesmo programa, pode-se ter

```
A a  = new A();  a.f();  // executa corpo f
A a1 = new A1(); a1.f(); // executa corpo f; c1
A a2 = new A2(); a2.f(); // executa corpo f; c2
```

Herança parece ser o mecanismo natural para concretizar esse tipo de adaptação de código de operações de uma classe, mas é um recurso pouco flexível, pois, por exemplo, se for desejado acrescentar a um objeto da hierarquia de `A.f` a combinação de códigos `c1;c2`, dever-se-á criar uma nova extensão de `A` da forma:

```
1 class A3 extends A {
2     ...
3     public void f() {super.f(); "código c1"; "código c2"}
4 }
```

Em uma aplicação em que muitas novas responsabilidades e suas combinações devam ser adicionadas a determinadas operações de uma classe, o uso puro de herança para esse efeito pode ter um efeito explosivo no número de subclasses.

O padrão estrutural `Decorator` surge com uma solução alternativa de alta flexibilidade, permitindo que um objeto assuma dinamicamente responsabilidades, possivelmente aproveitando adições de funcionalidade anteriormente definidas.

Nesse padrão de programação, deve-se criar uma classe, denominada decoradora, encarregada de centralizar toda adição e combinação de funcionalidades desejadas. Essa classe deve encapsular a classe a ser decorada e serve de base para criar subclasses com as extensões desejadas. Por exemplo, a classe `D`, a seguir, é uma decoradora da classe `A`:

```
1 class D extends A {
2     private A a;
3     public D(A a) {this.a = a;}
4     public void f() {a.f()}
5 }
```

Assim, para acrescentar uma funcionalidade `c1` ao método `A.f`, define-se uma extensão segundo o seguinte modelo:

```
class E1 extends D {
   public E1(A a) {super(a);}
   public void f() {
      super.f();
      "código adicional c1"
   }
}
```

E para uma segunda extensão de funcionalidade `c2`, pode-se ter:

```
class E2 extends D {
   public E2(A a) {super(a);}
   public void f() {
      super.f();
      "código adicional c2"
   }
}
```

A seguinte aplicação, que usa as classes `A`, `D`, `E1` e `E2`, mostra como decorações podem ser dinamicamente combinadas:

```
class Aplicação {
   public static void main(String[] args) {
      private A a = new();
      a.f(); // executa corpo de f
      A e1 = new E1(a);
      e1.f(); // executa corpo de f; c1
      A e2 = new E2(a);
      e2.f(); // executa corpo de f; c2
      A e3 = new E2(e1);
      e3.f(); // executa corpo de f; c1; c2
   }
}
```

Oberserve que, mais uma vez, o uso de composição é mais adequado do que herança como mecanismo de estender funcionalidades. Nesse padrão, o decorador contém o objeto a ser decorado, tem

sua interface e a função de repassar ao objeto decorado o pedido de serviço e, possivelmente, adicionar-lhe ações e evitar a necessidade de se ter um grande número de subclasses para suportar as combinações de extensões desejadas.

Em resumo, para criar uma estrutura que permita alta flexibilidade na adição de funcionalidades a operações de uma dada classe, pode-se usar uma estrutura de programa definida pelo padrão `Decorator`. Para isso, deve-se:

- criar uma classe decoradora que encapsula o objeto a ser decorado e que tem sua interface;
- redefinir as operações que se deseja estender para atuar sobre o objeto encapsulado;
- estender a classe decoradora para definir as funcionalidades a ser acrescentadas ao objeto encapsulado.

10.11 Façade

É prática comum em programação o encapsulamento de detalhes de implementação e seu uso via interfaces mais enxutas e de fácil uso. Por exemplo, usa-se uma função via referência ao seu nome sem que seja preciso ter acesso direto aos comandos e expressões que a implementam.

Em sistemas orientados por objetos permite-se que um objeto – chamado de fachada – forneça uma interface simples para um subsistema formado por um agregado de outros objetos, de forma que a fachada reúna o conjunto de responsabilidades do sistema e encapsule sua complexidade. Um exemplo da vida real análogo é o objeto pedal do acelerador de um automóvel que é fachada para todo o subsistema de aceleração do carro.

O padrão estrutural `Façade` define uma forma de criar fachadas de forma a simplificar a interface de um conjunto de classes de

um subsistema. Para motivar a aplicação do conceito de fachada, considere um subsistema com três pacotes, `P1`, `P2` e `P3`, definidos da seguinte forma:

- Pacote `P1`:

```
1 package P1;
2 public class A {
3    private int x;
4    ...
5    public int getX(){return x;}
6    public void setX(int x) {this.x = x;}
7 }
```

- Pacote `P2`:

```
1 package P2;
2 import P1.*;
3 public class B {
4    private int y;
5    ...
6    public int getY() {return y;}
7    public void setY(A a) {y = a.getX() + 1;}
8 }
```

- Pacote `P3`:

```
1 package P3;
2 import P2.*;
3 public class C {
4    private int z;
5    ...
6    public int getZ() {return z;}
7    public void setZ(B b) {z = b.getY() + 1;}
8 }
```

Um usuário desses pacotes é exemplificado pelo seguinte trecho de programa:

```
1 import P1.*; import P2.*; import P3.*;
2 public class Usuário1V1 {
3    public static void main(String[] args) {
4       A a = new A(); a.setX(100);
5       B b = new B(); b.setY(a);
6       C c = new C(); c.setZ(b);
7       System.out.print(c.getZ());
8    }
9 }
```

E um segundo usuário do mesmo subsistema poderia ser:

```
1 import P1.*; import P2.*; import P3.*;
2 public class Usuário2V1 {
3    public static void main(String[] args) {
4       A a = new A(); a.setX(1000);
5       B b = new B(); b.setY(a);
6       C c = new C(); c.setZ(b);
7       System.out.print(c.getZ());
8    }
9 }
```

Para criar uma fachada para esse subsistema, considere o pacote:

```
1  package F;
2  import P1.*; import P2.*; import P3.*;
3  public class Façade {
4     public int resultado(int x) {
5        A a = new A(); a.setX(x);
6        B b = new B(); b.setY(a);
7        C c = new C(); c.setZ(b);
8        return c.getZ();
9     }
10 }
```

Uma versão mais simples, usando o conceito do padrão `Façade`, do `Usuário1V1` é a seguinte:

```
1 import F.*;
2 public class Usuário1V1 {
3    public static void main(String[] args) {
4       Façade f = new Façade();
5       System.out.print(f.resultado(100));
6    }
7 }
```

E o mesmo se aplica ao `Usuário2V1`, cuja interface com o subsistema dos pacotes `P1`, `P2` e `P3` resume-se na chamada ao método `Façade.resultado`:

```
1 import F.*;
2 public class Usuário2V1 {
3    public static void main(String[] args) {
4       Façade f = new Façade();
5       System.out.print(f.resultado(1000));
6    }
7 }
```

Em resumo, para simplificar a interface de subsistemas complexos, pode-se criar fachadas para usos específicos. Para isso, segundo o padrão `Façade`, deve-se:

- criar uma classe, a fachada, com a definição de uma operação que implementa a interação desejada com os objetos do subsistema e devolve o resultado obtido;
- usar, sempre que possível, essa operação no lugar de fazer acesso direto ao subsistema.

10.12 Flyweight

Há situações em que se tem que alocar um grande número de pequenos objetos, os quais representam dados muito semelhantes, por exemplo, cada caractere em um processador de texto pode ser representado por um pequeno objeto que identifica seu fonte, estilo, tamanho e cor.

Uma organização de texto como essa pode ocupar muito espaço de memória e possivelmente demandar um custo maior de processamento. O padrão estrutural `Flyweight` deve ser usado para reduzir o número desses pequenos objetos, eliminando repetições desnecessárias, pela identificação de atributos intrínsecos, que caracterizam univocamente cada objeto, e os extrínsecos, que poderiam ser removidos dos objetos e informados via parâmetros de suas operações. Esse tipo de parametrização permite reduzir substancialmente o número de classes que implementam esses pequenos objetos, que passam a ser chamados de `Flyweight`[1].

Para ilustrar o uso do padrão `Flyweight`, suponha um conjunto Y bastante grande de classes simples, aqui denominadas `Bik`, para `i`:1..N, `k`:1..$M_{\texttt{i}}$, onde N e $M_{\texttt{i}}$ podem ser grandes números, e cada uma das classes `Bik` é uma pequena estrutura aqui modelada pela seguinte declaração:

```
1 class Bik {
2    private int i = Ii;
3    private int e = Ek;
4    public int f() {return i + e;}
5 }
```

na qual `Ii` e `Ek` são constantes, e o atributo `i` denota uma informação intrínseca, compartilhada por todas as suas instâncias, e

[1]Nomeados em analogia a lutadores da categoria peso-mosca.

`e` é uma informação extrínseca, particular de cada objeto do tipo `Bik`. Assim, para cada `i`:1..N, tem-se $M_{\texttt{i}}$ pequenas classes, uma para cada `Ek`.

Se os atributos extrínsecos das classes `Bik`, para todo `i` e `k`, for delas removidos, e transformados em parâmetros, a coleção Y dessas classes reduz-se-ia a `N` classes da forma:

```
1 class Bi {
2    private int i = I1;
3    public int f(int e) {return i + e;}
4 }
```

A seguir, para simplificar o exemplo, apenas as classes `B1`, `B2` e `B3` de `Y` serão focadas, e todas implementadas, sem perda de generalidade, segundo a seguinte estrutura:

```
1 class B1 {
2    private int i = 10;
3    public int f(int e) {return i + e;}
4 }
5 class B2 {
6    private int i = 10;
7    public int f(int e) {return i + e;}
8 }
9 class B3 {
10   private int i = 100;
11   public int f(int e) {return i + e;}
12 }
```

Observe que a remoção dos atributos extrínsecos das pequenas e numerosas classes, transformando-os em parâmetros de certas operações, trazem economia de espaço, pois reduzem o número de objetos a ser compartilhados.

O padrão estrutural `Flyweight` é semelhante ao padrão estrutural `Prototype`, o qual cria clones repetidos de objetos, enquanto

que, no **Flyweight**, os objetos também são criados uma única vez, mas, a partir daí, são reusados, ou seja, **Prototype** sempre cria novas instâncias e depois faz clonagens, e **Flyweight** cria novas instâncias e depois as compartilha.

A classe **A** a seguir recebe e encapsula um conjunto de instâncias, uma de cada classe **B1**, **B2** e **B3**, e disponibiliza operações para compartilhar os objetos encapsulados, na forma recomendada pelo padrão **Flyweight**:

```
1  class A {
2     private B1 b1;
3     private B2 b2;
4     private B3 b3;
5     public A(B1 b1, B2 b2, B3 b3) {
6         this.b1 = b1; this.b2 = b2; this.b3 = b3;
7     }
8     public B1 crieB1() {return b1;}
9     public B2 crieB2() {return b2;}
10    public B3 crieB3() {return b3;}
11 }
```

A classe **X** abaixo é uma usuária de **A**. Seu papel é obter objetos do conjunto encapsulado por **A**, ativar as operações de cada um passando-lhes valores extrínsecos e executar um dado processamento com valores retornados pelos objetos recuperados:

```
1  class X {
2     public int g(A a) {
3        B1 b1 = a.crieB1();
4        B2 b2 = a.crieB2();
5        B3 b3 = a.crieB3();
6        int  x  = b1.f(1) + b2.f(10) + b3.f(100);
7        return x;
8     }
9  }
```

O programa principal, apresentado a seguir, completa o exemplo de uso do padrão `Flyweight`, e tem a função de criar pequenos objetos das classes `B1`, `B2` e `B3`, armazená-los em um objeto da classe `A` e, depois, disparar o processamento definido pela operação `g` definida pela classe `X`.

```
1  public class UsaFlyweight {
2     public static void main(String[] args) {
3        B1 b1 = new B1();
4        B2 b2 = new B2();
5        B3 b3 = new B3();
6        A a = new A(b1, b2, b3);
7        X x = new X();
8        int r = x.g(a);
9        System.out.println("Resultado: " + r);
10    }
11 }
```

Em resumo, para programar segundo o padrão estrutural `Flyweight` deve-se:

- identificar e transformar os atributos extrínsecos em parâmetros das operações apropriadas da classes *peso-mosca*;
- criar um objeto de cada uma dessas classes à medida que forem necessários ou antecipadamente, conforme a aplicação;
- implementar nessas classes operações para compartilhar os objetos criados.

10.13 Memento

O padrão comportamental `Memento` implementa uma estrutura de programa em que se permite um objeto armazenar seus estados de modo a que possa ser restaurado para um dos estados anteriores,

como em uma operação de desfazer. A estrutura também oferece a possiblidade de recuperar o histórico dos estados do objeto. Esse padrão trabalha com três tipos de objetos:

- objeto originador: objeto cujo estado deve ser lembrado;
- objeto memento: objeto que implementa o processo de salvamento de estado do objeto originador para formar história de estados;
- objeto zelador: armazena o histórico, i.e., tem referências para todos os estados associados ao originador.

O objeto originador `A`, que é parte do exemplo a seguir de uma implementação do padrão `Memento`, define um estado interno aqui simbolizado pelo atributo `int estado`, ao qual referem-se as operações `getEstado` e `setEstado`.

```
1  import java.util.*;
2  class A {
3     private int estado;
4     public void setEstado(int estado) {this.estado=estado;}
5     public int  getEstado() {return estado;}
6     public Memento getMemento() {return new Memento(this);}
7     public void restaureEstado(Memento m) {
8        this.estado = m.estado;
9     }
10    public static class Memento {
11       private int estado;
12       public Memento(A a) {
13          this.estado = a.estado;
14       }
15    }
16 }
```

A classe interna `Memento` provê a operação `A.getMemento` para salvar o estado corrente do objeto originador, e, quando ativada, produz um objeto contendo o estado do originador salvo.

Observe que por ser interna ao originador, a classe `Memento` tem livre acesso ao estado do originador, ainda que declarado privativo.

O programa a seguir cria um objeto originador de lembranças, mostra o salvamento de quatro estados pelos quais ele passou, faz a restauração do originador para cada um dos estados salvos e imprime o resultado:

```
1  public class UsaMemento {
2     public static void main(String[] args) {
3        ArrayList<A.Memento> lembranças =
4                                  new ArrayList<A.Memento>();
5        A a = new A();
6        a.setEstado(10);
7        lembranças.add(a.getMemento());
8        a.setEstado(20);
9        lembranças.add(a.getMemento());
10       a.setEstado(30);
11       lembranças.add(a.getMemento());
12       a.setEstado(40);
13       lembranças.add(a.getMemento());
14
15       System.out.println("Lembranças:" );
16       for (int i=0; i<lembranças.size();i++) {
17            a.restaureEstado(lembranças.get(i));
18            System.out.print(" Estado " + i + "->" +
19                                          a.getEstado());
20       }
21       System.out.println();
22    }
23 }
```

O resultado impresso por `UsaMemento` é o seguinte:

```
Lembranças:
Estado 0->10 Estado 1->20 Estado 2->30 Estado 3->40
```

10.14 State

O padrão comportamental `State` modela um objeto que define um contexto de execução, encapsulando o estado de um objeto sobre o qual operações serão executadas. O objeto contexto reage em conformidade com o seu estado corrente.

Esse padrão de projeto usa uma interface que descreve o comportamento de cada estado de um objeto contextualizador e deve garantir as seguintes condições para o objeto que dá o contexto:

- objeto contextualizador deve ter seu comportamento alterado se seu estado interno mudar;
- cada estado gera um comportamento independente;
- a encapsulação de novos estados não deve alterar o comportamento relativos aos estados existentes.

Um exemplo bem simples de uso do padrão `State` é apresentado a seguir a partir da seguinte interface:

```
1 interface I {void imprima(String nome);}
```

a qual é implementada pelas classes `A` e `B`:

```
1  class A implements I {
2     public void imprima(String nome) {
3        System.out.println("Dra " + nome);
4     }
5  }
6  class B implements I {
7     public void imprima(String nome) {
8        System.out.println("Dr " + nome);
9     }
10 }
```

A classe `C` a seguir modela o contexto que define quando objetos da família da interface `I` são usados. Isso é feito por meio do atributo privado **`estado`**, que detém uma referência para o executor de operações definidas na classe `C`.

```
1 class C {
2    private I estado;
3    public C() {estado = new A();}
4    public void setEstado(I i) {estado = i;}
5    public void imprima(String nome){estado.imprima(nome);}
6 }
```

O programa **`UsaContexto`** a seguir testa a implementação do padrão apresentado:

```
1 public class UsaContexto {
2    public static void main(String[] args) {
3       C contexto = new C();
4       contexto.setEstado(new A());
5       contexto.imprima("Maria");
6       contexto.setEstado(new B());
7       contexto.imprima("José");
8       contexto.setEstado(new A());
9       contexto.imprima("Mariana");
10      contexto.setEstado(new B());
11      contexto.imprima("Pedro");
12      System.out.println();
13   }
14 }
```

imprimindo

```
Dra Maria
Dr José
Dra Mariana
Dr Pedro
```

Em resumo, para construir uma estrutura de programa segundo o padrão **`State`**, deve-se:

- definir a interface dos objetos executores de ações desejadas;
- declarar as classes dos executores desejados;
- criar uma classe de contextualização, cujos objetos devem encapsular referência ao objeto executor desejado, i.e., encapsular o estado corrente de execução das ações definidas na interface;
- configurar o objeto contextualizador conforme desejado.

10.15 Chain-of-Responsability

O padrão comportamental `Chain-of-Responsability` estabelece uma forma de determinar, durante a execução, o objeto de uma família que irá tratar uma dada mensagem. Os objetos são programados para tratar as mensagens recebidas ou, se estiverem ocupados com outras tarefas, passá-la para o próximo objeto de uma cadeia de possíveis tratadores.

Usualmente, o cliente que envia mensagens tem a responsabilidade de especificar o objeto que dará tratamento à sua mensagem. Com o padrão `Chain-of-Responsability`, o cliente apenas organiza os tratadores de mensagens em uma lista circular e, a partir daí, apenas submete seu pedido a essa cadeia de tratadores, e todo o restante do processamento é feito automaticamente.

Os tratadores são objetos de uma mesma família, da qual qualquer um é aceitável para tratar a mensagem do cliente. A questão é que, em um dado momento, o tratador que recebe a mensagem pode não estar disponível para tratá-la, e, nesse caso, um outro membro de sua cadeia de responsabilidades deve ser tentado. Essa cadeia deve ser uma lista circular para que mais de uma tentativa por tratador seja possível no caso de indisponibilidade temporária de todos os participantes.

A classe `H` apresentada a seguir ilustra um tratador de mensagens

na qual destaca-se a função `H.livre` da linha 7 que informa se o tratador está ou não disponível para tratar a mensagem recebida, fazendo-se um sorteio aleatório por meio do objeto `random`.

```
1  import java.util.Random;
2  class H {
3     private final static Random random = new Random();
4     private static int idCount = 1;
5     private int id = idCount++;
6     private H próximo;
7     private boolean livre() {return random.nextInt(3)==0;}
8     private H() {próximo = null;}
9     private H(H próximo) {this.próximo = próximo;}
10    public static H constróiCadeia(int n) {
11       H p, q, r;
12       r = p = new H();
13       p.próximo = p;
14       for (int i = 1; i < n; i++) {
15          q = r;
16          r = new H(q.próximo);
17          q.próximo = r;
18       }
19       return p;
20    }
21    public void trate(int m) {
22       if (livre()) {
23           System.out.println(" H" + id + " tratou " + m);
24       } else {
25           System.out.println(" H" + id + " ocupado ");
26           próximo.trate(m);
27       }
28    }
29 }
```

Os métodos construtores de `H` foram declarados privados apenas para forçar que a cadeia de tratadores seja somente criada pelo método estático público `H.constróiCadeia`, garantindo assim a

consistência do padrão, mas isso não é um dos seus requisitos.

O método público `H.trate` tem a função de tratar a mensagem recebida ou então a repassá-la a um outro tratador na cadeia de responsabilidades.

O programa a seguir constrói uma cadeia com cinco tratadores de mensagens e envia-lhe quatro pedidos de tratamento para ser processados por um dos participantes dessa cadeia.

```
1 public class UsaChain {
2    public static void main(String[] args) {
3       H p = H.constróiCadeia(5);
4       for (int m = 1; m < 5; m++) {
5           System.out.println("Mensagem #" + m + ":");
6           p.trate(m);
7       }
8    }
9 }
```

produzindo a seguinte impressão:

```
Mensagem #1:
 H1 ocupado
 H2 tratou 1
Mensagem #2:
 H1 ocupado
 H2 ocupado
 H3 ocupado
 H4 ocupado
 H5 ocupado
 H1 tratou 2
Mensagem #3:
 H1 tratou 3
 Mensagem #4:
 H1 tratou 4
```

Em resumo, para se construir um código conforme o padrão `Chain-of-Responsability`, no qual os tratadores de mensagens são organizados em uma cadeia de responsabilidades, deve-se:

- instalar os tratadores em uma lista circular;
- identificar o critério de disponibilidade de cada tratador;
- programar a operação de processamento de mensagens para decidir se o objeto corrente as trata ou as repassa ao próximo tratador na cadeia de responsabilidades.

10.16 Command

Há situações em que é necessária uma longa série de comandos `if`s para identificar o objeto responsável para executar uma determinda ação. E muitas vezes, deseja-se poder desfazer ações executadas bem como enfileirar ações a ser executadas posteriormente.

O padrão comportamental `Command` contribui para resolver essas situações, simplificando o processo de identificação do objeto executor da ação desejada, e recomendando encapsular as ações relativas a um objeto no próprio objeto, a fim de desacoplar-lhe daqueles que fazem a invocação dessas ações.

Assim, clientes desses objetos não precisam conhecer as ações a eles associadas a ser executadas por cada operação.

Para motivar o uso desse padrão, suponha as seguintes declarações de um grande número de classes `A1`, `A2`, ..., `An`:

```
1 class A1 {"atributos e operações de A1"}
2 class A2 {"atributos e operações de A2"}
3 ...
4 class An {"atributos e operações de An"}
```

que são usadas para criar os objetos `a1`, `a2`, ..., `an`, conforme mostrado a seguir, dentre os quais, no fluxo de execução, um deles é dinamicamente identificado por meio de uma série de comandos `if`s ou por m comando `switch` para executar o conjunto de ações a ele associado:

```
1  public class Chaveamento1 {
2     public static void main(String[] args) {
3        A1 a1 = new A1(...);
4        A2 a2 = new A2(...);
5        ...
6        An an = new An(...);
7        ...
8        Object objeto = "ai, para algum i:1..n"
9        ...
10       if (objeto == a1) ação1(...);
11       else if (objeto == a2) ação2(...);
12       ...
13       else if (objeto == an) açãon(...);
14       ...
15    }
16 }
```

Essa solução tem um problema de extensibilidade, porque o ninho de comandos `if` da linha 10 acima é dependente do número e tipo dos objetos criados. Qualquer alteração nessa configuração poderá ter alto impacto de manutenção do código. Outros problemas têm a ver com operações de desfazer ações ou de enfileirá-las para tratamento posterior.

O padrão **Command** soluciona esses problemas, recomendando encapsular as ações associadas a objetos das classes **A1**, ..., **An** em suas respectivas classes. Para isso, deve-se definir a seguinte interface, que especifica uma única operação **execute**:

```
1  public interface Command {void execute(...);}
```

e fazer as classes **A1**, ..., **An** implementarem essa interface, cada uma encapsulando em sua implementação do método **execute** as ações associadas aos respectivos objetos dessas classes que forma mencionados na aplicação **Chaveamento1** acima, conforme ilustra o seguinte código:

```
1  class A1 implements Command {
2     "atributos e operações de A1"
3     public void execute(...) {ação1(...);}
4  }
5  class A2 implements Command {
6     "atributos e operações de A2"
7     public void execute(...) {ação2(...);}
8  }
9  ...
10 class An implements Command {
11    "atributos e operações de An"
12    public void execute(...) {ação n(...);}
13 }
```

E para concluir, a seleção da ação a ser executada é feita automaticamente na linha 10 da aplicação **Chaveamento2** abaixo pelo mecanismo de *dynamic binding* do ambiente de execução.

```
1  public class Chaveamento2 {
2     public static void main(String[] args) {
3        Command a1 = new A1(...);
4        Command a2 = new A2(...);
5          ...
6        Command an = new An(...);
7        ...
8        Object objeto = "ai, para algum i:1..n"
9        ...
10       objeto.execute(...);
11    }
12 }
```

Nesse esquema, as ações associadas a ser executadas passam a estar fisicamente vinculada aos objetos da família **Command**, facilitando a introdução de operações de desfazer ou enfileirar ações, pois basta armazenar os objetos a elas associados.

Em resumo, usa-se o padrão `Command` quando:

- deseja-se evitar um ninho de `if`s para escolher a ação a ser executada, porque o número de teste pode ser elevado e pela dificuldade de se acrescentar ou eliminar testes da lista;
- deseja-se especificar, enfileirar e executar pedidos em diferentes momentos, por exemplo, criando filas para seu atendimento;
- precisa-se implementar operações desfazer.

Para isso deve-se:

- criar uma interface como `Command`;
- fazer as classes de interesse implementarem essa interface, encapsulando adequadamente as ações pertinentes;
- disparar a operação `execute` de `Command` para ativar as ações associadas ao objeto selecionado.

10.17 Observer

O padrão comportamental `Observer` modela uma arquitetura em que um objeto, dito observado, notifica automaticamente uma lista de objetos a ele associados, denominados observadores, sobre suas mudanças de estado. Para isso, o observado encapsula a lista de seus observadores, que pode ser dinamicamente alterada. E um observador pode observar simultamente vários observados.

Esse padrão define um relacionamento 1-para-n entre objetos, tal que, quando o estado de um objeto sob observação for modificado, todos seus observadores são notificados automaticamente.

Os observadores a ser notificados a respeito das mudanças no estado do objeto observado devem implementar a interface definida a seguir, a qual especifica o método a ser chamado pelo objeto observado para avisar cada observador que houve mudança de estado e informar o caminho de acesso a seu estado.

```
1 public interface Observador {
2    void atualize(Observado observado);
3 }
```

Os objetos a ser observados devem estender a classe `Observado` para herdar a estrutura de administração da lista de observadores associados, a qual possui operações para inserir observadores na lista, retirá-los da lista e, principalmente, notificar todos os observadores dessa lista as mudanças de estado.

```
1  class Observado {
2     private List <Observador> observadores =
3                                  new LinkedList<Observador>();
4     public void adicione(Observador observador) {
5        observadores.add(observador);
6     }
7     public void remova(Observador observador) {
8        observadores.remove(observador);
9     }
10    public void notificaTodos() {
11       Observador observador;
12       Iterator<Observador> it = observadores.iterator();
13       while (it.hasNext()) {
14          observador = it.next();
15          observador.atualize(this);
16       }
17    }
18 }
```

A operação `atualize` de cada observador pode precisar ter acesso aos valores que foram alterados no estado do observado. Assim, as classes que estenderem `Observado` devem prover operações públicas para dar aos observadores acesso aos campos de interesse. Por exemplo, a classe `Termômetro` a seguir descreve o comportamento de termômetros a ser observados e disponibiliza a operação

`getValor`, que informa o valor do campo privado `temperatura`. Observe que a operação `setValor` abaixo simula mudanças de estado do objeto corrente e executa a operacão `notificaTodos` herdada de `Observado`.

```
1  class Termômetro extends Observado {
2     private double temperatura;
3     public void setValor(double temperatura) {
4        this.temperatura = temperatura;
5        notificaTodos();
6     }
7     public double getValor() {
8        return this.temperatura;
9     }
10 }
```

Nesse contexto, os observadores de termômetros, definidos a seguir pelas classes `Celsius` e `Fahrenheit`, implementam a interface `Observador`, cada um definindo a semântica desejada para a operação `atualize`:

```
1  class Celsius implements Observador {
2     public void atualize(Observado observado) {
3        double v = ((Termômetro) observado).getValor();
4        System.out.println("Celsius = " + v);
5     };
6  }
7  class Fahrenheit implements Observador {
8     public void atualize(Observado observado) {
9        double v = ((Termômetro) observado).getValor();
10       System.out.println("Fahrenheit = " + (1.8*v+32));
11    }
12 }
```

A semântica desses observadores, que é propositalmente simples, apenas exibe o valor da temperatura lida do termômetro e a converte para a escala de valores pertinente.

A aplicação `UsaObservadores` a seguir cria um termômetro com dois observadores e provoca duas mudanças de estado no termômetro e produz a saída:

```
Celsius = 40.0
Fahrenheit = 104.0
Celsius = 30.0
Fahrenheit = 86.0
```

```
1  public class UsaObservadores {
2     public static void main(String[] args) {
3        Termômetro t = new Termômetro();
4        Observador celsius = new Celsius();
5        Observador fahrenheit = new Fahrenheit();
6        t.adicione(celsius);
7        t.adicione(fahrenheit);
8        t.setValor(40);
9        t.setValor(30);
10    }
11 }
```

Em suma, para usar o padrão `Observer`, deve-se:

- definir uma classe denominada `Observado` dotada de infraestrutura para administrar uma lista de objetos observadores;
- implementar a classe dos observados como uma extensão da classe `Observado`;
- implementar os observadores com a interface `Observador`, a qual especifica a operação `atualize` para prover a comunicação entre observados e observadores.

10.18 Strategy

O padrão comportamental `Strategy` é muito semelhante ao padrão `State`. A diferença é que o primeiro encapsula um algoritmo, enquanto o segundo, as informações de estado. Um bom exemplo do

uso do padrão **Strategy** é o uso da interface **LayoutManager** para encapsular a estratégia de disposição dos componentes gráficos em um painel, por meio das classes **FlowLayout**, **BorderLayout** e **GridLayout**, que implementam **LayoutManager**, cada uma usando um algoritmo próprio. Nesse modelo, o objeto **Container** tem seu leiaute definido pela operação **setLayout**, como em:

```
Container container = getContentPanel();
container.setLayout(new FlowLayout);
```

A motivação do padrão **Strategy** é poder definir uma família de algoritmos, encapsular cada um e torná-los intercambiáveis sem afetar seus clientes. Recomenda-se seu uso sempre que muitas classes relacionadas diferem apenas no comportamento, geralmente implementável por diferentes variantes de um mesmo algoritmo.

A estrutura desse padrão é delineada pelo seguinte esquema de programa, onde **class X {...}** é uma classe qualquer, e supõe-se que a aplicação faça uso de n estratégias denominadas **E1**, **E2**, ..., **En**, todas da família da interface **Estratégia**, que especifica o algoritmo **calculeX**, e cujas implementações definem as diversas estratégias.

```
1  interface Estratégia {X calculeX(...);}
2
3  class E1 implements Estratégia {
4     public X calculeX(...) {...}
5  }
6  class E2 implements Estratégia {
7     public X calculeX(...) {...}
8  }
9  ...
10 class En implements Estratégia {
11    public X calculeX(...) {...}
12 }
```

O objeto usuário das estratégias deve encapsular a referência uma das estratégias que usa internamente e deve permitir que essa estratégia possa ser alterada pela operação **setEstratégia**:

```
1  public class A {
2     private Estratégia estratégia;
3     public void setEstratégia(Estratégia estratégia) {
4        this.estratégia = estratégia;
5     }
6     public void f(...){
7        ...
8        X x = estratégia.calculeX(...);
9        ...
10    }
11    ...
12 }
```

E o programa principal cria as estratégias desejadas, o objeto usuário e define a estratégia, dentre as definidas, a ser usada pelas operações do usuário do tipo **A**.

```
1  public class UsaEstratégia {
2     public static void main(String[] args) {
3       E1 e1 = new Estratégia(...);
4       E2 e2 = new Estratégia(...);
5       ...
6       En en = new Estratégia(...);
7       A a = new A(...);
8       ...
9       a.setEstratégia(Ei); // para um dado Ei, i:1..n
10      a.f(...);
11      ...
12    }
13 }
```

Em resumo, para construir uma estrutura de programa segundo o padrão **Strategy**, deve-se:

- definir uma interface comum para os objetos implementadores de estratégias;
- declarar as classes dos implementadores de estratégias;
- criar uma classe que usa essas estratégias, cujos objetos encapsulam referência ao objeto implementador dessas estratégias;
- configurar o objeto usuário conforme desejado.

10.19 Template Method

O padrão comportamental `Template Method` define o esqueleto de um algoritmo, deixando alguns de seus passos para ser definidos em subclasses. Isso permite a redefinição de certos aspectos de um algoritmo sem mudar sua estrutura. Há alguma semelhança com o padrão `Strategy`, mas, no `Template Method`, todos os objetos compartilham um único algoritmo, que é definido pela superclasse.

Um exemplo clássico de uso desse padrão é a seguinte implementação do algoritmo de exibição de um texto em uma página, o qual executa sua tarefa em três partes: exibição de cabeçalho, do texto principal e de rodapé:

```
1  abstract class Página {
2     public final void exibePágina(String texto) {
3        exibeCabeçalho();
4        System.out.println(texto);
5        exibeRodapé();
6     }
7     public abstract void exibeCabeçalho();
8     public void exibeRodapé() {
9        System.out.println("Rodapé ascii");
10    }
11 }
```

O método `exibePágina` tem um corpo não-redefinível, assegurado pelo atributo `final`, mas a ação `exibeCabeçalho` ainda

precisa ser definida, e **exibeRodapé** pode ser redefinida em subclasses, como foram feitas no seguinte código:

```
1  class Ascii extends Página {
2     public void exibeCabeçalho() {
3        System.out.println("Cabeçalho ascii");
4     }
5  }
6  class Html extends Página {
7     public void exibeCabeçalho() {
8        System.out.println("Cabeçalho html");
9     }
10    public void exibeRodapé() {
11       System.out.println("Rodapé html");
12    }
13 }
```

O programa **UsaTemplate2** a seguir usa um mesmo algoritmo para exibir duas páginas com formatos distintos:

```
1  public class UsaTemplate2 {
2     public static void main(String[] args) {
3        String texto = "Corpo do texto";
4        Página página1 = new Ascii();
5        Página página2 = new Html();
6        página1.exibePágina(texto);
7        System.out.println("-------");
8        página2.exibePágina(texto);
9     }
10 }
```

Em resumo, para construir uma estrutura de programa segundo o padrão **Template Method**, deve-se:

- definir uma classe abstrata com o algoritmo desejado;
- declarar as subclasses dessa classe abstrata que definem ou redefinem métodos chamados pelo algoritmo definido;

- criar os objetos dessas subclasses para o acionamento do algoritmo.

10.20 Iterator

O padrão comportamental `Iterator` permite que objetos tenham acesso a objetos individuais de qualquer estrutura de dados, sem conhecer seu comportamento ou como a estrutura está armazenada.

Tipicamente, iteradores são recursos para acessar, um a um, os elementos de um objeto do tipo contêiner ou coleção. Para isso, associam a objetos de uma coleção a noção de objeto corrente que pode ser recuperado sequencialmente.

As principais operações de um objeto iterador são:

- `void primeiro()`: posiciona o iterador no primeiro elemento da coleção;
- `void próximo()`: avança a posição corrente para o próximo elemento;
- `T corrente()`: retorna a referência do elemento corrente;
- `boolean terminou()`: informa se há ou não mais elementos a ser inspecionados.

Todo iterador deve ter acesso a representação do objeto contêiner do qual ele recupera objetos. Com o objetivo de preservar encapsulação, usualmente implementa-se a classe do iterador com uma classe não-estática aninhada na classe do objeto contêiner, ao qual adiciona-se a operação `iterador` para retornar o objeto iterador associado.

O exemplo de uso do padrão `Iterator` a seguir trata da implementação de um conjunto de inteiros, definido pela classe `Inteiros`, cujos elementos podem ser percorridos de mais de uma forma.

A classe **Inteiros**, que possui as operações **tamanho**, **insira**, **remove** e **iterador**, é definida da seguinte forma:

```
1  class Inteiros {
2       private int s[], último;
3       public Inteiros(int m) {
4            if (m < 1) m = 1;
5            último = -1; s = new int[m] ;
6       }
7       public class Iterador {
8          private int corrente;
9          public void primeiro() {corrente = 0;}
10         public void próximo() {++corrente;}
11         public int  corrente() throws Exception {
12            if (corrente <= último) return s[corrente];
13            else throw new Exception();
14         }
15         public boolean terminou(){return corrente>último;}
16      }
17      public Iterador iterador() {return new Iterador();}
18      public int tamanho() {return último + 1;}
19      public void insira(int x) throws Exception {
20         if (último >= s.length) throw new Exception();
21         for (int i = 0; i < último; i++) {
22              if (x == s[i]) return;
23         }
24         s[++último] = x;
25      }
26      public void remove(int x) {
27         int i = 0;
28         while (i < s.length && s[i] != x) ++i;
29         if (i < s.length) --último;
30         while (i < s.length-1) {s[i++] = s[i];}
31      }
32 }
```

onde merecem destaque as linhas 12 e 15, que fazem acesso à re-

presentação encapsulada do tipo `Inteiros`.

O programa `UsaIterador` a seguir exemplifica como iteradores sobre um conjunto do tipo `Inteiros` são criados e usados:

```
1  public class UsaIterador {
2    public static void main(String[] args) throws Exception{
3       int x, y, n = 10;
4       Inteiros s = new Inteiros(n);
5       for (int i = 1; i <= n ; i++) {s.insira(i);}
6       Inteiros.Iterador a = s.iterador();
7       Inteiros.Iterador b = s.iterador();
8       for (a.primeiro(); !a.terminou(); a.próximo()) {
9           x = a.corrente();
10          for (b.primeiro(); !b.terminou(); b.próximo()) {
11             y = b.corrente();
12             if (x == y) {
13                 System.out.println(x + " repetido!");
14             }
15          }
16      }
17   }
18 }
```

Em resumo, para construir uma estrutura de programa que dá acesso, segundo o padrão `Iterator`, aos componentes de uma coleção de objetos, deve-se:

- definir uma classe pública não-estática interna à classe da coleção de objetos com as operações-padrão de um iterador;
- acrescentar à classe da coleção uma operação para criar e retornar um objeto dessa classe interna.

10.21 Mediator

O padrão comportamental `Mediator` é perfeitamente exemplificado pelo sistema de comunicação das torres de controle de voo de

aeroportos, que impõe que toda troca de mensagens seja feita de forma centralizada entre aviões e torre em vez de avião para avião.

Em software, o padrão `Mediator` define que um objeto centralizador encapsule um conjunto de outros objetos, chamados de colegas[2], que se interagem, e cuida de encaminhar a cada um deles as mensagens que lhes forem pertinentes, via uma estabelecida interface de comunicação.

Dessa forma, mediadores promovem um fraco acoplamento entre objetos, impedindo que eles se interajam diretamente e, por isso, permitem que a comunicação entre os objetos seja de fácil modificação.

A classe `Colega`, que define os aspectos comuns dos objetos que desejam se comunicar, tem seguinte estrutura:

```
1  abstract class Colega {
2     private Mediador mediador;
3     private String id;
4     public Colega(String id, Mediador m) {
5        this.id = id;
6        this.mediador = m;
7     }
8     public String id() {return this.id;}
9     public void notifica(String mensagem) {
10      mediador.notifica(mensagem, this);
11    }
12    public abstract void processe(String mensagem);
13 }
```

onde destaca-se o fato de que um objeto do tipo `Colega` encapsula a referência de seu mediador de comunicação e que gera notificações enviando mensagens via o seu mediador. O método abstrato `processe` destina-se a dar processamento individualizado a mensagens por cada subclasse de `Colega`.

[2]No original: colleague

A partir da classe abstrata **Colega** pode-se definir as categorias de objetos que participam do processo de comunicação via mediação centralizada. Respeitado o fato que devem estender **Colega**, essas categorias de objetos podem ser livremente definidas, mas para simplificar o presente exemplo, estão todas declaradas com uma mesma estrutura da seguinte forma:

```
1  class Colega1 extends Colega {
2     public Colega1(String id, Mediador m) {super(id,m);}
3     public void processe(String msg) {
4        System.out.println("  " + id() + " recebeu " + msg);
5     }
6  }
7  class Colega2 extends Colega {
8     public Colega2(String id, Mediador m) {super(id,m);}
9     public void processe(String msg) {
10       System.out.println("  " + id() + " recebeu " + msg);
11    }
12 }
13 class Colega3 extends Colega {
14    public Colega3(String id, Mediador m) {super(id,m);}
15    public void processe(String msg) {
16       System.out.println("  " + id() + " recebeu " + msg);
17    }
18 }
19 class Colega4 extends Colega {
20    public Colega4(String id, Mediador m) {super(id,m);}
21    public void processe(String msg) {
22       System.out.println("  " + id() + " recebeu " + msg);
23    }
24 }
```

O mediadores da comunicação entre objetos **Colegas** são modelados pela classe **Mediador**, que encapsula a lista de todos os colegas que devem participar do processo de mediação.

Mediadores devem ser preparados para receber notificações com-

postas de uma mensagem e da referência ao objeto que a gerou e providenciar seu envio a todos os demais colegas.

```
1  class Mediador {
2     private List <Colega> colegas=new LinkedList<Colega>();
3     public void adicione(Colega colega) {
4        colegas.add(colega);
5     }
6     public void remova(Colega colega) {
7        colegas.remove(colega);
8     }
9     public void notifica(String mensagem, Colega origem) {
10       Colega colega;
11       System.out.print("Mediador recebeu " + mensagem);
12       System.out.println(" de " + origem.id());
13       Iterator<Colega> it = colegas.iterator();
14       while (it.hasNext()) {
15          colega = it.next();
16          if (colega != origem) colega.processe(mensagem);
17       }
18    }
19 }
```

O programa `UsaMediador` a seguir mostra a comunicação entre quatro colegas:

```
1  public class UsaMediador {
2     public static void main(String[] args) {
3        Mediador m = new Mediador();
4        Colega c1 = new Colega1("c1",m); m.adicione(c1);
5        Colega c2 = new Colega2("c2",m); m.adicione(c2);
6        Colega c3 = new Colega3("c3",m); m.adicione(c3);
7        Colega c4 = new Colega4("c4",m); m.adicione(c4);
8        c1.notifica("abc"); c2.notifica("def");
9        c3.notifica("ghi"); c4.notifica("jkl");
10    }
11 }
```

`UsaMediador` cria um mediador e quatro objetos `Colegas`, cada um em sua própria categoria e compartilhando um mesmo mediador, e simula uma pequena troca de mensagens, produzindo a seguinte saída:

```
Mediador recebeu abc de c1
  c2 recebeu abc
  c3 recebeu abc
  c4 recebeu abc
Mediador recebeu def de c2
  c1 recebeu def
  c3 recebeu def
  c4 recebeu def
Mediador recebeu ghi de c3
  c1 recebeu ghi
  c2 recebeu ghi
  c4 recebeu ghi
Mediador recebeu jkl de c4
  c1 recebeu jkl
  c2 recebeu jkl
  c3 recebeu jkl
```

Em resumo, para construir uma estrutura de programa que administra a comunicação entre objetos segundo o padrão `Mediator`, deve-se:

- definir uma classe do mediador com recursos para encapsular uma coleção de objetos, denominados colegas, que desejam se comunicar via o mediador;
- definir as classes dos colegas, cuidando que mensagens sejam sempre dirigidas ao mediador, o qual deve retransmiti-las aos objetos colegas que participam do processo.

10.22 Interpreter

O padrão comportamental `Interpreter` descreve como definir a representação da gramática de uma pequena linguagem e imple-

mentar seu interpretador com base na árvore de sintaxe abstrata construída manualmente.

O uso de uma gramática permite organizar a estrutura da linguagem que se quer implementar e ajuda na construção manual da árvore de sintaxe abstrata de sentenças dessa linguagem.

Em geral, essas pequenas linguagens expressam termos que envolvem variáveis e valores que podem ser administrados por uma tabela similar a da seguinte implementação, a qual associa cadeias de caracteres, ou seja nomes de variáveis, a valores inteiros:

```
1  class Tabela {
2     private Map<String, Integer> tabela =
3                          new HashMap<String,Integer>();
4     private Iterator<Entry<String,Integer>> iterador;
5     private Map.Entry<String,Integer> par;
6     public int valor(String variável) {
7        return tabela.getOrDefault(variável,0);
8     }
9     public Tabela insira(String variável, int valor) {
10       tabela.put(variável,valor);
11       return this;
12    }
13    public void imprima() {
14       iterador = tabela.entrySet().iterator();
15       while (iterador.hasNext()) {
16          par = iterador.next();
17          System.out.print(
18              par.getKey() + " = " + par.getValue() + " ");
19       }
20    }
21    public void imprimaln() {
22       imprima();
23       System.out.println();
24    }
25 }
```

Para construir uma árvore de sintaxe abstrata, o padrão comportamental `Interpreter` recomenda que, a partir da gramática da linguagem a ser implementada, adote-se a seguinte metodologia de criação das interfaces e classes vinculadas ao seu interpretador, na qual, quando se referenciam termos da gramática, letras maiúsculas do início do alfabeto denotam símbolos não-terminais, letras minúsculas do fim são símbolos terminais:

- **Regra 1:** Para cada gramática, defina uma interface, e.g., denominada `Gramática`, para servir de interface raiz para os demais tipos a ser criados, conforme a seguinte definição genérica:
  ```
  interface Gramática {T interprete(Tabela t);}
  ```
  onde `T` é o tipo do resultado desejado para a operação indicada, `Tabela` define o mapeamento entre variáveis e valores. Aplicações mais específicas podem requerer a inclusão de outras operações nessa interface.
- **Regra 2:** Para cada não-terminal presente na gramática dada, e.g., `A`, criar uma nova interface com o nome desse símbolo, estendendo a interface raiz `Gramática`, como em:
  ```
  interface A extends Gramática {}
  ```
- **Regra 3:** Para cada ocorrência de um não-terminal, e.g., `B`, em produção da forma `A ::= B`, alterar a definição da interface associada a `B`, acrescentando à sua lista de interfaces estendidas o nome da interface `A`, produzindo:
  ```
  interface B extends A1, A2, ..., An, A {}
  ```
- **Regra 4:** Enxugar as definições de interfaces, eliminando possíveis redundâncias nas suas listas de tipos estendidos.
- **Regra 5:** Para cada lado direito de produção, criar uma nova classe que implemente a interface associada ao não-terminal do respectivo lado esquerdo. Essa classe deve encapsular referências aos objetos cujos tipos são as interfaces associadas

aos não-terminais presentes no lado direito considerado. Por exemplo, para uma produção como `C ::= AxBy`, onde `A` e `B` são não-terminais, e `x` e `y`, símbolos terminais, tem-se:

```
class CAxBy implements C {
   private A a;
   private B b;
   String x;
   CAxBy(A a, B b, String x) {
      this.a = a; this.b = b; this.x = x;
   }
   public T interprete(Tabela t); {
      ... a.interprete(t) ... x ... y ...
      ... b.interprete(t) ... return ...;
   }
}
```

Na classe acima, `CAxBy` é um nome único inventado, e merece destaque a declaração de campos privados relativos a símbolos não-terminais ou terminais que compõem o lado direito da produção e que os terminais envolvidos ajudam na definição das ações da operação `interprete`. Se um mesmo lado direito ocorrer em diferentes produções, deve-se criar uma nova classe para cada uma dessas ocorrências.

Para exemplificar uma aplicação do padrão `Interpreter`, considere a seguinte gramática de uma linguagem de expressões aritméticas:

```
Exp :: Variável | Constante |  Exp "+" Exp
     | Exp "*" Exp | "(" Exp ")"
Variável  ::= nome
Constante ::= número
```

onde *nome* e *número* são símbolos terminais que denotam, respectivamente, o nome de uma variável e um valor inteiro.

Pela regra 1, tem-se a interface:

```
interface Gramática {int interprete(Tabela t);}
```

Pela regra 2, os não-terminais identificados são `Exp`, `Variável` e `Constante`, que são modelados pelas interfaces:

```
interface Exp extends Gramática {}
interface Variável extends Gramática {}
interface Constante extends Gramática {}
```

Pela regra 3, devido às produções

```
Exp ::= Variável
Exp ::= Constante
```

as definições das interfaces `Variável` e `Constante` devem ser alteradas para:

```
interface Variável extends Gramática, Exp {}
interface Constante extends Gramática, Exp {}
```

Pela regra 4, as interfaces definidas pelas regras anteriores podem ser reescrita na seguinte forma mais simplificada:

```
interface Gramática {int interprete(Tabela tabela);}
interface Exp extends Gramática {}
interface Variável extends Exp {}
interface Constante extends Exp {}
```

Pela regra 5, novas classes com os nomes `Cvariavel`, `Csoma`, `Cconstante`, `CMultiplica`, `Cgrupo`, `Nome` e `Número` devem ser criadas para modelar lados direitos de produção da seguinte forma:

- O lado direito de `Exp ::= Variável` é modelado por:

```
class Cvariável implements Exp {
   private Variável v;
   public Cvariável(Variável v){this.v = v;}
   public int interprete(Tabela t) {
      return v.interprete(t);
   }
}
```

- O lado direito de `Exp ::= Constante` é modelado por:

```
class Cconstante implements Exp {
   private Constante c;
   public Cconstante(Constante c) {this.c = c;}
   public int interprete(Tabela t) {
      return c.interprete(t);
   }
}
```

- O lado direito de `Exp ::= Exp "+" Exp` é modelado por:

```
class Csoma implements Exp {
   private Exp exp1;
   private Exp exp2;
   public Csoma(Exp exp1, Exp exp2){
      this.exp1 = exp1; this.exp2 = exp2;
   }
   public int interprete(Tabela t) {
      return exp1.interprete(t) + exp2.interprete(t);
   }
}
```

- O lado direito de `Exp ::= Exp "*" Exp` é modelado por:

```
class Cmultiplica implements Exp {
   private Exp exp1;
   private Exp exp2;
   public Cmultiplica(Exp exp1, Exp exp2) {
      this.exp1 = exp1;
      this.exp2 = exp2;
   }
   public int interprete(Tabela t) {
      return exp1.interprete(t) * exp2.interprete(t);
   }
}
```

- O lado direito de `Exp ::= "(" Exp ")"` é modelado por:

```
class Cgrupo implements Exp {
   private Exp exp;
   public Cgrupo(Exp exp) {this.exp = exp;}
   public int interprete(Tabela t) {
```

```
        return exp.interprete(t);
    }
}
```

- O lado direito de `Variável ::=` *nome* é modelado por:

```
class Nome implements Variável {
    private String nome;
    public Nome(String nome){this.nome = nome;}
    public int interprete(Tabela t) {
        return t.valor(nome);
    }
}
```

- O lado direito de `Constante ::=` *número* é modelado por:

```
class Número implements Constante {
    private int valor;
    public Número(int valor) {this.valor = valor;}
    public int interprete(Tabela t) {return valor;}
}
```

O *parsing* manual da expressão `(1 + a) * (b + 4)`, usando as classes sintáticas definidas acima, está codificado no seguinte trecho de programa, que constrói a árvore de sintaxe abstrata dessa expressão:

```
Constante x = new Número(1);
Variável  a = new Nome("a");
Variável  b = new Nome("b");
Constante y = new Número(4);
Exp exp1 = new Csoma(x,a);
Exp exp2 = new Csoma(b,y);
Exp exp  = new Cmultiplica(exp1,exp2);
```

A construção da tabela de variáveis e a avaliação da expressão representada pelo objeto `exp`, a qual armazena a árvore de sintaxe abstrata calculada acima, são dadas por:

```
Tabela tabela = new Tabela();
tabela.insira("a",2).insira("b",4);
int resultado = exp.interprete(tabela);
```

O programa de teste a seguir define um mapeamento de variáveis a seus valores, por meio do objeto **tabela**, faz a montagem da árvore de sintaxe abstrata de uma expressão aritmética específica e executa a sua avaliação por meio da operação **interprete** definida na interface raiz **Gramática**:

```
1  public class UsaInterpretador {
2     public static void main(final String[] args) {
3       Tabela tabela = new Tabela();
4       System.out.println("Expressão: (1 + a) * (b + 4)");
5       tabela.insira("a",2).insira("b",4);
6       tabela.imprimaln();
7       Constante x = new Número(1);
8       Variável  a = new Nome("a");
9       Variável  b = new Nome("b");
10      Constante y = new Número(4);
11      Exp exp1 = new Csoma(x,a);
12      Exp exp2 = new Csoma(b,y);
13      Exp exp  = new Cmultiplica(exp1,exp2);
14      int resultado = exp.interprete(tabela);
15    }
16 }
```

e produz o seguinte resultado:

```
Expressão: (1 + a) * (b + 4)
a = 2 b = 4
Resultado 24
```

10.23 Visitor

O padrão comportamental **Visitor** permite que se definam novas operações para atuar sobre os elementos de um objeto contêiner sem alterar a classe dos elementos sobre os quais elas são aplicáveis, encapsulando as ações a ser executadas, e respeitando o Princípio Aberto-Fechado.

Para ilustrar os benefícios do uso desse padrão, considere o seguinte programa, que define uma classe `A` com duas operações denominadas `ação1` e `ação2`.

```
1  interface I {
2     public void ação1();
3     public void ação2();
4  }
5  class A implements I {
6      public void ação1() {
7          System.out.println("ação1 de A");
8      }
9      public void ação2() {
10         System.out.println("ação2 de A");
11     }
12 }
13
14 public class UsaVisitor1 {
15    public static void main(String[] args) {
16       I[] c = {new A(), new A(), new A(), new A()};
17       for (int i = 0; i < c.length ; i++) {
18           c[i].ação1();
19           c[i].ação2();
20       }
21    }
22 }
```

O programa principal acima cria uma coleção `c` de objetos do tipo `A`, e sobre cada um deles aplicam-se as operações definidas na classe `A`.

Suponha agora que também se deseja aplicar sobre os objetos armazenados em `c` uma nova operação, e.g., `ação3`. Para esse fim, deve-se alterar a interface `I` para incluir o protótipo de `ação3` e incluir na classe `A` a definição dessa nova operação, como mostra o seguinte código, onde destacam-se em vermelho as modificações realizadas.

```
interface I {
   public void ação1();
   public void ação2();
   public void ação3();
}
class A implements I {
   public void ação1() {
      System.out.println("ação1 de A");
   }
   public void ação2() {
      System.out.println("ação2 de A");
   }
   public void ação3() {
      System.out.println("ação3 de A");
   }
}
public class UsaVisitor2 {
   public static void main(String[] args) {
      I [] c = {new A(), new A(), new A(), new A()};
      for (int i = 0; i < c.length ; i++) {
         c[i].ação1();
         c[i].ação2();
         c[i].ação3();
      }
   }
}
```

Essa solução viola o Princípio Aberto-Fechado, porque a classe `A` teve que ser modificada para implementar o novo requisito.

Para contornar essa inconveniência, pode-se usar o padrão de programação `Visitor`, que permite construir estruturas de classes que suportam as alterações desejadas sem violar o citado princípio.

O primeiro passo é remover as operações em foco da classe `A`, colocando-as numa estrutura de programa, na qual declara-se uma nova classe para cada operação removida de `A`, mantendo-as dentro

da hierarquia de uma nova interface chamada `Tarefa`.

```
interface Tarefa {public void ação(A a);}
class Ação1 implements Tarefa {
    public void ação(A a) {
        System.out.println("ação1 de A");
    }
}
class Ação2 implements Tarefa {
    public void ação(A a) {
        System.out.println("ação2 de A");
    }
}
```

As operações removidas de `A` são substituídas por um único método denominado `execute`, conforme mostra o seguinte código:

```
interface I {void execute(Tarefa t);}
class A implements I {
    public void execute(Tarefa t) {t.ação(this);}
}
```

no qual aciona-se a operação `ação`, conforme o tipo dinâmico do parâmetro `t` fornecido. E o novo programa principal torna-se:

```
class UsaVisitor3 {
   public static void main(String[ ] args) {
      I [ ] c = {new A( ), new A( ), new A(), new A()};
      Tarefa t1 = new Ação1();
      Tarefa t2 = new Ação2();
      for (int i = 0; i < c.length ; i++) {
            c[i].execute(t1); c[i].execute(t2);
      }
   }
}
```

Para alterar o programa acima para incluir a nova operação `ação3`, basta criar uma nova classe derivada de `Tarefa`, e.g., `Ação3`, conforme destacado em vermelho no código a seguir.

```
1  interface Tarefa {public void ação(A a);}
2  class Ação1 implements Tarefa {
3      public void ação(A a) {
4          System.out.println("ação1 de A");
5      }
6  }
7  class Ação2 implements Tarefa {
8      public void ação(A a) {
9          System.out.println("ação2 de A");
10     }
11 }
12 class Ação3 implements Tarefa {
13     public void ação(A a) {
14         System.out.println("ação3 de A");
15     }
16 }
17 interface I {void execute(Tarefa t);}
18 class A implements I {
19     public void execute(Tarefa t) {t.ação(this);}
20 }
21 class UsaVisitor4 {
22    public static void main(String[] args) {
23      I [ ] c = new A( ), new A( ), new A(), new A();
24      Tarefa t1 = new Ação1();
25      Tarefa t2 = new Ação2();
26      Tarefa t3 = new Ação3();
27      for (int i = 0; i < c.length ; i++) {
28         c[i].execute(t1); c[i].execute(t2);
29         c[i].execute(t3);
30      }
31    }
32 }
```

Com o padrão `Visitor`, o Princípio Aberto-Fechado foi devidamente respeitado. A extensão efetuada no comportamento de `A` não demandou qualquer alteração no seu código, tendo sido suficiente apenas acrescentar uma nova classe para definir a operação `ação3`.

Conclusão

As ideias de reúso de projetos de software são derivadas de trabalho científico de Christopher Alexander [1] para a área de Arquitetura, o qual estudou meios de melhorar o processo de projeto de edifícios e áreas urbanas pelo uso disciplinado de elementos já estabelecidos para o projeto de novas edificações. Por exemplo, arcos e colunas são uma estratégia comprovada para construir edificações seguras, e a instalação de portas e janelas seguem um padrão bem estabelecido, e não há necessidade de reinvenção.

Essas ideias foram levadas para a área de desenvolvimento de software em 1987, quando Beck e Cunnigham [4] usaram as ideias de Alexander para programação com Smalltalk. Em 1990, Erich Gamma e John Vlissides, participantes de uma sessão do workshop OOPSLA'90 dirigida por Bruce Anderson, denominada *Towards an Architecture Handbook*, decidiram montar um catálogo de padrões e, com a partipação de Ricard Helm e Ralph Johnson, iniciaram a compilação desse catálogo, que redundou na publicação do livro *Design Patterns* [19] em 1995.

Padrões de projeto representam um grande avanço nas técnicas de estruturar software de qualidade, altamente reusável e de baixo custo de manutenção.

Exercícios

1. Analise os padrões de projeto `Singleton`, `Factory Method` e `Composite` sob o ponto de vista do Princípio da Resiliência ou Aberto-Fechado.

Referências bibliográficas

Há muitos livros-texto de boa qualidade sobre Padrões de Projeto. Com certeza, a principal referência é o livro de Erich Gamma, Helm Johnson and John Vlissides, *Design Patterns: Elements of Reusable Objet-Oriented Software*[19], publicado em 1995.

Outro texto de boa qualidade sobre esse tema é o livro de James W. Cooper, *Java Design Patterns - A Tutorial* [9], publicado em 2000.

A maior parte dos exemplos aqui apresentados foram inspirados nessas referências bibliográficas.

Capítulo 11

Estilo de Programação

Em programação orientada por objetos, o foco diretor do processo de criação de programas são os objetos que o sistema manipula e não as ações de cada um, as quais têm a função de atualizar o estado do objeto que as executa e de prover informações relevantes sobre ele.

Essa visão de um sistema tem efeito direto na estruturação de programas, pois no lugar de se preocupar com o fluxo da informação ao longo do processo de execução do sistema, o importante é o efeito que cada passo desse processo causa no estado dos objetos que o compõem, habilitando cada um a prover a informação ou resultado relevante quando solicitado.

Este capítulo discute formas de estruturação de programas orientados por objetos, aqui chamadas de estilo, de forma privilegiar diversos atributos ou requisitos de qualidade, como simplicidade, correção, modularidade, clareza e reusabilidade.

11.1 Efeito colateral em funções

Métodos de classe que retornam `void`, também chamados de procedimentos, devem produzir algum efeito colateral no ambiente de execução, mudando valores associados a parâmetros de chamada ou de variáveis não-locais.

Uma expressão tem efeito colateral quando sua avaliação, além de produzir um valor, altera o estado do ambiente de execução, mudando valores de variáveis ou de campos em seu escopo.

Linguagens de programação imperativas, incluindo as orientadas por objetos, normalmente permitem esse estilo de programação, que, via de regra, pode ser considerado de baixa qualidade, pois prejudica enormemente a compreensão do código implementado, como mostra o seguinte trecho de programa:

```
1  ...
2  int[] a = new int[20];
3  int i = 0;
4  a[0] = 0;
5  a[1] = 1;
6  a[i++] = i = a[i++];
7  if (i == 0)
8       System.out.print("A");
9  else System.out.print("B");
10 ...
```

E o mesmo pode ser dito em relação a efeito colateral em funções, que, ao extrapolar sua obrigação de simplesmente retornar um valor, prejudica a clareza do código, como ilustra o seguinte trecho de programa, do qual é impossível dizer o que será impresso sem uma análise do código das funções `getInt` e `f`.

```
1  ...
2  if (getInt() + getInt() == 2*getInt())
3       System.out.print("A ");
4  else System.out.print("B ");
5  if (x + f(x)  ==  f(x) + x)
6       System.out.print("C ");
7  else System.out.print("D ");
8  ...
```

Há, contudo, situações em que a tradição considera aceitável o uso de efeito colateral em funções, como na seguinte manipulação de geradores de números randômicos:

```
randômico.defineSemente(semente);
x = randômico.próximoNúmero();
```

onde a função `próximoNúmero` retorna um novo valor, possivelmente distinto, a cada chamada, como é de se esperar de um gerador de números aleatórios. Mesmo nesse caso, o uso de efeito colateral poderia ser evitado, conforme o seguinte trecho de programa:

```
randômico.defineSemente(semente);
randômico.gerePróximo();
x = randômico.valor();
```

onde substitui-se a função `próximoNúmero` usada anteriormente pelo procedimento `gerePróximo` e a função associada `valor`, assim separando a questão de retornar o número gerado da de gerá-lo.

Em linguagens orientadas por objetos, como Java, o efeito colateral vinculado a objetos pode ser causado por atribuições de valores a campos dos objetos. Isso pode ser feito diretamente quando esses campos são públicos ou via chamadas de métodos. Atribuição direta a campos deve ser evitada a todo custo em nome do respeito ao contrato da classe envolvida, devendo-se privilegiar o uso de métodos para provocar mudanças de estado do objeto corrente.

Uma observação mais atenta permite dizer que todo objeto tem dois estados: um abstrato, que faz parte de seu contrato com seus usuários, e um concreto, que de fato armazena o conjunto de valores de seus componentes. O relacionamento entre essas duas visões de um mesmo objeto é ilustrada pelo homomorfismo da Fig. 11.1, onde:

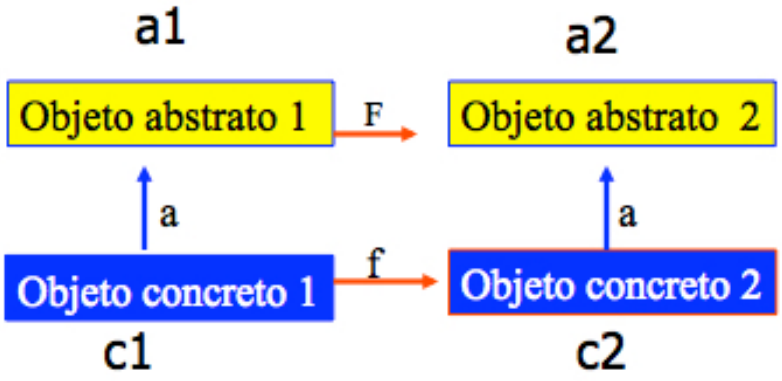

Figura 11.1 Estado Concreto X Estado Abstrato

- `F` representa uma operação no objeto abstrato, i.e., a operação percebida por seu usuário;
- `f` representa uma implementação de `F` considerando o estado concreto do objeto e
- `a` representa uma função de abstração que mapeia o objeto concreto no objeto abstrato imaginado pelo seu usuário.

Dado o estado concreto c_1, que corresponde ao estado abstrato a_1, seu novo estado abstrato a_2 é atingindo em decorrência da execução da operação $a_2 = \texttt{a}(\texttt{f}(c_1))$, a qual tem o mesmo valor que $\texttt{F}(\texttt{a}(c_1))$, i.e., a aplicação da implementaçao `f` da função abstrata `F` ao estado concreto c_1 seguida da abstração do estado concreto produzido deve ser, teoricamente, o mesmo que abstrair-se do estado concreto c_1 seguido do mapeamento abstrato `F` aplicado ao estado abstraído.

Em benefício da legibilidade, `F` não deve ter efeito colateral, devendo todo esse efeito ficar confinado à sua implementação `f`. Esse tipo de efeito colateral é considerado legítimo.

Legitimidade do Efeito Colateral

Efeito colateral no estado concreto dos objetos é inofensivo, mas efeito colateral que ocorre no estado abstrato pode ser nocivo à legibilidade do programa.

Os benefícios da separação dos estados de um objeto em concreto e abstrato são demonstrados a seguir pela implementação de um tipo `FilaDeInt`, na qual, para otimizar o cálculo do número de elementos armazenados na fila em um dado momento, a função `numDeElementos` guarda sempre o valor que foi obtido na última computação realizada e sinaliza esse fato via o campo interno `contadorDefinido`, indicando se recálculos devem ou não ser efetuados, por exemplo, devido a inserções ou remoções de elementos.

```
1  class FilaDeInt {
2     private int contador;
3     private boolean contadorDefinido = false;
4     private int[] fila;
5     public FilaDeInt(...) { ... }
6     ...
7     public void insere(...) {
8        ...
9        contadorDefinido = false;
10       ...
11    }
12    ...
13    public int numDeElementos() {
14       if (!contadorDefinido) {
15            contador = "computa número de elementos";
16            contadorDefinido = true;
17       };
18       return contador;
19    }
20 }
```

A função `numDeElementos()` produz efeito colateral interno, no estado concreto do objeto associado, mudando valores de campos privados, mas externamente não há efeito colateral visível, porque os campos do estado concreto do objeto não fazem parte do estado abstrato mentalizado pelo usuário do tipo `FilaDeInt`.

11.2 Objetos como máquinas de estado

Objetos oferecem serviços e têm estados, que subsidiam a execução desses serviços, os quais podem causar mudanças de estado. Serviços podem ser mais que mapeamentos dos valores dos parâmetros de um método no resultado que ele produz. O uso do estado do objeto nessas operações propicia implementações mais interessantes, como é demostrado a seguir por duas implementações do tipo abstrato de dados `ListaDeInt`, que é uma lista de inteiros com as operações básicas de pesquisar por um elemento na lista, inserir, remover ou mudar valores de elementos em uma dada posição.

Implementação tradicional

Suponha que `ListaDeInt` seja definido pelas seguintes operações:

- `boolean vazia()`: informa se a lista está ou não vazia.
- `int numDeElementos()`: fornece o número de elementos armazenado na lista.
- `int valor(int i) throws IdxErrado`: retorna o valor do `i`-ésimo elemento, para `0<i<=numDeElementos`.
- `void mude(int i, int v) throws IdxErrado`: muda valor do i-ésimo elemento para `v`, se `i` for válido.
- `void insira(int i,int v) throws IdxErrado`: se `0<i<=numDeElementos`, insere valor `v` após `i`-ésimo elemento, senão se `i==0`, insere o valor `v` antes do primeiro elemento, senão levanta exceção `IdxErrado`.
- `void remova(int i) throws IdxErrado`: levanta a exceção `IdxErrado` se `i<1||i>numElements()`, senão remove o `i`-ésimo elemento.
- `int pesquise(int v)`: devolve a posição do elemento de valor `v`. Retorna `0` se `v` não for encontrado na lista.

A versão de **ListaDeInt**, implementada a seguir, usa uma lista simplesmente encadeada de objetos da classe **Nodo**, que está declarada como uma classe aninhada, estática e privada, de interesse apenas do estado concreto e interno da lista.

```
1  public class ListaDeInt {
2     private static class Nodo {
3        public int valor;
4        public Nodo direito;
5        public Nodo(int valor) {this.valor = valor;}
6     }
7     private Nodo primeiroElemento;
8     private int nElementos = 0;
9     private Nodo getAddress(int i) throws IdxErrado {
10       Nodo e; int j;   // custo de getAddress: O(n)
11       if (i < 1 || i > nElementos) throw new IdxErrado();
12       for (j=1,e=primeiroElemento;j!=i; j++,e=e.direito);
13       return e;
14    }
15    public class IdxErrado extends Exception {  }
16    public ListaDeInt() {primeiroElemento = null;}
17    public boolean vazia() {return (nElementos == 0);}
18    public boolean numDeElementos() {return nElementos;}
19    public int valor(int i) throws IdxErrado {...}
20    public void mude(int i,int v) throws IdxErrado{...}
21    public int pesquise(int v) {...}
22    public void insira(int i, int v) throws IdxErrado{...}
23    public void remova(int i) throws IdxErrado {...}
24 }
```

A função interna **getAddress**, que é usada por algumas operações de **ListaDeInt**, verifica sempre se o índice do elemento do qual se quer o endereço de seu nodo é ou não válido, levantando a exceção **IdxErrado**, se necessário. As operações usuárias de **getAddress** devem tratar ou propagar essa exceção apropriadamente, como é feito na implementação da operação **valor**, que

retorna o `i`-ésimo inteiro da lista corrente, sem provocar qualquer efeito colateral. As operações **`getAddress`** e **`valor`** têm custo $O(n)$, onde n denota o comprimento da lista de inteiros.

```
1 public int valor(int i) throws IdxErrado {
2    Nodo e; int j;
3    e = getAddress(i);  // Custo O(n)
4    return e.info;
5 }
```

A operação **`mude`** provoca o efeito de alterar o estado da lista pela troca do valor do `i`-ésimo elemento, se `i` for válido.

```
1 public void mude(int i,int v) throws IdxErrado {
2    Nodo e; int j;
3    e = getAddress(i); // custo O(n)
4    e.info = v;
5 }
```

A pesquisa por um dado elemento retorna sua posição na lista ou, se nada encontrar, retorna 0, sendo tudo feito sem qualquer efeito colateral. Essa operação tem custo $O(n)$ porque tem sempre que percorrer a lista sequencialmente.

```
1 public int pesquise(int v) {
2    Nodo e; int j; // Custo de pesquise: O(n)
3    if (vazia()) then return 0;
4    for (j=1 , e=primeiroElemento ;
5         j<=nElementos; j++, e=e.direito) {
6            if (v == e.valor) return j;
7    }
8    return 0;
9 }
```

As operações **`insira`** e **`remova`** devem localizar o ponto da lista de nodos encadeados onde a operação terá efeito, devendo para

isso percorrer a lista sequencialmente ao custo $O(n)$ para achar os endereços dos nodos envolvidos nessas operações. Nos termos da programação por contrato, todos os necessários testes para assegurar a obediência ao contrato são realizados e as exceções devidas são propagadas para os usuários das operações envolvidas.

```
1  public void insira(int i, int v) throws IdxErrado {
2     Nodo anterior, novo; // custo de insira: O(n)
3     int j;
4     if (i<0 || i>nElementos) throw new IdxErrado();
5     novo = new Nodo(v);
6     if (i == 0) {
7         novo.direito = primeiroElemento;
8         primeiroElemento = novo;
9     } else {
10        anterior = getAddress(i);
11        novo.direito = anterior.direito;
12    }
13    nElementos++;
14 }
15 void remova(int i) throws IdxErrado {
16    Nodo anterior, aux; // custo de remova: O(n)
17    if (i<1 || i>nElementos) throw new IdxErrado();
18    if (i == 1) {
19        primeiroElemento = primeiroElemento.direito;
20    } else {
21        anterior = getAddresss(i-1);
22        aux = anterior.direito;
23        anterior.direito = aux.direito;
24    }
25    nElementos--;
26 }
```

A seguir, um exemplo de uso da classe `ListaDeInt` é apresentado para destacar que os custos de suas operações, exceto o da função `vazia`, são $O(n)$, o que sugere que a técnica de imple-

mentação usada não é uma boa solução.

```
1  class User {
2     public static void main(String[] a) {
3        IntList q;
4        int k, v1=1, v2=2, v3=3, v4=4, int n=5;
5        ...
6        k = q.pesquise(v1);     // custo O(n)
7        v4 = q.valor(k);        // custo O(n)
8        q.insira(n,v2);         // custo O(n)
9        q.mudeValor(n,v3);      // custo O(n)
10       q.remova(n);            // custo O(n)
11    }
12 }
```

Para reduzir os custos das operações de `IntList` poder-se-ia:

- Solução I: reescrever `pesquise` para retornar o endereço do nodo que contém o valor pesquisado e não seu número de ordem. A desvantagem dessa solução é que viola-se encapsulação e deve-se exportar a classe `Nodo` para manipulação direta da representação da lista, reduzindo grau de reúso da abstração.
- Solução II: prover suboperações comuns para implementar operações relacionadas, com a vantagem de possivelmente gerar economia de código, mas com a desvantagem proliferar suboperações específicas e aumentar o tamanho da interface ao colocar operações desse tipo do lado de fora.
- Solução III: mudar o contrato, no sentido de que a lista tenha a noção de nodo corrente ou nodo ativo, que todas as suas operações sejam relativas a esse nodo, o qual pode ser afetado pelas operações realizadas. Essa mudança de visão força uma redefinição da interface do tipo `ListaDeInt` para incorporar detalhes do estado corrente do objeto.

Implementação como máquina de estado

Um objeto do tipo `ListaDeInt` pode ser visto, abstratamente, como uma estrutura que possui um cursor que aponta para um nodo da lista para o qual muitas de suas operações são definidas.

A Fig. 11.2 mostra o estado abstrato de um objeto `ListaDeInt` visto como uma máquina de estado, onde `ativo` designa o nodo apontado pelo cursor da lista, referido como nodo corrente.

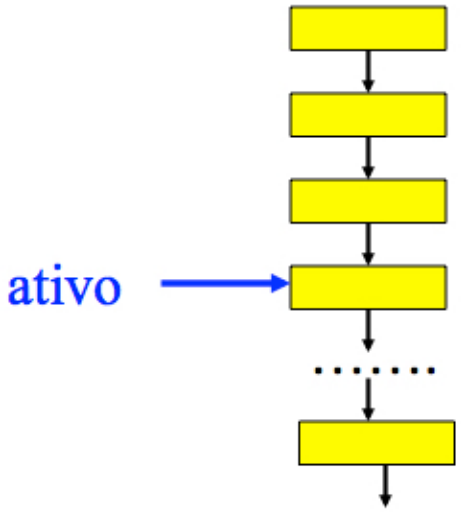

Figura 11.2 Vista do Estado Abstrato

Nesse cenário, as operações de `ListaDeint` passam a ser do tipo *insira à direita do cursor*, *insira à esquerda do cursor*, *informe o valor do elemento referenciado pelo cursor* e *posicione o cursor em um nodo com um dado valor*, sendo todas de custo $O(1)$, exceto a última, que envolve pesquisa, que continua com custo $O(n)$, pois toda a lista pode ter que ser percorrida.

O novo contrato de `ListaDeInt` passa a ser definido pelas seguintes operações:

- `int numDeElementos()`: o número de elementos.
- `boolean vazia()`: informa se a lista está ou não vazia.

- `void inicie() throws SemAtivo`: posiciona o cursor no início.
- `void finalize()`: posiciona o cursor na última posição.
- `void avance() throws SemAtivo`: avança o cursor uma posição
- `void recue() throws SemAtivo`: recua o cursor uma posição.
- `void posicione(int i) throws SemAtivo,IdxErrado`: posiciona o cursor no i-ésimo elemento.
- `boolean alémDoDireito()`: cursor além da última posição?
- `boolean aquémDoEsquerdo():` cursor aquém da primeira posição?
- `boolean éPrimeiro()`: cursor na primeira posição?
- `boolean éÚltimo():` cursor na última posição?
- `int valor() throws SemAtivo`: retorna o valor do elemento apontado pelo cursor.
- `void mude(int v) throws SemAtivo`: muda o valor do elemento corrente.
- `void insira(int v) throws OpInválida`: insere nodo à direita do corrente
- `void remova() throws OpInválida`: remove nodo corrente. O elemento seguinte torna-se o corrente. Se o removido for o último, corrente fica indefinido.
- `boolean pesquise(int v)`: move o cursor para o elemento da lista com valor `v` e retorna `true`; senão faz as chamadas a `alémDoDireito()` retornar `true` e retorna `false`.

Em atendimento às regras do contrato, as seguintes exceções podem ser levantadas:

- Elemento corrente indefinido:
  `public class SemAtivo extends Exception {}`

- Tentativa de acesso a elemento inexistente:
  `public class IdxErrado extends Exception {}`
- Tentativa de inserção com corrente indefinido:
  `public class OpInválida extends Exception {}`

A nova implementação de `ListaDeInt` opera com objetos que encapsulam os estados concretos como o mostrado na Fig. 11.3.

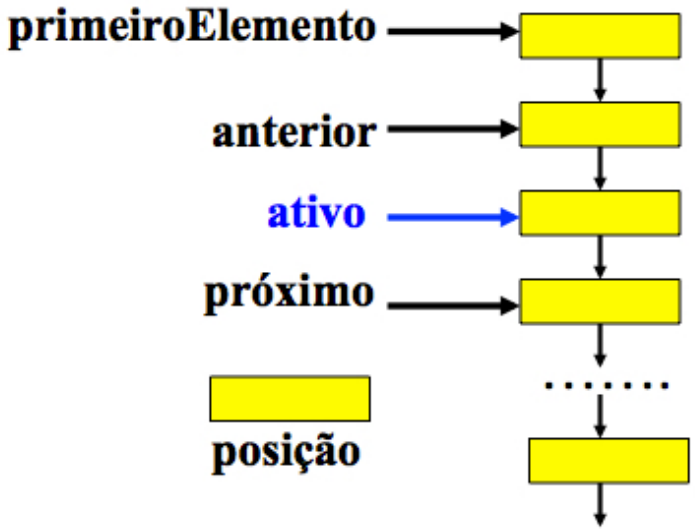

Figura 11.3 Vista do Estado Concreto

Os campos do estado concreto, todos declarados privados e indicados na Fig. 11.3, têm o seguinte propósito:

- `ativo` representa o cursor que aponta para o nodo corrente, sempre que houver um;
- `posição` dá a ordem do nodo apontado por `ativo` na lista.
- `primeiroElemento` dá o endereço do primeiro nodo da lista encadeada de objetos do tipo `Nodo` e
- os campos `anterior` e `próximo` são usados para tornar as operações com o nodo corrente mais eficiente.

Esses cinco campos privados fazem parte do estado corrente de um objeto do tipo`ListaDeInt`, sendo seus valores relacionados en-

tre si. A cada operação podem ser atualizados de forma a assegurar uma boa perfórmance das operações do tipo.

Os nodos da lista encadeada usada na implementação a seguir do tipo `ListaDeInt` são objetos da classe `Nodo`, definida como estrutura privada mostrada no código abaixo.

```
1  public class ListaDeInt {
2      private static class Nodo {
3          public int valor;
4          public Nodo direito;
5          public Nodo(int valor) {this.valor = valor;}
6      }
7      private int nElementos, posição;
8      private Nodo primeiroElemento;
9      private Nodo anterior, ativo, próximo;
10     public boolean vazia() {return (nElementos == 0); }
11     public int numDeElementos() { return nElementos; }
12     public boolean aquémDoEsquerdo() {...}
13     public boolean alémDoDireito() { ... }
14     public boolean éPrimeiro { ... }
15     public boolean éÚltimo() { ... }
16     public int valor() throws SemAtivo { ... }
17     public void mude(int v) throws SemAtivo { ... }
18     public void inicie() throws SemAtivo { ... }
19     public void avance() throws SemAtivo { ... }
20     public void recue()  throws SemAtivo { ... }
21     public void posicione(int i)
22                          throws IdxErrado, SemAtivo { ... }
23     public boolean pesquise(int v) { ... }
24     public void insira(int v) throws OpInválida { ... }
25     public void remova() throws OpInválida { ... }
26 }
```

O campo `ativo` pode ter um valor `null` ou então apontar para o nodo corrente da lista. Consistentemente, o campo `posição` indica a ordem do nodo apontado por `ativo`: um valor de `posição`

no intervalo $[1, \texttt{nElementos}]$ informa que `ativo` tem o endereço do nodo corrente da lista, mas quando `ativo` tem valor `null`, tem-se `posição` com valor `0` para indicar que a última operação realizada deixou o cursor aquém do primeiro elemento da lista ou então `posição` com valor `nElementos+1`, para o cursor além do último elemento. Essas configurações são reveladas pelas operações `aquémDoEsquerdo` e `alémDoDireito`:

```
1 public boolean aquémDoEsquerdo() {
2    return (vazia() || (posição == 0));
3 }
4 public boolean alémDoDireito() {
5    return (vazia() || (posição == nElementos+1));
6 }
```

Similarmente, a identificação de certos posicionamentos do cursor são obtidos pelas seguintes operações:

```
1 public boolean éPrimeiro() {return (posição == 1);}
2 public boolean éÚltimo() {return (posição == nElementos);}
```

Pode-se obter ou atualizar o valor do elemento corrente pelas operações `valor` e `mude`:

```
1  public int valor() throws SemAtivo {
2      if (aquémDoEsquerdo() || alémDoDireito())
3                                  throw new SemAtivo();
4      return ativo.valor ;
5  }
6  public void mude(int v) throws SemAtivo {
7      if (aquémDoEsquerdo() || alémDoDireito())
8                                  throw  new SemAtivo();
9      ativo.valor = v;
10 }
```

A ideia de que o objeto **ListaDeInt** seja visto como uma máquina de estado sugere a inclusão das operações **inicie**, **avance**, **recue** e **posicione** para redefinição do elemento corrente e reposicionamentos do cursor da lista adequadamente. Essas operações afetam os campos **ativo**, **anterior**, **próximo** e **posição**, definidores do estado do objeto.

```
1  public void inicie() throws SemAtivo {
2     if (vazia())  throw SemAtivo();
3     anterior = null;
4     ativo = primeiroElemento;
5     próximo = ativo.direito ;
6     posição = 1;
7  }
8  public void avance() throws SemAtivo {
9     if (alémDoDireito()) throw new SemAtivo();
10    if (aquémDoEsquerdo()) inicie();
11    else {anterior = ativo;
12          ativo = próximo;
13          if (próximo!=null) próximo = próximo.direito;
14          posição++;
15    }
16 }
17 public void recue() throws SemAtivo {
18    if (aquémDoEsquerdo()) throw new SemAtivo();
19    try {posicione(posição-1);} catch(IdxErrado e) { }
20 }
21 public void posicione(int i) throws IdxErrado, SemAtivo {
22    if (i<1||i>nElementos) throw new IdxErrado();
23    if (i < posição) inicie();
24    for ( ; posição!=i&&!alémDoDireito); avance());
25 }
```

A operação **pesquise(x)** percorre a lista de inteiros a partir de seu início em busca de um elemento com o valor dado pelo parâmetro **x**. Se esse elemento for encontrado, ele torna-

se o elemento corrente, e o método retorna **true**, caso contrário, **ativo** torna-se **null**, **alémdoDireito**, **true**, e a operação retorna **false**. O custo de **pesquise** é $O(n)$, haja vista que a lista pode ter que ser percorrida em toda sua extensão.

```
1 public boolean pesquise(int v) {
2     try {       // custo de pesquise: O(n)
3        for (inicie() ; !alémDoDireito() ; avance())
4           if (ativo.valor = v) return true;
5     } catch(SemAtivo e) { }
6     return false;
7 }
```

A operação **insira** exige que a lista de inteiros esteja vazia ou que tenha elemento ativo definido, i.e., listas não-vazias e sem elemento ativo definido não aceitam inserções. No caso de lista vazia, a operação insere o primeiro nodo com o valor passado pelo parâmetro. No caso de lista não-vazia com um elemento ativo definido, o novo nodo é inserido à direita desse elemento. Se essas condições não forem encontradas, a exceção **OpInválida** será lançada.

```
1 public void insira(int v) throws OpInválida {
2     Nodo novo = new Nodo(v);
3     if (vazia()) {
4        primeiroElemento = novo; ativo = novo;
5        anterior = null; próximo = null; posição = 1;
6     } else {
7        if (aquémDoEsquerdo() || alémDoDireito())
8                                throw new OpInválida();
9        novo.direito = próximo; ativo.direito = novo;
10       anterior = ativo; ativo = novo;
11    }
12    nElementos++; // custo O(1)
13 }
```

A operação **remova** exige que a lista não esteja vazia e que o elemento corrente esteja definido. Se essas condições não forem satisfeitas, uma exceção é levantada. Após a remoção do nodo corrente, **ativo** passa a ser o nodo definido como **próximo**.

```
1  public void remova() throws OpInválida {
2     if (aquémDoEsquerdo() || alémDoDireito() || vazia())
3                                        throw new OpInválida();
4     ativo = próximo;
5     if (ativo != null) próximo = próximo.direito;
6     if (posição == 1)  {
7         primeiroElemento = ativo;
8         if (primeiroElemento == null) posição = 0;
9     } else anterior.direito = ativo;
10    nElementos--;
11 }
```

A seguir, um exemplo de uso da nova classe **ListaDeInt** é apresentado para destacar que os custos de suas operações, exceto os das operações **pesquise** e **posicione**, são $O(1)$, o que sugere que a técnica de implementação usada pode ser uma boa solução.

```
1  class Usuário {
2     public static void main(String[] a) {
3        ListaDeInt q;  boolean b;
4        int v1=1, v2=2, v3=3, v4=4, n=5;
5        b = q.pesquise(v1);      // custo O(n)
6        v4 = q.valor();          // custo O(1)
7        q.posicione(n);          // custo O(n)
8        q.insira(v2);            // custo O(1)
9        q.mude(v3);              // custo O(1)
10       q.remova();              // custo O(1)
11    }
12 }
```

Sem dúvida, o desempenho das operações melhorou em relação à solução anteriormente apresentada.

11.3 Escolha de abstrações

A escolha das abstrações mais apropriadas a uma dada aplicação tem efeito direto na qualidade do código produzido, principalmente no que se refere à legibilidade. Para exemplificar, considere a implementação de um pequeno programa de configuração de computadores com determinados módulos de expansão, os quais podem ser uma placa, que contenha um leitor de cd, uma placa de rede ou um banco de memória extra, e um monitor de vídeo monocromático ou policromático. Todos os componentes têm preço fixo, exceto as placas de expansão, que admitem descontos.

A seguir, estão descritas duas soluções para esse mesmo problema, construídas para ilustrar como as abstrações escolhidas afetam a qualidade do programa produzido.

Configuração versão I

As famílias das placas de expansão e dos monitores de vídeo têm as hierarquias descritas na Fig. 11.4:

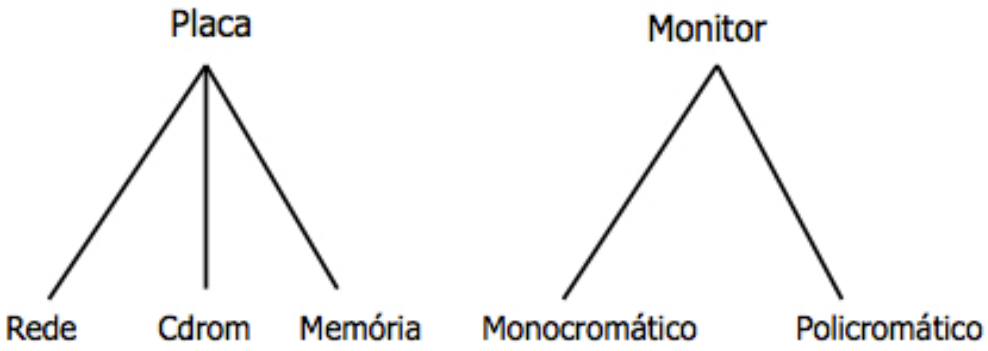

Figura 11.4 Hierarquia de Cartões e Monitores

A classe abstrata `Placa` define, via sua construtora, o preço base e o desconto admitido para o respectivo componente, e deixa em aberto a definição do seu nome identificador.

```
1 abstract class Placa {
2    private int preço, desconto;
3    public Placa(int preço, int desconto) {
4       this.preço = preço; this.desconto = desconto;
5    }
6    public abstract String nome();
7    public int preço() {return preço;}
8    public int desconto() {return desconto;}
9 }
```

Os três tipos de placas de expansão têm a seguinte definição:

```
1 class Cdrom extends Placa {
2    public Cdrom(int preço, int desconto) {
3       super(preço,desconto);
4    }
5    public Cdrom(int preço) {super(preço,30);}
6    public String nome() {return "Cdrom  ";}
7 }
8 class Rede extends Placa {
9    public Rede(int preço, int desconto) {
10      super(preço,desconto);
11   }
12   public Rede(int preço) {super(preço,30);}
13   public String nome() {return "Rede   ";}
14 }
15 class Memória extends Placa {
16   public Memória(int preço, int desconto) {
17      super(preço,desconto);
18   }
19   public Memória(int preço) {super(preço,30);}
20   public String nome() {return "Memória";}
21 }
```

E a classe abstrata `Monitor` estabelece que monitores de vídeo têm preço sem previsão de descontos, e disponibiliza as interfaces de operações para identificar o tipo de monitor e informar seu preço, por meio de apropriadas subclasses.

```
abstract class Monitor {
   private int preço;
   public Monitor(int preço) {this.preço = preço;}
   public int preço() {return preço;}
   public abstract String nome();
}
```

À família dos monitores de vídeo são incorporadas as classes concretas `Policromático` e `Monocromático`:

```
class Policromático extends Monitor {
   public Policromático(int preço) {super(preço);}
   public String nome() {return "Policromático";}
}
class Monocromático extends Monitor {
   public Monocromático(int preço) {super(preço);}
   public String nome() {return "Monocromático";}
}
```

A classe central da aplicação é `Computador`, a qual permite simular várias configurações do computador é a seguinte:

```
class Computador {
   private int preço;
   private Placa placa;
   private Monitor monitor;
   public Computador(int preço, Placa c, Monitor m) {
      this.preço = preço; placa = c; monitor = m;
   }
   public int preçoFinal() { ... }
   public void imprima() {
     System.out.println(placa.nome() +
      " + Monitor " +  monitor.nome() +
        " + Computador : preço final = " +  preçoFinal());
  }
}
```

A operação que determina o preço final do computador configurado com placa e monitor tem o seguinte código, no qual observa-se que se requer o conhecimento de que somente placas têm desconto!

```
1 public int preçoFinal() {
2    return this.preço + monitor.preço() +
3                  placa.preço() - placa.desconto();
4 }
```

O estoque disponível para venda é o definido pela classe **Estoque**, que constrói seis configurações e permite listar o preço de cada uma.

```
1  class Estoque {
2     private Placa rede    = new Rede(500,50);
3     private Placa cdrom   = new Cdrom(300,40);
4     private Placa mem     = new Memória(600);
5     private Monitor mono  = new Monocromático(400);
6     private Monitor poli  = new Policromático(700);
7     private Computador rm = new Computador(1000,rede,mono);
8     private Computador cm = new Computador(1000,cdrom,mono);
9     private Computador mm = new Computador(1000,mem,mono);
10    private Computador rp = new Computador(1000,rede,poli);
11    private Computador cp = new Computador(1000,cdrom,poli);
12    private Computador mp = new Computador(1000,mem,poli);
13    public void imprima() {
14       rm.imprima(); cm.imprima(); mm.imprima();
15       rp.imprima(); cp.imprima(); mp.imprima();
16    }
17 }
```

Finalmente, o programa principal, que cria o estoque:

```
1 public class Loja {
2    public static void main (String[] args) {
3       Estoque estoque = new Estoque(); estoque.imprima();
4    }
5 }
```

O código apresentado nesta solução tem um problema: para entender o funcionamento da operação `preçoFinal` definida na classe `Computador`, deve-se estar ciente que somente placas têm descontos, ou seja, para entender o funcionamento de um módulo deve-se ter conhecimento de detalhes de outros. No caso em pauta, mudanças na política de preços dos componentes do computador têm impacto tanto nas classes que implementam esses componentes quanto nas que os usam.

Configuração versão II

Com objetivo de dar uniformidade ao tratamento dos módulos de expansão dos computadores a ser configurados, é conveniente trabalhar com uma hierarquia com a da Fig. 11.5, onde `Placa` e `Monitores` são vistos como do mesmo tipo `Componente`.

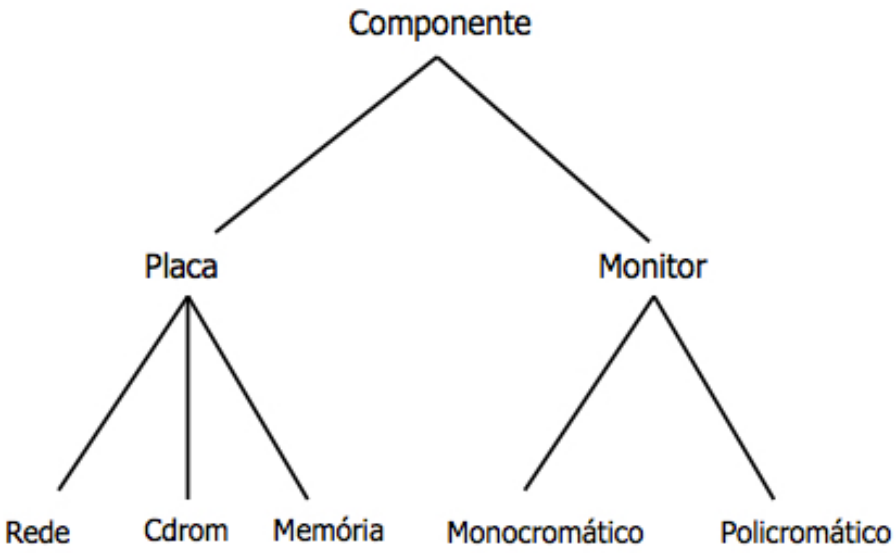

Figura 11.5 Nova Hierarquia de Cartões e Monitores

Nessa nova hierarquia todos os componentes compartilham definições de estruturas internas e operações, permitindo que, em princípio, todo componente possa ter desconto, como mostram as seguintes declarações de suas classes abstratas:

```
1  abstract class  Componente {
2     private int preço, desconto;
3     public Componente(int preço, int desconto) {
4        this.preço = preço; this.desconto = desconto;
5     }
6     public abstract String nome();
7     public int preço() return preço - desconto;
8  }
9  abstract class Placa extends Componente {
10    public Placa(int preço, int desconto) {
11       super(preço,desconto);
12    }
13 }
14 abstract class Monitor extends Componente {
15    public Monitor(int preço) {super(preço,0);}
16 }
```

As classes `Monocromático`, `Policromático`, `Cdrom`, `Rede` e `Memória` da versão I ficam inalteradas e podem ser usadas na versão II. A classe `Computador` somente é afetada na implementação de `preçoFinal`, pois, como agora é permitido descontos para todos os tipos de componentes, essa operação pode tratar todos os componentes de forma uniforme:

```
1  public int preçoFinal() {
2    return this.preço + monitor.preço() + placa.preço();
3  }
```

permanecendo inalterado todo o restante do código de `Computador`.

A versão II mostra-se superior à versão I porque na II diferentes componentes comportam-se de forma uniforme, facilitando o entendimento do funcionamento do programa, e consequentemente facilitando sua manutenção.

Configuração versão III

A aplicação em estudo ainda pode ser melhorada nos seguintes pontos:

- A herarquia da Fig. 11.5 parece demasiadamente elaborada, podendo ser simplificada para a da Fig. 11.6, de apenas dois níveis.

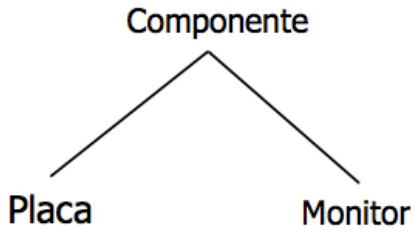

Figura 11.6 Nova Hierarquia de Cartões e Monitores

- A definição do nome de cada componente via a função `nome` não é uma boa solução: melhor seria ter um campo privado `nome` na classe `Componente`, e essa função apenas retornar seu valor, tornando a implementação mais flexível, conforme mostra a seguinte declaração:

```
1  class Componente {
2     private int preço, desconto;
3     private String nome;
4     public Componente(String nome,int preço,int desconto){
5        this.nome = nome; this.preço = preço;
6        this.desconto = desconto;
7     }
8     public String nome() {return nome;}
9     public int preço() {return preço - desconto;}
10 }
```

- Com as modificações acima, no lugar das cinco classes, **Rede**, **Cdrom**, **Memória**, **Monocromático** e **Policromático**, basta a definição das classes **Placa** e **Monitor**, pois, dentro de sua categoria, componentes agora são diferenciados pelo campo privado **nome**. Essas duas novas classes têm a função de fixar os preços e os valores dos descontos dos componentes, como está detalhado no seguinte código, no qual, por questão de uniformidade, monitores também podem ter descontos, tendo, por compatibilidade com o enunciado do problema, o valor 0 por *default*.

```
1  class Placa extends Componente {
2     public Placa(String nome,int preço,int desconto) {
3         super(nome,preço,desconto);
4     }
5     public Placa(String nome, int preço) {
6         super(nome,preço,30);
7     }
8  }
9
10 class Monitor extends Componente {
11    public Monitor(String nome,int preço,int desconto) {
12        super(nome,preço,desconto);
13    }
14    public Monitor(String nome, int preço) {
15        super(nome,preço,0);
16    }
17 }
```

O impacto das modificações realizadas acima na classe **Estoque** é limitado a adaptá-la para incorporar o nome de cada componente à lista de parâmetros de chamada de suas respectivas construtoras, conforme definido nas novas classes **Placa** e **Monitor**.

A nova declaração de **Estoque** é a seguinte:

```
class Estoque {
   private Placa rede    = new Placa("Rede   ",500,50);
   private Placa cdrom   = new Placa("Cdrom  ",300,40);
   private Placa mem     = new Placa("Memória",600);
   private Monitor mono  = new Monitor("Monocromático",400);
   private Monitor poli  = new Monitor("Policromático",700);
   private Computador rm = new Computador(1000,rede,mono);
   private Computador cm = new Computador(1000,cdrom,mono);
   private Computador mm = new Computador(1000,mem,mono);
   private Computador rp = new Computador(1000,rede,poli);
   private Computador cp = new Computador(1000,cdrom,poli);
   private Computador mp = new Computador(1000,mem,poli);
   public void imprima() {
      rm.imprima(); cm.imprima(); mm.imprima();
      rp.imprima(); cp.imprima(); mp.imprima();
   }
}
```

A classe **Computador**, que é a central da aplicação, é a mesma da versão II, e está repetida abaixo para facilitar a apresentação.

```
class Computador {
   private int preço;
   private Placa placa; private Monitor monitor;
   public Computador(int preço, Placa c, Monitor m) {
      this.preço = preço; placa = c; monitor = m;
   }
   public int preçoFinal() {
      return this.preço + monitor.preço() + placa.preço();
   }
   public void imprima() {
      System.out.println( placa.nome() + " +
         Monitor " + monitor.nome() + " +
            Computador : preço final = " + preçoFinal());
   }
}
```

E o programa principal permanece o mesmo:

```
1 public class Loja {
2    public static void main (String[] args) {
3       Estoque estoque = new Estoque();
4       estoque.imprima();
5    }
6 }
```

A aplicação descrita é propositalmente muito simples de forma a destacar com clareza o impacto da escolha das abstrações na qualidade do código produzido.

11.4 Acesso ao estado concreto

Princípio Restrição ao Acesso

Minimize a acessibilidade aos elementos que compõem o estado concreto de objetos.

Objetos são dotados de um estado abstrato e de um estado concreto. O estado abstrato é visão definida por sua sua interface pública. E a estrutura de dados que provê representação de qualquer objeto define seu estado concreto deve, em princípio, ser declarada privada à classe que a contém. O acesso direto a componentes dessa estrutura engessa a definição do objeto, limitando sua evolução.

Deve-se sempre dar preferência à definição de uma interface para o objeto baseada em na visão externa de seu estado abstrato. Entretanto, se, por algum motivo o acesso a algum campo interno for inevitável, é melhor declarar o campo como privado e definir acessores. Por exemplo, no lugar de declarar público um campo do tipo `public double peso;` melhor seria, no mínimo, substituir essa declaração pelo seguinte código:

```
private double peso;
public double pegaPeso() {return peso;}
public void defPeso(double peso) {
   this.peso = peso;
}
```

a qual, pelo menos, desvincula a representação de `peso` de seu uso.

O desrespeito ao Princípio da Restrição ao Acesso prejudica a autonomia do projetista do módulo de experimentar diferentes técnicas de implementação depois de o módulo ter sido liberado.

11.5 Nomeação de classes, objetos e métodos

Classes devem ser designadas conforme a natureza dos objetos que modelam. No Capítulo 9, Princípios de Projeto, Seção 9.2, define-se que cada objeto em um sistema deve representar um *papel*, uma *transação* ou um *ente*, devendo, consistentemente, ser nomeados por substantivos, como `Aluno`, `Compra` e `Pessoa`.

Não faz sentido designar classes com verbos, pois não é assim que normalmente se faz com as entidades associadas no mundo real, e, ainda, comandos em qualquer linguagem de programação normalmente usam os nomes de objetos com as funções de *sujeito* ou *predicado*, como em `x.desenha(a)`, que pode ser lido como o objeto `x` *desenha o objeto* `a`.

Alternativamente, classes que definem propriedades estruturais importantes podem ser nomeadas por adjetivos, como `Comparável` ou `Monitorado`, e nesse caso, espera-se que essas propriedades sejam válidas em toda a hierarquia de que participam.

Classes tipicamente contêm estruturas de dados que moldam a representação de seus objetos. Essas estruturas são resultados da composição de estruturas menores, que, por sua vez, têm nomes, que devem ser compatíveis com seus papéis no sistema, podendo ser

designadas por substantivos ou adjetivos. Por exemplo, declarações de campos de uma classe como `boolean vazio`, `Aluno nome`, `Aluno[] aprovados` e `boolean removido` são de uso frequente.

Por outro lado, as operações de classe que computam e retornam valores devem sempre ser designadas conforme o tipo de valor retornado, tal como ocorre com campos, e as operações destinadas a provocar efeito colateral nos valores internos de objeto devem ser designadas por verbos no tempo imperativo, como em `disciplina.matricule(aluno)` ou, então, no tempo infinitivo, como em `disciplina.cancelar(aluno)`.

Conclusão

Este capítulo discute formas de estruturação de programas orientados por objetos, aqui chamadas de estilo, de forma privilegiar diversos atributos ou requisitos de qualidade como simplicidade, correção, modularidade, clareza e reusabilidade.

Exercícios

1. Quais são as diferenças entre estado concreto e estado abstrato de um objeto?

2. Quais seriam as vantagens de tratar objetos como máquinas de estado?

Notas bibliográficas

Um excelente texto sobre programação orientada por objetos é o livro de Bertrand Meyer [41], que, nos capítulos 22 (*How to Find Classes*) e 23 (*Principles of Class Design*), apresenta muitos exemplos de aplicação dos conceitos aqui descritos.

Capítulo 12

Considerações Finais

O desenvolvimento de software de grande porte necessita de técnicas de organização de código dos programas que privilegiem extensibilidade e reusabilidade e que suportem o desenvolvimento sistemático de software de forma a garantir correção e robustez.

A orientação por objetos tem foco na identificação e implementação dos objetos que compõem um sistema, e, nesse contexto, tipos abstratos de dados são uma ferramenta extremamente útil e indispensável, pois permitem a implementação dos seguintes clássicos conceitos da Engenharia de Software:

- Abstração: foco no relevante enquanto ignoram-se detalhes de implementação.
- Modularidade: divisão de um problema em subproblemas independentes.
- Encapsulação e controle de visibilidade: *boas cercas fazem bons vizinhos*.
- Separação de interesses: conceitos relacionados devem ser agrupados em módulos, evitando seu espalhamento ao longo do código.

Claramente, todos esses conceitos levam a ideia de que modularizar é a atividade central no desenvolvimento de sistemas de grande porte, e que módulos devem ser autônomos e independen-

tes de forma a minimizar sua conectividade e reduzir os custos de manutenção.

Neste livro, diversas técnicas para o desenvolvimento de sistemas modulares e de alto grau de réuso são discutidos e avaliados, na expectativa de contribuir para disseminar os sólidos e bem fundamentados princípios de projeto de programas orientados por objetos, dos quais pode-se derivar o seguinte conjunto de mandamentos:

- Construa apenas classes para servir a um único e consistente propósito (responsabilidade única).
- Projete as classes de interesse geral como tipo abstrato de dados ou como empacotadoras de funções relacionadas.
- Use classes do tipo ficha-de-dados apenas localmente.
- Identifique claramente a natureza de cada classe: coisa, papel ou transação.
- Certifique-se que subclasses tenham sempre a mesma natureza de suas classes-base públicas.
- Assegure-se que cada subclasse expresse a relação "é um tipo especial de" e não "pode exercer o papel de" das classes base públicas.
- Assegure-se que toda subclasse respeite o contrato da respectiva superclasse
- Respeite os Princípios de Projeto Orientado por objetos.
- Identifique o tipo de herança a ser usada.
- Dê preferência à composição sobre herança, quando pertinente.
- Evite repetição de código: use abstrações de função, composição de objetos, hierarquia, etc.
- Use herança múltipla com parcimônia.
- Não redefina comportamento de operações da superclasse na subclasse, mas apenas especialize-o, respeitando o contrato.
- Não use o modificador `protected` sem uma causa justa.
- Não estenda classes utilitárias: use composição.

- Minimize a interação entre classes, reduza acoplamento e aumente coesão interna.
- Dê preferência às abstrações mais simples.
- Mova abstrações comuns para a classe-base.
- Migre comportamento comum para a classe-base.
- Não use hierarquia apenas para compor objetos, use composição diretamente.
- Prefira argumentos *default* à sobrecarga de funções e de construtores.
- Assegure-se que todo construtor sempre deixe o objeto criado em um estado bem definido.
- Assegure-se que toda subclasse trate o estado herdado de forma consistente com o esperado pela classe-base.
- Identifique o invariante de toda classe.
- Assegure-se que todas as operações verifiquem o invariante da classe na sua entrada e na sua saída.
- Assegure-se que a interface de toda classe seja consistente em relação a todas as suas operações.
- Não use construtoras para iniciar dados estáticos.
- O comportamento de um objeto frente a um erro deve ser bem definido e consistente.
- Pratique programação defensiva: sempre desconfie que parâmetros possam estar "errados".
- Use Programação por Contrato.
- Use Padrões-de-Projeto: evite reinventar.
- Privilegie decisões que favorecem reusabilidade.
- Preocupe-se com a extensibilidade do sistema.
- Documente seu programa.
- Mantenha a documentação sempre atualizada.

Obrigado pela atenção!

Bibliografia

[1] Alexander, C., Ishikawa S., Silverstein, M., Fiksdahl-King & Angel, S. *A Pattern Language*, Oxford University Press, New York, 1977.

[2] Arnold, K. ; Gosling, J. & Holmes, D. *The Java Programming Language*, Addison-Wesley, Third Edition, 2000, ISBN 0-201-70433-1.

[3] Arnold, K. ; Gosling, J. & Holmes, D. *A Linguagem de Programação Java, Quarta Edição*, Bookman 2007, ISBN 976-85-60031-64-1.

[4] Beck, K. & Cunningham, W. *Using Pattern Languages for Oriented-Object Programs*, OOP-SLA-87 Worshop on Specification and Design for Object-Oriented Programming, 1987.

[5] Bigonha, R.S. *A Linguagem Java, Série Programação Modular*, 2020, ISBN 999-99-99999-99-9.

[6] Cardelli, L. ; Wegner, P. On Understanding Types, Data Abstraction and Polymorphism, *ACM Computing Surveys*, 17, (4):471-522, December 1985.

[7] Conell, G ; Horstmann, C.S. & Cay S. *Core Java*, Makron Books, 1997.

[8] Coad, P. ; Mayfield, M. ; Kern, J. *Java Design: Building Better Apps and Applets*, Yourdon Press, 2nd Edition, 1999.

[9] Cooper, J. *Java Design Patterns: A Tutorial*, Addison-Wesley, 2000.

[10] Dahl, Ole-Johan; Nygaard, Kristen. SIMULA: an ALGOL-based simulation language, Communications of the ACM, v.9 n.9, p.671-678, Sept. 1966

[11] Deitel, H.M. ; Deitel, P.J. *Java: Como Programar*, Sexta Edição, Pearson, Prentice-Hall, 2005.

[12] Descartes, R. *Discurso do Método: Regras para a Direção do Espírito*, Coleção A Obra-Prima de Cada Autor, Editora Martin Claret, 2000

[13] Deremmer, F.; Kron, H. *Programming in-the-large versus Programming in-the-small*, In *International Conference on Reliable Software*, pages 114–171, Los Angeles, 1975.

[14] Dijkstra, E. *A Discipline of Programming.* Prentice Hall, 1976.

[15] Ege, R. *Programming in an Object-Oriented Environment*, Academic Press, INC, San Diego, California, 1992.

[16] Ellis, M. & Stroustrup, B. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[17] Fowler, Martin. *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[18] Gabriel, Richard E., *Patterns of Software: Tales From the Software Community*, Oxford University Press, 1996.

[19] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[20] Goldberg, A. & Robson, D. *Smalltalk-80: the Language and its Implementation*, Addison-Wesley, 1983.

[21] Goldberg, A. & Robson, D. *Smalltalk the language*, Addison-Wesley, 1989.

[22] Grand, Mark. *Patterns in Java*, Volume 2, Wiley, 1999.

[23] 001 Grand, Mark. *Java Enterprise Design Patterns - Patterns in Java*, Volume 3, Wiley, 2001.

[24] Hehl, M.E. *Linguagem de Programação Estruturada FORTRAN 77*, McGraw-Hill, São Paulo, 1986.

[25] Hoare, C.A.R. *Editorial: The Quality of Software*, In *Software, Practice and Experience*, vol. 2, No 2, pages 103–105, 1972.

[26] Hughes, J.K. *PL/I Structured Programming*, John Wiley & Sons, Inc, 1973.

[27] Ichbiah, J.D. ; Morser, S.P. *General Concepts of Simula 67 Programming Language*, 1967.

[28] Johnson, R.E. Frameworks=(Componentes + Patterns), *Communications of the ACM*, october 1997, vol. 40, No. 10.

[29] Laddad, R. *AspectJ in Action: Practical Aspect Oriented Programming*, Manning Publications Co, 2003.

[30] Lintz, B. P.; Swanson, E.B. *Software Maintenance: A User/Management Tug of War*, Data Management, pp. 26-30, april 1979.

[31] Lieberherr, K.J.; Doug, O. *Preventive program maintenance in Demeter/Java* (research demonstration). In *International*

Conference on Software Engineering, ACM Press, pages 604–605, Boston, MA, 1997.

[32] Liskov, B.; Zilles, S. Programming with Abstract Data Type. *ACM Sigplan Notices*, march, 1974.

[33] Liskov, B. et alii. *CLU Reference Manual*, Springer-Verlag, Berlin, 1981.

[34] Liskov, B. et alii. *Abstraction and Specification in Program Development*, MIT Press and McGraw-Hill, New York, 1986.

[35] Liskov, B. *Data Abstraction and Hierarchy*, *ACM Sigplan Notices*, 23,5 (May, 1988).

[36] Martin, R.C. *The Open-Closed Principle*. Disponível na página *www. objectmentor.com/resources/articles/ocp.pdf*. Acesso: 28/02/2005.

[37] Martin, R.C. *Design Principles and Design Patterns*. Disponível na página *www.objectmentor.com*. Acesso: 28/02/2005.

[38] Martin, R.C. *The Liskov Substitution Principles*. Disponível na página *www.objectmentor.com*. Acesso: 28/02/2005.

[39] Marinescu, Floyd. *JB Design Patterns: Advanced Patterns, Processes and Idioms*, John Wiley, 2002.

[40] Metsker, John, *Design Patterns Java Worldbook*, Addison-Wesley, 2001.

[41] Meyer, B. *Object-Oriented Software Construction*, Second Edition. Prentice Hall, 1997.

[42] Myers, G. J. *Reliable Software Through Composite Design*, Petrocelli/Charter, 1975.

[43] Myers, G. J. *Composite/Structured Design*, Van Nostrand Reinhold Company, 1978.

[44] Ossher, H.; Tarr, P. *Hyper/J: Multi-Dimensional Separation of Concerns for Java*, In *Proceedings of the 22nd international conference on Software Engineering*, pages 734–737. ACM Press, 2000.

[45] Ossher, H.; Tarr, P. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software, *Communications of the ACM*, 44(10):43–50, 2001.

[46] Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules, *Commun. ACM*, 15(12):1053–1058, 1972.

[47] Rising, Linda (editor). *The Patterns Handbook*, Cambridge, University Press, 1998.

[48] Rumbaugh, J.; Jacobson, I.; Booch, G. *The Unified Modeling Language Reference Manual*, Addison-Wesley Longman, 1999.

[49] Rumbaugh, J.; Jacobson, I.; Booch, G. *The Unified Modeling Language User Guide*, Addison-Wesley Longman, 1999.

[50] Shalloway, Allan & Trott, James R. *Design Patterns Explained*, Addison-Wesley, 2001.

[51] Scott, W. Ambler. *Análise e Projeto Orientados a Objeto*, Volume 2, IBPI Press, Livraria e Editora Infobook S.A., 1998

[52] Tirelo, F. ; Bigonha, R.S. ; Valente, M.T.O & Bigonha, M.A.S. Programação Orientada por Aspectos, IN *JAI 2004*, SBC, Salvador.

[53] von Staa, A. *Programação Modular*. Editora Campus, 2000.

[54] Valente, M. T. *Engenharia de Software Moderna*, Amazon, Submarino, Mercado Livre, 2020.

[55] Wexelblat, R.L. (ed.) *History of Programming Languages*, Academic Press, 1981, pp 25-74.

Índice Remissivo

Abstração
  constantes simbólicas, 35
Acoplamento
  por conteúdo, 74
  por controle, 78
  por dado comum, 75
  por elemento externo, 77
  por inclusão, 76
  por informação, 79
  por referência, 78

Camadas
  fluxo de mensagens, 84
  interface, 85, 86
  negócios, 85, 87
  persistência, 85, 88
  sistema, 85, 88
Classes
  aninhadas, 24
  depositárias, 66
  fantasma, 66
  ficha-de-dados, 24
  genéricas, 27, 47
  polivalentes, 67
Coesão
  coincidente, 72
  comunicacional, 72
  de classes, 70
  de métodos, 67
  de módulos, 71
  funcional, 73
  informacional, 73
  lógica, 72
  procedimental, 72
  temporal, 72
Contrato
  exceção disciplinada, 96
  invariante, 94
  pós-condição, 94
  pré-condição, 94
  princípio, 96
  subcontratação, 98
Critérios
  composibilidade, 34
  continuidade, 34, 35
  decomposibilidade, 34
  inteligibilidade, 34
  proteção modular, 34, 35
  reusabilidade, 29

Especificação
  algébrica, 20

métodos formais, 3
requisitos, 3, 4
validação, 6
verificaçao, 6
Extensibilidade
macros, 14
pré-processadores, 13
tipo abstrato de dados, 14

Fases
análise, 33
implementação, 33
projeto, 33
Fatores de Qualidade
externos, 2
internos, 2
Fatores Externos
compatibilidade, 2, 5
correção, 2
eficiência, 2, 4
extensibilidade, 2, 4, 40, 84
facilidade de uso, 2, 5
integridade, 2, 6
portabilidade, 2, 5, 84
reusabilidade, 2, 5, 40
robustez, 2, 3
verificabilidade, 2, 6
Fatores Internos
legibilidade, 6, 9
manutenibilidade, 6, 84
modularidade, 6, 9

Linguagens
Algol 60, 13
C, 61
C++, 18
CLU, 31, 81
Cobol, 13
Eiffel, 18
Fortran, 13
Java, 18
Lisp, 5, 43
Modula-2, 18, 61
PL/I, 13
Python, 18
Simscript, 13
Simula-67, 18, 43
Smalltak, 150, 224
Smalltalk, 5, 43

Módulos
abstração de dados, 47
função, 47
módulo-abstração, 64
módulo-função, 62
módulo-mistifório, 66
módulo-paramétrico, 65
módulo-tipo, 64
procedimento, 47
tipo abstrato, 47
tipo parametrizado, 47
Manutenção
custo, 7

manutenibilidade, 7
Metodologia
análise oo, 41
lacuna semântica, 11, 17
Modularidade
baixa conectividade, 46
classes aninhadas, 56
conectividade, 48
continuidade, 48
encapsulação, 22, 46
inteligibilidade, 48
interface explícita, 46
interface pequena, 46
manutenção, 48
ocultação de informação, 46
proteção, 48
reúso, 25, 101
tad, 11
unidade focal, 46
unidade linguística, 46

Objetos
comportamento, 18
contêineres, 25
estado abstrato, 237
estado concreto, 237
grafo, 42
máquina de estado, 232

Padrões de Projeto
abstract factory, 157
adapter, 167
bridge, 172
builder, 162
chain-of-responsability, 192
command, 195
composite, 174
decorator, 177
façade, 180
factory, 153
flyweight, 184
interpreter, 212
iterator, 206
mediator, 208
memento, 187
observer, 198
prototype, 160
proxy, 164
singleton, 151
state, 190
strategy, 201
template method, 204
visitor, 219
Pesquisadores
Alan Kay, 43
Ambler Scott, 91
B. Liskov, 31, 43, 81
B. Meyer, 10, 100, 110, 148
B.P. Lintz, 7
C.A.R. Hoare, 10
E.B. Swanson, 7
Erich Gamma, 148

Franklin DeRemer, 10
G.J. Myers, 60
Peter Coad, 148
R.C. Martin, 148
René Descartes, 42
Princípios
constantes simbólicas, 112
finalidade principal, 115
interface, 126
inversão, 127
resiliência, 129
responsabilidade, 145
segregação, 141
substituição, 135
Programação
ágil, 43
orientada por aspectos, 10
orientada por objetos, 10, 40
ponto grande, 1, 10
ponto pequeno, 1, 10
Projeto
bottom-up, 36, 39, 40
top-down, 36, 39

Qualidade
efeito colateral, 227

Reúso
componente aberto, 102
componente adaptável, 105
componente original, 103
macros, 14

Tipos
abstração, 12
abstratos de dados, 16, 18, 19, 29, 38, 40, 43, 64, 65
assinatura de operações, 21
encapsulação, 17
interfaces, 21, 25
parametrizados, 25, 27