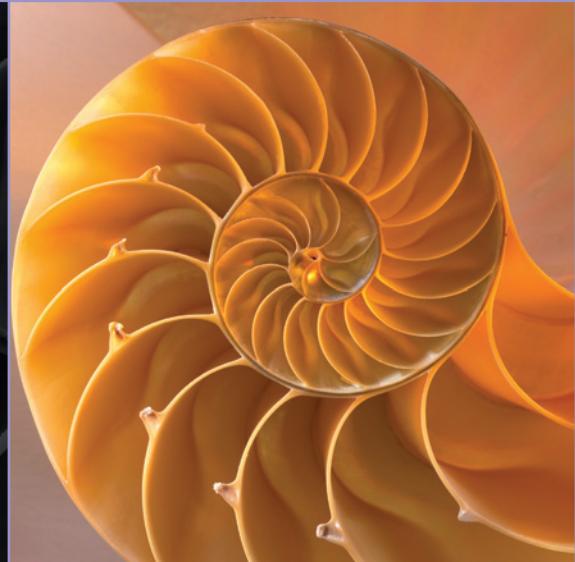


PADRÕES DE IMPLEMENTAÇÃO

UM CATÁLOGO DE PADRÕES
INDISPENSÁVEL PARA O DIA
A DIA DO PROGRAMADOR

KENT BECK



Kent Beck foi pioneiro em tecnologias orientadas a pessoas, como JUnit, Programação Extrema (XP) e padrões para desenvolvimento de software. O interesse de Beck é ajudar equipes a encontrar um estilo de desenvolvimento de software que satisfaça simultaneamente suas restrições econômicas, estéticas, emocionais e práticas. Seus livros buscam atingir positivamente a vida de todos aqueles que atuam na área.



B393p Beck, Kent.
Padrões de implementação [recurso eletrônico] / Kent Beck ; tradução: Jean Felipe Patikowski Cheiran ; revisão técnica: Marcelo Soares Pimenta. – Dados eletrônicos. – Porto Alegre : Bookman, 2013.

Editedo também como livro impresso em 2013.
ISBN 978-85-65837-96-5

1. Ciência da computação. 2. Engenharia de software – Padrões de implementação. I. Título.

CDU 004.4'2

Kent Beck

Padrões de Implementação

Tradução:

Jean Felipe Patikowski Cheiran

Revisão técnica:

Marcelo Soares Pimenta

Doutor em Informática pela Université Toulouse 1, França
Professor do Instituto de Informática da Universidade Federal do Rio Grande do Sul

Versão impressa
desta obra: 2013



2013

Obra originalmente publicada sob o título
Implementation Patterns, 1st Edition.
ISBN 9780321413093

Authorized translation from the English language edition, entitled IMPLEMENTATION PATTERS,1st Edition, 0321413091,by BECK,KENT, published by Pearson Education, Inc., publishing as Addison-Wesley Professional,Copyright © 2008. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education,Inc.

Portuguese language edition published by Bookman Companhia Editora Ltda., a Grupo A Educação S.A. company, Copyright © 2013

Tradução autorizada a partir do original em língua inglesa da obra intitulada IMPLEMENTATION PATTERNS,1^a Edição, 0321413091,autoria de BECK,KENT,publicado por Pearson Education, Inc., sob o selo Addison-Wesley Professional, Copyright © 2008. Todos os direitos reservados. Este livro não poderá ser reproduzido nem em parte nem na íntegra, nem ter partes ou sua íntegra armazenado em qualquer meio, seja mecânico ou eletrônico, inclusive fotorreprografia, sem permissão da Pearson Education,Inc.

A edição em língua portuguesa desta obra é publicada por Bookman Companhia Editora Ltda., uma empresa do Grupo A Educação S.A, Copyright © 2013

Gerente editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Coordenadora editorial: *Juliana Lopes Bernardino*

Assistente editorial: *Carina de Lima Carvalho*

Capa: *Crayon Editorial*

Crédito da imagem: ©ALEAIMAGE, 2008.©iStockphoto.com

Preparação de originais: *Pedro Barros*

Leitura final: *Laura Ávila de Souza*

Editoração: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
BOOKMAN EDITORA LTDA., uma empresa do GRUPO A EDUCAÇÃO S.A.
Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre – RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

Unidade São Paulo
Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center
Vila Anastácio – 05095-035 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Para Cindee: muito obrigado por seu encorajamento, sua insistência, sua calma, sua irritação, sua contribuição ao livro e seu chá. A dedicatória de um livro é como uma uva-passa para um elefante em comparação ao que você me deu.
Deus te abençoe.

Prefácio

Este é um livro sobre programação, especificamente sobre como escrever códigos de forma que outras pessoas consigam lê-los. Não existe mágica para isso; é como qualquer escrita: conheça seu público, tenha uma estrutura global clara em mente e expresse os detalhes de maneira que contribuam para o todo. A linguagem Java oferece bons meios de comunicar. Os padrões* de implementação aqui descritos são hábitos de programação em Java que possibilitam um código legível.

Outra forma de olhar para os padrões de implementação é pensando: “o que quero dizer ao leitor sobre este código?”. Programadores gastam muito tempo em suas próprias cabeças, e por isso tentar olhar para o mundo do ponto de vista de outra pessoa pode ser considerado uma grande mudança. Não é mais apenas “o que o computador fará com este código?”, mas “como posso comunicar para as pessoas o que estou pensando?”. Essa mudança de perspectiva é saudável e potencialmente proveitosa, uma vez que grande parte do dinheiro gasto em desenvolvimento de software é usado para entender códigos existentes.

Há um jogo na televisão americana chamado *Jeopardy*, em que o apresentador fornece respostas e os participantes tentam adivinhar as perguntas. “Uma palavra que descreve ser jogado por uma janela.” “O que é defenestração?” “Correto!”.

Codificar é exatamente como esse jogo: Java fornece respostas na forma de construtos básicos, e os programadores geralmente têm de adivinhar por conta própria quais questões e problemas são resolvidos por cada construto da linguagem. Se a resposta é “declare um campo como um Set.”, a questão poderia ser “como posso dizer aos outros programadores que uma coleção não contém duplicatas?”. O padrão de implementação fornece um catálogo de problemas comuns em programação e das funcionalidades de Java que tratam desses problemas.

Controlar o escopo é tão importante na escrita de livros quanto em desenvolvimento de software. Este livro não é várias coisas: não é um guia de

* N. de T.: “Padrão” é a tradução comumente adotada no Brasil para *pattern*, em inglês.

estilo, pois contém muita explicação e deixa as decisões finais para o leitor; não é um livro de projetos, pois está mais preocupado com decisões em pequena escala, como as que programadores tomam diariamente; não é um livro sobre padrões, pois o formato dos padrões é idiosíncratico e *ad hoc* (literalmente, “construído para um propósito específico”); não é um livro de linguagem, pois, embora cubra muitas funcionalidades de Java, assume que os leitores já sabem Java.

Na verdade, este livro foi feito sobre uma premissa bastante frágil: a importância de se ter um código bom. Vi tanto código feio render muito dinheiro, que passo a acreditar que basta um código ter a qualidade mínima ou suficiente para obter sucesso comercial ou ser amplamente utilizado. Entretanto, ainda acredito que a qualidade importa, mesmo que não forneça controle sobre o futuro. Empresas capazes de desenvolver produtos e lançá-los com confiança, mudar de direção em resposta a oportunidades e à concorrência e manter o otimismo diante de desafios e reveses tenderão a ser mais bem-sucedidas que aquelas que apresentam código de má qualidade e com defeitos.

Mesmo que não houvesse impacto econômico a longo prazo como consequência de cuidado na codificação, eu ainda optaria por escrever o melhor código possível. Setenta anos contêm pouco mais de 2 bilhões de segundos. São poucos segundos para ficar gastando em trabalhos dos quais não me orgulhe. Codificar bem dá satisfação: tanto durante a programação como por saber que outros serão capazes de entender, apreciar, usar e ampliar meu trabalho.

Enfim, então, este livro é sobre responsabilidade. Como programador, você recebeu tempo, talento, dinheiro e oportunidade. Como fará um uso responsável desses bens? As páginas a seguir trazem minha resposta para a questão: codificar para os outros tão bem quanto para mim mesmo e minha companheira, a CPU.

Agradecimentos

Primeiramente, por último e sempre, quero agradecer a Cynthia Andres, minha companheira, editora, apoiadora e chefe severa. Ao meu amigo Paul Petralia, que pegou este projeto desde o começo e fez telefonemas encorajadores no decorrer. Ao meu editor Chris Guzikowski, com quem aprendi a trabalhar junto durante o andamento do projeto. Ele me deu o apoio que precisava por parte da Pearson para terminar o livro. Obrigado à equipe de produção da Pearson: Julie Nahil, John Fuller e Cynthia Kogut. À Jennifer Kohnke, que produziu as ilustrações, que combinam informação e humanidade. Aos meus revisores, que forneceram feedback claro e em tempo para os rascunhos: Erich Gamma, Steve Metsker, Diomidis Spinellis, Tom deMarco, Michael Feathers, Doug Lea, Brad Abrams, Cliff Click, Pekka Abrahamson, Gregor Hohpe e Michele Marchesi. Muito obrigado, David Saff, por reconhecer a simetria entre estado e comportamento. Aos meus filhos, que permaneceram em casa me lembrando por que eu queria terminar: Lincoln, Lindsey, Forrest e Joëlle Andres-Beck.

Sumário

Capítulo 1	Introdução.....	1
	Guia do livro	3
	E agora...	4
Capítulo 2	Padrões	5
Capítulo 3	Uma teoria de programação.....	9
	Valores.....	10
	Comunicação.....	10
	Simplicidade	11
	Flexibilidade	12
	Princípios.....	13
	Consequências locais	13
	Minimizar repetição.....	13
	Lógica e dados em harmonia.....	14
	Simetria	14
	Expressão declarativa	15
	Ritmo de mudança.....	16
	Conclusão.....	17
Capítulo 4	Motivação	19
Capítulo 5	Classe	21
	Classe	22
	Nome simples de superclasse	23
	Nome qualificado de subclasse	24
	Interface abstrata	24
	Interface.....	25
	Classe abstrata.....	26

Interface versionada	27
Objeto valor	28
Especialização	30
Subclasse	31
Implementador	33
Classe interna	33
Comportamento específico de instância	34
Condicional	34
Delegação	36
Seletor plugável	38
Classe interna anônima	39
Classe biblioteca	39
Conclusão	41
 Capítulo 6 Estado.....	 43
Estado	44
Acesso	45
Acesso direto	46
Acesso indireto	46
Estado comum	47
Estado variável	48
Estado extrínseco	49
Variável	50
Variável local	50
Campo	52
Parâmetro	53
Parâmetro coletor	54
Parâmetro opcional	55
Var args (quantidade variável de argumentos)	55
Objeto parâmetro	56
Constante	57
Nome sugestivo de função	57
Tipo declarado	59
Inicialização	59
Inicialização ansiosa	60
Inicialização preguiçosa	60
Conclusão	61
 Capítulo 7 Comportamento	 63
Fluxo de controle	64
Fluxo principal	64

Mensagem	65
Mensagem de escolha	65
Envio duplo	66
Mensagem de decomposição (sequenciamento).....	67
Mensagem inversa	67
Mensagem convidativa	68
Mensagem explicativa.....	69
Fluxo excepcional	69
Cláusula de guarda	70
Exceção	72
Exceções verificadas.....	72
Propagação de exceção	73
Conclusão.....	73
 Capítulo 8 Métodos	75
Método composto.....	77
Nome revelador de intenção	79
Visibilidade de método.....	80
Objeto método.....	81
Método sobrescrito.....	83
Método sobrecarregado	83
Tipo de retorno de método	84
Comentário de método	84
Método auxiliar.....	85
Método de impressão para depuração.....	86
Conversão	86
Método de conversão.....	87
Construtor de conversão.....	87
Criação	88
Construtor completo.....	88
Método fábrica	89
Fábrica interna.....	90
Método acessador de coleção.....	90
Método de atribuição booleana	92
Método de consulta	92
Método de igualdade	93
Método get	94
Método set.....	95
Cópia de segurança.....	96
Conclusão.....	97

Capítulo 9	Coleções	99
	Metáforas	100
	Questões	101
	Interfaces	102
	Array	103
	Iterable	103
	Collection	104
	List	104
	Set	104
	SortedSet	105
	Map.....	106
	Implementações	106
	Collection.....	107
	List	107
	Set	108
	Map.....	109
	Collections.....	109
	Busca	110
	Ordenação	110
	Coleções imutáveis.....	111
	Coleções com apenas um elemento	112
	Coleções vazias	112
	Estendendo coleções	113
	Conclusão.....	113
Capítulo 10	Evoluindo frameworks.....	115
	Mudando frameworks sem mudar aplicações	115
	Atualizações incompatíveis	116
	Encorajando mudanças compatíveis	118
	Classe biblioteca	119
	Objetos.....	119
	Conclusão.....	127
Apêndice	Medindo desempenho.....	129
	Exemplo	129
	API	130
	Implementação	131
	MethodTimer	132
	Cancelando sobrecarga	134

Testes.....	134
Comparando coleções.....	135
Comparando ArrayList e LinkedList.....	137
Comparando conjuntos	138
Comparando mapas.....	140
Conclusão.....	141
Leituras sugeridas.....	143
Programação geral	143
Filosofia.....	144
Java	145
Índice	147
Índice de padrões	154

CAPÍTULO 1

Introdução

Estamos juntos nesta. Você pegou meu livro (ele agora é seu). Provavelmente, você já escreve códigos e, assim, já desenvolveu um estilo próprio por meio de suas experiências.

O objetivo deste livro é ajudá-lo a comunicar suas intenções por meio do código. Ele começa com um panorama sobre programação e padrões (Capítulos 2 a 4) e a seguir traz uma série de pequenos ensaios, padrões e dicas para usar as características do Java a fim de escrever um código legível (Capítulos 5 a 8). No fim, há um capítulo sobre como modificar as recomendações aqui apresentadas caso se esteja escrevendo frameworks em vez de aplicações. De modo geral, o texto está focado em técnicas de programação para melhorar a comunicação.

Há vários passos para se comunicar por meio de códigos. Primeiramente, tive de adquirir essa consciência para programar. Já programava há anos quando comecei a escrever padrões de implementação. Fiquei surpreso ao perceber que, apesar de chegar rápida e facilmente às decisões de programação, eu não conseguia explicar por que tinha tanta certeza de que um método deveria ser chamado de tal jeito ou um pedaço da lógica pertencia a tal objeto. O primeiro passo para me comunicar foi desacelerar, até ter consciência do que havia em minha mente, e parar de fingir que codificava por instinto.

O segundo passo foi reconhecer a importância das outras pessoas. Considerava gratificante programar, mas sou egocêntrico. Antes de escrever um código comunicativo, precisava acreditar que as outras pessoas eram tão importantes quanto eu. Programar dificilmente é uma comunicação solitária entre um homem e uma máquina. Importar-se com as outras pessoas é uma decisão deliberada e que requer prática.

E isso me leva ao terceiro passo. Depois de expor minhas ideias à luz do sol e ao ar fresco e de reconhecer que outras pessoas têm tanto direito de existência quanto eu, precisava demonstrar na prática minha nova perspectiva. Uso os padrões de implementação deste livro para programar de maneira consciente tanto para outras pessoas quanto para mim.

Pode-se ler este livro estritamente por seu conteúdo técnico – truques úteis com explicações. Entretanto, acho justo alertá-lo de que há muito mais aqui, ao menos para mim.

É possível que você encontre esses detalhes técnicos folheando os capítulos sobre padrões. Uma estratégia eficaz para aprender essa matéria é lê-la pouco antes de precisar usá-la. Como ferramenta de consulta, sugiro pular direto para o Capítulo 5 e percorrê-lo rapidamente até o fim, mantendo o livro ao seu lado enquanto programa. Depois de ter usado muitos dos padrões, você poderá, então, voltar ao material introdutório para descobrir os antecedentes filosóficos por trás das ideias que esteve usando.

Se estiver interessado em uma compreensão completa do livro, deve lê-lo integralmente, desde as primeiras páginas. Diferentemente da maioria dos livros, os capítulos deste são um pouco longos, de forma que é necessário concentração para ler do início ao fim.

Boa parte do material aqui contido está organizada como padrões. Quase não existe mais ineditismo em decisões de programação. É possível que um programador nomeie 1 milhão de variáveis em sua carreira, mas não irá propor um nome totalmente novo para cada variável. As restrições gerais para se dar nomes são sempre as mesmas: é preciso transmitir o propósito, o tipo e o tempo de vida da variável aos leitores; é preciso escolher um nome que seja fácil de ler; é preciso escolher um nome que seja fácil de escrever e formatar. Adicione a essas restrições gerais as especificidades da variável e terá um nome funcional. Nomear variáveis é um exemplo de padrão: a decisão e suas restrições se repetem, ainda que se possa criar um nome diferente a cada vez.

Alguns padrões precisam de distintas apresentações. Às vezes uma experiência explica melhor um padrão; outras vezes é um diagrama, ou um caso anterior, ou um exemplo. Em vez de encaixar a definição de cada padrão em um formato rígido, descrevi cada um da maneira que achei melhor.

O livro contém 77 padrões nomeados explicitamente, cada um cobrindo algum aspecto de como escrever um código legível. Além disso, há muitos padrões menores, ou variações de padrões, que menciono de passagem. Meu objetivo é oferecer recomendações de como realizar as tarefas de codificação mais comuns para ajudar futuros leitores a entenderem o que o código deve fazer.

Este livro pode ser classificado em algum ponto entre *padrões de projeto* e manual da linguagem Java. *Padrões de projeto* são decisões que se pode tomar algumas vezes por dia enquanto programa, as quais costumam regular a interação entre objetos. Quando se está codificando, aplicam-se vários padrões de implementação por minuto. Embora manuais de linguagem sirvam para descrever o que se pode fazer com Java, eles não explicam muito bem por que usar certo constructo ou o que alguém que leia seu código irá concluir.

Parte de minha filosofia ao escrever foi focar em tópicos que conheço bem. Questões de concorrência, por exemplo, não são abordadas nos padrões de implementação; não porque seja uma questão sem importância, mas porque não tenho muito a dizer sobre ela. Minha estratégia sempre foi isolar tanto quanto possível as partes concorrentes de minhas aplicações. Embora em geral isso tenha funcionado, não é algo que eu consiga explicar. Recomen-

do um livro do tipo *Concorrência em Java na prática* para se ter uma visão melhor sobre o tema.

Outro tópico não abordado são noções de processo de software. A dica sobre como se comunicar pelo código aqui apresentada funciona independentemente de o código ser escrito próximo do final de um longo ciclo ou segundos depois de um teste que fallhou. O bom é ter um software que custe menos no final, quaisquer que sejam as armadilhas sociológicas sob as quais foi escrito.

Também me recuso comentar os limites de Java. Tendo a ser conservador em minhas escolhas tecnológicas, pois fracassei muitas vezes forçando novas funcionalidades até seu limite (é uma boa estratégia de aprendizado, mas muito arriscada para a maioria dos desenvolvimentos). Assim, aqui há um subconjunto básico de Java. Se o leitor estiver motivado a usar as últimas funcionalidades de Java, poderá aprendê-las em outras fontes.

Guia do livro

Este livro está dividido em sete seções, como mostra a Figura 1.1:

- Introdução – capítulos curtos que descrevem a importância e o valor da comunicação por meio do código e a filosofia por trás dos padrões.
- Classe – padrões que descrevem como e por que se podem criar classes e como classes codificam a lógica.
- Estado – padrões para armazenar e recuperar estados.

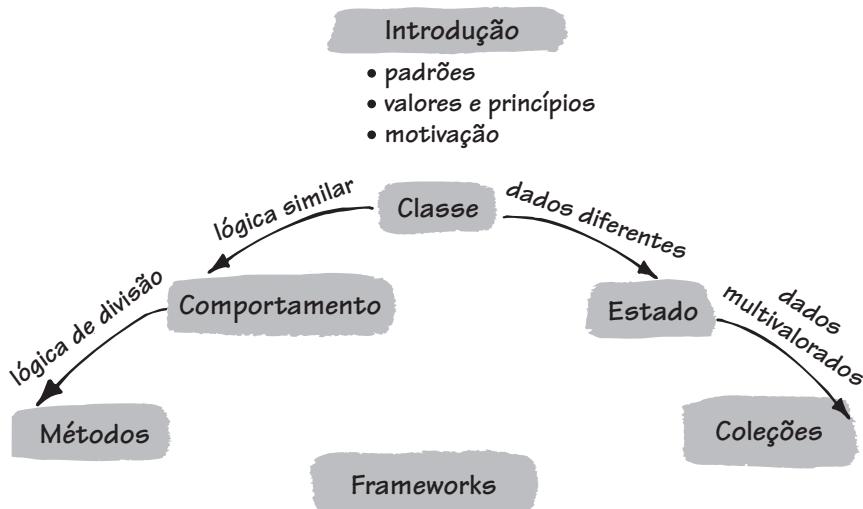


Figura 1.1 Visão geral do livro.

- Comportamento – padrões para a representação lógica, especialmente fluxos alternativos.
 - Métodos – padrões para escrever métodos, lembrando o que leitores provavelmente concluirão de sua escolha de decomposição e nomeação de métodos.
 - Coleções – padrões para escolher e usar coleções.
 - Frameworks expandidos – variações dos padrões anteriores quando se construem frameworks em vez de aplicações.
-

E agora...

... a essência do livro. Se você pretende ler este livro integralmente, apenas vire a página (suponho que descubra isso sozinho). Mas, se quiser navegar pelos padrões, comece pelo Capítulo 5. Boa implementação!

CAPÍTULO 2

Padrões

Muitas decisões na programação são únicas. A forma como se programa um site é bem diferente de como se constrói um marcapasso. Entretanto, à medida que as decisões se tornam cada vez mais técnicas, vem uma sensação de familiaridade. “Acho que acabei de escrever esse código...” A codificação seria mais eficaz se programadores gastassem menos tempo nas partes mundanas e repetitivas do trabalho e tivessem mais tempo para resolver problemas verdadeiramente únicos.

A maioria dos programadores segue um pequeno conjunto de regras:

- É mais frequente ler programas do que escrevê-los.
- Não existe esta coisa de “terminado”. Muito mais investimento será feito na modificação de programas do que em seu desenvolvimento inicial.
- Programas são estruturados a partir de um conjunto básico de conceitos de estado e fluxo de controle.
- Leitores precisam entender os detalhes e o conceito dos programas. Às vezes vão dos detalhes ao conceito; outras vezes, do conceito aos detalhes.

Padrões baseiam-se nessas regras gerais. Por exemplo, todo programador tem de decidir como estruturar uma iteração. Quando se está pensando em como escrever um laço, a maioria das questões de domínio já está resolvida e restam apenas questões puramente técnicas: o laço deve ser fácil de ler, fácil de escrever, fácil de verificar, fácil de modificar e eficiente.

Essa lista de interesses é o começo de um padrão. As restrições ou **forças** listadas afetam como é escrito cada laço de um programa. As forças reincidentem de modo previsível, e esta é a razão pela qual um padrão é um padrão: é um padrão de forças.

Há algumas formas razoáveis de escrever um laço, e cada uma implica prioridades diferentes entre as restrições. Se desempenho é o mais importante, pode-se estruturar o laço de uma forma; contudo, se o mais importante for facilidade de modificação, pode-se estruturá-lo diferentemente.

Cada padrão ilustra um ponto de vista sobre as prioridades relativas das forças. A maioria dos padrões vem com um pequeno ensaio sobre as alternativas para resolver o problema e os motivos por que a solução recomendada é melhor. Revelar o raciocínio por trás da recomendação de um padrão convida os leitores a decidirem sozinhos como preferem abordar um problema recorrente.

Desse modo, cada padrão também vem com a semente de uma solução. O padrão para laços sobre uma coleção pode sugerir: “Use o laço `for` de Java5 para expressar iteração”. Padrões levam princípios abstratos à prática, ajudando na escrita do código.

Padrões trabalham juntos. O padrão que sugere um laço `for` introduz o problema sobre qual nome dar à variável do laço. Em vez de englobar tudo em um só padrão, há outro padrão cobrindo especificamente o nome que se dá a variáveis.

Neste livro, o estilo de apresentação de padrões varia consideravelmente. Às vezes eles recebem nomes evidentes, com seções para discutir as forças e a solução. Alguns dos padrões menores, no entanto, estão embutidos em um padrão maior. Podem ser suficientes uma ou duas sentenças para a discussão sobre um padrão.

Trabalhar com padrões pode parecer restritivo, mas o seu uso talvez economize tempo e energia. Por exemplo, arrumar a cama exige menos energia se for um hábito do que quando se pensa em cada etapa do processo, planejando a ordem certa todas as vezes que for fazer isso. Há um padrão definido para se arrumar a cama que simplifica muito a tarefa. Se a cama está contra a parede ou o lençol é muito pequeno, adapta-se a estratégia à situação, mas, arrumar a cama pode ser uma rotina, liberando sua mente para tarefas mais interessantes e únicas. Quando um padrão se torna um hábito, é interessante não ter de fazer um debate só para escrever um laço. Se a equipe está insatisfeita com um padrão, ela pode discutir as opções para introduzir um novo padrão.

Nenhum conjunto de padrões funciona em todas as situações de programação. Mais adiante neste livro, estão os padrões que uso e que funcionaram bem no desenvolvimento de aplicações (com uma breve incursão no desenvolvimento de framework). Copiar cegamente o estilo de alguém não é tão eficaz quanto pensar a respeito e praticar seu próprio estilo, além de discutir e compartilhar um estilo com sua equipe.

Padrões funcionam melhor no auxílio para a tomada de decisões. Alguns padrões de implementação acabarão chegando às linguagens de programação, assim como usos estruturados de `setjmp()`/`longjmp()` tornaram-se o sistema atual para tratamento de exceções. Contudo, costuma ser necessário adaptar um padrão antes de usá-lo.

Este capítulo começou com a busca por uma forma mais barata, mais rápida e que consuma menos energia para resolver problemas comuns em programação. Usar padrões ajuda programadores a escreverem soluções razoáveis para problemas comuns, deixando mais tempo, energia e criatividade para aplicarem em problemas verdadeiramente únicos. Cada padrão une um problema comum de programação com uma discussão sobre os fatores que causam aquele problema, fornecendo recomendações concretas de como criar rapidamente

uma solução satisfatória. O resultado é um trabalho melhor, mais barato e mais rápido nas partes tediosas dos programas e mais tempo e energia para investir em problemas únicos de cada aplicação.

O próximo capítulo, “Uma teoria de programação”, descreve os valores e princípios subjacentes ao estilo de programar descrito por esses padrões de implementação.

CAPÍTULO 3

Uma teoria de programação

Nenhuma lista de padrões, não importa quão exaustiva seja, cobre todas as situações que surgem durante uma programação. Eventualmente (ou mesmo frequentemente), surge uma situação em que nenhuma solução ordinária se encaixa. Essa necessidade por abordagens gerais para problemas específicos é uma das razões para estudarmos a teoria da programação. Outra é alcançar uma sensação de segurança como resultado de saber o que fazer e por que fazer. Conversas sobre programação são mais interessantes quando envolvem teoria e prática.

Cada padrão carrega consigo um pouco de teoria. Há, no entanto, forças maiores e mais universais na programação do que aquelas que são cobertas por um padrão específico. Esta seção descreve essas preocupações transversais, as quais são divididas em dois tipos: valores e princípios. Os valores são os grandes temas universais da programação. Quando se está trabalhando bem, preza-se pela importância da comunicação com outras pessoas, removendo-se excessos de complexidade do código e mantendo-se abertas as opções. Tais valores – comunicação, simplicidade e flexibilidade – colorem todas as decisões que devem ser tomadas durante a programação.

Os princípios aqui descritos não são de tão longo alcance ou tão universais quanto os valores, mas todos estão presentes em muitos padrões. Os princípios criam pontes entre os valores (que são universais, mas muitas vezes difíceis de aplicar diretamente) e os padrões (que têm aplicação clara, mas específica). É muito importante explicitar os padrões para as situações em que nenhum padrão se aplica ou para aquelas em que se aplicam dois padrões mutuamente exclusivos. Diante de uma ambiguidade, conhecer os princípios permite que se “faça algo” consistente com o restante do trabalho e, provavelmente, seja bem-sucedido nisso.

Esses três elementos – valores, princípios e padrões – formam uma equação equilibrada para um estilo de desenvolvimento. Os padrões descrevem o que fazer, os valores fornecem motivação, e os princípios ajudam a traduzir a motivação em ação.

Os valores, princípios e padrões aqui presentes foram tirados de minhas práticas e reflexões e de conversas com outros programadores. Todos aproveitamos as experiências de gerações anteriores de programadores. O resultado é *um* estilo de desenvolvimento, e não o estilo de desenvolvimento. Valores e princípios diversos levarão a diferentes estilos. Uma das vantagens de se definir esse estilo em termos de valores, princípios e práticas é que fica mais fácil que os conflitos sobre programação sejam produtivos. Quando uma pessoa quer fazer algo de um jeito, e um colega, de outro, pode-se identificar o nível de desacordo e evitar perda de tempo. Se a discordância é em relação a princípios, discutir sobre onde colocar chaves não resolverá o conflito.

Valores

Três valores consistentes com a excelência em programação são comunicação, simplicidades e flexibilidade. Embora às vezes entrem em conflito, com frequência eles complementam. Os melhores programas oferecem muitas opções para extensões futuras, não contêm elementos estranhos e são fáceis de ler e entender.

Comunicação

Um código comunica bem quando um leitor consegue entendê-lo, modificá-lo ou usá-lo. Embora a programação estimule a pensar apenas no computador, coisas boas acontecem quando se pensa nas pessoas enquanto se programa. Assim, obtém-se um código mais claro, mais fácil de ler, que tem melhor relação custo-benefício. Além disso, o pensamento flui melhor, veem-se as coisas sob outra perspectiva, o nível de estresse diminui e atingem-se algumas necessidades sociais. Parte da atração pela programação é a oportunidade de conversar com algo fora de si mesmo. Entretanto, ninguém quer lidar com seres humanos difíceis, incompreensíveis e irritantes. Programar como se pessoas realmente não existissem, construir castelos de açúcar bem elaborados apenas na mente, perde a cor e a graça em pouco tempo.

Uma das primeiras ideias que me levaram ao foco em comunicação foi a programação literária (*Literate Programming*), de Knuth, isto é, um programa deve ser lido como um livro. Deve ter enredo, ritmo e agradáveis trocas de frase. Quando Ward Cunningham e eu lemos sobre programas literários pela primeira vez, decidimos tentar. Pegamos um trecho bem limpo de código em imagem Smalltalk, o ScrollController, e tentamos transformá-lo em uma história. Horas depois, tínhamos reescrito completamente o código à nossa maneira, gerando um artigo científico. Toda vez que nos deparamos com um trecho de lógica um tanto difícil de explicar, foi mais fácil reescrever o código do que explicar por que era difícil comprehendê-lo. As necessidades da comunicação mudaram nossa perspectiva sobre codificação.

Há um bom argumento econômico para se focar em comunicação enquanto se programa. A maioria dos custos de um software incorre depois de ele ser implantado. Pensando em minha experiência de modificação de códigos, vejo que gastei muito mais tempo lendo o código existente do que escrevendo o novo. Quem quer fazer um código barato, consequentemente deve fazê-lo fácil de ler.

Por ser mais realista, o foco na comunicação melhora o pensamento. Parte disso se deve ao fato de aproveitar mais o cérebro. Pensar “Como outra pessoa veria isso?” ativa neurônios diferentes daqueles ativados quando se está apenas concentrado em si e no computador. Afasta-se um pouco aquela perspectiva isolada e vê-se novamente o problema e a solução. Outro benefício é a redução da tensão por saber que se está cuidando dos negócios, fazendo a coisa certa. Por fim, por sermos uma espécie social, considerar explicitamente questões sociais é mais realista do que trabalhar fingindo que elas não existem.

Simplicidade

Em *The Visual Display of Quantitative Information*, Edward Tufte traz um exercício em que se pega um gráfico e começa a apagar todas as marcas que não adicionam informação. O gráfico resultante é novo e muito mais fácil de entender que o original.

Eliminar complexidade excessiva permite que aqueles que leem, usam e modificam programas os entendam mais rapidamente. Um pouco de complexidade é essencial, refletindo precisamente a complexidade do problema a ser resolvido. Boa parte da complexidade, contudo, representa as marcas de unhas em nossa luta para fazer o programa rodar a todo o custo. É esse excesso que diminui o valor do software, tornando menos provável que ele execute corretamente e aumentando a dificuldade de se fazerem mudanças bem-sucedidas no futuro. Parte da programação consiste em olhar para o que se fez até então e separar o joio do trigo.

A simplicidade está no olho de quem vê. O que é simples para um programador especialista, familiarizado com o poder de suas ferramentas, pode ser avassaladoramente complexo para um iniciante. Assim como um bom livro é escrito tendo-se o público-alvo em mente, bons programas também são escritos da mesma forma. É legal desafiar um pouco o público, mas complexidade em demasia o afasta.

A computação avança em ondas de complexidade e simplificação. Arquiteturas mainframe foram ficando cada vez mais barrocas até chegarem os minicomputadores. Estes não resolveram todos os problemas de um mainframe, mas se descobriu que, para muitas aplicações, aqueles problemas não eram tão importantes. Linguagens de programação também passam por ondas em que se tornam mais complexas e, depois, mais simples. C produz C++, que produz Java, que está ficando mais complicada.

Buscar simplicidade permite inovação. JUnit era muito mais simples que as ferramentas de teste que substituiu, ao gerar uma variedade de similares,

add-ons e novas técnicas de programação e teste. A última versão, JUnit 4,* perdeu aquele aspecto de “básico”, embora eu tenha gostado ou concordado com cada uma das decisões complexificadoras. Um dia, alguém criará uma forma muito mais simples que a JUnit para programadores escreverem testes, e essa nova ideia possibilitará uma nova onda de inovações.

Aplique simplicidade em todos os níveis. Formate o código de modo que nenhuma linha possa ser apagada sem que se perca informação. Projete sem elementos estranhos. Desafie os requisitos para descobrir quais são essenciais. Eliminar o excesso de complexidade ilumina o código restante, criando a oportunidade de abordá-lo de outra maneira.

Comunicação e simplicidade frequentemente andam lado a lado. Quanto menos complexidade em excesso, mais fácil é a compreensão do sistema. Quanto mais se foca em comunicação, fica mais fácil ver qual complexidade pode ser descartada. Às vezes, entretanto, uma simplificação pode dificultar a leitura do programa, casos em que se deve preferir comunicação a simplicidade. Tais situações são raras, mas geralmente apontam para uma simplificação em larga escala que ainda não se está vendo.

Flexibilidade

Dos três valores listados aqui, a flexibilidade é a justificativa dada para práticas ineficientes de codificação e projeto. Para recuperar uma constante, vi programas procurando uma variável de ambiente com o nome de um diretório contendo um arquivo no qual é encontrado o valor da constante. Por que toda essa complexidade? Flexibilidade. Programas devem ser flexíveis, mas apenas onde eles mudam. Se a constante nunca muda, toda a complexidade é um custo sem benefício.

Como a maior parte do custo de um programa incorre depois de ele ser implantado, a mudança deve ser fácil. A flexibilidade que se imagina pode ser necessária no futuro, mas é provável que não seja o que se precisa no momento de alterar o código. É por isso que a flexibilidade derivada de simplicidade e testes extensivos é mais eficaz que a flexibilidade gerada um projeto especulativo.

Escolha padrões que estimulem flexibilidade e tragam benefícios imediatos. Para padrões com custo imediato e benefícios apenas futuros, paciência costuma ser a melhor estratégia. Guarde-os de volta até se tornarem necessários. Dessa forma, consegue-se aproveitá-los precisamente da maneira que são necessários.

Flexibilidade pode causar aumento de complexidade. Por exemplo, opções configuráveis pelo usuário fornecem flexibilidade, mas adicionam a complexidade de um arquivo de configuração e a necessidade de, durante a programação, levar em conta as opções. Simplicidade pode estimular flexibilidade. No exemplo acima, se for possível encontrar uma forma de eliminar as opções configuráveis sem perder valor, será mais fácil mudar o programa posteriormente.

* N. de T.: Na época em que este livro foi escrito, 2008.

Melhorar a comunicabilidade do software também aumenta a flexibilidade. Quanto mais pessoas puderem ler, entender e modificar o código rapidamente, mais opções se tem para mudanças futuras.

Os padrões a seguir estimulam flexibilidade, ajudando programadores a criar aplicações simples e compreensíveis que podem ser mudadas.

Princípios

Padrões de implementação têm essa forma por um motivo. Cada um deles expressa um ou mais dos valores comunicação, simplicidade e flexibilidade. Princípios estão em outro nível de ideias gerais, mais específicos para programação que os valores, mas também fundamentam os padrões.

Examinar princípios é valioso por muitas razões. Princípios claros podem levar a novos padrões, assim como a tabela periódica dos elementos levou à descoberta de novos elementos. Princípios relacionados a conceitos gerais, e não específicos, podem fornecer uma explicação para a motivação por trás de um padrão. É melhor discutir a respeito de padrões contraditórios em termos de princípios do que das especificidades dos padrões envolvidos. Por fim, entender os princípios serve como guia em situações novas.

Por exemplo, diante de uma nova linguagem de programação, usam-se os princípios para se desenvolver um estilo de programação eficaz. Não é preciso imitar estilos existentes, nem, o que é pior, aproveitar o estilo que funciona com outra linguagem de programação (é possível escrever código FORTRAN em qualquer linguagem, mas não se deve fazê-lo). O entendimento dos princípios permite que se aprenda rapidamente e aja com integridade em situações novas. A seguir, é apresentada uma lista de princípios por trás dos padrões de implementação.

Consequências locais

Estruture o código de forma que mudanças tenham consequências locais. Se uma mudança *aqui* pode causar um problema *lá*, então o custo da mudança aumenta drasticamente. Códigos cuja maioria das consequências é local comunicam-se com eficiência. Pode-se entendê-lo gradualmente, não sendo necessário primeiro um entendimento do todo.

Como manter baixo o custo das mudanças é uma motivação básica para os padrões de implementação, o princípio de consequências locais é parte do raciocínio que gera a maioria dos padrões.

Minimizar repetição

Um princípio que contribui para se manter locais as consequências é minimizar a repetição. Quando se tem o mesmo código em vários pontos, ao mudar o código, é preciso decidir se isso mudará ou não todos os outros pontos em

que aparece. Assim, a mudança deixa de ser local. Quanto mais repetições de código, mais custo terá a mudança.

Um código copiado é apenas uma forma de repetição. Hierarquias paralelas de classes também são repetitivas e rompem o princípio de consequências locais. Se fazer uma mudança conceitual requer que se mudem duas ou mais hierarquias de classes, as mudanças terão consequências espalhadas, não locais. Para melhorar novamente o código, deve-se reestruturá-lo de forma que as mudanças sejam locais.

A duplicação pode não ser um passo óbvio até ser feita, e às vezes não o é nem depois disso. E, quando ela é encontrada, nem sempre se consegue imaginar uma boa maneira de eliminá-la. A duplicação não é algo ruim, mas aumenta o custo de mudanças.

Uma das formas de remover uma duplicação é dividir o programa em vários pequenos pedaços – pequenas declarações, pequenos métodos, pequenos objetos, pequenos pacotes. Grandes fragmentos de lógica costumam duplicar partes de outros grandes fragmentos de lógica. Essa comunhão é o que torna possíveis os padrões: ainda que haja diferenças entre diferentes pedaços de código, há também muitas similaridades. Comunicar claramente quais partes do programa são idênticas, quais partes são meramente similares e quais partes são completamente diferentes torna o código mais fácil de ler e mais barato de modificar.

Lógica e dados em harmonia

Outro princípio corolário ao das consequências locais é manter em harmonia lógica e dados. Devem-se colocar a lógica e os dados sobre os quais ela opera próximos um do outro, se possível no mesmo método, ou no mesmo objeto, ou ao menos no mesmo pacote. Para fazer uma mudança, é provável que se tenha de alterar a lógica e os dados ao mesmo tempo. Se eles estão em harmonia, as consequências dessa mudança serão locais.

No início, nem sempre é óbvio onde devem ficar a lógica e os dados para se satisfazer esse princípio. Pode-se estar escrevendo código em A e perceber que são necessários dados que estão em B. Só depois de o código estar funcionando é que se nota que ele está muito distante dos dados. Assim, é preciso optar: mover o código para próximo dos dados, mover os dados para próximo do código, colocá-los juntos em um objeto auxiliar, ou entender que, naquele momento, não é possível pensar em como juntá-los de forma que se comunicarem eficazmente.

Simetria

Outro princípio que se usa o tempo todo é a simetria. Programas são cheios de simetria. Um método `add()` é acompanhado de um método `remove()`. Em um grupo de métodos, todos têm os mesmos parâmetros. Todos os campos de um objeto têm o mesmo tempo de vida. Identificar e expressar claramente a simetria

torna o código mais fácil de ler. Se os leitores puderem entender uma metade da simetria, eles poderão entender rapidamente a outra.

Costuma-se discutir simetria em termos espaciais: bilateral, rotacional e assim por diante. A simetria em programas raramente é gráfica; ela é conceitual. No código, ela está onde a mesma ideia é expressada da mesma forma em todos os lugares em que aparece no código.

Eis um exemplo de código em que falta simetria:

```
void process() {  
    input();  
    count++;  
    output();  
}
```

A segunda declaração é mais concreta que as duas mensagens. Eu reescreveria isso com base na simetria, resultando em:

```
void process() {  
    input();  
    incrementCount();  
    output();  
}
```

Mas esse método ainda viola a simetria. As operações `input()` e `output()` são nomeadas de acordo com as intenções, e `incrementCount()`, de acordo com uma implementação. Buscando simetrias, penso no *motivo* para se incrementar o contador, talvez resultando em:

```
void process() {  
    input();  
    tally();  
    output();  
}
```

Muitas vezes, encontrar e expressar simetrias é um passo preliminar para se remover uma duplicação. Se uma ideia está presente em diversos pontos do código, torná-la simétrica em todos é um bom primeiro passo visando unificá-las.

Expressão declarativa

Outro princípio por trás dos padrões de implementação é expressar declarativamente o máximo possível da intenção. Uma programação imperativa é poderosa e flexível, mas lê-la exige que se siga o fluxo de execução. Deve-se construir mentalmente um modelo do estado do programa e dos fluxos de controle e de dados. Para partes de um programa que mais parecem fatos simples, sem sequência ou condicionais, é mais fácil ler um código simplesmente declarativo.

Por exemplo, em versões mais antigas de JUnit, classes podiam ter um método estático `suite()` que retornava um conjunto de testes para execução.

```
public static junit.framework.Test suite() {  
    Test result= new TestSuite();  
    ...complicated stuff...  
    return result;  
}
```

Agora vem a questão mais comum – que testes serão executados? Na maioria dos casos, o método `suite()` apenas agrupa os testes em diversas classes. Entretanto, como o método `suite()` é genérico, para se ter certeza, é preciso ler e entender o método.

O JUnit 4, por sua vez, usa o princípio de expressão declarativa para resolver o mesmo problema. Em vez de um método rodar uma suíte de testes, há um executor especial de testes que os roda em um conjunto de classes (o caso comum):

```
@RunWith(Suite.class)  
@TestClasses({  
    SimpleTest.class,  
    ComplicatedTest.class  
})  
class AllTests {  
}
```

Sabe-se que testes estão sendo agregados usando esse método; basta olhar a anotação `TestClasses` para descobrir quais testes serão executados. Como a expressão da suíte é declarativa, não é preciso suspeitar que haja exceções traíceiras. Essa solução propicia a força e a generalidade do método original `suite()`, mas o estilo declarativo torna o código mais fácil de ler. (A anotação `RunWith` fornece ainda mais flexibilidade para testes que o método `suite()`, mas isso fica para um outro livro.)

Ritmo de mudança

Um último princípio é unir lógica e dados que mudam em um mesmo ritmo e separar lógica e dados que mudam em ritmos diferentes – esses ritmos são uma espécie de simetria temporal. Às vezes o princípio ritmo de mudança aplica-se às mudanças que um programador faz. Por exemplo, ao escrever um software de impostos, separara-se o código que faz cálculos gerais de taxas do código que é específico de um ano. O código muda em ritmos diferentes. Quando são feitas mudanças para os anos seguintes, é interessante ter a certeza de que o código de anos anteriores ainda funciona. Separá-los dá mais segurança quanto às consequências locais das mudanças.

O ritmo de mudança aplica-se a dados. Todos os campos em um objeto deveriam mudar mais ou menos no mesmo ritmo. Por exemplo, campos que são modificados apenas durante a ativação de um único método devem ser variáveis locais. Dois campos que mudam juntos, mas fora de sincronia com seus campos vizinhos, provavelmente pertencem a um objeto auxiliar. Se, em um instrumento financeiro, valor e moeda podem ser alterados juntos, então

esses dois campos provavelmente estariam melhor expressos como um objeto auxiliar Money:

```
setAmount(int value, String currency) {  
    this.value= value;  
    this.currency= currency;  
}
```

que se torna:

```
setAmount(int value, String currency) {  
    this.value= new Money(value, currency);  
}
```

e depois:

```
setAmount(Money value) {  
    this.value= value;  
}
```

O princípio de ritmo de mudança é uma aplicação de simetria, mas simetria temporal. No exemplo acima, os campos originais value e currency são simétricos, pois mudam ao mesmo tempo. Contudo, são não simétricos aos outros campos do objeto. Expressar simetria colocando-os em seu próprio objeto demonstra a relação entre eles para os leitores e, provavelmente, gera novas oportunidades de, mais tarde, reduzir a duplicação e, ainda, localizar as consequências.

Conclusão

Este capítulo introduziu os fundamentos teóricos dos padrões de implementação. Os valores *comunicação, simplicidade e flexibilidade* fornecem ampla motivação para os padrões. Os princípios *consequências locais, minimizar repetição, lógica e dados em harmonia, simetria, expressão declarativa e ritmo de mudança* ajudam a transformar os valores em ações. Agora, chegamos aos padrões – soluções específicas para recorrentes problemas táticos na programação.

O próximo capítulo, “Motivação”, descreve os fatores econômicos que tornam o foco na comunicação por meio do código uma atividade valiosa.

CAPÍTULO 4

Motivação

Trinta anos atrás, Yourdon & Constantine, em *Structured Design*, classificaram a economia como condutor fundamental do projeto de software. Um programa deveria ser escrito para reduzir seu custo global. O custo de um software é dividido em custo inicial e custo de manutenção:

$$\text{custo}_{\text{total}} = \text{custo}_{\text{desenvolvimento}} + \text{custo}_{\text{manutenção}}$$

Conforme a indústria ganhava experiência no desenvolvimento de software, muitos ficaram surpresos ao descobrir que o custo de manutenção era muito maior que o de desenvolvimento. (Projetos com pouca ou nenhuma manutenção devem usar um conjunto de padrões de implementação muito diferente daquele apresentado no restante deste livro.)

A manutenção é algo caro porque entender o código existente consome tempo e é uma atividade passível de erros. Fazer mudanças, em geral, é fácil quando se sabe o que é preciso mudar. A parte custosa é aprender o que o código existente faz. Depois que as mudanças são feitas, elas precisam ser testadas e implantadas.

$$\text{custo}_{\text{manutenção}} = \text{custo}_{\text{entender}} + \text{custo}_{\text{mudar}} + \text{custo}_{\text{testar}} + \text{custo}_{\text{implantar}}$$

Uma estratégia para reduzir o custo global é investir mais no desenvolvimento inicial, visando diminuir ou eliminar a necessidade de manutenção. Esforços nesse sentido, em geral, têm falhado em reduzir os custos globais. Quando o código precisa de alterações inesperadas, nenhuma prevenção consegue deixá-lo perfeitamente preparado para isso. Tentativas prematuras de preparar o código geral para atender a necessidades futuras frequentemente interferem nas mudanças inesperadas que se revelam necessárias.

Aumentar substancialmente o investimento prévio no software vai de encontro a dois importantes princípios econômicos: o valor temporal do dinheiro e a incerteza do futuro. Como um dólar hoje vale mais que um dólar amanhã, em princípio, uma estratégia de implementação deveria encorajar o adiamento dos custos. Ainda, em razão da incerteza, uma estratégia de implementação deveria preferir benefícios imediatos a possíveis benefícios a longo prazo. Embora

isso possa soar como uma licença para desconsiderar o futuro, os padrões de implementação visam a benefícios imediatos ao estabelecer um código limpo que facilite o desenvolvimento futuro.

Minha estratégia para reduzir custos globais é pedir que todos os programadores tenham em mente o custo de se entender um código durante a fase de manutenção, focando, durante o desenvolvimento, na comunicação entre programadores. Os benefícios imediatos de um código limpo são uma menor quantidade de defeitos, um compartilhamento de código mais fácil e um desenvolvimento mais suave.

Uma vez que um conjunto de padrões de implementação tenha se tornado familiar, programa-se mais rápido e com menos pensamentos dispersivos. Quando comecei a escrever meu primeiro conjunto de padrões de implementação (*The Smalltalk Best Practice Patterns*, Prentice Hall, 1996), considerava-me um programador proficiente. Como estímulo a um foco nos padrões, não digitava um caractere de código a menos que tivesse colocado no papel o padrão que seria seguido. Aquilo era frustrante, como se estivesse programando com os dedos amarrados. Na primeira semana, cada minuto de codificação era precedido de uma hora de escrita. Na segunda semana, percebi que tinha definido a maioria dos padrões básicos e, em boa parte do tempo, estava seguindo padrões existentes. Na terceira semana, eu codificava muito mais rápido que antes, pois tinha assimilado meu próprio estilo e não perdia tempo com dúvidas.

Os padrões de implementação que escrevi foram apenas parcialmente criados por mim. Muito do meu estilo foi copiado de gerações anteriores de programadores. Eles eram os modelos para um código mais fácil de ler, entender e modificar, permitindo que eu escrevesse códigos de forma mais rápida e fluída que antes. Eu podia, ao mesmo tempo, me preparar para o futuro e escrever um código mais rapidamente.

Para os padrões de implementação indicados neste livro, inspirei-me tanto em códigos existentes quanto em meus próprios hábitos. Li e comparei códigos da JDK, da Eclipse e de minha experiência. Os padrões resultantes pretendem ser uma visão coerente de como escrever um código que as pessoas consigam entender. Outros pontos de vista ou conjuntos de valores resultariam em diferentes padrões. Por exemplo, eu esboço os padrões de implementação para desenvolvimento de um framework no Capítulo 10, com base em um conjunto distinto de prioridades, variando assim meu estilo de implementação básico.

Os padrões de implementação atendem a necessidades humanas assim como a necessidades econômicas. Programadores podem usar os padrões de implementação para satisfazer a necessidades humanas, assim como à necessidade de ter orgulho do trabalho bem feito ou de ser um membro confiável de uma comunidade. Ao longo dos próximos capítulos, discuto o impacto humano e econômico dos padrões.

CAPÍTULO 5

Classe

A ideia de classes veio muito antes de Platão. Os sólidos platônicos eram classes cujos exemplos podiam ser vistos no mundo. A esfera platônica era absolutamente perfeita, mas insubstancial. Podíamos tocar nas esferas a nossa volta, mas elas eram imperfeitas de alguma forma.

A programação orientada a objetos captou essa ideia por meio de filósofos ocidentais posteriores, dividindo programas em classes (descrições gerais de um conjunto de coisas similares) e em objetos (as coisas em si).

Classes são importantes para a comunicação, pois podem descrever muitas coisas específicas. Padrões em nível de classe têm o maior alcance entre os padrões de implementação. Padrões de projeto, ao contrário, costumam ser relativos às relações entre as classes.

Os seguintes padrões aparecem neste capítulo:

- Classe (class) – usa-se uma classe para dizer: “Esses dados vão juntos, e a lógica segue com eles”.
- Nome simples de superclasse (simple superclass name) – nomeiam-se as raízes das hierarquias de classes com nomes simples tirados da mesma metáfora.
- Nome qualificado de subclasse (qualified superclass name) – nomeiam-se subclasses para comunicar as similaridades e as diferenças em relação a uma superclasse.
- Interface abstrata (abstract interface) – separa-se a interface da implementação.
- Interface (interface) – especifica-se uma interface abstrata que não muda o tempo todo em uma interface Java.
- Interface versionada (versioned interface) – aumentam-se as interfaces de forma segura, introduzindo uma nova subinterface.
- Classe abstrata (abstract class) – especifica-se uma interface abstrata que provavelmente mudará em uma classe abstrata.

- Objeto valor (value object) – escreve-se um objeto que age como um valor matemático.
- Especialização (specialization) – expressam-se claramente as similaridades e as diferenças de computações relacionadas.
- Subclasse (subclass) – expressa-se a variação unidimensional por meio de uma subclasse.
- Implementador (implementor) – sobrescreve-se um método para expressar uma variante de computação.
- Classe interna (inner class) – empacota-se um código localmente útil em uma classe privada.
- Comportamento específico de instância (instance-specific behavior) – varia-se a lógica por instância.
- Condicional (conditional) – varia-se a lógica por condicionais explícitos.
- Delegação (delegation) – varia-se a lógica delegando-a a um ou vários tipos de objetos.
- Seletor plugável (pluggable selector) – varia-se a lógica executando reflexivamente um método.
- Classe interna anônima (anonymous inner class) – varia-se a lógica sobrescrevendo um ou dois métodos diretamente no método que cria um novo objeto.
- Classe biblioteca (library class) – representa-se uma coleção de funcionalidades que não se encaixam em nenhum outro objeto como um conjunto de métodos estáticos.

Classe

Os dados mudam mais frequentemente que a lógica – é essa observação que faz as classes funcionarem. Cada classe é uma declaração do tipo: “Esta lógica harmoniza-se e muda mais lentamente que os valores de dados sobre os quais opera. Esses valores de dados também se harmonizam, mudando em ritmos similares e sendo operados pela lógica relacionada”. Essa separação estrita entre dados que mudam e lógica que não muda não é absoluta. Às vezes, a lógica varia consideravelmente. Às vezes, os dados não mudam no decorrer de uma computação. Aprender como empacotar a lógica em classes e expressar variações naquela lógica é parte da eficiência na programação com objetos.

Organizar classes em hierarquias é uma forma de compressão, escolhendo a superclasse e a incluindo textualmente em todas as subclasses. Como ocorre com todas as técnicas de compressão, o código fica mais difícil de ler. É preciso entender o contexto da superclasse para entender a subclasse.

Usar a herança criteriosamente é outro aspecto da eficiência na programação com objetos. Criar uma subclasse é como dizer: “Sou como aquela super-

classe, porém diferente". (Não é estranho falar de *sobrescrever* um método em uma *subclasse*? Quão melhores seríamos como programadores se tivéssemos escolhido cuidadosamente nossas metáforas?)

Classes são elementos de projeto relativamente caros em programas feitos a partir de objetos e, por isso, deveriam ter uma função significativa. Reduzir o número de classes em um sistema é uma melhora, contanto que as classes remanescentes não fiquem inchadas.

Os padrões a seguir explicam como se comunicar declarando classes.

Nome simples de superclasse

Encontrar o nome preciso é um dos momentos mais satisfatórios na programação. Fica-se buscando uma inspiração. Muitas vezes o código fica complicado, mas parece que não precisaria ser. Aí, de repente, em uma conversa, alguém diz: "Olha só! Isso, na verdade, é um Escalonador". Todos voltam a sentar-se e soltam um suspiro. O nome certo resulta em uma sequência de simplificações e melhorias.

É muito importante escolher bem os nomes das classes, pois são o conceito-âncora do projeto. Depois de nomear as classes, os nomes das operações vêm naturalmente. Já o inverso raramente é verdadeiro, exceto quando a classe recebe, no início, um nome fraco.

Ao nomear classes, verifica-se uma disputa entre brevidade e expressividade. Costumam-se usar nomes de classes em uma conversa: "Você se lembrou de girar a Figura antes de transladá-la?". Os nomes devem ser curtos e incisivos; contudo, para torná-los precisos, às vezes são necessárias muitas palavras.

Uma solução para esse dilema é escolher uma metáfora forte para a computação. Com ela em mente, mesmo palavras simples trazem em si uma rica teia de associações, conexões e implicações. Por exemplo, no framework de desenho HotDraw, meu primeiro nome para o objeto de um desenho foi `ObjetoDesenhado`. Ward Cunningham veio com uma metáfora de tipografia: um desenho é como uma página impressa e diagramada. Como em uma página os itens gráficos são figuras, logo a classe se tornou `Figura`. No contexto da metáfora, `Figura` é mais curto, mais rico e mais preciso que `ObjetoDesenhado`.

Encontrar bons nomes pode levar tempo. O código pode estar "pronto" e funcionando por semanas, meses ou (em um caso notável para mim) anos quando se descobre um nome melhor para determinada classe. Às vezes é preciso esforçar-se um pouco mais para encontrar um nome: vasculhar um tesouro, fazer uma lista com os nomes menos adequados que se consegue imaginar, dar um passeio. Outras vezes é preciso seguir em frente com uma nova funcionalidade, confiando no tempo, na frustração e no seu subconsciente para produzir um nome melhor.

A conversa é uma ferramenta que costuma me ajudar a encontrar nomes melhores. Explicar o propósito de um objeto a outra pessoa me leva a procurar imagens ricas e evocativas para descrevê-lo. Essas imagens podem levar, por sua vez, a novos nomes.

Procure nomes com uma só palavra para classes importantes.

Nome qualificado de subclasse

Os nomes das subclasses têm duas funções: informar com qual classe se *parecem* e de qual se *diferem*. Mais uma vez, deve-se atingir um equilíbrio entre tamanho e expressividade. Diferentemente dos nomes na raiz das hierarquias, os nomes das subclasses não são usados com tanta frequência em conversas, de forma que podem ser expressivos, e não tão concisos. Inclua, no início do nome da superclasse, um ou mais modificadores para formar o nome da subclasse.

Uma exceção a essa regra é quando a subclasse é usada estritamente como um mecanismo de compartilhamento de implementação, sendo um conceito importante em si mesma. Dê às subclasses que servem como raízes de hierarquias um nome simples. Por exemplo, no HotDraw, a classe Handler [manipulador] apresenta operações de edição quando uma figura é selecionada – é chamada simplesmente de Handler, a despeito da extensão Figura. Há uma família inteira de manipuladores com nomes mais apropriados, como StretchyHandle [esticador] e TransparencyHandle [transparência]. Como é a raiz de sua hierarquia, Handler merece mais um nome simples de superclasse do que um nome qualificado de subclasse.

Outra dificuldade para se nomear subclasses são hierarquias de múltiplos níveis. Estas, em geral, são delegações que podem ser necessárias, mas, enquanto estão em ação, precisam de bons nomes. Em vez de simplesmente colocar os modificadores no início do nome da superclasse imediata, pense no nome sob a perspectiva do leitor. Ele precisa saber que esta classe se parece com qual outra? Use esta superclasse como base para o nome da subclasse.

Comunicação com as pessoas é o propósito dos nomes de classes. Para o computador, as classes poderiam simplesmente ser numeradas. Nomes de classes longos demais são difíceis de ler e formatar. Nomes de classes muito curtos sobrecarregam a memória de curto prazo do leitor. Agrupamentos de classes cujos nomes não têm relação serão difíceis de compreender e lembrar. Use nomes de classes que contêm a história de seu código.

Interface abstrata

A velha regra em desenvolvimento de software é codificar para interfaces, e não para implementações. Essa é outra forma de sugerir que uma decisão de projeto só deve ser visível nos pontos necessários. Se a maior parte de um código apenas identifica que se está lidando com uma coleção, há liberdade para se mudar a classe concreta posteriormente. Entretanto, em algum ponto, será preciso enviar uma classe concreta para que o computador consiga realizar um cálculo.

Ao usar o termo “interface”, refiro-me a “um conjunto de operações sem implementações”. Isso pode ser representado em Java como uma interface ou como uma superclasse. Os padrões a seguir sugerem quando cada um é apropriado.

Cada camada de interface tem custos. É uma coisa a mais para se aprender, entender, documentar, depurar, organizar, navegar e nomear. Maximizar o número de interfaces não minimiza o custo do software. Gaste com interfaces apenas onde será necessária a flexibilidade que elas criam. Como muitas vezes

não se sabe antecipadamente onde será preciso a flexibilidade de uma interface, combine a especulação de onde introduzir interfaces com a inclusão delas quando é exigida flexibilidade para se minimizar o custo.

Ainda que reclamemos da inflexibilidade de um software, há inúmeros casos em que não se precisa de sistema algum para flexibilizá-lo. De mudanças fundamentais, como o número de bits em um inteiro, a mudanças em larga escala, como novos modelos de negócio, a maior parte do software não precisa ser flexível na maioria dos casos.

Outro fator econômico implicado na introdução de interfaces é a imprevisibilidade do software. Nossa indústria parece viciada na ideia de que basta projetarmos softwares da maneira correta para não termos de mudar os sistemas. Recentemente, li uma lista de razões pelas quais um software muda, que incluía programadores fazendo um trabalho ruim na definição de requisitos, mudanças de ideia pelos responsáveis, etc. O único fator que faltava nessa lista era uma mudança legítima, pois se assume que uma mudança é sempre um erro. Por que nem sempre a previsão do tempo acerta? Porque o tempo muda de formas imprevisíveis. Por que não podemos listar de uma vez por todas os casos em que um sistema precisa ser flexível? Porque os requisitos e a tecnologia mudam de formas imprevisíveis. Isso não tira a responsabilidade de se fazer o melhor para desenvolver o sistema de que os clientes precisam naquele exato momento, mas sugere que há limites para a importância de o software ser “à prova de futuro”.

Unindo-se todos esses fatores – a necessidade de flexibilidade, o custo da flexibilidade, a imprevisibilidade de onde é preciso haver flexibilidade –, chego à conclusão de que a hora de introduzir flexibilidade é quando ela realmente é necessária. Isso tem um custo, devido às mudanças necessárias no software. Se não for possível que o próprio programador faça todas as mudanças necessárias, o custo é ainda maior – um tópico discutido em detalhes no Capítulo 10.

Os dois mecanismos de Java para interfaces abstratas, superclasses e interfaces têm perfis de custo diferentes para tais mudanças.

Interface

Um jeito de dizer “aqui está o que quero fazer e, a partir daqui, estão os detalhes com os quais não devo me preocupar” é declarar uma interface Java. Interfaces são uma das importantes inovações disponibilizadas em uma linguagem de mercado de massa em Java. Elas têm um bom equilíbrio e apresentam parte da flexibilidade da herança múltipla, sem complexidade e ambiguidade. Uma classe pode declarar-se participante de múltiplas interfaces. As interfaces revelam apenas operações, não campos, de forma que podem proteger os usuários de uma interface de potenciais mudanças em sua implementação.

Embora interfaces permitam mudanças em sua implementação, não encorajam mudanças na própria interface. Qualquer adição ou mudança em uma interface exige que se modifiquem todos os implementadores. Se não se conse-

gue modificar a implementação, o uso difundido de interfaces tem influência significativa na posterior evolução do projeto.

Um equívoco sobre interfaces que limita seu valor como forma de comunicação é que todas as operações precisam ser públicas. Muitas vezes desejei que houvesse operações visíveis somente no pacote de interfaces. Tornar públicos demais os elementos de um projeto não é um problema quando eles têm uso privado, mas, quando se disponibilizam interfaces para um grande público, seria melhor poder ser preciso em vez de acumular inércia contra mudanças futuras.

Dois estilos de nomear interfaces dependem de como se está pensando as interfaces. Interfaces como classes sem implementações deveriam ser nomeadas como se fossem classes (nome simples de superclasse, nome qualificado de subclasse). Um problema dessa abordagem é que os bons nomes são usados antes de se chegar à nomeação das classes. Uma interface chamada de Arquivo precisa de uma classe de implementação com um nome como ArquivoAtual, ArquivoReal ou (eca!) ArquivoImpl (um prefixo e uma abreviação). Em geral, é importante informar que se está lidando com um objeto concreto ou abstrato; e é menos importante informar se um objeto abstrato é implementado como uma interface ou uma superclasse. Adiar a distinção entre interfaces e superclasses funciona bem nesse estilo de nomeação, deixando o programador livre para mudar de ideia se for necessário.

Às vezes, nomear classes concretas com simplicidade é mais importante para a comunicação que esconder o uso de interfaces. Neste caso, usa-se o prefixo “I” nos nomes das interfaces. Se a interface é chamada IArquivo, a classe pode ser simplesmente chamada de Arquivo.

Classe abstrata

Outra forma de expressar, em Java, uma distinção entre interface abstrata e implementação concreta é usar uma superclasse. A superclasse é abstrata porque pode ser substituída por qualquer subclasse enquanto é executada, seja ela abstract para Java ou não.

O dilema de quando usar uma classe abstrata ou uma interface se resume a duas questões: mudanças na interface e a necessidade de uma única classe suportar simultaneamente múltiplas interfaces. Interfaces abstratas precisam suportar dois tipos de mudanças: mudança na implementação e mudanças da própria interface (nestas últimas, as interfaces Java não funcionam bem). Cada mudança em uma interface requer mudanças em todas as implementações. Com uma interface amplamente implementada, isso pode facilmente paralisar projetos existentes, sendo possível evoluir posteriormente apenas por meio de interfaces versionadas.

Classes abstratas não têm essa limitação. Desde que se possa especificar uma implementação padrão, novas operações podem ser adicionadas a uma classe abstrata sem atrapalhar os implementadores existentes.

Uma limitação das classes abstratas é que os implementadores podem apenas declarar fidelidade a uma única superclasse. Se forem necessárias outras visões da mesma classe, elas devem ser implementadas por interfaces Java.

Usar a palavra-chave `abstract` em uma classe informa aos leitores que será preciso um trabalho de implementação se quiserem usar a classe. Se houver qualquer chance de tornar útil e instanciável em si a raiz de uma hierarquia de classes, deve-se fazê-lo. Estando no caminho da abstração, é fácil ir longe demais e criar abstrações que nunca terão função. Empenhar-se em tornar instanciáveis as classes-raiz estimula a eliminação de abstrações que não valem a pena.

Interfaces e hierarquias de classe não são mutuamente exclusivas. Pode-se fornecer uma interface que diga “Eis como acessar esse tipo de funcionalidade” e uma superclasse que diga “Eis uma forma de implementar essa funcionalidade”. Neste caso, as variáveis deveriam ser declaradas com a interface sendo seu tipo, de forma que mantenedores futuros consigam substituir novas implementações se necessário.

Interface versionada

O que fazer quando é preciso mudar uma interface, mas não se pode fazê-lo? Normalmente, isso acontece quando se quer adicionar operações. Como adicionar uma operação quebrará todos os implementadores existentes, não é possível fazer isso. Contudo, pode-se declarar uma nova interface que estenda a original e adicionar a operação lá. Usuários que quiserem a nova funcionalidade deverão usar a interface estendida, desde que permaneça óbvia aos usuários a existência da nova interface. Em qualquer lugar que se queira acessar a nova operação, deve-se explicitar o tipo de seu objeto e convertê-lo ao novo tipo.

Por exemplo, considere um comando simples:

```
interface Command {
    void run();
}
```

Como essa interface foi disponibilizada e estendida milhares de vezes, é caro mudá-la. Contudo, para conseguir desfazer comandos, é preciso uma nova operação. A solução com interface versionada é assim:

```
interface ReversibleCommand extends Command {
    void undo();
}
```

As instâncias de `Command` funcionam como antes. As instâncias de `ReversibleCommand` funcionam em qualquer lugar em que funcione um `Command`. Para usar a nova operação:

```
...
Command recent= ...;
if (recent instanceof ReversibleCommand) {
    ReversibleCommand downcasted= (ReversibleCommand) recent;
    downcasted.undo();
}
...
```

Usar `instanceof` em geral reduz a flexibilidade por amarrar o código a certas classes. Neste caso, entretanto, pode ser justificável, pois permite a evolução das interfaces. Contudo, quando se passa a ter muitas interfaces alternativas, os clientes terão muito trabalho para lidar com todas as variações – sinal de que é hora de repensar o projeto.

Interfaces alternativas são uma solução feia para um problema feio, pois as interfaces não acomodam mudanças em sua estrutura tão facilmente quanto em suas implementações. Assim como em todas as decisões de projeto, é possível que se precise mudar as interfaces. Todos aprendemos sobre projetos por meio de implementação e manutenção. Interfaces alternativas criam uma nova linguagem de programação, que é como o Java, mas com novas regras. Escrever novas linguagens é um jogo diferente, com regras mais complicadas do que escrever aplicações. Todavia, quando se está estagnado precisando estender uma interface, é bom saber como fazer isso.

Objeto valor

Embora pensar em objetos com estado mutável seja um artifício valioso em computação, este não é o único método. A matemática desenvolveu-se ao longo de milênios como um jeito de pensar situações que podem ser reduzidas a um mundo abstrato de verdade e certeza absolutas, no qual declarações podem ser feitas sobre verdades eternas.

As linguagens de programação atuais são uma mistura dos dois estilos. Os chamados tipos primitivos do Java pertencem (em sua maioria) ao mundo da matemática. Quando se adiciona 1 a um número em Java, faz-se uma instrução matemática (exceto aquela parte em que alguém decidiu que um computador contaria apenas até 2^{32} ou 2^{64} e então se deveria voltar para o início). O valor de uma variável não é alterado quando se adiciona 1: cria-se um novo valor. Não há forma de mudar o 0 como se faz com a maioria dos objetos.

Esse estilo funcional de computação não muda um estado, apenas cria novos valores. Em uma situação (talvez momentânea) estática para a qual gostaria de dar instruções ou sobre a qual gostaria de fazer perguntas, o estilo funcional é adequado. Em uma situação que se altera com o tempo, o estado é adequado. Algumas situações poderiam ser pensadas em ambos os estilos. Como dizer qual método é mais útil?

Por exemplo, pode-se representar o desenho de uma figura como mudanças no estado de algum meio gráfico, como um bitmap. Alternativamente, pode-se descrever a mesma figura com uma descrição estática (Figura 5.1).

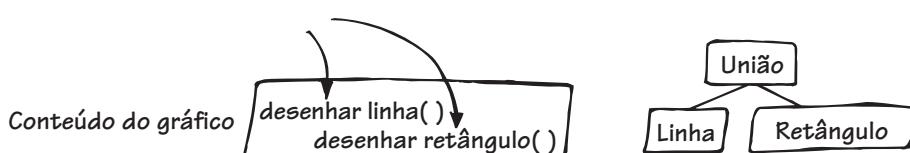


Figura 5.1 Gráficos representados como procedimentos e objetos.

Definir qual dessas representações é mais útil depende, de certa forma, de preferência pessoal, mas também da complexidade das figuras a serem desenhadas e da frequência com que mudam.

As interfaces procedurais são mais comuns que as funcionais. Um problema das procedurais é que a sequência das chamadas de procedimento torna-se parte importante (mas frequentemente implícita) do significado da interface. Mudar um programa desses é perigoso e difícil, pois mudanças que parecem pequenas têm consequências accidentais quando se altera o significado implícito de uma sequência.

A beleza das representações matemáticas é que a sequência raramente importa. Cria-se um mundo em que se pode fazer declarações absolutas e atemporais. Sempre que possível, deve-se criar micromundos da matemática, gerenciando-os a partir de um objeto com estado mutável.

Por exemplo, implemente um sistema contábil transformando as transações básicas em valores matemáticos imutáveis.

```
class Transaction {
    int value;
    Transaction(int value, Account credit, Account debit) {
        this.value= value;
        credit.addCredit(this);
        debit.addDebit(this);
    }
    int getValue() {
        return value;
    }
}
```

Não há como mudar qualquer dos valores de uma `Transaction` depois de ela ter sido criada. Além disso, o construtor declara que todas as transações são enviadas para duas contas. Quando se lê esse código, sabe-se que não há por que se preocupar com transações que flutuam perdidas ou com transações que têm valores alterados depois de terem sido enviadas.

Para implementar objetos com estilo de valor (isto é, objetos que agem como inteiros em vez de serem mutáveis), primeiro deve-se delimitar a fronteira entre o mundo de estados e o de valores. No exemplo acima, uma `Transaction` é

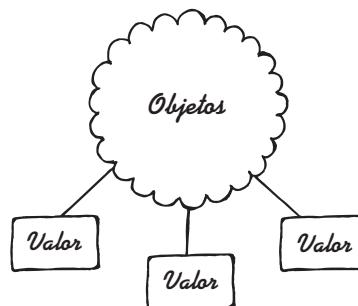


Figura 5.2 Objetos de estado mutável referindo-se a objetos imutáveis nas pontas.

um valor, e uma `Account` está em estado mutável. Defina, no construtor, todos os estados como um objeto com estilo de valor, sem outras atribuições de campos em qualquer outro lugar do objeto. Operações de objetos com estilo de valor sempre retornam novos objetos, os quais devem ser armazenados pelo requisitante da operação.

```
bounds.translateBy(10, 20); // Retângulo mutável  
bounds= bounds.translateBy(10, 20); // Retângulo com estilo
```

O maior argumento contra objetos com estilo de valor sempre foi seu desempenho. A necessidade de criar todos aqueles objetos intermediários pode criar uma sobrecarga no sistema de gerenciamento de memória. Mas, no custo geral, esse argumento não costuma se sustentar, pois a maior parte do programa não age como um gargalo de desempenho. Outras razões para não usar objetos com estilo de valor são a falta de familiaridade com o estilo e a dificuldade em delimitar as partes do sistema em que o estado muda e aquelas em que os objetos não mudam. A existência de uma maioria de objetos com estilo de valor é o pior dos dois mundos, uma vez que as interfaces tendem a ser mais complicadas e não se pode ter certeza sobre estados não mutáveis.

Tendo chegado até aqui, sinto que há muito mais para ser escrito sobre programação com os três principais estilos – objetos, funções e procedimentos – e sobre como mesclá-los de maneira eficaz. Para os propósitos deste livro, reitero que, às vezes, programas são mais bem expressos como uma combinação de objetos de estado mutável e objetos representando valores matemáticos.

Especialização

Comunicar a reciprocidade entre as similaridades e as diferenças de computações torna os programas mais fáceis de ler, usar e modificar. Na prática, nenhum programa é único. Muitos expressam ideias similares, e muitas partes de um programa frequentemente expressam ideias similares. Expressar claramente similaridades e diferenças permite que leitores entendam o código existente e descubram se suas necessidades atuais são supridas por uma das variações existentes. Caso não sejam, pode-se decidir o que é melhor: especializar o código existente de acordo com suas necessidades ou escrever um código inteiramente novo.

As variações mais simples são aquelas em que os estados diferem. A string “abc” é diferente de “def”, mas os algoritmos que operam sobre as duas são idênticos. Por exemplo, o comprimento de todas as strings é calculado do mesmo jeito.

As variações mais complexas são diferenças totais na lógica. Uma rotina de integração simbólica não tem a mesma lógica que uma rotina de composição tipográfica matemática, muito embora as duas possam dividir exatamente a mesma entrada.

Entre esses dois extremos – lógica idêntica com dados diferentes e lógica diferente com dados idênticos – há o enorme terreno comum da programação. A maior parte dos dados pode ser a mesma, mas levemente diferente. A maior

parte da lógica pode ser a mesma, mas levemente diferente. (Meu palpite é que a rotina de integração simbólica e a de composição tipográfica matemática têm pouco em comum no código.) Mesmo o limite entre lógica e dados é obscuro. Uma flag é um dado booleano, mas afeta o fluxo de controle. Um objeto auxiliar pode ser armazenado em um campo e ser usado para afetar uma computação.

Os padrões a seguir são uma variedade de técnicas para comunicar similaridade e diferença, principalmente na lógica. Variações em dados não parecem tão complicadas ou sutis. Expressões eficazes de similaridade e diferença na lógica abrem novas oportunidades para futuras expansões do código.

Subclasse

Declarar uma subclasse é uma forma de dizer “Estes objetos são como aqueles, exceto por...”. Quando se tem a superclasse correta, criar uma subclasse pode ser uma maneira poderosa de programar. Com o método correto de sobrepor, pode-se, com poucas linhas de código, introduzir a variante de um cálculo existente.

Quando os objetos se tornaram populares, criar subclasses parecia uma pílula mágica. Inicialmente, as subclasses eram usadas para classificação – um Trem era uma subclasse de Veículo a despeito de compartilharem ou não alguma implementação. Em tempo, algumas pessoas viram que, como herança era na verdade implementação compartilhada, elas poderiam ser usadas de modo mais eficaz para fatorar pedaços comuns da implementação. Contudo, logo as limitações de criar subclasses tornaram-se evidentes. Primeiramente, é uma carta que se pode utilizar apenas uma vez. Quando se descobre que um conjunto de variações não pode ser bem expresso como subclasses, há muito trabalho a ser feito para desemaranhar o código antes que se possa reestruturá-lo. Em segundo lugar, é preciso entender a superclasse antes que se consiga entender a subclasse. Conforme as superclasses vão ficando mais complicadas, isso se torna uma limitação. Em terceiro lugar, mudanças na superclasse são perigosas, visto que subclasses podem depender de propriedades sutis da implementação da superclasse. Por fim, todos esses problemas são agravados por hierarquias de herança profundas.

Um uso particularmente pernicioso da herança é criar hierarquias paralelas em que, para cada subclasse *nesta* hierarquia, é necessária uma subclasse *naquela* hierarquia. Essa é uma forma de duplicação que cria acoplamento implícito entre as hierarquias de classes. Para introduzir com sucesso uma nova variação, é preciso mudar ambas as hierarquias. Embora frequentemente se vejam hierarquias paralelas que não se sabe imediatamente como eliminar, o esforço nesse sentido aprimora o projeto.

Um exemplo disso foi um sistema de seguros (Figura 5.3). Algo estava definitivamente errado naquele cenário, pois um `ContratoDeSeguro` não pode se referir a um `ContratoDePensão`, nem é interessante descer o campo `produto` para as subclasses. A solução – a qual nunca alcançamos, mas levou um ano para chegarmos perto – foi mover a variação de forma que `Contrato` funcionasse do



Figura 5.3 Hierarquias paralelas.

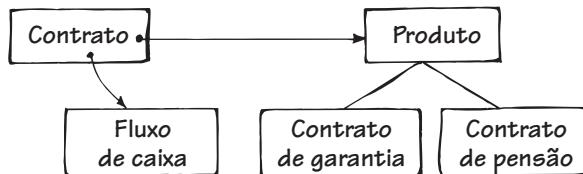


Figura 5.4 Hierarquia com duplicação eliminada.

mesmo jeito se fosse usado para seguro ou pensão. Isso exigiu a criação de um novo objeto para representar os futuros fluxos de caixa (Figura 5.4).

Com todos esses cuidados em mente, criar subclasses pode ser uma ferramenta poderosa para expressar computações do tipo “tema e variações”. A subclasse certa pode ajudar muitas pessoas a expressar exatamente a computação que querem com um ou dois métodos. Uma chave para alcançar subclasses úteis é fatorar completamente a lógica da superclasse em métodos que fazem uma única tarefa. Quando se escreve uma subclasse, deve-se ser capaz de sobrescrever exatamente um único método. Se os métodos da superclasse forem muito grandes, será preciso copiar o código e editá-lo (Figura 5.5).

Um código copiado introduz um terrível acoplamento implícito entre as duas classes. Não se pode mudar o código na superclasse sem examinar e, talvez, mudar todos os lugares para os quais foi copiado.

Meu objetivo em um projeto é poder trocar de estratégia à vontade, dependendo das necessidades do código existente. Deve-se visualizar o código expresso com condicionais, com subclasses, com delegação. Parece haver vantagens em uma estratégia diferente daquela que se está usando no momento? Dê alguns passos nessa direção e veja se melhora o código.

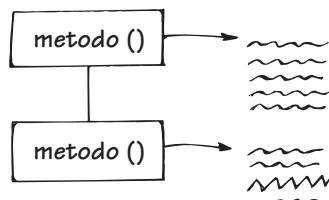


Figura 5.5 Código copiado e modificado em uma subclasse.

Uma última limitação de se criar subclasses é que elas não podem ser usadas para expressar lógica mutável. A variação que se quer deve ser conhecida ao se criar um objeto, o qual não pode ser alterado depois. Será necessário usar condicionais ou delegação para expressar uma lógica que muda.

Implementador

A mensagem polimórfica é a maneira fundamental de se expressar uma escolha em um programa construído com objetos. Para que a mensagem atinja seu objetivo de escolha, precisa haver mais de um tipo de objeto para potencialmente receber a mensagem.

Implementar o mesmo protocolo múltiplas vezes, seja expresso com uma interface Java e uma declaração `implements` ou como uma subclasse expressa em `extends`, é uma forma de dizer: “Do ponto de vista de uma parte do cálculo, desde que aconteça algo que condiga com a intenção do código, os detalhes do que acontece são irrelevantes”.

A beleza das mensagens polimórficas é que elas abrem um sistema para variação. Se parte do programa escreve alguns bytes em outro sistema, a introdução de um Socket abstrato permite que varie a implementação do socket sem afetar o código da chamada. Comparada à expressão procedural da mesma intenção, com sua lógica condicional explícita e fechada, a versão objeto/mensagem é mais clara, separando a expressão da intenção (escrever alguns bytes) da implementação (chamar uma pilha TCP/IP com certos parâmetros). Ao mesmo tempo, expressar a computação em forma de objetos e mensagens abre o sistema para futuras variações nunca sonhadas pelos programadores originais. Essa combinação fortuita de clareza de expressão e flexibilidade é o motivo para as linguagens orientadas a objetos terem se tornado o paradigma da programação dominante.

Esse recurso supremo é facilmente desperdiçado quando se escrevem programas procedurais em Java. Os padrões aqui apresentados têm como objetivo ajudar o leitor a expressar uma lógica clara e extensível.

Classe interna

Algumas vezes é preciso empacotar parte de uma computação, mas não se quer incorrer no custo de toda uma nova classe com seu próprio arquivo. Declarar classes pequenas e privadas (classes internas) é uma maneira barata de obter muitos dos benefícios de uma classe sem ter de arcar com todos os custos.

Às vezes uma classe interna estende apenas o `Object`. Algumas classes internas estendem outra superclasse, o que é útil para expressar refinamentos de outras classes que são interessantes apenas localmente.

Uma das características das classes internas é que, quando suas instâncias são criadas, elas são secretamente uma cópia do objeto que as criou. Isso é conveniente quando se quer acessar os dados dessa instância sem tornar explícito o relacionamento entre as duas classes:

```
public class InnerClassExample {
    private String field;

    public class Inner {
        public String example() {
            return field; // Uses the field from the enclosing instance
        }
    }

    @Test public void passes() {
        field= "abc";
        Inner bar= new Inner();
        assertEquals("abc", bar.example());
    }
}
```

Contudo, na classe interna acima, não há, na verdade, um construtor sem argumentos, mesmo que se declare um. Esse é um problema quando se criam instâncias de classes internas por reflexão.

```
public class InnerClassExample {
    public class Inner {
        public Inner() {
        }
    }

    @Test(expected=NoSuchMethodException.class)
    public void innerHasNoNoArgConstructor() throws Exception {
        Inner.class.getConstructor(new Class[0]);
    }
}
```

Para obter uma classe interna completamente desconectada das instâncias que a envolvem, declare-a static.

Comportamento específico de instância

Em teoria, todas as instâncias de uma classe dividem a mesma lógica. Atenuar essa restrição possibilita novos estilos de expressão. Todos esses estilos, todavia, acarretam em custo. Quando a lógica de um objeto é completamente determinada por sua classe, pode-se ler o código da classe para ver o que vai acontecer. Quando se tem instâncias com comportamentos diferentes, é preciso olhar exemplos vivos ou analisar o fluxo de dados para entender como um objeto vai se comportar.

Outro acréscimo no custo de comportamento específico de instância é quando a lógica muda conforme a computação avança. Para facilitar a leitura do código, tente atribuir comportamento específico de instância quando um objeto é criado, e não o altere depois disso.

Condicional

Expressões if/then e switch são a forma mais simples de comportamento específico de instância. Ao usar condicionais, diferentes objetos executarão diferentes

lógicas de acordo com seus dados. Condicionais como forma de expressão têm a vantagem de a lógica ainda estar toda em uma única classe. Leitores não precisam navegar para encontrar os possíveis caminhos para uma computação. Entretanto, condicionais têm a desvantagem de não poderem ser alterados, exceto pela modificação do código do objeto em questão.

Cada caminho de execução por meio de um programa tem alguma probabilidade de estar correto. Assumindo que as probabilidades de precisão dos caminhos são independentes, quanto mais caminhos houver em um programa, menores serão as chances de o programa estar correto. As probabilidades não são inteiramente independentes, mas o suficiente para que programas com mais caminhos sejam mais suscetíveis a defeitos que aqueles com menos caminhos. A proliferação de condicionais reduz a confiabilidade.

Esse problema é agravado quando condicionais são duplicados. Considere um simples editor gráfico. As figuras precisarão de um método `display()`:

```
public void display() {  
    switch (getType()) {  
        case RECTANGLE :  
            //...  
            break;  
        case OVAL :  
            //...  
            break;  
        case TEXT :  
            //...  
            break;  
        default :  
            break;  
    }  
}
```

As figuras precisarão também de um método para determinar se um ponto está contido dentro delas:

```
public boolean contains(Point p) {  
    switch (getType()) {  
        case RECTANGLE :  
            //...  
            break;  
        case OVAL :  
            //...  
            break;  
        case TEXT :  
            //...  
            break;  
        default :  
            break;  
    }  
}
```

Suponha agora que se queira adicionar um novo tipo de figura. Inicialmente, deve-se adicionar uma cláusula para cada declaração de `switch`. Além disso, para fazer essa mudança, é preciso modificar a classe `Figura` colocando

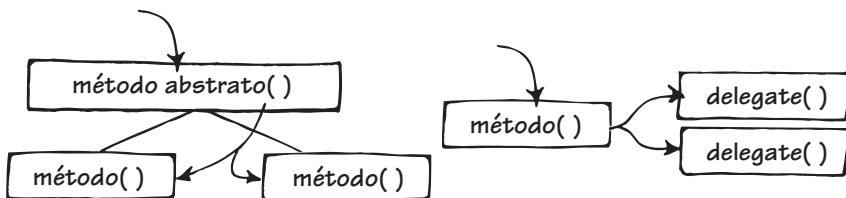


Figura 5.6 Lógica condicional representada por subclasses e delegação.

toda a funcionalidade existente em risco. Por último, todos que quiserem adicionar novas figuras devem coordenar essas mudanças em uma classe só.

Esses problemas podem ser todos eliminados convertendo-se a lógica condicional em mensagens, com subclasses ou delegação (a melhor técnica para isso depende do código). A lógica condicional duplicada, ou lógica em que o processamento é muito diferente dependendo do ramo de condicional seguido, é geralmente mais bem expressa como mensagens do que como lógica explícita. Além disso, uma lógica condicional que muda frequentemente é mais bem expressa como mensagens para simplificar mudanças em um ramo enquanto se minimizam os efeitos em outros ramos.

Em resumo, os pontos fortes dos condicionais – serem simples e locais – tornam-se obrigações quando são muito usados.

Delegação

Outra forma de executar lógica diferente em diferentes instâncias é delegar trabalho para um de diversos tipos possíveis de objetos. A lógica comum é mantida na classe de referência, e as variações, nos delegates.

Um exemplo do uso de um delegate para capturar variação é tratar a entrada de dados do usuário em um editor gráfico. Às vezes, pressionar um botão significa “criar um retângulo”; às vezes, significa “mover uma figura”; e assim por diante.

Uma forma de expressar a variação entre essas ferramentas do editor é com lógica condicional:

```

public void mouseDown() {
    switch (getTool()) {
        case SELECTING :
            //...
            break;
        case CREATING_RECTANGLE :
            //...
            break;
        case EDITING_TEXT :
            //...
            break;
        default :
            break;
    }
}
  
```

Isso tem todos os problemas dos condicionais discutidos há pouco: adicionar uma nova ferramenta requer modificar o código, e a duplicação do condicional (em `mouseUp()`, `mouseMove()`, etc.) dificulta a adição de novas ferramentas.

Fazer subclasses também não é uma solução imediata, pois o editor precisar mudar ferramentas durante seu tempo de vida. A delegação permite essa flexibilidade.

```
public void mouseDown() {
    getTool().mouseDown();
}
```

O código que costumava existir nas cláusulas da declaração switch é movido para as diversas ferramentas. Agora, novas ferramentas podem ser introduzidas sem se modificar o código do editor ou as ferramentas existentes. Ler o código, contudo, requer mais navegação, pois a lógica do mouse-down está espalhada em muitas classes. Entender qual será o comportamento do editor em uma dada situação requer que se entenda o tipo de ferramenta usado.

Delegates podem ser armazenados em campos (um “objeto plugável”), mas podem também ser computados em tempo de execução. O JUnit 4 computa dinamicamente o objeto que executará testes em uma dada classe. Se uma classe contém testes no estilo antigo, é criado um delegate; se a classe contém testes no estilo novo, é criado um delegate diferente. Trata-se de uma mistura de lógica condicional (para criar os delegates) e delegação.

A delegação pode ser usada para compartilhamento de código da mesma forma que é usada para comportamento específico de instância. Um objeto que delega a um Stream pode ser envolvido em comportamento específico de instância se o tipo de Stream puder ser mudado em tempo de execução ou se puder compartilhar a implementação de Stream com todos os outros usuários.

Um truque comum na delegação é passar o delegador como parâmetro para um método delegado.

```
GraphicEditor
public void mouseDown() {
    tool.mouseDown(this);
}

RectangleTool
public void mouseDown(GraphicEditor editor) {
    editor.add(new RectangleFigure());
}
```

Se um delegate precisa enviar uma mensagem para si mesmo, “si mesmo” é algo ambíguo. Às vezes, a mensagem deve ser enviada para o objeto delegador; às vezes, para o delegate. No exemplo dado, RectangleTool adiciona uma figura, mas o faz no delegador GraphicEditor, e não em si mesmo. GraphicEditor poderia ter sido passado como parâmetro para o método delegado `mouseDown()`, mas nesse caso parece mais simples armazenar uma referência de retorno permanente na ferramenta. Passar GraphicEditor como parâmetro permite usar a mesma ferramenta em múltiplos editores, mas, se isso não é importante, o código com o ponteiro de retorno pode ser mais simples.

```

GraphicEditor
public void mouseDown() {
    tool.mouseDown();
}

RectangleTool
private GraphicEditor editor;
public RectangleTool(GraphicEditor editor) {
    this.editor= editor;
}
public void mouseDown() {
    editor.add(new RectangleFigure());
}

```

Seletor plugável

Vamos supor que seja necessário um comportamento específico de instância, mas apenas para um ou dois métodos, e o programador não se importa em ter todas as variações do código em uma classe. Neste caso, armazena-se em um campo o nome do método para ser invocado, sendo depois chamado por reflexão.

Originalmente, cada teste em JUnit precisava ser armazenado em sua própria classe (Figura 5.7), pois cada subclasse tinha apenas um método. As classes pareciam conceitualmente pesadas como um meio de representar uma única classe.

Implementando-se um `runTest()` genérico, o `TesteDeLista` com nomes diferentes executará métodos de teste diferentes. Presume-se que o nome do teste também é o nome de um método recuperado e executado quando o teste é rodado. Eis a versão simples do código para implementar a versão com seletor plugável de execução de um teste.

```

String name;
public void runTest() throws Exception {
    Class[] noArguments= new Class[0];
    Method method= getClass().getMethod(name, noArguments);
    method.invoke(this, new Object[0]);
}

```

A hierarquia de classes simplificada usa uma única classe (Figura 5.8). Da mesma forma que ocorre com todas as técnicas de compressão de código, o código modificado somente é fácil de ler caso se entenda o “truque”.

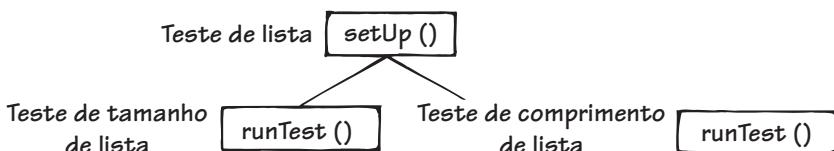


Figura 5.7 Subclasses triviais para representar testes diferentes.

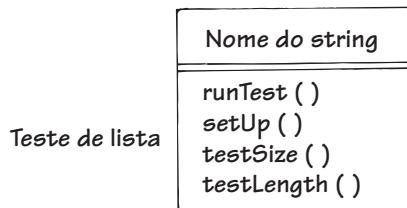


Figura 5.8 Um seletor plugável ajuda a empacotar testes em uma só classe.

Quando os seletores plugáveis se tornaram conhecidos, as pessoas tinham a tendência de usá-los excessivamente. Olhava-se para um código, decidia-se que ele não poderia ser chamado e então ele era deletado. Isso fazia o sistema quebrar, pois, em algum ponto, era invocado por um seletor plugável. Os custos de se usar seletores plugáveis são consideráveis, mas um uso limitado para resolver um problema difícil pode justificar o investimento.

Classe interna anônima

Java oferece mais uma alternativa para comportamento específico de instância: classes internas anônimas. A ideia é criar uma classe que seja usada apenas em um lugar, que pode sobreescriver um ou mais métodos para propósitos estritamente locais. Como isso é usado em apenas um lugar, a classe pode ser referida implicitamente em vez de por meio de um nome.

O uso eficaz de classes internas anônimas baseia-se em ter uma API extremamente simples – como implementar `Runnable` com seu único método `run()` – ou em ter uma superclasse que forneça a maior parte da implementação necessária para que a classe interna anônima seja implementada de forma simples. Como o código para a classe interna anônima interrompe a apresentação do código em que ela está embutida, ele deve ser curto para não distrair o leitor.

Uma limitação das classes internas anônimas é que o código a ser colocado na instância precisa ser conhecido quando se escreve a classe (diferentemente de delegates, que podem ser adicionados depois) e elas não podem ser mudadas depois que uma instância tiver sido criada. Classes internas anônimas são difíceis de testar diretamente e, por isso, não devem conter lógica complicada. Como não são nomeadas, o programador não tem a oportunidade de expressar sua intenção para uma classe interna anônima com um nome bem escolhido.

Classe biblioteca

Onde se coloca uma funcionalidade que não se encaixa em objeto algum? Uma solução é criar métodos estáticos em uma classe que ficaria vazia. Ninguém espera jamais criar instâncias nessa classe. Ela está lá apenas como um contêiner para funções na biblioteca.

Embora sejam bastante comuns, classes biblioteca não escalam* bem. Colocar toda a lógica em métodos estáticos acaba com a maior vantagem da programação com objetos: um namespace privado de dados compartilhados que pode ser usado para ajudar a simplificar a lógica. Sempre que possível, tente transformar classes biblioteca em objetos.

Às vezes isso é simples como encontrar um lar melhor para um método. A classe biblioteca Collections, por exemplo, tem um método `sort(List)`. Esse parâmetro indica que esse método provavelmente pertence, em vez disso, a `List`.

Uma forma incremental de converter uma classe biblioteca em um objeto é converter os métodos estáticos em métodos de instância. No início, mantenha a mesma interface delegando o método estático a um método de instância. Em uma classe chamada `Library`, por exemplo,

```
public static void method(...params...) {  
    ...some logic...  
}
```

torna-se:

```
public static void method(...params...) {  
    new Library().instanceMethod(...params...);  
}  
private void instanceMethod(...params...) {  
    ...some logic...  
}
```

Agora, se muitos dos métodos têm listas de parâmetros similares (e se não têm, os métodos provavelmente pertencem a classes diferentes), converta os parâmetros de método para parâmetros de construtor:

```
public static void method(...params...) {  
    new Library(...params...).instanceMethod();  
}  
private void instanceMethod() {  
    ...some logic...  
}
```

A seguir, mude a interface movendo a criação de instância para os clientes e eliminando os métodos estáticos.

```
public void instanceMethod(...params...) {  
    ...some logic...  
}
```

Essa experiência pode inspirá-lo a renomear a classe e os métodos de forma que se leia claramente o código do cliente.

* N. de R.T.: “Escalar” é um termo traduzido literalmente do inglês *to scale* e usado frequentemente como um comportamento relativo a escalabilidade, ou seja, como se comporta quando o tamanho ou volume aumenta. Diz-se que “escala bem” quando o comportamento mantém sua proporcionalidade ao tamanho ou volume, sem degradar muito; diz-se que “escala mal” em caso contrário.

Conclusão

Um pacote de classes agrupa estados relacionados. O próximo capítulo apresenta padrões que comunicam decisões sobre o estado.

CAPÍTULO 6

Estado

Os padrões neste capítulo descrevem como você pode comunicar seu modo de usar estados. Objetos são pacotes convenientes do *comportamento* que é apresentado para o mundo exterior e do *estado* usado para dar suporte a esse comportamento. Uma das vantagens dos objetos é que eles dividem todo o estado de um programa em pequenos pedaços, cada qual sendo seu próprio computadorzinho. Grandes bibliotecas de estado, indiscriminadamente referenciadas, tornam difíceis as mudanças de código futuras, pois é difícil prever o efeito que uma mudança de código causa no estado. Com objetos é mais fácil analisar qual estado será afetado por uma mudança, porque o namespace do estado referenciável é muito menor.

Este capítulo contém os seguintes padrões:

- Estado (state) – computa-se com valores que mudam com o tempo.
- Acesso (access) – mantém-se a flexibilidade limitando acesso a estados.
- Acesso direto (direct access) – acessa-se diretamente o estado dentro de um objeto.
- Acesso indireto (indirect access) – acessa-se o estado por meio de um método para garantir maior flexibilidade.
- Estado comum (commom state) – armazena-se como campo o estado comum de todos os objetos de uma classe.
- Estado variável (variable state) – armazena-se o estado cuja presença difere de instância a instância, como um mapa.
- Estado extrínseco (extrinsic state) – armazena-se o estado com um propósito especial associado a um objeto em um mapa mantido pelo usuário daquele estado.
- Variável (variable) – fornece um namespace para acessar o estado.
- Variável local (local variable) – mantém o estado para um escopo individual.

- Campo (field) – armazena o estado durante a vida de um objeto.
- Parâmetro (parameter) – comunica o estado durante a ativação de um único método.
- Parâmetro coletor (collecting parameter) – passa-se um parâmetro para coletar resultados complicados de múltiplos métodos.
- Objeto parâmetro (parameter object)– consolidam-se longas listas de parâmetros usadas frequentemente em um objeto.
- Constante (constant) – armazenar como uma constante o estado que não varia.
- Nome sugestivo de função (role-suggesting name) – nomeiam-se variáveis conforme a função que assumem em uma computação.
- Tipo declarado (declared type) – declara-se um tipo geral para variáveis.
- Inicialização (initialization) – sempre que possível, inicializam-se declarativamente variáveis.
- Inicialização ansiosa (eager initialization) – inicializam-se campos no momento da criação da instância.
- Inicialização preguiçosa (lazy initialization) – pouco antes de serem usados, inicializam-se campos cujo cálculo de valores é caro.

Estado

O mundo perdura. Se um minuto atrás o Sol estava no alto do céu, pode-se ter certeza de que ele ainda está no céu, embora tenha se deslocado um pouco. Se eu me interessasse por cálculos, poderia prever sua nova posição com base na observação anterior, no conhecimento que tenho sobre a rotação da Terra e na medição da passagem do tempo.

Pensar no mundo em termos de coisas em mudança provou-se útil por um bom tempo. Os indígenas americanos da região onde moro observavam o monte McLaughlin na primavera. Se a neve derretia o suficiente para que uma silhueta de uma águia aparecesse na neve restante, era hora de descer ao rio Rogue para a pesca primaveril de salmão. O estado da neve na montanha era uma dica valiosa da presença de uma refeição saborosa nadando em águas distantes.

Quando os pioneiros da computação escolheram metáforas para a programação de computadores, eles se fixaram nessa ideia de estado mudando com o tempo. O cérebro humano tem uma ampla variedade de estratégias, inatas e aprendidas, para lidar com o estado.

Contudo, um estado também impõe problemas para programadores. Assim que se presume o que é um bit de estado, o código fica em risco. A presunção pode ser incorreta ou o estado pode mudar. Muitas ferramentas desejáveis de programação, como refatoradores automatizados, são mais fáceis de construir se não houver noção de estado. Por fim, concorrência e estado não atuam

bem juntos. Muitos dos problemas de programas paralelos desaparecem se não houver estado.

Linguagens de programação funcionais dispensam completamente os estados mutáveis. Nunca se tornaram populares. Acho que o estado é uma metáfora valiosa para nós, tendo em vista que nossos cérebros são estruturados e condicionados a lidar com estados mutáveis. A atribuição simples ou a programação sem variáveis forçam-nos a descartar muitas estratégias de pensamento, não sendo opções atraentes.

Linguagens orientadas a objetos são uma boa estratégia para lidar com estados, pois oferecem a oportunidade de evitar o problema de o estado mudar “pelas suas costas” ao particioná-lo no sistema em pedacinhos discretos, cada qual tendo acesso limitado aos outros. É mais fácil manter o rastro de um punhado de bytes do que de mega ou gigabytes. O problema de presumir incorretamente o valor de um estado permanece, mas, com objetos, tem-se a chance de rápida e precisamente revisar todo o acesso a uma variável.

Uma chave para gerenciar efetivamente os estados é colocar estados similares juntos e garantir que estados diferentes fiquem separados. Duas indicações de que dois bits de estado são similares são o fato de os dois bits serem usados na mesma computação e o de os dois bits viverem e morrerem ao mesmo tempo. Se dois pedaços de estado são usados juntos e têm o mesmo tempo de vida, provavelmente é uma boa ideia tê-los próximos.

Acesso

Uma dicotomia em linguagens de programação é a distinção entre acessar valores armazenados e invocar computações. Acessar memória é como invocar uma função que retorna os valores armazenados. Invocar uma função é como ler um local de memória cujo conteúdo tem de ser computado, e não simplesmente retornado. Apesar disso, as linguagens de programação separam invocar computação e acessar memória, de forma que precisamos comunicar eficazmente a diferença.

Decidir o que armazenar e o que computar afeta a legibilidade, a flexibilidade e o desempenho de programas. Em alguns casos, essas metas entram em conflito entre si e com sua preferência de programação. Em outros casos, o contexto muda de forma que a divisão entre armazenado e computado, que antes era razoável, deixa de fazer sentido. Tomar decisões que funcionam e manter a flexibilidade para mudar de ideia no futuro é a chave do bom desenvolvimento de software. É essa necessidade por mudanças futuras que torna importante comunicar claramente suas decisões entre armazenar e computar.

Um dos objetivos dos objetos era gerenciar o armazenamento. Cada objeto age como um pequeno computador que tem sua própria memória, isolado de certa forma dos demais. Linguagens atuais, incluindo Java, escurecem os limites entre os objetos, criando campos públicos. A facilidade de acesso interobjetos não vale a perda de independência entre os objetos.

Acesso direto

A forma mais simples de dizer “estou buscando dados” ou “estou armazenando dados” é usar o acesso direto à variável:

```
x= 10;
```

O acesso direto à variável tem a vantagem de produzir expressões claras. Quando se lê `x = 10;`, sabe-se exatamente o que vai acontecer. Essa clareza tem como consequência a perda de flexibilidade. Se o programador armazena um valor em uma variável, isso é tudo o que ele pode fazer. Se ele armazena naquela variável valores vindos de muitos lugares do programa, então, para fazer uma mudança, provavelmente terá de mudar todos esses lugares.

Outra desvantagem do acesso direto é o fato de ser um detalhe de implementação, abaixo do nível da maioria dos pensamentos de um programador. Estabelecer uma variável com valor 1 pode fazer que a porta da minha garagem abra, mas o código que reflete esse detalhe de implementação não comunica bem. Compare:

```
doorRegister= 1;
```

com:

```
openDoor();
```

ou, com objetos:

```
door.open();
```

A maioria dos pensamentos que tenho durante a programação não tem relação alguma com armazenamento. Aplicar amplamente o acesso direto confunde a comunicação. Para aquelas partes do programa em que realmente se pensa onde e o que armazenar, usa-se o acesso direto para comunicar esse pensamento. Decisões de armazenagem têm um papel diferente para cada programador, de forma que não há regras comuns para o uso do acesso direto. As pessoas ficam tentando formular regras: acesso direto apenas dentro de métodos acessadores e talvez dentro de construtores; acesso direto apenas dentro de uma só classe, ou dentro de uma classe e de todas as suas subclasses, ou talvez dentro de um pacote inteiro. Não há regra universal. Os programadores precisam pensar, comunicar e aprender. É isso que faz um profissional.

Acesso indireto

É possível ocultar acessos e mudanças de estado com invocações. Esses métodos acessadores propiciam flexibilidade, mas diminuem a clareza e a objetividade. Clientes não mais presumem que determinado valor é armazenado diretamente. Assim, pode-se mudar de ideia sobre decisões de armazenagem sem afetar o código do cliente.

Minha estratégia padrão para acessar um estado é permitir acesso direto dentro de uma classe (incluindo classes internas) e acesso indireto para clientes. Essa estratégia tem a vantagem de permitir acesso direto e claro ao estado na maioria dos casos. Contudo, se muitos acessos a um estado do objeto estão fora do objeto, ocorre um problema de projeto mais profundo.

Outra estratégia é usar exclusivamente acesso indireto, mas isso resulta em perda de clareza. A maioria dos métodos get e set é trivial e frequentemente supera em número os métodos úteis, tornando o código difícil de ler. Todos aqueles métodos get e set são também muito tentadores. Em vez de descobrir onde fica um cálculo, é melhor implementá-lo em qualquer lugar e usar métodos accessores para entregar o estado necessário para que funcione.

Um caso em que o acesso indireto é necessário é quando dois pedaços de dados estão acoplados. Às vezes esse acoplamento é muito direto, como se fosse um valor em cache:

```
Rectangle void setWidth(int width) {
    this.width= width;
    area= width * height;
}
```

Outras vezes o acoplamento é menos direto, por meio de um listener:

```
Widget void setBorder(int width) {
    this.width= width;
    notifyListeners();
}
```

Tal acoplamento não é atraente (é fácil esquecer-se de manter as restrições implicadas), mas pode ser a melhor opção disponível. Neste caso, acesso indireto é melhor.

Estado comum

Muitos cálculos dividem os mesmos elementos de dados, ainda que os valores sejam diferentes. Quando se encontra esse tipo de cálculo, deve-se comunicá-lo declarando campos em uma classe. Por exemplo, todos os cálculos com pontos cartesianos requerem uma abscissa e uma ordenada. Como todos os pontos cartesianos compartilham a necessidade por esses valores, eles são mais claramente expressos como campos:

```
class Point {
    int x;
    int y;
}
```

Contraste essa técnica com o estado variável, em que objetos da mesma classe podem ter elementos de dados diferentes. A vantagem do estado comum é que fica evidente no código, seja nos próprios campos ou no construtor completo, qual dado é necessário para se ter um objeto bem formado. O leitor vai

querer saber o que é preciso para invocar a funcionalidade de seu objeto. E o estado comum comunica isso de forma clara e precisa.

Todo estado comum em um objeto deveria ter os mesmos escopo e tempo de vida. De vez em quando, fico tentado a introduzir um campo que é usado apenas por um subconjunto dos métodos em um objeto, ou que somente é válido enquanto um método está sendo computado. Nesses casos, posso inviavelmente melhorar meu código encontrando outro lugar para armazenar o dado em questão, como, talvez, um parâmetro ou um objeto auxiliar.

Estado variável

Às vezes, dependendo de como o mesmo objeto é usado, são necessários elementos de dados diferentes. Não são apenas os valores que diferem; elementos completamente diferentes são apresentados em objetos da mesma classe.

Estados variáveis costumam ser armazenados como um mapa cujas chaves são os nomes dos elementos (representados como strings ou enumerações) e cujos valores são os dos dados.

```
class FlexibleObject {  
    Map<String, Object> properties= new HashMap<String, Object>();  
    Object getProperty(String key) {  
        return properties.get(key);  
    }  
    void setProperty(String key, Object value) {  
        properties.set(key, value);  
    }  
}
```

Um estado variável é muito mais flexível que um estado comum, mas seu principal defeito é que não comunica bem. Quais elementos de dados precisam estar presentes para que um objeto com apenas estados variáveis funcione corretamente? Ler cuidadosamente o código e, talvez, observar a execução ajudarão a encontrar a resposta para essa questão.

Li códigos em que o programador usava excessivamente estados variáveis. Todo objeto de uma dada classe tinha exatamente as mesmas chaves em seu mapa de propriedades. Teria sido muito mais fácil ler se tivesse a mesma informação como declarações de campo.

Um caso que parece justificar a presença de estados variáveis é quando o estado de um campo implica a necessidade de outros campos. Por exemplo, se tenho um componente de interface cuja flag `bordered` é verdadeira, posso ter também `borderWidth` e `borderColor`. Seria possível comunicar isso com estado variável, conforme o projeto no topo da Figura 6.1. Um estado comum também pode comunicar isso, como na parte de baixo da Figura 6.1.

A solução com estados comuns viola o princípio de que todas as variáveis em um objeto deveriam ter o mesmo tempo de vida. O polimorfismo fornece uma explicação mais clara da situação. Uma classe representa o estado sem borda, e outra, o estado com borda. `Bordered` tem um estado comum para representar seus parâmetros.

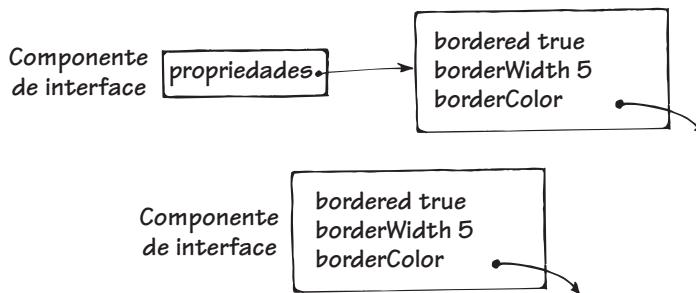


Figura 6.1 Borda representada por estado variável e estado comum.

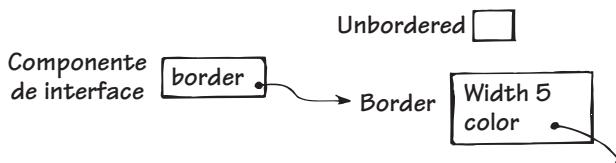


Figura 6.2 Um objeto auxiliar limpa o projeto.

A presença de muitas variáveis com um prefixo comum indica que pode ser útil um objeto auxiliar qualquer.

Use estado comum sempre que possível. Use estado variável para os campos de um objeto que podem ou não ser necessários dependendo do uso.

Estado extrínseco

Às vezes uma parte do programa precisa de estado associado a um objeto, mas o restante do sistema não se importa. Por exemplo, a informação de onde um objeto é armazenado no disco é útil para o mecanismo de persistência, mas não para o resto do código. Colocar esse dado em um campo violaria o princípio da simetria. Os outros campos são úteis para todo o sistema.

Armazene informação com propósito especial associada a um objeto próximo de onde ela será usada, e não no objeto. No exemplo acima, o mecanismo de persistência armazenaria um IdentityMap cujas chaves são os objetos armazenados e cujos valores são a informação sobre onde são armazenados.

Um problema do estado extrínseco é que fica difícil copiar um objeto. Replicar um objeto com estado extrínseco não é tão simples quanto replicar seus campos. Em vez disso, todo o estado extrínseco deve ser copiado corretamente, e isso poderia exigir tratamento diferente, dependendo de como o objeto é usado. Outro problema é a dificuldade em depurar objetos com estado extrínseco. Um inspetor convencional não mostra todos os dados associados a um objeto. Em razão dessas dificuldades, é raro haver estado extrínseco, mas eles são úteis quando necessários.

Variável

Em Java, objetos são referidos por variáveis. Leitores precisam conhecer o escopo, o tempo de vida, a função e o tipo de tempo de execução (runtime type) das variáveis. Embora tenham sido inventados planos elaborados para comunicar toda essa informação com nomes de variáveis, é preferível simplificar o código usando nomes simples.

O escopo de variáveis, o espaço no qual elas podem ser referidas, pode ser de três tipos: locais, acessíveis apenas dentro do escopo atual; campos, que podem ser acessados em qualquer lugar dentro de um objeto; ou estáticas, que podem ser acessadas por qualquer objeto de uma dada classe. O escopo de campos pode ser estendido pelos modificadores `public`, `package` (o padrão, uma estranha escolha por ser o menos usado), `protected` e `private`.

Quando se faz uso abundante de todas as combinações disponíveis, é importante para os leitores que as distinções sejam claras, a ponto de referenciá-las codificando-as no nome. Entretanto, para reduzir o acoplamento, devem-se usar sobretudo variáveis locais e campos com apenas um campo estático ocasional e o modificador `private`. Ao usar esse conjunto limitado de combinações possíveis, basta o contexto para informar ao leitor se ele está olhando para uma variável local ou um campo. Se ele puder ver a declaração, a variável é local; se não puder, é um campo. Isso elimina a necessidade de incluir informação de escopo no nome das variáveis. Em troca, tem-se um código com nomes de variáveis uniformes e fáceis de ler. Tudo isso pressupõe que é possível dividir seu código em pequenos trechos: uma propriedade que se pode aproveitar aplicando os outros padrões de implementação ou, mais notadamente, compondo métodos.

O tempo de vida das variáveis pode ser menor que seu escopo. Um campo só poderia ser válido se determinado método estiver ativo na pilha, e isso seria feio. Esforce-se para garantir que o tempo de vida das variáveis seja próximo a seu escopo. Além disso, assegure-se de que todas as variáveis irmãs (aqueles definidas no mesmo escopo) tenham o mesmo tempo de vida.

O tipo de uma variável é adequadamente comunicado por meio da declaração do tipo. Assegure-se de que o tipo declarado comunique o mais claramente possível (veja “Tipo declarado”). A única exceção a esse conselho são os nomes de variáveis que comportam múltiplos valores (aqueles que contêm uma coleção) e que deveriam estar no plural. A diferença entre um valor simples e valores múltiplos é importante para os leitores.

Com escopo, tempo de vida e tipo adequadamente comunicados por outros meios, o nome pode ser usado para transmitir a função da variável na computação. Reduzindo-se ao mínimo a informação a ser transportada, fica-se livre para escolher nomes simples e que facilitem a leitura.

Variável local

Variáveis locais só são acessíveis a partir de seu ponto de declaração e até o fim de seu escopo. Seguindo o princípio de que a informação deveria se espalhar

o mínimo possível, declare variáveis locais pouco antes de elas serem usadas e dentro do escopo mais interno possível.

Há um punhado de funções comuns para as variáveis locais:

- Coletor: uma variável que coleta informação para uso posterior. Frequentemente, o conteúdo dos coletores é retornado como o valor de uma função. Quando um coletor for retornado, nomeie-o como `result` ou `results`.
- Contador: um coletor especial que coleta a contagem de alguns outros objetos.
- Explicação: quando se tem uma expressão complicada, atribua pedaços da expressão a variáveis locais para ajudar os leitores a navegar pela complexidade:

```
int top= ...;
int left= ...
int height= ...;
int bottom= ...;
return new Rectangle(top, left, height, width);
```

Embora não sejam computacionalmente necessárias, as variáveis locais de explicação ajudam naquilo que, de outra forma, seria uma expressão longa e complicada.

Variáveis locais explicativas caminham na direção de métodos auxiliares. A expressão torna-se o corpo do método, e o nome da variável local sugere um nome para o método. Às vezes esses auxiliares são introduzidos para simplificar o método de chamada; outras vezes ajudam a eliminar a duplicação de expressões comuns.

- Reúso: quando um valor de expressão muda, mas é preciso usar o mesmo valor mais de uma vez, armazena-se o valor em uma variável local. Por exemplo, se for necessário o mesmo timestamp para muitos objetos, não se pode buscar um novo tempo para cada objeto:

```
for (Clock each: getClocks())
    each.setTime(System.currentTimeMillis());
```

Em vez disso, para os propósitos do programador, uma variável local para reúso congela o tempo:

```
long now= System.currentTimeMillis();
for (Clock each: getClocks())
    each.setTime(now);
```

- Elemento: o último uso comum de variáveis locais é manter os elementos de uma coleção que está sendo iterada. Como no exemplo dado, `each` [cada] é um nome claro e simples para um elemento de uma variável local. Quando se quer saber o que está sendo iterado, pode-se realçar na cláusula `for` acima.

Para laços aninhados, adicione o nome da coleção ao nome do elemento local para distingui-los:

```
broadcast() {  
    for (Source eachSender: getSenders())  
        for (Destination eachReceiver: getReceivers())  
            ...;  
}
```

Campo

O escopo e o tempo de vida de um campo são os mesmos do objeto ao qual está anexado. Como a fidelidade primária dos campos é ao objeto como um todo, declare campos juntos ou no início ou no final da classe. No início, as declarações dão ao leitor um importante contexto para ser usado enquanto lê o restante do código. Deixando as declarações para o final, envia-se a mensagem: “O comportamento é o principal; dados são apenas detalhes de implementação”. Embora eu concorde filosoficamente com a afirmação de que, em um programa, a lógica é mais importante que os dados, ainda gosto, quando estou lendo um código, de ver primeiro as declarações, onde quer que estejam.

Uma das opções em um campo é declará-lo como final, o que informa aos leitores que o valor do campo não muda depois que o construtor tiver sido executado. Embora eu mantenha mentalmente o rastro de quais campos são finais e quais não são, não declaro explicitamente os campos. A clareza a mais não compensa a maior complexidade, mas, se estiver escrevendo um código que será mudado por muitas pessoas durante um bom tempo, parece valer a pena manter explícita a distinção entre campos finais e voláteis.

A lista de regras para campos não é tão abrangente quanto a lista para variáveis locais. Contudo, a seguir estão descritas algumas funções comuns para campos:

- **Auxiliar:** campos auxiliares mantêm referências a objetos usados por muitos dos métodos de um objeto. Se um objeto é passado como parâmetro para muitos métodos, considere substituir o parâmetro por um campo auxiliar estabelecido no construtor completo.
- **Flag:** campos de flag booleanos significam “este objeto pode agir de duas formas distintas”. Se há um método setter para a flag, significa ainda “e o comportamento pode mudar durante a vida do objeto”. Campos de flag são bons se usados apenas em poucos condicionais. Se as decisões sobre a redação do código com base na flag forem duplicadas, considere mudar para um campo de estratégia.
- **Estratégia:** se você quiser expressar que existem meios alternativos para realizar uma parte da computação de um objeto, armazene em um campo um objeto que desempenhe apenas a parte variável da computação. Se essa variação no comportamento não mudar durante o tempo de vida de

um objeto, coloque o campo de estratégia no construtor completo. Caso contrário, forneça métodos para alterá-lo.

- Estado: campos de estado são como campos de estratégia naquela parte do comportamento de objeto delegada a eles. Contudo, campos de estado, quando acionados, estabelecem o estado seguinte por conta própria. Campos de estratégia, se for necessário, são alterados por outros objetos. Máquinas de estado implementadas assim podem ser difíceis de ler, uma vez que os estados e as transições não são expressos em um único lugar. Contudo, para máquinas de estado simples, podem ser suficientes.
- Componentes: esses campos mantêm objetos ou dados que são “pertencentes” ao objeto referido.

Parâmetro

Além das variáveis não privadas (campos ou campos estáticos), a única forma de comunicar o estado de um objeto para outro é por meio de parâmetros. Como variáveis não privadas introduzem forte acoplamento entre classes e como esse acoplamento tende a crescer com o tempo, parâmetros são preferíveis em todos os casos em que são possíveis campos estáticos e parâmetros.

O acoplamento introduzido por um parâmetro é mais fraco que aquele introduzido pela referência permanente de um objeto a outro. Por exemplo, cálculos no interior da estrutura de uma árvore por vezes precisam do pai de um nodo. Em vez de ter uma referência permanente ao pai (Figura 6.3), passar um parâmetro aos métodos que o exigem enfraquece o acoplamento entre os nodos. Sem referências permanentes a um pai, é possível, por exemplo, ter uma subárvore como parte de muitas árvores.

Se muitas mensagens transmitidas de um objeto para outro precisarem do mesmo parâmetro, talvez seja melhor anexar permanentemente o parâmetro ao objeto chamado. Parâmetros são fios finos que amarram os objetos, mas, como Gulliver com os liliputianos, uma quantidade suficiente de fios finos pode incapacitar a modificação de um objeto.

A Figura 6.4 mostra um parâmetro simples.

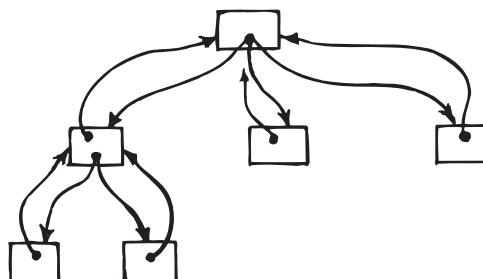


Figura 6.3 Estrutura de uma árvore altamente acoplada com ponteiros para os pais.



Figura 6.4 Um parâmetro simples introduz pouco acoplamento.

O que é ilustrado com o código:

```
Server s= new Server();
s.a(this);
```

Repetir o mesmo parâmetro cinco vezes aumenta substancialmente o acoplamento:

```
Server s= new Server();
s.a(this);
s.b(this);
s.c(this);
s.d(this);
s.e(this);
```

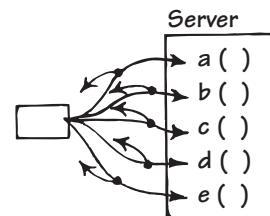


Figura 6.5 Parâmetro repetido aumenta o acoplamento.

Neste caso, os dois objetos são mais bem preparados para operar independentemente caso os parâmetros sejam substituídos por um ponteiro:

```
Server s= new Server(this);
s.a();
s.b();
s.c();
s.d();
s.e();
```

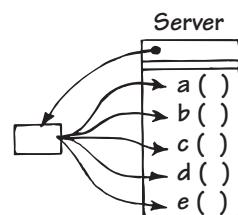


Figura 6.6 Referência reduz o acoplamento.

Parâmetro coletor

Cálculos que reúnem resultados de muitas invocações de métodos precisam de uma maneira de mesclar esses resultados. Um jeito é retornar um valor a partir de todos os métodos. Isso funciona se o valor for simples, como um inteiro.

```
Node
int size() {
    int result= 1;
    for (Node each: getChildren())
        result+= each.size();
    return result;
}
```

Quando mesclar resultados é mais complicado que uma simples adição, é melhor passar um parâmetro que coletará os resultados. Por exemplo, um parâmetro coletor ajuda a linearizar uma árvore:

```
Node
asList() {
    List results= new ArrayList();
    addTo(results);
    return results;
}
addTo(List elements) {
    elements.add(getValue());
    for (Node each: getChildren())
        each.addTo(elements);
}
```

Outros exemplos de parâmetros coletores mais complicados são o `GraphicsContext`, passado ao redor de uma árvore de componentes de interface, e o `TestResult`, passado em torno de uma árvore de testes em JUnit.

Parâmetro opcional

Alguns métodos podem aproveitar um parâmetro ou fornecer um parâmetro padrão caso não haja nenhum. Nesses casos, colocam-se os parâmetros obrigatórios no início da lista de parâmetros e incluem-se os parâmetros opcionais no final. Isso torna invariável a maior quantidade de parâmetros possível, e os parâmetros opcionais aparecem como alternativas no final.

Os construtores `ServerSocket` demonstram parâmetros opcionais. O construtor básico não aceita argumentos, mas há uma versão com um número de porta opcional e outra com um número de porta opcional e um comprimento de backlog opcional:

```
public ServerSocket()
public ServerSocket(int port)
public ServerSocket(int port, int backlog)
```

Linguagens com parâmetros de palavras-chave podem expressar parâmetros opcionais mais diretamente. Como Java tem apenas parâmetros posicionais, a única maneira de expressar um parâmetro opcional é por convenção. Algumas pessoas chamam isso de padrão de parâmetro telescópico, fazendo uma analogia física sobre como coleções de parâmetros ficam uma sobre outra.

Var args (quantidade variável de argumentos)

Alguns métodos podem usar qualquer quantidade de certo tipo de parâmetro. Uma solução simples é sempre passar uma coleção como o parâmetro. Chamadores desses métodos, contudo, são atrapalhados pela presença de coleções intermediárias:

```
Collection<String> keys= new ArrayList<String>();
keys.add(key1);
keys.add(key2);
object.index(keys);
```

Esse problema é bastante comum, de forma que Java fornece um mecanismo para passar um número variável de argumentos para um método. Declarando-se o método acima como `method(Class... classes)`, o cliente pode invocar o método com qualquer número de argumentos:

```
object.index(key1, key2);
```

Var arg deve ser o último parâmetro. Se um método tem var arg e argumentos opcionais, como descrito acima, os argumentos opcionais devem ir antes do var arg.

Objeto parâmetro

Se um grupo de parâmetros é passado para muitos métodos, considere construir um objeto cujos campos são exatamente esses parâmetros e passe o objeto como parâmetro. Depois de substituir a longa lista de parâmetros pelo objeto parâmetro, verifique se há pedaços de código usando apenas os campos do objeto parâmetro, de forma que se possa transformá-los em métodos do objeto parâmetro.

Por exemplo, é comum em bibliotecas gráficas de Java representarem-se retângulos como parâmetros independentes `x`, `y`, `width` [largura] e `height` [altura]. Às vezes esses quatro parâmetros atravessam muitas camadas de invocação de métodos, resultando em um código mais extenso e mais difícil de ler do que o necessário.

```
setOuterBounds(x, y, width, height);
setInnerBounds(x + 2, y + 2, width - 4, height - 4);
```

Explicitar o retângulo como um objeto explica melhor o código:

```
setOuterBounds(bounds);
setInnerBounds(bounds.expand(-2));
```

A introdução do objeto parâmetro encurta o código, explica sua intenção e fornece um lar para o algoritmo de expansão e contração de um retângulo, o qual teria de ser repetido em todos os lugares em que fosse necessário (com erros frequentes quando os programadores esquecessem que o fator de expansão deve ser duplicado para largura e altura). Muitos objetos poderosos surgem como objetos parâmetro.

A motivação inicial para introduzir objetos parâmetro é melhorar a legibilidade e eles podem se tornar lares importantes para a lógica. O fato de dados aparecerem juntos em muitas listas de parâmetros é uma clara indicação de que estão fortemente relacionados. Uma classe com uma lista fixa de campos é uma forma explícita de comunicar “Esse conjunto de dados está fortemente relacionado”.

O argumento mais usado contra objetos parâmetro é o desempenho – alojar todos esses objetos parâmetro leva tempo. Na maioria dos casos, na prática, isso não é um problema. Se a alocação de objetos torna-se um gargalo, o objeto parâmetro pode ser incorporado ao código e retornado à lista explícita de parâmetros, onde necessário. O melhor código para otimizar é legível, fatorado e testado; objetos parâmetro podem contribuir para essas metas.

Constante

Às vezes há valores de dados que são necessários em diversos lugares dentro de um programa, mas que não mudam. Se esses valores são conhecidos no momento de compilar, armazene em variáveis declaradas static final e refira a variável em todo o programa. É comum dar nomes às constantes apenas com caracteres em maiúsculas, de forma a enfatizar que não são variáveis ordinárias.

Parte da importância de se usar constantes é evitar uma classe inteira de erros quando usados. Quando se tem 5 embutido por todo o código e decide-se mudar 5 para 6, é fácil esquecer uma instância. Se 5 tem dois significados implícitos – digamos “desenhar uma borda” e “o que vem a seguir é um pacote de confirmação” –, mudar a constante é uma ação ainda mais propensa a erros. A mais forte razão para se usar constantes, todavia, é poder usar os nomes das constantes para comunicar o que se quer informar com o valor. Leitores conseguem entender Color.WHITE muito mais facilmente que 0xFFFFF. Se a codificação da cor muda, o código que usa a constante não precisa ser alterado.

Um uso comum de constantes é comunicar variações de uma mensagem em uma interface. Por exemplo, para centralizar o texto, pode-se invocar setJustification(Justification.CENTERED). Uma vantagem desse estilo de API é que é possível adicionar novas variantes dos métodos existentes incluindo novas constantes sem quebrar os implementadores. Contudo, essas mensagens não comunicam tão bem quanto ter um método separado para cada variação. Nesse estilo, a mensagem acima seria `justifyCentered()`. Uma interface em que todas as invocações de um método têm constantes literais como argumentos pode ser melhorada dando-se a ela métodos separados para cada valor constante.

Nome sugestivo de função

Como se decide que nome dar a uma variável? Muitas restrições conflitantes afetam essa questão. Quero comunicar minha intenção por meio dos nomes, o que costuma exigir nomes longos. Gostaria que os nomes fossem curtos, para simplificar a formatação do código. Os nomes são muito mais lidos do que digitados; por isso, deveriam ser otimizados para legibilidade, e não para facilidade de digitação. É preciso expressar tanto a forma como os dados são usados na variável quanto a função que assumem na computação.

Há muitos pedaços de informação necessários quando se está tentando entender uma variável. Qual é seu propósito na computação? Como é referido

o objeto pela variável usada? Quais são o escopo e o tempo de vida da variável? Quantas vezes a variável é referida?

Muitas técnicas de nomeação de variáveis incluem informação de tipo nos nomes. A minha, não. Qual é o objetivo de se dizer repetidas vezes ao compilador os tipos das variáveis se for preciso voltar e incorporar novamente aquela mesma informação nos nomes das variáveis? Entendo a inclusão de informação de tipo em nomes de variáveis em linguagens que não se esforçam muito para prevenir erros de tipagem, como C. Mas Java fornece amplo suporte para evitar esses erros.

Se quero saber o tipo de uma variável, minha IDE dá um feedback rápido sobre isso. Usar métodos curtos e compostos também fornece uma referência rápida para as variáveis, as variáveis locais e os parâmetros mais comumente usados.

Outra faceta das variáveis que os leitores precisam entender é o seu escopo. Algumas práticas de nomeação de variáveis codificam o escopo como um prefixo no nome, de forma que `fCount` é um campo e `lCount` é uma variável local. Mais uma vez, compondo-se métodos relativamente curtos, raramente o escopo de uma variável causa confusão. Se não se consegue ver uma declaração da variável por esse método, muito provavelmente trata-se de um campo (uso outras técnicas para evitar a maioria dos campos estáticos).

Isso acaba com a função da variável como parte primordial da informação que se tem de transmitir a partir dos nomes de variáveis, levando geralmente a nomes curtos e claros. Se tenho de me esforçar demais para encontrar um nome, geralmente é porque não entendo muito bem a computação.

Há alguns nomes de variáveis recorrentes em um código:

- `result` – armazena o objeto que será retornado por uma função;
- `each` – armazena os elementos individuais de uma coleção que estão iterando (embora ultimamente eu tenha preferido usar a forma singular do nome da coleção, por exemplo `for (Node child: getChildren())`);
- `count` – armazena contagens.

Quando há múltiplas variáveis às quais se gostaria de dar o mesmo nome, qualifica-se o nome: `eachX` e `eachY` ou `rowCount` e `columnCount`.

Por vezes, fico tentado a abreviar palavras em nomes de variáveis. Isso otimiza a digitação, mas dificulta a leitura. Como as variáveis são mais lidas do que escritas, trata-se de uma falsa economia. Às vezes fico tentado a usar muitas palavras para uma variável, o que a torna muito longa para uma digitação confortável. Quando isso acontece, observo o contexto. Por que preciso de tantas palavras para distinguir a função dessa variável da de outras variáveis? Muitas vezes isso me leva a simplificar o projeto, permitindo-me dar, com consciência tranquila, nomes curtos para as variáveis.

Resumindo, comunico a função de uma variável por meio de seu nome. Tudo o mais que é importante sobre a variável – seu tempo de vida, escopo e tipo – pode ser comunicado por meio do contexto.

Tipo declarado

Uma das características de Java e de outras linguagens tipadas de modo pessimista é a necessidade de declarar os tipos de suas variáveis. Como é preciso declarar o tipo, pode-se também usar o tipo declarado como uma oportunidade para comunicar. Escolha o tipo declarado que comunica como a variável deve ser usada, e não como ela é implementada.

`List<Person> members= new ArrayList<Person>()` diz que `membros` é usado como uma `List`. Espera-se ver operações invocadas como `get()` e `set()`, já que o que diferencia `List` de `Collection` é o acesso indexado a elementos.

Quando elaborei esse padrão, anotei-o dogmaticamente. Então, experimentei a regra rígida de que todas as variáveis devem ser declaradas da forma mais genérica possível. Mas descobri que não valia a pena o esforço para generalizar todos os tipos. Às vezes uma variável era uma `List`, e então eu a passava para um método em que apenas o protocolo `Collection` fosse usado. A inconsistência entre as declarações era um problema maior para os leitores do que a falta de precisão em declará-la como uma `List` em todo lugar que fosse usada. Agora eu diria mais gentilmente: é útil, sempre que possível, declarar variáveis e métodos com um tipo universal. Perder um pouco de precisão e generalidade para manter a consistência é uma troca razoável.

A melhor coisa em generalizar tipos declarados é que isso abre opções para mudar as classes concretas durante modificações posteriores. Se declaro uma variável como um `ArrayList`, não consigo facilmente mudá-la depois para um `HashSet` da mesma forma que o faria se a declarasse como uma `Collection`. Em geral, quanto mais distante uma decisão se propaga, menos flexibilidade se tem para mudanças futuras. Para manter a flexibilidade, permita que o mínimo possível de informação se espalhe de forma bem restrita. Há mais informação na declaração “`membros` contém um `ArrayList`” que na declaração “`membros` contém uma `Collection`”.

Concentrar-se na comunicação é uma boa heurística para manter a flexibilidade, e tipos declarados são um exemplo disso. Quando digo que uma variável possui uma `Collection`, estou falando com precisão. Comunicar bem proporciona a melhor flexibilidade.

Inicialização

Antes de programar, é preciso saber com o que se pode contar. Ser capaz de fazer suposições precisas ajuda o programador a se concentrar em aprender o que é necessário saber. O estado das variáveis ajuda o programador a fazer suposições. A inicialização é o processo de colocar variáveis em um estado conhecido antes de elas serem usadas.

Há muitas questões sobre a inicialização de variáveis. Uma delas é o desejo de tornar a inicialização tão declarativa quanto possível. Se a inicialização e a declaração andam juntas, há um lugar para buscar respostas para questões sobre a variável. Outra questão é o desempenho. Pode ser necessário inicializar

variáveis difíceis de inicializar algum tempo depois de elas surgirem. Por exemplo, no Eclipse, as classes são carregadas o mais tarde possível para manter baixo o tempo de inicialização do sistema.

A seguir, estão listados dois padrões de inicialização: a ansiosa e a preguiçosa.

Inicialização ansiosa

Um estilo é inicializar a variável tão logo ela surja – quando é declarada ou quando é criado o objeto no qual vive (declaração ou construtor). Uma vantagem da inicialização ansiosa é que se pode assegurar que as variáveis são inicializadas antes de serem usadas.

Se possível, inicialize as variáveis na declaração. Isso deixa próximos os tipos declarados e reais, facilitando a vida dos leitores.

```
class Library {  
    List<Person> members= new ArrayList<Person>();  
    ...  
}
```

Initialize campos no construtor caso não possam ser inicializados na declaração:

```
class Point {  
    int x, y;  
    Point(int x, int y) {  
        this.x= x;  
        this.y= y;  
    }  
}
```

Há certa simetria em inicializar todos os campos de um objeto no mesmo lugar, seja nas declarações ou no construtor. Contudo, misturar os dois estilos não parece causar nenhuma confusão, desde que os objetos sejam mantidos em um tamanho razoável.

Inicialização preguiçosa

A inicialização ansiosa funciona bem quando o programador não se importa com as consequências de computar um valor de variável quando a variável ganha existência. Quando a computação é dispendiosa e se quer diminuir o custo (talvez porque a variável possa nunca ser usada), cria-se um método getter e inicializa-se o campo quando o getter for chamado pela primeira vez:

```
Library.Collection<Person> getMembers() {  
    if (members == null)  
        members= new ArrayList<Person>();  
    return members;  
}
```

A inicialização preguiçosa costumava ser uma técnica mais utilizada. A limitação do poder computacional bruto era uma questão mais presente. Assim, a inicialização preguiçosa é importante quando o poder computacional é um recurso limitado. Um ambiente de recursos restritos, como o Eclipse, em que a inicialização do sistema deve ser rápida, usa-se inicialização preguiçosa para evitar carregar plug-ins até estarem prestes a serem usados.

Um campo inicializado de forma preguiçosa é mais difícil de ler que um inicializado ansiosamente. O leitor tem de olhar ao menos dois lugares antes de entender o tipo de implementação do campo. Enquanto se codifica, está-se armazenando informação para futuros leitores. Felizmente, há poucas dúvidas frequentes, e assim poucas técnicas são suficientes para responder a maioria delas. A inicialização preguiçosa diz “O desempenho é importante”.

Conclusão

Os padrões de estado falam sobre como comunicar decisões sobre a representação de um estado no programa. O próximo capítulo apresenta o outro lado da moeda: como comunicar decisões sobre o fluxo de controle.

CAPÍTULO 7

Comportamento

John Von Neumann contribuiu com uma das primeiras metáforas da computação – uma sequência de instruções que são executadas uma a uma. Essa metáfora permeia a maioria das linguagens de programação, incluindo Java. O tópico deste capítulo é como expressar o comportamento de um programa. Os padrões são:

- Fluxo de controle (control flow) – expressa computações como uma sequência de passos.
- Fluxo principal (main flow) – expressa claramente o principal fluxo de controle.
- Mensagem (message) – expressa o fluxo de controle enviando uma mensagem.
- Mensagem de escolha (choosing message) – variam-se os implementadores de uma mensagem para expressar escolhas.
- Envio duplo (double dispatch) – variam-se os implementadores de mensagens ao longo de dois eixos para expressar escolhas que se propagam.
- Mensagem de decomposição (decomposing message) – quebram-se cálculos complicados em pedaços coesos.
- Mensagem inversa (reversing message) – torna simétricos os fluxos de controle enviando uma sequência de mensagens ao mesmo recebedor.
- Mensagem convidativa (inviting message) – estimula-se uma variação futura enviando uma mensagem que pode ser implementada de diferentes formas.
- Mensagem explicativa (explaining message) – envia-se uma mensagem que explica o propósito de um amontoado de lógica.
- Fluxo excepcional (exceptional flow) – expressam-se os fluxos de controle não usuais da forma mais clara possível, sem interferir na expressão do fluxo principal.

- Cláusula de guarda (guard clause) – expressam-se fluxos locais excepcionais com um retorno prematuro.
- Exceção (exception) – expressam-se fluxos não locais excepcionais com exceções.
- Exceção verificada (checked exception) – garante-se que exceções sejam capturadas declarando-as explicitamente.
- Propagação de exceção (exception propagation) – propagam-se exceções transformando-as conforme o necessário, de forma que a informação que elas contêm é adequada ao capturador.

Fluxo de controle

Por que temos fluxos de controle em programas? Há linguagens como Prolog que não têm a noção explícita de um fluxo de controle. Pedaços de lógica flutuam como em uma sopa, esperando as condições certas para se tornarem ativas.

Java é membro da família de linguagens em que a sequência de controle é um princípio fundamental de organização. Instruções adjacentes executam em sequência. Condicionais fazem o código executar apenas em certas circunstâncias. Laços executam o código repetidamente. Mensagens são enviadas para ativar uma de muitas sub-rotinas. Exceções fazem o controle saltar para cima da pilha.

Todos esses mecanismos contribuem para formar um ambiente rico para expressar computações. O autor/programador decide se o fluxo que tem em mente deve ser expresso como um fluxo principal com exceções, como múltiplos fluxos alternativos (todos igualmente importantes) ou como uma combinação deles. Agrupam-se pedaços do fluxo de controle de forma que o leitor casual possa entendê-los de forma abstrata no início, ficando os detalhes maiores disponíveis para aqueles que precisarem entendê-los. Alguns agrupamentos são rotineiros em uma classe; outros são de delegação de controle para outro objeto.

Fluxo principal

Programadores costumam ter em mente um fluxo principal de controle em seus programas. O processamento começa em um ponto e termina em outro. Pode haver decisões e exceções ao longo do caminho, mas a computação tem um trajeto a seguir. Use a linguagem de programação para expressar claramente esse fluxo.

Alguns programas, particularmente aqueles projetados para funcionar com segurança em circunstâncias hostis, não têm um fluxo principal visível. Contudo, esses programas são minoria. Usar o poder expressivo da linguagem de programação para expressar com clareza fatos pouco executados e raramente mutáveis de um programa obscurece sua parte mais poderosa: a parte que será lida, entendida e mudada frequentemente. Não que condições excep-

cionais não sejam importantes, mas é mais valioso focar a expressão clara do fluxo principal da computação.

Portanto, expresse claramente o fluxo principal do programa. Use exceções e cláusulas de guarda para expressar condições não usuais ou condições de erros.

Mensagem

Um dos meios básicos de se expressar lógica em Java é a mensagem. Linguagens procedurais usam chamadas de procedimento como um mecanismo para esconder informações:

```
compute() {  
    input();  
    process();  
    output();  
}
```

Isso significa “Para entender essa computação, basta saber que ela consiste desses três passos; os detalhes não são importantes por enquanto”. Uma das belezas de programar com objetos é que o mesmo procedimento também expressa algo mais rico. Para cada método, há potencialmente um conjunto inteiro de computações estruturadas de maneira similar mas com detalhes diferentes. E, como bônus, não é preciso resolver os detalhes de todas as variações futuras quando se escreve a parte invariante.

Usar mensagens como o principal mecanismo de fluxo de controle indica que as mudanças são o estado-base de programas. Cada mensagem é um lugar potencial onde se pode alterar o recebedor da mensagem sem mudar quem a envia. Em vez de dizer “Há algo ali cujos detalhes não importam”, a versão baseada em mensagens do procedimento diz “Nesse ponto da história, algo interessante acontece em torno da ideia de entrada, e os detalhes podem variar”. Usar sabiamente essa flexibilidade, tornando claras e diretas as expressões de lógica quando possível e adiando detalhes, é uma habilidade importante para quem quer escrever programas que comunicam de modo eficaz.

Mensagem de escolha

Às vezes envio mensagens para escolher uma implementação, da mesma forma que se usa uma declaração de *case* em linguagens procedurais. Por exemplo, se quero apresentar um gráfico em uma de muitas formas, envio uma mensagem polimórfica para comunicar que uma escolha será tomada em tempo de execução.

```
public void displayShape(Shape subject, Brush brush) {  
    brush.display(subject);  
}
```

A mensagem `display()` escolhe a implementação com base no tipo do pincel (brush) em tempo de execução. Então tem-se a liberdade para implementar uma variedade de pincéis: `ScreenBrush`, `PostscriptBrush`, etc.

O livre uso de mensagens de escolha leva a códigos com poucos condicionais explícitos. Cada mensagem de escolha é um convite à extensão posterior. Cada condicional explícito é outro ponto do programa que necessitará de modificação explícita para modificar o comportamento do programa inteiro.

Para ler códigos que usam muitas mensagens de escolha, é preciso ter a habilidade de aprender. Um dos problemas das mensagens de escolha é que o leitor pode ter de olhar muitas classes para entender os detalhes de um caminho particular da computação. Como programador, pode-se ajudar o leitor a navegar dando aos métodos nomes que revelam sua intenção. Também se deve saber quando uma mensagem de escolha é exagerada. Se não houver variações possíveis em uma computação, não se deve introduzir um método apenas para fornecer a possibilidade de variação.

Envio duplo

Mensagens de escolha são boas para expressar uma dimensão única de variabilidade. No exemplo em “Mensagem de escolha”, a dimensão foi o tipo do meio em que a forma seria desenhada. Quando se precisa expressar duas dimensões de variabilidade independentes, podem-se colocar em cascata duas mensagens de escolha.

Por exemplo, suponha que se queira indicar que um oval em Postscript é computado diferentemente de um retângulo na tela. Primeiro, deve-se decidir onde quer que fique a computação. As computações-base parecem pertencer a `Brush` (Pincel), de forma que envia-se a mensagem de escolha primeiro a `Shape` (Forma) e, depois, a `Brush`:

```
displayShape(Shape subject, Brush brush) {  
    subject.displayWith(brush);  
}
```

Agora cada `Shape` pode implementar `displayWith()` diferentemente. Em vez de se ocupar com detalhes, contudo, eles acrescentam seu tipo à mensagem e transferem para `Brush`:

```
Oval.displayWith(Brush brush) {  
    brush.displayOval(this);  
}  
Rectangle.displayWith(Brush brush) {  
    brush.displayRectangle(this);  
}
```

Assim, os diferentes tipos de pincéis têm a informação de que precisam para agir:

```
PostscriptBrush.displayRectangle(Rectangle subject) {
    writer.print(subject.left() + " " + ... + " rect");
}
```

O envio duplo introduz uma duplicação, gerando, em consequência, perda de flexibilidade. Os nomes dos tipos dos recebedores da primeira mensagem de escolha ficam espalhados nos métodos do recebedor da segunda mensagem de escolha. Neste exemplo, isso significa que, para adicionar uma nova Shape, seria preciso adicionar métodos em todos os Brushes. Se uma dimensão é mais propícia a mudanças que outra, faça com que ela receba a segunda mensagem de escolha.

O cientista da computação dentro de mim quer generalizar para envio triplo, quádruplo, quíntuplo. Entretanto, tentei envio triplo apenas uma vez, e não durou muito. Sempre encontrei formas mais claras de expressar lógica multidimensional.

Mensagem de decomposição (sequenciamento)

Quando se tem um algoritmo complicado, composto de muitos passos, às vezes podem-se agrupar passos relacionados e enviar uma mensagem para invocá-los. A finalidade da mensagem não é fornecer um gancho para especialização ou algo sofisticado assim. É apenas a boa e velha decomposição funcional. A mensagem está lá simplesmente para invocar a subsequência de passos na rotina.

Mensagens de decomposição precisam ser nomeadas descritivamente. A maioria dos leitores deve estar apta a compreender o que precisa saber sobre o propósito da subsequência apenas a partir do nome. Apenas leitores interessados nos detalhes da implementação deveriam ser obrigados a ler o código invocado pela mensagem de decomposição.

Dificuldades para nomear uma mensagem de decomposição indicam que este não é o padrão certo a ser usado. Outra indicação são longas listas de parâmetros. Se vejo esses sintomas, reverto o método invocado pela mensagem de decomposição e aplico um padrão diferente, como objeto método, que me ajude a comunicar a estrutura do programa.

Mensagem inversa

A simetria pode melhorar a legibilidade do código. Considere o seguinte código:

```
void compute() {
    input();
    helper.process(this);
    output();
}
```

Embora esse método seja composto de três outros, falta simetria. A legibilidade do método é melhorada introduzindo-se um método auxiliar que revela

a simetria latente. Agora, quando se lê `compute()`, não é preciso rastrear quem envia as mensagens – todas elas vão para `this`.

```
void process(Helper helper) {  
    helper.process(this);  
}  
void compute() {  
    input();  
    process(helper);  
    output();  
}
```

Dessa forma, o leitor consegue entender a estrutura do método `compute()` lendo apenas uma classe.

De vez em quando, o método auxiliar invocado por uma mensagem inversa torna-se importante. Às vezes, o uso excessivo de mensagens inversas pode obscurecer a necessidade de mover a funcionalidade.

```
void input(Helper helper) {  
    helper.input(this);  
}  
void output(Helper helper) {  
    helper.output(this);  
}
```

O código acima provavelmente estaria mais bem estruturado movendo-se todo o método `compute()` para a classe `Helper`:

```
compute() {  
    new Helper(this).compute();  
}  
Helper.compute() {  
    input();  
    process();  
    output();  
}
```

Algumas vezes sinto-me tolo introduzindo métodos “apenas” para satisfazer a um impulso “estético”, como a simetria. A estética vai além disso. Ela ocupa mais o cérebro humano do que um pensamento lógico estritamente linear. Depois de cultivar um senso de estética de códigos, as impressões estéticas recebidas de um código são um feedback valioso sobre sua qualidade. Esses sentimentos que borbulham sob a superfície do pensamento simbólico podem ser tão valiosos quanto padrões explicitamente nomeados e justificados.

Mensagem convidativa

Às vezes, quando se está escrevendo um código, espera-se que as pessoas queiram alterar uma parte da computação em uma subclasse. Envia-se uma mensagem nomeada apropriadamente para comunicar a possibilidade de refinamento posterior, o que encoraja os programadores a, mais tarde, adequarem a computação a seus próprios propósitos.

Se houver uma implementação padrão da lógica, torne-a a implementação da mensagem. Se não, declare o método abstrato para explicitar o convite.

Mensagem explicativa

A distinção entre intenção e implementação sempre foi importante no desenvolvimento de software. É isso que permite que se entenda a computação, primeiramente em essência e, depois, se necessário, em detalhes. Podem-se usar mensagens para fazer essa distinção, enviando uma com o nome do problema que se está resolvendo, o qual por sua vez envia uma mensagem cujo nome é o modo como o problema vai ser resolvido.

O primeiro exemplo que vi disso foi em Smalltalk. Transliterado, o método que me chamou a atenção foi este:

```
highlight(Rectangle area) {
    reverse(area);
}
```

Pensei: “Por que isso é útil? Por que não apenas chamar `reverse()` diretamente em vez de chamar o método intermediário `highlight()`?”. Depois de pensar um pouco, percebi que, embora `highlight()` não tivesse um propósito computacional, ele servia para comunicar uma intenção. A chamada de código podia ser escrita em termos do problema que estava tentando resolver, isto é, realçar uma área da tela.

Considere introduzir uma mensagem explicativa quando estiver tentado a comentar uma única linha de código. Quando vejo:

```
flags |= LOADED_BIT; // Set the loaded bit
```

Preferiria ler:

```
setLoadedFlag();
```

ainda que a implementação de `setLoadedFlag()` seja trivial. O método de uma linha está ali para comunicar.

```
void setLoadedFlag() {
    flags |= LOADED_BIT;
}
```

Às vezes os métodos auxiliares invocados por mensagens explicativas tornam-se pontos valiosos para extensões futuras. É legal aproveitar a sorte quando ela aparece. Contudo, o propósito principal ao se invocar uma mensagem explicativa é comunicar sua intenção mais claramente.

Fluxo excepcional

Assim como programas têm um fluxo principal, eles também podem ter um ou mais fluxos excepcionais. São caminhos da computação cuja comunicação não

é tão importante, pois sua execução e suas alterações são menos frequentes e conceitualmente menos importantes que as do fluxo principal. Expressse o fluxo principal claramente, e esses caminhos excepcionais, o mais claramente possível, sem obscurecer o fluxo principal. Cláusulas de guarda e exceções são duas formas de expressar fluxos excepcionais.

É mais fácil ler programas se as instruções executam uma depois da outra. Leitores podem usar confortáveis e conhecidas técnicas de leitura de texto em prosa para entender a intenção do código. Às vezes, porém, existem múltiplos caminhos em um programa. Expressar da mesma maneira todos esses caminhos resultaria em uma tigela de minhocas, com flags atribuídas *aqui* e usadas *ali* e retornando valores com significados especiais. Responder à questão básica “Quais instruções são executadas?” torna-se um exercício que combina arqueologia e lógica. Pegue o fluxo principal, expresse-o claramente e use exceções para expressar outros caminhos.

Cláusula de guarda

Embora os programas tenham um fluxo principal, algumas situações necessitam de desvios desse fluxo. As cláusulas de guarda são uma forma de expressar situações excepcionais simples e locais, com consequências puramente locais. Compare o seguinte:

```
void initialize() {  
    if (!isInitialized()) {  
        ...  
    }  
}
```

com:

```
void initialize() {  
    if (isInitialized())  
        return;  
    ...  
}
```

Quando leio a primeira versão, forço-me a procurar uma cláusula *else* enquanto estou lendo a cláusula *then*. Mentalmente coloco a condição em uma pilha. Isso acaba sendo uma distração enquanto estou lendo o corpo da cláusula *then*. As primeiras duas linhas da segunda versão simplesmente apresentam um fato: o recebedor não foi inicializado.

As cláusulas *if*, *then* e *else* expressam fluxos de controle alternativos e igualmente importantes. Uma cláusula de guarda é adequada para expressar uma situação diferente, em que um dos fluxos de controle é mais importante que o outro. No exemplo de inicialização acima, o fluxo de controle importante é o que acontece quando o objeto é inicializado. Além disso, há um fato simples que deve ser notado: mesmo que seja solicitado que um objeto inicialize múltiplas vezes, ele executará o código de inicialização uma única vez.

Antigamente havia um mandamento na programação: toda rotina deve ter uma só entrada e uma só saída. Era uma maneira de prevenir a possível confusão quando se entrava e saía de muitos locais na mesma rotina. Isso fazia sentido quando era aplicado a programas na linguagem FORTRAN ou assembly, escritos com muitos dados globais, em que era difícil até mesmo entender qual instrução era executada. Em Java, com métodos pequenos e maioria de dados locais, isso é desnecessariamente conservador. Entretanto, esse resquício do folclore de programação, instintivamente obedecido, evita o uso de cláusulas de guarda.

Cláusulas de guarda são particularmente úteis quando há múltiplas condições:

```
void compute() {
    Server server= getServer();
    if (server != null) {
        Client client= server.getClient();
        if (client != null) {
            Request current= client.getRequest();
            if (current != null)
                processRequest(current);
        }
    }
}
```

Condicionais aninhados geram erros. A versão com cláusula de guarda do mesmo código observa os pré-requisitos de processar um pedido sem estruturas complexas de controle:

```
void compute() {
    Server server= getServer();
    if (server == null)
        return;
    Client client= server.getClient();
    if (client == null)
        return;
    Request current= client.getRequest();
    if (current == null)
        return;
    processRequest(current);
}
```

Uma variante da cláusula de guarda é a declaração `continue` usada em laço, que diz “Não se importe com esse elemento. Passe para o próximo”.

```
while (line = reader.readLine()) {
    if (line.startsWith('#') || line.isEmpty())
        continue;
    // Normal processing here
}
```

Novamente, a intenção é apontar a diferença (estritamente local) entre processamento normal e excepcional.

Exceção

Exceções são úteis para expressar saltos no fluxo do programa que alcançam níveis de invocação de função. Quando se nota um problema em níveis muito altos da pilha – o disco está cheio ou uma conexão de rede foi perdida –, pode-se lidar razoavelmente com esse fato apenas nos níveis mais baixos da pilha de chamadas. Inserir uma exceção no ponto de descoberta e capturá-la no ponto em que pode ser tratada é muito melhor que atravancar todo o código precedente com verificações explícitas para todas as condições excepcionais possíveis, sendo que nenhuma delas pode ser tratada.

Exceções têm custo. São uma espécie de vazamento no projeto. O fato de o método chamado lançar uma exceção influencia no projeto e na implementação de todos os possíveis métodos chamadores até que seja alcançado o método que captura a exceção. Elas tornam difícil rastrear o fluxo de controle, uma vez que instruções adjacentes podem estar em métodos, objetos ou pacotes diferentes. Um código que poderia ser escrito com condicionais e mensagens, mas é implementado com exceções, é terrivelmente difícil de ler, pois fica-se eternamente tentando descobrir o que mais está acontecendo além de uma estrutura de controle simples. Resumindo, sempre que possível, expresse o fluxo de controle com sequência, mensagens, iteração e condicionais (nesta ordem). Use exceções apenas quando não fazê-lo confundiria o fluxo de controle comunicado com clareza.

Exceções verificadas

Um dos perigos das exceções é o que acontece quando elas são lançadas, mas ninguém as captura. O programa fecha – é isso que acontece. Mas todos gostariam de controlar quando o programa fecha inesperadamente, exibindo as informações necessárias para diagnosticar a situação e mostrar ao usuário o que aconteceu.

Exceções que são lançadas mas não capturadas são um risco ainda maior quando pessoas diferentes escrevem o código que lança a exceção e o código que captura a exceção. Qualquer falha de comunicação resulta em fechamento abrupto e grosseiro do programa.

Para evitar essa situação, o Java tem exceções verificadas, que são declaradas explicitamente pelo programador e verificadas pelo compilador. Um código sujeito a ter uma exceção verificada deve capturar a exceção ou passá-la adiante.

Exceções verificadas têm um custo considerável. O primeiro é das próprias declarações, pois podem facilmente aumentar em 50% o tamanho das declarações de método, e mais um pouco para lê-las e entendê-las ao longo dos níveis entre o lançador e o capturador. Exceções verificadas também dificultam a mudança de códigos. Refatorar o código com exceções verificadas é mais difícil e tedioso do que codificar sem elas, ainda que IDEs modernas reduzam o fardo.

Propagação de exceção

Exceções ocorrem em diferentes níveis de abstração. Capturar e relatar uma exceção de nível baixo pode ser confuso para alguém que não a esperava. Quando um servidor web mostra uma página de erro com o rastro da pilha encabeçado por um `NullPointerException`, não tenho certeza do que deveria fazer com a informação. Vejo em vez disso uma mensagem que diz “O programador não considerou o cenário que acabou de ser apresentado”. Eu não me importaria se a página também fornecesse um indicador para informações adicionais, o qual eu enviaria a um programador para que diagnosticasse o problema, mas mostrar detalhes não traduzidos não é nada útil.

Exceções de nível baixo frequentemente contêm informações valiosas para o diagnóstico de um defeito. Insira a exceção de nível baixo na exceção de nível mais alto, para que, quando a exceção for exibida ou impressa – em um registro, por exemplo –, haja informação suficiente para se encontrar o defeito.

Conclusão

Em um programa construído por objetos, o controle flui entre os métodos. O próximo capítulo descreve o uso de métodos para expressar os conceitos de uma computação.

CAPÍTULO 8

Métodos

A lógica é dividida em métodos, e não espremida em um grande aglomerado. Por quê? Quais problemas podem ser resolvidos introduzindo-se um novo método? Qual é o objetivo de se ter métodos? Conceitualmente, ao menos, pode-se organizar qualquer programa em uma rotina gigante com controle em todos os pontos. Embora antigamente os programas fossem estruturados assim (com eventuais reversões recentes), um grande amontoado de lógica representa problemas. O mais sério deles é ser difícil de ler. Com uma rotina gigante, é difícil distinguir as partes importantes das menos importantes. É difícil entender parte do programa e deixar alguns detalhes para mais tarde. É difícil separar o que é importante para invocadores de alguma funcionalidade daquilo que é relevante para quem precisa modificar essa funcionalidade. O segundo problema é que a maioria dos problemas encontrados durante a programação não é única. Em vez de implementar tudo a partir do zero, é conveniente (e produtivo) simplesmente invocar uma solução anterior. A rotina gigantesca não fornece formas convenientes de se referir a partes para reúso posterior.

Dividir a lógica de um programa em métodos permite que se diga: “Estes pedaços de lógica não estão intimamente conectados”. Dividir os métodos em classes, e as classes em pacotes, possibilita a comunicação. Colocar este código neste método e aquele código naquele método sinaliza aos leitores que os dois pedaços de código não estão intimamente relacionados. Eles podem ser lidos e entendidos separadamente. Além disso, a nomeação dos métodos possibilita comunicar ao leitor qual é o propósito desse pedaço de computação, a despeito de sua implementação. Leitores podem frequentemente obter o que precisam somente a partir da leitura dos nomes dos métodos.

Métodos também resolvem bem o problema de reúso. Quando se escreve uma nova rotina e precisa-se de um pedaço de lógica que já existe como método, pode-se invocar aquele método.

Dividir uma grande computação em métodos é conceitualmente simples: coloque próximos os pedaços que vão juntos e separe os pedaços que vão separados. Na prática, entretanto, gastam-se tempo, energia e criatividade descobrindo previamente o que vai junto e o que não vai e, depois, pensando sobre a

melhor maneira de fazer a divisão. O que no momento parece ser uma boa divisão pode não funcionar quando se muda posteriormente a lógica do sistema. As divisões precisam simplificar a carga de trabalho global. Saber qual divisão funciona melhor é algo que vem com a experiência. Eis alguns conselhos que adquiri com o tempo.

Alguns problemas comuns em se dividir um programa em métodos são o tamanho, o propósito e a nomeação dos métodos. Ao criarem-se vários métodos muito pequenos, os leitores têm dificuldade em seguir a expressão fragmentada de ideias. Pouquíssimos métodos levam a duplicação e, consequente, perda de flexibilidade. Existem diversas tarefas clichês em programação, e criar um novo método é um passo comum no cumprimento de muitas dessas tarefas. Métodos que resolvem um desses problemas recorrentes costumam ser fáceis de nomear, mas aqueles que resolvem problemas únicos são mais difíceis, embora importantes para os leitores.

Eis os padrões relacionados a métodos:

- Método composto (composed method) – escrevem-se métodos compostos por chamadas a outros métodos.
- Nome revelador de intenção (intention-revealing name) – nomeiam-se métodos conforme aquilo que se destinam a fazer.
- Visibilidade de método (method visibility) – torna-se um método o mais privado possível.
- Objeto método (method object) – transformam-se métodos complexos em seus próprios objetos.
- Método sobreescrito (overridden method) – sobreescrivem-se métodos para expressar a especialização.
- Método sobrecarregado (overloaded method) – fornecem-se interfaces alternativas para a mesma computação.
- Tipo de retorno de método (method return type) – declara-se o tipo de retorno mais geral possível.
- Comentário de método (method comment) – comentam-se métodos para comunicar informações, e não para facilitar a leitura do código.
- Método auxiliar (helper method) – criam-se pequenos métodos privados para expressar mais sucintamente a computação principal.
- Método de impressão para depuração (debug print method) – usa-se `toString()` para imprimir informação útil de depuração.
- Conversão (conversion) – expressa-se a conversão limpa de um tipo de objeto em outro.
- Método de conversão (conversion method) – para conversões simples e limitadas, fornece-se um método no objeto fonte que retorna o objeto convertido.

- Construtor de conversão (conversion constructor) – para a maioria das conversões, fornece-se um método na classe do objeto convertido que pega o objeto fonte como um parâmetro.
- Criação (creation) – expressa-se de forma clara a criação de um objeto.
- Construtor completo (complete constructor) – escrevem-se construtores que retornam objetos completamente formados.
- Método fábrica (factory method) – expressam-se criações mais complexas como um método estático em uma classe, e não um construtor.
- Fábrica interna (internal factory) – encapsula-se em um método auxiliar a criação de um objeto que pode precisar de esclarecimento ou refinamento posterior.
- Método acessador de coleção (collection accessor method) – fornecem-se métodos que permitem acesso limitado a coleções.
- Método de atribuição booleana (boolean setting method) – caso ajude na comunicação, fornecem-se dois métodos para atribuir valores booleanos, um para cada estado.
- Método de consulta (query method) – retornam-se valores booleanos com métodos chamados de `asXXX`.
- Método de igualdade (equality method) – definem-se juntos `equals()` e `hashCode()`.
- Método get (getting method) – fornece-se ocasionalmente acesso a campos com um método que retorna aquele campo.
- Método set (setting method) – ainda que menos frequentemente, fornece-se a habilidade de atribuir valores a campos com um método.
- Cópia de segurança (safe copy) – evitam-se erros de sinônimos ao copiar objetos passados para dentro ou para fora de métodos acessadores.

Método composto

Escreva métodos compostos de chamadas a outros métodos, cada qual aproximadamente no mesmo nível de abstração.

Um dos sinais de um método pobramente composto é uma mistura de níveis de abstração:

```
void compute() {  
    input();  
    flags |= 0x0080;  
    output();  
}
```

Códigos como esse parecem dissonantes ao leitor. É mais fácil entender um código que flui, e mudanças abruptas nos níveis de abstração quebram esse

fluxo. “O que é aquela coisa enrolada ali?” é a pergunta que vem à cabeça quando se lê o método exemplificado há pouco. O que ele significa?

Uma objeção ao uso de muitos métodos pequenos é a perda de desempenho imposta pela invocação de todos eles. Enquanto escrevia este livro, codifiquei um pequeno programa de benchmark que comparava um laço de um milhão de iterações com um milhão de mensagens. A diferença foi de 20 a 30% em média – o que não é suficiente para afetar o desempenho da maioria dos programas. A combinação de CPUs mais rápidas e a natureza fortemente localizada de gargalos de desempenho torna o desempenho de um código uma questão que deve ser deixada de lado até que se tenha estatísticas a partir de conjuntos de dados realistas.

Qual deve ser o tamanho de um método? Algumas pessoas recomendam limites numéricos: menos de uma página ou entre 5 e 15 linhas. Embora seja verdade que a maioria dos códigos legíveis satisfaça esse limite, limitar leva à questão “por quê?”. Por que pedaços de lógica funcionam melhor quando têm aproximadamente esse tamanho?

Os leitores de código precisam resolver muitos problemas que geram influências opostas no tamanho dos métodos. Ao ler a estrutura geral, vale a pena ver montes de código de uma só vez. O espaço em branco no método dá indicações sobre a estrutura geral e a complexidade do código. Existem condicionais e laços? Com que profundidade estão aninhadas as estruturas de controle? Quanto trabalho é preciso para realizar a tarefa sugerida pelo nome do método?

O mesmo método grande que permitiu que eu me orientasse torna-se um obstáculo quando volto para tentar entender detalhes do código. Só consigo memorizar um detalhe por vez, e um método de mil linhas contém muitos detalhes. Para entender os detalhes, quero detalhes intimamente relacionados reunidos e isolados daqueles irrelevantes.

Permitir navegação e revisão ao mesmo tempo é o desafio do autor de código que divide a lógica em métodos. Acho que meu código é mais bem lido quando o quebro em métodos relativamente pequenos (ao menos pelos padrões C). O truque é reconhecer quando se tem conjuntos de detalhes relativamente independentes que possam ser submetidos a métodos de suporte. Às vezes tenho muitos detalhes para permitir uma fácil compreensão, mas que não são fáceis de particionar. Neste caso, crio um objeto método para dar a todos os detalhes um lugar em que fiquem organizados.

Outra questão ao escolher o tamanho do método é a especialização. Métodos corretamente dimensionados podem ser completamente sobrescritos, sem a necessidade de copiar o código para a subclasse e editá-lo, tampouco de sobrescrever dois métodos para uma única mudança conceitual.

Escreva métodos com base em fatos, não em especulação. Coloque seu código em funcionamento e, então, decida como ele deve ser estruturado. Quando se gasta muito tempo estruturando o código de antemão, ao descobrir algo durante a implementação, será preciso desfazer todo aquele trabalho e refazê-lo. Quando tenho todos os detalhes de uma lógica, é muito mais fácil compor métodos de forma sensata. Às vezes acho que sei como os métodos deveriam

ser compostos, mas, quando divido a lógica, o resultado parece difícil de ler. Nesses casos, é útil alinhar todos os métodos até voltar a ter um método gigantesco e então reparticioná-lo com base na experiência recente.

Nome revelador de intenção

Métodos devem receber um nome de acordo com o propósito que um potencial invocador poderia ter em mente para usá-los. Há outros pedaços de informação interessantes de se transmitir pelo nome – estratégia de implementação, por exemplo. Entretanto, comunique a intenção no nome, e outras informações sobre o método, de outras formas.

A estratégia de implementação é a informação estranha mais frequentemente incluída nos nomes dos métodos. Por exemplo:

```
Customer.linearCustomerSearch(String id)
```

Isso pode parecer superior a:

```
Customer.find(String id)
```

pois dá mais informações sobre o método. Contudo, o objetivo de um autor de códigos não é simplesmente contar tudo o que puder sobre o programa o mais rápido que conseguir. Às vezes é necessário se conter. A menos que a estratégia de implementação seja relevante aos usuários, deixe-a fora do nome. Os curiosos podem olhar no corpo do método para ver como ele é implementado. Ainda que `Customer` ofereça buscas lineares e por hash, seria melhor comunicar, por meio dos nomes, a distinção pela perspectiva do invocador:

```
Customer.find(String id)  
Customer.fastFind(String id)
```

(Na realidade, nessa situação talvez fosse preferível apenas oferecer um `find()` que satisfizesse todos os usuários, mas isso não vem ao caso.) Não importa muito aos usuários do método se a versão rápida de `find()` é implementada com uma tabela hash ou com uma árvore.

Pense em nomes de métodos baseando-se em como vão aparecer no código que os chama. É lá que os leitores provavelmente verão o nome pela primeira vez. Por que foi invocado este método, e não outro? Essa é uma questão que pode ser proveitosamente respondida pelo nome do método. O método de chamada deve contar uma história. Nomeie métodos para que ajudem a contar a história.

Quando você está implementando métodos por analogia com uma interface existente, dê a seus métodos os mesmos nomes usados na interface. Quando houver um tipo especial de iterador, chame seus métodos de `hasNext()` e `next()`, mesmo que não implemente de maneira formal a interface `Iterator`. Quando os métodos são apenas um pouco similares, pense se está sendo usada a metáfora certa e, depois, expresse as diferenças como prefixos no nome do método.

Visibilidade de método

Os quatro níveis de visibilidade – pública, pacote, protegida, privada – dizem algo diferente sobre as intenções de um método.

As duas grandes restrições opostas na visibilidade de métodos são a necessidade de ter alguma funcionalidade revelada para os usuários externos e a necessidade de manter flexibilidade no futuro. Quanto mais revelados são os métodos, mais difícil fica mudar a interface para um objeto, se necessário. Em desenvolvimento com JUnit, Erich Gamma e eu frequentemente discordamos sobre a visibilidade de métodos. Meu conhecimento em Smalltalk sugere que tornar visíveis os métodos é potencialmente valioso para os clientes. A experiência de Erich em Eclipse ensinou-lhe a valorizar a flexibilidade decorrente de se revelar o mínimo possível. Aos poucos, estou concordando com seu ponto de vista.

Há dois prejuízos quando se prefere visibilidade. Um é a perda de flexibilidade no futuro. Uma interface muito limitada torna mais fácil realizar mudanças. O outro prejuízo é a invocação do objeto. Uma interface restrita demais faz todos os clientes realizarem mais trabalho que o necessário para usar um objeto. Controlar esses prejuízos é fundamental para tomar boas decisões de visibilidade.

Minha estratégia geral em relação à visibilidade é limitá-la o quanto puder. Se essa fosse a única questão no que se refere a visibilidade, seria fácil. Uma ferramenta poderia atribuir visibilidade a métodos. O desafio real acontece quando não se está mais trabalhando com conhecimentos sobre os quais se tem certeza, quando um código fora de seu controle direto começa a invocar os métodos. Aí é preciso especular, decidir quais métodos serão públicos ou protegidos, o que obriga o programador a fazer a manutenção deles ou pagar um custo substancial para mudá-los.

- **Público:** quem declara público um método diz que acredita que ele é útil fora do pacote em que é declarado. Tornar público um método significa que se aceita a responsabilidade de fazer a manutenção dele, seja não o alterando, seja arrumando todos os chamadores caso o altere, ou, ao menos, notificando os programadores que o chamam.

```
public Object next();
```

Esta declaração diz que, agora e no futuro próximo, `next()` estará disponível aos clientes.

- **Pacote:** a visibilidade pacote é uma declaração de que o método é útil para objetos no pacote, mas que não se está disposto a disponibilizá-lo a objetos externos. Trata-se de um tipo de declaração esquisito – outros objetos precisam desse método, mas nem todos os outros objetos; apenas os meus. Trate a visibilidade pacote como uma sugestão de que a funcionalidade deve ser movida para que o método seja menos visível ou talvez o método seja muito mais útil do que se suspeitava e pode valer a pena torná-lo público.

- **Protegido:** a visibilidade protegida é apenas útil para se fornecer um código a ser reutilizado em subclasses. Embora pareça mais restritiva que a visibilidade pacote, as duas, na verdade, são ortogonais, uma vez que subclasses fora do pacote podem ver e invocar métodos protegidos.
- **Privado:** métodos privados propiciam o máximo de flexibilidade no futuro, já que garantem que se conseguirá encontrar e mudar todos os chamadores, não importando se intrusos usem e estendam o código ou não. Ao tornar privado um método, está-se dizendo que o valor desse método a quem está de fora não compensa torná-lo muito mais disponível.

Revele os métodos aos poucos, começando com uma visibilidade mais limitada e os revelando conforme necessário. Se um método não precisa mais ser tão visível, reduza sua visibilidade. Fazer isso apenas funciona quando se tem acesso a todos os chamadores do código, de forma que se tenha certeza de que não está atrapalhando um cliente ao eliminar a visualização de um método do qual ele dependia. Muitas vezes descubro que métodos que inicialmente pensei em fazer privados tornam-se, quando começo a usar um objeto de novas maneiras, membros valiosos de uma interface.

Declarar métodos como finais é semelhante a escolher sua visibilidade. Fazer isso indica que, embora o programador não se importe que as pessoas usem o método, ele não permitirá que ninguém o altere. Se as invariantes mantidas pelo método forem bem complicadas e sutis, esse nível de autoproteção pode ser justificado. Paga-se pela garantia de que ninguém accidentalmente quebrará seu objeto, eliminando-se a possibilidade de alguém proveitosamente sobrescrevê-lo, tendo, em vez disso, mais trabalho para realizar o serviço de outra forma. Eu, particularmente, não uso final, e às vezes fico frustrado ao encontrar métodos finais quando tenho uma razão legítima para sobrescrever um método.

Declarar um método como static torna-o visível mesmo que o chamador não tenha acesso a uma instância da classe (sujeito a modificação por outras palavras-chave de visibilidade). Métodos estáticos são limitados por não poderem depender de qualquer estado de instância, de forma que não são um bom repositório para lógicas complexas. Métodos estáticos podem ser reaproveitados, mas, uma vez sobrescritos, o método da superclasse não pode ser invocado. Um bom uso de métodos estáticos é feito na substituição de construtores.

Objeto método

Esse é um dos meus padrões favoritos, provavelmente porque o uso muito pouco, mas os resultados são espetaculares quando o faço. Criar um objeto método pode ajudá-lo a transformar uma massa emaranhada de código empacotada em um método impossível em um código legível e claro que revela detalhes aos poucos. É um padrão que aplico depois que parte do código esteja funcionando; quanto mais completo, melhor é o método.

Para criar um objeto método, procure por um método longo com muitos parâmetros e variáveis temporárias. Tentar extrair qualquer parte do método

resultaria em longas listas de parâmetros com submétodos difíceis de nomear. Eis os passos para criar um objeto método (pois essa refatoração não é automaticamente suportada no momento em que escrevo este livro):

1. Crie uma classe com um nome que lembre o método. Por exemplo, `complexCalculation()` torna-se `ComplexCalculator`.
2. Crie, na nova classe, um campo para cada parâmetro, variável local e campo usado no método. Dê a esses campos os mesmos nomes que tinham no método original (é possível arrumar os nomes depois).
3. Crie um construtor que receba como parâmetros os parâmetros de método do método original e os campos do objeto original usado pelo método.
4. Copie o método para um novo método, `calculate()`, na nova classe. Os parâmetros, as variáveis locais e os campos usados no método antigo tornam-se referências a campos no novo objeto.
5. Substitua o corpo do método original pelo código que cria uma instância da nova classe e que invoca `calculate()`. Por exemplo:

```
complexCalculation() {  
    new ComplexCalculator().calculate();  
}
```

6. Se os campos receberam atribuições no método original, atribua-as depois que `calculate()` retornar:

```
complexCalculation() {  
    ComplexCalculator calculator= new ComplexCalculator();  
    calculator.calculate();  
    mean= calculator.mean;  
    variance= calculator.variance;  
}
```

Assegure-se de que o código refatorado funciona como o código antigo. Agora começa a diversão. O código na nova classe é fácil de refatorar. É possível extrair métodos e nunca ter que passar quaisquer parâmetros, pois todos os dados usados pelo método estão armazenados nos campos. Frequentemente, ao se começar a extrair métodos, descobre-se que algumas variáveis podem ser rebaixadas de campos para variáveis locais. Da mesma maneira, há informações que podem ser passadas para um só método como parâmetro, não sendo armazenadas em um campo. Quando se começa a extrair métodos, pode-se descobrir que subexpressões comuns que eram difíceis de isolar antes tornaram-se métodos auxiliares úteis, com nomes significativos.

Às vezes, quando desconfio que um objeto método pode ser chamado, o método original já foi desmembrado. Neste caso, alinhe todos os submétodos, de forma que fiquem em um só lugar antes de começar. Uma indicação clara de que é preciso alinhar mais antes de fazer o método objeto é a necessidade de chamar métodos no objeto original. Retorne, junte-os e comece de novo.

Método sobrescrito

Uma das belezas de se programar com objetos é a variedade de formas pelas quais se pode expressar as diferenças entre cálculos similares. Métodos sobrescritos são uma maneira clara de expressar uma variação. Métodos declarados como abstratos em uma superclasse são um convite claro para especializar um cálculo, mas qualquer método não declarado como final poderia servir para expressar a variação de um cálculo existente. Métodos bem compostos na superclasse fornecem uma infinidade de potenciais ganchos em que se pode amarrar o código. Se o código da superclasse está em pedaços pequenos e coesos, então é possível sobreescrivê-los inteiramente.

Sobreescrivendo um método não é uma escolha rígida: ou isto ou aquilo. É possível executar os códigos da subclasse e da superclasse invocando `super.method();`. Apenas faça isso para invocar o método de mesmo nome. Se a subclasse explicitamente opta, às vezes, por invocar seu próprio código e, outras, por invocar o código da superclasse para uma variedade de métodos, será difícil acompanhar a classe, e é fácil quebrá-la acidentalmente. Se houver necessidade de invocar diferentes métodos de superclasse, pode-se melhorar o código reestruturando o fluxo de controle até que não seja mais preciso ir e voltar entre subclasse e superclasse.

Métodos de superclasse grandes demais criam um dilema: copiar o código para a subclasse e editá-lo, ou encontrar outro jeito de expressar a variação? O problema de copiar é que alguém pode vir depois e mudar o código que foi copiado da superclasse, quebrando o código sem que o programador (ou ele) saiba.

Método sobreescrito

Quando se declara o mesmo método com tipos diferentes de parâmetros, está-se dizendo: “Eis os formatos alternativos para os parâmetros desse método”. Um exemplo é um método que possa receber uma `String` representando um nome de arquivo ou um `OutputStream`, dando uma interface simples aos usuários que querem falar em termos de nomes de arquivos, mas preservando a flexibilidade para aqueles que preferem passar em um fluxo [`stream`] já formado (para teste, por exemplo). Métodos sobreescritos, se houver diversas maneiras legítimas de passar os parâmetros, tiram do chamador a responsabilidade de converter os parâmetros.

Uma variante de sobreescrita é usar o mesmo nome de método com diferentes quantidades de parâmetros. O problema desse estilo é que os leitores que quiserem perguntar “O que acontece quando invoco este método?” precisam ler não apenas o nome do método, mas também a lista de parâmetros, a fim de saberem o suficiente para descobrir o resultado da invocação do método. Se a sobreescrita é complicada, leitores precisam entender regras sutis de resolução de sobreescrita para conseguirem determinar estaticamente qual método será invocado para certos tipos de argumentos.

Todos os métodos sobreescritos devem servir para o mesmo propósito, variando apenas no tipo de parâmetros. Tipos de retorno diversos para

diferentes métodos sobrecarregados dificultam muito a leitura do código. É melhor encontrar um novo nome para a nova intenção. Dê nomes diferentes a diferentes computações.

Tipo de retorno de método

O tipo de retorno de um método indica se o método é um procedimento que funciona por efeito colateral ou uma função que retorna um certo tipo de objeto. O tipo de retorno mágico `void` permite que Java evite uma palavra-chave para distinguir entre procedimentos e funções.

Caso você esteja escrevendo uma função, escolha um tipo de retorno que expresse sua intenção. Às vezes se quer que o tipo de retorno seja específico: uma classe concreta ou um dos tipos primitivos. Contudo, todo programador gostaria que seu método fosse o mais amplamente aplicável possível; assim, escolha o mais abstrato tipo de retorno que expresse sua intenção. Isso preserva a flexibilidade para que se altere o tipo de retorno concreto, caso se torne necessário no futuro.

Generalizar o tipo de retorno também pode ser uma maneira de se esconder detalhes de implementação. Por exemplo, retornar uma `Collection` em vez de uma `List` pode estimular usuários a não assumirem que os elementos estão em uma ordem fixa.

Tipos de retorno são uma área comum para mudanças, conforme evolui o programa. Pode-se começar retornando uma classe concreta e, depois, descobrir que muitos métodos relacionados retornam diferentes classes concretas, cada uma compartilhando ou não a mesma interface. Expressar essa similaridade declarando a interface comum (se necessário) e retornando a nova interface em todos os métodos ajudará os leitores a entenderem a similaridade.

Comentário de método

Expresse a maior quantidade possível de informação por meio dos nomes e da estrutura do código. Adicione comentários para expressar informações que não estejam óbvias no código. Quando necessário, adicione comentários javadoc para explicar o propósito de métodos e classes.

Muitos comentários são completamente redundantes em um código escrito para comunicar. O custo de escrevê-los e mantê-los coerentes com o código não é compensado pelo valor que agregam. Comentários de métodos estão em um nível inútil de abstração. Se houver restrições entre dois métodos (por exemplo, um deve ser chamado antes de outro), onde fica o comentário? Comentários devem ser atualizados separadamente do código, e não há feedback imediato quando eles deixam de ser válidos.

Testes automatizados conseguem transmitir informações que não se encaixam naturalmente nos comentários de métodos. No exemplo anterior, posso

escrever um teste que garante que a exceção adequada seja lançada caso os métodos sejam invocados na ordem errada (embora eu preferisse eliminar ou encapsular essa restrição). Testes automatizados têm muitas vantagens. Escrevê-los é um exercício valioso de projeto, especialmente quando feito antes da implementação. Se os testes funcionam, eles são consistentes com o código. Ferramentas automatizadas de refatoração podem ajudar a manter os testes atualizados a um custo baixo.

Dito isso, a comunicação é ainda o principal valor desses padrões de implementação. Se um comentário de método é o melhor meio possível para comunicar, escreva um bom comentário.

Método auxiliar

Métodos auxiliares são uma consequência de métodos compostos. Quando se vai dividir grandes métodos em diversos métodos menores, é preciso ter aqueles métodos menores, que são os auxiliares. Seu propósito é tornar mais legíveis computações em larga escala, escondendo detalhes temporariamente irrelevantes e permitindo que se expresse sua intenção por meio do nome do método. Métodos auxiliares costumam ser declarados como privados ou protegidos caso se pretenda que a classe seja refinada por uma subclasse.

Pode-se criar um método como auxiliar privado e então descobrir que usuários externos desejam invocá-lo. Se o método é útil internamente, há uma chance de ele ser útil externamente também. Mesmo que seu pequeno auxiliar nunca “se forme”, continua sendo um valioso ponto de comunicação.

Os métodos auxiliares tendem a ser curtos, mas podem ser curtos demais. Hoje mesmo, eliminei um auxiliar que retornava apenas um construtor de classe. Descobri que:

```
return testClass.getConstructor().newInstance();
```

comunica tão bem quanto:

```
return getTestConstructor().newInstance();
```

Entretanto, esse método auxiliar ainda seria justificado se subclasses estivessem prevalecendo sobre a computação do construtor.

Elimine auxiliares (ao menos temporariamente) quando a lógica de um método se tornar pouco clara. Alinhe todos os métodos auxiliares, observe com cuidado a lógica e extraia novamente métodos que fazem sentido.

Outro propósito para métodos auxiliares é eliminar subexpressões comuns. Caso chame um método auxiliar em todos os lugares de uma classe que precisem de um pequeno cálculo, mudar essa expressão é fácil. Se as mesmas uma, duas ou três linhas são duplicadas em todo o objeto, perde-se a oportunidade de comunicar seu propósito por meio de um nome bem escolhido de método e, além disso, torna-se difícil mudar.

Método de impressão para depuração

Há muitas razões potenciais para transformar um objeto em um string. Pode-se querer apresentar o objeto a um usuário, armazenar o objeto para recuperação futura ou apresentar a parte interna do objeto a um programador.

A interface Object é bem restrita, contendo onze métodos. Um desses métodos, `toString()`, transforma o recebedor em um string, mas com qual propósito? Um atrativo é satisfazer vários propósitos de uma vez. Entretanto, esses compromissos raramente funcionam. É diferente o que um corretor de títulos, um programador e um banco de dados querem saber sobre um objeto.

Existem vantagens em investir em impressão de alta qualidade para depuração. Descobrir um detalhe interno importante sobre um objeto pode exigir meio minuto de cliques de mouse. Transforme esse detalhe com `toString()`, e a mesma informação estará disponível em um segundo, com um só clique. Prefiro gastar meu tempo avaliando o programa durante a depuração a ficar navegando pelos objetos no ambiente de desenvolvimento. Em sessões intensas de depuração, manter o foco pode economizar minutos ou horas de esforço.

Por ser público, `toString()` está sujeito a abusos. Se um objeto não aceita o protocolo necessário, sabe-se que as pessoas analisam o string mostrado para obter informações úteis. Códigos assim são frágeis, pois são comuns mudanças em `toString()`. A melhor política para prevenir esse abuso é fazer seu melhor para garantir que os objetos tenham todos os protocolos de que os clientes precisam.

Por esse motivo, quando precisar transcrever um objeto para um formato amigável ao programador, use bastante `toString()`. Transcreva outros métodos como string no objeto ou em classes separadas.

Conversão

Às vezes temos um objeto A e precisamos que o objeto B passe para algum cálculo futuro. Mas como representar essa conversão de um objeto fonte em um objeto destino?

Como com todos os padrões, o objetivo dos padrões de conversão é comunicar claramente a intenção do programador. Entretanto, há alguns fatores técnicos que influenciam na escolha da forma mais eficaz de expressar conversão. Um deles é o número de conversões necessárias. Se um objeto precisa apenas ser convertido em outro objeto, utiliza-se uma abordagem simples. Uma quantidade potencialmente ilimitada de conversões requer uma abordagem diferente. Outra questão a considerar são as dependências entre classes. Não vale a pena introduzir uma nova dependência somente para ter uma expressão conveniente de conversão.

Implementar a conversão é uma questão completamente diferente. Algumas vezes se cria um objeto real do novo tipo copiando informações do objeto fonte; outras vezes se pode implementar a interface do objeto destino sem copiar informações da fonte. Como alternativa a uma conversão, ocasionalmente se pode apenas encontrar uma interface comum para ambos os objetos e codificar nesse sentido.

Método de conversão

Quando se precisa expressar conversão entre objetos de tipo similar e há um número limitado de conversões, representa-se a conversão como um método no objeto fonte.

Por exemplo, suponha que você quer implementar coordenadas cartesianas e polares. Para criar um método de conversão, você implementaria:

```
class Polar {  
    Cartesian asCartesian() {  
        ...  
    }  
}
```

e vice-versa. Perceba que o tipo de retorno do método de conversão é uma classe específica para o objeto destino. O objetivo da conversão é obter um objeto com protocolo diferente. Alternativamente, seria possível implementar `getX()` e `getY()` em `Polar`, declarar `Polar` e `Cartesian` como implementadores de `Point` e ignorar totalmente a conversão.

Métodos de conversão têm a vantagem de serem fáceis de ler, sendo muito populares (mais de uma centena de exemplos no Eclipse). Entretanto, para criar um e introduzi-lo, é preciso ser capaz de modificar o protocolo do objeto fonte. Eles também introduzem uma dependência do objeto fonte pelo objeto destino. Se não houver essa dependência, não vale a pena introduzi-la somente para a conveniência de um método de conversão. Por fim, métodos de conversão tornam-se pesados quando há um número ilimitado de potenciais conversões. É difícil ler uma classe com vinte diferentes `asThis()` e `asThat()`. Uma opção é mudar os clientes de forma que eles consigam manipular o objeto fonte em vez de solicitarem conversão.

Essas desvantagens levaram-me a usar métodos de conversão com moderação e apenas em situações nas quais estou convertendo para objetos de tipo similar. Em outras circunstâncias, uso construtores de conversão para expressar conversão.

Construtor de conversão

Um construtor de conversão toma o objeto fonte como parâmetro e retorna um objeto destino. Ele é útil ao converter um objeto fonte para muitos destinos, pois as conversões não se amontoam no objeto fonte.

Por exemplo, `File` possibilita que um construtor de conversão converte um `String` com o nome de um arquivo em um objeto adequado para leitura, escrita e exclusão. Embora fosse conveniente ter `String.asFile()`, não há fim na quantidade de conversões assim feitas; portanto, é melhor ter `File(String name)`, `URL(String spec)` e `StringReadStream(String contents)`. Caso contrário, `String` teria um número ilimitado de métodos de conversão.

Quando se precisa de liberdade para implementar uma conversão retornando algo diferente de uma classe concreta, o construtor de conversão pode

ser expresso como um método fábrica retornando um tipo mais geral (ou colocado em uma classe diferente daquela criada pelo método).

Criação

Antigamente (meio século atrás), programas eram grandes e indiferenciadas massas de código e dados. O controle podia fluir de qualquer lugar para qualquer lugar. Dados podiam ser acessados de qualquer lugar. Cálculos, o propósito original dos computadores, ocorriam à velocidade da luz (relativamente falando) e com precisão. Então as pessoas descobriram um fato estranho: programas eram escritos tanto para ser mudados quanto para ser executados. Controle para todos os lados, códigos que se modificavam sozinhos e dados acessados de qualquer lugar, tudo isso era ótimo para a execução, mas terrível caso se quisesse mudar o programa depois. E assim começou a longa e falha estrada para encontrar modelos de computação em que uma mudança *aqui* não causasse um problema imprevisto *lá*.

Programas menores costumam ser mais fáceis de modificar que programas maiores. Uma primeira estratégia para fazer programas mais fáceis de modificar foi dividir um grande computador rodando um programa grande em um monte de computadores menores (objetos) rodando pequenos programas. Objetos acomodam mudanças futuras ao fornecer um horizonte de eventos além do qual mudanças no programa têm custo baixo.

Essa subdivisão tem fins humanos: acomodar nossas mentes falíveis, mutáveis e inventivas, e não para o bem do computador. O computador roda do mesmo jeito, não importa se o código é uma grande e feia maçaroca ou uma trabalhada rede de objetos que se completam. Para leitores humanos, a criação de um objeto tem um significado: um estado harmoniza-se para apoiar uma computação cujos detalhes, por enquanto, são irrelevantes.

A criação de objetos exige um equilíbrio entre a necessidade de expressar clara e diretamente e a necessidade por flexibilidade. Os padrões de implementação relacionados a criação propiciam técnicas para expressar variações sobre o tema “faça de mim um objeto”.

Construtor completo

Objetos precisam de algumas informações antes de conseguirem computar. Comunique os pré-requisitos a potenciais usuários fornecendo construtores que retornem objetos preparados para computar. Se há múltiplas formas de inicializar um objeto, forneça múltiplos construtores, cada um retornando um objeto bem formado.

```
new Rectangle(0, 0, 50, 200);
```

Às vezes é mais fácil alcançar flexibilidade criando-se um objeto com um construtor sem nenhum argumento seguido de uma série de métodos de atribui-

ção. Contudo, isso não informa a combinação de parâmetros necessária para o funcionamento correto do objeto.

```
Rectangle box= new Rectangle();
box.setLeft(0);
box.setWidth(50);
box.setHeight(200);
box.setTop(0);
```

Conigo me virar sem alguns desses parâmetros? Não há como responder apenas lendo a interface. Se vejo um construtor com quatro argumentos, entretanto, sei que todos os quatro argumentos são necessários.

Construtores obrigam os clientes a ter uma classe concreta. Ao escrever o código invocando o construtor, o programador pode ficar satisfeito por querer usar a classe concreta. Mas, caso se queira tornar o código mais abstrato, deve-se introduzir um método fábrica. Mesmo que se tenha um método fábrica, é interessante fornecer um construtor completo abaixo dele, de forma que leitores curiosos consigam rapidamente entender quais parâmetros são necessários para criar um objeto.

Ao implementar um construtor completo, afunile todos os construtores em um só construtor mestre que faz toda a inicialização. Isso assegura que todos os construtores variantes criarião objetos que satisfaçam todas as invariantes necessárias para o funcionamento adequado e informa quais são as invariantes aos modificadores futuros da classe.

Método fábrica

Uma maneira alternativa de representar a criação de objetos é com um método estático sobre a classe. Esses métodos têm algumas vantagens sobre os construtores: podem retornar um tipo mais abstrato (uma subclasse ou a implementação de uma interface) e podem ser nomeados segundo sua intenção, não apenas sua classe. Contudo, métodos fábrica adicionam complexidade, de forma que deveriam ser usados quando suas vantagens são importantes, e não apenas como um passo natural.

Convertido para um método fábrica, o exemplo do Rectangle ficaria parecido com isto:

```
Rectangle.create(0, 0, 50, 200);
```

Caso se pretenda algo mais complexo que criar um objeto, como gravar objetos em uma cache ou criar uma subclasse que será escolhida em tempo de execução, o método fábrica é algo útil. Porém, como leitor, sempre fico curioso quando vejo um método fábrica. O que mais acontece ali além da criação de um objeto? Não quero desperdiçar o tempo dos meus leitores, portanto, se tudo o que está acontecendo é a criação básica do objeto, expresso-a como um construtor. Se algo mais acontece ao mesmo tempo, introduzo um método fábrica para indicar esse fato aos leitores curiosos.

Uma variante do método fábrica é reunir métodos fábrica relacionados como métodos de instância de um objeto fábrica especial. Isso é útil quando se tem muitas classes concretas que variam ao mesmo tempo. Por exemplo, cada sistema operacional poderia ter um objeto fábrica diferente para criar os objetos que fazem chamadas ao sistema operacional.

Fábrica interna

O que fazer quando a criação de um objeto auxiliar é privada, mas complexa ou sujeita a mudanças pelas subclasses? Faça um método que crie e retorne o novo objeto.

Fábricas internas são comuns em inicializações preguiçosas. O objetivo do método getter é declarar que a variável está sendo inicializada preguiçosamente:

```
getX() {  
    if (x == null)  
        x= ...;  
    return x;  
}
```

Isso é muita coisa para um só método informar. Se o cálculo de x for complicado, é interessante transferi-lo para uma fábrica interna:

```
getX() {  
    if (x == null)  
        x= computeX();  
    return x;  
}
```

Fábricas internas são também um convite ao refinamento de subclasses. Uma computação que usa os mesmos algoritmos em diferentes estruturas de dados pode ser expressa por meio de fábricas internas. Alternativamente, podem-se passar as estruturas de dados como parâmetros de um objeto auxiliar.

Método acessador de coleção

Suponha que você tem um objeto que contém uma coleção. Como fornecer acesso a essa coleção? A solução mais simples é fornecer um método que faça isto:

```
List<Book> getBooks() {  
    return books;  
}
```

Isso dá aos clientes máxima flexibilidade, mas cria diversos problemas. O estado interno que depende do conteúdo da coleção pode ser invalidado, sem que se perceba, caso se retorne a coleção inteira. Fornecer esse acesso para

todos os propósitos também deixa passar uma oportunidade de criar um protocolo rico e significativo para seus objetos.

Uma alternativa é envelopar a coleção em uma coleção imutável antes de retorná-la. Infelizmente, para o compilador, o envelope apenas finge ser uma coleção. Se alguém tenta modificar a coleção envelopada, é lançada uma exceção. Depurar esse erro, especialmente na produção de um código, custa caro.

```
List<Book> getBooks() {  
    return Collections.unmodifiableList(books);  
}
```

Em vez disso, ofereça métodos que forneçam acesso limitado e significativo às informações na coleção.

```
void addBook(Book arrival) {  
    books.add(arrival);  
}  
int bookCount() {  
    return books.size();  
}
```

Se os clientes precisarem iterar sobre os elementos da coleção, forneça um método que retorne um iterador:

```
Iterator getBooks() {  
    return books.iterator();  
}
```

Isso evita que clientes modifiquem a coleção, exceto aquela desagradável operação `remove()` no `Iterator`. Caso se queira garantir que clientes não mudem o conteúdo da coleção, retorne um iterador que lance uma exceção se um elemento for removido. É preciso frisar que apenas ser notificado do erro em tempo de execução é arriscado e potencialmente caro para depurar.

```
Iterator<Book> getBooks() {  
    final Iterator<Book> reader= books.iterator();  
    return new Iterator<Book>() {  
  
        public boolean hasNext() {  
            return reader.hasNext();  
        }  
  
        public Book next() {  
            return reader.next();  
        }  
  
        public void remove() {  
            throw new UnsupportedOperationException();  
        }  
    };  
}
```

Caso se esteja duplicando a maioria dos protocolos da coleção, é provável que haja um problema de projeto. Se o objeto agia mais para seus clientes, ele não precisava oferecer tanto acesso às suas entradas.

Método de atribuição booleana

Qual é a melhor maneira de oferecer um protocolo que atribua um estado booleano? A solução mais simples é um método de atribuição simples:

```
void setValid(boolean newState) {  
    ...  
}
```

Se os clientes precisam da flexibilidade desse acesso, esse estilo está bom. Entretanto, quando todas as chamadas ao método de atribuição são constantes true ou false, pode-se oferecer uma interface mais expressiva fornecendo dois métodos, um para cada valor booleano:

```
void valid() {...}  
void invalid() {...}
```

O código que usa essa interface é mais legível, e neste código é mais fácil descobrir estaticamente onde está atribuído o estado. Contudo, caso veja um código assim:

```
...  
if (...boolean expression...)  
    cache.valid();  
else  
    cache.invalid();
```

vá em frente e forneça `setValidity(boolean)` em seu lugar.

Método de consulta

Às vezes um objeto precisa tomar uma decisão baseando-se no estado de outro objeto. Isso não é ideal, porque o objeto em geral deve tomar decisões sozinho. Todavia, quando um objeto precisa oferecer critérios de decisão como parte de seu protocolo, faça isso com um método cujo nome seja prefixado com uma forma do verbo “ser” (“é” ou “era”) ou “ter”.

Se um objeto tem muita lógica que depende do estado de outro objeto, talvez a lógica esteja no lugar errado. Por exemplo, quando se lê um método assim:

```
if (widget.isVisible())  
    widget.doSomething();  
else  
    widget.doSomethingElse();
```

é provável que o elemento de interface (*widget*) seja um método perdido.

Tente mover a lógica e veja se fica mais legível. Às vezes essas mudanças violam suas preconcepções sobre a responsabilidade de cada objeto por partes da computação. Acreditar e agir de acordo com o que veem seus olhos costuma melhorar o projeto. O resultado é mais legível e, geralmente, mais útil que um cenário rígido traçado antes dos testes.

Método de igualdade

Quando dois objetos precisam ser comparados por igualdade – por exemplo, porque são usados como chaves em uma tabela hash –, mas suas identidades não importam, implemente `equals()` e `hashCode()`. Como dois objetos iguais devem ter o mesmo valor hash, compute o hash usando apenas dados usados na computação de igualdade.

Por exemplo, caso esteja escrevendo um software financeiro, pode haver instrumentos financeiros com números seriais. É possível que isso leve a um método de igualdade como este:

```
Instrument
public boolean equals(Object other) {
    if (!other instanceof Instrument)
        return false;
    Instrument instrument= (Instrument) other;
    return getSerialNumber().equals(instrument.getSerialNumber());
}
```

Observe a cláusula de guarda no início do método. Em teoria, dois objetos quaisquer podem ser comparados por igualdade, de forma que seu código deve estar preparado para essa eventualidade. Sabendo que comparações entre classes indicam um erro de programação, elimine a cláusula de guarda e permita o lançamento de uma `ClassCastException`. Ou então lance uma `IllegalArgumentException` dentro da cláusula de guarda.

Como o número serial é a única informação usada na comparação de igualdade, é o único dado que deve ser usado para computar o valor hash:

```
Instrument
public int hashCode() {
    return getSerialNumber.hashCode();
}
```

Observe que, para pequenos conjuntos de dados, 0 funciona bem como um código hash.

Vinte anos atrás, essa questão de igualdade parecia mais importante. Lembro-me de gastar um tempo considerável elaborando esquemas de igualdade. Uma tirinha que circulava na época mostrava duas pessoas sentadas ao balcão de uma lanchonete. Uma delas diz ao garçom: “Quero o que ele está comendo”. Ao ouvir isso, o garçom agarra o prato da outra pessoa e o coloca na frente da primeira.

Os métodos `equals()` e `hashCode()` de hoje são os vestígios que permanecem daquela preocupação com igualdade. Quem for usá-los precisa seguir as regras. Do contrário, ocorrerão defeitos estranhos, como colocar um objeto em uma coleção e não ser capaz de recuperá-lo logo depois.

Outra alternativa ao uso da igualdade é garantir que dois objetos imutáveis iguais sejam o mesmo objeto. Por exemplo, alocar `Instruments` em um método fábrica permite isto:

```
Instrument
static Instrument create(String serialNumber) {
    if (cache.containsKey(serialNumber))
        return cache.get(serialNumber);
    Instrument result= new Instrument(serialNumber);
    cache.put(serialNumber, result);
    return result;
}
```

Método get

Uma maneira de oferecer acesso ao estado de um objeto é fornecer um método que retorne esse estado. Por convenção, em Java, esses métodos são prefixados com “get”. Por exemplo,

```
int getX() {
    return x;
}
```

Essa convenção é uma forma de metadados. Tentei dar aos métodos `get` nomes a partir da variável que retornavam, mas logo desisti. Se leitores achavam mais fácil ler “`getX`” do que “`x`”, a despeito de minha opinião pessoal, era melhor escrever o que esperavam.

A maneira de escrever métodos `get` não é uma questão tão importante ou interessante quanto a necessidade de escrevê-los ou, ao menos, se é preciso torná-los visíveis. Seguindo-se o princípio de unir lógica e dados, a necessidade de métodos `get` com visibilidade pública – ou pacote – é uma indicação de que a lógica deveria estar em outro lugar. Em vez de escrever o método `get`, tente mover a lógica que usa os dados.

Há algumas exceções à minha aversão a métodos `get` visíveis. Uma é quando se tem um conjunto de algoritmos localizados em seus próprios objetos. Algoritmos precisam acessar dados e precisam de um método `get` para recebê-los. Outra exceção é quando se quer um método público, e isso só é implementado ao retornar o valor de um campo. Por fim, métodos `get` que serão invocados por ferramentas quase sempre terão de ser públicos.

Métodos `get` internos (privados ou protegidos) são úteis para implementar inicialização preguiçosa ou cache. Como ocorre com todas as abstrações adicionais, esses refinamentos devem ser postergados até serem necessários.

Método set

Se precisar de um método para atribuir o valor de um campo, nomeie-o a partir do nome do campo, prefixado com “set”. Por exemplo:

```
void setX(int newX) {  
    x = newX;  
}
```

Sou ainda mais relutante em tornar visíveis métodos set – estes são no-meados pela implementação, e não pela intenção. Se um pedaço útil de uma interface é mais bem implementado por atribuição de um campo, tudo bem, mas o nome do método deveria ser escrito da perspectiva do código cliente. É mais fácil compreender qual problema o cliente está resolvendo ao atribuir um valor e fornecer um método que aborde diretamente aquele problema.

Usar um método set como parte da interface deixa a implementação vazar:

```
paragraph.setJustification(Paragraph.CENTERED);
```

Nomear a interface de acordo com o propósito do método ajuda o método a falar:

```
paragraph.centered();
```

mesmo se implementação de centered() for um método set:

```
Paragraph:centered() {  
    setJustification(CENTERED);  
}
```

Métodos set usados internamente (privados ou protegidos) podem ser valiosos para atualizar informações dependentes. Por exemplo, pode ser necessário reexibir nosso parágrafo (paragraph) sempre que a justificação for alterada. Isso poderia ser implementado em um método set:

```
private void setJustification(...) {  
    ...  
    redisplay();  
}
```

Esse uso de um método set funciona como um simples mecanismo de restrição, garantindo que se *este* dado muda, *aquele* dado dependente *ali* muda para corresponder (neste caso, o conteúdo do parágrafo e a informação mostrada na tela).

Métodos set fragilizam o código. Um princípio é evitar ações a distância. Se o objeto A depende dos detalhes da representação interna do objeto B, uma mudança no código de B também exigirá mudança no código de A, não porque algo fundamental em A tenha mudado, mas apenas porque mudaram as suposições sobre as quais A foi escrito. É melhor mover a lógica e os dados

juntos. Talvez A devesse ser proprietário dos dados, ou B devesse oferecer um protocolo mais significativo.

Da mesma forma que com métodos get, quando se tem uma ferramenta que precisa invocar métodos set, marque-os “Apenas para uso da ferramenta” e torne-os públicos. Ofereça uma interface mais comunicativa e modular.

Cópia de segurança

Quando se usa um método get ou set, enfrentam-se potenciais problemas com sinônimos, já que dois objetos presumem ter acesso exclusivo a um terceiro. Problemas com sinônimos são um sintoma de problemas de projeto ainda maiores, tais como falta de clareza sobre qual objeto é responsável por quais dados. Contudo, é possível evitar alguns defeitos fazendo-se uma cópia do objeto antes de retorná-lo ou armazená-lo:

```
List<Book> getBooks() {  
    List<Book> result= new ArrayList<Book>();  
    result.addAll(books);  
    return result;  
}
```

Neste caso, talvez seja melhor fornecer métodos acessadores de coleção. Entretanto, se for necessário prover acesso à coleção inteira, trata-se de uma forma segura de fazê-lo.

Métodos set também podem ser escritos com cópias de segurança:

```
void setBooks(List<Book> newBooks) {  
    books= new ArrayList<Book>();  
    books.addAll(newBooks);  
}
```

Lembro-me de revisar um sistema bancário que abusava de cópias de segurança. Cada método acessador (get ou set) tinha duas versões: uma “de segurança”, e a outra não. Para eliminar defeitos de sinônimos, enormes estruturas de objetos eram copiadas sempre que era invocado um método de cópia de segurança. O sistema era lento demais, de forma que os clientes costumavam usar a versão não segura, resultando em inúmeros defeitos de sinônimos. Mas nunca se abordou o problema de projeto, ou seja, que os objetos não ofereciam protocolos suficientemente significativos.

A cópia de segurança é basicamente um paliativo usado para proteger o código de acessos externos sobre os quais não se tem controle. Raramente deveria ser parte da semântica principal de uma implementação. Objetos imutáveis e métodos compostos propiciam interfaces mais simples e comunicativas, menos passíveis de erros.

Conclusão

Este capítulo descreveu os padrões para a criação de métodos, finalizando os padrões relacionados à linguagem Java. O próximo capítulo descreve padrões para uso de classes de coleção.

CAPÍTULO 9

Coleções

Devo dizer que não esperava que esse capítulo chegasse a tanto. Quando comecei a escrevê-lo, pensei que terminaria com um documento de API – tipos e operações. A ideia básica é simples: uma coleção diferencia objetos que estão na coleção daqueles que não estão. Isso não basta?

O que descobri é que coleções são um tópico muito mais rico do que suspeitava, tanto em estrutura como nas possibilidades que oferecem para comunicar. O conceito de coleções mistura várias metáforas diferentes. A metáfora enfatizada muda o uso que se faz das coleções. Cada uma das interfaces da coleção comunica uma variação diferente sobre o tema amontoado de objetos. Cada uma das implementações também informa variações – a maioria a respeito de desempenho. O resultado é que conhecer bem coleções ajuda muito a comunicar bem o código.

Comportamentos similares aos de uma coleção costumavam ser implementados por vínculos na própria estrutura de dados: cada página de um documento teria vínculos com as páginas anteriores e seguintes. Recentemente, isso mudou, e passou-se a usar um objeto separado para a coleção que relaciona elementos, propiciando a flexibilidade de se colocar o mesmo objeto em várias coleções diferentes sem precisar modificá-lo.

Coleções são importantes por serem uma forma de expressar um dos mais fundamentais tipos de variação na programação: a variação de número. A variação de lógica é expressa com condicionais ou mensagens polimórficas. A variação na cardinalidade dos dados é expressa colocando-se os dados em uma coleção. Os detalhes precisos daquela coleção revelam muito a um leitor sobre a intenção do programador original.

Há um ditado antigo (em termos de computação) que diz que os únicos números interessantes são 0, 1 e muitos (esse ditado não foi criado por um analista numérico). Se a ausência de um campo expressa “zero” e a presença de um campo expressa “um”, então um campo contendo uma coleção é uma forma de expressar “muitos”.

Coleções pairam em um mundo estranho, que fica entre um construto de linguagem de programação e uma biblioteca. Coleções são tão úteis e seu uso

é tão bem entendido que parece ser o momento de se ter uma linguagem tradicional que permita declarações como plural `unique Book books;` em vez do atual `Collection<Book> books= new HashSet<Book>();`. Enquanto as coleções não se tornam elementos de primeira classe de uma linguagem, é importante saber como usar a biblioteca atual de coleção para expressar de forma objetiva ideias comuns.

O restante deste capítulo é dividido em seis partes: as metáforas por trás das coleções, as questões a serem expressas por meio de coleções, as interfaces de coleções e o que significam para o leitor, as implementações de coleções e o que dizem, um panorama das funções disponíveis nas classes `Collections` e, por fim, uma discussão sobre como estender coleções por meio de herança.

Metáforas

Como já foi mencionado, coleções misturam metáforas diferentes. A primeira é aquela de uma variável multivalorada. Há uma sensação de que uma variável que se refere a uma coleção na verdade se refere a diversos objetos ao mesmo tempo. Olhando dessa forma, a coleção desaparece como um objeto separado. A identidade da coleção não é interessante, apenas os objetos a que se refere são. Como ocorre com todas as variáveis, podem-se atribuir elementos a uma variável multivalorada (adicionar e remover), recuperar seu valor e enviar mensagens à variável (com o laço `for`).

A metáfora de variável multivalorada acaba em Java, pois as coleções são objetos separados com identidade própria. A segunda metáfora das coleções é a dos objetos – uma coleção é um objeto. Pode-se recuperar uma coleção, transferi-la, testá-la por igualdade e enviar mensagens a ela. Coleções podem ser compartilhadas entre objetos, embora isso crie a possibilidade de problemas de sinônimos. Como coleções são um conjunto de interfaces e implementações relacionadas, elas estão abertas a extensão, tanto com interfaces expandidas como com novas implementações. Dessa forma, assim como coleções “são” variáveis multivaloradas, elas também “são” objetos.

A combinação das duas metáforas leva a alguns efeitos estranhos. Em razão de a coleção ser implementada como um objeto que pode ser transferido, obtém-se o equivalente a uma chamada por referência, em que, em vez de passar o conteúdo de uma variável para uma rotina, passa-se a própria variável. Mudanças no valor da variável são refletidas na rotina chamadora. Chamadas por referência saíram de moda em linguagens há algumas décadas, por causa do risco de consequências não intencionais. Era difícil depurar programas quando não se tinha certeza de todos os lugares onde uma variável poderia ser modificada. Algumas das convenções de programação com coleções existem para evitar situações em que seja difícil ler o código e prever onde a coleção pode ser modificada.

Uma terceira metáfora útil para pensar em coleções é a de conjuntos matemáticos. Uma coleção é uma pilha de objetos, da mesma forma que um conjunto matemático é uma pilha de elementos. Um conjunto divide o mundo em

coisas do conjunto e coisas fora do conjunto. Uma coleção divide o mundo dos objetos em objetos que estão na coleção e objetos que não estão. Duas operações básicas com conjuntos matemáticos são encontrar sua cardinalidade (o método `size()` das coleções) e testar a inclusão (representado pelo método `contains()`).

A metáfora matemática é apenas aproximada para coleções. As outras operações básicas sobre conjuntos – união, intersecção, diferença e diferença simétrica – não estão diretamente representadas nas coleções. Um debate interessante é discutir se isso ocorre porque essas operações são intrinsecamente menos úteis ou se elas não são usadas por não estarem disponíveis.

Questões

Coleções são usadas para expressar muitos conceitos ortogonais em programas. Em princípio, é necessário expressar-se com a maior precisão possível. Com coleções, isso significa usar a interface mais geral possível como declaração e a classe de implementação mais específica. Entretanto, não se trata de uma regra absoluta. Analisei JUnit com cuidado e generalizei todas as declarações de variáveis. O resultado foi uma bagunça, pois não havia uniformidade. A confusão de se ter o mesmo objeto declarado como um `Iterable` em um lugar, uma `Collection` em outro e uma `List` em outro dificultou a leitura do código, sem muitas vantagens. Era mais fácil apenas declarar cada variável como uma `List`.

O primeiro conceito expresso por coleções é seu tamanho. Vetores* (que são coleções primitivas) têm um tamanho fixo, estabelecido quando são criados. A maioria das coleções pode mudar de tamanho depois de ser criada.

Um segundo conceito expresso pelas coleções é se a ordem dos elementos é importante ou não. Cálculos em que os elementos afetam uns aos outros ou em que usuários externos do cálculo atribuem importância à chamada de coleções requerem coleções que preservem a ordem. A ordem pode ser aquela em que foram adicionados elementos ou pode ser criada por alguma influência externa, como comparação lexicográfica.

Outra questão expressa por coleções é a unicidade dos elementos. Há computações em que a presença ou a ausência de um elemento é suficiente, e outras em que um elemento precisa ser capaz de estar presente múltiplas vezes em uma coleção para a comparação estar correta.

Como são acessados os elementos? Às vezes é suficiente iterar sobre os elementos, fazendo alguns cálculos com eles, um de cada vez. Em outros momentos, é importante poder armazenar e recuperar elementos com uma chave.

* N. de T.: Optou-se pelo termo vetor como tradução de *array*, pois a comunidade Java o usa mais do que o termo arranjo, utilizado em alguns textos sobre estruturas de dados.

Coluna: desempenho

Na maior parte do tempo, os programadores não têm que se preocupar com o desempenho de operações em pequena escala. Essa é uma boa mudança em relação ao que se fazia antigamente, quando ajuste de desempenho era uma ocupação diária. Entretanto, recursos computacionais não são infinitos. Quando a experiência mostra que o desempenho precisa ser melhor e as medições mostram onde estão os gargalos, é importante expressar claramente decisões relativas ao desempenho. Muitas vezes, um melhor desempenho resulta em inferioridade de alguma outra qualidade no código, como legibilidade ou flexibilidade. É importante que a necessidade de desempenho tenha a menor quantidade de consequências possível.

Codificar visando o desempenho pode violar o princípio de consequências locais. Uma pequena mudança em uma parte do programa pode degradar o desempenho em outra. Quando um método funciona eficientemente apenas se a coleção pode testar inclusão rapidamente, então uma substituição inocente de `ArrayList` para `HashSet` em um outro lugar no programa pode tornar o método intoleravelmente lento. Consequências distantes são outra questão a ser considerada quando se codifica visando desempenho.

O desempenho está conectado a coleções porque a maioria das coleções pode crescer sem limites. A estrutura de dados que contém os caracteres que se está escrevendo precisa ser capaz de suportar milhões de caracteres. É importante inserir o milionésimo caractere tão rápido quanto foi inserido o primeiro.

Minha estratégia geral para codificação visando o desempenho com coleções é usar a mais simples implementação possível no início e escolher uma classe de coleção mais especializada quando necessário. Quando tomo decisões relacionadas a desempenho, tento concentrá-las o máximo possível, mesmo que isso exija mudanças no projeto. Então, quando o desempenho está bom novamente, paro de ajustar.

Por fim, considerações de desempenho são comunicadas por meio da escolha da coleção. Se uma busca linear é rápida o bastante, uma `Collection` genérica é boa o suficiente. Se a coleção cresce demais, será importante estar apto a testar ou acessar elementos por uma chave, sugerindo um `Set` ou um `Map`. Tempo e espaço podem ser otimizados pela seleção sensata de coleções.

Interfaces

Leitores de códigos baseados em coleções procuram por respostas a diferentes questões quando analisam as interfaces declaradas para as variáveis e as implementações escolhidas para essas variáveis. A declaração da interface conta ao

leitor sobre a coleção: se a coleção está em uma ordem particular, se há elementos duplicados e se existe qualquer forma de procurar elementos por meio de uma chave ou apenas por iteração.

As interfaces descritas a seguir são:

- **Vetor (Array)** – vetores são a coleção mais simples e menos flexível: rápida, com tamanho fixo e sintaxe de acesso simples.
- **Iterativa (Iterable)** – a interface básica de coleção, permitindo que ela seja usada para iteração, mas nada mais.
- **Coleção (Collection)** – permite adicionar, remover e testar elementos.
- **Lista (List)** – uma coleção cujos elementos estão ordenados e podem ser acessados por sua localização na coleção (isto é, “dê-me o terceiro elemento”).
- **Conjunto (Set)** – uma coleção sem duplicatas.
- **Conjunto Ordenado (SortedSet)** – uma coleção ordenada sem duplicatas.
- **Mapa (Map)** – uma coleção cujos elementos são armazenados e recuperados por chave.

Array

Vetores são a interface mais simples para coleções. Infelizmente, não têm o mesmo protocolo de outras coleções, sendo, portanto, mais difícil mudar de um vetor para uma coleção do que de um tipo de coleção para outro. Diferentemente da maior parte das coleções, o tamanho de um vetor é fixado quando criado. Vetores são também diferentes por serem embutidos na linguagem, e não disponibilizados por uma biblioteca.

Vetores são mais eficientes em tempo e espaço do que outras coleções graças a suas operações simples. Os testes de tempo que fiz para acompanhar este livro sugerem que o acesso a um vetor (isto é, `elements[i]`) é mais de dez vezes mais rápido que a operação equivalente em `ArrayList` (`elements.get(i)`). (Como esses números variam substancialmente em diferentes ambientes operacionais, caso se importe com a diferença de desempenho, cronometre as operações por sua conta.) A flexibilidade das outras classes de coleção as torna mais valiosas na maioria dos casos, mas vetores são um artifício conveniente quando se precisa de mais desempenho em uma pequena parte do aplicativo.

Iterable

Declarar uma variável como `Iterable` apenas indica que ela contém múltiplos valores. `Iterable` é a base para o construto de laço em Java 5. Qualquer objeto declarado como `Iterable` pode ser usado em um laço `for`. Isso é implementado chamando-se discretamente o método `iterator()`.

Uma das questões a serem comunicadas quando se usam coleções é se os clientes irão modificá-las. Infelizmente, `Iterable` e seu auxiliar, `Iterator`, não

fornecem maneiras de se declarar que uma coleção não deve ser modificada. Quando se tem um Iterator, pode-se invocar o método `remove()`, que deleta um elemento do Iterable subjacente. Embora os Iterables estejam a salvo de terem elementos adicionados, eles podem ter elementos removidos sem que seja notificado o objeto proprietário da coleção.

Como descrito no “Método acessador de coleção” (Capítulo 8), há poucas maneiras de evitar que uma coleção seja modificada: envelopá-la em uma coleção não modificável, criar um iterador personalizado que lance uma exceção quando um cliente tenta modificá-la ou retornar uma cópia segura.

Iterable é simples. Ele nem mesmo permite que se meça o tamanho de instâncias; tudo o que se pode fazer é iterar sobre os elementos. Subinterfaces de Iterable têm comportamento mais útil.

Collection

`Collection` herda de `Iterable`, mas inclui métodos para adicionar, remover, buscar e contar elementos. Declarar uma variável ou um método como uma `Collection` cria muitas opções para uma classe de implementação. Ao deixar a declaração o mais vagamente especificada possível, conserva-se a liberdade de mudar as classes de implementação posteriormente, sem que a mudança se propague pelo código.

Coleções são um pouco como a noção matemática de conjuntos, com a diferença de que as operações que realizam o equivalente a união, intersecção e diferença (`addAll()`, `retainAll()` e `removeAll()`) modificam o recebedor em vez de retornar coleções recém-alocadas.

List

Para `Collection`, `List` inclui a ideia de elementos estarem em uma ordem estável. Um elemento pode ser recuperado fornecendo-se seu índice para a coleção. Uma sequência estável é importante quando os elementos de uma coleção integram. Por exemplo, uma fila de mensagens que precisariam ser processadas na ordem de chegada deveria ser armazenada em uma lista.

Set

Um `Set` é uma coleção que não contém duplicatas (elementos que informariam que são `equals()` um ao outro). Isso corresponde rigorosamente à noção matemática de conjunto, ainda que a metáfora seja limitada, pois adicionar um elemento a um `Set` modifica a coleção em vez de retornar uma nova coleção com o elemento adicionado.

Um `Set` descarta informações que a maioria das coleções mantém – o número de vezes que um elemento aparece. Isso não é um problema nos casos em que a presença ou a ausência de um elemento é interessante, e não tem relevância o número de vezes que o elemento aparece. Por exemplo, se quero saber

quais autores de livros estão em uma biblioteca, não me importa quantos livros cada autor escreveu. Apenas quero saber quem são eles. Um Set é uma maneira adequada de implementar essa consulta.

Os elementos em um Set não estão em uma ordem específica. Apenas porque, uma vez, se iterou por meio deles em determinada ordem, não significa que os elementos vão aparecer na mesma ordem da próxima vez. A falta de ordem previsível não é uma limitação nos casos em que os elementos não interagem.

As vezes se quer armazenar duplicatas em uma coleção, mas também removê-las para uma operação particular. Nesses casos, crie um Set temporário e o passe para a operação:

```
printAuthors(new HashSet<Author>(getAuthors()));
```

SortedSet

Os atributos de ordenação e unicidade das coleções não são mutuamente exclusivos. Há momentos em que se gostaria de manter uma coleção em ordem, mas eliminar as duplicatas. SortedSet armazena elementos ordenados mas únicos.

Diferentemente de uma List ordenada, que está relacionada à ordem em que os elementos foram adicionados ou aos índices explícitos passados em add(int, Objeto), a ordenação em um SortedSet é fornecida por um Comparator. Na ausência de uma ordem explícita, é usada a “ordem natural” dos elementos. Por exemplo, strings são ordenados de forma lexicográfica.

Para computar os autores que contribuem para uma biblioteca, pode-se usar um SortedSet:

```
public Collection<String> getAlphabeticalAuthors() {
    SortedSet<String> results= new TreeSet<String>();
    for (Book each: getBooks())
        results.add(each.getAuthor());
    return results;
}
```

Esse exemplo usa um ordenamento padrão de strings. Se um Book tem seu autor representado por um objeto, o código deveria parecer com isto:

```
public Collection<String> getAlphabeticalAuthors() {
    Comparator<Author> sorter= new Comparator<Author>() {
        public int compare(Author o1, Author o2) {
            if (o1.getLastName().equals(o2.getLastName()))
                return o1.getFirstName().compareTo(o2.getFirstName());
            return o1.getLastName().compareTo(o2.getLastName());
        }
    };
    SortedSet<Author> results= new TreeSet<Author>(sorter);
    for (Book each: getBooks())
        results.add(each.getAuthor());
    return results;
}
```

Map

A última interface de coleção é `Map`, um híbrido das outras interfaces. Um `Map` armazena valores por chaves, mas, diferentemente de uma `List`, a chave pode ser qualquer objeto, e não apenas um inteiro. As chaves de um `Map` devem ser únicas, como conjuntos, embora os valores possam conter duplicatas. Os elementos de um `Map` não estão em uma ordem particular, da mesma forma que em um `Set`.

Como `Map` não é totalmente igual a qualquer das outras interfaces de coleção, ele fica sozinho e não herda de nenhuma delas. Mapas são duas coleções ao mesmo tempo: uma coleção de chaves conectadas a uma coleção de valores. Não se pode simplesmente pedir a um `Map` seu iterador, pois não está claro se quer-se um iterador sobre as chaves, sobre os valores ou sobre os pares chaves-valores.

Mapas são úteis para implementar dois dos padrões de implementação: estado extrínseco e estado variável. O estado extrínseco sugere armazenar dados com propósito especial relacionados a um objeto separadamente do próprio objeto. Uma forma de implementar estado extrínseco é com um `Map` cujas chaves sejam os objetos e cujos valores sejam os dados relacionados. No estado variável, diferentes instâncias da mesma classe armazenam diferentes campos de dados. Para implementar isso, o objeto deve conter um mapa que mapeie a partir de strings (representando os nomes dos campos virtuais) para os valores.

Implementações

Escolher classes de implementação para coleções é principalmente uma questão de desempenho. Como ocorre com todas as questões de desempenho, é melhor escolher uma implementação simples para começar e então ajustá-la baseando-se nas experiências.

Nesta seção, cada interface introduz implementações alternativas (Fig. 9.1). Como considerações de desempenho dominam a escolha da classe de implementação, cada conjunto de alternativas é acompanhado por medidas de desempenho para operações importantes. O Apêndice deste livro fornece o código-fonte da ferramenta que usei para reunir esses dados.

De longe, a maioria das coleções é implementada por `ArrayList`, ficando `HashSet` em um distante segundo lugar (cerca de 3.400 referências a `ArrayList` no Eclipse+JDK contra cerca de 800 a `HashSet`). A solução rápida e descuidada é escolher qualquer dessas classes que sirva para suas necessidades. Entretanto, para as situações em que a experiência mostra que o desempenho é importante, o restante desta seção mostra os detalhes das implementações alternativas.

Um último fator para escolher uma classe de implementação de coleção é o tamanho das coleções envolvidas. Os dados apresentados a seguir mostram o desempenho de coleções com tamanho de um a cem mil. Se suas coleções apenas contêm um ou dois elementos, escolher a classe de implementação pode ser

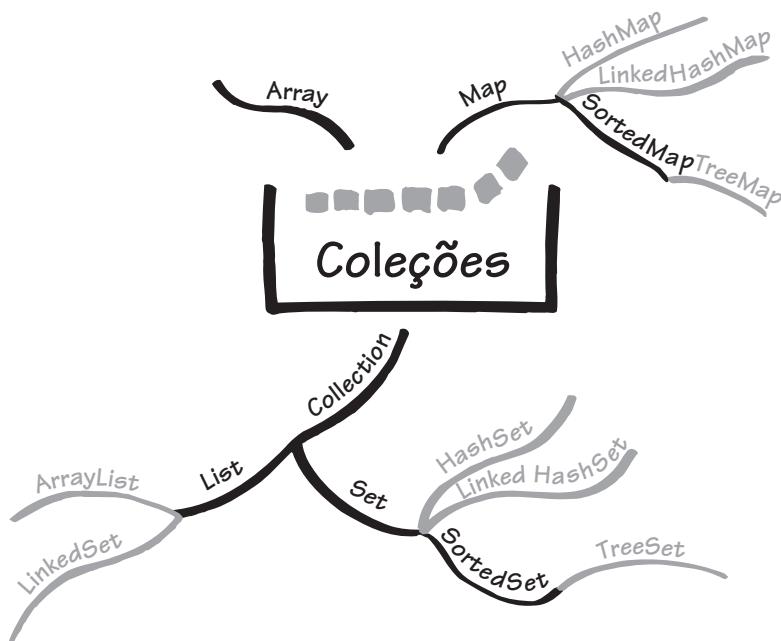


Figura 9.1 Interfaces e classes de coleção.

diferente de quando se espera que chegue a milhões de elementos. Neste caso, pouco se ganha ao mudar as classes de implementação, e será preciso procurar mudanças algorítmicas em grande escala para melhorar o desempenho.

Collection

A classe padrão na implementação de uma `Collection` é `ArrayList`. O potencial problema de desempenho com `ArrayList` é que `contains(Object)` e outras operações que dependem dela, como `remove(Object)`, gastam um tempo proporcional ao tamanho da coleção. Se o perfil de desempenho mostra que um desses métodos é um gargalo, considere substituir sua `ArrayList` por um `HashSet`. Antes de fazer isso, certifique-se de que o algoritmo não é sensível ao descarte de elementos duplicados. Quando se tem dados que, com certeza, não contêm duplicatas, a troca não fará diferença. A Figura 9.2 compara o desempenho de `ArrayList` e de `HashSet`. (Veja o Apêndice para detalhes de como obtive essa informação.)

List

`List` adiciona ao protocolo `Collection` a ideia de que os elementos estão em uma ordem estável. As duas implementações de `List` comumente usadas são `ArrayList` e `LinkedList`.

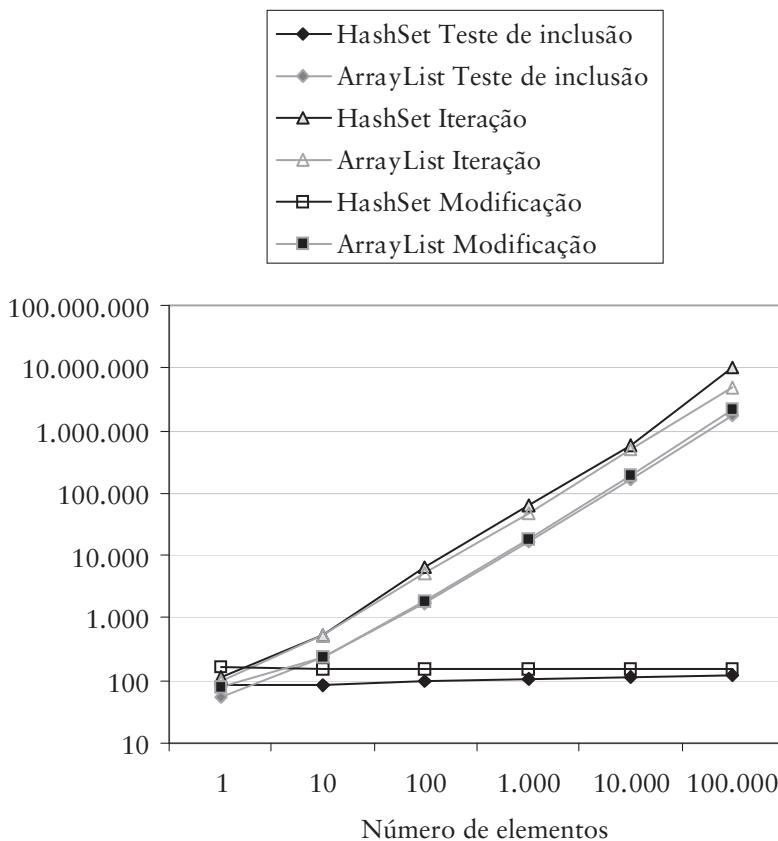


Figura 9.2 Comparação entre ArrayList e HashSet como implementações de Collection.

Os perfis de desempenho dessas duas implementações são imagens espelhadas. ArrayList é rápida para acessar elementos e lenta para adicionar ou remover elementos, enquanto LinkedList é lenta para acessar elementos e rápida para adicionar ou remover elementos (Figura 9.3). Se você tiver um perfil dominado por chamadas como `add()` ou `remove()`, considere trocar uma ArrayList por uma LinkedList.

Set

Há três implementações principais de Set: HashSet, LinkedHashSet e TreeSet (que, na verdade, implementa SortedSet). HashSet é a mais rápida, mas não se tem certeza de que seus elementos estejam em ordem. Uma LinkedHashSet mantém os elementos na ordem em que foram adicionados, mas com 30% de perda a mais de tempo para adição e remoção de elementos (veja Figura 9.4). TreeSet mantém seus elementos ordenados de acordo com um Comparator, mas adicionar ou remover elementos e testar um elemento levam um tempo proporcional a $\log n$, em que n é o tamanho da coleção.

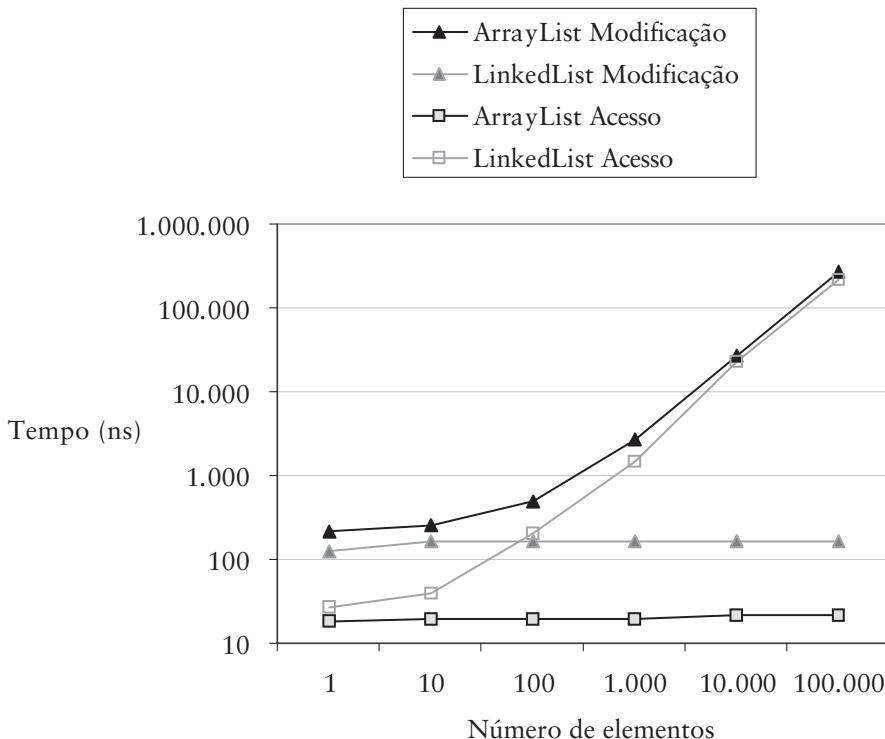


Figura 9.3 Comparação entre ArrayList e LinkedList.

Escolha uma LinkedHashSet se precisar ordenar os elementos para ficarem estáveis. Usuários externos, por exemplo, podem gostar de obter elementos sempre na mesma ordem.

Map

As implementações de Map seguem um padrão similar ao de Set. HashMap é mais rápido e simples. LinkedHashMap preserva a ordem dos elementos, iterando sobre eles na ordem em que foram inseridos. TreeMap (na verdade, uma implementação de SortedMap) itera sobre entradas com base na ordem das chaves, mas a inserção e os testes de inclusão levam um tempo proporcional a $\log n$. A Figura 9.5 resume a forma como o desempenho é melhorado com essas implementações de Map.

Collections

A classe utilitária Collections é uma classe biblioteca que fornece funcionalidades de coleção que não se encaixam bem em nenhuma das interfaces de coleção. A seguir é apresentada uma visão geral do que está disponível.

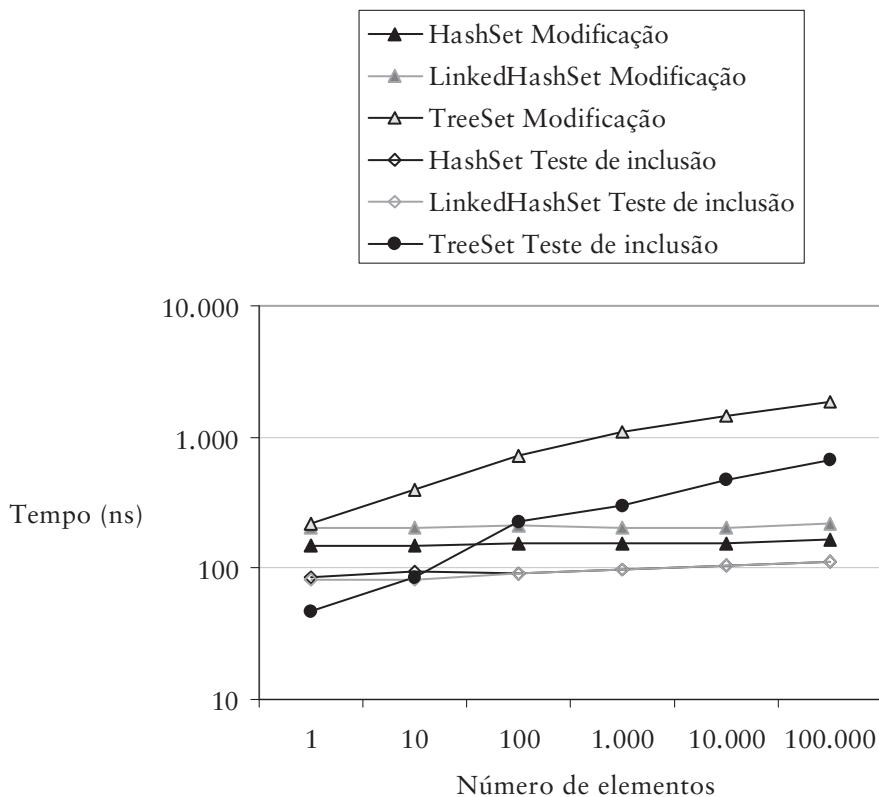


Figura 9.4 Comparação entre implementações de Set.

Busca

A operação `indexOf()` leva um tempo proporcional ao tamanho da lista. Contudo, se os elementos estão classificados, a pesquisa binária pode encontrar o índice de um elemento em tempo $\log_2 n$. Chame `Collections.binarySearch(lista, elemento)` para retornar o índice de um elemento na lista. Se o elemento não aparecer na lista, será retornado um número negativo. Se a lista não estiver ordenada, os resultados são imprevisíveis.

A pesquisa binária apenas melhora o desempenho em listas com acesso aleatório de tempo constante, como `ArrayList` (veja Figura 9.6).

Ordenação

`Collections` também permitem que operações mudem a ordem dos elementos de uma lista. `Reverse(list)` inverte a ordem de todos os elementos da lista. `Shuffle(list)` coloca os elementos em ordem aleatória. `Sort(list)` e `sort(list, comparator)` colocam os elementos em ordem ascendente. Diferentemente da pesquisa binária, o desempenho de ordenação é mais ou menos o mesmo para `ArrayList` e `LinkedList`, pois os elementos são primeiramente copiados em

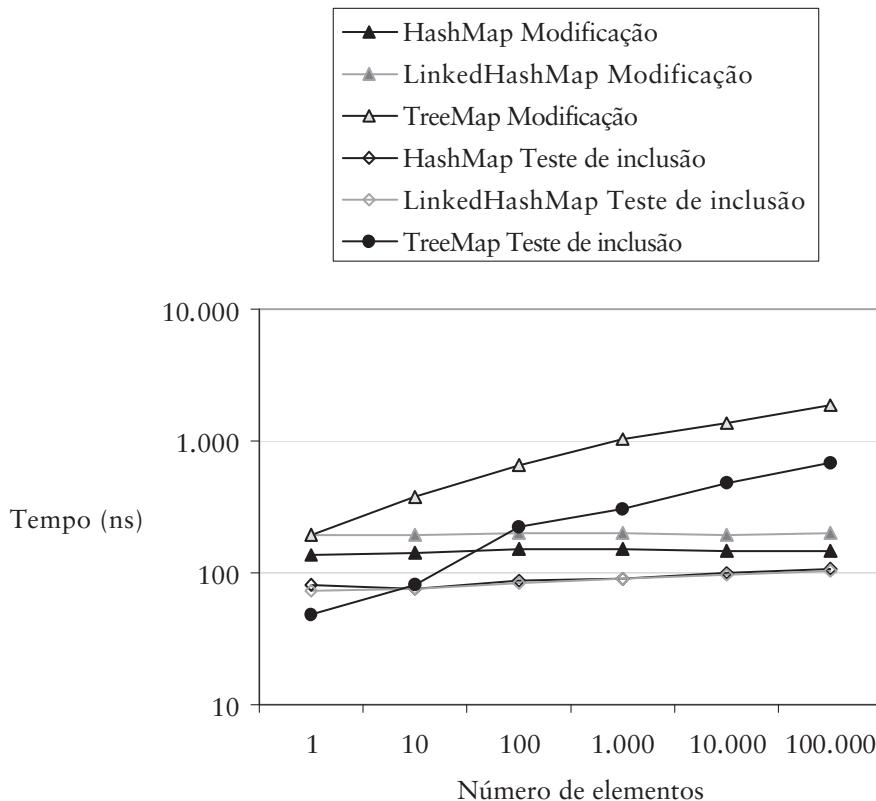


Figura 9.5 Comparação entre implementações de Map.

um vetor, o vetor é classificado e, então, os elementos são copiados de volta (execute o teste de tempo Sorting no Apêndice para verificar isso).

Coleções imutáveis

Como dito na discussão sobre Iterable, mesmo a mais básica interface de coleção permite que ela seja modificada. Se estiver passando uma coleção para um código não confiável, é possível evitar que ele seja modificado tendo Collections envelopada em uma implementação que lance uma exceção em tempo de execução se o cliente tentar modificá-la. Há variantes que funcionam com Collection, List, Set e Map.

```
@Test(expected=UnsupportedOperationException.class)
public void unmodifiableCollectionsThrowExceptions() {
    List<String> l= new ArrayList<String>();
    l.add("a");
    Collection<String> unmodifiable= Collections.unmodifiableCollection(l);
    Iterator<String> all= unmodifiable.iterator();
    all.next();
    all.remove();
}
```

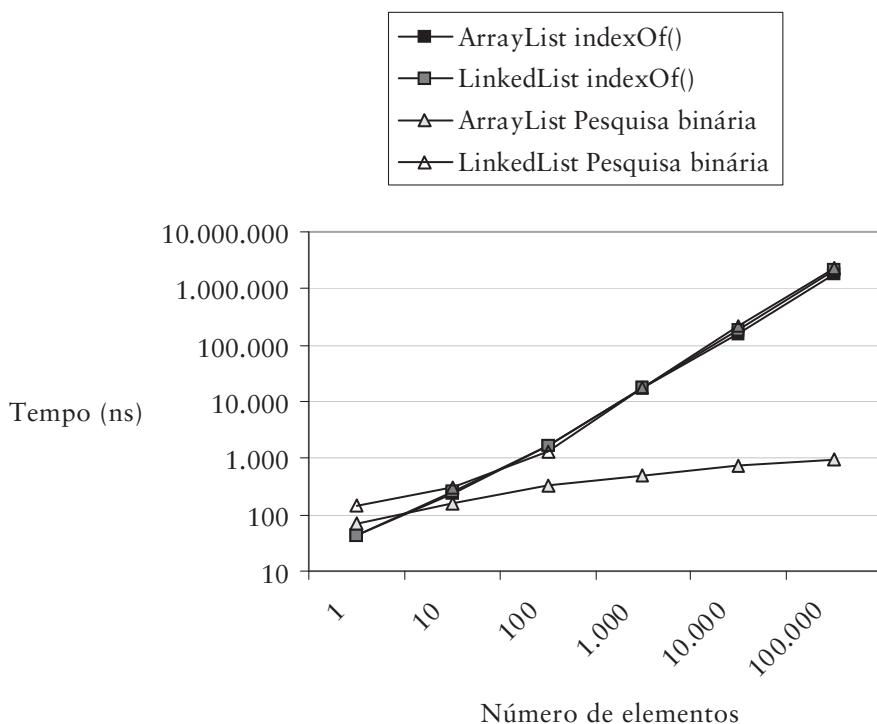


Figura 9.6 Comparação entre `indexOf()` e pesquisa binária.

Coleções com apenas um elemento

Se você tiver um só elemento e precisar passá-lo a uma interface que espera uma coleção, poderá rapidamente convertê-lo chamando `Collections.singleton()`, que retorna um Set. Há também variantes que convertem para uma List ou um Map. Nada que as coleções retornam é modificável.

```
@Test public void exampleOfSingletonCollections() {
    Set<String> longWay= new HashSet<String>();
    longWay.add("a");
    Set<String> shortWay= Collections.singleton("a");
    assertEquals(shortWay, longWay);
}
```

Coleções vazias

Da mesma forma, se você precisar usar uma interface que espera uma coleção e souber que não há elementos para isso, `Collections` criará uma coleção vazia imutável.

```
@Test public void exampleOfEmptyCollection() {
    assertTrue(Collections.emptyList().isEmpty());
}
```

Estendendo coleções

Frequentemente tenho visto classes que estendem uma das classes de coleção. Uma Library [biblioteca] contendo uma lista de livros, por exemplo, poderia ser implementada estendendo-se `ArrayList`:

```
class Library extends ArrayList {...}
```

Essa declaração propicia implementações de `add()` e `remove()`, iteração e outras operações de coleção.

Há muitos problemas em se estender classes de coleção para obter comportamento de coleção. Primeiramente, muitas das operações oferecidas por coleções serão inapropriadas para clientes. Por exemplo, os clientes em geral não devem poder usar `clear()` [limpar] em uma Library ou convertê-la a um vetor, com `toArray()`. No mínimo, as metáforas ficam misturadas e confusas. E, pior, todas essas operações precisam ser deserdadas, implementando-as e lançando uma `UnsupportedOperationException`. Não é interessante herdar algumas poucas linhas de código úteis e gastar ainda mais linhas eliminando funcionalidade que não se quer. O segundo problema der herdar de classes de coleção é que se desperdiça herança, um recurso precioso. Ao aproveitar umas poucas linhas de implementação útil, impede-se o uso da herança de uma forma altamente importante.

Nessa situação, é melhor delegar a uma coleção em vez de herdar a estrutura de uma:

```
class Library {  
    Collection<Book> books= new ArrayList<Book>();  
    ...  
}
```

Com esse projeto, pode-se revelar apenas aquelas operações que fazem sentido e dar-lhes nomes significativos. Fica-se livre para usar a herança para compartilhar implementação com outras classes modelo. Se uma Library oferece acesso a livros por meio de diferentes chaves, podem-se nomear as operações adequadamente:

```
Book getBookByISBN(ISBN);  
Book getBookByID(UniqueID);
```

Apenas estenda coleções se estiver implementando uma classe de coleção com fins gerais, algo que poderia ser adicionado a `java.util`. Em todos os outros casos, armazene elementos em uma coleção secundária.

Conclusão

Este capítulo descreveu os padrões para o uso de classes de coleção. Todos os padrões anteriores foram escritos com um viés para o desenvolvimento de

aplicativo em que simplicidade e facilidade de comunicação levavam à redução de custos, embora fosse possível mudar o projeto de todo o aplicativo de uma só vez. O próximo capítulo descreve como modificar esses padrões ao criar frameworks, nos quais a complexidade é aceitável caso preserve a capacidade de continuar evoluindo o framework, mas sem permitir que se mude todo o código do aplicativo.

CAPÍTULO 10

Evoluindo frameworks

Os padrões de implementação apresentados nos capítulos anteriores presumem que mudar código é mais barato que entender e comunicar sua intenção. Isso valeu em boa parte de minha experiência de desenvolvimento. Entretanto, o desenvolvimento de framework, em que o código cliente não pode ser mudado pelos desenvolvedores do framework, viola essa hipótese. Mudar o projeto de JUnit, por exemplo, é geralmente fácil, mas sua implementação pode ser muito cara se os criadores dessa ferramenta e aqueles que escrevem testes tiverem de mudar seus códigos também. Atualizações incompatíveis são tão caras que devemos evitá-las sempre que possível.

Quando lançamos JUnit 4, gastamos praticamente metade do orçamento de engenharia para reduzir os custos de implantação para nossos clientes. Tentamos assegurar que os novos testes funcionassem com as ferramentas antigas e que os testes antigos funcionassem com ferramentas novas. Também trabalhamos para garantir que tivéssemos liberdade para fazer mudanças futuras no JUnit sem quebrar o código do cliente.

Este capítulo esboça como mudam os padrões de implementação ao desenvolvermos frameworks. Trata dos desafios do desenvolvimento de framework e fala sobre como reduzir o impacto de atualizações incompatíveis e sobre como projetar frameworks que evitem atualizações incompatíveis. Evoluir frameworks enquanto minimiza-se a interrupção para os clientes requer maior complexidade, reduzindo-se as funcionalidades visíveis ao cliente e havendo uma comunicação cuidadosa sobre as mudanças necessárias.

Mudando frameworks sem mudar aplicações

O principal dilema no desenvolvimento e na manutenção de frameworks é que eles precisam evoluir, mas há um grande custo para se quebrar o código cliente. A atualização de framework perfeita adiciona nova funcionalidade sem alterar funcionalidades existentes. Atualizações compatíveis, contudo, nem sempre

são possíveis. Manter retrocompatibilidade* costuma adicionar complexidade ao framework. Em algum momento, o custo de manter uma compatibilidade perfeita supera o valor para os clientes. Melhorar a economia do desenvolvimento de frameworks depende de reduzir a probabilidade de uma atualização incompatível e de diminuir o custo das atualizações necessárias. Enquanto no desenvolvimento convencional reduzir a complexidade ao mínimo é uma estratégia poderosa para tornar o código fácil de entender, no desenvolvimento de frameworks, em geral, há uma melhor relação custo-benefício em adicionar complexidade de forma a aumentar a habilidade do desenvolvedor para melhorar o framework sem quebrar o código cliente.

Apesar de a compatibilidade ter uma importância maior no desenvolvimento de frameworks, a simplicidade continua sendo um valor importante. Frameworks complexos têm menos chances de serem usados que aqueles mais simples. Adicione a menor quantidade possível de complexidade de forma a manter o equilíbrio entre liberdade futura para desenvolvimento e custo para clientes.

Uma tendência nos padrões de implementação apresentados nos capítulos anteriores era o código ser o mais amplamente aplicável, mas sem perder legibilidade. No desenvolvimento de frameworks, a aplicabilidade é sacrificada pelo ganho de liberdade para alterar o projeto no futuro. Por exemplo, costumo tornar campos protegidos na maior parte do código, mas, quando desenvolvo frameworks, deixo-os privados. Isso dificulta que clientes usem minhas superclasses, mas me permite mudar as representações de dados do framework sem afetar as aplicações dos clientes. Um framework com todos os campos protegidos seria mais imediatamente utilizável, mas mais difícil de evoluir no futuro.

O objetivo são frameworks complexos o suficiente para evoluir, mas simples de serem usados, e estritamente aplicáveis para evoluir, mas amplamente aplicáveis para serem úteis. Essas restrições adicionais no projeto tornam o desenvolvimento de frameworks mais arriscado e mais caro que o desenvolvimento de aplicações. Felizmente, variantes dos padrões de implementação podem ajudá-lo a construir, implantar e modificar frameworks úteis e passíveis de mudança.

Atualizações incompatíveis

Mesmo que uma atualização de framework possa quebrar o código do cliente, há formas de reduzir o custo da atualização para os clientes. Organizar a atualização em pequenas etapas alerta os clientes sobre o que está por vir e os deixa decidir quando realizar o investimento de alterar o código. Por exemplo, depreciar o código mas deixá-lo funcionando por um ou mais lançamentos (*releases*) informa que os clientes precisam mudar para a nova API. A depreciação é um

* N. de T.: Compatibilidade com o que existia.

exemplo de estratégia mais generalista de manter duas arquiteturas diferentes para resolver o mesmo problema. As arquiteturas paralelas adicionam complexidade, mas reduzem a quebra causada pela atualização.

As classes de coleção Java demonstram arquiteturas paralelas. As antigas classes `Vector` e `Enumerator` foram feitas visando compatibilidade futura quando fossem introduzidas novas classes enraizadas em `Collection`. Agora (e para sempre, no caso de Java), pode-se executar código usando as coleções antigas.

Pacotes podem ser uma forma de oferecer aos clientes acesso incremental a atualizações. Ao introduzir novas classes em um novo pacote, consegue-se dar a elas os mesmos nomes das classes antigas. Por exemplo, se eu puder atualizar `org.junit.Assert` para `org.junit.novoemelhorado.Assert`, os clientes precisarão apenas alterar as cláusulas de importação para usarem a nova classe. Mudar importações é menos arriscado e intrusivo do que mudar código.

Outra estratégia incremental é mudar a API ou a implementação, mas não ambas, no mesmo lançamento. O lançamento intermediário, seja com a nova interface para o código antigo ou com a antiga interface para o código novo, dá a todos – fornecedores do framework e clientes – tempo para se acostumarem com a mudança. Há tempo para resolver quaisquer problemas técnicos causados pela nova abordagem enquanto os problemas ainda são pequenos.

As classes de coleção trazem outro ponto sobre a atualização de frameworks: aposentar funcionalidades obsoletas. Parte do acordo entre o fornecedor de um framework e seu cliente é a frequência com que o código do cliente será forçado a se atualizar para funcionar com um novo lançamento de framework. O compromisso dos mantenedores de Java* é que um código antigo funcionará perpetuamente. O Eclipse, por sua vez, concorda em manter a compatibilidade apenas dentro de lançamentos inteiros. Como fornecedor de framework, é preciso equilibrar com cuidado a necessidade de evoluir rapidamente o framework com a necessidade de seus clientes trabalharem com uma plataforma estável enquanto se escolhe uma estratégia de aposentadoria de funcionalidades.

O Eclipse fornece um exemplo de como reduzir o custo de atualizações incompatíveis: oferecer ferramentas automáticas para atualizar o código cliente. Eclipse reduziu o custo da atualização possivelmente incompatível da versão 2.x para a 3.0, assegurando que a maioria dos plug-ins funcionaria inalterada na versão 3.0, mas também oferecendo uma ferramenta de conversão para deixar os plug-ins 2.x em total conformidade com 3.0. A ferramenta adicionava arquivos necessários e movia a funcionalidade entre os arquivos de forma que o código antigo funcionava nativamente na nova versão. Ao combinar estratégias, o Eclipse mantém grande parte da liberdade para melhorar seu framework rapidamente evolutivo, mas ainda servindo os clientes existentes com funcionalidades, na maior parte, estáveis.

Pode-se reduzir o custo da alteração de código se os clientes puderem mudar para a funcionalidade atualizada com uma operação simples de localizar/ substituir. Quando o nome de um método muda, é mais barato para os clientes

* N. de T.: Na época em que o livro foi escrito, a linguagem Java e seu contexto (máquina virtual, compilador, etc.) eram gerenciados e mantidos pela Sun. Atualmente, é pela Oracle.

que os argumentos permaneçam na mesma ordem. Talvez algum dia seja possível propagar conjuntos de refatorações junto com atualizações de frameworks, mas por enquanto reduzir o custo de atualizações pode limitar as opções de projeto.

Outro fator no gerenciamento de atualizações incompatíveis é a composição e o crescimento da comunidade de clientes. Se os clientes atuais estão ansiosos por utilizar a mais recente funcionalidade, eles estarão dispostos a realizar a atualização. Se uma atualização permite que se aumente dramaticamente a base de clientes, então deve-se estar preparado para aturar resmungos de alguns clientes existentes. As reclamações de 400 clientes hoje não parecerão tão significativas se em seis meses houver 4.000 clientes felizes. Mas cuidado para não trocar clientes reais por fantasmas, ficando com um framework atualizado, porém sem clientes.

Esta seção detalhou como gerenciar atualizações incompatíveis em frameworks. De longe, a mais desejável situação é uma atualização que introduza nova funcionalidade sem afetar o código cliente existente. O restante deste capítulo discute os padrões de implementação para escrever frameworks que podem ser atualizados sem incomodar os clientes.

Encorajando mudanças compatíveis

Para que a atualização de um framework mantenha a compatibilidade, o código cliente deve depender da menor quantidade possível de detalhes de framework. Entretanto, o código cliente precisa depender de alguns detalhes; caso contrário, não há razão para o framework existir. Idealmente, clientes dependerão apenas de detalhes que não se quer mudar. Como crescimento e mudanças são imprevisíveis, não se pode decidir antecipadamente quais detalhes não se quer mudar. Pode-se, entretanto, jogar com probabilidades: reduzir o número de detalhes visíveis e revelar detalhes menos suscetíveis a mudanças, entregando funcionalidade útil enquanto se conserva a liberdade para mudar o projeto.

Uma decisão a tomar é definir quais variantes de compatibilidade serão oferecidas. A atualização é retrocompatível, de forma que os clientes ainda conseguem invocar métodos antigos e passar objetos antigos para o framework? A atualização é compatível com futuras mudanças, de forma que se possa passar objetos no novo estilo para clientes e eles funcionarem como objetos antigos? O estilo ou os estilos de compatibilidade que se escolhe afetam o esforço necessário para desenvolver e testar uma atualização. Nossa mais recente lançamento de JUnit, por exemplo, foi muito mais caro, pois escolhemos oferecer compatibilidade passada e futura. Usuários relataram muitos defeitos de compatibilidade que não consideramos enquanto estávamos codificando. Estou satisfeito com nossa decisão de compatibilidade. Tínhamos uma base enorme de testes para dar suporte e clientes que, na maior parte, não se apressaram em fazer a atualização. Entretanto, dar suporte à compatibilidade passada e futura teve consequências surpreendentes.

A maioria dos frameworks em Java é representada como objetos que são criados, usados ou refinados por clientes. Esta seção descreve como representar frameworks para que os clientes consigam usar a funcionalidade de que precisam e, ao mesmo tempo, o desenvolvedor possa continuar evoluindo o framework. Atingir esse equilíbrio requer atenção cuidadosa em relação a como são usados e criados os objetos e a como são estruturados os métodos.

Classe biblioteca

Um estilo de API simples e razoavelmente à prova de futuro é a classe biblioteca. Se for possível representar todas as funcionalidades como chamadas a procedimentos com parâmetros simples, os clientes estarão bem protegidos de mudanças futuras. Quando se libera uma nova versão de uma classe biblioteca, é preciso apenas garantir que todos os métodos existentes funcionem da mesma forma que antes. Novas funcionalidades são representadas como novos procedimentos ou novas variantes dos procedimentos existentes.

A classe `Collections` é um exemplo de API representada como uma classe biblioteca. Clientes usam essa classe invocando métodos estáticos, e não a instanciando. Novas versões das classes de coleção adicionam novos métodos estáticos, deixando inalterada a funcionalidade existente.

O grande problema em representar uma API como classe biblioteca é o número limitado de conceitos e variantes que podem ser facilmente expressos. Conforme proliferaram variações de funcionalidade e produtos de variações, é fácil estourar o número de procedimentos necessários. Além disso, clientes podem apenas mudar os dados que enviam para o framework, mas não conseguem conectar quaisquer variações de lógica.

Objetos

Presumindo-se que o framework será representado como objetos, tem-se a difícil tarefa de equilibrar simplicidade e complexidade, flexibilidade e especificidade, a fim de que o framework seja útil e estável para clientes e evolutível para o programador. O truque é, contanto que se possa gerenciá-lo, escrever o framework de forma que clientes dependam apenas dos detalhes menos suscetíveis a mudanças.

Discutirei quatro questões da representação de frameworks como objetos:

- Estilo de uso – clientes usarão o framework instanciando seus objetos, configurando seus objetos e/ou refinando ou implementando seus objetos?
- Abstração – detalhes de classe serão representados como interfaces ou como classes? Como a visibilidade será usada para revelar apenas detalhes relativamente estáveis?
- Criação – como os objetos serão criados?
- Métodos – como serão estruturados os métodos para que sejam úteis a clientes e propícios a mudança?

Estilo de uso

Frameworks podem suportar três principais estilos de uso: instanciação, configuração e implementação. Cada estilo oferece diferentes combinações de usabilidade, flexibilidade e estabilidade. Podem-se também misturar esses estilos em um só framework para oferecer um melhor equilíbrio entre liberdade de projeto para os desenvolvedores do framework e poder para os clientes.

O estilo mais simples de usar é a instanciação. Quando quero um socket, escrevo `new ServerSocket()`. Uma vez instanciado, um objeto de framework funciona invocando métodos. A instanciação funciona quando a única forma de variação de que os clientes precisam é a variação nos dados, e não na lógica.

A configuração é um estilo mais complexo e flexível. Nela, o cliente cria objetos de um framework, mas passa para eles seus próprios objetos que serão chamados em momentos predeterminados. Um `TreeSet`, por exemplo, pode ser chamado com um `Comparator` definido pelo cliente para permitir ordenação arbitrária de elementos.

```
Comparator<Author> byFirstName= new Comparator<Author>() {  
    public int compare(Author book1, Author book2) {  
        return book1.getFirstName().compareTo(book2.getFirstName());  
    }  
};  
SortedSet<Author> sorted= new TreeSet<Author>(byFirstName);
```

A configuração é mais flexível que a instanciação, pois pode acomodar variações na lógica e nos dados. Contudo, oferece menos liberdade ao projetista do framework, pois, quando se começa a chamar um objeto cliente, é preciso continuar a chamar aquele objeto da mesma forma e no mesmo tempo ou corre-se o risco de quebrar o código cliente. Outra limitação da configuração é que ela consegue manipular apenas algumas dimensões de variabilidade. Um objeto dado pode ter somente uma ou duas opções de configuração antes de se tornar complexo demais para fácil utilização.

Quando clientes precisam de mais maneiras de encaixar sua própria lógica do que as opções fornecidas pela configuração, então é preciso oferecer o uso por implementação. Na implementação, os clientes criam suas próprias classes a serem utilizadas pelo framework. Enquanto as classes do cliente estenderem uma classe do framework ou implementarem uma interface do framework (a escolha disso é abordada na próxima seção), o cliente está livre para incluir qualquer lógica que queira.

Dos três estilos de uso de objetos, a implementação tem maior potencial para restringir liberdade de projeto no futuro. Cada detalhe da superclasse ou interface fornecida pelo framework precisa ser preservado caso se queira garantir que o código cliente continuará funcionando. Cada detalhe revelado na abstração de um framework é uma faca de dois gumes: oferece ao cliente um lugar para enganchar seu código, mas compromete o desenvolvedor do framework a dar suporte ao detalhe ou correr o risco de quebrar o código do cliente.

O exemplo Comparator apresentado anteriormente demonstra uma versão simples do estilo de uso de frameworks por implementação. O comparador byFirstName [peloPrimeiroNome] é uma implementação da abstração do comparador do framework coleção (neste caso, uma classe). Neste caso, a implementação é simples, pois há apenas um pedaço de lógica a ser conectado, curto o suficiente para aparecer em uma linha com o restante do código. Implementações podem também residir em classes internas ou classes autônomas, se forem mais complexas.

O estilo de uso por implementação escala muito melhor que a configuração porque pode lidar com qualquer número de variações independentes, cada qual representada por um método gancho definido pelo framework.

JUnit mistura os quatro estilos de uso:

- JUnitCore é uma classe biblioteca com um método estático `run(Class...)` para rodar todos os testes em todas as classes.
- JUnitCore também é instanciável, e as instâncias fornecem um controle mais refinado sobre a execução e a notificação de testes.
- As anotações `@Test`, `@Before` e `@After` são uma forma de configuração em que escritores de testes podem identificar pedaços de código que devem ser executados em determinados momentos.
- A anotação `@RunWith` é uma forma de implementação em que escritores de testes que precisam de comportamentos não padrão para a execução de testes podem implementar seus próprios executores (*runners*).

Abstração

O estilo de uso de frameworks por implementação introduz a questão de representar entidades abstratas como uma interface ou como uma superclasse comum. As duas abordagens têm vantagens e desvantagens para desenvolvedores e clientes de frameworks e não são mutuamente exclusivas. Um framework pode oferecer aos clientes uma interface e uma implementação padrão daquela interface.

Interface

A grande vantagem de se oferecer aos clientes uma interface é que ela registra pouquíssimos detalhes. Clientes não conseguem, “acidentalmente”, usar mais do framework do que pretendiam. Contudo, essa proteção tem um custo. Enquanto as interfaces permanecem imutáveis, está tudo bem, mas introduzir um novo método em uma interface quebrará todas as implementações daquela interface nos clientes. Se for possível garantir que os clientes apenas usem a interface e não a implementem, todavia, podem-se introduzir novos métodos sem quebrar o código do cliente. Apesar da fragilidade das interfaces, elas são amplamente usadas no mundo Java para expressar abstrações, o que é por si um argumento a seu favor.

As interfaces têm uma vantagem menor: muitas classes dos clientes podem ser implementadas de uma só vez. Implementar muitas interfaces relacionadas em uma só classe pode ser uma forma clara e direta de informar. Entretanto, é provável que uma classe que implementa simultaneamente interfaces não relacionadas seja fragmentada de forma a comunicar seus propósitos com mais clareza.

Uma variação de interfaces que fornece flexibilidade adicional ao custo de certa complexidade são as versionadas. Quando se adicionam operações a uma interface, quebra-se o código cliente. Todavia, é possível criar uma subinterface e colocar as novas operações lá. Os clientes podem passar objetos de acordo com a nova interface onde quer que seja esperada a velha interface, mas o código existente continua funcionando como antes.

A flexibilidade adicional, contudo, aumenta a complexidade do framework. O framework precisa explicitamente verificar em tempo de execução sempre que quiser invocar operações na nova interface. Por exemplo, AWT tem duas versões da interface de gerenciamento de layout. Em alguns poucos lugares, AWT tem um código assim:

```
...
if (layout instanceof LayoutManager2) {
    LayoutManager2 layout2= (LayoutManager2) layout;
    layout2.newOperation();
}
...
...
```

Interfaces versionadas são um compromisso razoável quando se é obrigado a introduzir novas operações em uma abstração baseada em interface sem afetar o código cliente. Não são uma solução a longo prazo para abstrações que mudam com frequência, em razão da complexidade que criam para o cliente e para os desenvolvedores do framework. Abstrações mutáveis devem ser representadas como superclasses a fim de acomodar bem a mudança.

Superclasse

A alternativa para definir uma abstração por uma interface é pedir que os clientes passem uma instância de uma classe ou uma de suas subclasses. As vantagens e desvantagens desse estilo são inversas às das interfaces: classes podem especificar mais detalhes que interfaces, mas adicionar uma operação à superclasse não quebra o código existente. Diferentemente de interfaces, todavia, classes clientes podem estender apenas uma classe do framework.

Os detalhes de uma superclasse visível aos clientes são os métodos e campos públicos e protegidos da superclasse. Cada método ou campo promete não mudar. Se muitos detalhes são visíveis, o que pode ser uma excessiva quantidade de promessas, restringem-se severamente mudanças futuras de projeto.

Ter cuidado ao escrever uma superclasse pode reduzir essas restrições a essencialmente o que é disponibilizado pelas interfaces. Campos em um framework devem ser sempre privados. Se os clientes precisam acessar os dados nos campos, forneça-os por meio de getters. Examine cuidadosamente seus métodos e torne públicos apenas os métodos essenciais ou, melhor ainda, torne-os

protegidos. Seguir essas regras permite que se defina uma superclasse que expõe poucos detalhes a mais que a interface equivalente, mas que permite mais flexibilidade para que clientes associem (via *books*) sua própria lógica.

A palavra-chave `abstract` propicia uma maneira de comunicar aos clientes onde eles precisam preencher com lógica. Fornecer uma razoável implementação padrão dos métodos, onde possível, dá aos clientes a habilidade de começar mais facilmente. Entretanto, introduzir novos métodos abstratos em uma superclasse cria uma atualização incompatível, pois os clientes devem implementar os métodos antes de suas subclasses voltarem a ficar complicadas.

A palavra-chave `final`, quando aplicada à classe, evita que clientes criem subclasses, forçando o estilo de uso de framework por instanciação ou configuração. Quando aplicada a um método, essa palavra-chave permite que o desenvolvedor do framework presuma que um código particular está sendo executado, mesmo em um método visível ao cliente. Embora eu respeite a prerrogativa dos desenvolvedores de framework de quererem simplificar suas tarefas de programação, também tenho me frustrado com métodos e classes finais. Certa vez, gastei dois dias tentando infrutiferamente criar de maneira programática eventos SWT para testes. Eram (desnecessariamente, pareceu-me) as classes finais que me impediam de codificar o que precisava. Acabei escrevendo minhas próprias classes de eventos para duplicar os eventos SWT de forma que pudesse testar sem uma GUI. Guardar `final` para situações em que há um retorno substancial para o programador e poucos problemas aos clientes melhorará as relações entre o desenvolvedor do framework e o cliente.

Ainda no tópico visibilidade, devo apontar uma falha no esquema de pacotes de Java. Frameworks organizados em vários pacotes precisam de uma declaração de visibilidade que diga “Visível dentro do framework, mas não para clientes”. Uma solução para esse problema é separar os pacotes em publicados e internos e comunicar as diferenças incluindo o nome “interno” nos caminhos do pacote interno. No Eclipse, por exemplo, veem-se pacotes como `org.eclipse.jdt...` e `org.eclipse.jdt.internal....`.

Pacotes internos são um meio termo entre revelar e ocultar detalhes do framework. Clientes podem escolher sozinhos quanta responsabilidade querem aceitar ao construir sobre partes instáveis do framework. Às vezes, a funcionalidade necessária ao cliente está no framework, sendo apenas mal classificada (dependendo da perspectiva) pelos desenvolvedores.

Criação

Se o framework publica quaisquer classes concretas, é preciso decidir como clientes podem instanciá-las. Como ocorre com as outras decisões de projeto do framework, na escolha de seu estilo de instanciação, devem-se equilibrar generalidade, complexidade, facilidade de aprendizado e facilidade de evolução. Os quatro estilos descritos a seguir não têm instanciação do cliente, construtores, métodos fábrica e objetos fábrica e não são mutuamente exclusivos. Pode-se usar mais que um estilo para um objeto ou podem-se usar estilos diferentes para diferentes partes de seu framework.

Sem criação

A opção mais simples e menos poderosa é proibir os clientes de criarem diretamente objetos no framework. O exemplo do evento SWT apresentado anteriormente demonstra isso. Ao construírem eventos sempre dentro do framework, os desenvolvedores conseguem garantir que os eventos sejam bem formados. O código do framework pode ficar mais simples se for possível presumir invariantes sobre eventos.

A limitação de não se permitir que clientes criem instâncias de classes de framework é que isso impede usos legítimos para as classes não previstos pelos desenvolvedores do framework. Para tarefas de programação muito difíceis, em que qualquer redução de complexidade é bem-vinda, eliminar a possibilidade de que os clientes criem objetos pode ser uma boa opção. A utilidade de um framework muitas vezes não é aquela originalmente esperada pelos seus desenvolvedores. Excluir a possibilidade de usos inesperados reduz a chance de se encontrarem valiosos usos adicionais para o framework.

Construtores

Oferecer aos clientes a possibilidade de criar objetos por meio de construtores é uma opção simples, mas cria restrições substanciais para uma mudança futura. Quando se divulga um construtor, promete-se que não mudarão o nome da classe, os parâmetros necessários para criação, o pacote da classe e (o mais restritivo de tudo) a classe concreta do objeto retornado.

A maioria das bibliotecas de Java oferece criação por meio de construtores. Quando a Sun divulgou que listas são criadas dizendo-se `ArrayList()`, estava se comprometendo a manter uma classe chamada `ArrayList` no pacote `java.util` que não alterasse a classe concreta retornada. Essas são todas as restrições substanciais de projeto a serem mantidas por um futuro indeterminado, limitando-se os tipos de mudanças que a Sun pode fazer.

A vantagem de representar a criação de objetos como um construtor é ser algo simples e claro para os clientes. Se os clientes precisam de uma interface de criação fácil de usar e o programador não se importa em desistir da possibilidade de alterar o nome, o pacote e a classe concreta de suas abstrações, então os construtores são uma opção razoável.

Fábricas estáticas

Fábricas estáticas adicionam certa complexidade à criação de objetos para os clientes, mas deixam o desenvolvedor do framework mais livre para futuras mudanças de projeto. Se um cliente criou uma lista dizendo `ArrayList.create()` em vez de usar um construtor, então o nome, o pacote e a classe concreta do objeto retornado poderiam ser mudados sem afetar o código cliente. Um passo adicional seria concentrar os métodos fábrica em uma classe biblioteca: `Collections.createArrayList()`. Com esse estilo de fábrica, a única classe que precisaria permanecer no pacote original `java.util` é a classe biblioteca. Todas as outras classes poderiam ser movidas se necessário. No entanto, quanto mais abstrata for a criação, mais difícil será ver, a partir da leitura do código, onde os objetos são criados.

Outra vantagem dos métodos fábrica é que permitem comunicar claramente aos clientes o significado de variações na construção. Os propósitos de dois construtores com diferentes conjuntos de parâmetros nem sempre são óbvios, mas os nomes dos métodos fábrica podem sugerir as razões pelas quais os clientes poderiam criar um objeto de cada jeito.

Objeto fábrica

Pode-se também representar a criação de instância enviando mensagens a um objeto fábrica em vez de invocar um método estático. Por exemplo, uma `CollectionFactory` forneceria métodos para criar todos os diferentes tipos de coleções, podendo ser usada assim: `Collections.factory().createArrayList()`. Um objeto fábrica propicia até mais flexibilidade que uma fábrica estática, mas é mais complexo de ler. É preciso rastrear a execução do código para ver quando são criadas certas classes.

Quando a fábrica é acessada globalmente, um objeto fábrica não propicia mais flexibilidade que métodos estáticos de fábrica. Objetos fábrica mostram sua força quando usados localmente. Por exemplo, tendo-se coleções especiais que economizam uso em dispositivos móveis, seria possível inicializar os objetos que precisam criar coleções com a coleção que economiza espaço, se o código fosse executado em um aparelho portátil, e com uma coleção padrão, quando executado em um servidor.

Objetos fábrica podem ser úteis para criar conjuntos de classes que se harmonizam. Se os elementos de interface Windows funcionam juntos, mas não com os elementos de interface Linux, oferecer a criação por meio de um objeto fábrica é uma maneira de ajudar clientes a criar apenas classes compatíveis.

Conclusão de criação

A forma como se representa a criação de objetos em um framework afeta a facilidade de uso ou alteração do framework. Uma estratégia é oferecer métodos fábrica para as classes suscetíveis a mudanças e construtores para classes estáveis. Entretanto, há também valor em uma estratégia de criação consistente em que todos os objetos são criados por meio de métodos fábrica ou objetos fábrica.

Métodos

Além de criação de objetos, outros métodos também afetam a facilidade de usar e evoluir o framework. A estratégia geral permanece: revele a menor quantidade possível de detalhes, mas não deixe de ajudar os clientes a resolver seus problemas.

Getters e setters visíveis ao cliente são adequados apenas quando as estruturas de dados são estáveis. Encorajar clientes a depender de estruturas de dados internas reduz drasticamente as opções de evolução futura do framework. Nesse sentido, setters são piores que getters. Muitas vezes é possível descobrir uma maneira alternativa de computar um valor que costumava ficar armazenado em um campo. Tente entender qual problema o cliente resolve ao atribuir um valor. Em vez de publicar um setter, publique um método com nome baseado no problema que o cliente precisa resolver, e não em sua implementação.

Por exemplo, ao escrever uma biblioteca de elementos de interface gráfica, pode-se oferecer um método setter `setVisible(boolean)` de `Widget`. O que acontece quando se introduz um terceiro estado inativo? Para simplificar isso para o cliente, publique métodos que revelem sua intenção como `visible()` e `invisible()`. É isso que significa `setVisible()` para o cliente. Com esses métodos ativos, adicionar `inactive()` à superclasse atinge a meta de não afetar o código cliente.

Abstrações baseadas em interface são levemente diferentes. Adicionar `inactive()` em uma interface quebra quaisquer implementações de `Widget` pelo cliente. Em vez disso, defina um tipo enumerado `States` que registre os possíveis estados do elemento de interface e disponibilize um método `setVisible(State)`. A variante com o booleano é um exemplo de informação de projeto que vazia para os clientes. Booleanos implicam a existência de apenas dois estados possíveis. O projeto com um método só e um tipo enumerado como parâmetro permite a liberdade de se adicionarem outros estados logo que se tornem necessários. Isso não significa que getters e setters nunca devam ser divulgados para uso do cliente. Se uma importante funcionalidade do framework é implementada retornando ou atribuindo um campo, disponibilize o acessador. Entretanto, nomeie o método de forma que não revele sua implementação aos clientes.

Outra estratégia em nível de método para que desenvolvedores de frameworks mantenham a compatibilidade é fornecer valores padrões quando se adicionam parâmetros em métodos disponibilizados. Quando se adiciona um parâmetro em um método, invocações do método precisarão ser mudadas antes de o código do cliente ser compilado. Contudo, é possível manter o código do cliente funcionando quando se consegue manter o método antigo e fazê-lo invocar o novo método com um parâmetro padrão.

Por exemplo, suponha que em JUnit se queira a capacidade de passar um `TestResult` a um método que roda testes nas classes. Pode-se modificar o método apenas adicionando o parâmetro.

```
public TestResult run(Class... classes) {  
    ....run tests in classes...  
}  
public void run(TestResult result, Class... classes) {  
    ...run tests in classes...  
}
```

Qualquer cliente que invocasse `run(Class...)` teria de mudar para adicionar um parâmetro `TestResult`. Entretanto, o método original pode fornecer um parâmetro padrão:

```
public TestResult run(Class... classes) {  
    TestResult result= new TestResult();  
    run(result, classes);  
    return result;  
}
```

Ao fornecer o parâmetro padrão, o código cliente continua funcionando, mesmo se a interface também oferecer um novo método.

Conclusão

Desenvolvimento e evolução de frameworks requerem alguns padrões de implementação diferentes daqueles do desenvolvimento de aplicações. A mudança na economia do desenvolvimento – antes dominada pelo custo de entender o código e agora dominada pelo custo de atualizar o código do cliente – pede uma mudança substancial de práticas e valores. Para o desenvolvimento de frameworks, a simplicidade, diretiva primordial no desenvolvimento de aplicativos, tem prioridade mais baixa que a necessidade de se manter livre o posterior crescimento do framework. Isso é complicado quando partes de um aplicativo são extraídas para criar um framework. Muitas decisões de projeto precisarão ser revisitadas para tornar um framework eficaz.

Frameworks evoluem em uma variedade de formas. Às vezes os cálculos dos métodos existentes precisam melhorar; outras vezes, eles precisam funcionar com novos tipos de parâmetros. Há casos em que o framework pode ser usado, com um pequeno ajuste, para resolver um problema inteiramente inesperado. Em outros casos, detalhes de implementação no framework precisam ser publicados.

Uma metáfora que nos serviu bem em JUnit é olhar para o framework como a interseção de todas as funcionalidades úteis em um domínio, e não como uma união. Esse é o trabalho do desenvolvedor de frameworks: garantir que os clientes consigam estender o framework para resolver o restante de seus problemas. É atraente tentar resolver uma ampla gama de problemas com um framework. O problema é que a funcionalidade adicionada torna o framework muito mais difícil de aprender e usar para todos os clientes.

Se cada usuário potencial de um framework tiver 90% de requisitos comuns e apenas 10% de necessidades únicas, um framework para satisfazer a todos os desenvolvedores seria muito maior que um que satisfizesse apenas às necessidades comuns. A meta de um desenvolvedor de framework é descobrir as necessidades comuns dos usuários, e não todas suas necessidades únicas. Se a maioria dos usuários tiver de adicionar a mesma funcionalidade, então ela pertence ao framework, mas funcionalidades isoladas são mais bem manipuladas diretamente por quem as necessita.

Uma forma de estimular o tamanho adequado de frameworks é derivá-los de muitos exemplos concretos em vez de começar a partir de um caso geral. Escrevi o precursor do JUnit depois de algumas tentativas de automatizar testes em meu código. Cada variante resolia apenas um problema que eu encontrava. Somente depois de ter escrito o mesmo código muitas vezes, fui capaz de ver quais problemas eram comuns a todos os testes e precisavam ser cobertos pelo framework e quais problemas eram restritos a uma situação individual.

Pegue os conceitos para seu framework de uma ou mais metáforas claras e consistentes. Por exemplo, se contabilidade de entradas duplas (*double-entry bookkeeping*) é a metáfora usada para registrar históricos, os clientes saberão que devem procurar Conta e Transação. Ao escolher e aplicar metáforas de forma

consciente e comunicá-las a seus clientes, seus frameworks ficam fáceis de entender, usar e estender.

Implantar um framework não precisa ser o fim da evolução e do crescimento. Cuidados durante sua construção podem resultar em uma base estável para aplicações de clientes e um alicerce dinâmico para o desenvolvimento futuro do framework.

APÊNDICE

Medindo desempenho

Este apêndice descreve o framework utilizado para medir os dados sobre desempenho de coleções descrito no Capítulo 9. O problema é estabelecido de forma suficientemente simples – comparando-se precisamente o tempo necessário para realizar várias operações conforme escalam*. Entretanto, o problema se torna mais complicado quando a precisão do cronômetro é muito menor que o tempo necessário para completar as operações. O testador de tempo aqui apresentado supera esse problema ao realizar operações mais vezes. Ele adapta a precisão do cronômetro, corrigindo a quantidade de tempo de relógio usada para medir cada operação.

Medir precisamente o desempenho de operações em uma implementação otimizada de Java requer mais conhecimento – do framework ou dos próprios testes – do que apresentamos. Para obter resultados precisos, é preciso saber o que o otimizador provavelmente fará com seu código, para evitar que uma implementação inteligente elimine sua operação por completo. Se os resultados de medida de desempenho não corresponderem à sua intuição, entenda como um convite a cavar mais fundo. Assim, você aprenderá algo sobre medição de desempenho ou sobre o código que está medindo.

O código listado aqui basta para que sejam gerados os dados necessários para este livro, mas ele poderia ser feito de maneira mais geral. Por exemplo: os parâmetros para tempo são representados por constantes em vez de variáveis, não há interface por linhas de comando e o relatório é simples e mostrado no console. Uma das habilidades importantes em programação é associar o custo ao benefício. Aprender a usar padrões é uma habilidade diretamente ligada a aprender quando usá-los e quando deixá-los de lado.

Exemplo

O cronômetro deveria ser capaz de medir operações escritas da forma mais simples possível. Seguindo a linha de JUnit, operações a serem testadas são

* N. de R. T.: Ou seja, à medida que o tamanho das coleções aumenta.

representadas por métodos. Instâncias de métodos testadores de tempo serão construídas com um tamanho particular, de forma que testes de tempo possam ser feitos conforme os dados escalam. Por exemplo, para testar o tempo necessário para procurar em uma lista, o raciocínio seria algo como:

```
public class ListSearch {  
    private List<Integer> numbers;  
    private int probe;  
  
    public ListSearch(int size) {  
        numbers= new ArrayList<Integer>();  
        for (int i= 0; i < size; i++)  
            numbers.add(i);  
        probe= size / 2;  
    }  
  
    public void search() {  
        numbers.contains(probe);  
    }  
}
```

O resultado de executar o framework com essa classe será o tempo necessário para executar `search()` para coleções de tamanho 1, 10, 100 e assim por diante.

API

A interface externa para o cronômetro é a classe `MethodsTimer`. Ela é criada com um vetor de métodos:

```
public class MethodsTimer {  
    private final Method[] methods;  
  
    public MethodsTimer(Method[] methods) {  
        this.methods= methods;  
    }  
}
```

Invoque um `MethodsTimer` enviando a ele `report()`. Por exemplo, para cronometrar operações na `ListSearch` acima, execute o seguinte:

```
public static void main(String[] args) throws Exception {  
    MethodsTimer tester= new MethodsTimer(ListSearch.class.getDeclaredMethods());  
    tester.report();  
}
```

Executar esse método faz com que os resultados sejam mostrados no console:

search	34.89	130.61	989.73	9911.19	97410.83	990953.62
--------	-------	--------	--------	---------	----------	-----------

Isso significa que a operação de busca leva 35 nanossegundos para uma lista de um elemento, 131 nanossegundos para uma lista de dez elementos e assim por diante.

O cronômetro não é totalmente preciso. Rodá-lo na classe de cronômetro `Nothing` cronometra um método vazio que, teoricamente, deveria retornar resultados zero. Em vez disso, as respostas (em minha máquina, neste caso) são de uns poucos nanossegundos:

nothing	1.92	-3.24	0.62	0.37	-0.74	2.30
---------	------	-------	------	------	-------	------

Mantenha a precisão em mente se estiver cronometrando operações muito curtas. Por exemplo, para cronometrar acesso a um vetor, é preciso escrever um método que acesse um vetor dez vezes para obter tempos precisos. No entanto, geralmente o objetivo para o cronômetro é que programadores escrevam operações simples e deixem o framework repeti-las tantas vezes quanto necessário, obtendo dessa forma um tempo preciso.

Implementação

Observe que o tempo imprime seis respostas para cada método cronometrado; isso porque o cronômetro é usado para testar operações conforme escalam. O método `report()` é um laço aninhado no qual o laço externo itera através dos métodos para ser cronometrado, e o laço interno itera sobre tamanhos de 1, 10,..., 100.000.

```
private static final int MAXIMUM_SIZE= 100000;
public void report() throws Exception {
    for (Method each : methods) {
        System.out.print(each.getName() + "\t");
        for (int size= 1; size <= MAXIMUM_SIZE; size*= 10) {
            MethodTimer r= new MethodTimer(size, each);
            r.run();
            System.out.print(String.format("%.2f\t", r.getMethodTime()));
        }
        System.out.println();
    }
}
```

O relatório é tão simples quanto possível, com tabulações inseridas entre os elementos de dados de forma que estes possam facilmente ser colados em uma planilha. Em um cronômetro cheio de funcionalidades, provavelmente computaríamos todos os `MethodTimers` primeiro, para, depois, relatá-los em uma segunda etapa, a fim de que o relatório pudesse ser mais flexível.

MethodTimer

`MethodsTimer` depende de um objeto auxiliar `MethodTimer`, um comando que calcula o tempo necessário para executar um só método. O método será invocado tantas vezes quanto forem necessárias, até que tenha decorrido tempo suficiente para uma leitura precisa. O tempo levado pelo método, então, é o total dividido pelo número de invocações.

Cada método `MethodTimer` é construído com um método e um tamanho. Em virtude do fato de o método poder ser invocado muitas vezes, cada `MethodTimer` armazena um cache de um objeto que pode ser enviado por uma mensagem invocando o método. Criar uma nova instância para cada invocação seria algo bastante demorado. Se uma operação leva 50 nanossegundos, ela precisará ser executada 20 milhões de vezes para coletar um segundo de dados. Criar uma lista com 100 mil elementos como aquela usada em `ListSearch` (acima) leva aproximadamente 50 milissegundos em minha máquina; logo, executar esse único teste levaria uma semana e meia. Armazenando um cache da instância, o teste leva pouco mais que um segundo, apenas.

Esse reúso de instâncias é um projeto diferente daquele de JUnit. Em JUnit, uma instância nova é criada para cada teste executado; os testes podem então livremente promover mudanças no estado da instância, uma vez que estão isolados dos testes subsequentes. O testador de tempo, no entanto, não fornece tal liberdade. Cada método cronometrado precisa deixar o estado da instância de teste exatamente como o encontrou.

Aqui está o construtor de `MethodTimer`:

```
private final int size;
private final Method method;
private Object instance;

MethodTimer(int size, Method method) throws Exception {
    this.size= size;
    this.method= method;
    instance= createInstance();
}
```

Cada método tem uma classe, e essa é a classe instanciada quando uma operação é cronometrada. Isso faz com que a implementação atual não suporte a herança de métodos de cronometragem e, consequentemente, a chamada de `getDeclaredMethods()` em `MethodsTimer`. `getDeclaredMethods()` apenas retorne métodos declarados na classe e não nas superclasses. Um plano mais interessante seria anotar os métodos a serem cronometrados e procurar em toda a cadeia de superclasses quando esses métodos precisassem ser encontrados. Assim, hierarquias eliminariam alguma duplicação (podem ser vistas nos cronômetros usados por este livro). Neste apêndice, mais adiante, estão os cronômetros específicos usados. Ressalte-se novamente que este framework pretende apenas cumprir satisfatoriamente os propósitos deste livro. Um framework usado por muitas pessoas requer um projeto com outra filosofia, ou seja, deve haver um

investimento muito maior, uma vez que qualquer melhoria possibilitada por um projeto mais poderoso (embora caro) é reembolsada milhares de vezes.

Criar a instância para usá-la envolve encontrar um construtor que aceita um int como parâmetro e o invoca.

```
private Object createInstance() throws Exception {  
    Constructor<?> constructor= method.getDeclaringClass().getConstructor(new Class[]{int.class});  
    return constructor.newInstance(new Object[]{size});  
}
```

Há, na realidade, três fatores fundamentais para computar o tempo necessário à execução de uma única invocação do método: o número de iterações, o tempo total para invocar o método tal quantidade de vezes e o custo de invocar um método por reflexão. Uma vez que a invocação do método por reflexão pode ser cara se comparada ao tempo necessário para executá-lo (aproximadamente 150 nanossegundos em minha máquina), remover essa sobrecarga melhora a precisão do cronômetro. Cada um desses fatores será computado pelo método run() e armazenado em campos.

```
private long totalTime;  
private int iterations;  
private long overhead;  
double getMethodTime() {  
    return (double) (totalTime - overhead) / (double) iterations;  
}
```

O método run() é o coração do cronômetro. Ele invoca o método para ser cronometrado uma vez, duas vezes, quatro vezes e assim por diante, até que um segundo tenha sido consumido. Então, ele computa o custo de sobrecarga de quaisquer invocações dinâmicas que foram necessárias. Essa dependência temporal entre run() e qualquer método de consulta (como getMethodTime() acima) é um pouco inadequada. A alternativa é ter o construtor também computando o tempo de execução. Relutamos em ter construtores fazendo trabalho significativo, pois é mais interessante ter liberdade para desacoplar a criação de instâncias do trabalho que está sendo feito. Projetando dessa forma, é possível optar por criar uma coleção de MethodTimers e passá-la adiante, serializá-la ou enviá-la pela rede sem preocupações com o desempenho.

```
void run() throws Exception {  
    iterations= 1;  
    while (true) {  
        totalTime= computeTotalTime();  
        if (totalTime > MethodsTimer.ONE_SECOND)  
            break;  
        iterations*= 2;  
    }  
    overhead= overheadTimer(iterations).computeTotalTime();  
}
```

Observe o uso de outra constante, ONE_SECOND, em `MethodsTimer`. Usar constantes para configuração é uma maneira de fornecer, com baixo custo, alguma flexibilidade a usuários que estejam dispostos a editar o código fonte, mas é melhor que essas constantes sejam colocadas juntas, a fim de que sejam facilmente encontradas.

```
static final int ONE_SECOND= 1000000000;
```

Cancelando sobrecarga

Tudo o que resta do framework são os métodos para computar o custo de sobrecarga da invocação dinâmica do método. O método estático de fabricação `overheadTimer()` tem a intenção de comunicar o propósito desse cronômetro especial.

É um pouco estranho ter um `MethodTimer` invocando outro como parte de seu trabalho, mas, depois de muito experimentos, essa foi a melhor forma que encontrei para estruturar o código.

```
private static MethodTimer overheadTimer(int iterations) throws Exception {  
    return new MethodTimer(iterations);  
}  
  
private MethodTimer(int iterations) throws Exception {  
    this(0, MethodTimer.Overhead.class.getMethod("nothing", new Class[0]));  
    this.iterations= iterations;  
}  
  
public static class Overhead {  
    public Overhead(int size) {  
    }  
  
    public void nothing() {  
    }  
}
```

Testes

Aqui estão os testes usados para gerar os dados apresentados no capítulo “Coleções”. Eles demonstram o uso do framework de cronômetro, algumas das peculiaridades das classes Coleção e, ao mesmo tempo, algumas limitações do projeto apresentado aqui.

Comparando coleções

O primeiro exemplo compara o uso de um Set e de uma ArrayList como uma Collection. O construtor cria as duas coleções com um dado tamanho e as inicializa. Os dados usados como elementos são strings. Os valores hash dos strings não serão randomicamente distribuídos. Entretanto, procurei usar Integers como dados e o comportamento pareceu ainda mais estranho. Uma vez que os valores hash para grandes conjuntos raramente são distribuídos de forma randômica, se o desempenho em larga escala é importante para você, você pode querer criar um dado “típico” representando os elementos que computa, se estiver cronometrando coleções para seu próprio uso.

Cada conjunto de testes de tempo é representado como uma classe. As cronometragens em si são representadas pelos métodos daquela classe. A classe armazena coleções para serem computadas. Observe que ambas são declaradas como Collection. A classe também armazena uma sonda, que é um elemento a ser procurado posteriormente.

```
public class SetVsArrayList {  
    private Collection<String> set;  
    private Collection<String> arrayList;  
    private String probe;  
}
```

Para inicializar uma instância, as coleções são preenchidas com elementos. Observe que a sonda está no meio da coleção, de forma a fornecer um cenário de “pior caso”. Um exame mais completo do desempenho de coleção usaria elementos em seu início, meio e fim.

```
public SetVsArrayList(int size) {  
    set= new HashSet<String>(size);  
    arrayList= new ArrayList<String>(size);  
    for (int i= 0; i < size; i++) {  
        String element= String.format("a%d", i);  
        set.add(element);  
        arrayList.add(element);  
    }  
    probe= String.format("a%d", size / 2);  
}
```

O primeiro par de operações compara o tempo necessário para testar a inclusão no conjunto. A única diferença entre os dois métodos é a coleção testada. O tempo necessário para verificar a inclusão em HashSet é próximo à constante, enquanto o tempo necessário para verificar a inclusão em ArrayList cresce linearmente com o tamanho da coleção.

```
public void setMembership() {  
    set.contains(probe);  
}
```

```
public void arrayListMembership() {  
    arrayList.contains(probe);  
}
```

A duplicação entre esses dois métodos sugere uma API alternativa com menos duplicação, na qual a classe de teste conteria uma implementação abstrata dos métodos a serem cronometrados. Cada instância seria inicializada com uma classe concreta para ser testada:

```
public class CollectionOperations {  
    Collection<String> collection;  
    String probe;  
    public void membership() {  
        collection.contains(probe);  
    }  
}
```

A classe concreta de coleção poderia ser inicializada em um construtor ou em uma subclasse. Enquanto esse projeto resulta em menos duplicação e seria superior para um framework amplamente distribuído, o projeto atual contempla os dados necessários a este livro, e, assim, a duplicação será mantida.

Outro par de métodos em `SetVsArrayList` mede o tempo necessário para iterar sobre as coleções. O tempo necessário é linear no número de elementos.

```
public void setIteration() {  
    Iterator<String> all= set.iterator();  
    while (all.hasNext())  
        all.next();  
}  
  
public void arrayListIteration() {  
    Iterator<String> all= arrayList.iterator();  
    while (all.hasNext())  
        all.next();  
}
```

Tentei endireitar a iteração com um laço `for`, `for(String each: set)`, mas a implementação Java foi inteligente o bastante para perceber o laço com corpo vazio e eliminar o laço inteiramente. Em geral, um dos desafios ao escrever métodos de cronômetro é mantê-los simples, enquanto não permitimos que otimizações de Java os eliminem completamente. Verifique sempre seus resultados para ter certeza de que fazem sentido. Se não fizerem, tente outra forma de expressar o mesmo cálculo.

Os métodos de cronômetro finais verificam o tempo necessário para modificar uma coleção. Os métodos são cuidadosos para deixar as coleções inalteradas no resultado. Essa é uma limitação do framework de cronometragem:

uma vez que cada método será invocado muitas vezes com o mesmo objeto, os métodos precisam deixar o estado do objeto inalterado.

```
public void setModification() {  
    set.add("b");  
    set.remove("b");  
}  
  
public void arrayListModification() {  
    arrayList.add("b");  
    arrayList.remove("b");  
}
```

As cronometragens dessas operações são as mesmas da cronometragem do teste de inclusão: HashSet é fortemente constante e ArrayList é linear.

Comparando ArrayList e LinkedList

Este teste é similar ao teste acima, exceto pelo fato de as variáveis a testar serem declaradas como List em vez de Collection, e serem inicializadas com uma ArrayList e uma LinkedList.

LinkedList

```
public class Lists {  
  
    private List<String> arrayList;  
    private List<String> linkedList;  
    private final int size;  
}
```

Em vez de armazenar um elemento para sondar, como quando testamos coleções, este teste lembra o tamanho das coleções para ser usado posteriormente com o método get(int) de List.

```
public Lists(int size) {  
    this.size= size;  
    arrayList= new ArrayList<String>(size);  
    linkedList= new LinkedList<String>();  
    for (int i= 0; i < size; i++) {  
        String element= String.format("a%d", i);  
        arrayList.add(element);  
        linkedList.add(element);  
    }  
}
```

O primeiro par de testes mede o tempo necessário para modificar uma coleção inserindo e então removendo um elemento. Perceba que o elemento é

inserido no começo da coleção. `ArrayList` otimiza a inserção no final, de forma que leva tempo constante, igual a `LinkedList`, em vez de tempo linear.*

```
public void arrayListModification() {  
    arrayList.add(0, "b");  
    arrayList.remove(0);  
}  
  
public void linkedListModification() {  
    linkedList.add(0, "b");  
    linkedList.remove(0);  
}
```

Os outros testes medem o tempo necessário para acessar um elemento. Os resultados são um espelho do teste de modificação: o acesso de `ArrayList` é constante e o acesso de `LinkedList` é linear. Minha primeira versão desse teste acessava o elemento no final da coleção, mas `LinkedList` revelou otimizar esse caso procurando de trás para frente por índices maiores que metade de seu tamanho.

```
public void arrayListAccess() {  
    arrayList.get(size / 2);  
}  
  
public void linkedListAccess() {  
    linkedList.get(size / 2);  
}
```

Comparando conjuntos

Conjuntos seguem o mesmo padrão básico usado na comparação de listas acima. As duas operações comparadas são modificação e verificação de inclusão, uma vez que a maioria das outras operações é construída a partir destas duas ou mostra perfil de desempenho similar. No código abaixo será apresentada apenas uma variante de cada um dos métodos de cronometragem, já que as outras são idênticas, exceto pelo conjunto ao qual as mensagens são enviadas.

As três implementações de conjuntos comparadas são `HashSet`, `LinkedHashSet` e `TreeSet`. Falando corretamente, `TreeSet` é uma implementação de `SortSet`, mas

* N. de R. T.: `ArrayList` e `LinkedList` são duas implementações de `List`. `LinkedList` permite inserções e remoções com tempo constante mas somente acesso sequencial aos elementos. `ArrayList`, por outro lado, permite acesso aleatório aos elementos portanto com tempo de acesso constante embora a adição e remoção de elementos (exceto no final) requeira deslocar elementos e por isto tome mais tempo.

achei que seria útil comparar o custo de sobrecarga imposto por manter os elementos ordenados.

```
public class Sets {  
    private Set<String> hashSet;  
    private Set<String> linkedHashSet;  
    private Set<String> treeSet;  
    private String probe;  
}
```

O construtor inicializa cada um dos conjuntos com elementos idênticos e inicializa uma sonda para ser usada posteriormente na cronometragem da verificação de inclusão. Perceba que cada conjunto é criado com a capacidade de armazenar um número adequado de elementos. Toda a literatura Java que li enfatiza a importância de pré-alocar o tamanho correto de coleções. Entretanto, minhas medições mostram que pré-alocar zero elementos está dentro de 10% da pré-alocação do número final de elementos.

```
public Sets(int size) {  
    hashSet= new HashSet<String>(size);  
    linkedHashSet= new LinkedHashSet<String>(size);  
    treeSet= new TreeSet<String>();  
    for (int i= 0; i < size; i++) {  
        String element= String.format("a%d", i);  
        hashSet.add(element);  
        linkedHashSet.add(element);  
        treeSet.add(element);  
    }  
    probe= String.format("a%d", size / 2);  
}
```

Para cronometrar a verificação de inclusão, procure no conjunto pela sonda pré-computada. As outras implementações de conjunto são cronometradas similarmente.

```
public void hashSetContains() {  
    hashSet.contains(probe);  
}
```

O método de cronometragem de modificação adiciona e, então, remove o mesmo elemento, deixando o conjunto inalterado.

```
public void hashSetModification() {  
    hashSet.add("b");  
    hashSet.remove("b");  
}
```

Comparando mapas

Cronometrar mapas assemelha-se muito a cronometrar conjuntos. Novamente, há três implementações disponíveis na biblioteca Java – `HashMap`, `LinkedHashMap` e `TreeMap`. Os resultados de cronometragem são os mesmos, já que os conjuntos são implementados como mapas. Isto é, um `HashSet` usa internamente um `HashMap` para armazenar os elementos, e assim por diante.

```
public class Maps {  
    private Map<String, String> hashMap;  
    private Map<String, String> linkedHashMap;  
    private Map<String, String> treeMap;  
    private String probe;
```

Iniciar os mapas requer enviar a eles `put()` em vez de `add()`. Escolhi ter apenas uma chave e um valor, uma vez que, para propósitos de cronometragem, não importa quais são os valores.

```
public Maps(int size) {  
    hashMap= new HashMap<String, String>(size);  
    linkedHashMap= new LinkedHashMap<String, String>(size);  
    treeMap= new TreeMap<String, String>();  
    for (int i= 0; i < size; i++) {  
        String element= String.format("a%d", i);  
        hashMap.put(element, element);  
        linkedHashMap.put(element, element);  
        treeMap.put(element, element);  
    }  
    probe= String.format("a%d", size / 2);  
}
```

Os dois métodos de cronometragem usam as operações de mapa `containsKey()` e `put()` em vez das operações de conjuntos `contains()` e `add()`. No mais, são idênticos.

```
public void hashMapContains() {  
    hashMap.containsKey(probe);  
}  
public void hashMapModification() {  
    hashMap.put("b", "b");  
    hashMap.remove("b");  
}
```

Conclusão

O framework e os elementos dados oferecem lições em diversos níveis. Uma delas é a importância da obtenção de dados. Crenças amplamente difundidas, como “pré-alocação de conjuntos melhora o desempenho”, merecem um exame minucioso. Antes de tornar um programa complexo, assegure-se de que a complexidade trará benefícios. Às vezes, a única forma de descobrir se obtemos os benefícios é nos dispondo a medi-los.

Outra lição dos métodos do framework de cronometragem é a importância de ajustar o estilo de código com base no contexto. Eu teria projetado e codificado o framework de maneira muito diferente se tivesse um público maior ou mais testes de cronometragem para escrever. Como pôde-se observar, a simplificação de premissas reduziu o esforço total de escrever o framework e os testes. Conselhos dogmáticos como “sempre inclua toda a flexibilidade que puder imaginar” ou “codifique para o hoje; esqueça o amanhã” estão igualmente equivocados.

Finalmente, o código deste capítulo oferece exemplos de muitos dos padrões de implementação constantes do restante do livro. Construtores Completos, Nomes que Revelam Intenção, e assim por diante, são representados em cada linha de código. Se eles de fato comunicaram minhas intenções, é algo que apenas você pode julgar. Se não serviram para isso, descubra padrões que lhe sirvam melhor para a comunicação. Esta é a grande lição deste livro, afinal: a tarefa do programador é comunicar-se com outros programadores, e não apenas com a máquina. Programar, portanto, é uma tarefa humana, feita de humanos para humanos. Não precisa ser uma fuga da sociedade; pode ser um meio de conexão. Ah, e também de escrita de bons códigos.

Leituras sugeridas

Programação geral

Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997. ISBN 013476904X.

Os padrões de implementação para Smalltalk. Muitos são similares aos padrões listados aqui, mas há também diferenças significativas, pois as linguagens são muito diferentes. Escrevê-lo me forçou a desacelerar e pensar sobre decisões que estive tomando por instinto.

Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999. ISBN 0201485672.

É fácil dizer que projetos deveriam mudar um pouco por vez. Esse livro introduz como fazer essas mudanças.

Eric Freeman and Elisabeth Freeman, *Head First Design Patterns*, O'Reilly Media, 2004. ISBN 0596007124.

Uma introdução alternativa e visualmente orientada para padrões de projeto.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. ISBN 0201633612.

A descrição clássica de estruturas que se repetem em larga escala no código.

Daniel Hoffman and David Weiss, *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001. ISBN 0201703696.

Descreve a base teórica de um bom software.

Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000. ISBN 020161622X.

A atitude de um programador profissional é clara neste livro: curioso, honesto e sempre aprendendo.

Brian Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999. ISBN 020161586X.

Outra demonstração de programadores cuidadosamente profissionais no seu trabalho.

Donald Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms*, 3rd Edition, Addison-Wesley, 1997. ISBN 0201896834.

Donald Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, 3rd Edition, Addison-Wesley, 1997. ISBN 0201896842.

Donald Knuth, *The Art of Computer Programming: Volume 3, Searching and Sorting*, 2nd Edition, Addison-Wesley, 1998. ISBN 0201896850.

O professor Knuth claramente ama programar e transmite esse amor através de sua escrita.

Donald Knuth, *Literate Programming*, Center for the Study of Language and Information, 1992. ISBN 0937073806.

Um dos primeiros livros que se concentra na necessidade de programadores se comunicarem com outros programadores. Fonte de uma de minhas citações favoritas: “Um programa deveria ser lido como um livro.” Nem sempre vale a pena ir tão longe, mas a atitude está certa.

Steve McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd Edition, Microsoft Press, 2004. ISBN 0735619670.

Avalia as técnicas necessárias para programar responsávelmente.

Diomidis Spinellis, *Code Reading*, Addison-Wesley, 2003. ISBN 0201799405.

Uma visão do outro lado: como ler código. Este livro é uma imagem no espelho do livro “Padrões de implementação”; lendo para o entendimento, ao contrário de escrever para o entendimento.

Edward Yourdon, *Techniques of Program Structure and Design*, Prentice Hall, 1975. ISBN 013901702X.

Uma das mais antigas explicações sobre o que torna um programa bom. Os princípios ainda são os mesmos, ainda que os exemplos pareçam ultrapassados.

Edward Yourdon and Larry Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, 1979. ISBN 0138544719.

Esse livro apresenta o equivalente das leis da física para projeto de software e fundamenta a discussão sobre a economia de desenvolvimento.

Filosofia

Christopher Alexander, *Notes on the Synthesis of Form*, Harvard University Press, 1964. ISBN 0674627512.

Explica a teoria por trás dos padrões: decisões recorrentes com padrões recorrentes de restrições e soluções similares.

Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979. ISBN 0195024028.

A descrição teórica de projeto e construção com padrões. Um tema comum são as vantagens de projetar um pouco por vez usando o feedback de projetos, construções e usos anteriores.

Christopher Alexander, Sara Ishikawa, Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, *A Pattern Language*, Oxford University Press, 1977. ISBN 0195019199.

Um exemplo de uma ampla linguagem de padrões. Também útil no projeto de espaços de trabalho e casas.

Richard Gabriel, *Patterns of Software*, Oxford University Press, 1996. ISBN 019510269X.

Uma coleção de ensaios sobre a aplicação do pensamento em padrões no desenvolvimento de software.

Robert Grudin, *The Grace of Great Things*, Ticknor and Fields, 1990. ISBN 0395588685.

Celebra e encoraja o projeto excepcionalmente bom.

Leonard Koren, *Wabi-Sabi for Artists, Designers, Poets, and Philosophers*, Stone Bridge Press, 1994. ISBN 1880656124.

Projeto eficaz não é a busca por perfeição, mas, sim, por suficiência. Wabi-sabi é uma estética japonesa de real beleza, às vezes rígida, mas sempre funcional.

D'Arcy Thompson, *On Growth and Form*, Cambridge University Press, 1961. ISBN 0521437768.

Um livro às vezes difícil sobre a forma que a complexidade é criada e expressa no mundo natural.

Edward Tufte, *The Visual Display of Quantitative Information*, Graphics Press, 1983. ISBN 0961392142.

Um exemplo cativante de pensamento baseado em princípios, neste caso sobre design gráfico.

Java

Joshua Bloch, *Effective Java Programming Language Guide*, Addison-Wesley, 2001. ISBN 0201310058.

Uma descrição inicial de como usar Java, incluindo uma quantidade razoável de informação implícita sobre por que Java é do jeito que é.

Bruce Eckel, *Thinking in Java, 4th Edition*, Prentice Hall, 2006. ISBN 0131872486.

Minha bíblia de Java. Quando eu preciso saber como alguma coisa funciona em Java, esse é o livro que abro.

Steven Metsker, *Design Patterns Java Workbook*, Addison-Wesley, 2002. ISBN 0201743973.

Mostra como Java afeta os padrões de projeto.

Índice

A

- Abstração em frameworks, estilos de implementação de
 - interfaces, 121–122
 - questões na representação de framework como objeto, 119
 - superclasses, 121–124
 - visão geral, 121
- `abstract`, palavra-chave, 26–27, 122–123
- Acesso
 - a coleções, 90–91
 - direto, 46
 - indireto, 46–47
 - visão geral de, 45–46
- Algoritmos, métodos `get` e, 94
- API (interface de programação de aplicativos), para medição de desempenho, 130–131
- Aplicações, mudar frameworks sem mudar aplicações, 115–116
- Arquiteturas paralelas, atualizações e, 117
- `ArrayList`, classe
 - classes de implementação de coleções, 106–108
 - comparando com `LinkedList`, 137–139
- Arrays
 - como interface de coleção, 103–104
 - na API de medição de desempenho, 130–131
 - tamanho fixo de, 101
- Atualizações, gerenciando atualizações incompatíveis em frameworks, 116–118

B

- Bibliotecas, biblioteca `Collections`, 109–110
- Buscas, coleção, 110–112
- Buscas binárias, 110–111

C

- Campo de estado, papel de campo, 52–53
- Campos, 51–53
- Campos componente, 52–53
- Campos de estratégia, papel de campo, 52–53
- Chamadas de procedimentos, 65–66
- Classes, 21–41
 - abstrata, padrão, 25–26
 - biblioteca, padrão, 39–41
 - comportamento específico de instância, padrão, 33–34
 - condicional, padrão, 33–37
 - delegação, padrão, 36–38
 - especialização, padrão, 30–32
 - estado, *veja Estado*
 - implementador, padrão, 32–33
 - interface, padrão, 25–27
 - interface abstrata, padrão, 24–25
 - interface versionada, padrão, 26–28
 - interna, padrão, 32–34
 - interna anônima, padrão, 39–40
 - métodos organizados conforme, 75
 - nome qualificado de subclasse, padrão, 24
 - nome simples de superclasse, padrão, 23–24
 - objeto valor, padrão, 27–31
 - padrão, 22–23
 - seletor Plugável, padrão, 38–40
 - subclasse, padrão, 31–33
 - visão geral de, 21–23
- Classes de implementação de coleções
 - `ArrayList`, classe, 107–108
 - `List`, classe, 107–108
 - `Map`, classe, 108–110
 - `Set`, classe, 107–109
 - visão geral de, 106–108

- Classificação, subclasses usadas para, 31–32
Cláusulas de guarda, 69–72
Coleções, 99–114
 acessando, 90–92
 array, interface de, 103–104
 atualizando frameworks e, 117
 busca, 110–112
 Collection, interface, 103–104
 Collections, classe, 109–110
 com apenas um elemento, 112–113
 comparando, 135–137
 comparando ArrayList e LinkedList, 137–139
 comparando Sets, 138–140
 comparando Maps, 139–141
 desempenho e, 102
 estendendo, 112–114
 imutável, 112
 interfaces para, 103
 iterable, interface de, 103–104
 list, interface de, 103–105
 map, interface de, 105–106
 metáfora combinada por, 100–101
 ordenação, 111–112
 questões sobre, 101–102
 set, interface de, 104–105
 sorted set, interface de, 104–106
 vazias, 112–113
 visão geral, 99–100
Collection, interface, 103–104
Collections, classe
 busca, 110–112
 encorajando mudança compatível em frameworks, 119
 ordenamento, 111–112
 visão geral de, 109–110
Comentários de método, 84–85
Compartilhar código, delegação e, 37–38
Compatibilidade no desenvolvimento de frameworks
 encorajando mudança compatível, 118–119
 gerenciando atualizações incompatíveis, 116–118
 visão geral, 115–116
Complexidade
 eliminando excessiva, 11
 flexibilidade e, 12–13
 flexibilidade no framework e, 121–122
Comportamentos, 63–73
 cláusula de guarda, padrão, 69–72
 controle de fluxo, padrão, 64–65
 envio duplo, padrão, 66–67
 exceção, padrão, 72
 exceção verificada, padrão, 72–73
 fluxo excepcional, padrão, 69–70
 fluxo principal, padrão, 64–65
 mensagem, padrão, 65–66
 mensagem convidativa, padrão, 68–69
mensagem de decomposição (Sequenciamento), padrão, 67
mensagem de escolha, padrão, 65–66
mensagem explicativa, padrão, 69
mensagem inversa, padrão, 67–68
objeto, 43
propagação de exceção, padrão, 72–73
Compressão, hierarquias de organização de classes, 22–23
Comunicação
 benefícios de código limpo, 20
 comentários de métodos e, 84–85
 como valor em programação, 10–11
 da intenção, 1, 99
 flexibilidade e, 58–59
 nomes de classes e, 24
 simplicidade e, 11–12
Concorrência, estado e, 44
Conicionais
 cláusulas de guarda e, 70–71
 mensagens de escolha e, 65–66
 visão geral de, 33–37
Configuração, estilos de uso de frameworks, 120
Constantes, 57
Construtores
 completos, 88–90
 criando frameworks, 123–125
 de conversão, 87–88
 MethodTimer, objeto auxiliar, 132
 métodos fábrica comparados a, 89–90
Contador, variáveis locais, 50–51
Conversão
 construtores, 87–88
 métodos, 86–88
 visão geral de, 86–87
Cópia segura, métodos de, 95–97
Criação de objetos, 87–88
Criação em frameworks, métodos de
 construtores, 123–125
 fábricas estáticas, 124–125
 objetos fábrica, 124–125
 questões na representação de framework como objeto, 119
 resumo de, 125–126
 sem criação, 123–124
 visão geral, 123–124
Custos, estratégias de desenvolvimento e, 19–20
Custos de manutenção, custos de desenvolvimento comparados a, 19–20

D

Dados

- interação entre similaridades e diferenças em programação, 30–32
 princípio de alocar lógica e dados juntos, 13–15

- Declaração de tipo
 tipo declarado, padrão, 58–59
 variáveis, 50–51
- Declarar métodos, 80–81
- Declarativa, programação
 initialização de variáveis e, 59–60
 programação imperativa comparada a, 15–16
- Decomposição, mensagens de, 67
- Delegação
 compartilhar código e, 37–38
 comportamento específico de instância e, 36–38
- Desempenho
 coleções e, 102
 limitações de objetos parâmetro, 57
- Desenvolvimento, custos de desenvolvimento comparados a custos de manutenção, 19–20
- Diferenças, interação entre similaridades e diferenças em programação, 30–31
- Duplicação
 envio duplo e, 66
 minimizando repetição em codificação e, 13–14
- E**
- Eclipse, 117
- Economia de tempo, padrões para, 6
- Elemento, variáveis locais, 51–52
- Elementos, acessando em coleções, 102
- Envio duplo, mensagens de, 66–67
- equals(), 94
- Escopo variável, 49–50
- Especialização
 tamanho de método e, 78
 visão geral de, 30–32
- Estado, 43–61
 acesso, padrão, 45–46
 acesso direto, padrão, 46
 acesso indireto, padrão, 46–47
 campo, padrão, 51–53
 coletor, padrão, 54–55
 comum, padrão, 46–49
 constante, padrão, 57
 extrínseco, padrão, 49–50
 initialização, padrão, 59–60
 initialização ansiosa, padrão, 59–60
 initialização preguiçosa, padrão, 60–61
 nome sugestivo de papel, padrão, 57–59
 objeto parâmetro, padrão, 56–57
 padrão, 44–45
 parâmetro, padrão, 52–55
 parâmetro opcional, padrão, 55
 tipo declarado, padrão, 58–59
 var args, padrão, 55–56
 variável, padrão, 47–49
 variável, padrão, 49–51
 variável local, padrão, 50–52
 visão geral de, 43–44
- Estado booleano, 92
- Estendendo classes de coleção, 112–114
- Estilo
 de desenvolvimento, 9–10
 de padrões, 6
- Estilo de uso
 de frameworks, 120–121
 em JUnit, 121
 questões na representação de framework como objeto, 119
- Estilo funcional vs. estilo procedural de programação, 27–29
- Estratégia de atualização incremental, 117
- Exceções
 propagação de, 72–73
 verificadas, 72–73
 visão geral de, 72
- F**
- Fábricas estáticas, 124–125
- Fábricas internas, 90–91
- final, palavra-chave, estilos de uso de frameworks e, 122–123
- final, valores de campos, 51–52
- Flag, campos, 52–53
- Flexibilidade
 como valor em programação, 11–13
 complexidade de frameworks e, 121–122
 construtores e, 88
 delegação e, 36–37
 envio duplo e, 66
 estado variável e, 47–48
 interfaces e, 24–25
 tipos declarados e, 58–59
- Fluxo de controle
 cláusulas de guarda e, 69–72
 fluxo excepcional, 69–70
 fluxo principal, 64–65
 mensagens. *Veja Mensagens*
 visão geral de, 64–65
- Fluxo principal
 cláusulas de guarda e, 69–72
 fluxo excepcional e, 69–70
 visão geral de, 64–65
- Framework, desenvolvimento de, 115–128
 abstração e, 121
 classe biblioteca e, 119
 criando frameworks, 123–126
 encorajando mudança compatível, 118–119
 gerenciando atualizações incompatíveis, 116–118
 interfaces para, 121–122
 medição de desempenho. *Veja Medição de desempenho*
 métodos, 125–127

mudar frameworks sem mudar aplicações, 115–116
 opções de estilo de uso, 120–121
 representação de objetos de, 119–120
 superclasses e, 121–124
 visão geral de, 115

H

`hashCode()`, 94
`HashMap` classe, 108–109, 139–141
`HashSet` classe, 107–108, 138–140

Herança
 hierarquias paralelas e, 31–33
 uso sensato de, 22–23

Hierarquias
 herança e hierarquias paralelas, 31–33
 interfaces e hierarquias de classes, 26–27
 nomes de subclasses e, 24
 organizando classes em, 22–23

I

`If/then`, comandos de comportamento específico de instância, 34–35

Implementação
 comunicando a estratégia de implementação, 79
 estilos de uso de frameworks, 120–121
 intenção comparada a, 69
 medição de desempenho, 131

Implementadores, 32–33

Impressão para depuração, método de, 85–87
`indexOf()`, 110–112

Inicialização

ansiosa (*eager initialization*), 59–60
 de variáveis, 59–60
 preguiçosa (*lazy initialization*), 60–61

Inovação, simplicidade e, 11–12

Instanciação, estilos de uso de frameworks, 120

Instâncias

classes internas anônimas e, 39–40
 comportamento específico de instância, 33–35
 condicionais e, 34–37
 delegação e, 36–37
 seletores plugáveis e, 38–40

Intenção

coleções para comunicação, 99
 comunicação via código, 1
 mensagens explicativas para esclarecimento, 69
 nomes reveladores de intenção para métodos, 79–80

Interfaces

abstratas, 24–25
 benefícios e nomeação, 25–26
 framework, 121–122
 procedural vs. funcional, 28–29
 versões para extensão, 26–28
 visão geral de, 25–27

Interfaces para coleções
`array`, interface de, 103–104
`collection`, interface de, 103–104
`Iterable`, interface, 103–104
`list`, interface de, 103–105
`map`, interface de, 105–106
`set`, interface de, 104–105
`sorted set`, interface de, 104–106
 visão geral, 103

Interfaces versionadas
 interfaces em frameworks, 121–122
 visão geral de, 26–28

Invocação vs. acesso, 45
`Iterable`, interface de coleções, 103–104

J

Java
 como linguagem tipada, 58–59
 leituras sugeridas, 145–146
 objetos referenciados por variáveis, 49–51
`JUnit`, 121

L

Laços
 cláusulas de guarda e, 70–71
 estruturação, 5–6

Legibilidade de código
 melhorando simetria, 67–68
 mensagens de escolha e, 66

Leitores de código, métodos e, 78

Linguagens tipadas, 58–59

`LinkedHashMap`, classe, 108–109, 139–141

`LinkedHashSet`, classe, 107–109, 138–140

`LinkedList`, classe, 107–108, 137–139

`List`, classe de coleções, 107–108

`List`, interface de coleções, 103–105

Literate Programming (Knuth), 10

Lógica
 agregação em classes, 22–23
 comunicando variação em, 99
 interação entre similaridades e diferenças em programação, 30–32
 mensagens para expressar, 65–66
 métodos para dividir, 75
 objetos parâmetro para encaixe, 56
 princípio de alocar de lógica e dados juntos, 13–15

M

`Map`, classe
 coleções, 108–110
 comparando `HashMap`, `LinkedHashMap` e `TreeMap`, 139–141
`Map`, interface de coleções, 105–106

- Matemática**
- conjuntos matemáticos como metáfora para coleções, 101
 - tipos primitivos (Java) e, 27–29
- Medição de desempenho**, 129–141
- API para, 130–131
 - cancelar sobre carga de invocação de métodos, 134
 - comparando ArrayList e LinkedList, 137–139
 - comparando coleções, 135–137
 - comparando Maps, 139–141
 - comparando Sets, 138–140
 - exemplo, 129–130
 - implementando, 131
 - MethodTimer, objeto auxiliar, 132–134
 - testes, 134
 - visão geral de, 129
- Mensagens**
- como mecanismo de fluxo de controle, 65–66
 - convidativa, 68–69
 - de decomposição, 67
 - de escolha, 65–66
 - envio duplo, 66–67
 - explicativa, 69
 - inversa, 67–68
- Mensagens polimórficas**, 32–33
- tipos primitivos (Java), 27–29
- Metadados**, métodos get e, 94
- Metáforas de coleções**, 100–101
- MethodsTimer**, classe, 130
- MethodTimer**, objeto auxiliar, 132–134
- Métodos**, 75–97
- acessadores, *ver* Métodos acessadores
 - auxiliar, *ver* Métodos auxiliares
 - comentário de método, padrão, 84–85
 - composto, padrão, 77–79
 - construtor completo, padrão, 88–90
 - construtor de conversão, padrão, 87–88
 - conversão, padrão, 86–87
 - cópia de segurança, padrão, 95–97
 - criação, padrão, 87–88
 - de atribuição booleana, padrão, 92
 - de consulta, padrão, 92–93
 - de conversão, padrão, 86–88
 - de igualdade, padrão, 93–94
 - de impressão para depuração, padrão, 85–87
 - fábrica, *ver* Métodos fábrica
 - framework, 125–127
 - get, padrão, 94–95
 - níveis de visibilidade, 79–81
 - nome revelador de intenção, padrão, 79–80
 - objeto método, padrão, 81–83
 - questões na representação de framework como objeto, 120
 - set, padrão, 95–96
 - sobre carregado, padrão, 82–84
 - sobrescrito, padrão, 82–83
- tipo de retorno de método, padrão, 83–84
 - visão geral de, 75–77
- Métodos acessadores**
- acesso indireto com, 46–47
 - coleções, 90–92
- Métodos auxiliares**
- mensagens explicativas invocando, 69
 - MethodTimer, objeto auxiliar, 132–134
 - visão geral de, 84–86
- Métodos fábrica**
- criando frameworks, 124–125
 - fábricas internas, 90–91
 - visão geral de, 89–90
- Motivação, economia e fatores humanos na**, 19–20
- Mudança, estado e**, 43
- N**
- Nomes**
- classe, 23–24
 - interface, 25–26
 - mensagens de decomposição, 67
 - método, 79–80
 - subclasse, 24
 - variável, 50–51, 57–59
- O**
- Objetos**
- comparações de igualdade, 93–94
 - comportamento e estado de, 43
 - conversão de, 86–87
 - criando, 87–88
 - métodos de consulta, 92–93
 - métodos fábrica para criação, 89–90
 - métodos get, 94–95
 - objetos fábrica, 124–125
 - objetos método, 81–83
 - objetos no estilo valor, 27–31
 - referenciados por variáveis, 49–51
 - representando frameworks como, 119–120
- Objetos parâmetro**, 56–57
- Opcionais, parâmetros**, 55
- Ordem, em coleções**, 101
- Ordenando coleções**, 111–112
- overheadTimer()**, 134
- P**
- Pacote, níveis de visibilidade de métodos**, 79–81
- Pacotes, classes organizadas como**, 75
- Pacotes internos, desenvolvimento de framework e**, 122–123
- Padrões**
- forças e, 5–6
 - relacionamentos de valores e princípios, 9
 - visão geral de, 5–7

- Papel auxiliar de campos, 52–53
 Papel coletor, variáveis locais, 50–51
 Papel explicação, variáveis locais, 50–52
 Papel reúso, variáveis locais, 51–52
Parâmetros
 coletores, 54–55
 objetos parâmetro, 56–57
 opcionais, 55
 para comunicar estado, 52–55
 var args, 55–56
Princípio de consequências locais, 12–14
Princípio de minimização de repetição em escrever código, 13–14
Princípios, programação
 alocar lógica e dados juntos, 13–15
 como ponte entre valores e padrões, 9
 consequência locais de mudanças no código, 12–14
 expressão declarativa, 15–16
 minimizando repetição, 13–14
 simetria de código, 14–16
 taxa de mudança organizada para simetria temporal, 16–17
 visão geral de, 12–13
Programação
 comunicando via código, 1
 criando código legível, xi
 estilo procedural vs. estilo funcional, 27–29
 leis comuns a, 5
 leituras sugeridas, 143–145
Programação imperativa, 15–16
Programação orientada a objetos
 classes e, 21
 estado e, 45
- R**
- Refatorando código, 81–83
Retrocompatibilidade, complexidade em framework e, 116
Reusando código, 75
Reverse(list), ordenando coleções, 111–112
- S**
- Seletores plugáveis, 38–40
Set, classe
 coleções, 107–109
 comparando HashSet, LinkedHashSet e TreeSet, 138–140
Set, interface de coleções, 104–105
Shuffle(list), ordenando coleções, 111–112
Simetria
 inicialização ansiosa e, 59–60
 legibilidade de código e, 67–68
 princípios de programação, 14–16
 taxa de mudança organizada para simetria temporal, 16–17
 Similaridades, interação entre similaridades e diferenças em programação, 30–31
Simplicidade
 como valor de programação, 11–12
 exceções e, 72
Software
 economia de projeto de software, 19–20
 interfaces e, 24–25
Sort(list), 111–112
Sorted set, interface de coleções, 104–106
Static, métodos, 80–81
Structured Design (Yourdon and Constantine), 19
Subclasses
 nomes, 24
 usos e limitações de, 31–33
 visão geral, 31–33
Superclasses
 classes abstratas como, 25–27
 framework, 121–124
 métodos, 82–83
 subclasses e, 31–32
Switch, comandos de comportamento específico de instância, 34–35
- T**
- Tamanho, em coleções, 101
Taxa de mudança, organizada para simetria temporal, 16–17
Tempo de vida de variáveis, 50–51
Teoria da programação, 9–17
 alocando lógica e dados juntos, 13–15
 comunicação, 10–11
 consequências locais de mudanças no código, 12–14
 expressão declarativa, 15–16
 flexibilidade, 11–13
 leituras sugeridas, 144–146
 minimizando repetição, 13–14
 princípios, 12–13
 simetria do código, 14–16
 simplicidade, 11–12
 taxa de mudança organizada para simetria temporal, 16–17
 valores consistentes com excelência em programação, 10
 visão geral, 9–10
The Smalltalk Best Practice Patterns (Beck), 20
The Visual Display of Quantitative Information (Tufte), 11
 Tipos de retorno de métodos, 83–84
 Tomada de decisão, padrões que ajudam na, 6
toString(), 85–87

TreeMap, classe, 107–108, 139–141
TreeSet, classe, 108–109, 138–140

V

Valores, programação
 comunicação, 10–11
 flexibilidade, 11–13
 simplicidade, 11–12
 temas universais em programação, 9
 visão geral de, 10
Var args, 55–56
Variações
 coleções para comunicar variação de número, 99
 especialização e, 30–31
 métodos sobrescritos para expressar, 82–83

Variáveis
 constantes, 57
 inicialização de, 59–60
 inicialização preguiçosa (lazy initialization) de, 60–61
 locais, 50–52
 multivaloradas, 100
 nomes, 57–59
 tipos declarados, 58–59
 visão geral de, 49–51
Variáveis locais, 50–52
Visibilidade
 de métodos, níveis, 79–81
 desenvolvimento de frameworks e, 122–123
void, tipo de retorno em Java, 83–84
Von Neumann, John, 63

Índice de padrões

Acesso	45	Iterable.....	103
Acesso direto.....	46	List.....	104, 107
Acesso indireto.....	46	Map	106, 109
Array.....	103	Mensagem.....	65
Busca.....	110	Mensagem convidativa.....	68
Campo	52	Mensagem de decomposição (de sequenciamento)	67
Classe.....	22	Mensagem de escolha.....	65
Classe abstrata	26	Mensagem explicativa.....	69
Classe interna	33	Mensagem inversa.....	67
Classe interna anônima	39	Método acessador de coleção	90
Cláusula de guarda.....	70	Método auxiliar	85
Coleções com apenas um elemento.....	112	Método composto	77
Coleções imutáveis	111	Método de atribuição booleana.....	92
Coleções vazias	112	Método de consulta.....	92
Collection.....	104	Método de conversão	87
Comentário de método.....	84	Método de igualdade.....	93
Comportamento específico de instância.....	34	Método de impressão para depuração	86
Condicional.....	34	Método fábrica	89
Constante	57	Método get	94
Construtor completo	88	Método set	95
Construtor de conversão	87	Método sobrecarregado	83
Conversão	86	Método sobreescrito	83
Cópia de segurança	96	Nome qualificado de subclasse	24
Criação	88	Nome revelador de intenção.....	79
Delegação.....	36	Nome simples de superclasse	23
Envio duplo.....	66	Nome sugestivo de função.....	57
Especialização	30	Objeto método	81
Estado	44	Objeto parâmetro.....	56
Estado comum	47	Objeto valor	28
Estado extrínseco	49	Ordenação	110
Estado variável	48	Parâmetro	53
Exceção.....	72	Parâmetro coletor.....	54
Exceções verificadas	72	Parâmetro opcional	55
Fábrica interna	90	Propagação de exceção.....	73
Fluxo de controle	64	Seletor plugável	38
Fluxo excepcional	69	Set	104, 108
Fluxo principal.....	64	SortedSet	105
Implementador.....	32	Subclasse	31
Inicialização	59	Tipo de retorno de método	84
Inicialização ansiosa.....	60	Tipo declarado	59
Inicialização preguiçosa.....	60	Var args (quantidade variável de argumentos)	55
Interface	25	Variável	50
Interface abstrata	24	Variável local.....	50
Interface versionada	27	Visibilidade de método	80