



PRACTICAL OBJECT-ORIENTED DESIGN IN RUBY

AN AGILE PRIMER

SANDI METZ

Foreword by Obie Fernandez

Elogios ao design prático orientado a objetos em Ruby

"Isso é ótimo! Suas descrições são tão vibrantes e vívidas que estou redescobrindo a verdade enterrada nos princípios OO que, de outra forma, são tão internalizados que me esqueço de explorá-los. Seus pensamentos sobre design e conhecimento do futuro são especialmente eloquentes."

—Ian McFarland, presidente, New Context, Inc.

"Como um programador autodidata, este foi um mergulho extremamente útil em alguns conceitos de POO com os quais eu definitivamente gostaria de me familiarizar melhor! E não estou sozinho: há uma placa afixada no trabalho que diz: "WWSMD? - O que Sandi Metz faria"?

—Jonathan Mukai, pivô em Nova York

"Meticulosamente pragmático e primorosamente articulado, o Practical Object Oriented Design em Ruby disponibiliza conhecimento que de outra forma seria evasivo para um público que dele precisa desesperadamente. As prescrições são apropriadas tanto como regras para iniciantes quanto como diretrizes para profissionais experientes."

—Katrina Owen, desenvolvedora, Bengler

"Eu acredito que este será o livro Ruby mais importante de 2012. Além de o livro ser 100% correto, Sandi tem um estilo de escrita fácil, com muitas analogias excelentes que esclarecem todos os pontos."

—Avdi Grimm, autor de *Exceptional Ruby and Objects on Rails*

"Embora Ruby seja uma linguagem orientada a objetos, pouco tempo é gasto na documentação sobre o que OO realmente significa ou como deve direcionar a maneira como construímos programas. Aqui, Metz traz isso à tona, cobrindo a maioria dos princípios-chave do desenvolvimento e design OO de uma maneira envolvente e fácil de entender. Esta é uma obrigação para qualquer estante Ruby respeitável."

—Peter Cooper, editor, *Ruby Weekly*

"Tão bom que não consegui largar! Esta é uma leitura obrigatória para quem deseja fazer programação orientada a objetos em qualquer linguagem, sem mencionar que mudou completamente a maneira como abordo os testes."

—Charles Max Wood, apresentador de programa de vídeo e áudio, TeachMeToCode.com

"Destilar práticas assustadoras de design OO com exemplos e explicações claras torna este livro um livro tanto para novatos quanto para especialistas. Vale a pena o estudo por qualquer pessoa interessada em que o design OO seja feito de maneira correta e 'leve'. Gostei muito deste livro."

—Manuel Pais, editor, InfoQ.com

"Se você se considera um programador Ruby, deveria ler este livro. Ele está repleto de ótimos conselhos práticos e técnicas de codificação que você pode começar a aplicar imediatamente em seus projetos."

—Ylan Segal, grupo de usuários Ruby de San Diego

"Este é o melhor livro sobre OO que já li. É curto, doce, mas potente. Ele passa lentamente de técnicas simples para mais avançadas, cada exemplo melhorando o anterior. As ideias apresentadas são úteis não apenas em Ruby, mas também em linguagens estáticas como C#. Altamente recomendado!"

—Kevin Berridge, gerente de engenharia de software, Pointe Blank

Soluções e organizador, Burning River Developers Meetup

“O livro é simplesmente perfeito! A elegância do Ruby brilha, mas também funciona como um A a Z da programação orientada a objetos em geral.”

—Emil Rondahl, consultor de C# e .NET

“Este é um livro excepcional sobre Ruby, no qual Metz oferece uma visão prática de como escrever código idiomático, limpo e sustentável em Ruby. Absolutamente fantástico, recomendado para meus amigos hackers Ruby.”

—Zachary “Zee” Spencer, freelancer e treinador

“Este é o melhor livro de programação que li em anos. Sandi fala sobre princípios básicos, mas provavelmente ainda estamos fazendo coisas erradas e ela nos mostra por que e como. O livro tem a combinação perfeita de código, diagramas e palavras. Não posso recomendá-lo o suficiente e se você realmente quer ser um programador melhor, você lerá e concordará.

—Derick Hitchcock, desenvolvedor sênior, SciMed Solutions

“Prevejo que isso se tornará um clássico. Tenho uma familiaridade desconfortável com a literatura de programação, e este livro está em um nível completamente diferente. Fico surpreso quando encontro um livro que oferece novos insights e ideias, e ainda mais surpreso quando consigo fazê-lo, não apenas uma vez, mas ao longo das páginas. Este livro é excelentemente escrito, bem organizado, com explicações lúcidas de conceitos técnicos de programação.”

—Han S. Kang, engenheiro de software e membro do LA Rubyists

“Você deveria ler este livro se você ganha a vida escrevendo software. Os futuros desenvolvedores que herdarem seu código agradecerão.”

—Jose Fernandez, engenheiro de software sênior da New Relic

“A abordagem de Metz sobre o assunto está fortemente enraizada na teoria, mas a explicação sempre permanece fundamentada em preocupações do mundo real, o que me ajudou a internalizá-la. O livro é claro e conciso, mas atinge um tom mais amigável do que conciso.”

—Alex Strasheim, administrador de rede, Ensemble Travel Group

“Este é um livro incrível sobre como pensar orientado a objetos quando você está programando em Ruby. Embora existam alguns capítulos que são mais específicos para Ruby, este livro pode ser um ótimo recurso para desenvolvedores em qualquer linguagem. Resumindo, não posso recomendar este livro o suficiente.”

—James Hwang, thriceprime.com

“Quer você esteja apenas começando em sua carreira de desenvolvimento de software ou já esteja programando há anos (como eu), é provável que aprenda muito com o livro da Sra. Metz. Ela faz um trabalho fantástico ao explicar os porquês de um software bem projetado, juntamente com os comois.”

—Gabe Hollombe, artesão de software, avantbard.com

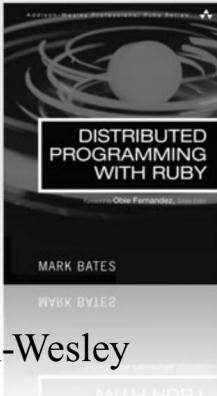
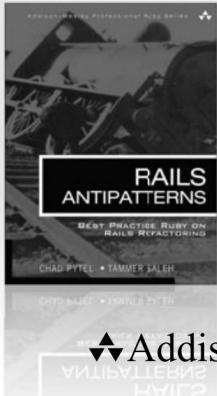
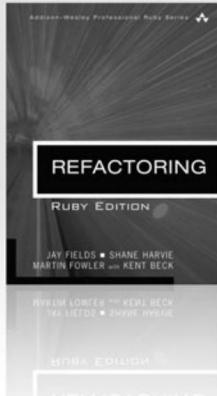
“Resumindo, este está entre os cinco melhores livros de programação que já li. Acredito que daqui a vinte anos este será considerado um dos trabalhos definitivos sobre programação orientada a objetos. Pretendo relê-lo pelo menos uma vez por ano para evitar que minhas habilidades caiam na atrofia. Se você é um desenvolvedor OO relativamente novo, intermediário ou até mesmo avançado em qualquer linguagem, comprar este livro é a melhor maneira que conheço de aprimorar suas habilidades de design OO.”

—Brandon Hays, desenvolvedor de software freelancer

PRÁTICO ORIENTADO A OBJETOS DESIGN EM RUBI

Addison-Wesley Série Ruby Profissional

Obie Fernandez, editor da série



▼ Addison-Wesley

Visite informit.com/ruby para obter uma lista completa dos produtos disponíveis.

Addison-Wesley Professional Ruby Series oferece aos leitores

O com informações práticas, orientadas para as pessoas e aprofundadas sobre aplicando a plataforma Ruby para criar soluções tecnológicas dinâmicas. A série baseia-se na premissa de que a necessidade de livros de referência especializados, escritos por profissionais experientes, nunca será satisfeita apenas por blogs e pela Internet.



PRÁTICO ORIENTADO A OBJETOS DESIGN EM RUBI

Uma cartilha ágil

Sandi Metz

▲ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianápolis • São Francisco
Nova York • Toronto • Montreal • Londres • Munique • Paris • Madri

Cidade do Cabo • Sydney • Tóquio • Singapura • Cidade do México

Muitas das designações utilizadas pelos fabricantes e vendedores para distinguir os seus produtos são reivindicadas como marcas comerciais. Quando essas designações aparecem neste livro, e o editor estava ciente de uma reivindicação de marca registrada, as designações foram impressas com letras iniciais maiúsculas ou em letras maiúsculas.

O autor e o editor tomaram cuidado na preparação deste livro, mas não oferecem nenhuma garantia expressa ou implícita de qualquer tipo e não assumem nenhuma responsabilidade por erros ou omissões. Nenhuma responsabilidade será assumida por danos incidentais ou consequenciais relacionados ou decorrentes do uso das informações ou programas aqui contidos.

A editora oferece excelentes descontos neste livro quando solicitado em grandes quantidades para compras em grandes quantidades ou vendas especiais, que podem incluir versões eletrônicas e/ou capas personalizadas e conteúdo específico para o seu negócio, objetivos de treinamento, foco de marketing e interesses de marca. Para mais informações por favor entre em contato:

Vendas Corporativas e Governamentais dos
EUA (800) 382-3419
corpsales@pearsontechgroup.com

Para vendas fora dos Estados Unidos, entre em contato com:

Vendas Internacionais
international@pearson.com

Visite-nos na Web: informit.com/aw

Dados de catalogação na publicação da Biblioteca do Congresso

METZ, Sandi.

Design prático orientado a objetos em Ruby: uma cartilha ágil / Sandi Metz.

pág. cm.

Inclui referências bibliográficas e índice.

ISBN 0-321-72133-0 (papel alcalino)

1. Programação orientada a objetos (Ciéncia da Computação) 2. Ruby (Linguagem de programas de computador) I. Título.

QA76.64.M485 2013

005.1'17—dc23

2012026008

Direitos autorais © 2013 Pearson Education, Inc.

Todos os direitos reservados. Impresso nos Estados Unidos da América. Esta publicação é protegida por direitos autorais e a permissão deve ser obtida do editor antes de qualquer reprodução proibida, armazenamento em um sistema de recuperação ou transmissão de qualquer forma ou por qualquer meio, eletrônico, mecânico, fotocópia, gravação ou similar. Para obter permissão para usar o material deste trabalho, envie uma solicitação por escrito para Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, ou envie sua solicitação por fax para (201) 236-3290 .

ISBN-13: 978-0-321-72133-4 ISBN-10:

Texto 0-321-72133-0

impresso nos Estados Unidos na RR Donnelley em Crawfordsville, Indiana.

Segunda impressão, abril de 2013

Editor chefe

Mark Taub

Editor de Aquisições

Debra Williams Cauley

Editor de Desenvolvimento

Michael Thurston

Editor chefe

John Fuller

Editor de Projeto

Elizabeth Ryan

Empacotador

Palavras laser

Editor de cópia

Phyllis Crittenden

Indexadora Constance A. Angelo

Revisor

Gina Delaney

Coordenador de Publicação

Kim Boedigheimer

Designer de capa

Chuti Prasertsith

Compositor

Palavras laser

Para Amy, que leu tudo primeiro

Esta página foi intencionalmente deixada em branco

Conteúdo

Prefácio xv

Introdução xvii

Agradecimentos xxi

Sobre o Autor XXIII

1 Design Orientado a Objetos 1

Em louvor ao design 2

O design do problema resolve 2

Por que mudar é difícil 3

Uma definição prática de design 4

As ferramentas do design 4

Princípios de Design 5

Padrões de Projeto 6

O Ato de Design 7

Como o design falha 7

Quando projetar 8

Julgando o Design 10

Uma Breve Introdução à Programação Orientada a Objetos 11

Linguagens Processuais 12

Linguagens Orientadas a Objetos 12

Resumo 14

2 Projetando Classes com uma Única Responsabilidade 15

Decidindo o que pertence a uma classe 16

Agrupando Métodos em Classes 16

Organizando o código para permitir alterações fáceis 16

Criando Classes que Têm uma Responsabilidade Única	17
Um Exemplo de Aplicação: Bicicletas e Engrenagens	17
Por que a responsabilidade única é importante	21
Determinando se uma classe tem uma única responsabilidade	22
Determinando quando tomar decisões de projeto	22
Escrevendo código que abraça a mudança	24
Depende do comportamento, não dos dados	24
Aplicar responsabilidade única em todos os lugares	29
Finalmente, a Roda Real	33
Resumo	34

3 Gerenciando Dependências 35

Compreendendo Dependências	36
Reconhecendo Dependências	37
Acoplamento entre objetos (CBO)	37
Outras Dependências	38
Escrevendo código fracamente acoplado	39
Injetar Dependências	39
Isolar Dependências	42
Remover dependências de ordem de argumento	46
Gerenciando a Direção de Dependência	51
Revertendo Dependências	51
Escolhendo a direção da dependência	53
Resumo	57

4 Criando Interfaces Flexíveis 59

Noções básicas sobre interfaces	59
Definindo Interfaces	61
Interfaces Públicas	62
Interfaces Privadas	62
Responsabilidades, Dependências e Interfaces	62
Encontrando a Interface Pública	63
Um exemplo de aplicação: Bicycle Touring Company	63
Construindo uma Intenção	64
Usando Diagramas de Sequência	65

Perguntar “O quê” em vez de dizer “Como”	69
Buscando Independência de Contexto	71
Confiando em Outros Objetos	73
Usando Mensagens para Descobrir Objetos	74
Criando um aplicativo baseado em mensagens	76
Escrevendo código que apresenta sua melhor (inter) face	76
Criar Interfaces Explícitas	76
Honre as Interfaces Públicas de Outros	78
Tenha cuidado ao depender de interfaces privadas	79
Minimizar o Contexto	79
A Lei de Deméter	80
Definindo Deméter	80
Consequências das Violações	80
Evitando Violações	82
Ouvindo Deméter	82
Resumo	83

5 Reduzindo custos com Duck Typing 85

Compreendendo a digitação de pato	85
Com vista para o Pato	87
Complicando o Problema	87
Encontrando o Pato	90
Consequências da digitação de pato	94
Escrevendo código que depende de patos	95
Reconhecendo Patos Escondidos	96
Depositando confiança em seus patos	98
Documentando tipos de pato	98
Compartilhando código entre patos	99
Escolhendo seus patos com sabedoria	99
Vencendo o medo de pato digitando	100
Subvertendo tipos de pato com digitação estática	100
Digitação estática versus dinâmica	101
Abraçando a digitação dinâmica	102
Resumo	104

6 Adquirindo comportamento por meio de herança 105

- Compreendendo a herança clássica 105
- Reconhecendo onde usar a herança 106
 - Começando com uma aula de concreto 106
 - Incorporando Vários Tipos 109
 - Encontrando os tipos incorporados 111
 - Escolhendo Herança 112
 - Desenhando Relacionamentos de Herança 114
- Aplicação incorreta de herança 114
- Encontrando a Abstração 116
 - Criando uma Superclasse Abstrata 117
 - Promovendo Comportamento Abstrato 120
 - Separando o Resumo do Concreto 123
 - Usando o método de modelo padrão 125
 - Implementando cada método de modelo 127
- Gerenciando o acoplamento entre superclasses e subclasses 129
 - Compreendendo o Acoplamento 129
 - Desacoplando Subclasses Usando Mensagens Hook 134
- Resumo 139

7 Compartilhando Comportamento de Função com Módulos 141

- Compreendendo as Funções 142
 - Encontrando Funções 142
 - Organizando Responsabilidades 143
 - Removendo Dependências Desnecessárias 145
 - Escrevendo o Código Concreto 147
 - Extraindo a Abstração 150
 - Procurando Métodos 154
 - Herdando Comportamento de Papel 158
- Escrevendo Código Herdável 158
 - Reconhecer os Antipadrões 158
 - Insista na Abstração 159
 - Honrar o Contrato 159

Use o método de modelo padrão 160
 Desacoplar preventivamente classes 161
 Criar Hierarquias Superficiais 161

Resumo 162

8 Combinando Objetos com Composição 163

 Compondo uma bicicleta com peças 164
 Atualizando a Classe de Bicicleta 164
 Criando uma hierarquia de peças 165
 Compondo o Objeto Peças 168
 Criando uma Parte 169
 Tornando o objeto Parts mais parecido com um array 172
 Fabricação de Peças 176
 Criando a PartsFactory 177
 Aproveitando o PartsFactory 178
 A Bicicleta Composta 180
 Decidindo entre herança e composição 184
 Aceitando as Consequências da Herança 184
 Aceitando as Consequências da Composição 187
 Escolhendo Relacionamentos 188
Resumo 190

9 Projetando Testes Econômicos 191

 Teste Intencional 192
 Conhecendo suas intenções 193
 Sabendo o que testar 194
 Saber quando testar 197
 Sabendo como testar 198
 Testando Mensagens Recebidas 200
 Excluindo Interfaces Não Utilizadas 202
 Provando a Interface Pública 203
 Isolando o Objeto em Teste 205
 Injetando Dependências Usando Classes 207
 Injetando Dependências como Funções 208

Testando Métodos Privados	213
Ignorando Métodos Privados Durante os Testes	213
Removendo métodos privados da classe em teste	214
Escolhendo testar um método privado	214
Testando Mensagens de Saída	215
Ignorando Mensagens de Consulta	215
Provando Mensagens de Comando	216
Testando tipos de pato	219
Testando Funções	219
Usando testes de função para validar duplas	224
Testando Código Herdado	229
Especificando a Interface Herdada	229
Especificando Responsabilidades da Subclasse	233
Testando Comportamento Único	236
Resumo	240

Posfácio 241

Índice 243

Prefácio

Um dos principais truismos do desenvolvimento de software é que, à medida que seu código cresce e os requisitos para o sistema que você está construindo mudam, será adicionada lógica adicional que ainda não está presente no sistema atual. Em quase todos os casos, a capacidade de manutenção ao longo da vida do código é mais importante do que otimizar o seu estado atual.

A promessa de usar design orientado a objetos (OO) é que seu código será mais fácil de manter e evoluir do que de outra forma. Se você é novo em programação, como desvendar esses segredos da manutenção usando OO? O fato é que muitos de nós nunca tivemos treinamento holístico para escrever código limpo e orientado a objetos; em vez disso, aprendemos nossas técnicas por osmose com colegas e uma infinidade de livros mais antigos e fontes on-line. Ou se recebemos uma cartilha em OO durante a escola, isso foi feito em linguagens como Java ou C++. (Os sortudos aprenderam usando Smalltalk!)

O Design Prático Orientado a Objetos em Ruby de Sandi Metz cobre todos os conceitos básicos de OO usando a linguagem Ruby, o que significa que está pronto para inaugurar incontáveis Ruby e Orienta os recém-chegados para os próximos passos em seu desenvolvimento profissional como profissionais maduros gramas.

O próprio Ruby, assim como o Smalltalk, é uma linguagem totalmente orientada a objetos (OO). Tudo nele, até mesmo construções de dados primitivos, como strings e números, é representado por objetos com comportamento. Ao escrever suas próprias aplicações em Ruby, você faz isso codificando seus próprios objetos, cada um encapsulando algum estado e definindo seu próprio comportamento. Se você ainda não tem experiência em OO, pode ser assustador saber como iniciar o processo. Este livro orienta você em cada passo do caminho, desde as questões mais básicas sobre o que colocar em uma aula, passando por conceitos básicos como o Princípio da Responsabilidade Única, até fazer compensações entre herança e composição, e descobrir como testar objetos isoladamente.

A melhor parte, porém, é a voz de Sandi. Ela tem muita experiência e é um dos membros da comunidade mais legais que você já conheceu, e acho que ela fez um ótimo trabalho

transmitir esse sentimento em sua escrita. Conheço Sandi há vários anos e me perguntei se seu manuscrito estaria à altura do prazer de realmente conhecer Sandi na vida real. Fico feliz em dizer que sim, e é por isso que estou feliz em recebê-la como nossa mais nova autora da Professional Ruby Series.

—Obie Fernandez, editor da série
Série Ruby Profissional Addison Wesley

Introdução

Queremos fazer o nosso melhor trabalho e queremos que o trabalho que fazemos tenha significado. E, sendo todo o resto igual, preferimos nos divertir ao longo do caminho.

Aqueles de nós cujo trabalho é escrever software têm uma sorte incrível. Construir software é um prazer sem culpa porque podemos usar nossa energia criativa para fazer as coisas. Organizamos nossas vidas para ter as duas coisas; podemos desfrutar do puro ato de escrever código com a certeza de que o código que escrevemos tem utilidade. Produzimos coisas que importam. Somos artesãos modernos, construímos estruturas que constituem a realidade atual e, não menos que pedreiros ou construtores de pontes, temos orgulho justificado das nossas realizações.

Todos os programadores compartilham isso, desde o novato mais entusiasmado até o mais velho aparentemente cansado, seja trabalhando na startup de Internet mais leve ou na empresa mais sóbria e há muito consolidada. Queremos fazer o nosso melhor trabalho. Queremos que nosso trabalho tenha significado. Queremos nos divertir ao longo do caminho.

E por isso é especialmente preocupante quando o software dá errado. Software ruim impede nosso propósito e interfere em nossa felicidade. Onde antes nos sentíamos produtivos, agora nos sentimos frustrados. Onde antes era rápido, agora é lento. Onde antes pacífico, agora frustrado.

Essa frustração ocorre quando custa muito caro fazer as coisas. Nossas calculadoras internas estão sempre funcionando, comparando o valor total realizado com o esforço total despendido. Quando o custo de realizar um trabalho excede o seu valor, nossos esforços parecem desperdiçados. Se programar dá alegria é porque nos permite ser úteis; quando se torna doloroso é um sinal de que acreditamos que poderíamos e deveríamos fazer mais. Nosso prazer segue os passos do trabalho.

Este livro é sobre como projetar software orientado a objetos. Não é um livro acadêmico, é uma história de programador sobre como escrever código. Ele ensina como organizar o software de modo a ser produtivo hoje e permanecer assim no próximo mês e no próximo ano. Ele mostra como escrever aplicações que possam ter sucesso no presente e ainda assim se adaptar ao

futuro. Ele permite que você aumente sua produtividade e reduza seus custos durante toda a vida útil de suas aplicações.

Este livro acredita em seu desejo de fazer um bom trabalho e fornece as ferramentas necessárias para melhor utilizá-lo. É totalmente prático e, como tal, é, em sua essência, um livro sobre como escrever código que lhe traga alegria.

Quem pode achar este livro útil?

Este livro pressupõe que você pelo menos tentou escrever software orientado a objetos. Não é necessário que você sinta que teve sucesso, apenas que tenha feito a tentativa em qualquer linguagem orientada a objetos (OO). O Capítulo 1 contém uma breve visão geral da programação orientada a objetos (OOP), mas seu objetivo é definir termos comuns, não ensinar programação.

Se você deseja aprender design OO (OOD), mas ainda não fez nenhuma programação orientada a objetos, pelo menos faça um tutorial antes de ler este livro. O design OO resolve problemas; sofrer desses problemas é quase um pré-requisito para compreender essas soluções. Programadores experientes podem pular esta etapa, mas a maioria dos leitores ficará mais feliz se escrever algum código OO antes de iniciar este livro.

Este livro usa Ruby para ensinar OOD, mas você não precisa conhecer Ruby para entender os conceitos aqui contidos. Existem muitos exemplos de código, mas todos são bastante simples. Se você programou em qualquer linguagem OO, achará Ruby fácil de entender.

Se você vem de uma linguagem OO de tipo estaticamente, como Java ou C++, você tem o conhecimento necessário para se beneficiar da leitura deste livro. O fato de Ruby ser tipado dinamicamente simplifica a sintaxe dos exemplos e destila as ideias de design em sua essência, mas cada conceito neste livro pode ser traduzido diretamente para uma linguagem OO tipada estaticamente.

Como ler este livro

O Capítulo 1, Design Orientado a Objetos, contém uma visão geral dos porquês, quando e porquês do design OO, seguido por uma breve visão geral da programação orientada a objetos. Este capítulo é independente. Você pode lê-lo primeiro, por último ou, francamente, ignorá-lo completamente, embora se você estiver preso a um aplicativo que sofre de falta de design, poderá achar que é uma história reconfortante.

Se você tem experiência em escrever aplicações orientadas a objetos e deseja começar imediatamente, você pode começar com segurança com o Capítulo 2. Se você fizer isso e então se deparar com um

termo desconhecido, volte e navegue na seção Introdução à Programação Orientada a Objetos do Capítulo 1, que apresenta e define termos OO comuns usados ao longo do livro.

Os Capítulos 2 a 9 explicam progressivamente o design orientado a objetos. O Capítulo 2, Projetando classes com uma única responsabilidade, aborda como decidir o que pertence a uma única classe. O Capítulo 3, Gerenciando Dependências, ilustra como os objetos ficam emaranhados uns com os outros e mostra como mantê-los separados. Esses dois capítulos enfocam objetos e não mensagens.

No Capítulo 4, Criando interfaces flexíveis, a ênfase começa a mudar do design centrado no objeto para o design centrado na mensagem. O Capítulo 4 trata da definição de interfaces e se preocupa em como os objetos se comunicam entre si. O Capítulo 5, Reduzindo Custos com Duck Typing, trata da digitação de patos e introduz a ideia de que objetos de classes diferentes podem desempenhar funções comuns. O Capítulo 6, Adquirindo comportamento por meio de herança, ensina as técnicas de herança clássica, que são então usadas no Capítulo 7, Compartilhando comportamento de função com módulos, para criar funções do tipo pato. O Capítulo 8, Combinando objetos com composição, explica a técnica de construção de objetos por meio de composição e fornece diretrizes para escolher entre composição, herança e compartilhamento de função do tipo pato. O Capítulo 9, Projetando testes com boa relação custo-benefício, concentra-se no design de testes, que ilustra usando código dos capítulos anteriores do livro.

Cada um desses capítulos baseia-se nos conceitos do anterior. Eles estão cheios de código e são melhor lidos em ordem.

Como usar este livro

Este livro significará coisas diferentes para leitores de diferentes origens. Aqueles que já estão familiarizados com o OOD encontrarão coisas em que pensar, possivelmente encontrarão alguns novos pontos de vista e provavelmente discordarão de algumas sugestões. Como não existe uma autoridade final sobre OOD, os desafios aos princípios (e a este autor) irão melhorar a compreensão de todos. No final, você deve ser o árbitro de seus próprios desígnios; cabe a você questionar, experimentar e escolher.

Embora este livro deva ser do interesse de leitores de vários níveis, ele foi escrito com o objetivo específico de ser acessível a novatos. Se você é um desses novatos, esta parte da introdução é especialmente para você. Saiba isto: o design orientado a objetos não é magia negra. São simplesmente coisas que você ainda não conhece. O fato de você ter lido até aqui indica que você se preocupa com design; esse desejo de aprender é o único pré-requisito para se beneficiar deste livro.

Os capítulos 2 a 9 explicam os princípios do OOD e fornecem regras de programação muito explícitas; essas regras significarão coisas diferentes para os novatos do que para os especialistas. Se você é um novato, comece seguindo estas regras com fé cega, se necessário. Esse a obediência precoce evitará o desastre até que você ganhe experiência suficiente para fazer suas próprias decisões. Quando as regras começarem a incomodar, você terá experiência suficiente crie suas próprias regras e sua carreira como designer terá começado.

Agradecimentos

É uma maravilha que este livro exista; o fato de isso acontecer se deve aos esforços e ao incentivo de muitas pessoas.

Ao longo do longo processo de escrita, Lori Evans e TJ Stankus forneceram feedback antecipado sobre cada capítulo. Eles moram em Durham, Carolina do Norte, e portanto não poderiam escapar de mim, mas esse fato não diminui meu apreço pela ajuda deles.

No meio do livro, depois que se tornou impossível negar que sua redação levaria aproximadamente o dobro do tempo inicialmente estimado, Mike Dalessio e Gregory Brown leram os rascunhos e deram feedback e apoio inestimáveis. Seu incentivo e entusiasmo mantiveram o projeto vivo durante os dias sombrios.

Quando estava quase concluído, Steve Klabnik, Desi McAdam e Seth Wax revisaram o livro e, assim, atuaram como graciosos substitutos para você, gentil leitor. Suas impressões e sugestões causaram mudanças que beneficiarão todos os que as seguirem.

Os últimos rascunhos receberam leituras cuidadosas e completas de Katrina Owen, Avdi Grimm e Rebecca Wirfs-Brock, e o livro foi muito melhorado por seu feedback gentil e atencioso. Antes de ajudarem, Katrina, Avdi e Rebecca eram estranhos para mim; Estou grato pelo seu envolvimento e humilde pela sua generosidade. Se você achar este livro útil, agradeça-lhes na próxima vez que os vir.

Também sou grato ao Gotham Ruby Group e a todos que expressaram seu apreço pelas palestras de design que dei na GoRuCo 2009 e 2011. O pessoal da GoRuCo arriscou-se com um desconhecido e me deu um fórum para expressar essas ideias; este livro começou aí. Ian McFarland e Brian Ford assistiram a essas palestras e o seu entusiasmo imediato e contínuo por este projeto foi contagioso e convincente.

O processo de escrita foi muito auxiliado por Michael Thurston, da Pearson Education, que era como um transatlântico de calma e organização navegando pelo mar caótico de minhas ondas opostas de escrita rebelde. Você pode, espero, ver o problema que ele enfrentou. Ele insistiu, com infinita paciência e graça, que a escrita fosse organizada em uma estrutura legível. Acredito que seus esforços valeram a pena e espero que você concorde.

Meus agradecimentos também a Debra Williams Cauley, minha editora na Addison-Wesley, que ouviu um discurso inoportuno no corredor em 2006, na primeira conferência Ruby on Rails em Chicago, e lançou a campanha que resultou neste livro. Apesar dos meus melhores esforços, ela não aceitaria um não como resposta. Ela habilmente passou de uma discussão para outra até que finalmente encontrou aquela que convenceu; isso reflete com precisão sua persistência e dedicação.

Tenho uma dívida com toda a comunidade de design orientado a objetos. Não fui eu que inventei as ideias deste livro, sou apenas um tradutor e estou sobre ombros de gigantes. Escusado será dizer que, embora todo o crédito por estas ideias pertença a outros, as falhas de tradução são exclusivamente minhas.

E, finalmente, este livro deve sua existência à minha parceira Amy Germuth. Antes de este projeto começar eu não conseguia imaginar escrever um livro; sua visão do mundo como um lugar onde as pessoas faziam essas coisas fazia com que isso parecesse possível. O livro em suas mãos é uma homenagem à sua paciência ilimitada e ao seu apoio infinito.

Obrigado a todos e cada um.

Sobre o autor

Sandi Metz tem 30 anos de experiência trabalhando em projetos que sobreviveram para crescer e mudar. Ela escreve código todos os dias como arquiteta de software na Duke University, onde sua equipe resolve problemas reais para clientes que possuem grandes aplicativos orientados a objetos que estão evoluindo há 15 anos ou mais. Ela está empenhada em disponibilizar software útil de maneiras extremamente práticas. *O Design Prático Orientado a Objetos em Ruby* é a destilação de muitos anos de desenhos em quadro branco e o culminar lógico de uma vida inteira de conversas sobre design OO. Sandi falou na Ruby Nation e diversas vezes na Gotham Ruby User's Conference e mora em Durham, NC.

Esta página foi intencionalmente deixada em branco

CAPÍTULO 1

Design Orientado a Objetos

O mundo é processual. O tempo flui e os eventos, um por um, passam. Seu procedimento matinal pode ser levantar, escovar os dentes, fazer café, vestir-se e depois começar a trabalhar. Estas atividades podem ser modeladas utilizando software processual; como você conhece a ordem dos eventos, você pode escrever código para fazer cada coisa e então deliberadamente encadear as coisas, uma após a outra.

O mundo também é orientado a objetos. Os objetos com os quais você interage podem incluir uma esposa e um gato, ou um carro velho e uma pilha de peças de bicicleta na garagem, ou seu coração batendo e o plano de exercícios que você usa para mantê-lo saudável. Cada um desses objetos vem equipado com seu próprio comportamento e, embora algumas das interações entre eles possam ser previsíveis, é perfeitamente possível que seu cônjuge pise inesperadamente no gato, causando uma reação que aumenta rapidamente a frequência cardíaca de todos e lhe dá novas sensações. apreciação pelo seu regime de exercícios.

Num mundo de objetos, novos arranjos de comportamento emergem naturalmente. Você não precisa escrever explicitamente o código para o procedimento `cônjuge_steps_on_cat`, tudo o que você precisa é de um objeto cônjuge que execute etapas e um objeto gato que não goste de ser pisado. Coloque esses dois objetos juntos em uma sala e combinações inesperadas de comportamento aparecerão.

Este livro trata do projeto de software orientado a objetos e vê o mundo como uma série de interações espontâneas entre objetos. O design orientado a objetos (OOD) exige que você deixe de pensar no mundo como uma coleção de procedimentos predefinidos e passe a modelar o mundo como uma série de mensagens que passam entre objetos. As falhas do OOD podem parecer falhas na técnica de codificação, mas na verdade são falhas de

perspectiva. O primeiro requisito para aprender a fazer design orientado a objetos é mergulhar nos objetos; uma vez que você adquire uma perspectiva orientada a objetos, o resto acontece naturalmente.

Este livro orienta você através do processo de imersão. Este capítulo começa com uma discussão geral sobre OOD. Ele defende o design e depois descreve quando fazê-lo e como julgá-lo. O capítulo termina com uma breve visão geral da programação orientada a objetos que define os termos usados ao longo do livro.

Em louvor ao design

O software é construído por um motivo. A aplicação alvo – seja um jogo trivial ou um programa para orientar a radioterapia – é o ponto principal. Se a programação dolorosa fosse a maneira mais econômica de produzir software funcional, os programadores seriam moralmente obrigados a sofrer estoicamente ou a encontrar outros empregos.

Felizmente, você não precisa escolher entre prazer e produtividade. As técnicas de programação que tornam a escrita do código um prazer se sobrepõem àquelas que produzem software com mais eficiência. As técnicas de design orientado a objetos resolvem os dilemas morais e técnicos da programação; segui-los produz software econômico usando código no qual também é um prazer trabalhar.

O design do problema resolve Imagine

escrever um novo aplicativo. Imagine que esta aplicação vem equipada com um conjunto completo e correto de requisitos. E se quiser, imagine mais uma coisa: uma vez escrita, esta aplicação nunca mais precisará ser alterada.

Neste caso, o design não importa. Como um artista de circo girando pratos em um mundo sem atrito ou gravidade, você poderia programar o aplicativo para entrar em movimento e depois recuar orgulhosamente e vê-lo funcionar para sempre. Por mais vacilantes que fossem, as placas de código giravam continuamente, oscilando e girando, mas nunca caindo.

Desde que nada tenha mudado.

Infelizmente, algo vai mudar. Sempre acontece. Os clientes não sabiam o que queriam, não disseram o que queriam dizer. Você não entendeu as necessidades deles, aprendeu a fazer algo melhor. Mesmo os aplicativos perfeitos em todos os aspectos não são estáveis. O aplicativo foi um grande sucesso, agora todo mundo quer mais. A mudança é inevitável. É onipresente, onipresente e inevitável.

Em louvor ao design

A mudança de requisitos é o equivalente de programação ao atrito e à gravidade.

Introduzem forças que aplicam pressões repentinhas e inesperadas que vão contra os planos mais bem elaborados. É a necessidade de mudança que faz com que o design seja importante.

Aplicativos fáceis de alterar são um prazer de escrever e uma alegria de estender.

Eles são flexíveis e adaptáveis. Os aplicativos que resistem à mudança são exatamente o oposto; toda mudança é cara e cada uma faz com que a próxima custe mais. Poucos aplicativos difíceis de alterar são agradáveis de trabalhar. Os piores deles gradualmente se tornam filmes de terror pessoais, onde você interpreta um programador infeliz, correndo loucamente de um prato giratório para outro, tentando evitar o som de louças quebrando.

Por que a mudança é difícil

Os aplicativos orientados a objetos são compostos de partes que interagem para produzir o comportamento do todo. As partes são *objetos*; as interações são incorporadas nas *mensagens* que passam entre eles. Levar a mensagem certa ao objeto de destino correto requer que o remetente da mensagem saiba coisas sobre o destinatário. Este conhecimento cria dependências entre os dois e essas dependências impedem a mudança.

O design orientado a objetos trata do *gerenciamento de dependências*. É um conjunto de técnicas de codificação que organizam dependências de modo que os objetos possam tolerar mudanças. Na ausência de design, as dependências não gerenciadas causam estragos porque os objetos sabem muito uns sobre os outros. Mudar um objeto força a mudança em seus colaboradores, que por sua vez, força a mudança em seus colaboradores, *ad infinitum*. Um aprimoramento aparentemente insignificante pode causar danos que se irradiam em círculos concêntricos sobrepostos, acabando por não deixar nenhum código intocado.

Quando os objetos sabem demais, eles têm muitas expectativas sobre o mundo em que residem. Eles são exigentes, precisam que as coisas sejam “exatas”. Essas expectativas os restringem. Os objetos resistem a serem reutilizados em diferentes contextos; eles são dolorosos de testar e suscetíveis de serem duplicados.

Em uma aplicação pequena, um design ruim é passível de sobrevivência. Mesmo que tudo esteja conectado a todo o resto, se você conseguir manter tudo na cabeça de uma vez, ainda poderá melhorar o aplicativo. O problema com pequenos aplicativos mal projetados é que, se forem bem-sucedidos, eles se transformam em grandes aplicativos mal projetados. Eles gradualmente se tornam poços de alcatrão nos quais você tem medo de pisar para não afundar sem deixar rastros. Mudanças que deveriam ser simples podem se espalhar pelo aplicativo, quebrando o código em todos os lugares e exigindo reescrita extensa. Os testes são apanhados no fogo cruzado e começam a parecer mais um obstáculo do que uma ajuda.

Uma definição prática de design Cada aplicativo é uma coleção de código; a organização do código é o *design*. Pode-se confiar que dois programadores isolados, mesmo quando compartilham ideias comuns sobre design, resolverão o mesmo problema organizando o código de maneiras diferentes. O design não é uma linha de montagem onde trabalhadores com treinamento semelhante constroem widgets idênticos; é um estúdio onde artistas com ideias semelhantes esculpem aplicações personalizadas. O design é, portanto, uma arte, a arte de organizar códigos.

Parte da dificuldade do design é que todo problema tem dois componentes. Você não deve apenas escrever o código para o recurso que planeja entregar hoje, mas também criar um código que possa ser alterado posteriormente. Por qualquer período que se estenda após a entrega inicial da versão beta, o custo da mudança acabará eclipsando o custo original do aplicativo. Como os princípios de design se sobrepõem e cada problema envolve uma mudança de prazo, os desafios de design podem ter um número desconcertante de soluções possíveis. Seu trabalho é de síntese; você deve combinar uma compreensão geral dos requisitos de sua aplicação com o conhecimento dos custos e benefícios das alternativas de design e, em seguida, elaborar um arranjo de código que seja econômico no presente e que continue a sê-lo no futuro.

Levar o futuro em consideração pode parecer introduzir uma necessidade de habilidades psíquicas normalmente consideradas fora do domínio da programação. Não tão. O futuro que o design considera não é aquele em que você antecipa requisitos desconhecidos e escolhe preventivamente um deles para implementar no presente. Programadores não são médiuns. Projetos que antecipam requisitos futuros específicos quase sempre terminam mal. O design prático não antecipa o que acontecerá com a sua aplicação, apenas aceita que algo acontecerá e que, no presente, você não pode saber o quê.

Não adivinha o futuro; preserva suas opções para acomodar o futuro. Não escolhe; isso deixa espaço para você se mover.

O objetivo do design é permitir que você faça o design *posteriormente* e seu objetivo principal é reduzir o custo da mudança.

As Ferramentas do Design Design

não é o ato de seguir um conjunto fixo de regras, é uma jornada ao longo de um caminho ramificado em que as escolhas anteriores fecham algumas opções e abrem o acesso a outras. Durante o projeto você percorre um labirinto de requisitos onde cada conjuntura representa um ponto de decisão que tem consequências para o futuro.

Assim como um escultor possui cinzeis e limas, um designer orientado a objetos possui ferramentas – princípios e padrões.

Princípios de design

O acrônimo SOLID, cunhado por Michael Feathers e popularizado por Robert Martin, representa cinco dos princípios mais conhecidos do design orientado a objetos: Responsabilidade Única, Aberto-Fechado, Substituição de Liskov, Segregação de Interface e Inversão de Dependência. Outros princípios incluem DRY (Don't Repeat Yourself) de Andy Hunt e Dave Thomas e a Lei de Demeter (LoD) do projeto Demeter da Northeastern University.

Os próprios princípios serão tratados ao longo deste livro; a questão por enquanto é “De onde eles vieram?” Existe prova empírica de que esses princípios têm valor ou são apenas a opinião de alguém que você pode desconsiderar livremente? Em essência, quem diz?

Todos esses princípios começaram como escolhas que alguém fez ao escrever o código. Os primeiros programadores OO notaram que alguns arranjos de código facilitavam suas vidas, enquanto outros os dificultavam. Essas experiências os levaram a desenvolver opiniões sobre como escrever um bom código.

Os acadêmicos acabaram se envolvendo e, precisando escrever dissertações, decidiram quantificar a “bondade”. Este desejo é louvável. Se pudéssemos contar coisas, isto é, calcular métricas sobre nosso código e correlacioná-las a aplicações de alta ou baixa qualidade (para as quais também precisamos de uma medida objetiva), poderíamos fazer mais coisas que reduzem custos e menos coisas que os criam. Ser capaz de medir a qualidade mudaria o design OO de uma opinião infinitamente contestada para uma ciência mensurável.

Na década de 1990, Chidamber, Kemerer¹ e Basili² fizeram exatamente isto. Eles pegaram aplicativos orientados a objetos e tentaram quantificar o código. Eles nomearam e mediram coisas como o tamanho geral das classes, os emaranhados que as classes têm entre si, a profundidade e a amplitude das hierarquias de herança e o número de métodos que são invocados como resultado de qualquer mensagem enviada. Eles escolheram arranjos de código que consideraram importantes, criaram fórmulas para contá-los e, em seguida, correlacionaram as métricas resultantes com a qualidade dos aplicativos incluídos. A pesquisa deles mostra uma correlação definitiva entre o uso dessas técnicas e código de alta qualidade.

1. Chidamber, SR e Kemerer, CF (1994). Um conjunto de métricas para design orientado a objetos. *IEEE Trans. Suave. Eng.* 20(6): 476–493.

2. Relatório Técnico Basili (1995). Univ. de Maryland, Dep. de Ciência da Computação, College Park, MD, 20742 EUA. Abril de 1995. *Uma Validação de Métricas de Design Orientado a Objetos como Indicadores de Qualidade.*

Embora esses estudos pareçam provar a validade dos princípios de design, eles trazem, para qualquer programador experiente, uma ressalva. Esses primeiros estudos examinaram aplicações muito pequenas escritas por estudantes de pós-graduação; isto por si só é suficiente para justificar a observação das conclusões com cautela. O código nessas aplicações pode não ser representativo de aplicações OO do mundo real.

No entanto, acontece que cautela é desnecessária. Em 2001, Laing e Coleman examinaram diversas aplicações do Goddard Space Flight Center da NASA (ciência de foguetes) com a intenção expressa de encontrar “uma maneira de produzir software mais barato e de maior qualidade”. tinha 1.617 classes e mais de 500.000 linhas de código. A sua investigação apoia os estudos anteriores e confirma ainda mais que os princípios de design são importantes.

Mesmo que você nunca tenha lido esses estudos, pode ter certeza de suas conclusões. Os princípios de um bom design representam verdades mensuráveis e segui-los melhorará seu código.

Padrões de Projeto Além

dos princípios, o projeto orientado a objetos envolve *padrões*. O chamado Gang of Four (Gof), Erich Gamma, Richard Helm, Ralph Johnson e Jon Vlissides, escreveram o trabalho seminal sobre padrões em 1995. Seu livro *Design Patterns* descreve padrões como “soluções simples e elegantes para problemas específicos”. em design de software orientado a objetos” que você pode usar para “tornar seus próprios projetos mais flexíveis, modulares, reutilizáveis e compreensíveis”.

A noção de padrões de design é incrivelmente poderosa. Nomear problemas comuns e resolvê-los de maneiras comuns coloca o que está confuso em foco. Os *Design Patterns* deram a toda uma geração de programadores os meios para se comunicar e colaborar.

Os padrões têm um lugar na caixa de ferramentas de todo designer. Cada padrão conhecido é uma solução de código aberto quase perfeita para o problema que resolve. No entanto, a popularidade dos padrões levou a uma espécie de abuso de padrões por parte de programadores novatos, que, num excesso de zelo bem-intencionado, aplicaram padrões perfeitamente bons aos problemas errados. A aplicação incorreta de padrões resulta em código complicado e confuso, mas esse resultado não é o mesmo.

3. Laing, Victor e Coleman, Charles. (2001). Componentes principais do objeto ortogonal
Métricas Orientadas (323-08-14).

4. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Padrões de Projeto, Elementos de Software Orientado a Objetos Reutilizáveis*. Nova York, NY: Addison-Wesley Publishing Company, Inc.

O Ato de Design

culpa do próprio padrão. Uma ferramenta não pode ser criticada pelo seu uso, o usuário deve dominá-la.

Este livro não trata de padrões; no entanto, irá prepará-lo para compreendê-los e fornecer-lhe o conhecimento para escolhê-los e usá-los adequadamente.

O Ato de Design Com a

descoberta e propagação de princípios e padrões de design comuns, todos os problemas de OOD pareceriam ter sido resolvidos. Agora que as regras subjacentes são conhecidas, quão difícil pode ser projetar software orientado a objetos?

Acontece que é muito difícil. Se você pensa no software como um mobiliário personalizado, então os princípios e padrões são como ferramentas para trabalhar madeira. Saber como o software deve ficar quando estiver pronto não faz com que ele se construa sozinho; os aplicativos surgem porque algum programador aplicou as ferramentas. O resultado final, seja um lindo gabinete ou uma cadeira frágil, reflete a experiência do programador com as ferramentas de design.

Como o design falha A

primeira maneira pela qual o design falha é devido à falta dele. Os programadores inicialmente sabem pouco sobre design. Contudo, isto não é um impedimento, pois é possível produzir aplicações funcionais sem saber nada sobre design.

Isso é verdade para qualquer linguagem OO, mas algumas linguagens são mais suscetíveis que outras e uma linguagem acessível como Ruby é especialmente vulnerável. Ruby é muito amigável; a linguagem permite que quase qualquer pessoa crie scripts para automatizar tarefas repetitivas, e uma estrutura opinativa como Ruby on Rails coloca aplicações web ao alcance de todos os programadores. A sintaxe da linguagem Ruby é tão suave que qualquer pessoa abençoada com a habilidade de encadear pensamentos em ordem lógica pode produzir aplicações funcionais. Programadores que não sabem nada sobre design orientado a objetos podem ter muito sucesso em Ruby.

Contudo, aplicações bem sucedidas mas *não concebidas* carregam as sementes da sua própria destruição; eles são fáceis de escrever, mas gradualmente tornam-se impossíveis de mudar. A experiência passada de um programador não prevê o futuro. A promessa inicial de desenvolvimento indolor falha gradualmente e o otimismo se transforma em desespero à medida que os programadores começam a receber cada solicitação de mudança com “Sim, posso adicionar esse recurso, *mas isso quebrará tudo*”.

Programadores um pouco mais experientes encontram diferentes falhas de design. Esses programadores estão cientes das técnicas de design OO, mas ainda não entendem

como aplicá-los. Com a melhor das intenções, esses programadores caem na armadilha do overdesign. Um pouco de conhecimento é perigoso; à medida que seu conhecimento aumenta e a esperança retorna, eles *projetam* incansavelmente. Num excesso de entusiasmo, aplicam princípios de forma inadequada e vêem padrões onde não existem. Eles constroem belos e complicados castelos de código e depois ficam angustiados ao se verem cercados por paredes de pedra. Você pode reconhecer esses programadores porque eles começam a receber solicitações de mudança com “Não, não posso adicionar esse recurso; *não foi projetado para fazer isso.*”

Finalmente, o software orientado a objetos falha quando o ato de projetar é separado do ato de programar. O design é um processo de descoberta progressiva que depende de um ciclo de feedback. Este ciclo de feedback deve ser oportuno e incremental; as técnicas iterativas do movimento de software Agile (<http://agilemanifesto.org/>) são, portanto, perfeitamente adequadas para a criação de aplicações OO bem projetadas. A natureza iterativa do desenvolvimento ágil permite que o design se ajuste regularmente e evolua naturalmente. Quando o design é ditado de longe, nenhum dos ajustes necessários pode ocorrer e falhas iniciais de compreensão ficam consolidadas no código. Os programadores que são forçados a escrever aplicativos projetados por especialistas isolados começam a dizer: “Bem, certamente posso escrever isso, *mas não é o que você realmente quer e acabará se arrependendo*”.

When to Design Agile

acredita que seus clientes não podem definir o software que desejam antes de vê-lo, então é melhor mostrá-los o quanto antes. Se essa premissa for verdadeira, segue-se logicamente que você deve criar software em pequenos incrementos, iterando gradualmente até chegar a um aplicativo que atenda à verdadeira necessidade do cliente. A Agile acredita que a maneira mais econômica de produzir o que os clientes realmente desejam é colaborar com eles, construindo software um pouco por vez, de modo que cada parte entregue tenha a oportunidade de alterar ideias sobre a próxima. A experiência Agile é que essa colaboração produz software diferente do que foi inicialmente imaginado; o software resultante não poderia ter sido previsto por nenhum outro meio.

Se o Agile estiver correto, duas outras coisas também são verdadeiras. Primeiro, não há absolutamente nenhum sentido em fazer um Big Up Front Design (BUFD) (porque não pode estar correto) e, segundo, ninguém pode prever quando a aplicação será concluída (porque você não sabe de antemão o que será). acabará por fazer).

Não deveria ser surpresa que algumas pessoas se sintam desconfortáveis com o Agile. “Não sabemos o que estamos fazendo” e “Não sabemos quando terminaremos” podem ser um

O Ato de Design

difícil vender. O desejo pelo BUFD persiste porque, para alguns, proporciona uma sensação de controle que de outra forma não existiria. Por mais reconfortante que seja esse sentimento, é uma ilusão temporária que não sobreviverá ao ato de redigir o requerimento.

O BUFD inevitavelmente leva a um relacionamento antagônico entre clientes e programadores. Como qualquer grande projeto criado antes do software funcional não pode ser correto, escrever a aplicação conforme especificado garante que ela não atenderá às necessidades do cliente. Os clientes descobrem isso quando tentam usá-lo. Eles então solicitam alterações. Os programadores resistem a essas mudanças porque têm um cronograma a cumprir, que muito provavelmente já estão atrasados. O projeto gradualmente se torna condenado à medida que os participantes deixam de trabalhar para torná-lo bem-sucedido e passam a se esforçar para evitar serem culpados pelo seu fracasso.

As regras deste compromisso são claras para todos. Quando um projeto perde o prazo de entrega, mesmo que isso tenha acontecido por causa de alterações na especificação, a culpa é dos programadores. Se, no entanto, for entregue no prazo, mas não atender à necessidade real, a especificação deve estar errada e a culpa é do cliente. Os documentos de design do BUFD começam como roteiros para o desenvolvimento de aplicações, mas gradualmente se tornam foco de divergências. Eles não produzem software de qualidade; em vez disso, fornecem palavras rigorosamente analisadas que serão invocadas no final, lutando contra a defesa contra ser a pessoa que acaba segurando a batata quente da culpa.

Se a insanidade é fazer a mesma coisa repetidamente e esperar resultados diferentes, o Manifesto Ágil foi onde começamos coletivamente a recuperar os sentidos. O Agile funciona porque reconhece que a certeza é inatingível antes da existência da aplicação; A aceitação dessa verdade pelo Agile permite que ele forneça estratégias para superar a desvantagem do desenvolvimento de software sem conhecer o objetivo nem o cronograma.

No entanto, só porque o Agile diz “não faça um grande design inicial” não significa que ele diz para você não fazer nenhum design. A palavra *design* quando usada no BUFD tem um significado diferente do que quando usada no OOD. O objetivo do BUFD é especificar completamente e documentar totalmente o funcionamento interno futuro previsto de todos os recursos do aplicativo proposto. Se houver um arquiteto de software envolvido, isso pode se estender à decisão antecipada de como organizar todo o código. OOD está preocupado com um domínio muito mais restrito. Trata-se de organizar o código que você possui para que seja fácil alterá-lo.

Os processos ágeis garantem mudanças e sua capacidade de fazer essas mudanças depende do design do seu aplicativo. Se você não conseguir escrever um código bem projetado, terá que reescrever seu aplicativo a cada iteração.

O Agile, portanto, não proíbe o design, ele o exige. Não requer apenas design, requer um design realmente bom. Ele precisa do seu melhor trabalho. Seu sucesso depende de código simples, flexível e maleável.

Julgando o design

Antigamente, os programadores às vezes eram julgados pelo número de linhas de código (chamadas de *linhas de código fonte* ou SLOC) que produziam. É óbvio como essa métrica surgiu; qualquer chefe que pense na programação como uma linha de montagem onde trabalhadores com formação semelhante trabalham para construir dispositivos idênticos pode facilmente desenvolver a crença de que a produtividade individual pode ser avaliada simplesmente pesando a produção. Para gerentes que precisavam desesperadamente de uma maneira confiável de comparar programadores e avaliar software, o SLOC, apesar de todos os seus problemas óbvios, era muito melhor do que nada; era pelo menos uma medida reproduzível de *alguma coisa*.

Esta métrica claramente não foi desenvolvida por programadores. Embora o SLOC possa fornecer um parâmetro para medir o esforço individual e a complexidade da aplicação, ele não diz nada sobre a qualidade geral. Ele penaliza o programador eficiente enquanto recompensa o prolixo e está pronto para ser manipulado pelo especialista em detrimento da aplicação subjacente. Se você sabe que o programador novato sentado ao seu lado será considerado mais produtivo porque escreve muito código para produzir um recurso que você poderia produzir com muito menos linhas, qual é a sua resposta? Essa métrica altera a estrutura de recompensa de maneiras que prejudicam a qualidade.

No mundo moderno, o SLOC é uma curiosidade histórica que foi amplamente substituída por métricas mais recentes. Existem inúmeras gemas Ruby (uma pesquisa no Google sobre *métricas Ruby* resultará nas mais recentes) que avaliam quão bem o seu código segue os princípios OOD. O software de métricas funciona escaneando o código-fonte e contando coisas que predizem a qualidade. Executar um conjunto de métricas em seu próprio código pode ser esclarecedor, humilhante e, às vezes, alarmante. Aplicativos aparentemente bem projetados podem acumular um número impressionante de violações de OOD.

Métricas de OOD ruins são indiscutivelmente um sinal de design ruim; código com pontuação baixa será difícil de alterar. Infelizmente, boas pontuações não provam o contrário, ou seja, não garantem que a próxima mudança que você fizer será fácil ou barata. O problema é que é possível criar belos designs que antecipam demais o futuro. Embora esses projetos possam gerar métricas de OOD muito boas, se anteciparem o futuro errado, será caro corrigi-los quando o futuro real finalmente chegar. As métricas OOD não conseguem identificar projetos que fazem a coisa errada da maneira certa.

A história de advertência sobre o SLOC que deu errado se estende, portanto, às métricas OOD. Leve-os com cautela. As métricas são úteis porque são imparciais e

produza números a partir dos quais você possa inferir algo sobre software; no entanto, não são indicadores diretos de qualidade, mas são substitutos para uma medição mais profunda. A métrica de software definitiva seria *o custo por recurso durante o intervalo de tempo que importa*, mas isso não é fácil de calcular. Custo, recurso e tempo são individualmente difíceis de definir, rastrear e medir.

Mesmo que você pudesse isolar um recurso individual e rastrear todos os custos associados, o *intervalo de tempo importante* afetará como o código deve ser julgado. Às vezes, o valor de ter o recurso agora é tão grande que compensa qualquer aumento futuro nos custos. Se a falta de um recurso for forçá-lo a sair do mercado hoje, não importa quanto custará lidar com o código amanhã; você deve fazer o melhor que puder no tempo que tiver. Assumir esse tipo de compromisso de design é como pegar emprestado o tempo do futuro e é conhecido como assumir *dívidas técnicas*. Este é um empréstimo que eventualmente precisará ser reembolsado, muito provavelmente com juros.

Mesmo quando você não assume dívidas técnicas intencionalmente, o design leva tempo e, portanto, custa dinheiro. Como seu objetivo é escrever software com o menor custo por recurso, sua decisão sobre quanto design fazer depende de duas coisas: suas habilidades e seu prazo. Se o design ocupar metade do seu tempo este mês e não começar a devolver dividendos durante um ano, pode não valer a pena. Quando o ato de projetar impede que o software seja entregue no prazo, você perdeu. Entregar metade de um aplicativo bem projetado pode ser o mesmo que não entregar nenhum aplicativo. No entanto, se o design ocupar metade do seu tempo esta manhã, pagar esse tempo esta tarde e continuar a fornecer benefícios durante a vida útil da aplicação, você obterá uma espécie de juros compostos diários sobre o seu tempo; esse esforço de design compensa para sempre.

O ponto de equilíbrio do design depende do programador. Programadores inexperientes que fazem muitos projetos antecipatórios podem nunca chegar a um ponto em que seus esforços anteriores de projeto sejam recompensados. Designers qualificados que escrevem códigos cuidadosamente elaborados esta manhã podem economizar dinheiro esta tarde. Sua experiência provavelmente está em algum lugar entre esses extremos, e o restante deste livro ensina habilidades que você pode usar para mudar o ponto de equilíbrio a seu favor.

Uma breve introdução à programação orientada a objetos Os

aplicativos orientados a

objetos são compostos de objetos e das mensagens que passam entre eles. As mensagens serão as mais importantes das duas, mas nesta breve introdução (e nos primeiros capítulos do livro) os conceitos terão o mesmo peso.

Linguagens Processuais

A programação orientada a objetos é *orientada a objetos* em relação à programação não orientada a objetos ou *processual*. É instrutivo pensar nesses dois estilos em termos de suas diferenças. Imagine uma linguagem de programação processual genérica, na qual você cria scripts simples. Nesta linguagem você pode definir variáveis, ou seja, inventar nomes e associar esses nomes a bits de dados. Uma vez atribuídos, os dados associados podem ser acessados consultando as variáveis.

Como todas as linguagens procedurais, esta conhece um conjunto pequeno e fixo de diferentes tipos de dados, coisas como strings, números, arrays, arquivos e assim por diante. Esses diferentes tipos de dados são conhecidos como *tipos de dados*. Cada tipo de dados descreve um tipo de coisa muito específico. O tipo de dados *string* é diferente do tipo de dados *do arquivo*. A sintaxe da linguagem contém operações integradas para fazer coisas razoáveis com os vários tipos de dados.

Por exemplo, ele pode concatenar strings e ler arquivos.

Como você cria variáveis, você sabe que tipo de coisa cada uma contém. Suas expectativas sobre quais operações você pode usar baseiam-se no seu conhecimento do tipo de dados de uma variável. Você sabe que pode anexar strings, fazer contas com números, indexar em arrays, ler arquivos e assim por diante.

Todos os tipos de dados e operações possíveis já existem; essas coisas estão incorporadas na sintaxe da linguagem. A linguagem pode permitir que você crie funções (agrupue algumas das operações predefinidas sob um novo nome) ou defina estruturas de dados complexas (reúna alguns dos tipos de dados predefinidos em um arranjo nomeado), mas você não pode criar operações totalmente novas ou novos tipos de dados. O que você vê é tudo que você obtém.

Nesta linguagem, como em todas as linguagens procedurais, existe um abismo entre dados e comportamento. Dados são uma coisa, comportamento é algo completamente diferente. Os dados são empacotados em variáveis e depois repassados para o comportamento, que poderia, francamente, fazer qualquer coisa com eles. Os dados são como uma criança cujo comportamento manda para a escola todas as manhãs; não há como saber o que realmente acontece enquanto está fora de vista. As influências sobre os dados podem ser imprevisíveis e em grande parte impossíveis de rastrear.

Linguagens Orientadas a Objetos Agora imagine

um tipo diferente de linguagem de programação, uma linguagem orientada a objetos baseada em classes, como Ruby. Em vez de dividir dados e comportamento em duas esferas separadas, que nunca se encontrarão, Ruby os combina em uma única coisa, um *objeto*.

Os objetos têm comportamento e podem conter dados, dados aos quais somente eles controlam o acesso.

Os objetos invocam o comportamento uns dos outros, enviando *mensagens uns aos outros*.

Ruby tem um *objeto string* em vez de um *tipo de dados string*. As operações que funcionam com strings são incorporadas aos próprios objetos de string, e não à sintaxe da linguagem. Os objetos String diferem porque cada um contém sua própria *sequência* pessoal de dados, mas são semelhantes porque cada um se comporta como os outros. Cada string *encapsula* ou oculta seus dados do mundo. Cada objeto decide por si mesmo quanto ou quanto pouco de seus dados expor.

Como os objetos string fornecem suas próprias operações, Ruby não precisa saber nada em particular sobre o tipo de dados string; ele precisa apenas fornecer uma maneira geral para os objetos enviarem mensagens. Por exemplo, se strings entendem a mensagem `concat`, Ruby não precisa conter sintaxe para concatenar strings, ele apenas precisa fornecer uma maneira para um objeto enviar `concat` para outro.

Mesmo o aplicativo mais simples provavelmente precisará de mais de uma string, número, arquivo ou array. Na verdade, embora seja verdade que ocasionalmente você pode precisar de um floco de neve único e individual de um objeto, é muito mais comum desejar fabricar um monte de objetos que tenham comportamento idêntico, mas encapsulam dados diferentes.

Linguagens OO baseadas em classes, como Ruby, permitem definir uma *classe* que fornece um modelo para a construção de objetos semelhantes. Uma classe define *métodos* (definições de comportamento) e *atributos* (definições de variáveis). Os métodos são invocados em resposta a mensagens. O mesmo nome de método pode ser definido por muitos objetos diferentes; cabe ao Ruby encontrar e invocar o método correto do objeto correto para qualquer mensagem enviada.

Uma vez que a classe String existe, ela pode ser usada para *instanciar* ou criar repetidamente novas *instâncias* de um objeto string. Cada String recém-instanciada *implementa* os mesmos métodos e usa os mesmos nomes de atributos, mas cada uma contém seus próprios dados pessoais. Eles compartilham os mesmos métodos, então todos se comportam como Strings; eles contêm dados diferentes, portanto representam dados diferentes.

A classe String define um tipo que é mais do que meros *dados*. Conhecer o tipo de um objeto permite que você tenha expectativas sobre como ele se comportará. Em uma linguagem processual, as variáveis possuem um único tipo de dados; o conhecimento desse tipo de dados permite que você tenha expectativas sobre quais operações são válidas. Em Ruby um objeto pode ter vários tipos, um dos quais sempre virá de sua classe. O conhecimento do(s) tipo(s) de um objeto permite, portanto, que você tenha expectativas sobre as mensagens às quais ele responde.

Ruby vem com várias classes predefinidas. Os mais imediatamente reconhecíveis são aqueles que se sobrepõem aos tipos de dados usados pelas linguagens procedurais. Por exemplo, a classe String define strings, a classe Fixnum, inteiros. Existe uma classe pré-existente para cada tipo de dados que você esperaria que uma linguagem de programação fornecesse.

No entanto, as próprias linguagens orientadas a objetos são construídas usando objetos e é aqui que as coisas começam a ficar interessantes.

A classe String , ou seja, o modelo para novos objetos string, é *ela mesma um objeto*; é uma instância da classe Class . Assim como todo objeto string é uma instância específica de dados da classe String , todo objeto de classe (String, Fixnum, *ad infinitum*) é uma instância específica de dados da classe Class. A classe String fabrica novas strings, a classe Class fabrica novas classes.

As linguagens OO são, portanto, abertas. Eles não limitam você a um pequeno conjunto de tipos integrados e operações pré-definidas; você pode inventar seus próprios tipos totalmente novos. Cada aplicação OO gradualmente se torna uma linguagem de programação exclusiva, adaptada especificamente ao seu domínio.

Se esta linguagem lhe traz prazer ou dor, é uma questão do design e a preocupação deste livro.

Resumo Se uma

aplicação durar o suficiente, ou seja, se tiver sucesso, seu maior problema será lidar com mudanças. Organizar o código para acomodar mudanças com eficiência é uma questão de design. Os elementos mais visíveis do design são os princípios e os padrões, mas, infelizmente, mesmo a aplicação correta dos princípios e o uso adequado dos padrões não garantem a criação de uma aplicação fácil de alterar.

As métricas OO expõem o quanto bem um aplicativo segue os princípios de design OO. Métricas ruins implicam fortemente em dificuldades futuras; no entanto, boas métricas são menos úteis. Um design que faz a coisa errada pode produzir ótimas métricas, mas ainda pode custar caro mudar.

O truque para obter o máximo retorno do seu investimento em design é adquirir uma compreensão das teorias de design e aplicá-las de maneira adequada, no momento certo e nas quantidades certas. O design depende da sua capacidade de traduzir a teoria em prática.

Qual é a diferença entre teoria e prática?

Em teoria, não há nenhum. Se a teoria fosse prática, você poderia aprender as regras do OOD, aplicá-las de forma consistente e criar um código perfeito a partir de hoje; seu trabalho aqui estaria concluído.

No entanto, não importa quanto profundamente a teoria acredite que isto seja verdade, a prática sabe melhor. Ao contrário da teoria, a prática suja as mãos. É a prática que estabelece tijolos, constrói pontes e escreve códigos. A prática vive no mundo real de mudança, confusão e incerteza. Enfrenta escolhas concorrentes e, fazendo uma careta, escolhe o mal menor; esquia-se, protege-se, rouba Pedro para pagar Paulo. Ela ganha a vida fazendo o melhor que pode com o que tem.

A teoria é útil e necessária e tem sido o foco deste capítulo. Mas chega já; é hora de praticar.

CAPÍTULO 2

Projetando aulas com uma única responsabilidade

A base de um sistema orientado a objetos é a *mensagem*, mas a estrutura organizacional mais visível é a *classe*. As mensagens estão no centro do design, mas como as classes são tão óbvias, este capítulo começa pequeno e se concentra em como decidir o que pertence a uma classe. A ênfase do design mudará gradualmente das aulas para as mensagens ao longo dos próximos capítulos.

Quais são suas aulas? Quantos você deveria ter? Que comportamento eles implementarão? Quanto eles sabem sobre outras classes? Quanto de si mesmos eles deveriam expor?

Essas perguntas podem ser esmagadoras. Cada decisão parece permanente e repleta de perigos. Não tema. Nesta fase a sua primeira obrigação é respirar fundo e *insistir para que seja simples*. Seu objetivo é modelar seu aplicativo, usando classes, de forma que ele faça o que deveria fazer *agora* e também seja fácil de alterar *posteriormente*.

Estes são dois critérios muito diferentes. Qualquer um pode organizar o código para que funcione agora. A aplicação de hoje pode ser subjugada pela pura força de vontade. É um alvo permanente a uma distância conhecida. Está à sua mercê.

Criar um aplicativo fácil de alterar, entretanto, é uma questão diferente. Seu aplicativo precisa funcionar agora apenas uma vez; deve ser fácil mudar para sempre.

Essa qualidade de fácil mutabilidade revela a arte da programação. Alcançar isso requer conhecimento, habilidade e um pouco de criatividade artística.

Felizmente, você não precisa descobrir tudo do zero. Muita reflexão e pesquisa foram feitas para identificar as qualidades que tornam um aplicativo fácil de mudar.

As técnicas são simples; você só precisa saber o que são e como usá-los.

Decidindo o que pertence a uma classe

Você tem uma aplicação em mente. Você sabe o que deveria fazer. Você pode até ter pensado em como implementar os comportamentos mais interessantes. O problema não é de conhecimento técnico, mas de organização; você sabe como escrever o código, mas não onde colocá-lo.

Agrupando métodos em classes Em uma linguagem

OO baseada em classes como Ruby, os métodos são definidos em classes. As classes que você criar afetarão para sempre a maneira como você pensa sobre seu aplicativo. Eles definem um mundo virtual, que restringe a imaginação de todos a jusante. Você está construindo uma caixa da qual pode ser difícil pensar.

Apesar da importância de agrupar corretamente os métodos em classes, nesta fase inicial do seu projeto você não conseguirá acertar. Você nunca saberá menos do que sabe agora. Se a sua aplicação for bem-sucedida, muitas das decisões que você toma hoje precisarão ser alteradas mais tarde. Quando esse dia chegar, sua capacidade de fazer essas alterações com êxito será determinada pelo design do seu aplicativo.

O design é mais a arte de preservar a mutabilidade do que o ato de alcançar a perfeição.

Organizando o código para permitir mudanças fáceis Afirmar que o código

deve ser fácil de mudar é o mesmo que afirmar que as crianças devem ser educadas; é impossível discordar da afirmação, mas de forma alguma ajuda os pais a criar um filho agradável. A ideia de *fácil* é muito ampla; você precisa de definições concretas de *facilidade* e critérios específicos pelos quais julgar o código.

Se você definir *fácil de mudar* como

- As alterações não têm efeitos colaterais inesperados
- Pequenas mudanças nos requisitos requerem correspondentemente pequenas mudanças no código
- O código existente é fácil de reutilizar
- A maneira mais fácil de fazer uma alteração é adicionar um código que por si só seja fácil de alterar

Criando classes que têm uma única responsabilidade

Então o código que você escreve deve ter as seguintes qualidades. O código deve ser

- **Transparente** As consequências da mudança devem ser óbvias no código que é mudando e em código distante que depende dele
- **Razoável** O custo de qualquer mudança deve ser proporcional aos benefícios que a mudança alcança
- **Utilizável** O código existente deve ser utilizável em contextos novos e inesperados
- **Exemplar** O próprio código deve encorajar aqueles que o alteram a perpetuar essas qualidades

Um código transparente, razoável, utilizável e exemplar (TRUE) não apenas atende às necessidades de hoje, mas também pode ser alterado para atender às necessidades do futuro. O primeiro passo na criação de um código VERDADEIRO é garantir que cada classe tenha uma responsabilidade única e bem definida.

Criando classes que têm uma única responsabilidade

Uma classe deve fazer a menor coisa útil possível; isto é, deveria ter uma única responsabilidade.

Ilustrar como criar uma classe que tenha uma única responsabilidade e explicar por que isso é importante requer um exemplo, o que por sua vez requer uma pequena divergência no domínio das bicicletas.

Um exemplo de aplicação: bicicletas e engrenagens As bicicletas são máquinas maravilhosamente eficientes, em parte porque usam engrenagens para proporcionar vantagem mecânica aos humanos. Ao andar de bicicleta, você pode escolher entre uma marcha pequena (que é fácil de pedalar, mas não muito rápida) ou uma marcha grande (que é mais difícil de pedalar, mas faz você avançar). As engrenagens são ótimas porque você pode usar as pequenas para subir colinas íngremes e as grandes para voar de volta para baixo.

As engrenagens funcionam alterando a distância que a bicicleta percorre cada vez que seus pés completam um círculo com os pedais. Mais especificamente, seu equipamento controla quantas vezes as rodas giram cada vez que os pedais giram. Em uma marcha pequena, seus pés giram várias vezes para fazer as rodas girarem apenas uma vez; em uma marcha grande, cada rotação completa do pedal pode fazer com que as rodas girem várias vezes. Veja a Figura 2.1.

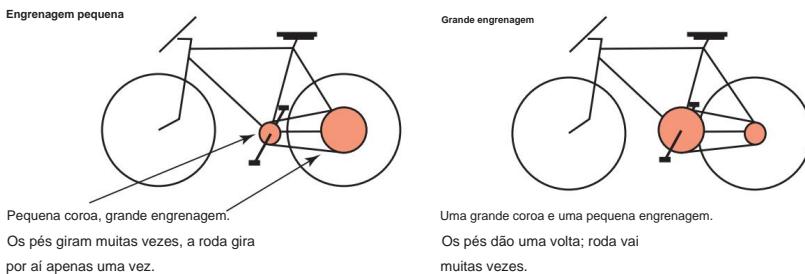


Figura 2.1 Engrenagens de bicicletas pequenas versus grandes.

Os termos *pequeno* e *grande* não são muito precisos. Para comparar diferentes marchas, os ciclistas use a proporção do número de seus dentes. Essas proporções podem ser calculadas com este script Ruby simples:

```

1 coroa = 52 2 engrenagens          # número de dentes
= 11

Proporção 3      = coroa / cog.to_f
Proporção de 4 opções de venda      # -> 4.72727272727273
5

6 coroa = 30
7 engrenagem      = 27
Proporção 8      = coroa / cog.to_f
Proporção de 9 opções de venda      # -> 1.11111111111111

```

A engrenagem criada pela combinação de uma coroa de 52 dentes com uma roda dentada de 11 dentes (52×11) tem uma proporção de cerca de 4,73. Cada vez que seus pés pressionam os pedais *uma vez*, suas rodas darão quase *cinco* voltas. O 30×27 é um equipamento muito mais fácil; cada revolução do pedal faz com que as rodas girem um pouco mais de uma vez.

Acredite ou não, há pessoas que se preocupam profundamente com as engrenagens das bicicletas. Você pode ajudá-los escrevendo um aplicativo Ruby para calcular relações de transmissão.

A aplicação será feita de classes Ruby, cada uma representando alguma parte do domínio. Se você ler a descrição acima procurando por substantivos que representem objetos no domínio, você verá palavras como *bicicleta* e *equipamento*. Esses substantivos representam candidatos mais simples para serem classes. A intuição diz que *bicicleta* deveria ser uma classe, mas nada na descrição acima lista qualquer comportamento para bicicleta, portanto, por enquanto, ela não se qualifica. A *engrenagem*, entretanto, possui coroas, engrenagens e relações, ou seja, possui dados e comportamento. Merece ser uma aula. Tomando o comportamento do script acima, você cria esta classe Gear simples:

```

1 classe de equipamento
2 attr_reader :chainring, :cog
3 def inicializar (roda dentada, roda dentada)
4   @chainring = coroa
5   @cog           = engrenagem
6 fim
7
8 Razão de 8 defesas
9   coroa / cog.to_f
10 fim
11 fim
12
13 coloca Gear.new(52, 11).ratio 14 coloca          # -> 4.72727272727273
Gear.new(30, 27).ratio                           # -> 1.111111111111111

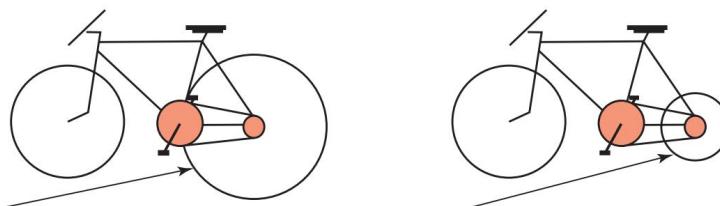
```

Esta classe Gear é a própria simplicidade. Você cria uma nova instância do Gear fornecendo o número de dentes para a coroa e a roda dentada. Cada instância implementa três métodos: coroa, roda dentada e proporção.

Gear é uma subclasse de Object e, portanto, herda muitos outros métodos. Um Gear consiste em tudo que ele implementa diretamente mais tudo que ele herda, então o equipamento completo conjunto de comportamentos, isto é, o conjunto total de mensagens às quais ele pode responder, é bastante grande. A herança é importante para o design do seu aplicativo, mas este caso simples em que o Gear herda do objeto é tão básico que, pelo menos por enquanto, você pode agir como se esses objetos herdados métodos não existem. Formas mais sofisticadas de herança serão abordadas em Capítulo 6, Adquirindo comportamento por meio de herança.

Você mostra sua calculadora Gear para uma amiga ciclista e ela acha útil, mas imediatamente pede uma melhoria. Ela tem duas bicicletas; as bicicletas têm exatamente o mesmo engrenagem, mas eles têm tamanhos de rodas diferentes. Ela gostaria que você também calculasse o efeito da diferença nas rodas.

Uma bicicleta com rodas enormes percorre muito mais distância durante cada rotação da roda do que um com rodas minúsculas, conforme mostrado na Figura 2.2.



Roda grande – uma rotação percorre um longo caminho. Roda pequena – uma rotação quase não leva a lugar nenhum.

Figura 2.2 Efeito do tamanho da roda na distância percorrida.

Os ciclistas (pelo menos os dos Estados Unidos) usam algo chamado *engrenagem polegadas* para compare bicicletas que diferem tanto na marcha quanto no tamanho das rodas. A fórmula segue:

polegadas de engrenagem = diâmetro da roda * relação de transmissão

onde

diâmetro da roda = diâmetro do aro + duas vezes o diâmetro do pneu.

Você altera a classe Gear para adicionar este novo comportamento:

```

1 classe de equipamento
2     attr_reader :chainring, :cog, :rim, :tire
3     def inicializar (rodadentada, rodadentada, aro, pneu)
4         @chainring = coroa
5         @cog      = engrenagem
6         @aro      = aro
7         @pneu     = pneu
8     fim
9
10    Proporção de 10 defesas
11        coroa / cog.to_f
12    fim
13
14    def gear_inches
15        # pneu dá duas voltas no aro em relação ao diâmetro
16        proporção * (aro + (pneu * 2))
17    fim
18    fim
19
20    coloca Gear.new(52, 11, 26, 1.5).gear_inches
21    # -> 137.09090909090909
22
23    coloca Gear.new(52, 11, 24, 1.25).gear_inches 24 # -> 125.27272727272727

```

O novo método `gear_inches` assume que os tamanhos dos aros e pneus são dados em polegadas, que pode ou não estar correto. Com essa ressalva, a classe Gear atende às especificações (tais como são) e ao código, com exceção do seguinte bug, funciona.

```

1 coloca Gear.new(52, 11).ratio # isso não funcionava?
2 # ArgumentError: número errado de argumentos (2 por 4)
3 #

```

Criando classes que têm uma única responsabilidade

```

4 #
5 #
6

```

O bug acima foi introduzido quando o método gear_inches foi adicionado.

Gear.initialize foi alterado para exigir dois argumentos adicionais, aro e pneu.

Alterar o número de argumentos que um método requer interrompe todos os chamadores existentes do método.

Normalmente, isso seria um problema terrível que teria que ser resolvido instantaneamente, mas como o aplicativo é tão pequeno que o Gear.initialize atualmente não tem outros chamadores, o bug pode ser ignorado por enquanto.

Agora que existe uma classe Gear rudimentar , é hora de fazer a pergunta: Esta é a melhor maneira de organizar o código?

A resposta, como sempre, é: depende. Se você espera que o aplicativo permaneça estático para sempre, o Gear em sua forma atual pode ser bom o suficiente. Porém, já é possível prever a possibilidade de todo um aplicativo de calculadoras para ciclistas.

Gear é a primeira de muitas classes de um aplicativo que irá *evoluir*. Para evoluir com eficiência, o código deve ser fácil de alterar.

Por que a responsabilidade única é importante Aplicativos

fáceis de alterar consistem em classes fáceis de reutilizar. Classes reutilizáveis são unidades conectáveis de comportamento bem definido que possuem poucos emaranhados. Um aplicativo fácil de alterar é como uma caixa de blocos de construção; você pode selecionar apenas as peças necessárias e montá-las de maneiras imprevistas.

Uma classe que tem mais de uma responsabilidade é difícil de reutilizar. As várias responsabilidades provavelmente estão completamente enredadas *na classe*. Se você quiser reutilizar parte (mas não todo) de seu comportamento, é impossível obter apenas as partes necessárias.

Você se depara com duas opções e nenhuma delas é particularmente atraente.

Se as responsabilidades estiverem tão acopladas que você não possa usar apenas o comportamento necessário, você poderá duplicar o código de interesse. Esta é uma ideia terrível. Código duplicado leva a manutenção adicional e aumenta bugs. Se a classe estiver estruturada de forma que você possa acessar apenas o comportamento necessário, você poderá reutilizar a classe inteira. Isso apenas substitui um problema por outro.

Como a classe que você está reutilizando está confusa sobre o que faz e contém diversas responsabilidades confusas, ela tem muitos motivos para mudar. Ele pode mudar por um motivo que não está relacionado ao seu uso e, cada vez que ele muda, existe a possibilidade de quebrar todas as classes que dependem dele. Você aumenta a chance de seu aplicativo quebrar inesperadamente se depender de classes que fazem muita coisa.

Determinando se uma classe tem uma única responsabilidade

Como você pode determinar se a classe Gear contém comportamento que pertence a outro lugar? Uma maneira é fingir que é senciente e interrogá-lo. Se você reformular cada um de seus métodos como uma pergunta, fazer a pergunta deverá fazer sentido. Por exemplo, “*Por favor, Sr. Gear, qual é a sua proporção?*” parece perfeitamente razoável, enquanto “*Por favor, Sr. Gear, quais são seus gear_inches?*” está em terreno instável e “*Por favor, Sr. Gear, qual é o seu pneu (tamanho)?*” é simplesmente ridículo.

Não resista à ideia de “*qual é o seu pneu?*” é uma pergunta que pode ser feita legitimamente. Dentro da classe Gear, pneu pode parecer algo diferente de relação ou gear_inches, mas isso não significa nada. Do ponto de vista de qualquer outro objeto, qualquer coisa a que o Gear possa responder é apenas mais uma mensagem. Se o Gear responder, alguém o enviará, e esse remetente poderá ter uma surpresa desagradável quando o Gear mudar.

Outra maneira de entender o que uma classe está realmente fazendo é tentar descrevê-la em uma frase. Lembre-se de que uma classe deve fazer a menor coisa útil possível.

Essa coisa deveria ser simples de descrever. Se a descrição mais simples que você puder criar usar a palavra “e”, a classe provavelmente terá mais de uma responsabilidade. Se usar a palavra “ou”, então a turma tem mais de uma responsabilidade e elas nem estão muito relacionadas.

Os designers OO usam a palavra coesão para descrever esse conceito. Quando tudo em uma classe está relacionado ao seu propósito central, diz-se que a classe é *altamente coesa* ou tem uma responsabilidade única. O Princípio da Responsabilidade Única (SRP) tem suas raízes na ideia de Rebecca Wirfs-Brock e Brian Wilkerson de Design Orientado à Responsabilidade (RDD). Eles dizem “Uma classe tem responsabilidades que cumprem seu propósito”. O SRP não exige que uma classe faça apenas uma coisa muito restrita ou que mude por apenas um único motivo minucioso; em vez disso, o SRP exige que uma classe seja coesa – que tudo o que a classe faz esteja altamente relacionado ao seu propósito.

Como você descreveria a responsabilidade da classe Gear? Que tal “*Calcular a relação entre duas rodas dentadas?*” Se isso for verdade, a classe, tal como existe atualmente, faz demais. Talvez “*Calcular o efeito que uma engrenagem tem sobre uma bicicleta?*” Dito desta forma, gear_inches está de volta ao solo sólido, mas o tamanho dos pneus ainda é bastante instável.

A aula não parece certa. Gear tem mais de uma responsabilidade, mas não é óbvio o que deveria ser feito.

Determinando quando tomar decisões de design

É comum se encontrar em uma situação em que você sabe que algo não está certo em uma aula. Essa classe é realmente um *Gear*? Tem aros e pneus, pelo amor de Deus! Talvez *Gear* devesse ser *bicicleta*? Ou talvez haja uma *roda* aqui em algum lugar?

Se você soubesse quais solicitações de recursos chegariam no futuro, você poderia tomar decisões de design perfeitas hoje. Infelizmente, você não. Qualquer coisa pode acontecer. Você pode perder muito tempo dividido entre alternativas igualmente plausíveis antes de jogar os dados e escolher a errada.

Não se sinta obrigado a tomar decisões de design prematuramente. Resista, mesmo que você tema que seu código desanime os gurus do design. Ao se deparar com uma classe imperfeita e confusa como Gear, pergunte-se: “*Qual é o custo futuro de não fazer nada hoje?*”

Este é um aplicativo (muito) pequeno. Tem um desenvolvedor. Você está intimamente familiarizado com a classe Gear . O futuro é incerto e você nunca saberá menos do que sabe agora. O curso de ação mais econômico pode ser esperar por mais informações.

O código na classe Gear é *transparente* e *razoável*, mas isso não reflete um design excelente, apenas que a classe não tem dependências, portanto alterações nela não têm consequências. Se adquirisse dependências, violaria repentinamente ambos os objetivos e deveria ser reorganizado *naquele momento*. Convenientemente, as novas dependências fornecerão as informações exatas de que você precisa para tomar boas decisões de design.

Quando o custo futuro de não fazer nada for igual ao custo atual, adie a decisão. Tome a decisão apenas quando for necessário, com as informações que você tem naquele momento.

Embora haja um bom argumento para deixar o Gear como está por enquanto, você também pode apresentar um argumento defensável de que ele deveria ser alterado. A estrutura de cada classe é uma mensagem para futuros mantenedores da aplicação. Ele revela suas intenções de design. Para o bem ou para o mal, os padrões que você estabelece hoje serão replicados para sempre.

Gear *mente* sobre suas intenções. Não é *utilizável* nem *exemplar*. Tem múltiplas responsabilidades e por isso não deve ser reutilizado. Não é um padrão que deva ser replicado.

Há uma chance de alguém reutilizar o Gear ou criar um novo código que siga seu padrão enquanto você espera por melhores informações. Outros desenvolvedores acreditam que suas intenções estão refletidas no código; quando o código mente, você deve estar alerta para que os programadores acreditem e depois propaguem essa mentira.

Essa tensão “melhore agora” versus “melhore mais tarde” sempre existe. Os aplicativos nunca são perfeitamente projetados. Toda escolha tem um preço. Um bom designer comprehende esta tensão e minimiza os custos ao fazer compromissos informados entre as necessidades do presente e as possibilidades do futuro.

Escrevendo código que aceita mudanças

Você pode organizar o código de forma que o Gear seja fácil de mudar, mesmo que você não saiba quais mudanças virão. Como a mudança é inevitável, a codificação em um estilo mutável traz grandes recompensas futuras. Como um bônus adicional, a codificação nesses estilos melhorará seu código hoje, sem nenhum custo extra.

Aqui estão algumas técnicas conhecidas que você pode usar para criar código que aceite mudanças.

Depende do comportamento, não dos dados

O comportamento é capturado em métodos e invocado pelo envio de mensagens. Quando você cria classes que têm uma única responsabilidade, cada pedacinho de comportamento fica em um e apenas um lugar. A frase “Don’t Repeat Yourself” (DRY) é um atalho para essa ideia. O código DRY tolera alterações porque qualquer mudança no comportamento pode ser feita alterando o código em apenas um lugar.

Além do comportamento, os objetos geralmente contêm dados. Os dados são mantidos em uma variável de instância e podem ser qualquer coisa, desde uma string simples ou um hash complexo. Os dados podem ser acessados de duas maneiras; você pode consultar diretamente a variável de instância ou pode agrupar a variável de instância em um método acessador.

Ocultar variáveis de instância

Sempre envolva variáveis de instância em métodos acessadores em vez de se referir diretamente às variáveis, como o método ratio faz abaixo:

```

1 classe de equipamento
2 def inicializar (roda dentada, roda dentada) @chainring = coroa
3           @cog 5 end
4           = engrenagem
5
6
7 Proporção de 7 defesas
8   @chainring / @cog.to_f
9   # <-- caminho para a ruína
10  fim
11 fim

```

Oculte as variáveis, mesmo da classe que as define, envolvendo-as em métodos. Ruby fornece attr_reader como uma maneira fácil de criar métodos de encapsulamento:

```

1 classe de equipamento
2 attr_reader :chainring, :cog # <-----
3 def inicializar (roda dentada, roda dentada)
4   @chainring = coroa
5   @cog       = engrenagem
6 fim
7
8 Razão de 8 defesas
9   coroa / cog.to_f           # <-----
10 fim
11 fim

```

Usar attr_reader fez com que Ruby criasse métodos wrapper simples para as variáveis.

Aqui está uma representação virtual daquela criada para cog:

```

1   # implementação padrão via attr_reader
2 engrenagem definida
3   @cog
4 fim

```

Este método cog é agora o único lugar no código que entende o que significa cog.

Cog se torna o resultado do envio de uma mensagem. A implementação deste método muda a engrenagem dos dados (que são referenciados em todo lugar) ao comportamento (que é definido uma vez).

Se a variável de instância @cog for referida dez vezes e de repente precisar ser ajustado, o código precisará de muitas alterações. No entanto, se @cog estiver envolvido em um método, você pode alterar o significado de engrenagem implementando sua própria versão do método. Seu novo método pode ser tão simples quanto a primeira implementação abaixo, ou mais complicado, como o segundo:

```

1   # uma simples reimplementação da engrenagem
2 engrenagem definida
3   @cog   *fator_de_ajuste_imprevisto
4 fim

```

```

1   #um mais complexo
2 engrenagem definida
3   @cog   *(foo? ? bar_adjustment : baz_adjustment)
4 fim

```

O primeiro exemplo poderia ter sido feito fazendo uma alteração no valor da variável de instância. No entanto, você nunca pode ter certeza de que eventualmente não precisará de algo como o segundo exemplo. O segundo ajuste é uma simples mudança de comportamento quando feita em um método, mas um código que destrói a bagunça quando aplicado a um monte de referências de variáveis de instância.

Lidar com dados como se fossem um objeto que entende mensagens introduz dois novos problemas. A primeira questão envolve visibilidade. Agrupar a variável de instância `@cog` em um método `cog` *público* expõe essa variável a outros objetos em seu aplicativo; qualquer outro objeto agora pode enviar engrenagens para um Gear. Teria sido igualmente fácil criar um método de encapsulamento *privado*, que transformasse os dados em comportamento sem expor esse comportamento a todo o aplicativo. A escolha entre essas duas alternativas é abordada no Capítulo 4, Criando interfaces flexíveis.

A segunda questão é mais abstrata. Como é possível agrupar cada variável de instância em um método e, portanto, tratar qualquer variável como se fosse apenas outro objeto, a distinção entre *dados* e um *objeto* regular começa a desaparecer. Embora às vezes seja conveniente pensar em partes do seu aplicativo como dados sem comportamento, a maioria das coisas é melhor pensada como objetos simples e antigos.

Independentemente de quão longe seus pensamentos vão nessa direção, você deve ocultar os dados de si mesmo. Isso protege o código de ser afetado por alterações inesperadas.

Os dados muitas vezes têm um comportamento que você ainda não conhece. Envie mensagens para acessar variáveis, mesmo que você as considere dados.

Ocultar estruturas de dados

Se ser anexado a uma variável de instância é ruim, depender de uma estrutura de dados complicada é pior. Considere a seguinte classe ObscuringReferences :

```

1 classe ObscuringReferences 2 attr_reader :
dados 3 def inicializar (dados)

4     @dados = dados
5 fim
6
7 diâmetros definidos
8     # 0 é o aro, 1 é o pneu data.collect
9     { |cell| célula[0] + (célula[1] * 2)}
10
11 fim
12 #...muitos outros métodos que indexam no array
13 fim

```

Esta classe espera ser inicializada com um conjunto bidimensional de aros e pneus:

```
1 # tamanhos de aros e pneus (agora em milímetros!) em uma matriz 2d 2 @data = [[622, 20],  
[622, 23], [559, 30], [559, 40]]
```

`ObscuringReferences` armazena seu argumento de inicialização na variável `@data` e usa obedientemente o `attr_reader` do Ruby para agrupar a variável de instância `@data` em um método. O método dos diâmetros envia a mensagem de dados para acessar o conteúdo da variável. Esta classe certamente faz tudo o que é necessário para ocultar a variável de instância de si mesma.

No entanto, como `@data` contém uma estrutura de dados complicada, apenas ocultar a variável de instância não é suficiente. O método `data` apenas retorna o array. Para fazer algo útil, cada remetente de dados deve ter conhecimento completo de qual parte dos dados está em qual índice da matriz.

O método dos diâmetros sabe não apenas como calcular os diâmetros, mas também onde encontrar aros e pneus no conjunto. Ele sabe explicitamente que, se iterar sobre os dados, os aros estarão em `[0]` e os pneus em `[1]`.

Depende da estrutura do array. Se essa estrutura mudar, esse código deverá mudar. Quando você tem dados em um array, não demora muito para que você tenha referências à estrutura do array por toda parte. As referências estão vazando. Eles escapam do encapsulamento e se insinuam por todo o código. Eles não estão SECOS. O conhecimento de que os aros estão em `[0]` não deve ser duplicado; deve ser conhecido em apenas um lugar.

Este exemplo simples é bastante ruim; imagine as consequências se os dados retornassem uma matriz de hashes referenciados em muitos lugares. Uma mudança em sua estrutura ocorreria em cascata por todo o seu código; cada mudança representa uma oportunidade de criar um bug tão furtivo que suas tentativas de encontrá-lo farão você chorar.

Referências diretas em estruturas complicadas são confusas, porque obscurecem o que os dados realmente são, e são um pesadelo de manutenção, porque cada referência precisará ser alterada quando a estrutura do array mudar.

Em Ruby é fácil separar estrutura de significado. Assim como você pode usar um método para agrupar uma variável de instância, você pode usar a classe Ruby `Struct` para agrupar uma estrutura. No exemplo a seguir, `RevealingReferences` tem a mesma interface da classe anterior. Ele usa um array bidimensional como argumento de inicialização e implementa o método dos diâmetros. Apesar destas semelhanças externas, a sua implementação interna é muito diferente.

```

1 aulaReferências Reveladoras
2 attr_reader : rodas
3 def inicializar (dados)
4     @rodas = wheelify(dados)
5 fim
6
7 diâmetros definidos
8     rodas.collect {|roda| roda.rim +
9         (roda.pneu * 2)}
10 fim
11 # ... agora todos podem enviar aro/pneu para roda
12
13 Roda = Struct.new(:aro, :pneu)
14 def wheelify (dados)
15     data.collect {|célula|
16         Wheel.new(célula[0], célula[1])}
17 fim
18 fim

```

O método dos diâmetros acima agora não tem conhecimento da estrutura interna do variedade. Tudo o que os diâmetros sabem é que as rodas de mensagens retornam um número enumerável e que cada coisa enumerada responde ao aro e ao pneu. O que antes eram referências a cell[1] foram transformados em mensagens enviadas para wheel.tire.

Todo o conhecimento da estrutura da matriz de entrada foi isolado dentro do

Método wheelify , que converte o array de Arrays em um array de Structs.

A documentação oficial do Ruby (<http://ruby-doc.org/core/classes/Struct.html>)

define Struct como “uma maneira conveniente de agrupar vários atributos, usando

métodos acessadores, sem ter que escrever uma classe explícita.” Isso é exatamente o que

wheelify faz; ele cria pequenos objetos leves que respondem ao aro e ao pneu.

O método wheelify contém o único trecho de código que entende a estrutura do array recebido.

Se a entrada mudar, o código mudará apenas neste

lugar. São necessárias quatro novas linhas de código para criar a estrutura da roda e definir o

método wheelify , mas essas poucas linhas de código são um pequeno inconveniente em comparação ao custo permanente de indexar repetidamente em uma matriz complexa.

Este estilo de código permite proteger contra alterações em dados de propriedade externa estruturas e para tornar seu código mais legível e revelador de intenção. Ele negocia a indexação em uma estrutura para enviar mensagens a um objeto. O método wheelify acima isola as informações estruturais confusas e seca o código. Isso faz isso classe muito mais tolerante à mudança.

Embora possa ser mais fácil ter apenas um conjunto de Rodas para começar, não é sempre possível. Se você puder controlar a entrada, passe um objeto útil, mas se você estiver compelido a assumir uma estrutura bagunçada, esconder a bagunça até de você mesmo.

Aplique responsabilidade única em todos os lugares

Criar classes com uma única responsabilidade tem implicações importantes para o design, mas a ideia de responsabilidade única pode ser empregada de forma útil em muitas outras partes do seu código.

Extraia responsabilidades extras dos métodos

Os métodos, assim como as classes, devem ter uma única responsabilidade. Todas as mesmas razões aplicar; ter apenas uma responsabilidade os torna fáceis de alterar e reutilizar. Todos as mesmas técnicas de design funcionam; faça perguntas sobre o que eles fazem e tente descreva suas responsabilidades em uma única frase.

Veja o método dos diâmetros da classe RevealingReferences:

```

1 diâmetros definidos
2     rodas.collect {|roda| roda.rim +
3         (roda.pneu * 2)}
4 fim
```

Este método tem claramente duas responsabilidades: itera sobre as rodas e calcula o diâmetro de cada roda.

Simplifique o código separando-o em dois métodos, cada um com uma responsabilidade. Esta próxima refatoração move o cálculo do diâmetro de uma única roda para o seu próprio método. A refatoração introduz um envio de mensagem adicional, mas neste ponto seu design, você deve agir como se o envio de uma mensagem fosse gratuito. O desempenho pode ser melhorado mais tarde, se necessário. No momento, o objetivo mais importante do design é escrever código que seja facilmente alterável.

```

1     # primeiro - itera sobre o array
2 diâmetros definidos
3     rodas.collect {|roda| diâmetro(roda)}
4 fim
5
6     # segundo - calcule o diâmetro de UMA roda
7 diâmetro def (roda)
8     roda.rim + (roda.pneu * 2))
9 fim
```

Você precisará obter o diâmetro de apenas uma roda? Observe o código novamente; você já fiz. Esta refatoração não é um caso de overdesign, ela apenas reorganiza o código que está atualmente em uso. O fato de que o método singular agora pode ser chamado de outros lugares é um efeito colateral gratuito e feliz.

Separar a iteração da ação que está sendo executada em cada elemento é uma tarefa caso comum de responsabilidade múltipla que é fácil de reconhecer. Em outros casos o problema não é tão óbvio.

Lembre-se do método gear_inches da classe Gear :

```

1 def gear_inches
2     # pneu dá duas voltas no aro em relação ao diâmetro
3     proporção * (aro + (pneu * 2))
4 fim

```

Gear_inches é uma responsabilidade da classe Gear ? É razoável que assim seja.

Mas se for, por que esse método parece tão errado? É confuso e incerto e parece provável que cause problemas mais tarde. A causa raiz do problema é que o método *em si* tem mais de uma responsabilidade.

Escondido dentro de gear_inches está o cálculo do diâmetro da roda. Extraindo esse cálculo neste novo método de diâmetro tornará mais fácil examinar o responsabilidades da classe.

```

1 def gear_inches
2     proporção * diâmetro
3     fim
4
5 diâmetro definido
6     aro + (pneu * 2)
7 fim

```

O método gear_inches agora envia uma mensagem para obter o diâmetro da roda. Notar que a refatoração não altera a forma como o diâmetro é calculado; apenas isola o comportamento em um método separado.

Faça essas refatorações mesmo quando você não conhece o design final. Eles são necessário, não porque o design seja claro, mas porque não é. Você não precisa saber onde você usará boas práticas de design para chegar lá. Boas práticas revelam design.

Essa simples refatoração torna o problema óbvio. Gear é definitivamente responsável para calcular gear_inches , mas o Gear não deve calcular o diâmetro da roda.

O impacto de uma única refatoração como essa é pequeno, mas o efeito cumulativo desse estilo de codificação é enorme. Métodos que têm uma única responsabilidade conferem os seguintes benefícios:

- **Expor qualidades anteriormente ocultas** Refatorar uma classe para que todos os seus métodos tenham uma única responsabilidade tem um efeito esclarecedor sobre a classe. Mesmo que você não pretenda reorganizar os métodos em outras classes hoje, fazer com que cada um deles sirva a um único propósito torna mais óbvio o conjunto de coisas que a classe faz.
- **Evite a necessidade de comentários** Quantas vezes você viu um comentário desatualizado? Como os comentários não são executáveis, eles são apenas uma forma de documentação decadente. Se um trecho de código dentro de um método precisar de um comentário, extraia esse trecho em um método separado. O novo nome do método tem o mesmo propósito do comentário antigo.
- **Incentive a reutilização** Os métodos pequenos incentivam um comportamento de codificação saudável para sua aplicação. Outros programadores reutilizarão os métodos em vez de duplicar o código. Eles seguirão o padrão que você estabeleceu e, por sua vez, criará novos métodos pequenos e reutilizáveis. Este estilo de codificação se propaga.
- **São fáceis de migrar para outra classe** Quando você obtém mais informações de design e decide fazer alterações, métodos pequenos são fáceis de migrar. Você pode reorganizar o comportamento sem fazer muita extração e refatoração de métodos. Pequenos métodos reduzem as barreiras para melhorar seu design.

Isole responsabilidades extras nas classes Uma

vez que cada método tenha uma única responsabilidade, o escopo da sua classe ficará mais aparente. A classe Gear tem algum comportamento semelhante ao de uma roda. Este aplicativo precisa de uma classe Wheel ?

Se as circunstâncias permitirem que você crie uma classe Wheel separada , talvez você deva. Por enquanto, imagine que você opte por não criar uma classe nova, permanente e publicamente disponível neste momento. Talvez alguma restrição de design tenha sido imposta a você, ou talvez você esteja tão incerto sobre para onde está indo que não deseja criar uma nova classe da qual outros possam começar a depender, para não mudar de ideia.

Pode parecer impossível para o Gear ter uma única responsabilidade, a menos que você remova seu comportamento semelhante a uma roda; o comportamento extra está no Gear ou não. No entanto, definir a escolha do design em termos de um ou outro é míope. Existem outras opções. Seu objetivo é preservar a responsabilidade única no Gear e, ao mesmo tempo, assumir o menor número possível de compromissos de design. Como você está escrevendo código mutável, será melhor atendido por

adiar decisões até que você seja absolutamente forçado a tomá-las. Qualquer decisão que você fazer antes de um requisito explícito é apenas um palpite. Não decida; preserve seu capacidade de tomar uma decisão *mais tarde*.

Ruby permite que você remova a responsabilidade pelo cálculo do diâmetro do pneu Equipamento sem se comprometer com uma nova classe. O exemplo a seguir estende o anterior Wheel Struct com um bloco que adiciona um método para calcular o diâmetro.

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel
3 def inicializar (roda dentada, roda dentada, aro, pneu)
4   @chainring = coroa
5   @cog      = engrenagem
6   @roda     = Wheel.new(aro, pneu)
7 fim
8
9 Proporção de 9 defesas
10 coroa / cog.to_f
11 fim
12
13 def gear_inches
14   relação * diâmetro da roda
15 fim
16
17 Wheel = Struct.new(:rim, :tire) do
18   diâmetro definido
19   aro + (pneu * 2)
20 fim
21 fim
22 fim

```

Agora você tem uma roda que pode calcular seu próprio diâmetro. Incorporando esta roda em Obviamente, o equipamento não é o objetivo do design a longo prazo; é mais um experimento em código organização. Isso limpa o Gear , mas adia a decisão sobre o Wheel.

Incorporar Wheel dentro do Gear sugere que você espera que uma Wheel só existem no contexto de um Gear. Se você levantar a cabeça deste livro por um momento e olhar para o mundo real, o bom senso sugere o contrário. Neste caso, basta existem informações agora para apoiar a criação de uma classe Wheel independente. No entanto, nem todos os domínios são tão claros.

Se você tiver uma turma confusa com muitas responsabilidades, separe-as responsabilidades em diferentes classes. Concentre-se na classe primária. Decidir

Finalmente, a roda real

sus responsabilidades e impor sua decisão com firmeza. Se você identificar responsabilidades extras que você ainda não pode remover, isole-os. Não permita que responsabilidades estranhas vazar para sua classe.

Finalmente, a roda real

Enquanto você pondera sobre o design da classe Gear , o futuro chega. Você mostra o seu calculadora para sua amiga ciclista novamente e ela lhe diz que é muito legal, mas que enquanto você está escrevendo calculadoras, ela também gostaria de ter uma para “circunferência da roda da bicicleta”. Ela tem um computador na bicicleta que calcula a velocidade; este computador tem que ser configurado com a circunferência da roda da bicicleta para fazer seu trabalho.

Esta é a informação que você estava esperando; é uma nova solicitação de recurso que fornece as informações exatas que você precisa para tomar a próxima decisão de projeto.

Você sabe que a circunferência de uma roda é PI vezes seu diâmetro. Seu incorporado Roda já calcula diâmetro; é simples adicionar um novo método para calcular a circunferência. Estas alterações são pequenas; a verdadeira mudança aqui é que agora seu aplicação tem uma necessidade explícita de uma classe Wheel que possa usar independentemente de Engrenagem. É hora de libertar Wheel para ser uma classe separada.

Porque você já isolou cuidadosamente o comportamento da roda dentro do Gear aula, essa mudança é indolor. Basta converter o Wheel Struct em um independente Classe Wheel e adicione o novo método de circunferência :

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel
3 def inicializar (roda dentada, roda dentada = zero)
4   @chainring = coroa
5   @cog       = engrenagem
6   @roda      = roda
7 fim
8
9 Proporção de 9 defesas
10 coroa / cog.to_f
11 fim
12
13 def gear_inches
14   relação * diâmetro da roda
15 fim
16 fim
17

```

Roda de 18 classes

```
19 attr_reader :rim, :tire
20
21 def inicializar(aro, pneu)
22     @aro = aro
23     @pneu = pneu
24 end
25
26 diâmetro de definição
27     aro + (pneu * 2)
28 end
29
30 circunferência definida
31     diâmetro * Matemática::PI
32 end
33 end
34
35 @wheel = Wheel.new(26, 1.5) 36 puts
@wheel.circumference 37 # -> 91.106186954104
38
39 coloca Gear.new(52, 11, @wheel).gear_inches 40 # -> 137.090909090909
41
42 coloca Gear.new(52, 11).ratio
43 # -> 4.72727272727273
```

Ambas as classes têm uma única responsabilidade. O código não é perfeito, mas de certa forma atinge um padrão mais elevado: é *bom o suficiente*.

Resumo O

caminho para software orientado a objetos mutável e sustentável começa com classes que têm uma única responsabilidade. Classes que fazem uma coisa *isolam* aquela coisa do resto do seu aplicativo. Este isolamento permite alterações sem consequências e reutilização sem duplicação.

CAPÍTULO 3

Gerenciando Dependências

As linguagens de programação orientadas a objetos afirmam que são eficientes e eficazes devido à forma como modelam a realidade. Os objetos refletem as qualidades de um problema do mundo real e as interações entre esses objetos fornecem soluções. Essas interações são inevitáveis. Um único objeto não pode saber tudo, então inevitavelmente terá que conversar com outro objeto.

Se você pudesse espiar um aplicativo ocupado e observar as mensagens enquanto elas passam, o tráfego poderia parecer excessivo. Há muita coisa acontecendo. No entanto, se nos afastarmos e adoptarmos uma visão global, um padrão torna-se óbvio. Cada mensagem é iniciada por um objeto para invocar algum comportamento. Todo o comportamento está disperso entre os objetos. Portanto, para qualquer comportamento desejado, um objeto o conhece pessoalmente, o herda ou conhece outro objeto que o conhece.

O capítulo anterior preocupou-se com o primeiro deles, ou seja, comportamentos que uma classe deveria implementar pessoalmente. O segundo, comportamento hereditário, será abordado no Capítulo 6, Adquirindo comportamento por meio da herança. Este capítulo trata do terceiro, obter acesso ao comportamento quando esse comportamento é implementado em outros objetos.

Como os objetos bem projetados têm uma única responsabilidade, a sua própria natureza exige que colaborem para realizar tarefas complexas. Esta colaboração é poderosa e perigosa. Para colaborar, um objeto deve saber algo sobre outros. *Saber* cria uma dependência. Se não forem gerenciadas com cuidado, essas dependências estrangularão seu aplicativo.

Compreendendo as dependências

Um objeto depende de outro objeto se, quando um objeto muda, o outro pode ser forçado a mudar por sua vez.

Aqui está uma versão modificada da classe Gear , onde Gear é inicializado com quatro argumentos familiares. O método gear_inches usa dois deles, aro e pneu, para crie uma nova instância de Wheel. A roda não mudou desde a última vez que você a viu Capítulo 2, Projetando classes com uma única responsabilidade.

```
1 classe de equipamento
2 attr_reader :chainring, :cog, :rim, :tire
3 def inicializar (roda dentada, roda dentada, aro, pneu)
4     @chainring = coroa
5     @cog      = engrenagem
6     @rim      = aro
7     @pneu      = pneu
8 fim
9
10 def gear_inches
11     relação * Roda.nova(aro, pneu).diâmetro
12 fim
13
14 Proporção de 14 defesas
15     coroa / cog.to_f
16 fim
17 #...
18 fim
19
20 Roda de 20 classes
21 attr_reader :rim, :tire
22 def inicializar(aro, pneu)
23     @aro      = aro
24     @pneu     = pneu
25 fim
26
27 diâmetro de definição
28     aro + (pneu * 2)
29 fim
30 #...
31 fim
32
33 Gear.new(52, 11, 26, 1,5).gear_inches
```

Examine o código acima e faça uma lista das situações em que o Gear seria forçado a mudar devido a uma mudança no Wheel. Este código parece inocente, mas é sorrateiramente complexo. Gear tem pelo menos quatro dependências de Wheel, enumeradas a seguir. A maioria das dependências é desnecessária; eles são um efeito colateral do estilo de codificação. O Gear não precisa deles para fazer seu trabalho. A própria existência deles enfraquece o Gear e torna mais difícil mudar.

Reconhecendo Dependências

Um objeto tem uma dependência quando conhece

- O nome de outra classe. Gear espera que exista uma classe chamada Wheel . • O nome de uma mensagem que pretende enviar a alguém que não seja você.
O Gear espera que uma instância Wheel responda ao diâmetro.
- Os argumentos que uma mensagem requer. Gear sabe que Wheel.new requer um aro e um pneu.
- A ordem desses argumentos. Gear conhece o primeiro argumento para Wheel.new deve ser aro, o segundo, pneu.

Cada uma dessas dependências cria uma chance de que o Gear seja forçado a mudar devido a uma mudança no Wheel. Algum grau de dependência entre essas duas classes é inevitável, afinal elas *devem* colaborar, mas a maioria das dependências listadas acima são desnecessárias. Essas dependências desnecessárias tornam o código menos *razoável*.

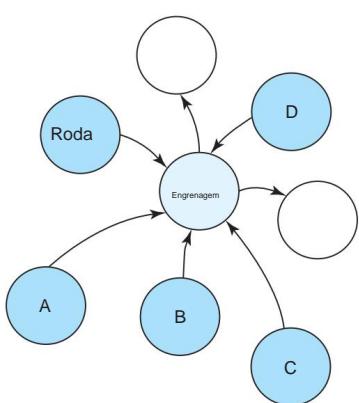
Como aumentam a chance de o Gear ser forçado a mudar, essas dependências transformam pequenos ajustes de código em grandes empreendimentos, onde pequenas mudanças se espalham pelo aplicativo, forçando muitas mudanças.

Seu desafio de design é gerenciar dependências para que cada classe tenha o menor número possível de dependências. possível; uma classe deve saber apenas o suficiente para fazer seu trabalho e nada mais.

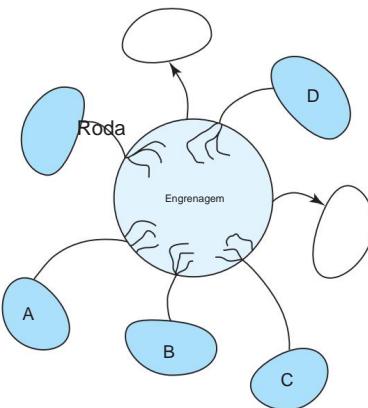
Acoplamento entre objetos (CBO)

Essas dependências *acoplam* o Gear ao Wheel. Alternativamente, você poderia dizer que cada acoplamento *cria* uma dependência. Quanto mais Gear sabe sobre Wheel, mais fortemente acoplados eles ficam. Quanto mais fortemente acoplados dois objetos estiverem, mais eles se comportarão como uma única entidade.

Se você fizer uma alteração no Wheel , poderá achar necessário fazer uma alteração no Gear. Se você quiser reutilizar o Gear, o Wheel vem junto. Ao testar o Gear, você também testará o Wheel .



A marcha depende da roda, A, B, C e D



Gear e suas dependências agem como uma coisa só

Figura 3.1 Dependências emaranham objetos entre si.

A Figura 3.1 ilustra o problema. Neste caso, Gear depende de Wheel e quatro outros objetos, acoplando o Gear a cinco coisas diferentes. Quando o código subjacente foi primeiro escrito tudo funcionou bem. O problema permanece adormecido até que você tente usar o Gear em outro contexto ou para alterar uma das classes nas quais o Gear depende. Quando esse dia chegar, a verdade dura e fria será revelada; apesar das aparências, Gear não é uma entidade independente. Cada uma de suas dependências é um lugar onde outro objeto está preso a ele. As dependências fazem com que esses objetos atuem como uma única coisa. Eles se movem em sincronia; eles mudam juntos.

Quando dois (ou três ou mais) objetos estão tão fortemente acoplados que se comportam como um unidade, é impossível reutilizar apenas uma. Mudanças em um objeto forçam mudanças em todos. Esquerda dependências não verificadas e não gerenciadas fazem com que um aplicativo inteiro se torne uma bagunça emaranhada. Chegará o dia em que será mais fácil reescrever tudo do que mudar alguma coisa.

Outras Dependências

O restante deste capítulo examina os quatro tipos de dependências listadas acima e sugere técnicas para evitar os problemas que eles criam. Porém, antes de ir adiante, vale a pena mencionar alguns outros problemas comuns relacionados à dependência que será abordado em outros capítulos.

Um tipo de dependência especialmente destrutivo ocorre quando um objeto conhece outro que conhece outro que sabe alguma coisa; isto é, onde muitas mensagens são encadeadas para alcançar o comportamento que vive em um objeto distante. Este é o “saber o nome de uma mensagem que você planeja enviar para alguém que não seja *auto*-dependência, apenas ampliado. O encadeamento de mensagens cria uma dependência entre o objeto original e

Escrevendo código fracamente acoplado

cada objeto e mensagem ao longo do caminho até seu alvo final. Esses acoplamentos adicionais aumentam muito a chance de o primeiro objeto ser forçado a mudar porque uma mudança em *qualquer* um dos objetos intermediários pode afetá-lo.

Este caso, uma violação da Lei de Deméter, recebe tratamento especial no Capítulo 4,

Criação de interfaces flexíveis.

Outra classe inteira de dependências é a dos testes de código. No mundo fora deste livro, os testes vêm em primeiro lugar. Eles impulsionam o design. No entanto, eles se referem ao código e, portanto, dependem do código. A tendência natural dos programadores “novatos em testes” é escrever testes fortemente acoplados ao código. Esse acoplamento forte leva a uma frustração incrível; os testes são interrompidos toda vez que o código é refatorado, mesmo quando o comportamento fundamental do código não muda. Os testes começam a parecer caros em relação ao seu valor.

O sobreacoplamento de teste para código tem a mesma consequência que o sobreacoplamento de código para código. Esses acoplamentos são dependências que fazem com que as alterações no código sejam transmitidas em cascata aos testes, forçando-os a mudar por sua vez.

O projeto de testes é examinado no Capítulo 9, Projetando testes com boa relação custo-benefício.

Apesar dessas palavras de advertência, seu aplicativo não está fadado a se afogar em dependências desnecessárias. Contanto que você os reconheça, evitá-los é bastante simples.

O primeiro passo para este futuro melhor é compreender as dependências com mais detalhes; portanto, é hora de examinar algum código.

Escrevendo código fracamente acoplado

Cada dependência é como um pontinho de cola que faz com que sua classe grude nas coisas que toca. Alguns pontos são necessários, mas aplique muita cola e sua aplicação endurecerá e se tornará um bloco sólido. Reduzir dependências significa reconhecer e remover aquelas de que você não precisa.

Os exemplos a seguir ilustram técnicas de codificação que reduzem dependências desacoplando o código.

Injetar Dependências Referir-se a

outra classe pelo seu nome cria um grande ponto pegajoso. Na versão do Gear que discutimos (repetida abaixo), o método gear_inches contém uma referência explícita à classe Wheel:

¹ classe de equipamento

² attr_reader :chainring, :cog, :rim, :tire ³ def inicializar(chainring, cog, rim, tire) ⁴ @chainring = chainring

⁴

```

5      @cog          = engrenagem
6      @rim          = aro
7      @pneu         = pneu
8 fim
9
10 def gear_inches
11     relação * Roda.nova(aro, pneu).diâmetro
12 fim
13 #...
14 fim
15
16 Gear.new(52, 11, 26, 1,5).gear_inches

```

A consequência imediata e óbvia desta referência é que se o nome do

A classe da roda muda, o método gear_inches do Gear também deve mudar.

À primeira vista, esta dependência parece inócuia. Afinal, se um Gear precisa

fale com uma Roda, algo, em algum lugar, deve criar uma nova instância da Roda
aula. Se o próprio Gear souber o nome da classe Wheel , o código no Gear deverá ser
alterado se o nome de Wheel mudar.

Na verdade, lidar com a mudança de nome é uma questão relativamente menor. Você provavelmente tem
uma ferramenta que permite fazer uma localização/substituição global dentro de um projeto. Se o nome da roda
mudanças no Wheely, encontrar e corrigir todas as referências não é tão difícil. No entanto,
o fato de que a linha 11 acima deve mudar se o nome da classe Wheel mudar é o
menos dos problemas com este código. Existe um problema mais profundo que é muito menos visível, mas
significativamente mais destrutivo.

Quando o Gear codifica uma referência ao Wheel profundamente dentro de seu gear_inches
método, ele está declarando explicitamente que só está disposto a calcular polegadas de engrenagem para
instâncias de Roda. Gear se recusa a colaborar com qualquer outro tipo de objeto, mesmo que
esse objeto tem diâmetro e usa engrenagens.

Se seu aplicativo se expandir para incluir objetos como discos ou cilindros e você
precisa saber as polegadas das engrenagens que as utilizam, você *não pode*. Apesar do fato
que discos e cilindros têm naturalmente um diâmetro que você nunca pode calcular sua engrenagem
polegadas porque o Gear está preso na roda.

O código acima expõe um anexo injustificado a tipos estáticos. Não é a
classe do objeto que é importante, é a *mensagem* que você planeja enviar para ele. Necessidades de equipamento
acesso a um objeto que pode responder ao diâmetro; um tipo de pato, se você quiser (veja
Capítulo 5, Reduzindo custos com Duck Typing). Gear não se importa e não deveria
saber sobre a classe desse objeto. Não é necessário que o Gear saiba sobre o
existência da classe Wheel para calcular gear_inches. Não precisa

Escrevendo código fracamente acoplado

saiba que a Roda espera ser inicializada com um aro e depois um pneu; só precisa de um objeto que conheça o diâmetro.

Pendurar essas dependências desnecessárias no Gear reduz simultaneamente a capacidade de reutilização do Gear e aumenta sua suscetibilidade de ser forçado a mudar desnecessariamente. O Gear se torna menos útil quando sabe muito sobre *outros* objetos; se soubesse menos, poderia fazer mais.

Em vez de ser colado ao Wheel, esta próxima versão do Gear espera ser inicializada com um objeto que possa responder ao diâmetro:

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel 3 def inicializar(chainring, cog, wheel)

4         @chainring = coroa @cog @wheel
5                     =
6                     engrenagem = roda
7 fim
8
9 def gear_inches
10    relação * diâmetro da roda
11 fim
12 #...
13 fim
14
15 # Gear espera um 'pato' que conheça o 'diâmetro' 16 Gear.new(52, 11, Wheel.new(26,
16.5)).gear_inches

```

O Gear agora usa a variável `@wheel` para armazenar e o método `wheel` para acessar esse objeto, mas não se engane, o Gear não sabe ou não se importa que o objeto possa ser uma instância da classe `Wheel`. O Gear só sabe que contém um objeto que responde ao diâmetro.

Essa mudança é tão pequena que é quase invisível, mas a codificação nesse estilo traz enormes benefícios. Mover a criação da nova instância `Wheel` para fora do Gear separa as duas classes. O Gear agora pode colaborar com qualquer objeto que implemente diâmetro. Como bônus extra, esse benefício era gratuito. Nenhuma linha adicional de código foi escrita; a dissociação foi alcançada reorganizando o código existente.

Essa técnica é conhecida como *injeção de dependência*. Apesar de sua reputação assustadora, a injeção de dependência é realmente simples assim. Anteriormente, o Gear tinha dependências explícitas na classe `Wheel` e no tipo e ordem de seus argumentos de inicialização, mas através da injeção essas dependências foram reduzidas a uma única dependência no método do diâmetro . O Gear agora é mais inteligente porque sabe menos.

Usar a injeção de dependência para moldar o código depende da sua capacidade de reconhecer que a responsabilidade de saber o nome de uma classe e a responsabilidade de saber o nome de uma mensagem a ser enviada para essa classe podem pertencer a objetos diferentes. Só porque o Gear precisa enviar o diâmetro para algum lugar não significa que o Gear deva saber sobre o Wheel.

Isso deixa a questão de onde reside a responsabilidade de saber sobre a classe Wheel real; o exemplo acima evita convenientemente essa questão, mas será examinado com mais detalhes posteriormente neste capítulo. Por enquanto, basta entender que esse conhecimento *não* pertence ao Gear.

Isolar Dependências É melhor

quebrar todas as dependências desnecessárias, mas, infelizmente, embora isso seja sempre tecnicamente possível, pode não ser *realmente* possível. Ao trabalhar em um aplicativo existente, você poderá se deparar com severas restrições sobre o quanto pode realmente ser alterado. Se for impedido de alcançar a perfeição, seus objetivos deverão mudar para melhorar a situação geral, deixando o código melhor do que você o encontrou.

Portanto, se você não puder remover dependências desnecessárias, deverá isolá-las dentro de sua classe. No Capítulo 2, Projetando classes com uma única responsabilidade, você isolou responsabilidades externas para que fossem fáceis de reconhecer e remover quando o impulso certo surgisse; aqui você deve isolar dependências desnecessárias para que sejam fáceis de detectar e reduzir quando as circunstâncias permitirem.

Pense em cada dependência como uma bactéria alienígena que está tentando infectar sua classe. Dê à sua turma um sistema imunológico vigoroso; colocar em quarentena cada dependência. As dependências são invasores estrangeiros que representam vulnerabilidades e devem ser concisas, explícitas e isoladas.

Isolar a criação de instância

Se você está tão restrito que não consegue alterar o código para injetar uma Wheel em um Gear, você deve isolar a criação de uma nova Wheel dentro da classe Gear . A intenção é expor explicitamente a dependência e, ao mesmo tempo, reduzir seu alcance na sua classe.

Os próximos dois exemplos ilustram essa ideia.

No primeiro, a criação da nova instância de Wheel foi movida do método gear_inches do Gear para o método de inicialização do Gear . Isso limpa o método gear_inches e expõe publicamente a dependência no método de inicialização . Observe que esta técnica cria incondicionalmente uma nova roda cada vez que uma nova engrenagem é criada.

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :rim, :tire
3 def inicializar (roda dentada, roda dentada, aro, pneu)
4     @chainring = coroa
5     @cog      = engrenagem
6     @rim      = aro
7     @pneu     = Wheel.new(aro, pneu)
8
9 def gear_inches
10    relação * diâmetro da roda
11  fim
12 #...

```

A próxima alternativa isola a criação de uma nova Roda em seu próprio espaço explicitamente definido. método da roda . Este novo método cria preguiçosamente uma nova instância de Wheel, usando Ruby ||= operador. Neste caso, a criação de uma nova instância de Wheel é adiada até gear_inches invoca o novo método wheel .

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :rim, :tire
3 def inicializar (roda dentada, roda dentada, aro, pneu)
4     @chainring = coroa
5     @cog      = engrenagem
6     @rim      = aro
7     @pneu     = pneu
8 fim
9
10 def gear_inches
11    relação * diâmetro da roda
12 fim
13
Roda 14 def
15     @wheel ||= Wheel.new(aro, pneu)
16 fim
17 #...

```

Em ambos os exemplos, Gear ainda sabe demais; ainda leva aro e pneu como argumentos de inicialização e ainda cria sua própria nova instância de Wheel. A engrenagem ainda está preso à roda; ele não pode calcular as polegadas de engrenagem de nenhum outro tipo de objeto.

No entanto, uma melhoria foi feita. Esses estilos de codificação reduzem o número de dependências em gear_inches enquanto expõe publicamente a dependência do Gear em

Roda. Eles revelam dependências em vez de ocultá-las, diminuindo as barreiras à reutilização e tornando o código mais fácil de refatorar quando as circunstâncias permitirem. Essa mudança torna o código mais ágil; pode adaptar-se mais facilmente ao futuro desconhecido.

A maneira como você gerencia dependências em nomes de classes externas tem efeitos profundos em seu aplicativo. Se você estiver atento às dependências e desenvolver o hábito de injetá-las rotineiramente, suas classes serão naturalmente fracamente acopladas. Se você ignorar esse problema e deixar as referências de classe caírem onde devem, seu aplicativo será mais parecido com um grande tapete tecido do que com um conjunto de objetos independentes. Um aplicativo cujas classes estão repletas de referências de nomes de classes emaranhadas e obscuras é pesado e inflexível, enquanto aquele cujas dependências de nomes de classes são concisas, explícitas e isoladas pode facilmente se adaptar a novos requisitos.

Isolar mensagens externas vulneráveis Agora

que você isolou as referências a nomes de classes externas, é hora de voltar sua atenção para *mensagens externas*, ou seja, mensagens que são “enviadas para alguém que não é você”. Por exemplo, o método `gear_inches` abaixo envia a proporção e a roda para si mesmo, mas envia o diâmetro para a roda:

```
1 def gear_inches
relação 2 * diâmetro da roda
3 fim
```

Este é um método simples e contém a única referência do Gear a `wheel.diameter`.

Neste caso o código está bom, mas a situação pode ser mais complexa. Imagine que calcular `gear_inches` exigisse muito mais matemática e que o método fosse mais ou menos assim:

```
1 def gear_inches 2 #...
algumas linhas de matemática assustadora 3 foo =
some_intermediate_result * wheel.diameter 4 #... mais linhas de matemática
assustadora
5 fim
```

Agora `wheel.diameter` está profundamente inserido em um método complexo. Este método complexo depende da resposta da engrenagem à roda e da resposta da roda ao diâmetro.

Incorporar essa dependência externa dentro do método `gear_inches` é desnecessário e aumenta sua vulnerabilidade.

Sempre que você muda *alguma coisa*, você tem a chance de quebrá-la; gear_inches agora é um método complexo e isso torna mais provável que ele precise ser alterado e mais suscetível a ser danificado quando isso acontecer. Você pode reduzir a chance de ser forçado a fazer uma alteração em gear_inches removendo a dependência externa e encapsulando-a em um método próprio, como neste próximo exemplo:

```
1 def gear_inches 2 #...
algunas linhas de matemática assustadora 3 foo =
some_intermediate_result * diâmetro 4 #... mais linhas de matemática
assustadora
5 fim
6
7 diâmetro definido
8 rodas.diâmetro
9 fim
```

O novo método do diâmetro é exatamente o método que você teria escrito se tivesse muitas referências a wheel.diameter espalhadas por todo o Gear e quisesse SECÁ-las. A diferença aqui é o tempo; normalmente seria defensável adiar a criação do método de diâmetro até que você precisasse secar o código; entretanto, neste caso o método é criado preventivamente para remover a dependência de gear_inches.

No código original, gear_inches sabia que aquela roda tinha um diâmetro. Esse conhecimento é uma dependência perigosa que acopla gear_inches a um objeto externo e um de *seus* métodos. Após essa mudança, gear_inches ficou mais abstrato. Gear agora isola wheel.diameter em um método separado e gear_inches pode depender de uma mensagem enviada para si mesmo.

Se a Roda alterar o nome ou assinatura de *sua* implementação de diâmetro, o os efeitos colaterais do Gear ficarão confinados a este método simples de embalagem.

Essa técnica se torna necessária quando uma classe contém referências incorporadas a uma *mensagem* que provavelmente será alterada. Isolar a referência fornece alguma segurança contra ser afetado por essa mudança. Embora nem todo método externo seja candidato a esse isolamento preventivo, vale a pena examinar seu código, procurando e agrupando as dependências mais vulneráveis.

Uma forma alternativa de eliminar esses efeitos colaterais é evitar o problema desde o início, invertendo a direção da dependência. Essa ideia será abordada em breve, mas primeiro há mais uma técnica de codificação a ser abordada.

Remover dependências de ordem de argumento

Ao enviar uma mensagem que requer argumentos, você, como remetente, não pode evitar ter conhecimento desses argumentos. Essa dependência é inevitável. No entanto, a passagem de argumentos geralmente envolve uma segunda dependência, mais útil. Muitas assinaturas de métodos não requerem apenas argumentos, mas também que esses argumentos sejam passados em uma ordem específica e fixa.

No exemplo a seguir, o método de inicialização do Gear leva três argumentos: coroa, engrenagem e roda. Ele não fornece padrões; cada um desses argumentos é obrigatório. Nas linhas 11–14, quando uma nova instância do Gear é criada, os três argumentos devem ser passados e *na ordem correta*.

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel
3 def inicializar(chainring, cog,
4   wheel)
5   @chainring = coroa @cog
6   @roda      = roda
7   fim
8   ...
9   fim
10
11 Engrenagem.novo(
12 52,
13 11,
14 Wheel.new(26, 1,5)).gear_inches

```

Os remetentes de new dependem da ordem dos argumentos conforme especificados no método de inicialização do Gear . Se essa ordem mudar, todos os remetentes serão forçados a mudar.

Infelizmente, é bastante comum mexer nos argumentos de inicialização.

Especialmente no início, quando o design ainda não está totalmente definido, você pode passar por vários ciclos de adição e remoção de argumentos e padrões. Se você usar argumentos de ordem fixa, cada um desses ciclos poderá forçar alterações em muitos dependentes. Pior ainda, você pode evitar fazer alterações nos argumentos, mesmo quando seu projeto exige isso, porque você não consegue alterar todos os dependentes novamente.

Use hashes para argumentos de inicialização

Há uma maneira simples de evitar depender de argumentos de ordem fixa. Se você tiver controle sobre o método de inicialização do Gear , altere o código para obter um hash de opções em vez de uma lista fixa de parâmetros.

Escrevendo código fracamente acoplado

O próximo exemplo mostra uma versão simples desta técnica. A inicialização O método agora leva apenas um argumento, args, um hash que contém todas as entradas. O método foi alterado para extrair seus argumentos deste hash. O hash em si é criado nas linhas 11–14.

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel
3 def inicializar (args)
4     @chainring = args[:chainring]
5     @cog       = argumentos[:cog]
6     @wheel = args[:wheel]
7 fim
8 ...
9 fim
10
11 Engrenagem.novo(
12 : coroa => 52,
13 : cog 14 :wheel      => 11,
=> Wheel.new(26, 1,5)).gear_inches

```

A técnica acima tem diversas vantagens. A primeira e mais óbvia é que remove todas as dependências da ordem dos argumentos. O Gear agora é gratuito para adicionar ou remover argumentos de inicialização e padrões, seguro de que nenhuma mudança ocorrerá tem efeitos colaterais em outro código.

Essa técnica adiciona verbosidade. Em muitas situações a verbosidade é prejudicial, mas em neste caso tem valor. A verbosidade existe na intersecção entre as necessidades do presente e a incerteza do futuro. Usar argumentos de ordem fixa requer menos código hoje, mas você paga por essa diminuição no volume de código com um aumento no risco que as alterações serão aplicadas em cascata aos dependentes posteriormente.

Quando o código na linha 11 mudou para usar um hash, ele perdeu a dependência do argumento ordem, mas ganhou uma dependência dos nomes das chaves no hash do argumento. Esse a mudança é saudável. A nova dependência é mais estável que a antiga e, portanto, este código enfrenta menos risco de ser forçado a mudar. Além disso, e talvez inesperadamente, o hash fornece um novo benefício secundário: os nomes -chave no hash fornecem informações explícitas documentação sobre os argumentos. Este é um subproduto do uso de um hash, mas o fato o fato de não ser intencional não o torna menos útil. Os futuros mantenedores deste código serão grato pela informação.

Os benefícios que você obtém usando esta técnica variam, como sempre, com base no seu situação pessoal. Se você estiver trabalhando em um método cuja lista de parâmetros é longa

e extremamente instável, em uma estrutura destinada a ser usada por outros, provavelmente reduzirá os custos gerais se você especificar argumentos em um hash. No entanto, se você estiver escrevendo um método para uso próprio que divide dois números, é muito mais simples e talvez mais barato simplesmente passar os argumentos e aceitar a dependência da ordem. Entre estes dois extremos encontra-se um caso comum, o do método que requer alguns argumentos muito estáveis e opcionalmente permite vários argumentos menos estáveis. Neste caso, a estratégia mais rentável poderá ser a utilização de ambas as técnicas; isto é, pegar alguns argumentos de ordem fixa, seguidos por um hash de opções.

Definir padrões explicitamente

Existem muitas técnicas para adicionar padrões. Padrões simples não booleanos podem ser especificados usando Ruby || método, como neste próximo exemplo:

```

1 # especificando padrões usando || 2 def
2   inicializar(args) @chainring =
3     args[:chainring] || 40 || 18 = args[:cog] = args[:roda]
4   @cog
5   @roda
6 end

```

Esta é uma técnica comum, mas que você deve usar com cautela; há situações em que ele pode não fazer o que você deseja. O || o método atua como uma condição ou ; ele primeiro avalia a expressão à esquerda e, em seguida, se a expressão retornar falso ou nulo, avalia e retorna o resultado da expressão à direita. O uso de || acima, portanto, depende do fato de que o método [] de Hash retorna nulo para chaves ausentes.

No caso em que args contém uma chave :boolean_thing cujo padrão é true, o uso de || dessa forma, torna impossível para o chamador definir explicitamente a variável final como falsa ou nula. Por exemplo, a seguinte expressão define @bool como true quando :boolean_thing está faltando e também quando está presente, mas definido como false ou nil:

```
@bool = args[:boolean_thing] || verdadeiro
```

Esta qualidade de || significa que se você usar valores booleanos como argumentos, ou usar argumentos onde você precisa distinguir entre falso e nil, é melhor usar o método fetch para definir padrões. O método fetch espera que a chave que você está buscando esteja no hash e fornece diversas opções para lidar explicitamente com chaves ausentes. Sua vantagem sobre || é que ele não retorna automaticamente nulo quando não consegue encontrar sua chave.

Escrevendo código fracamente acoplado

No exemplo abaixo, a linha 3 usa fetch para definir @chainring para o padrão, 40, somente se a chave :chainring não estiver no hash args . Definindo os padrões desta forma significa que os chamadores podem realmente fazer com que @chainring seja definido como falso ou nulo, algo isso não é possível ao usar o || técnica.

```

1 # especificando padrões usando fetch
2 def inicializar (args)
3     @chainring = args.fetch(:chainring, 40)
4     @cog       = args.fetch(:cog, 18)
5     @roda      = args[:roda]
6 fim

```

Você também pode remover completamente os padrões de inicialização e isolá-los dentro de um método de empacotamento separado. O método padrão abaixo define um segundo hash que é mesclado no hash de opções durante a inicialização. Neste caso, mescle tem o mesmo efeito que buscar; os padrões serão mesclados somente se suas chaves não estiverem em o haxixe.

```

1 # especificando padrões mesclando um hash de padrões
2 def inicializar (args)
3     args = padrões.merge(args)
4     @chainring = args[:chainring]
5 #
6 ...
6 fim
7
8 padrões de definição
9     {:anel de corrente => 40, :cog => 18}
10 fim

```

Esta técnica de isolamento é perfeitamente razoável para o caso acima, mas é especialmente útil quando os padrões são mais complicados. Se seus padrões são mais que simples números ou strings, implemente um método padrão .

Isolar inicialização multiparâmetro

Até agora, todos os exemplos de remoção de dependências de ordem de argumentos foram para situações em que você controla a assinatura do método que precisa ser alterado. Você nem sempre terá esse luxo; às vezes você será forçado a depender de um método que requer argumentos de ordem fixa onde você não possui e, portanto, não pode alterar o método em si.

Imagine que o Gear faz parte de um framework e que seu método de inicialização requer argumentos de ordem fixa. Imagine também que seu código tem muitos lugares onde você deve criar uma nova instância do Gear. O método de inicialização do Gear é *externo* ao sua aplicação; faz parte de uma interface externa sobre a qual você não tem controle.

Por mais terrível que esta situação pareça, você não está condenado a aceitar as dependências. Assim como você secaaria o código repetitivo dentro de uma classe, SECE a criação de novas instâncias do Gear criando um único método para encapsular a interface externa. O as classes em seu aplicativo devem depender do código que você possui; use um embrulho método para isolar dependências externas.

Neste exemplo, a classe `SomeFramework::Gear` não pertence ao seu aplicativo; faz parte de uma estrutura externa. Seu método de inicialização requer argumentos de ordem fixa. O módulo `GearWrapper` foi criado para evitar múltiplas dependências na ordem desses argumentos. `GearWrapper` isola todo o conhecimento do externo interface em um só lugar e, igualmente importante, fornece uma interface aprimorada para sua aplicação.

Como você pode ver na linha 24, o `GearWrapper` permite que sua aplicação crie um novo instância do Gear usando um hash de opções.

```

1 # Quando o Gear faz parte de uma interface externa
2 módulos SomeFramework
3
4     attr_reader :chainring, :cog, :wheel
5     def inicializar(roda dentada, roda dentada)
6         @chainring = coroa
7         @cog      = engrenagem
8         @roda    = roda
9     fim
10 ...
11 fim
12 fim
13
14 # envolva a interface para se proteger de alterações
15 GearWrapper de 15 módulos
16 def self.gear(args)
17     SomeFramework::Gear.new(args[:chainring],
18                             args[:cog],
19                             args[:roda])
20 fim
21 fim
22

```

```
23 # Agora você pode criar um novo Gear usando um hash de argumentos.
```

```
24 GearWrapper.gear ( 25 :
```

```
coroa => 52, 26 : engrenagem
```

```
=> 11,
```

```
27 : roda
```

```
=> Wheel.new(26, 1.5)).gear_inches
```

Há duas coisas a serem observadas sobre o GearWrapper. Primeiro, é um módulo Ruby em vez de uma classe (linha 15). GearWrapper é responsável por criar novas instâncias de SomeFramework::Gear. Usar um módulo aqui permite definir um objeto separado e distinto para o qual você pode enviar a mensagem gear (linha 24) e, ao mesmo tempo, transmitir a ideia de que você não espera ter instâncias do GearWrapper. Você já deve ter experiência na inclusão de módulos em aulas; no exemplo acima, GearWrapper não deve ser incluído em outra classe, mas sim responder diretamente à mensagem do equipamento .

A outra coisa interessante sobre o GearWrapper é que seu único propósito é criar instâncias de alguma outra classe. Os designers orientados a objetos têm uma palavra para objetos como este; eles os chamam *de fábricas*. Em alguns círculos, o termo fábrica adquiriu uma conotação negativa, mas o termo usado aqui é desprovido de bagagem. Um objeto cujo propósito é criar outros objetos é uma fábrica; a palavra fábrica não implica mais nada, e seu uso é a maneira mais conveniente de comunicar essa ideia.

A técnica acima para substituir um hash de opções por uma lista de argumentos de ordem fixa é perfeita para casos em que você é forçado a depender de interfaces externas que não pode alterar. Não permita que esses tipos de dependências externas permeiem seu código; proteja-se envolvendo cada um em um método que pertence ao seu próprio aplicativo.

Gerenciando a direção das dependências As

dependências sempre têm uma direção; anteriormente neste capítulo foi sugerido que uma maneira de gerenciá-los seria reverter essa direção. Esta seção se aprofunda em como decidir a direção das dependências.

Invertendo Dependências Todos os

exemplos usados até agora mostram a engrenagem dependendo da roda ou do diâmetro, mas o código poderia facilmente ter sido escrito com a direção das dependências invertida.

A roda poderia, em vez disso, depender da engrenagem ou da proporção. O exemplo a seguir ilustra uma forma possível de reversão. Aqui a Roda foi alterada para depender do Gear e

engrenagem_polegadas. Gear ainda é responsável pelo cálculo real, mas espera um argumento de diâmetro a ser passado pelo chamador (linha 8).

```

1 classe de equipamento
2 attr_reader :chainring, :cog
3 def inicializar (roda dentada, roda dentada)
4     @chainring = coroa
5     @cog           = engrenagem
6 fim
7
8 def gear_inches (diâmetro)
9     proporção * diâmetro
10 fim
11
Proporção de 12 defesas
13     coroa / cog.to_f
14 fim
15 #...
16 fim
17
Roda de 18 classes
19 attr_reader :rim, :tire, :gear
20 def inicializar (aro, pneu, coroa, roda dentada)
21     @aro = aro
22     @pneu          = pneu
23     @engrenagem    = Gear.new(coroa, roda dentada)
24 fim
25
26 diâmetro de definição
27     aro + (pneu * 2)
28 fim
29
30 def gear_inches
31     gear.gear_inches(diâmetro)
32 fim
33 #...
34 fim
35
36 Wheel.new(26, 1,5, 52, 11).gear_inches

```

Esta inversão de dependências não causa nenhum dano aparente. Calculando gear_inches ainda requer colaboração entre Gear e Wheel e o resultado do cálculo é

não afetado pela reversão. Poderíamos inferir que a direção da dependência não importa, que não faz diferença se Gear depende de Wheel ou vice-versa.

Na verdade, numa aplicação que nunca mudou, a sua escolha não importaria.

No entanto, a sua aplicação *irá* mudar e é nesse futuro dinâmico que esta decisão atual terá repercussões. As escolhas que você faz sobre a direção das dependências têm consequências de longo alcance que se manifestam durante a vida útil do seu aplicativo. Se você acertar, seu aplicativo será agradável de trabalhar e fácil de manter. Se você errar, as dependências assumirão gradualmente o controle e o aplicativo se tornará cada vez mais difícil de alterar.

Escolhendo a Direção da Dependência Finja por um

momento que suas turmas são pessoas. Se você lhes desse conselhos sobre como se comportar, você lhes diria para *dependerem de coisas que mudam com menos frequência do que você*.

Esta breve declaração desmente a sofisticação da ideia, que se baseia em três verdades simples sobre código:

- Algumas classes têm maior probabilidade do que outras de sofrer alterações nos requisitos.
- As classes concretas têm maior probabilidade de mudar do que as classes abstratas.
- Mudar uma classe que tem muitos dependentes resultará em consequências generalizadas.

Existem maneiras pelas quais essas verdades se cruzam, mas cada uma é uma noção separada e distinta.

Compreendendo a probabilidade de mudança A

ideia de que algumas classes têm maior probabilidade de mudar do que outras se aplica não apenas ao código que você escreve para seu próprio aplicativo, mas também ao código que você usa, mas não escreveu . As classes base Ruby e o outro código da estrutura em que você confia têm sua própria probabilidade inerente de mudança.

Você tem sorte porque as classes base do Ruby mudam com muito menos frequência do que o seu próprio código. Isso torna perfeitamente razoável depender do método *, como gear_inches silenciosamente faz, ou esperar que as classes String e Array do Ruby continuem a funcionar como sempre. As classes base Ruby sempre mudam com menos frequência do que suas próprias classes e você pode continuar a depender delas sem pensar duas vezes.

As aulas de estrutura são outra história; só você pode avaliar o quanto maduras são suas estruturas. Em geral, qualquer framework que você usar será mais estável que o código

você escreve, mas certamente é possível escolher uma estrutura que esteja passando por tal desenvolvimento rápido que seu código muda com mais frequência que o seu.

Independentemente de sua origem, cada classe usada em sua aplicação pode ser classificada uma escala de probabilidade de sofrer uma mudança em relação a todas as outras classes. Essa classificação é uma informação importante a ser considerada ao escolher a direção de dependências.

Reconhecendo Concreções e Abstrações

A segunda ideia preocupa-se com a concretude e a abstração do código. O

O termo *resumo* é usado aqui exatamente como o Merriam-Webster o define, como “desassociado de qualquer instância específica” e, como tantas coisas em Ruby, representa uma ideia sobre código em oposição a uma restrição técnica específica.

Este conceito foi ilustrado anteriormente no capítulo durante a seção sobre injeção dependências. Lá, quando o Gear dependia do Wheel e do Wheel.new e do Wheel.new(rim, tire), dependia de um código extremamente concreto. Depois do código foi alterado para injetar uma roda no Gear, o Gear de repente passou a depender de algo muito mais abstrato, ou seja, o fato de ter acesso a um objeto que poderia responder à mensagem de diâmetro .

Sua familiaridade com Ruby pode levá-lo a considerar essa transição algo natural, mas considere por um momento o que seria necessário para realizar esse mesmo troque em uma linguagem de tipo estaticamente. Como as linguagens de tipo estaticamente têm compiladores que agir como testes de unidade para tipos, você não seria capaz de injetar qualquer objeto aleatório em engrenagem. Em vez disso, você teria que declarar uma *interface*, definir o diâmetro como parte de essa interface, inclua a interface na classe Wheel e diga ao Gear que a classe que você estão injetando é um *tipo* dessa interface.

Os rubistas ficam justificadamente gratos por evitar essas oscilações, mas as linguagens que forçam você ser explícito sobre essa transição oferece um benefício. Eles fazem isso dolorosamente, inevitavelmente e explicitamente claro que você está definindo uma interface abstrata. É impossível criar uma abstração inconscientemente ou por acidente; em linguagens de tipo estaticamente definir uma interface é *sempre* intencional.

Em Ruby, quando você injeta Wheel no Gear de forma que o Gear dependa de um *Duck* quem responde ao diâmetro, você está, ainda que casualmente, definindo uma interface. Esta interface é uma abstração da ideia de que uma determinada categoria de coisas terá um diâmetro. A abstração foi colhida de uma classe concreta; a ideia está agora “desassociada de qualquer instância específica.”

O que há de maravilhoso nas abstrações é que elas representam qualidades. É menos provável que mudem do que as classes concretas das quais provêm.

foram extraídos. Depender de uma abstração é sempre mais seguro do que depender de uma concreção porque, por sua própria natureza, a abstração é mais estável. Ruby não faz você declarar explicitamente a abstração para definir a interface, mas para fins de design você pode se comportar como se sua interface virtual fosse tão real quanto uma classe. Na verdade, no restante desta discussão, o termo “classe” significa tanto *classe* quanto esse tipo de *interface*.

Essas interfaces podem ter dependentes e por isso devem ser levadas em consideração durante o projeto.

Evitando classes carregadas de dependência

A ideia final, a noção de que ter objetos carregados de dependência tem muitas consequências, também merece um exame mais profundo. As consequências de mudar uma classe carregada de dependência são bastante óbvias – não tão aparentes são as consequências de *ter* uma classe carregada de dependência. Uma classe que, se alterada, causará mudanças no aplicativo, estará sob enorme pressão para *nunca* mudar. Sempre. Em qualquer circunstância. Sua inscrição pode ser permanentemente prejudicada por sua relutância em pagar o preço exigido para fazer uma alteração nesta classe.

Encontrando as dependências que importam Imagine

cada uma dessas verdades como um continuum ao longo do qual se enquadra todo o código do aplicativo. As classes variam em sua probabilidade de mudança, seu nível de abstração e seu número de dependentes. Cada qualidade é importante, mas as decisões de projeto interessantes ocorrem no local onde a *probabilidade de mudança* cruza com o *número de dependentes*. Algumas das combinações possíveis são saudáveis para sua aplicação; outros são mortais.

A Figura 3.2 resume as possibilidades.

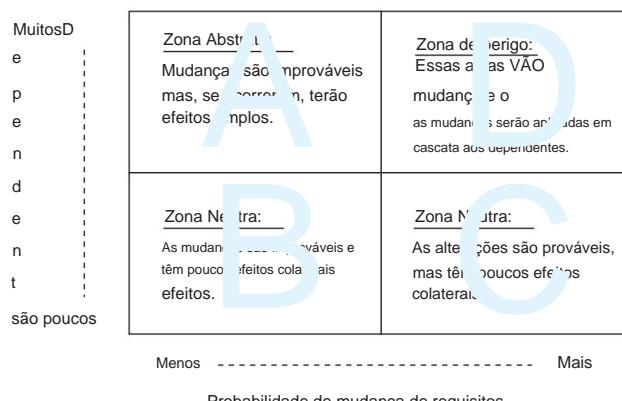


Figura 3.2 Probabilidade de mudança versus número de dependentes

A probabilidade de alteração dos requisitos é representada no eixo horizontal. O número de dependentes está na vertical. A grade é dividida em quatro zonas, rotuladas de A a D. Se você avaliar todas as classes em um aplicativo bem projetado e colocá-las nesta grade, elas serão agrupadas nas Zonas A, B e C.

Classes que têm pouca probabilidade de mudança, mas contêm muitos dependentes, caem na Zona A. Essa Zona geralmente contém classes ou interfaces abstratas. Numa aplicação cuidadosamente projetada, esse arranjo é inevitável; as dependências agrupam-se em torno de abstrações porque as abstrações têm menos probabilidade de mudar.

Observe que as classes não se tornam abstratas porque estão na Zona A; em vez disso, eles acabam aqui precisamente porque já são abstratos. A sua natureza abstrata torna-os mais estáveis e permite-lhes adquirir muitos dependentes com segurança. Embora a residência na Zona A não garanta que uma classe seja abstrata, certamente sugere que deveria ser.

Ignorando a Zona B por um momento, a Zona C é o oposto da Zona A. A Zona C contém código que provavelmente mudará, mas tem poucos dependentes. Essas classes tendem a ser mais concretas, o que as torna mais propensas a mudar, mas isso não importa porque poucas outras classes dependem delas.

As classes da Zona B são as que menos preocupam durante o projeto porque são quase neutras nos seus potenciais efeitos futuros. Eles raramente mudam e têm poucos dependentes.

As zonas A, B e C são locais legítimos para código; A Zona D, entretanto, é apropriadamente chamada de Zona de Perigo. Uma turma acaba na Zona D quando tem mudança garantida e tem muitos dependentes. As alterações nas classes da Zona D são caras; solicitações simples tornam-se pesadelos de codificação à medida que os efeitos de cada mudança se espalham por cada dependente. Se você tem uma classe *concreta* muito específica que tem muitos dependentes e acredita que ela reside na Zona A, ou seja, acredita que é pouco provável que ela mude, pense novamente. Quando uma classe concreta tem muitos dependentes, seus alarmes devem estar tocando. Essa classe pode na verdade ser um ocupante da Zona D.

As classes da Zona D representam um perigo para a saúde futura da aplicação. Essas são as classes que tornam difícil alterar um aplicativo. Quando uma simples mudança tem efeitos em cascata que forçam muitas outras mudanças, uma classe da Zona D está na raiz do problema. Quando uma mudança interrompe algum código distante e aparentemente não relacionado, a falha de design se originou aqui.

Por mais deprimente que isso seja, na verdade existe uma maneira de piorar as coisas. Você pode garantir que qualquer aplicativo se tornará gradualmente insustentável, tornando suas classes da Zona D *mais* propensas a mudar do que seus dependentes. Isso maximiza as consequências de cada mudança.

Resumo

Felizmente, compreender esta questão fundamental permite-lhe tomar medidas preventivas ação para evitar o problema.

Depender de coisas que mudam com menos frequência do que você é uma heurística que substitui todas as ideias desta seção. As zonas são uma forma útil de organizar seus pensamentos, mas na névoa do desenvolvimento pode não ser óbvio quais aulas vão para onde. Muitas vezes você está explorando o caminho para um projeto e a qualquer momento o futuro não está claro.

Seguir esta regra simples em todas as oportunidades fará com que seu aplicativo desenvolva um design saudável.

Resumo O

gerenciamento de dependências é fundamental para a criação de aplicativos preparados para o futuro. A injeção de dependências cria objetos fracamente acoplados que podem ser reutilizados de novas maneiras. O isolamento de dependências permite que os objetos se adaptem rapidamente a mudanças inesperadas. Depender de abstrações diminui a probabilidade de enfrentar essas mudanças.

A chave para gerenciar dependências é controlar sua direção. O caminho para a manutenção do nirvana é pavimentado com classes que dependem de coisas que mudam com menos frequência do que elas.

Esta página foi intencionalmente deixada em branco

CAPÍTULO 4

Criando Flexível Interfaces

É tentador pensar em aplicações orientadas a objetos como sendo a soma de suas classes. As aulas são muito visíveis; as discussões de design geralmente giram em torno das responsabilidades e dependências da classe. Classes são o que você vê em seu editor de texto e o que você faz check-in em seu repositório de código-fonte.

Há detalhes de design que devem ser capturados neste nível, mas uma aplicação orientada a objetos é mais do que apenas classes . É *composto por classes* , mas *definido* por mensagens. As classes controlam o que está no seu repositório de código-fonte; as mensagens refletem a aplicação viva e animada.

O design, portanto, deve preocupar-se com as mensagens que passam entre os objetos. Trata não apenas do que os objetos conhecem (suas responsabilidades) e quem eles conhecem (suas dependências), mas também de como eles conversam entre si. A conversa entre objetos ocorre através de suas *interfaces*; este capítulo explora a criação de interfaces flexíveis que permitem que os aplicativos cresçam e mudem.

Noções básicas sobre interfaces Imagine dois

aplicativos em execução, conforme ilustrado na Figura 4.1. Cada um consiste em objetos e nas mensagens que passam entre eles.

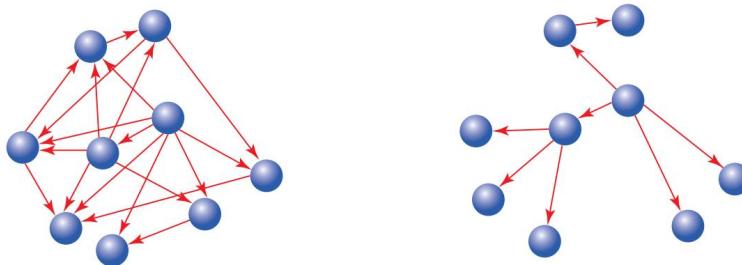


Figura 4.1 Padrões de comunicação.

Na primeira aplicação, as mensagens não apresentam padrão aparente. Todo objeto pode enviar qualquer mensagem para qualquer outro objeto. Se as mensagens deixassem rastros visíveis, esses rastros acabariam por desenhar uma esteira tecida, com cada objeto conectado entre si.

Na segunda aplicação, as mensagens possuem um padrão claramente definido. Aqui os objetos se comunicam de maneiras específicas e bem definidas. Se essas mensagens deixassem rastros, os rastros se acumulariam para criar um conjunto de ilhas com pontes ocasionais entre elas.

Ambas as aplicações, para melhor ou para pior, são caracterizadas pelos padrões de suas mensagens.

Os objetos da primeira aplicação são difíceis de reutilizar. Cada um se expõe demais e sabe demais sobre o próximo. Esse excesso de conhecimento resulta em objetos que são fina, explícita e desastrosamente ajustados para fazer apenas o que fazem agora. Nenhum objeto está sozinho; para reutilizar tudo que você precisa, para mudar uma coisa você deve mudar tudo.

A segunda aplicação é composta por objetos plugáveis, semelhantes a componentes. Cada revela o mínimo possível sobre si mesmo e sabe o mínimo possível sobre os outros.

O problema de design na primeira aplicação não é necessariamente uma falha na injeção de dependência ou responsabilidade única. Essas técnicas, embora necessárias, não são suficientes para impedir a construção de uma aplicação cujo design causa sofrimento. As raízes deste novo problema não residem no que cada classe *faz*, mas no que *revela*. Na primeira aplicação cada turma revela tudo. Cada método em qualquer classe pode ser invocado por qualquer outro objeto.

A experiência lhe diz que todos os métodos de uma classe não são iguais; alguns são mais gerais ou mais propensos a mudar do que outros. A primeira aplicação não percebe isso.

Permite que todos os métodos de qualquer objeto, independentemente de sua granularidade, sejam invocados por outros.

Na segunda aplicação, os padrões de mensagens são visivelmente limitados. Esta aplicação tem algum acordo, alguma barganha, sobre quais mensagens podem passar entre seus objetos. Cada objeto possui um conjunto claramente definido de métodos que espera que outros usem.

Esses métodos expostos constituem a *interface pública* da classe .

A palavra *interface* pode se referir a vários conceitos diferentes. Aqui o termo é usado para se referir ao tipo de interface que está *dentro de* uma classe. As classes implementam métodos, alguns desses métodos destinam-se a ser usados por outros e esses métodos constituem sua interface pública.

Um tipo alternativo de interface é aquele que abrange todas as classes e é independente de qualquer classe. Utilizada neste sentido, a palavra *interface* representa um conjunto de mensagens onde as próprias mensagens definem a interface. Muitas classes diferentes podem, como parte do seu todo, implementar os métodos que a interface requer. É quase como se a interface definisse uma classe *virtual* ; isto é, qualquer classe que implemente os métodos necessários pode agir como o tipo de *interface* .

O restante deste capítulo abordará o primeiro tipo de interface, ou seja, métodos dentro de uma classe e como e o que expor a outras classes. O Capítulo 5, Reduzindo Custos com Duck Typing, explora o segundo tipo de interface, aquela que representa um conceito mais amplo que uma classe e é definido por um conjunto de mensagens.

Definindo Interfaces Imagine a

cozinha de um restaurante. Os clientes pedem comida de um menu. Esses pedidos entram na cozinha por uma janelinha (aquele que tem a campainha ao lado, “pedido!”) e a comida acaba saindo. Para uma imaginação ingênua pode parecer que a cozinha está cheia de pratos mágicos de comida que estão esperando, ansioso por serem pedidos, mas na realidade a cozinha está cheia de pessoas, comida e atividade frenética e cada pedido desencadeia uma nova construção e processo de montagem.

A cozinha faz muitas coisas, mas felizmente não expõe todas elas aos seus clientes. Possui uma interface *pública* que se espera que os clientes utilizem; o cardápio.

Dentro da cozinha acontecem muitas coisas, muitas outras mensagens são passadas, mas essas mensagens são *privadas* e, portanto, invisíveis para os clientes. Mesmo que tenham feito o pedido, os clientes não são bem-vindos para entrar e mexer a sopa.

Esta distinção entre público e privado existe porque é a forma mais eficaz de fazer negócios. Se os clientes orientassem a culinária, teriam que ser reeducados sempre que a cozinha ficasse sem ingrediente e precisasse fazer uma substituição.

Usar um cardápio evita esse problema, pois permite que cada cliente peça o que deseja, sem saber nada sobre *como* a cozinha faz isso.

Cada uma de suas aulas é como uma cozinha. A classe existe para cumprir uma única responsabilidade, mas implementa muitos métodos. Esses métodos variam em escala e granularidade e vão desde métodos amplos e gerais que expõem a responsabilidade principal da classe at-

pequenos métodos utilitários que devem ser usados apenas internamente. Alguns desses métodos representam o menu da sua classe e devem ser públicos; outros tratam de detalhes de implementação interna e são privados.

Interfaces Públicas

Os métodos que compõem a interface pública da sua classe constituem a face que ela apresenta ao mundo. Eles:

- Revelar sua principal responsabilidade
- Espera-se que sejam invocados por outros
- Não mudará por capricho
- São seguros para que outros possam depender
- Estão minuciosamente documentados nos testes

Interfaces Privadas

Todos os outros métodos da classe fazem parte de sua interface privada. Eles:

- Lidar com detalhes de implementação
- Não se espera que sejam enviados por outros objetos
- Pode mudar por qualquer motivo
- Não são seguros para que outros dependam
- Pode nem ser referenciado nos testes

Responsabilidades, dependências e interfaces O Capítulo 2, Projetando

classes com uma única responsabilidade, tratou da criação de classes que têm uma única responsabilidade – um único propósito. Se você pensa que uma classe tem um único propósito, então as coisas que ela faz (suas responsabilidades mais específicas) são o que lhe permite cumprir esse propósito. Há uma correspondência entre as declarações que você pode fazer sobre essas responsabilidades mais específicas e os métodos públicos das classes.

Na verdade, os métodos públicos deveriam ser lidos como uma descrição de responsabilidades. A interface pública é um contrato que articula as responsabilidades da sua classe.

O Capítulo 3, Gerenciando Dependências, tratava de dependências e sua mensagem principal era que uma classe deveria depender apenas de classes que mudam com menos frequência do que ela.

faz. Agora que você está dividindo cada classe em uma parte pública e uma parte privada, essa ideia de depender de coisas menos mutáveis também se aplica aos métodos *dentro* de uma classe.

As partes públicas de uma classe são as partes estáveis; as partes privadas são as partes mutáveis. Ao marcar métodos como públicos ou privados, você informa aos usuários de sua classe de quais métodos eles podem depender com segurança. Quando suas classes usam métodos públicos de outras pessoas, você confia que esses métodos serão estáveis. Quando você decide depender dos métodos privados de outras pessoas, você entende que está confiando em algo que é inherentemente instável e, portanto, aumenta o risco de ser afetado por uma mudança distante e não relacionada.

Encontrando a Interface Pública Encontrar e definir

interfaces públicas é uma arte. Apresenta um desafio de design porque não existem regras definidas. Há muitas maneiras de criar interfaces “suficientemente boas” e os custos de uma interface “insuficientemente boa” podem não ser óbvios durante algum tempo, tornando difícil aprender com os erros.

O objetivo do design, como sempre, é manter o máximo de flexibilidade futura e ao mesmo tempo escrever apenas código suficiente para atender aos requisitos atuais. Boas interfaces públicas reduzem o custo de mudanças imprevistas; interfaces públicas ruins aumentam isso.

Esta seção apresenta um novo aplicativo para ilustrar uma série de regras práticas sobre interfaces e uma nova ferramenta que auxilia na sua descoberta.

Um exemplo de aplicação: Empresa de passeios de bicicleta Conheça a FastFeet, Inc., uma empresa de passeios de bicicleta. FastFeet oferece passeios de bicicleta de estrada e de montanha. FastFeet administra seus negócios usando um sistema de papel. Atualmente não possui nenhuma automação.

Cada viagem oferecida pela FastFeet segue um roteiro específico e pode ocorrer diversas vezes ao ano. Cada um tem limitações quanto ao número de clientes que podem ir e requer um número específico de guias que atuam como mecânicos.

Cada rota é avaliada de acordo com sua dificuldade aeróbica. Os passeios de mountain bike têm uma classificação adicional que reflete a dificuldade técnica. Os clientes têm um nível de condicionamento aeróbico e um nível de habilidade técnica em mountain bike para determinar se uma viagem é adequada para eles.

Os clientes podem alugar bicicletas ou optar por trazer as suas próprias. FastFeet tem algumas bicicletas disponíveis para aluguel ao cliente e também compartilha um pool de aluguel de bicicletas com lojas de bicicletas locais. As bicicletas de aluguel vêm em vários tamanhos e são adequadas para passeios de bicicleta de estrada ou de montanha.

Considere o seguinte requisito simples, que será referido posteriormente como um *caso de uso*: Um cliente, para escolher uma viagem, gostaria de ver uma lista de viagens disponíveis de dificuldade apropriada, em uma data específica, onde as bicicletas para aluguel estão disponíveis. .

Construindo uma intenção Começar com

o primeiro pedaço de código em um aplicativo totalmente novo é intimidante.

Ao adicionar código a uma base de código existente, geralmente você está estendendo um design existente. Aqui, porém, você deve colocar a caneta no papel (figurativamente) e tomar decisões que determinarão para sempre os padrões desta aplicação. O design que será ampliado posteriormente é aquele que você está estabelecendo agora.

Você sabe que não deve mergulhar e começar a escrever código. Você pode acreditar que deveria começar a escrever testes, mas essa crença não torna isso fácil. Muitos designers novatos têm sérias dificuldades em imaginar o primeiro teste. Escrever esse teste exige que você tenha uma ideia sobre o que deseja testar, algo que talvez ainda não tenha.

A razão pela qual os gurus que testam primeiro podem facilmente começar a escrever testes é que eles têm muita experiência em design. Nesta fase, já construíram um mapa mental de possibilidades de objetos e interações nesta aplicação. Eles não estão apegados a nenhuma ideia específica e planejam usar testes para descobrir alternativas, mas sabem tanto sobre design que já formaram uma intenção sobre a aplicação. É esta intenção que lhes permite especificar o primeiro teste.

Esteja você consciente delas ou não, você já formou algumas intenções próprias. A descrição dos negócios da FastFeet provavelmente lhe deu ideias sobre possíveis classes neste aplicativo. Você provavelmente espera ter aulas de Cliente, Viagem, Rota, Bicicleta e Mecânico.

Essas classes vêm à mente porque representam *substantivos* no aplicativo que possuem *dados* e *comportamento*. Chame-os de *objetos de domínio*. São óbvios porque são persistentes; eles representam coisas grandes e visíveis do mundo real que acabarão sendo representadas em seu banco de dados.

Os objetos de domínio são fáceis de encontrar, mas não estão no centro de design do seu aplicativo. Em vez disso, são uma armadilha para os incautos. Se você se fixar em objetos de domínio, tenderá a forçar o comportamento deles. Especialistas em design *percebem* objetos de domínio sem se concentrar neles; eles se concentram não nesses objetos, mas nas mensagens que passam entre eles. Essas mensagens são guias que levam você a descobrir outros objetos, igualmente necessários, mas muito menos óbvios.

Antes de se sentar ao teclado e começar a digitar, você deve formar uma intenção sobre os objetos e as mensagens necessárias para satisfazer este caso de uso. Você seria melhor atendido se tivesse uma maneira simples e barata de melhorar a comunicação para explorar o design, que não exigisse que você escrevesse código.

Felizmente, algumas pessoas muito inteligentes refletiram longamente sobre esta questão e criaram um mecanismo eficaz para fazer exatamente isso.

Usando diagramas de sequência Existe

uma maneira perfeita e de baixo custo de experimentar objetos e mensagens: *diagramas de sequência*.

Os diagramas de sequência são definidos na Unified Modeling Language (UML) e são um dos muitos diagramas suportados pela UML. A Figura 4.2 mostra uma amostra de alguns diagramas.

Se você adotou a UML com alegria, já conhece o valor dos diagramas de sequência. Se você não está familiarizado com UML e acha o gráfico alarmante, não se preocupe.

Este livro não está se transformando em um guia UML. O design leve e ágil não requer a criação e manutenção de pilhas de artefatos. No entanto, os criadores de

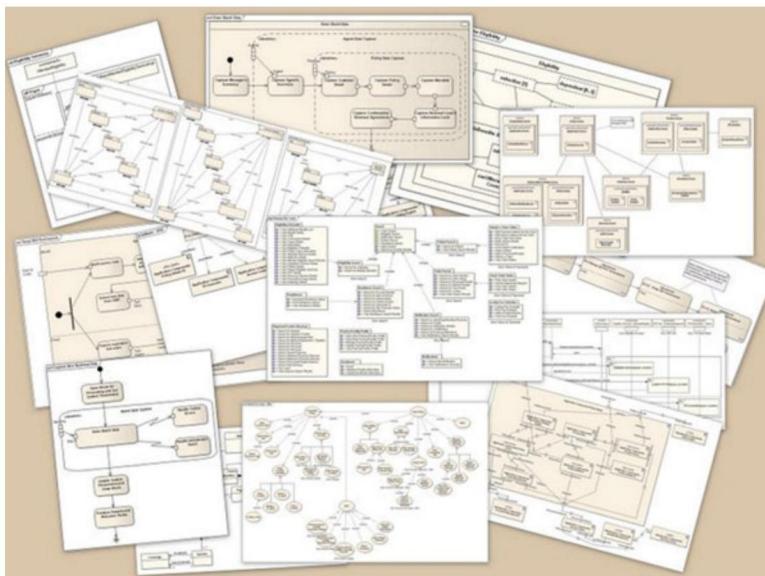


Figura 4.2 Exemplos de diagramas UML.

A UML pensou muito em como comunicar o design orientado a objetos e você pode aproveitar seus esforços. Existem diagramas UML que fornecem maneiras excelentes e transitórias de explorar e comunicar possibilidades de projeto. Usa-os; você não precisa reinventar esta roda.

Os diagramas de sequência são bastante úteis. Eles fornecem uma maneira simples de experimentar diferentes arranjos de objetos e esquemas de passagem de mensagens. Eles trazem clareza aos seus pensamentos e fornecem um veículo para colaborar e se comunicar com outras pessoas. Pense neles como uma forma leve de adquirir uma intenção sobre uma interação. Desenhe-os em um quadro branco, altere-os conforme necessário e apague-os quando cumprirem seu propósito.

A Figura 4.3 mostra um diagrama de sequência simples. Este diagrama representa uma tentativa de implementar o caso de uso acima. Mostra Moe, um Cliente e a classe Trip , onde Moe envia a mensagensuit_trips para Trip e recebe uma resposta.

A Figura 4.3 ilustra as duas partes principais de um diagrama de seqüência. Como você pode ver, eles mostram duas coisas: *objetos* e as *mensagens* que passam entre eles. Os parágrafos seguintes descrevem as partes deste diagrama, mas observe que a polícia da UML não irá prendê-lo se você divergir do estilo oficial. Faça o que for melhor para você.

No diagrama de exemplo, cada objeto é representado por duas caixas com nomes idênticos, dispostas uma acima da outra e conectadas por uma única linha vertical. Contém dois objetos, o Cliente Moe e a classe Trip . As mensagens são mostradas como linhas horizontais.

Quando uma mensagem é enviada, a linha é rotulada com o nome da mensagem. As linhas da mensagem terminam ou começam com uma seta; esta seta aponta para o receptor. Quando um objeto está ocupado processando uma mensagem recebida, ele fica *ativo* e sua linha vertical é expandida para um retângulo vertical.

O diagrama também contém uma única mensagem, suitable_trips , enviada por Moe para a classe Trip . Portanto, este diagrama de sequência pode ser lido da seguinte forma: O cliente Moe envia a mensagem adequa_trips para a classe Trip , que é acionada para processá-la e ao finalizar retorna uma resposta.

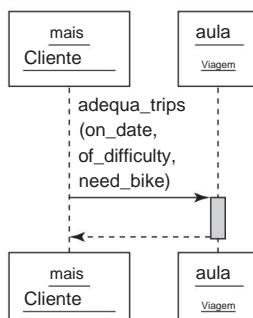


Figura 4.3 Um diagrama de sequência simples.

Este diagrama de sequência é quase uma tradução literal exata do caso de uso.

Os substantivos no caso de uso tornaram-se objetos no diagrama de sequência e a ação do caso de uso transformou-se em uma mensagem. A mensagem requer três parâmetros: `on_date`, `of_difficulty` e `need_bike`.

Embora este exemplo sirva de forma bastante adequada para ilustrar as partes de um diagrama de sequência, o design que ele implica deve fazer você pensar. Neste diagrama de sequência, Moe espera que a classe Trip encontre uma viagem adequada para ele. Parece razoável que Trip seja responsável por encontrar viagens para um encontro e uma dificuldade, mas Moe também pode precisar de uma bicicleta e ele claramente espera que Trip cuide disso também.

Desenhar este diagrama de sequência expõe a mensagem transmitida entre o Cliente Moe e a classe Trip e solicita que você faça a pergunta: “O Trip deveria ser responsável por descobrir se uma bicicleta apropriada está disponível para cada viagem adequada?” ou, de forma mais geral, “Este receptor deveria ser responsável por responder a esta mensagem?”

É aí que reside o valor dos diagramas de sequência. Eles especificam explicitamente as mensagens que passam entre objetos e, como os objetos só devem se comunicar usando interfaces públicas, os diagramas de sequência são um veículo para expor, experimentar e, em última análise, definir essas interfaces.

Além disso, observe agora que você desenhou um diagrama de sequência, esta conversa de design foi invertida. A ênfase do design anterior estava nas classes e em quem e o que elas sabiam. De repente, a conversa mudou; agora gira em torno de mensagens. Em vez de decidir sobre uma classe e depois descobrir suas responsabilidades, agora você está decidindo sobre uma mensagem e para onde enviá-la.

Esta transição do design baseado em classes para o design baseado em mensagens é um ponto de viragem na sua carreira de design. A perspectiva baseada em mensagens produz aplicações mais flexíveis do que a perspectiva baseada em classes. Mudar a questão fundamental do design de “Eu sei que preciso desta classe, o que ela deve fazer?” para “Preciso enviar esta mensagem, quem deve respondê-la?” é o primeiro passo nessa direção.

Você não envia mensagens porque tem objetos, você tem objetos porque envia mensagens.

Do ponto de vista da passagem de mensagens, é perfeitamente razoável que um Cliente envie a mensagem `suit_trips`. O problema não é que o Cliente não deva enviar, o problema é que a Trip não deva recebê-lo.

Agora que você tem a mensagem `suit_trips` em mente, mas não tem lugar para enviá-la, você deve construir algumas alternativas. Os diagramas de sequência facilitam a exploração das possibilidades.

Se a classe Trip não deveria estar descobrindo se há bicicletas disponíveis para uma viagem, talvez haja uma classe Bicycle que deveria. Trip pode ser responsável por `adequa_trips` e

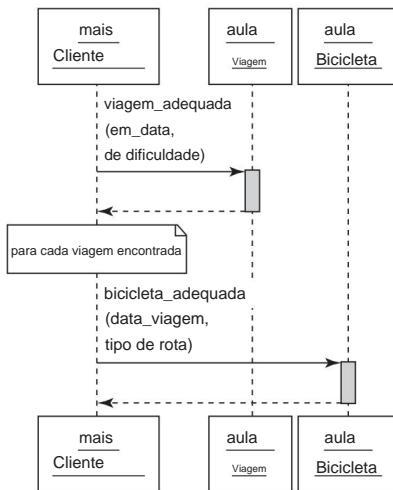


Figura 4.4 Moe fala com viagem e bicicleta.

Bicicleta para bicicleta adequada. Moe pode obter a resposta que precisa se conversar com os dois. Esse diagrama de sequência se parece com a Figura 4.4.

Para cada um desses diagramas, considere o que Moe precisa saber.

Na Figura 4.3, ele sabe que:

- Ele quer uma lista de viagens.
- Existe um objeto que implementa a mensagem `suit_trips`.

Na Figura 4.4, ele sabe que:

- Ele quer uma lista de viagens.
- Existe um objeto que implementa a mensagem `suit_trips`.
- Parte de encontrar uma viagem adequada significa encontrar uma bicicleta adequada.
- Existe outro objeto que implementa a mensagem `suit_bicycle`.

Infelizmente, a Figura 4.4 representa uma melhoria em algumas áreas, mas um fracasso em outras. Este design elimina responsabilidades externas da `Trip`, mas, infelizmente, apenas as transfere para o `Cliente`.

O problema na Figura 4.4 é que Moe não só sabe *o que* quer, mas também sabe *como* outros objetos devem colaborar para fornecê-lo. A classe `Cliente` tornou-se proprietária das regras de aplicação que avaliam a adequação da viagem.

Quando Moe sabe como decidir se uma viagem é adequada, ele não está ordenando o comportamento de um cardápio, ele está indo para a cozinha e cozinhando. A classe Cliente está cooptando responsabilidades que pertencem a outro lugar e vinculando-se a uma implementação que pode mudar.

Perguntar “o quê” em vez de dizer “como”

A distinção entre uma mensagem que pede o que o remetente quer e uma mensagem que diz ao receptor como se comportar pode parecer subtil, mas as consequências são significativas. Compreender esta diferença é uma parte fundamental da criação de classes reutilizáveis com interfaces públicas bem definidas.

Para ilustrar a importância do *quê* versus *como*, é hora de dar um exemplo mais detalhado. Deixe um pouco de lado o problema do design do cliente/viagem; ele retornará em breve.

Volte sua atenção para um novo exemplo envolvendo viagens, bicicletas e mecânicos.

Na Figura 4.5, uma viagem está prestes a partir e é necessário garantir que todas as bicicletas programadas para uso estejam em boas condições. O caso de uso para esse requisito é: Para começar, uma viagem precisa garantir que todas as suas bicicletas estejam mecanicamente corretas. Trip poderia saber exatamente como preparar uma bicicleta para uma viagem e poderia pedir a um mecânico para fazer cada uma dessas coisas:

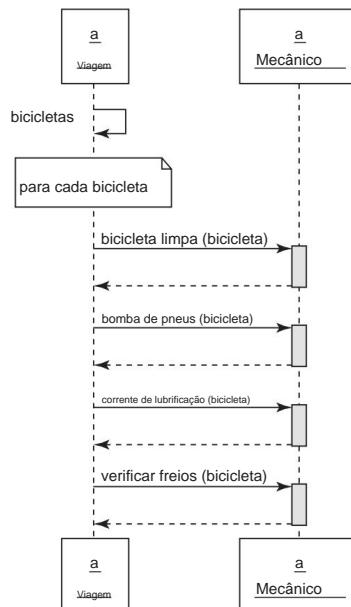


Figura 4.5A

Viagem conta um Mecânico

como preparar cada

Bicicleta

Na Figura 4.5:

- A interface pública do Trip inclui o método bicicletas.
- A interface pública do Mechanic inclui métodos clean_bicycle, pump_tires, lube_chain e check_brakes.
- Trip espera estar segurando um objeto que possa responder a clean_bicycle, pump_tires, lube_chain e check_brakes.

Neste design, Trip conhece muitos detalhes sobre o que o Mechanic faz. Como Trip contém esse conhecimento e o utiliza para direcionar o Mecânico, Trip deverá mudar se o Mecânico adicionar novos procedimentos ao processo de preparação da bicicleta. Por exemplo, se o Mecânico implementar um método para verificar o kit de reparo da bicicleta como parte da preparação da viagem , a viagem deverá mudar para invocar esse novo método.

A Figura 4.6 mostra uma alternativa onde Trip pede ao Mecânico para preparar cada Bicicleta, deixando os detalhes de implementação para o Mecânico.

Na Figura 4.6:

- A interface pública do Trip inclui o método bicicletas.
- A interface pública do Mechanic inclui o método prepare_bicycle.
- Trip espera estar segurando um objeto que possa responder a prepare_bicycle.

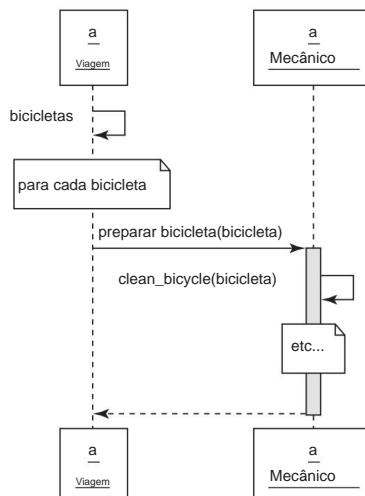


Figura 4.6A

Viagem pergunta um Mecânico para preparar cada

.

Trip agora renunciou a uma grande responsabilidade para o Mechanic. A Trip sabe que quer que cada uma de suas bicicletas esteja preparada e confia no Mecânico para realizar essa tarefa. Como a responsabilidade de saber *como* foi cedida ao Mecânico, Trip sempre obterá o comportamento correto, independentemente de futuras melhorias no Mecânico.

Quando a conversa entre Trip e Mechanic mudou de *como* para o *quê*, um efeito colateral foi que o tamanho da interface pública no Mechanic foi drasticamente reduzido. Na Figura 4.5 o Mechanic expõe vários métodos, na Figura 4.6 sua interface pública consiste em um único método, *prepare_bicycle*. Como o Mechanic promete que sua interface pública é estável e imutável, ter uma interface pública pequena significa que há poucos métodos dos quais outros possam depender. Isso reduz a probabilidade de algum dia o Mechanic mudar sua interface pública, quebrando sua promessa e forçando mudanças em muitas outras classes.

Essa mudança nos padrões de mensagens é uma grande melhoria na capacidade de manutenção do código, mas Trip ainda sabe muito sobre Mechanic. O código seria mais flexível e de fácil manutenção se Trip pudesse atingir seus objetivos sabendo ainda menos.

Buscando independência de contexto As coisas

que Trip sabe sobre outros objetos constituem seu *contexto*. Pense desta forma: Trip *tem* uma responsabilidade única, mas *espera* um contexto. Na Figura 4.6, Trip espera estar segurando um objeto Mechanic que possa responder à mensagem *prepare_bicycle*.

Contexto é um casaco que Trip usa em todos os lugares; qualquer uso do Trip, seja para testes ou outros, exige que seu contexto seja estabelecido. Preparar uma viagem *sempre* requer preparar bicicletas e ao fazer isso o Trip *sempre* envia a mensagem *prepare_bicycle* para seu Mecânico. Você não pode reutilizar Trip a menos que forneça um objeto do tipo Mecânico que possa responder a esta mensagem.

O contexto que um objeto espera tem um efeito direto na dificuldade de sua reutilização. Objetos que possuem um contexto simples são fáceis de usar e testar; eles esperam poucas coisas do ambiente. Objetos que possuem um contexto complicado são difíceis de usar e de testar; eles exigem uma configuração complicada antes de poderem fazer qualquer coisa.

A melhor situação possível é que um objeto seja completamente independente de seu contexto. Um objeto que pudesse colaborar com outros sem saber quem são ou o que fazem poderia ser reutilizado de maneiras novas e imprevistas.

Você já conhece a técnica para colaborar com outras pessoas sem saber quem elas são: injeção de dependência. O novo problema aqui é Trip invocar o comportamento correto do Mechanic sem saber o que o Mechanic faz. Trip quer colaborar com o Mechanic, mantendo a independência do contexto.

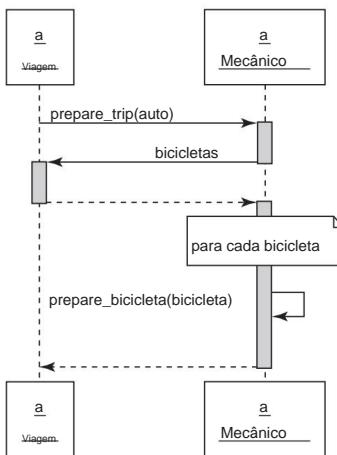


Figura 4.7A Viagem pergunta um Mecânico para preparar o Viagem.

À primeira vista isso parece impossível. As viagens têm bicicletas, as bicicletas devem estar preparadas, e mecânicos preparam bicicletas. Fazendo Trip pedir ao mecânico para preparar uma bicicleta parece inevitável.

No entanto, não é. A solução para este problema reside na distinção entre *o que* e *como*, e chegar a uma solução exige concentração no que Trip deseja.

O que Trip quer é estar preparado. O conhecimento de que deve ser preparado é completa e legitimamente dentro das responsabilidades da Trip . No entanto, o facto de as *bicicletas* necessitarem de ser preparadas pode pertencer à província de Mecânico. A necessidade de preparação da bicicleta está mais na forma como uma viagem é preparada do que no que Viagem quer.

A Figura 4.7 ilustra um terceiro diagrama de sequência alternativo para preparação de viagem . Neste exemplo, Trip apenas diz ao Mecânico o que quer, ou seja, estar preparado, e se apresenta como um argumento.

Neste diagrama de sequência, Trip não sabe nada sobre Mecânico , mas ainda consegue colaborar com ele para preparar as bicicletas. Trip diz ao Mecânico o que quer e passa sozinho como uma discussão, e Mechanic imediatamente liga de volta para Trip para obter o lista das bicicletas que precisam ser preparadas.

Na Figura 4.7:

- A interface pública do Trip inclui bicicletas.
- A interface pública do Mechanic inclui prepare_trip e talvez prepare_bicicleta.

- Trip espera estar segurando um objeto que possa responder a `prepare_trip`.
- O mecânico espera que o argumento passado junto com `prepare_trip` responda a bicicletas.

Todo o conhecimento sobre como os mecânicos preparam as viagens agora está isolado dentro do Mecânico e o contexto da Viagem foi reduzido. Ambos os objetos são agora mais fáceis de alterar, testar e reutilizar.

Confiando em outros objetos Os

designs ilustrados nas Figuras 4.5 a 4.7 representam um movimento em direção a um código cada vez mais orientado a objetos e, como tal, refletem os estágios de desenvolvimento do designer iniciante.

A Figura 4.5 é bastante processual. Um Trip diz a um Mecânico como preparar uma Bicicleta, quase como se o Trip fosse o programa principal e o Mecânico um monte de funções que podem ser chamadas. Neste design, Trip é o único objeto que sabe exatamente como preparar uma bicicleta; preparar uma bicicleta requer o uso de uma viagem ou a duplicação do código. O contexto do Trip é amplo, assim como a interface pública do Mechanic . Estas duas classes não são ilhas com pontes entre elas, mas sim um único tecido tecido.

Muitos novos programadores orientados a objetos começam trabalhando dessa maneira, escrevendo código processual. É inevitável; este estilo reflete de perto as melhores práticas de suas antigas linguagens procedurais. Infelizmente, a codificação em estilo processual anula o propósito da orientação a objetos. Ele reintroduz exatamente os problemas de manutenção que o OOP foi projetado para evitar.

A Figura 4.6 é mais orientada a objetos. Aqui, um Trip pede a um Mecânico que prepare uma Bicicleta. O contexto do Trip é reduzido e a interface pública do Mechanic é menor. Além disso, a interface pública do Mechanic agora é algo que qualquer objeto pode usar de forma lucrativa; você não precisa de uma viagem para preparar uma bicicleta. Esses objetos agora se comunicam de algumas maneiras bem definidas; eles são menos acoplados e mais facilmente reutilizáveis.

Esse estilo de codificação coloca as responsabilidades nos objetos corretos, uma grande melhoria, mas continua exigindo que o Trip tenha mais contexto do que o necessário.

Trip ainda sabe que mantém um objeto que pode responder a `prepare_bicycle` e deve *sempre* ter esse objeto.

A Figura 4.7 é muito mais orientada a objetos. Neste exemplo, Trip não sabe nem se importa que tenha um Mecânico e não tem ideia do que o Mecânico fará. Trip apenas mantém um objeto para o qual enviará `prepare_trip`; ele confia que o receptor desta mensagem se comportará de maneira adequada.

Expandindo essa ideia, Trip poderia colocar vários desses objetos em um array e enviar a mensagem `prepare_trip` para cada um, confiando em cada preparador para fazer o que quer que faça por causa do tipo de coisa que é. Dependendo de como o Trip estava sendo usado, ele poderia ter muitos preparadores ou poucos. Esse padrão permite adicionar preparadores recém-introduzidos ao Trip sem alterar nenhum código, ou seja, você pode *estender* o Trip sem *modificá-lo*.

Se os objetos fossem humanos e pudessem descrever suas próprias relações, na Figura 4.5 Trip estaria dizendo ao Mecânico: “Eu sei o que quero e sei como você faz isso”; na Figura 4.6: “Eu sei o que quero e sei o que você faz” e na Figura 4.7: “Eu sei o que quero e *confio em* você para fazer a sua parte”.

Essa confiança cega é a pedra angular do design orientado a objetos. Ele permite que os objetos colaborem sem se vincularem ao contexto e é necessário em qualquer aplicação que espera crescer e mudar.

Usando Mensagens para Descobrir Objetos Armado

com o conhecimento sobre a distinção entre *o quê* e *como*, e a importância do contexto e da confiança, é hora de retornar ao problema de design original das Figuras 4.3 e 4.4.

Lembre-se de que o caso de uso para esse problema dizia: Um cliente, para escolher uma viagem, gostaria de ver uma lista de viagens disponíveis de dificuldade apropriada, em uma data específica, onde há bicicletas para aluguel disponíveis.

A Figura 4.3 foi uma tradução literal deste caso de uso, no qual Trip tinha muita responsabilidade. A Figura 4.4 foi uma tentativa de transferir a responsabilidade de encontrar bicicletas disponíveis de Viagem para Bicicleta, mas ao fazê-lo impôs ao Cliente a obrigação de saber demais sobre o que torna uma viagem “adequada”.

Nenhum desses designs é muito reutilizável ou tolerante a mudanças. Esses problemas são revelados, inevitavelmente, nos diagramas de sequência. Ambos os designs contêm uma violação do princípio da responsabilidade única. Na Figura 4.3, Trip sabe demais. Na Figura 4.4, o Cliente sabe demais, diz aos outros objetos como se comportar e requer muito contexto.

É completamente razoável que o Cliente envie a mensagem `suit_trips`. Essa mensagem se repete em ambos os diagramas de sequência porque parece intrinsecamente correta. É exatamente o que o Cliente deseja. O problema não está no remetente, está no destinatário. Você ainda não identificou um objeto cuja responsabilidade seja implementar este método.

Esta aplicação necessita de um objeto para incorporar as regras na interseção Cliente, Viagem e Bicicleta. O método `adequa_trips` fará parte de sua interface pública.

A compreensão de que você precisa de um objeto ainda indefinido pode ser alcançada por meio de muitas rotas. A vantagem de descobrir esse objeto perdido por meio de diagramas de sequência é que o custo de estar errado é muito baixo e os impedimentos para mudar de ideia são extremamente poucos. Os diagramas de sequência são experimentais e serão descartados; sua falta de apego a eles é uma característica. Eles não refletem o seu design final, mas em vez disso criam uma intenção que é o ponto de partida para o seu design.

Independentemente de como você chega a esse ponto, agora está claro que você precisa de um novo objeto, um objeto que você descobriu devido à necessidade de enviar-lhe uma mensagem.

Talvez o aplicativo deva conter uma classe `TripFinder`. A Figura 4.8 mostra um diagrama de sequência onde um `TripFinder` é responsável por encontrar viagens adequadas.

`TripFinder` contém todo o conhecimento sobre o que torna uma viagem adequada. Conhece as regras; a sua função é fazer o que for necessário para responder a esta mensagem. Ele fornece um

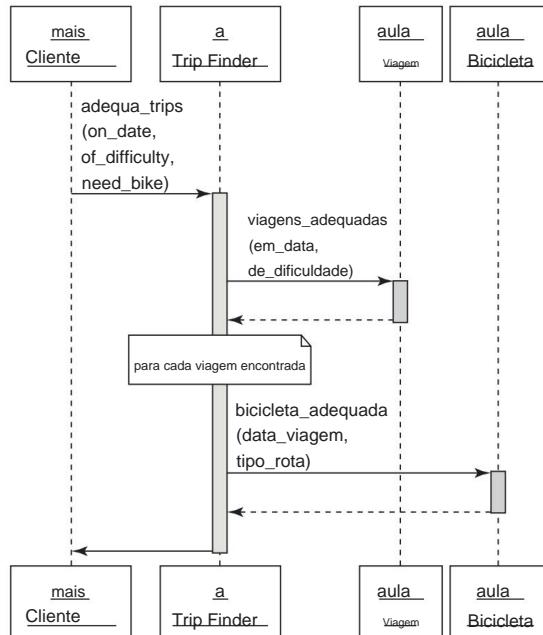


Figura 4.8 Moe pergunta ao Trip Finder para uma viagem adequada.

interface pública consistente enquanto oculta detalhes de implementação interna confusos e mutáveis.

Mover esse método para o TripFinder disponibiliza o comportamento para qualquer outro objeto. Num futuro desconhecido, talvez outras empresas de turismo utilizem o TripFinder para localizar viagens adequadas através de um serviço Web. Agora que este comportamento foi extraído de Customer, ele pode ser utilizado, isoladamente, por qualquer outro objeto.

Criando um aplicativo baseado em mensagens

Esta seção

usou diagramas de sequência para explorar o design, definir interfaces públicas e descobrir objetos.

Os diagramas de sequência são extremamente úteis de forma transitória; eles tornam compreensíveis conversas que de outra forma seriam impossivelmente complicadas. Volte pelas últimas páginas e imagine tentar esta discussão sem elas.

Por mais úteis que sejam, são uma ferramenta e nada mais. Eles ajudam a manter o foco nas mensagens e permitem formar uma intenção racional sobre a primeira coisa a ser afirmada em um teste. Mudar sua atenção dos objetos para as mensagens permite que você se concentre no design de um aplicativo baseado em interfaces públicas.

Escrevendo código que apresenta sua melhor (inter) face

A clareza de suas interfaces revela suas habilidades de design e reflete sua autodisciplina. Como as habilidades de design estão sempre melhorando, mas nunca são aperfeiçoadas, é porque mesmo o belo design de hoje pode parecer feio à luz das exigências de amanhã, é difícil criar interfaces perfeitas.

Isso, entretanto, não deve impedi-lo de tentar. As interfaces evoluem e para isso devem primeiro nascer. É mais importante que exista uma interface bem definida do que perfeita.

Pense em interfaces. Crie-os intencionalmente. São suas interfaces, mais do que todos os seus testes e qualquer código, que definem sua aplicação e determinam seu futuro.

A seção a seguir contém regras básicas para a criação de interfaces.

Crie interfaces explícitas

Seu objetivo

é escrever código que funcione hoje, que possa ser facilmente reutilizado e que possa ser adaptado para uso inesperado no futuro. Outras pessoas invocarão seus métodos; é sua obrigação comunicar quais são confiáveis.

Cada vez que você criar uma classe, declare suas interfaces. Os métodos na interface *pública* devem

- Ser explicitamente identificado como tal
- Seja mais sobre o *quê* do que *como*
- Tenha nomes que, até onde você possa prever, não mudarão
- Use um hash como parâmetro de opções

Seja igualmente intencional em relação à interface privada; torná-lo inevitavelmente óbvio. Os testes, por servirem como documentação, podem apoiar esse esforço. Não teste métodos privados ou, se necessário, separe esses testes dos testes de métodos públicos.

Não permita que seus testes engane outras pessoas, dependendo involuntariamente da interface privada e mutável.

Ruby fornece três palavras-chave relevantes: público, protegido e privado. O uso dessas palavras-chave atende a dois propósitos distintos. Primeiro, eles indicam quais métodos são estáveis e quais são instáveis. Segundo, eles controlam a visibilidade de um método para outras partes do seu aplicativo. Esses dois propósitos são muito diferentes. Transmitir informações de que um método é estável ou instável é uma coisa; tentar controlar como os outros o usam é outra bem diferente.

Palavras-chave públicas, protegidas e privadas A

palavra-chave privada denota o tipo de método menos estável e fornece a visibilidade mais restrita. Os métodos privados devem ser chamados com um receptor implícito ou, inversamente, nunca podem ser chamados com um receptor explícito.

Se a classe Trip contiver o método privado fun_factor, você não poderá enviar self.fun_factor de Trip ou a_trip.fun_factor de outro objeto. No entanto, você pode enviar fun_factor, padronizando self (o receptor implícito) de dentro de instâncias de Trip e suas subclasses.

A palavra-chave protegida também indica um método instável, mas com restrições de visibilidade ligeiramente diferentes. Os métodos protegidos permitem receptores explícitos, desde que o receptor seja self ou uma instância da mesma classe ou subclasse de self.

Assim, se o método fun_factor do Trip estiver protegido, você sempre poderá enviar self.fun_factor. Além disso, você pode enviar a_trip.fun_factor,

mas apenas dentro de uma classe onde self é o mesmo tipo de coisa (classe ou subclasse) que a_trip.

A palavra-chave public indica que um método é estável; métodos públicos são visíveis em todos os lugares.

Para complicar ainda mais as coisas, Ruby não apenas fornece essas palavras-chave, mas também fornece vários mecanismos para contornar as restrições de visibilidade impostas por private e protected . Os usuários de uma classe podem redefinir qualquer método para público , independentemente de sua declaração inicial. As palavras-chave privadas e protegidas são mais barreiras flexíveis do que restrições concretas. Qualquer um pode passar por eles; é simplesmente uma questão de despender esforço.

Portanto, qualquer noção de que você possa impedir o acesso ao método usando essas palavras-chave é uma ilusão. As palavras-chave não negam acesso, apenas dificultam um pouco. Usá-los envia duas mensagens:

- Você acredita que tem *hoje* melhores informações do que os programadores terão *em o futuro*.
- Você acredita que esses futuros programadores precisam ser impedidos de usar acidentalmente um método que você atualmente considera instável.

Estas crenças podem estar corretas, mas o futuro está muito distante e nunca se pode ter certeza. Os métodos aparentemente mais estáveis podem mudar regularmente e os inicialmente mais instáveis podem sobreviver ao teste do tempo. Se a ilusão de controle for um conforto, fique à vontade para usar as palavras-chave. No entanto, muitos programadores Ruby perfeitamente competentes os omitem e, em vez disso, usam comentários ou uma convenção especial de nomenclatura de métodos (Ruby on Rails, por exemplo, adiciona um '_' inicial aos métodos privados) para indicar as partes *públicas* e *privadas* das interfaces.

Estas estratégias são perfeitamente aceitáveis e, por vezes, até preferíveis. Eles fornecem informações sobre a estabilidade do método sem impor restrições de visibilidade.

O uso deles permite que futuros programadores façam boas escolhas sobre quais métodos confiar com base no aumento de informações que possuem naquele momento.

Independentemente de como você escolha fazê-lo, desde que encontre alguma forma de transmitir essas informações, você terá cumprido suas obrigações para com o futuro.

Honre as interfaces públicas de outras pessoas

Faça o possível para interagir com outras classes usando apenas suas interfaces públicas. Suponha que os autores dessas classes foram tão intencionais quanto você agora e estão tentando desesperadamente, através do tempo e do espaço, comunicar quais métodos são

confiável. As distinções público/privado que eles fizeram têm como objetivo ajudá -lo e é melhor prestar atenção a elas.

Se o seu design forçar o uso de um método privado em outra classe, primeiro repense o seu design. É possível que um esforço empenhado revele uma alternativa; você deve se esforçar muito para encontrar um.

Quando você depende de uma interface privada, aumenta o risco de ser forçado a mudar. Quando essa interface privada faz parte de um framework externo que passa por lançamentos periódicos, essa dependência é como uma bomba-relógio que explodirá no pior momento possível. Inevitavelmente, a pessoa que criou a dependência parte para pastagens mais verdes, a estrutura externa é atualizada, o método privado depende de mudanças e o aplicativo quebra de uma forma que confunde os mantenedores atuais.

A dependência de um método privado de uma estrutura externa é uma forma de dívida técnica. Evite essas dependências.

Tenha cuidado ao depender de interfaces privadas Apesar de seus melhores esforços, você poderá descobrir que *precisa* depender de uma interface privada. Esta é uma dependência perigosa que deve ser isolada usando as técnicas descritas no Capítulo 3. Mesmo que você não possa evitar o uso de um método privado, você pode evitar que o método seja referenciado em muitos lugares da sua aplicação. Depender de uma interface privada aumenta o risco; mantenha esse risco ao mínimo, isolando a dependência.

Minimize o contexto

Construa interfaces públicas com o objetivo de minimizar o contexto que exigem de outras pessoas. Tenha em mente a distinção entre o *quê* e *como* ; crie métodos públicos que permitam aos remetentes obter o que desejam sem saber como sua classe implementa seu comportamento.

Por outro lado, não sucumba a uma classe que tenha uma interface pública mal definida ou ausente. Ao se deparar com uma situação como a da classe Mechanic da Figura 4.5, não desista e diga-lhe como se comportar invocando todos os seus métodos. Mesmo que o autor original não tenha definido uma interface pública, não é tarde demais para criar uma para você.

Dependendo de quantas vezes você planeja usar essa nova interface pública, pode ser um novo método que você define e coloca na classe Mechanic , uma nova classe wrapper que você cria e usa em vez de Mechanic, ou um único método de empacotamento que você coloca em sua própria classe. Faça o que melhor atende às suas necessidades, mas crie algum tipo de interface pública definida e use-a. Isso reduz o contexto da sua classe, tornando-a mais fácil de reutilizar e mais simples de testar.

A Lei de Deméter

Depois de ler sobre responsabilidades, dependências e interfaces, você agora está equipado para explorar a Lei de Deméter.

A Lei de Demeter (LoD) é um conjunto de regras de codificação que resulta em objetos fracamente acoplados. O fraco acoplamento é quase sempre uma virtude, mas é apenas um componente do projeto e deve ser equilibrado com necessidades concorrentes. Algumas violações do Demeter são inofensivas, mas outras expõem uma falha na identificação e definição correta de interfaces públicas.

Definindo Demeter

restringe o conjunto de objetos para os quais um método pode *enviar* mensagens; proíbe o roteamento de uma mensagem para um terceiro objeto por meio de um segundo objeto de tipo diferente. Deméter é frequentemente parafraseada como “fale apenas com seus vizinhos imediatos” ou “use apenas um ponto”.

Imagine que o método `depart` do `Trip` contém cada uma das seguintes linhas de código:

```
cliente.bicicleta.roda.pneu
```

```
cliente.bicicleta.roda.rotate
```

```
hash.keys.sort.join(',')
```

Cada linha é uma cadeia de mensagens contendo vários pontos (pontos). Essas correntes são coloquialmente chamadas de *desastres de trem*; cada nome de método representa um vagão de trem e os pontos são as conexões entre eles. Esses trens são uma indicação de que você pode estar violando Demeter.

Consequências das Violações Deméter tornou-se

uma “lei” porque um ser humano assim decidiu; não se deixe enganar por seu nome grandioso. Como lei, é mais como “passar fio dental todos os dias” do que como a gravidade.

Você pode preferir não confessar ao seu dentista, mas violações ocasionais não causarão o colapso do universo.

O Capítulo 2 afirmou que o código deve ser *transparente*, *razoável*, *utilizável* e *exemplar*.

jogar. Algumas das cadeias de mensagens acima falham quando avaliadas em relação a TRUE:

- Se a roda trocar de pneu ou girar, a partida poderá ter que ser trocada. O `Trip` não tem nada a ver com a roda, mas alterações na roda podem forçar alterações no `Trip`. Isto aumenta desnecessariamente o custo da mudança; o código não é *razoável*.
- Trocar pneu ou girar pode quebrar alguma coisa na partida. Como `Trip` é distante e aparentemente não relacionado, o fracasso será completamente inesperado. Este código não é *transparente*.

- A viagem não pode ser reutilizada a menos que tenha acesso a um cliente com uma bicicleta que tenha roda e pneu. Requer muito contexto e não é facilmente *utilizável*.
- Este padrão de mensagens será replicado por outros, produzindo mais código com problemas semelhantes. Esse estilo de código, infelizmente, se reproduz sozinho. Não é *exemplar*.

As duas primeiras cadeias de mensagens são quase idênticas, diferindo apenas porque uma recupera um atributo distante (pneu) e a outra invoca um comportamento distante (rotação). Até mesmo designers experientes discutem sobre a firmeza com que Demeter se aplica a cadeias de mensagens que retornam *atributos*. Pode ser mais barato, *no seu caso específico*, alcançar objetos intermediários para recuperar atributos distantes. Equilibre a probabilidade e o custo da mudança com o custo da remoção da violação. Se, por exemplo, você estiver imprimindo um relatório de um conjunto de objetos relacionados, a estratégia mais racional poderá ser especificar explicitamente os objetos intermediários e alterar o relatório se for necessário. Como o risco incorrido pelas violações da Demeter é baixo para atributos estáveis, esta pode ser a estratégia mais rentável.

Essa compensação é permitida desde que você não altere o valor do atributo recuperado. Se o depart envia customer.bicycle.wheel.tire com a intenção de alterar o resultado, não está apenas recuperando um atributo, está implementando um comportamento que pertence ao Wheel. Nesse caso, customer.bicycle.wheel.tire torna-se igual a customer.bicycle.wheel.rotate; é uma cadeia que atravessa muitos objetos para chegar a um comportamento distante. O custo inerente deste estilo de codificação é alto; esta violação deve ser removida.

A terceira cadeia de mensagens, hash.keys.sort.join é perfeitamente razoável e, apesar da abundância de pontos, pode não ser uma violação do Demeter. Em vez de avaliar esta frase contando os “pontos”, avalie-a verificando os tipos dos objetos intermediários.

hash.keys retorna um enumerável

hash.keys.sort também retorna um Enumerable

hash.keys.sort.join retorna uma String

Por esta lógica, há uma leve violação de Deméter. No entanto, se você aceitar que hash.keys.sort.join realmente retorna um Enumerable de Strings, todos os objetos intermediários têm o mesmo tipo e não há violação de Demeter. Se você remover os pontos *desta* linha de código, seus custos poderão aumentar em vez de diminuir.

Como você pode ver, Deméter é mais sutil do que parece à primeira vista. As suas regras fixas não são um fim em si mesmas; como todo princípio de design, ele existe a serviço de seus objetivos gerais. Certas “violações” do Demeter reduzem a flexibilidade e a capacidade de manutenção da sua aplicação, enquanto outras fazem todo o sentido.

Evitando violações

Uma maneira comum de remover “desastres de trem” do código é usar a delegação para evitar os “pontos”. Em termos de orientação a objetos, *delegar* uma mensagem é passá-la para outro objeto, geralmente por meio de um método wrapper. O método wrapper encapsula ou oculta o conhecimento que de outra forma seria incorporado na cadeia de mensagens.

Existem várias maneiras de realizar a delegação. Ruby contém `delegated.rb` e `forwardable.rb` e a estrutura Ruby on Rails inclui o método `delegado`. Cada um deles existe para tornar mais fácil para um objeto interceptar automaticamente uma mensagem enviada para *si mesmo* e, em vez disso, enviá-la para outro lugar.

A delegação é uma solução tentadora para o problema de Deméter porque remove as evidências visíveis de violações. Essa técnica às vezes é útil, mas cuidado, ela pode resultar em um código que obedece à letra da lei, ignorando seu espírito. Usar delegação para ocultar o acoplamento forte não é o mesmo que desacoplar o código.

Ouvindo Deméter

Deméter está tentando lhe dizer algo e não é “use mais delegação”.

Cadeias de mensagens como `customer.bicycle.wheel.rotate` ocorrem quando suas ideias de design são indevidamente influenciadas por objetos que você já conhece. Sua familiaridade com as interfaces públicas de objetos conhecidos pode levá-lo a encadear longas cadeias de mensagens para chegar a um comportamento distante.

Alcançar objetos díspares para invocar um comportamento distante equivale a dizer: “há algum comportamento lá no fundo que preciso aqui mesmo e *sei como consegui-lo*”. O código sabe não apenas *o que* deseja (girar), mas *como* navegar por vários objetos intermediários para alcançar o comportamento desejado. Assim como Trip, anteriormente, sabia como o Mecânico deveria preparar uma bicicleta e, portanto, estava fortemente acoplado ao Mecânico, aqui o método `depart` sabe como navegar por uma série de objetos para fazer uma roda girar e, portanto, está fortemente acoplado ao seu objeto geral.

estrutura.

Este acoplamento causa todos os tipos de problemas. O mais óbvio é que aumenta o risco de Trip ser forçado a mudar devido a uma mudança não relacionada em algum lugar da cadeia de mensagens. No entanto, há outro problema aqui que é ainda mais sério.

Quando o método `depart` conhece esta cadeia de objetos, ele se vincula a uma implementação muito específica e não pode ser reutilizado em nenhum outro contexto. Os clientes devem ter sempre Bicicletas, que por sua vez devem ter Rodas que giram.

Considere como seria essa cadeia de mensagens se você tivesse começado decidindo o que o departamento deseja do cliente. Do ponto de vista baseado em mensagens, a resposta é óbvia:

cliente.passeio

O método ride do cliente oculta detalhes de implementação do Trip e reduz seu contexto e suas dependências, melhorando significativamente o design. Quando o FastFeet muda e começa a liderar caminhadas, é muito mais fácil generalizar de customer.ride para customer.depart ou customer.go do que desembaraçar os dez elementos dessa cadeia de mensagens de seu aplicativo.

Os destroços de trem das violações do Demeter são pistas de que existem objetos cujas interfaces públicas estão faltando. Ouvir Deméter significa prestar atenção ao seu ponto de vista. Se você mudar para uma perspectiva baseada em mensagens, as mensagens que você encontrar se tornarão interfaces públicas nos objetos que elas o levarão a descobrir. No entanto, se você estiver preso às amarras dos objetos de domínio existentes, acabará montando suas interfaces públicas existentes em longas cadeias de mensagens e, assim, perderá a oportunidade de encontrar e construir interfaces públicas flexíveis.

Resumo Os

aplicativos orientados a objetos são definidos pelas mensagens que passam entre os objetos. Essa passagem de mensagens ocorre em interfaces “públicas”; interfaces públicas bem definidas consistem em métodos estáveis que expõem as responsabilidades de suas classes subjacentes e fornecem o máximo benefício a um custo mínimo.

Focar nas mensagens revela objetos que, de outra forma, poderiam passar despercebidos. Quando as mensagens são *confiáveis* e perguntam o que o remetente deseja, em vez de dizer ao destinatário como se comportar, os objetos desenvolvem naturalmente interfaces públicas que são flexíveis e reutilizáveis de maneiras novas e inesperadas.

Esta página foi intencionalmente deixada em branco

CAPÍTULO 5

Reduzindo custos com Digitação de pato

O objetivo do design orientado a objetos é reduzir o custo da mudança. Agora que você sabe que as mensagens estão no centro do design da sua aplicação e que está comprometido com a construção de interfaces públicas rigorosamente definidas, você pode combinar essas duas ideias em uma poderosa técnica de design que reduz ainda mais seus custos.

Essa técnica é conhecida como *digitação de pato*. Os tipos Duck são interfaces públicas que não estão vinculadas a nenhuma classe específica. Essas interfaces entre classes adicionam enorme flexibilidade ao seu aplicativo, substituindo dependências dispendiosas de classe por dependências de mensagens mais tolerantes.

Objetos do tipo pato são camaleões definidos mais por seu comportamento do que por sua classe. É assim que a técnica recebe esse nome; se um objeto grasha como um pato e anda como um pato, então sua classe é imaterial, é um pato.

Este capítulo mostra como reconhecer e explorar tipos de patos para tornar sua aplicação mais flexível e mais fácil de alterar.

Compreendendo Duck Typing Linguagens de

programação usam o termo “tipo” para descrever a categoria do conteúdo de uma variável. As linguagens processuais fornecem um número pequeno e fixo de tipos, geralmente usados para descrever tipos de *dados*. Até mesmo a linguagem mais humilde define tipos para conter strings, números e arrays.

É o conhecimento da categoria do conteúdo de uma variável, ou do seu tipo, que permite a uma aplicação ter uma expectativa sobre como esse conteúdo se comportará.

Os aplicativos assumem razoavelmente que os números podem ser usados em expressões matemáticas, strings concatenadas e matrizes indexadas.

Em Ruby essas expectativas sobre o comportamento de um objeto vêm na forma de crenças sobre sua interface pública. Se um objeto conhece o tipo de outro, ele sabe a quais mensagens esse objeto pode responder.

Uma instância da classe Mechanic contém, obviamente, a interface pública completa do Mechanic. É evidente que qualquer objeto que se apegue a uma instância do Mecânico pode tratar a instância como se fosse um Mecânico; o objeto, por sua própria natureza, implementa a interface pública da classe Mechanic .

Entretanto, você não está limitado a esperar que um objeto responda a apenas *uma* interface. Um objeto Ruby é como um festeiro em um baile de máscaras que muda de máscara para se adequar ao tema. Pode expor um rosto diferente para cada espectador; ele pode implementar muitas interfaces diferentes.

Assim como a beleza está no mundo físico, na sua aplicação o tipo de objeto está nos olhos de quem vê. Os usuários de um objeto não precisam e não devem se preocupar com sua classe. A classe é apenas uma maneira de um objeto adquirir uma interface pública; a interface pública que um objeto obtém por meio de sua classe pode ser uma das várias que ele contém. Os aplicativos podem definir muitas interfaces públicas que não estão relacionadas a uma classe específica; essas interfaces atravessam a classe. Os usuários de qualquer objeto podem esperar alegremente que ele aja como qualquer uma ou todas as interfaces públicas que ele implementa. Não é o que um objeto é que importa, é o que ele faz.

Se cada objeto confia que todos os outros serão o que espera em um determinado momento, e qualquer objeto pode ser qualquer tipo de coisa, as possibilidades de design são infinitas. Estas possibilidades podem ser usadas para criar designs flexíveis que são maravilhas da criatividade estruturada ou, alternativamente, para construir designs aterrorizantes que são incompreensivelmente caóticos.

Usar essa flexibilidade com sabedoria requer que você reconheça esses tipos entre classes e construa suas interfaces públicas tão intencional e diligentemente quanto você fez com os tipos dentro da classe no Capítulo 4, Criando interfaces flexíveis. Os tipos entre classes, tipos duck, possuem interfaces públicas que representam um contrato que deve ser explícito e bem documentado.

A melhor maneira de explicar os tipos de patos é explorar as consequências de não usá-los. Esta seção contém um exemplo que passa por diversas refatorações, resolvendo um problema complicado de design ao encontrar e implementar um tipo pato.

Com vista para o pato No código

a seguir, o método prepare de Trip envia a mensagem prepare_bicycles para o objeto contido em seu parâmetro mecânico . Observe que a classe Mechanic não é referenciada; mesmo que o nome do parâmetro seja mecânico, o objeto que ele contém pode ser de qualquer classe.

```

1 aula Viagem 2
attr_reader :bicicletas, :clientes, :veículo
3
4     # este argumento 'mecânico' poderia ser de qualquer classe
5 def preparar(mecânico)
6     mecânico.prepare_bicicletas(bicicletas)
7 fim
8
9     #...
10 fim
11
12 # se acontecer de você passar uma instância desta classe, 13 # funciona
13
14 Mecânico classe 14
15 def prepare_bicycles(bicicletas)
16     bicicletas.each {|bicicleta| prepare_bicycle(bicicleta)}
17 fim
18
19 def prepare_bicycle(bicicleta) #...
20
21 fim
22 fim

```

A Figura 5.1 contém o diagrama de seqüência correspondente, onde um objeto externo inicia tudo enviando preparação para Trip, passando um argumento.

O método prepare não tem dependência explícita da classe Mechanic , mas depende do recebimento de um objeto que possa responder a prepare_bicycles. Essa dependência é tão fundamental que é fácil passar despercebida ou desconsiderada, mas mesmo assim ela existe.

O método de preparação da Trip acredita firmemente que seu argumento contém um preparador de bicicletas.

Complicando o problema

Você já deve ter notado que este exemplo é como o diagrama de sequência da Figura 4.6 do Capítulo 4. A próxima refatoração melhorou o design, empurrando o conhecimento

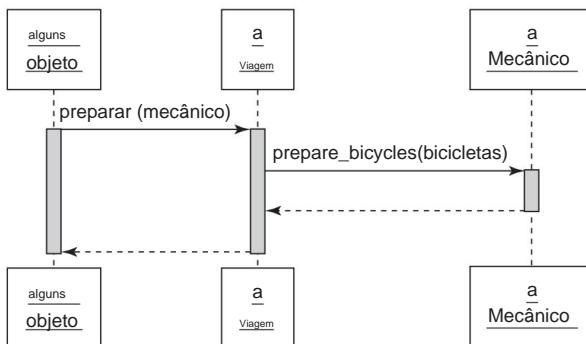


Figura 5.1 A viagem se prepara pedindo a um mecânico que prepare as bicicletas.

de como uma viagem é preparada para o Mecânico. O próximo exemplo aqui, infelizmente, não é melhoria em tudo.

Imagine que os requisitos mudam. Além de mecânico, preparação de viagem agora envolve um coordenador de viagem e um motorista. Seguindo o padrão estabelecido código, você cria novas classes TripCoordinator e Driver e dá a elas o comportamento pelo qual são responsáveis. Você também altera o método de preparação do Trip para invocar o comportamento correto de cada um de seus argumentos.

O código a seguir ilustra a mudança. O novo TripCoordinator e

As aulas de motorista são simples e inofensivas, mas o método de preparação do Trip agora é uma causa para alarme. Refere-se a três classes diferentes por nome e conhece métodos específicos implementados em cada uma. Os riscos aumentaram dramaticamente. O método de preparação da viagem pode ser forçado a mudar por causa de uma mudança em outro lugar e pode quebrar inesperadamente como o resultado de uma mudança distante e não relacionada.

```

1 # A preparação da viagem fica mais complicada
2
3 attr_reader :bicicletas, :clientes, :veículo
4
5 def preparar (preparadores)
6     preparadores.each {|preparador|
7         preparador de caso
8             quando mecânico
9                 preparer.prepare_bicycles(bicicletas)
10            quando TripCoordinator
11                preparer.buy_food(clientes)
12            quando motorista
  
```

```

13     preparer.gas_up(veículo)
14     preparer.fill_water_tank(veículo)
15     fim
16 }
17 fim
18 fim
19
20 # quando você apresenta TripCoordinator e Driver
Coordenador de viagem de 21 turmas
22 def comprar_comida(clientes)
23     #...
24 fim
25 fim
26
Motorista classe 27
28 def gas_up(veículo)
29     #...
30 fim
31
32 def fill_water_tank(veículo)
33     #...
34 fim
35 fim

```

Este código é a primeira etapa de um processo que o colocará em um beco sem saída. Código como este é escrito quando os programadores ficam cegos pelas classes existentes e negligenciar a percepção de que ignoraram mensagens importantes; este dependente carregado o código é uma consequência natural de uma perspectiva baseada em classes.

As raízes do problema são bastante inocentes. É fácil cair na armadilha de pensando no método de preparação original como esperando uma instância real de Mecânico. Seu cérebro técnico certamente reconhece que o argumento de prepare pode ser legalmente de qualquer classe, mas isso não salva você; no fundo do seu coração você pensa o argumento como sendo um Mecânico.

Como você sabe que o Mecânico entende o prepare_bicycle e está confiante de que está passando por um Mecânico, inicialmente está tudo bem. Essa perspectiva funciona bem até que algo mude e instâncias de classes diferentes do Mecânico comecem a aparecerem na lista de argumentos. Quando isso acontecer, prepare-se para lidar repentinamente com objetos que não entendem prepare_bicycle.

Se sua imaginação de design é limitada por classe e você se vê lidando inesperadamente com objetos que não entendem a mensagem que você está enviando, seu

A tendência é ir em busca de mensagens que esses novos objetos *entendam*. Como os novos argumentos são instâncias de TripCoordinator e Driver, você naturalmente examina as interfaces públicas dessas classes e encontra `buy_food`, `gas_up` e `fill_water_tank`. Este é o comportamento que o preparado deseja.

A maneira mais óbvia de invocar esse comportamento é enviar essas mesmas mensagens, mas agora você está preso. Cada um dos seus argumentos é de uma classe diferente e implementa métodos diferentes; você deve determinar a classe de cada argumento para saber qual mensagem enviar. Adicionar uma instrução `case` que ativa a classe resolve o problema de enviar a mensagem correta para o objeto correto, mas causa uma explosão de dependências.

Conte o número de novas dependências no método `prepare`. Depende de classes específicas, nenhuma outra servirá. Baseia-se nos nomes explícitos dessas classes. Ele conhece os nomes das mensagens que cada classe entende, juntamente com os argumentos que essas mensagens exigem. Todo esse conhecimento aumenta o risco; muitas mudanças distantes agora terão efeitos colaterais neste código.

Para piorar a situação, esse estilo de código se propaga. Quando outro novo preparador de viagem aparecer, você ou a próxima pessoa na linha de programação adicionará uma nova ramificação `when` à instrução `case`. Sua aplicação acumulará cada vez mais métodos como este, onde o método conhece muitos nomes de classes e envia uma mensagem específica com base na classe. O ponto final lógico deste estilo de programação é uma aplicação rígida e inflexível, onde eventualmente se torna mais fácil reescrever tudo do que alterar alguma coisa.

A Figura 5.2 mostra o novo diagrama de sequência. Cada diagrama de sequência até agora tem sido mais simples que seu código correspondente, mas esse novo diagrama parece assustadoramente complicado. Essa complexidade é um aviso. Os diagramas de sequência devem ser sempre mais simples que o código que representam; quando não estão, algo está errado com o design.

Encontrando o Pato A

chave para remover as dependências é reconhecer que, como o método `prepare` do `Trip` serve a um único propósito, seus argumentos chegam desejando colaborar para atingir um único objetivo. Cada argumento está aqui pela mesma razão e essa razão não está relacionada à classe subjacente do argumento.

Evite se desviar do seu conhecimento do que a classe de cada argumento já faz; em vez disso, pense sobre o que o preparo precisa. Considerado a partir de `prepare's`

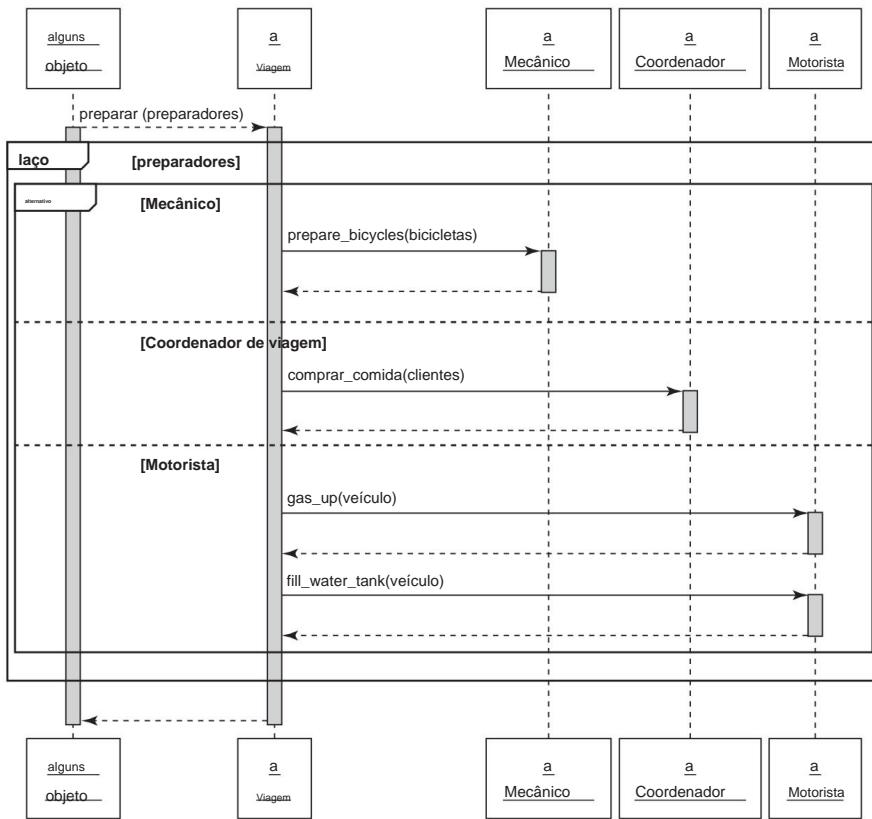


Figura 5.2 Trip conhece muitas classes e métodos concretos.

ponto de vista, o problema é simples. O método prepare quer preparar a viagem. Seus argumentos chegam prontos para colaborar na preparação da viagem. O design seria mais simples se a preparação apenas confiasse neles para fazê-lo.

A Figura 5.3 ilustra esta ideia. Aqui, o método `prepare` não tem uma expectativa pré-determinada sobre a classe de seus argumentos; em vez disso, espera que cada um seja um “Preparador”.

Essa expectativa vira o jogo perfeitamente. Você se libertou das classes existentes e inventou um tipo de pato. O próximo passo é perguntar qual mensagem o método de preparação pode enviar de forma proveitosa a cada Preparador. Deste ponto de vista, a resposta é óbvia: prepare_trip.

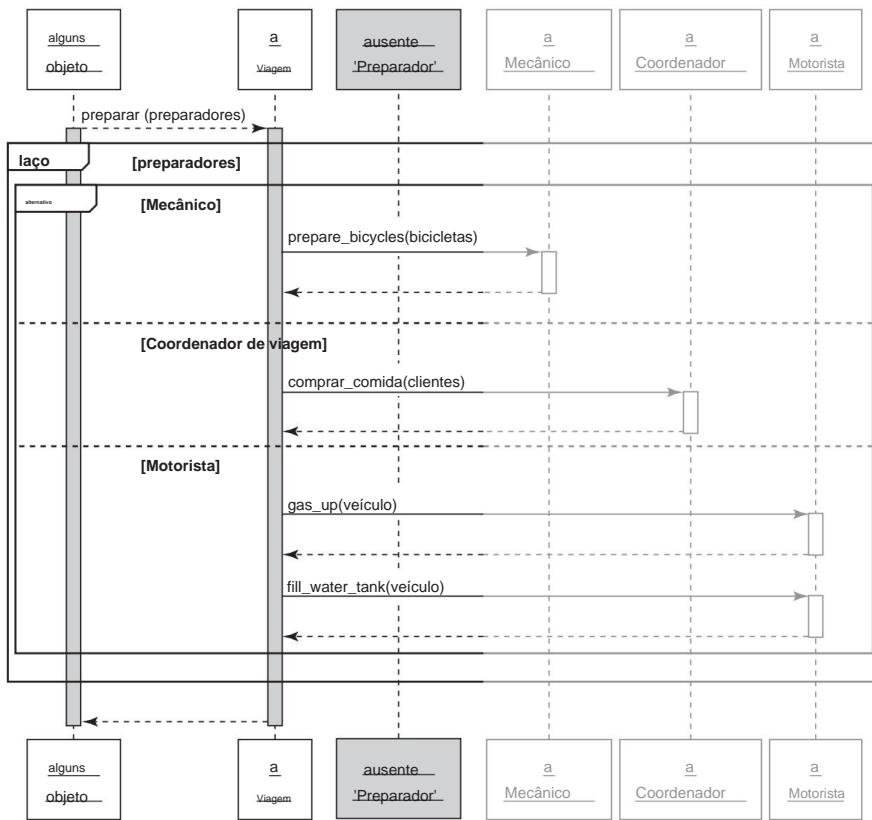


Figura 5.3 O Trip precisa que cada argumento atue como um preparador.

A Figura 5.4 apresenta a nova mensagem. O método `prepare` do `Trip` agora espera que seus argumentos sejam `Preparadores` que possam responder a `prepare_trip`.

Que tipo de coisa é o `Preparador`? Neste ponto não tem existência concreta; é uma abstração, um acordo sobre a interface pública de uma ideia. É uma invenção do design.

Objetos que implementam `prepare_trip` são `Preparadores` e, inversamente, objetos que interagem com `Preparadores` só precisam confiar neles para implementar a interface do `Preparador`. Depois de ver essa abstração subjacente, será fácil corrigir o código. `Mecânico`, `Coordenador` de `Viagem` e `Motorista` devem se comportar como `Preparadores`; eles deveriam implementar `prepare_trip`.

Aqui está o código para o novo design. O método `prepare` agora espera seu argumento ser um `Preparador` e a classe de cada argumento implementa a nova interface.

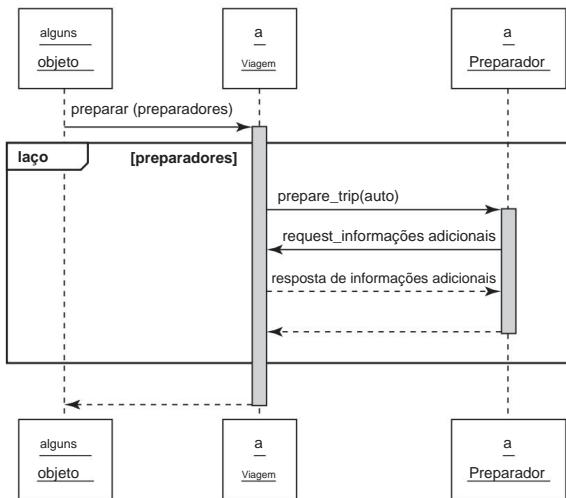


Figura 5.4 Trip colabora com o pato preparador.

```

1 # A preparação da viagem fica mais fácil 2 aulas Viagem 3
4
5 def prepare(preparadores) preparers.each {| 
6     preparer| 
7         preparer.prepare_trip(self)}
8 fim
9 fim
10
11 # quando todo preparador é um Pato 12 # que responde a
12 'prepare_trip' Mecânico classe 13
13
14 def prepare_trip(viagem) trip.bicycles.each {| 
15     bicicleta| prepare_bicycle(bicicleta)}
16
17 fim
18
19     #...
20 fim
21
22 classe TripCoordinator 23 def
23     prepare_trip(viagem)
  
```

```

24     comprar_comida(viagem.clientes)
25 fim
26
27 #...
28 fim
29
Motorista classe 30
31 def prepare_trip(viagem) veículo =
32     viagem.veículo gas_up(veículo)
33     fill_water_tank(veículo)
34
35 fim
36 #...
37 fim

```

O método prepare agora pode aceitar novos Preparadores sem ser forçado a alterar, e é fácil criar Preparadores adicionais se necessário.

Consequências da Digitação de Pato Esta nova

implementação tem uma simetria agradável que sugere uma correção no design, mas as consequências da introdução de um tipo de pato são mais profundas.

No exemplo inicial, o método prepare depende de uma classe concreta. Neste exemplo mais recente, o preparo depende do tipo de pato. O caminho entre esses exemplos passa por um emaranhado de códigos complicados e carregados de dependências.

A concretude do primeiro exemplo torna-o simples de compreender, mas perigoso de ampliar. A alternativa final, tipo pato, é mais abstrata; impõe exigências um pouco maiores à sua compreensão, mas em troca oferece facilidade de extensão. Agora que você descobriu o problema, você pode extrair um novo comportamento do seu aplicativo sem alterar nenhum código existente; você simplesmente transforma outro objeto em um Preparer e o passa para o método prepare do Trip .

Esta tensão entre os custos de concretagem e os custos de captação é divertida.

fundamental para o design orientado a objetos. O código concreto é fácil de entender, mas caro para estender. O código abstrato pode inicialmente parecer mais obscuro, mas, uma vez compreendido, é muito mais fácil de mudar. O uso de um tipo pato move seu código ao longo da escala, de mais concreto para mais abstrato, tornando o código mais fácil de estender, mas lançando um véu sobre a classe subjacente do pato.

A capacidade de tolerar a ambigüidade sobre a classe de um objeto é a marca registrada de um designer confiante. Depois que você começar a tratar seus objetos como se fossem definidos por

pelo seu comportamento e não pela sua classe, você entra em um novo reino de design flexível e expressivo.

Polimorfismo O

termo *polimorfismo* é comumente usado em programação orientada a objetos, mas seu uso na fala cotidiana é raro o suficiente para justificar uma definição.

O polimorfismo expressa um conceito muito específico e pode ser utilizado, dependendo das suas inclinações, tanto para comunicar como para intimidar.

De qualquer forma, é importante ter uma compreensão clara do seu significado.

Primeiro, uma definição geral: *Morph* é a palavra grega para forma, *morfismo* é o estado de ter uma forma e *polimorfismo* é o estado de ter muitas formas. Os biólogos usam essa palavra. Os famosos tentilhões de Darwin são polimórficos; uma única espécie tem muitas formas.

Polimorfismo em POO refere-se à capacidade de muitos objetos diferentes responderem à mesma mensagem. Os remetentes da mensagem não precisam se preocupar com a classe do destinatário; os receptores fornecem sua própria versão específica do comportamento.

Uma única mensagem, portanto, tem muitas (poli) formas (morfos).

Existem várias maneiras de obter polimorfismo; digitação com pato, como você certamente adivinhou, é uma delas. Herança e compartilhamento de comportamento (via módulos Ruby) são outros, mas esses são tópicos para os próximos capítulos.

Os métodos polimórficos respeitam uma barganha implícita; eles concordam em ser intercambiáveis *do ponto de vista do remetente*. Qualquer objeto que implemente um método polimórfico pode ser substituído por qualquer outro; o remetente da mensagem não precisa saber ou se preocupar com essa substituição.

Essa substituibilidade não acontece por mágica. Quando você usa polimorfismo, cabe a você garantir que todos os seus objetos estejam bem comportados. Essa ideia é abordada no Capítulo 7, Compartilhando comportamento de função com módulos.

Escrevendo código que depende de patos

Usar a digitação duck depende de sua capacidade de reconhecer os locais onde seu aplicativo se beneficiaria de interfaces entre classes. É relativamente fácil implementar um tipo de pato; seu desafio de design é perceber que você precisa de um e abstrair sua interface.

Esta seção contém padrões que revelam caminhos que você pode seguir para descobrir patos.

Reconhecendo patos escondidos

Muitas vezes já existem tipos de patos não reconhecidos, escondidos no código existente. Vários padrões de codificação comuns indicam a presença de um pato escondido. Você pode substituir o seguinte por patos:

- Declarações de caso que ativam a aula
- tipo de? e é_a?
- responde a?

Declarações de caso que ativam a aula

O padrão mais comum e óbvio que indica um pato não descoberto é o exemplo você já viu; uma instrução case que ativa os nomes de classe do domínio objetos da sua aplicação. O seguinte método de preparação (igual ao acima) deve prender sua atenção como se estivesse tocando trombetas.

```

1 aula de viagem
2 attr_reader :bicicletas, :clientes, :veículo
3
4 def preparar (preparadores)
5     preparadores.each {|preparador|
6         preparador de caso
7             quando mecânico
8                 preparer.prepare_bicycles(bicicletas)
9             quando TripCoordinator
10                preparer.buy_food(clientes)
11             quando motorista
12                 preparer.gas_up(veículo)
13                 preparer.fill_water_tank(veículo)
14         fim
15     }
16 fim
17 fim

```

Ao ver esse padrão, você sabe que todos os preparadores devem compartilhar algo em comum; eles chegam aqui por causa daquela coisa comum. Examine o código e pergunte-se: "O que é que prepara os desejos de cada um de seus argumentos?"

A resposta a essa pergunta sugere a mensagem que você deve enviar; esta mensagem começa a definir o tipo de pato subjacente.

Escrevendo código que depende de patos

Aqui o método `prepare` quer seus argumentos para preparar a viagem. Por isso, `prepare_trip` torna-se um método na interface pública do novo pato `Preparer`.

tipo de? e é_a?

Existem várias maneiras de verificar a classe de um objeto. A declaração de caso acima é uma das elas. O tipo de? e é_a? mensagens (são sinônimos) também verificam a classe.

Reescrever o exemplo anterior da seguinte maneira não melhora o código.

```

1 se preparer.kind_of?(Mecânico)
2     preparer.prepare_bicycles(bicicleta)
3 elsif preparer.kind_of?(TripCoordinator)
4     preparer.buy_food(clientes)
5 elsif preparer.kind_of?(Motorista)
6     preparer.gas_up(veículo)
7     preparer.fill_water_tank(veículo)
8 fim

```

Usando `kind_of?` não é diferente de usar uma instrução `case` que ativa a classe; eles são a mesma coisa, causam exatamente os mesmos problemas e deveriam ser corrigido usando as mesmas técnicas.

responde a?

Programadores que entendem que não devem depender de nomes de classes, mas que ainda não deram o salto para os tipos pato ficam tentados a substituir `kind_of?` com `responde a?`. Por exemplo:

```

1 se preparer.responds_to?(:prepare_bicycles)
2     preparer.prepare_bicycles(bicicleta) 3 elsif
3     preparer.responds_to?(:buy_food)
4     preparer.buy_food(clientes)
5 elsif preparador.responds_to?(:gas_up)
6     preparer.gas_up(veículo)
7     preparer.fill_water_tank(veículo)
8 fim

```

Embora isso diminua um pouco o número de dependências, esse código ainda tem muitas. Os nomes das classes desapareceram, mas o código ainda está muito vinculado à classe. Que objeto será conhecido `prepare_bicycles` além do Mecânico? Não se deixe enganar pela remoção de referências de classe explícitas. Este exemplo ainda espera classes muito específicas.

Mesmo se você estiver em uma situação em que mais de uma classe implemente `prepare_bicycles` ou `buy_food`, esse padrão de código ainda conterá dependências desnecessárias; ele controla em vez de confiar em outros objetos.

Depositando confiança em seus patos

O uso de `kind_of?`, `is_a?`, `responds_to?` e declarações `case` que ativam suas classes indicam a presença de um pato não identificado. Em cada caso, o código está efetivamente dizendo “Eu sei quem você é e por isso sei o que você faz”.

Esse conhecimento expõe a falta de confiança na colaboração de objetos e atua como uma pedra de moinho no pescoço do seu objeto. Ele introduz dependências que dificultam a alteração do código.

Assim como nas violações do Demeter, esse estilo de código é uma indicação de que está faltando um objeto, cuja interface pública você ainda não descobriu. O fato de o objeto ausente ser do tipo pato em vez de uma classe concreta não importa de forma alguma; é a interface que importa, não a classe do objeto que a implementa.

Aplicações flexíveis são construídas em objetos que operam com base na confiança; é seu trabalho tornar seus objetos confiáveis. Ao ver esses padrões de código, concentre-se nas expectativas do código infrator e use essas expectativas para encontrar o tipo de pato. Depois de ter um tipo de pato em mente, defina sua interface, implemente-a quando necessário e confie que esses implementadores se comportarão corretamente.

Documentando tipos de pato O tipo

mais simples de tipo de pato é aquele que existe meramente como um acordo sobre sua interface pública. O código de exemplo deste capítulo implementa esse tipo de pato, onde diversas classes diferentes implementam `prepare_trip` e podem, portanto, ser tratadas como Preparadores.

O tipo Preparer duck e sua interface pública são uma parte concreta do design, mas uma parte virtual do código. Os preparadores são abstratos; isso lhes dá força como ferramenta de design, mas essa mesma abstração torna o tipo pato menos óbvio no código.

Ao criar tipos de pato você deve documentar e testar suas interfaces públicas. Felizmente, bons testes são a melhor documentação, então você já está na metade do caminho; você só precisa escrever os testes.

Consulte o Capítulo 9, Projetando testes econômicos, para obter mais informações sobre como testar tipos de patos.

Compartilhando código entre patos

Neste capítulo, os patos preparadores fornecem versões específicas da classe do comportamento necessário pela sua interface. Mecânico, Motorista e Coordenador de Viagem , cada um implementando método preparar _viagem. Essa assinatura de método é a única coisa que eles têm em comum. Eles compartilhe apenas a interface, não a implementação.

Depois que você começar a usar tipos de pato, entretanto, você descobrirá que as classes que implementam muitas vezes eles precisam compartilhar algum comportamento em comum. Escrever patos que compartilham código é um dos tópicos abordados no Capítulo 7.

Escolhendo seus patos com sabedoria

Todos os exemplos até agora declaram inequivocamente que você não deve usar kind_of? ou responde a? para decidir qual mensagem enviar a um objeto, mas você não precisa olhar Longe de encontrar resmas de código bem recebido que fazem exatamente isso.

O código a seguir é um exemplo da estrutura Ruby on Rails (active_record/relações/finder_methods.rb). Este exemplo usa patentemente a classe para decidir como lidar com a sua contribuição, uma técnica que está em oposição direta às diretrizes indicadas acima. O primeiro método abaixo decide claramente como se comportar com base na classe de seus argumentos argumento.

Se enviar uma mensagem com base na classe do objeto receptor é a sentença de morte para sua aplicação, por que esse código é aceitável?

```
1 # Um wrapper conveniente para <tt>find(:first, *args)</tt>.
2 # Você pode passar para este método 3 # os mesmos argumentos que você
3 # pode passar para <tt>find(:first)</tt>.
4 def primeiro(*args)
5   se args.qualquer?
6     if args.first.kind_of?(Inteiro) || (carregado? && !
7       args.first.kind_of?(Hash))
8       to_a.first(*args)
9     outro
10      apply_finder_options(args.first).primeiro
11    fim
12  mais
13    encontrar_primeiro
14  fim
15 fim
```

A principal diferença entre este exemplo e os anteriores é a estabilidade das classes que estão sendo verificadas. Quando depende primeiro de Integer e Hash, depende das classes principais do Ruby que são muito mais estáveis do que são. A probabilidade de Inteiro ou Hash mudar de forma a forçar a mudança primeiro é extremamente pequena. Essa dependência é segura. Provavelmente existe um tipo de pato escondido em algum lugar deste código, mas provavelmente não reduzirá os custos gerais do aplicativo para localizá-lo e implementá-lo.

Neste exemplo você pode ver que a decisão de criar um novo tipo de pato depende de julgamento. O objetivo do design é reduzir custos; leve esta medida para todas as situações. Se a criação de um tipo pato reduzir dependências instáveis, faça-o.

Use seu melhor julgamento.

O pato subjacente do exemplo acima abrange Inteiro e Hash e, portanto, sua implementação exigiria alterações nas classes base Ruby. Alterar classes base é conhecido como *monkey patching* e é um recurso encantador do Ruby, mas pode ser perigoso em mãos não treinadas.

Implementar tipos de patos em suas próprias classes é uma coisa, mudar as classes base do Ruby para introduzir novos tipos de patos é outra bem diferente. As compensações são diferentes; os riscos são maiores. Nenhuma dessas considerações deve impedir que você faça patches de Ruby quando necessário; entretanto, você deve ser capaz de defender eloquientemente essa decisão de design. O padrão de prova é alto.

Vencendo o medo da digitação com pato

Este capítulo até agora evitou delicadamente o campo de batalha da digitação dinâmica versus estática, mas o problema não pode mais ser evitado. Se você tem experiência em programação de digitação estática e acha alarmante a ideia de digitação com pato, esta seção é para você.

Se você não está familiarizado com o argumento, está feliz em usar Ruby e foi convencido pelo discurso anterior sobre digitação com pato, você pode folhear esta seção sem medo de perder novos conceitos importantes. Você pode, no entanto, achar o que se segue útil se precisar se defender de argumentos feitos por seus amigos com digitação mais estática.

Subvertendo tipos de pato com digitação estática No início deste capítulo, *tipo* foi definido como a categoria do conteúdo de uma variável.

As linguagens de programação são digitadas *estaticamente* ou *dinamicamente*. A maioria (embora não todas) das linguagens de tipo estaticamente exige que você declare explicitamente o tipo de cada variável e cada parâmetro do método. Linguagens digitadas dinamicamente omitem esse requisito; eles permitem que você coloque qualquer valor em qualquer variável e passe qualquer argumento para qualquer método, sem qualquer declaração adicional. Ruby, obviamente, é digitado dinamicamente.

Depender da digitação dinâmica deixa algumas pessoas desconfortáveis. Para alguns, esse desconforto é causado pela falta de experiência; para outros, pela crença de que a digitação estática é mais confiável.

O problema da falta de experiência se cura sozinho, mas a crença de que a digitação estática é fundamentalmente preferível muitas vezes persiste porque se auto-reforça. Os programadores que temem a digitação dinâmica tendem a verificar as classes de objetos em seu código; essas mesmas verificações subvertem o poder da digitação dinâmica, impossibilitando o uso de tipos pato.

Métodos que não podem se comportar corretamente a menos que conheçam as classes de seus argumentos falharão (com erros de tipo) quando novas classes aparecerem. Os programadores que acreditam na digitação estática consideram essas falhas como prova de que é necessária mais verificação de tipo. Quando mais verificações são adicionadas, o código se torna menos flexível e ainda mais dependente da classe. As novas dependências causam falhas de tipo adicionais, e o programador responde a essas falhas adicionando ainda mais verificações de tipo. Qualquer pessoa presa nesse loop naturalmente terá dificuldade em acreditar que a solução para seu problema de tipo é remover completamente a verificação de tipo.

A digitação de pato fornece uma saída para essa armadilha. Ele remove as dependências da classe e, assim, evita as falhas de tipo subsequentes. Ele revela abstrações estáveis das quais seu código pode depender com segurança.

Digitação estática versus digitação dinâmica

Esta seção compara digitação dinâmica e estática, na esperança de acalmar quaisquer medos que o impeçam de se comprometer totalmente com tipos dinâmicos.

A digitação estática e dinâmica fazem promessas e cada uma tem custos e benefícios.

Os aficionados da digitação estática citam as seguintes qualidades:

- O compilador descobre erros de tipo em tempo de compilação.
- Informações de tipo visíveis servem como documentação.
- O código compilado é otimizado para execução rápida.

Essas qualidades representam pontos fortes em uma linguagem de programação somente se você aceitar este conjunto de suposições correspondentes:

- Erros de tipo de tempo de execução ocorrerão, a menos que o compilador execute verificações de tipo.
- Os programadores não compreenderão o código de outra forma; eles não podem inferir a propriedade de um objeto digitado a partir de seu contexto.
- O aplicativo será executado muito lentamente sem essas otimizações.

Os proponentes da digitação dinâmica listam estas qualidades:

- O código é interpretado e pode ser carregado dinamicamente; não há ciclo de compilação/make.
- O código-fonte não inclui informações explícitas de tipo.
- A metaprogramação é mais fácil.

Essas qualidades são pontos fortes se você aceitar este conjunto de suposições:

- O desenvolvimento geral de aplicativos é mais rápido sem um ciclo de compilação/criação.
- Os programadores acham o código mais fácil de entender quando ele não contém declarações de tipo; eles podem inferir o tipo de um objeto a partir de seu contexto.
- A metaprogramação é um recurso desejável da linguagem.

Adotando a Digitação Dinâmica Algumas dessas

qualidades e suposições são baseadas em fatos empíricos e são fáceis de avaliar. Não há dúvida de que, para certas aplicações, um código de tipo estaticamente bem otimizado irá superar uma implementação de tipo dinâmico. Quando um aplicativo digitado dinamicamente não pode ser ajustado para ser executado com rapidez suficiente, a digitação estática é a alternativa. Se você precisa, você deve.

Argumentos sobre o valor das declarações de tipo como documentação são mais subjetivos. Aqueles com experiência em digitação dinâmica consideram as declarações de tipo uma distração.

Aqueles acostumados com a digitação estática podem ficar desorientados pela falta de informações de tipo. Se você vem de uma linguagem de tipo estaticamente, como Java ou C++, e se sente desamparado pela falta de declarações de tipo explícitas em Ruby, aguente firme. Há muitas evidências anedóticas que sugerem que, uma vez acostumado com isso, você achará essa sintaxe menos detalhada mais fácil de ler, escrever e entender.

Metaprogramação (isto é, escrever código que escreve código) é um tópico sobre o qual os programadores tendem a ter fortes sentimentos e o lado do argumento que eles apoiam está relacionado à sua experiência passada. Se você resolveu um grande problema com uma metaprogramação simples e elegante, você se torna um defensor da vida. Por outro lado, se você enfrentou a difícil tarefa de depurar uma metaprogramação excessivamente inteligente, completamente obscura e possivelmente desnecessária, você pode percebê-la como uma ferramenta para os programadores infligirem dor uns aos outros e desejarem baní-la para sempre. .

A metaprogramação é um bisturi; embora perigoso nas mãos erradas, é uma ferramenta que nenhum bom programador deveria dispensar. Confere grande poder e requer



Figura 5.5 Instrumentos cortantes: úteis, mas não para todos.

grande responsabilidade. O facto de não se poder confiar em algumas pessoas com facas não significa que os instrumentos cortantes devam ser retirados das mãos de todos. A metaprogramação, usada com sabedoria, tem grande valor; a facilidade de metaprogramação é um forte argumento a favor da digitação dinâmica.

As duas qualidades restantes são a verificação de tipo em tempo de compilação da digitação estática e a falta de um ciclo de compilação/criação da digitação dinâmica. Os defensores da digitação estática afirmam que prevenir erros de tipo inesperados em tempo de execução é tão necessário e valioso que seu benefício supera a maior eficiência de programação obtida com a remoção do compilador.

Este argumento baseia-se na premissa da tipagem estática de que:

- O compilador realmente *pode* evitar erros de tipo acidentais.
- Sem a ajuda do compilador, esses erros de tipo *ocorrerão*.

Se você passou anos programando em uma linguagem de tipo estaticamente, você pode aceitar essas afirmações como verdade. No entanto, a digitação dinâmica está aqui para abalar os alicerces de sua crença. Para esses argumentos, a digitação dinâmica diz “Não pode” e “Eles não vão”.

O compilador *não pode* salvá-lo de erros de tipo acidentais. Qualquer linguagem que permita converter uma variável para um novo tipo é vulnerável. Assim que você começar a lançar, todas as apostas serão canceladas; o compilador pede licença e você depende de sua própria inteligência para evitar erros de tipo. O código é tão bom quanto seus testes; falhas de tempo de execução ainda podem ocorrer. A noção de que a digitação estática oferece segurança, por mais reconfortante que seja, é uma ilusão.

Além disso, na verdade não importa se o compilador pode salvá-lo ou não; você não precisa ser salvo. No mundo real, erros de tipo de tempo de execução evitáveis pelo compilador *quase nunca* ocorrem. Isso simplesmente não acontece.

Isso não significa que você nunca encontrará um erro de tipo de tempo de execução. Poucos programadores sobrevivem sem enviar uma mensagem para uma variável não inicializada ou assumir que um array tem elementos quando na verdade está vazio. No entanto, descobrindo em

tempo de execução que nil não entende a mensagem que recebeu não é algo que compilador poderia ter evitado. Esses erros são igualmente prováveis em ambos os sistemas de tipo.

A digitação dinâmica permite que você troque a verificação de tipo em tempo de compilação, uma restrição séria que tem alto custo e oferece pouco valor, pelos enormes ganhos de eficiência proporcionados removendo o ciclo de compilação/make. Este comércio é uma pechincha. Pegue.

A digitação Duck é baseada na digitação dinâmica; para usar a digitação de pato você deve *abraçar* esse dinamismo.

Resumo

As mensagens estão no centro das aplicações orientadas a objetos e passam entre objetos ao longo de interfaces públicas. A digitação Duck separa essas interfaces públicas de classes específicas, criando tipos virtuais que são definidos pelo que fazem e não por quem são.

A digitação Duck revela abstrações subjacentes que, de outra forma, poderiam ser invisíveis. Depender dessas abstrações reduz o risco e aumenta a flexibilidade, tornando seu aplicação mais barata de manter e mais fácil de alterar.

CAPÍTULO 6

Adquirindo Comportamento Através da herança

Aplicativos bem projetados são construídos com código reutilizável. Objetos independentes pequenos e confiáveis, com contexto mínimo, interfaces claras e dependências injetadas são inherentemente reutilizáveis. Este livro concentrou-se, até agora, na criação de objetos exatamente com essas qualidades.

A maioria das linguagens orientadas a objetos, entretanto, possui outra técnica de compartilhamento de código, incorporada à própria sintaxe da linguagem: *herança*. Este capítulo oferece um exemplo detalhado de como escrever código que utiliza herança adequadamente. Seu objetivo é ensiná-lo a construir uma hierarquia de herança tecnicamente sólida; seu objetivo é prepará-lo para decidir se deve fazê-lo.

Depois de entender como usar a herança clássica, os conceitos serão facilmente transferidos para outros mecanismos de herança. A herança é, portanto, um tema para dois capítulos. Este capítulo contém um tutorial que ilustra como escrever código herdável. O Capítulo 7, Compartilhando comportamento de função com módulos, expande essas técnicas para o problema de compartilhamento de código por meio de módulos Ruby.

Compreendendo a herança clássica

A ideia de herança pode parecer complicada, mas como acontece com toda complexidade, existe uma abstração simplificadora. A herança é, em sua essência, um mecanismo para *delegação automática de mensagens*. Define um caminho de encaminhamento para mensagens não compreendidas. Ele cria

relacionamentos tais que, se um objeto não puder responder a uma mensagem recebida, ele delega essa mensagem a outro. Você não precisa escrever código para delegar explicitamente a mensagem; em vez disso, você define um relacionamento de herança entre dois objetos e o encaminhamento acontece automaticamente.

Na herança clássica esses relacionamentos são definidos pela criação de subclasses.

As mensagens são encaminhadas da subclasse para a superclasse; o código compartilhado é definido na hierarquia de classes.

O termo *clássico* é uma brincadeira com a palavra *classe*, não uma referência a uma técnica arcaica, e serve para distinguir esse mecanismo de superclasse/subclasse de outras técnicas de herança. JavaScript, por exemplo, possui herança protótipica e Ruby possui *módulos* (mais sobre módulos no próximo capítulo), sendo que ambos também fornecem uma maneira de compartilhar código por meio de delegação automática de mensagens.

Os usos e maus usos da herança são melhor compreendidos por meio de exemplos, e o exemplo deste capítulo fornece uma base completa nas técnicas da herança clássica. O exemplo começa com uma única classe e passa por diversas refatorações para chegar a um conjunto satisfatório de subclasses. Cada etapa é pequena e de fácil compreensão, mas é necessário um capítulo inteiro de código para ilustrar todas as ideias.

Reconhecendo onde usar a herança

O primeiro desafio é reconhecer onde a herança seria útil. Esta seção ilustra como saber quando você tem o problema que a herança resolve.

Suponha que FastFeet conduza viagens de bicicleta de estrada. As bicicletas de estrada são leves, com guiador curvo (barra suspensa), bicicletas estreitas e cansadas, destinadas a estradas pavimentadas. A Figura 6.1 mostra uma bicicleta de estrada.

Os mecânicos são responsáveis por manter as bicicletas funcionando (não importa quantos abusos os clientes façam sobre elas) e levam uma variedade de peças sobressalentes em cada viagem. As peças sobressalentes de que necessitam dependem das bicicletas que utilizam.

Começando com uma classe Concrete a aplicação

do FastFeet já possui uma classe `Bicycle`, mostrada abaixo. Cada bicicleta de estrada que viaja é representada por uma instância desta classe.

As bicicletas têm tamanho total, cor da fita do guidão, tamanho do pneu e tipo de corrente.

Pneus e correntes são peças integrantes e por isso sempre devem ser levadas peças sobressalentes. A fita do guiador pode parecer menos necessária, mas na vida real é igualmente necessária. Nenhum ciclista que se preze toleraria fita adesiva suja ou rasgada; os mecânicos devem levar fita sobressalente na cor correta e correspondente.



Figura 6.1 Uma bicicleta de estrada leve, com barra rebatível e cansada.

Bicicleta de 1 classe

```
2 attr_reader :tamanho, :tape_color
3
4 def inicializar (args)
5   @tamanho = args[:tamanho]
6   @tape_color = args[:tape_color]
7 fim
8
9 # cada bicicleta tem os mesmos padrões para # tamanho de
10 pneu e corrente
11 peças sobressalentes def
12   {cadeia:           '10 velocidades',
13    tamanho_do_pneu: '23',
14    cor_da_fita:    cor_da_fita}
15 fim
16
17 # Muitos outros métodos...
18 fim
19
20 bicicleta = bicicleta.nova(
21   tamanho:          'M',
22   tape_color:      'vermelho' )
```

```

23
24 bike.size           # -> 'M'
25 peças sobressalentes para bicicletas
26 # -> {:tire_size => "23", => "10"
27#   :corrente      velocidades", :tape_color
28#   => "vermelho"}

```

As instâncias de bicicleta podem responder às mensagens sobressalentes, tamanho e tape_color e um mecânico pode descobrir quais peças de reposição levar pedindo a cada bicicleta seu peças sobressalentes. Apesar do método de reposição cometer o pecado de incorporar strings padrão diretamente dentro de si mesmo, o código acima é bastante razoável. Este modelo de uma bicicleta obviamente está faltando alguns parafusos e não é algo que você possa realmente *andar*, mas servirá para o exemplo deste capítulo.

Esta classe funciona bem até que algo mude. Imagine que FastFeet começa para liderar viagens de mountain bike.

As bicicletas de montanha e as bicicletas de estrada são muito parecidas, mas existem diferenças claras entre eles. As mountain bikes devem ser pedaladas em caminhos de terra em vez de pavimentadas estradas. Eles têm quadros robustos, pneus grossos, guidão de barra reta (com punho de borracha punhos em vez de fita) e suspensão. A bicicleta da Figura 6.2 tem suspensão dianteira apenas, mas algumas bicicletas de montanha também têm suspensão traseira ou “total”.



Figura 6.2 Uma mountain bike robusta, com barra reta, suspensão dianteira e pneus grossos.

Sua tarefa de design é adicionar suporte para mountain bikes ao aplicativo FastFeet.

Muito do comportamento que você precisa já existe; mountain bikes são definitivamente bicicletas. Eles têm um tamanho geral de bicicleta e um tamanho de corrente e pneu. As únicas diferenças entre as bicicletas de estrada e de montanha são que as bicicletas de estrada precisam de fita no guiador e as bicicletas de montanha têm suspensão.

Incorporando Vários Tipos Quando uma classe

concreta pré-existente contém a maior parte do comportamento que você precisa, é tentador resolver esse problema adicionando código a essa classe. O próximo exemplo faz exatamente isso, altera a classe Bicycle existente para que as peças sobressalentes funcionem tanto para bicicletas de estrada quanto de montanha.

Como você pode ver abaixo, três novas variáveis foram adicionadas, juntamente com seus acessadores correspondentes. As novas variáveis front_shock e rear_shock contêm peças específicas para mountain bike. A nova variável de estilo determina quais peças aparecem na lista de peças sobressalentes. Cada uma dessas novas variáveis é tratada adequadamente pelo método de inicialização .

O código para adicionar essas três variáveis é simples, até mesmo mundano; a mudança para peças sobressalentes se mostra mais interessante. O método spares agora contém uma instrução if que verifica o conteúdo da variável style. Essa variável de estilo atua para dividir as instâncias de Bicycle em duas categorias diferentes — aquelas cujo estilo é :road e aquelas cujo estilo é qualquer outra coisa.

Se algum alarme disparar enquanto você analisa este código, fique tranquilo, pois eles serão silenciados em breve. Este exemplo é simplesmente um desvio que ilustra um *antipadrão*, ou seja, um padrão comum que parece ser benéfico, mas na verdade é prejudicial, e para o qual existe uma alternativa bem conhecida.

Observação

Caso você esteja confuso com os tamanhos dos pneus abaixo, saiba que os tamanhos dos pneus de bicicleta são, por tradição, inconsistentes. As bicicletas de estrada são originárias da Europa e utilizam dimensionamento métrico; um pneu de 23 milímetros tem pouco menos de uma polegada de largura. As mountain bikes são originárias dos Estados Unidos e fornecem tamanhos de pneus em polegadas. No exemplo abaixo, o pneu de mountain bike de 2,1 polegadas tem duas vezes mais largura que o pneu de bicicleta de estrada de 23 mm.

```

Bicicleta de 1 classe

2 attr_reader :estilo, :tamanho, :tape_color,
3                               :choque_frontal, :choque_traseiro
4

5 def inicializar (args)
6     @estilo          = args[:estilo]
7     @tamanho         = args[:tamanho]
8     @tape_color      = args[:tape_color]
9     @front_shock    = args[:front_shock]
10    @rear_shock     = args[:rear_shock]
11  fim
12
13  # verificar o "estilo" começa a descer uma ladeira escorregadia
14 peças sobressalentes def
15  if estilo == :estrada
16      { cadeia:           '10 velocidades',
17        tamanho_do_pneu: # milímetros
18        cor_da_fita: cor_da_fita }
19  outro
20      { cadeia:           '10 velocidades',
21        tamanho_do_pneu: # polegadas
22        choque_traseiro: choque_traseiro }
23  fim
24 fim
25 fim
26
27 bicicleta = Bicicleta.new(
28     estilo:            :montanha,
29     tamanho:           'S',
30     front_shock:      'Manitou',
31     amortecedor_traseiro: 'Raposa')
32
33 peças sobressalentes para bicicletas
34 # -> {:tamanho_do_pneu => "2.1", 35 # :cadeia 36 #
35                               => "10 velocidades",
36                               :rear_shock => 'Fox'}

```

Este código toma decisões sobre peças de reposição com base no valor mantido no estilo; estruturar o código dessa forma tem muitas consequências negativas. Se você adicionar um novo estilo você deve alterar a instrução if . Se você escrever um código descuidado onde está a última opção o padrão (assim como o código acima) um estilo inesperado fará algo , mas talvez não seja o que você espera. Além disso, o método de peças de reposição começou contendo

strings padrão incorporadas, algumas dessas strings agora são duplicadas em cada lado da declaração if.

A bicicleta possui uma interface pública implícita que inclui peças sobressalentes, tamanho e todas as peças individuais. O método do tamanho ainda funciona, as peças sobressalentes geralmente funcionam, mas os métodos das peças agora não são confiáveis. É impossível prever, para qualquer instância específica de Bicycle, se uma peça específica foi inicializada. Objetos segurando uma instância de Bicycle podem, por exemplo, ficar tentados a verificar o estilo antes de enviá-lo tape_color ou rear_shock.

O código não era ótimo para começar; essa mudança não fez nada para melhorá-lo.

A classe inicial Bicycle era imperfeita, mas suas imperfeições estavam ocultas – encapsuladas dentro da classe. Essas novas falhas têm consequências mais amplas. A bicicleta agora tem mais de uma responsabilidade, contém coisas que podem mudar por diversos motivos e não pode ser reutilizada como está.

Esse padrão de codificação levará ao sofrimento, mas tem valor. Ele ilustra vividamente um antipadrão que, uma vez percebido, sugere um design melhor.

Este código contém uma instrução if que verifica *um atributo que contém a categoria self* para determinar qual mensagem enviar para *self*. Isso deve trazer de volta lembranças de um padrão discutido no capítulo anterior sobre digitação com pato, onde você viu uma instrução if que verificava a classe de *um objeto* para determinar qual mensagem enviar para *aquele objeto*.

Em ambos os padrões, um objeto decide qual mensagem enviar com base na categoria do receptor. Você pode pensar na classe de *um objeto* meramente como um caso específico de *um atributo que contém a categoria do self*; considerados desta forma, esses padrões são os mesmos. Em cada caso, se o remetente pudesse falar, estaria dizendo “Eu sei quem você é e por isso sei o que você faz”. Esse conhecimento é uma dependência que aumenta o custo da mudança.

Fique atento a esse padrão. Embora às vezes inocente e ocasionalmente defensável, sua presença pode expor uma falha dispendiosa em seu projeto. O Capítulo 5, Reduzindo custos com a digitação de pato, usou esse padrão para descobrir um tipo de pato ausente; aqui o padrão indica um subtipo ausente, mais conhecido como subclasse.

Encontrando os tipos incorporados A instrução

if no método Spacetime ativa uma variável chamada style, mas teria sido igualmente natural chamar esse tipo ou categoria de variável.

Variáveis com esses tipos de nomes são sua dica para perceber o padrão subjacente.

Tipo e *categoria* são palavras perigosamente semelhantes àquelas que você usaria ao descrever uma classe. Afinal, o que é uma classe senão uma categoria ou tipo?

A variável `style` efetivamente divide as instâncias de `Bicycle` em dois tipos diferentes de coisas. Essas duas coisas compartilham muito comportamento, mas diferem na dimensão de estilo. Parte do comportamento da bicicleta se aplica a todas as bicicletas, alguns apenas às bicicletas de estrada e alguns apenas às bicicletas de montanha. Esta única classe contém vários tipos diferentes, mas relacionados.

Este é exatamente o problema que a herança resolve; o de tipos altamente relacionados que compartilham um comportamento comum, mas diferem em alguma dimensão.

Escolhendo a herança Antes de

prosseguir para o próximo exemplo, vale a pena examinar a herança com mais detalhes. A herança pode parecer uma arte misteriosa, mas, como a maioria das ideias de design, é simples quando vista da perspectiva certa.

Nem é preciso dizer que os objetos recebem mensagens. Não importa quão complicado seja o código, o objeto receptor trata qualquer mensagem de duas maneiras. Ele responde diretamente ou passa a mensagem para algum outro objeto para obter uma resposta.

A herança fornece uma maneira de definir dois objetos como tendo um relacionamento tal que, quando o primeiro recebe uma mensagem que não entende, ele encaminha *automaticamente*, ou delega, a mensagem ao segundo. É simples assim.

A palavra *herança* sugere uma árvore genealógica biológica onde alguns progenitores ficam no topo e os descendentes se ramificam abaixo. Esta imagem da árvore genealógica é, no entanto, um pouco enganadora. Em muitas partes do mundo biológico é comum que os descendentes tenham dois ancestrais. Você, por exemplo, provavelmente tem dois pais. Linguagens que permitem que objetos tenham múltiplos pais são descritas como tendo *herança múltipla* e os projetistas dessas linguagens enfrentam desafios interessantes. Quando um objeto com múltiplos pais recebe uma mensagem que não entende, para qual pai ele deve encaminhar essa mensagem? Se mais de um dos seus pais implementar a mensagem, qual implementação terá prioridade? Como você pode imaginar, as coisas ficam complicadas rapidamente.

Muitas linguagens orientadas a objetos evitam essas complicações fornecendo *herança única*, em que uma subclasse só pode ter uma superclasse pai. Ruby faz isso; tem herança única. Uma superclasse pode ter muitas subclasses, mas cada subclasse só pode ter uma superclasse.

O encaminhamento de mensagens via herança clássica ocorre entre *classes*. Como os tipos `duck` atravessam classes, eles não usam herança clássica para compartilhar comportamento comum. Os tipos `Duck` compartilham código por meio de módulos Ruby (mais sobre módulos no próximo capítulo).

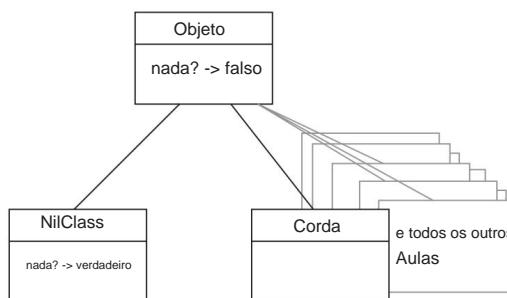
Mesmo que você nunca tenha criado explicitamente sua própria hierarquia de classes, você usa herança. Quando você define uma nova classe, mas não especifica sua superclasse, Ruby define automaticamente a superclasse da sua nova classe como Object. Cada classe que você cria é, por definição, uma subclasse de algo.

Você também já se beneficia da delegação automática de mensagens para superclasses. Quando um objeto recebe uma mensagem que não entende, Ruby encaminha automaticamente essa mensagem para a cadeia da superclasse em busca de uma implementação de método correspondente. Um exemplo simples é ilustrado na Figura 6.3, que mostra como Ruby objetos respondem ao nulo? mensagem.

Lembre-se que em Ruby, nil é uma instância da classe NilClass; é um objeto como qualquer outro. Ruby contém duas implementações de nil?, uma em NilClass e a outro em Objeto. A implementação em NilClass retorna incondicionalmente verdadeiro, aquele em Object, falso.

Quando você envia nada? para uma instância de NilClass, obviamente, responde verdadeiro. Quando você envia nada? para qualquer outra coisa, a mensagem sobe na hierarquia de um superclasse para a próxima até chegar ao Objeto, onde invoca a implementação que responde falso. Assim, nil informa que é *nulo* e todos os outros objetos informam que eles não são. Esta solução elegantemente simples ilustra o poder e a utilidade do herança.

O fato de que mensagens desconhecidas são delegadas na hierarquia da superclasse implica que as subclasses são tudo o que suas superclasses são e muito *mais*. Uma instância de String é uma String, mas também é um Objeto. Toda String é assumida como contendo



Quando uma instância de NilClass recebe o zero? mensagem, é implementação retorna verdadeiro.

Quando instância de outras classes recebe o zero? mensagem, a mensagem sobe automaticamente na superclasse hierarquia para objeto, cujo a implementação retorna falso.

Figura 6.3 NilClass respostas verdadeiras para nada?, string (e todos os outros) respondem falso.

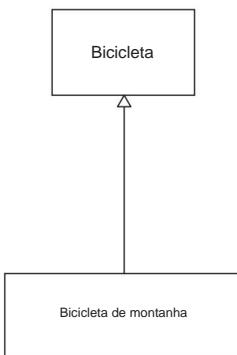


Figura 6.4 MountainBike é uma subclasse de bicicleta.

Toda a interface pública do objeto e deve responder adequadamente a qualquer mensagem definida nessa interface. As subclasses são, portanto, especializações de suas superclasses.

O exemplo atual de Bicycle incorpora vários tipos dentro da classe. É hora de abandonar esse código e voltar para a versão original do Bicycle. Talvez as mountain bikes sejam uma especialização da bicicleta; talvez esse problema de design possa ser resolvido usando herança.

Desenhando relacionamentos de herança Assim

como você usou diagramas de sequência UML para comunicar a passagem de mensagens no Capítulo 4, Criando interfaces flexíveis, você pode usar diagramas de classes UML para ilustrar relacionamentos de classe.

A Figura 6.4 contém um diagrama de classes. As caixas representam classes. A linha de conexão indica que as classes estão relacionadas. O triângulo vazio significa que o relacionamento é de herança. A extremidade pontiaguda do triângulo está anexada à caixa que contém a superclasse. Assim, a figura mostra Bicycle como uma superclasse de MountainBike.

Aplicação incorreta da herança Sob a

premissa de que a jornada é mais útil que o destino e que experimentar erros comuns por procuração é menos doloroso do que experimentá-los pessoalmente, a próxima seção continua a mostrar códigos que não merecem emulação. O código ilustra dificuldades comuns encontradas por novatos. Se você tem prática no uso de herança e se sente confortável com essas técnicas, sinta-se à vontade para dar uma olhada rápida. No entanto, se você é novo em herança ou acha que todas as suas tentativas deram errado, siga em frente com cuidado.

A seguir está uma primeira tentativa de uma subclasse MountainBike . Esta nova subclasse é um descendente direto da classe Bicycle original . Ele implementa dois métodos, inicializar e peças de reposição. Ambos os métodos já estão implementados em Bicycle, portanto, eles são substituídos pelo MountainBike .

No código a seguir, cada um dos métodos substituídos envia super.

```

1 classe MountainBike < Bicicleta
2 attr_reader :front_shock, :rear_shock
3
4 def inicializar (args)
5     @front_shock = args[:front_shock]
6     @rear_shock = args[:rear_shock]
7     super(argumentos)
8 fim
9
10 peças sobressalentes def
11     super.merge (choque traseiro: choque traseiro)
12 fim
13 fim

```

Enviar super em qualquer método passa essa mensagem pela cadeia da superclasse. Assim, para exemplo, o envio de super no método de inicialização do MountainBike (linha 7 acima) invoca o método de inicialização de sua superclasse, Bicycle.

Colocando a nova classe MountainBike diretamente sob a classe Bicycle existente estava cegamente otimista e, previsivelmente, a execução do código expõe diversas falhas. Instâncias de MountainBike têm algum comportamento que simplesmente não faz sentido. O exemplo a seguir mostra o que acontece se você perguntar a uma MountainBike seu tamanho e peças sobressalentes. Ele informa seu tamanho corretamente, mas diz que tem pneus finos e insinua que precisa de fita de guidador, ambas incorretas.

```

1 mountain_bike = MountainBike.new(
2                     tamanho:           'S',
3                     front_shock: 'Manitou',
4                     amortecedor traseiro: 'Raposa')
5
6 mountain_bike.size # -> 'S'
7
8 mountain_bike.peças
9 # -> {tire_size => "23", :chain 10 # => "10
velocidades",

```

```

11 #      :tape_color =>           <- não aplicável
12 #      nulo, :front_shock => 'Manitou',
13 #      :rear_shock => "Fox"

```

Não é nenhuma surpresa que as instâncias de MountainBike contenham uma confusão confusa do comportamento das bicicletas de estrada e de montanha. A classe Bicycle é uma classe concreta que não foi escrito para ser subclassificado. Combina o comportamento geral de todas as bicicletas com o comportamento específico das bicicletas de estrada. Quando você bate MountainBike em Bicycle, você herdam todo esse comportamento – o geral e o específico, seja ele aplicável ou não.

A Figura 6.5 torna sérias liberdades com diagramas de classes para ilustrar essa ideia. Isto mostra o comportamento da bicicleta de estrada incorporado na bicicleta. A forma como este código é organizado faz com que o MountainBike herde um comportamento que não deseja ou não precisa.

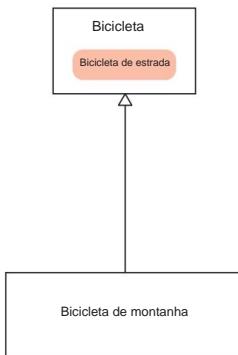


Figura 6.5 Bicicleta combina o comportamento geral da bicicleta com o comportamento específico da bicicleta de estrada.

A classe Bicycle contém um comportamento apropriado tanto para um par quanto para um pai de MountainBike. Alguns dos comportamentos em Bicycle são corretos para MountainBike, alguns estão errados e outros nem mesmo se aplicam. Conforme está escrito, a bicicleta não deve atuar como a superclasse de MountainBike.

Como o design é evolutivo, esta situação surge o tempo todo. O problema aqui começou com os nomes dessas classes.

Encontrando a Abstração

No início havia uma ideia, uma bicicleta, e ela foi modelada como uma classe única, Bicicleta. O designer original escolheu um nome genérico para um objeto que na verdade era um pouco mais especializado. A classe Bicycle existente não representa *qualquer* tipo de bicicleta, ela representa um tipo específico – uma bicicleta de estrada.

Esta escolha de nomenclatura é perfeitamente apropriada em uma aplicação onde toda bicicleta é uma bicicleta de estrada. Quando existe apenas um tipo de bicicleta, escolher RoadBike como nome da classe é desnecessário, talvez até excessivamente específico. Mesmo que você suspeite que um dia terá mountain bikes, a bicicleta é uma ótima escolha para o nome de primeira classe e é suficiente para o dia a dia.

No entanto, agora que existe MountainBike , o nome de Bicycle é enganoso. Esses dois nomes de classes *implicam* herança; você espera imediatamente que a MountainBike seja uma especialização da bicicleta. É natural escrever código que crie MountainBike como uma subclasse de Bicycle. Esta é a estrutura correta, os nomes das classes estão corretos, mas o código em Bicycle agora está muito errado.

As subclasses são *especializações* de suas superclasses. Uma MountainBike deve ser tudo o que uma bicicleta é e muito mais. Qualquer objeto que espera uma bicicleta deve ser capaz de interagir com uma MountainBike na feliz ignorância de sua classe real.

Estas são as regras da herança; quebrá-los por sua conta e risco. Para que a herança funcione, duas coisas devem ser sempre verdadeiras. Primeiro, os objetos que você está modelando devem realmente ter um relacionamento de generalização-especialização. Segundo, você deve usar as técnicas de codificação corretas.

Faz todo o sentido modelar a mountain bike como uma especialização da bicicleta; a relação está correta. No entanto, o código acima é uma bagunça e, se propagado, levará ao desastre. A atual classe Bicycle mistura o código geral das bicicletas com o código específico das bicicletas de estrada. É hora de separar essas duas coisas, de mover o código da bicicleta de estrada de Bicycle para uma subclasse RoadBike separada.

Criando uma Superclasse Abstrata A Figura 6.6

mostra um novo diagrama de classes onde Bicycle é a superclasse de MountainBike e RoadBike. Este é o seu objetivo; é a estrutura de herança que você pretende criar. Bicycle conterá o comportamento comum, e MountainBike e RoadBike adicionarão especializações. A interface pública da bicicleta deverá incluir peças sobressalentes e tamanho, e as interfaces de suas subclasses adicionarão suas peças individuais.

Bicycle agora representa uma classe *abstrata* . O Capítulo 3, Gerenciando Dependências, definiu abstrato como sendo desassociado de qualquer instância específica, e essa definição ainda é válida. Esta nova versão da Bicicleta não definirá uma bicicleta completa, apenas os pedaços que todas as bicicletas partilham. Você pode esperar criar instâncias de MountainBike e RoadBike, mas Bicycle não é uma classe para a qual você enviaria a nova mensagem. Não faria sentido; A bicicleta não representa mais uma bicicleta inteira.

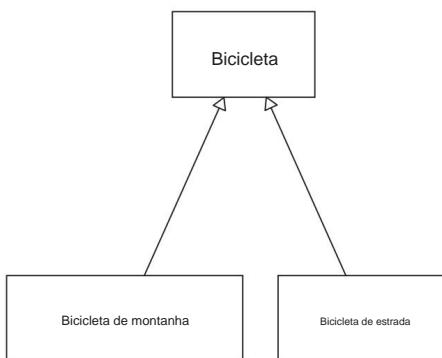


Figura 6.6 Bicicleta como a superclasse de Bicicleta de montanha e Bicicleta de estrada.

Algumas linguagens de programação orientadas a objetos possuem sintaxe que permite declarar explicitamente as classes como abstratas. Java, por exemplo, possui a palavra-chave abstrata . O próprio compilador Java impede a criação de instâncias de classes às quais esta palavra-chave foi aplicado. Ruby, de acordo com sua natureza confiável, não contém tal palavra-chave e não impõe tal restrição. Somente o bom senso impede que outros programadores criação de instâncias de Bicicleta; na vida real, isso funciona muito bem.

Classes abstratas existem para serem subclassificadas. Este é o seu único propósito. Eles fornecem um repositório comum para comportamento que é compartilhado por um conjunto de subclasses – subclasses que por sua vez fornecem especializações.

Quase nunca faz sentido criar uma superclasse abstrata com apenas uma classe. Mesmo que a classe Bicycle original contenha comportamento geral e específico e é possível imaginar modelá-lo como duas classes desde o início, faça não. Independentemente de quão fortemente você espera ter outros tipos de bicicletas, naquele dia pode nunca chegar. Até que você tenha um requisito específico que o obrigue a lidar com outras bicicletas, a classe atual de bicicletas é boa o suficiente.

Mesmo que agora você precise de dois tipos de bicicletas, isso ainda pode não ser o momento certo para se comprometer com a herança. Criar uma hierarquia tem custos; o melhor Uma maneira de minimizar esses custos é maximizar suas chances de obter a abstração correta antes de permitir que subclasses dependam dele. Embora as duas bicicletas que você conhece forneciam uma boa quantidade de informações sobre a abstração comum, três bicicletas forneceriam muito mais. Se você pudesse adiar essa decisão até que FastFeet pedisse um terceiro tipo de bicicleta, suas chances de encontrar a abstração certa aumentariam dramaticamente.

A decisão de adiar a criação da hierarquia de bicicletas obriga você a escrever Classes MountainBike e RoadBike que duplicam uma grande quantidade de código. Uma decisão de prosseguir com a hierarquia aceita o risco de ainda não ter informações suficientes para identificar a abstração correta. Sua escolha sobre esperar ou prosseguir

depende de quanto tempo você espera que uma terceira bicicleta apareça e quanto você espera que custe a duplicação. Se uma terceira bicicleta for iminente, talvez seja melhor duplicar o código e aguardar melhores informações. No entanto, se o código duplicado precisar ser alterado todos os dias, pode ser mais barato prosseguir e criar a hierarquia. Você deve esperar, se puder, mas não tenha medo de avançar com base em dois casos concretos, se isso lhe parecer melhor.

Por enquanto, suponha que você tenha um bom motivo para criar uma hierarquia de bicicletas , mesmo que conheça apenas duas bicicletas. O primeiro passo na criação da nova hierarquia é criar uma estrutura de classes que espelhe a Figura 6.6. Ignorando por um momento a correção do código, a maneira mais simples de fazer essa alteração é renomear Bicycle para RoadBike e criar uma nova classe Bicycle vazia. O exemplo a seguir faz exatamente isso.

```

1 classe Bicicleta # Esta
2 classe agora está vazia.
3 # Todo o código foi movido para RoadBike.
4 fim
5
RoadBike de 6 classes < Bicicleta
7 # Agora uma subclasse de Bicycle.
8 # Contém todo o código da antiga classe Bicycle.
9 fim
10
11 class MountainBike < Bicycle 12 # Ainda uma
subclasse de Bicycle (que agora está vazia).
13 # O código não mudou.
14 fim

```

A nova classe RoadBike é definida como uma subclasse de Bicycle. A classe MountainBike existente já é uma subclasse de Bicicleta. Seu código não mudou, mas seu comportamento certamente mudou porque sua superclasse agora está vazia. O código do qual o MountainBike depende foi removido de seu pai e colocado em um par.

Esse rearranjo de código apenas mudou o problema, conforme ilustrado na Figura 6.7. Agora, em vez de conter muito comportamento, Bicycle não contém nenhum comportamento. O comportamento comum exigido por todas as bicicletas está preso dentro da RoadBike e, portanto, é inacessível à MountainBike.

Esse rearranjo melhora muito porque é mais fácil promover o código a uma superclasse do que rebaixá-lo a uma subclasse. As razões para isto ainda não são óbvias, mas o serão à medida que o exemplo avança.

As próximas iterações concentram-se em alcançar esta nova estrutura de classes, movendo o comportamento comum para Bicycle e usando esse comportamento de forma eficaz nas subclasses.

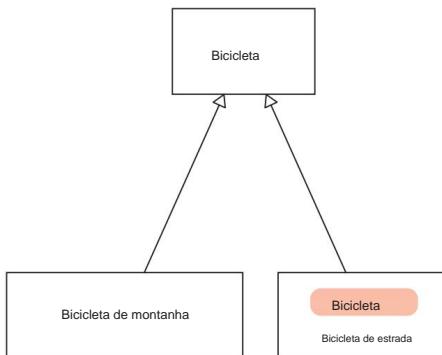


Figura 6.7 Agora

Bicicleta de estrada Contém todo o comportamento comum.

RoadBike ainda contém tudo o que precisa e, portanto, ainda funciona, mas MountainBike agora está seriamente quebrado. Por exemplo, eis o que acontece se você crie instâncias de cada subclasse e pergunte o tamanho delas. RoadBike retorna o correto resposta, MountainBike simplesmente explode.

```

1 road_bike = RoadBike.new(
2                         tamanho:           'M',
3                         tape_color: 'vermelho' )
4
5 road_bike.size #=> "M"
6
7 mountain_bike = MountainBike.new(
8                         tamanho:           'S',
9                         front_shock: 'Manitou',
10                        amortecedor traseiro: 'Raposa')
11
12 mountain_bike.tamanho
13 # NoMethodError: método indefinido 'tamanho'
  
```

É óbvio porque esse erro ocorre; nem MountainBike nem qualquer uma de suas superclasses implementar tamanho.

Promovendo Comportamento Abstrato

Os métodos de tamanho e peças sobressalentes são comuns a todas as bicicletas. Esse comportamento pertence Interface pública da bicicleta . Ambos os métodos estão atualmente bloqueados no RoadBike; o

A tarefa aqui é movê-los para Bicicleta para que o comportamento possa ser compartilhado. Como o código que trata do tamanho é mais simples, é o lugar mais natural para começar.

Promover o comportamento de tamanho para a superclasse requer três alterações, conforme mostrado no exemplo abaixo. O leitor de atributos e o código de inicialização passam de RoadBike para Bicycle (linhas 2 e 5), e o método de inicialização de RoadBike adiciona um envio de super (linha 14).

```

1 classe Bicicleta 2
attr_reader :tamanho           # <- promovido da RoadBike
3
4 def inicializar(args={})
5     @size = args[:size] # <- promovido da RoadBike
6 fim
7 fim
8
9 classe RoadBike < Bicicleta 10
attr_reader :tape_color
11
12 def inicializar (args)
13     @tape_color = args[:tape_color] super(args) # <-
14     RoadBike agora DEVE enviar 'super'
15 fim
16 #...
17 fim

```

RoadBike agora herda o método size de Bicycle. Quando uma RoadBike recebe tamanho, o próprio Ruby delega a mensagem na cadeia da superclasse, procurando por uma implementação e encontrando aquela em Bicycle. Esta delegação de mensagens acontece automaticamente porque RoadBike é uma subclasse de Bicycle.

Compartilhar o código de inicialização que define a variável @size , entretanto, exige um pouco mais de você. Esta variável é definida no método de inicialização de Bicycle , um método que RoadBike também implementa ou substitui.

Quando RoadBike substitui a inicialização, ele fornece um receptor para esta mensagem, que satisfaz perfeitamente Ruby e evita a delegação automática da mensagem para Bicycle. Se ambos os métodos de inicialização precisarem ser executados, a RoadBike agora será obrigada a fazer a própria delegação; ele deve enviar super para passar explicitamente esta mensagem para Bicycle, como fez na linha 14 acima.

Antes dessa mudança, a RoadBike respondia corretamente ao tamanho , mas a MountainBike não. O comportamento que eles têm em comum agora definido em Bicicleta, seu comportamento comum

superclasse. A magia da herança é tal que agora ambos respondem corretamente ao tamanho como mostrado abaixo.

```

1 road_bike = RoadBike.new(
2                         tamanho:           'M',
3                         tape_color: 'vermelho' )
4
5 road_bike.size # -> "M"
6
7 mountain_bike = MountainBike.new(
8                         tamanho:           'S',
9                         front_shock: 'Manitou',
10                        amortecedor traseiro: 'Raposa')
11
12 mountain_bike.size # -> 'S'

```

O leitor de alerta notará que o código que trata do tamanho da bicicleta foi movido duas vezes. Isto estava na classe original de bicicleta , foi transferido para RoadBike e agora foi promovido de volta para Bicicleta. O código não mudou; acabou de ser movido duas vezes.

Você pode ficar tentado a ignorar o intermediário e simplesmente deixar esse trecho de código em Bicicleta para começar, mas essa estratégia de empurrar tudo para baixo e depois puxar algumas coisas para cima é uma parte importante dessa refatoração. Muitas das dificuldades da herança são causadas pela incapacidade de separar rigorosamente o concreto do abstrato.

O código original da bicicleta misturava os dois. Se você começar esta refatoração com isso primeira versão do Bicycle, tentando isolar o código concreto e empurrá- lo para RoadBike, qualquer falha de sua parte deixará perigosos vestígios de concretude em a superclasse. No entanto, se você começar movendo cada pedaço do código Bicycle para RoadBike, você pode identificar e promover cuidadosamente as partes abstratas sem medo de deixar artefatos concretos.

Ao decidir entre estratégias de refatoração, na verdade, ao decidir entre estratégias de design estratégias em geral, é útil fazer a pergunta: “O que acontecerá se eu estiver errado?” Em neste caso, se você criar uma superclasse vazia e inserir os bits abstratos de código nela, o pior que pode acontecer é você não conseguir encontrar e promover toda a abstração.

Esta falha de “promoção” cria um problema simples, que é facilmente encontrado e facilmente resolvido. fixo. Quando um pouco da abstração fica para trás, o descuido se torna visível assim que já que outra subclasse precisa do mesmo comportamento. Para dar a todas as subclasses acesso ao comportamento, você será forçado a duplicar o código (em cada subclasse) ou promovê-lo (para a superclasse comum). Porque até os programadores mais juniores foram ensinados para não duplicar o código, esse problema é percebido independentemente de quem trabalha no aplicativo

no futuro. O curso natural dos acontecimentos é tal que a abstração é identificada e promovida e o código melhora. As falhas de promoção têm, portanto, baixas consequências.

Entretanto, se você tentar esta refatoração na direção oposta, tentando converter uma classe existente de concreta para abstrata empurrando apenas as partes concretas em uma nova subclasse, você pode acidentalmente deixar resquícios de comportamento concreto atrás. Por definição, este comportamento concreto residual não se aplica a todos os possíveis nova subclasse. As subclasses começam assim a violar a regra básica da herança; eles são não são verdadeiramente especializações de suas superclasses. A hierarquia se torna indigna de confiança.

Hierarquias não confiáveis forçam os objetos que interagem com elas a conhecerem seus peculiaridades. Programadores inexperientes não entendem e não conseguem consertar uma hierarquia defeituosa; quando solicitados a usar um, eles incorporarão o conhecimento de suas peculiaridades em seus próprios código, muitas vezes verificando explicitamente as classes de objetos. Conhecimento da estrutura de a hierarquia vaza para o resto do aplicativo, criando dependências que aumentam o custo da mudança. Este não é um problema que você queira deixar para trás. As consequências de uma falha de rebaixamento pode ser generalizada e grave.

A regra geral para refatorar em uma nova hierarquia de herança é organizar código para que você possa promover abstrações em vez de rebaixar concreções.

À luz desta discussão, a questão colocada há alguns parágrafos poderia ser mais ser formulado de forma útil: "O que acontecerá quando eu estiver errado?" Cada decisão que você toma inclui dois custos: um para implementá-lo e outro para alterá-lo quando você descobrir que você estava errado. Levando ambos os custos em consideração ao escolher entre alternativas motiva você a fazer escolhas conservadoras que minimizem o custo da mudança.

Com isso em mente, volte sua atenção para as peças sobressalentes.

Separando o abstrato do concreto

RoadBike e MountainBike implementam uma versão de peças sobressalentes. Bicicletas de estrada definição (repetida abaixo) é a original que foi copiada do concreto
Aula de bicicleta . É independente e, portanto, ainda funciona.

```

1 classe RoadBike < Bicicleta
2     ...
3 peças sobressalentes definitivas
4         {cadeia: '10 velocidades',
5             tamanho_do_pneu: '23',
6             cor_da_fita: cor_da_fita}
7 fim
8 fim

```

A definição de peças sobressalentes em MountainBike (também repetida abaixo) é remanescente da primeira tentativa de subclassificação. Este método envia super, esperando que uma superclasse também implemente peças sobressalentes.

```

1 classe MountainBike < Bicicleta
2     ...
3 peças sobressalentes definitivas
4         super.merge({rear_shock: rear_shock})
5 fim
6 fim

```

A bicicleta, no entanto, ainda não implementa o método de peças sobressalentes , portanto, enviar peças sobressalentes para uma MountainBike resulta na seguinte exceção NoMethodError :

```

1 mountain_bike.spares 2 #
NoMethodError: super: nenhum método de superclasse 'spares'

```

A correção desse problema obviamente requer a adição de um método de reposição ao Bicycle, mas fazer isso não é tão simples quanto promover o código existente do RoadBike.

A implementação de peças sobressalentes da RoadBike sabe demais. Os atributos chain e tire_size são comuns a todas as bicicletas, mas tape_color deve ser conhecido apenas por bicicletas de estrada. Os valores chain e tire_size codificados não são os padrões corretos para todas as subclasses possíveis. Este método tem muitos problemas e não pode ser promovido tal como está.

Ele mistura um monte de coisas diferentes. Quando essa mistura estranha estava escondida dentro de um único método de uma única classe, era possível sobreviver, até mesmo (dependendo da sua tolerância) ignorável, mas agora que você gostaria de compartilhar apenas parte desse comportamento, você deve desvendar a bagunça e separar as partes abstratas das partes concretas.

As captações serão promovidas até Bicicleta, as partes de concreto permanecerão em Bicicleta de estrada.

Deixe de lado por um momento os pensamentos sobre o método geral de peças sobressalentes e concentre-se em promover apenas as peças que todas as bicicletas compartilham, corrente e tamanho do pneu. Eles são atributos, como tamanho, e devem ser representados por acessadores e setters em vez de valores codificados. Aqui estão os requisitos:

- As bicicletas têm corrente e tamanho de pneu.
- Todas as bicicletas compartilham o mesmo padrão de corrente.

- As subclasses fornecem seu próprio padrão para tamanho de pneu.
- Instâncias concretas de subclasses podem ignorar padrões e fornecer valores específicos da instância.

O código para coisas semelhantes deve seguir um padrão semelhante. Aqui está o novo código que lida com size, chain e tire_size de maneira semelhante.

Bicicleta de 1 classe

```

2 attr_reader :tamanho, :chain, :tire_size
3
4 def inicializar(args={})
5     @tamanho      = args[:tamanho]
6     @corrente     = args[:cadeia]
7     @tamanho_do_pneu = args[:tamanho_do_pneu]
8 fim
9     #...
10 fim

```

RoadBike e MountainBike herdam as definições attr_reader em Bicycle e ambos enviam super em seus métodos de inicialização . Todas as bicicletas agora entendem o tamanho, chain e tire_size e cada um pode fornecer valores específicos da subclasse para esses atributos. O primeiro e o último requisitos listados acima foram atendidos.

Apesar do acúmulo, não há nada de especial nesse código. O bom senso sugere que deveria ter sido escrito assim no início; já é hora dessa versão apareceu. É herdável por subclasses, certamente, mas nada no código sugere que espera ser herdado.

Atender aos dois requisitos que tratam da inadimplência, entretanto, acrescenta algo interessante.

Usando o padrão de método de modelo

A próxima mudança altera o método de inicialização do Bicycle para enviar mensagens para obter padrões. Existem duas novas mensagens, default_chain e default_tire_size, em linhas 6 e 7 abaixo.

Embora agrupar os padrões em métodos seja uma boa prática em geral, esses novos os envios de mensagens têm um propósito duplo. O principal objetivo da Bicycle ao enviar essas mensagens é para dar às subclasses a oportunidade de contribuir com especializações, substituindo-as.

Esta técnica de definir uma estrutura básica na superclasse e enviar mensagens adquirir contribuições específicas da subclasse é conhecido como padrão *de método de modelo* .

No código a seguir, MountainBike e RoadBike aproveitam apenas um dessas oportunidades de especialização. Ambos implementam default_tire_size, mas nenhum deles implementa default_chain. Cada subclasse, portanto, fornece seu próprio padrão para tamanho do pneu, mas herda o padrão comum para corrente.

Bicicleta de 1 classe

```

2 attr_reader :tamanho, :chain, :tire_size
3
4 def inicializar(args={})
5     @tamanho      = args[:tamanho]
6     @corrente     = args[:chain] @tire_size      || cadeia_padrão
7     = args[:tire_size] || tamanho_do_pneu_padrão
8 fim
9
10 def default_chain '10 velocidades'          # <- padrão comum
11
12 fim
13 fim
14
15 RoadBike classe 15 < Bicicleta
16 #...
17 def default_tire_size # <- padrão da subclasse
18     '23'
19 fim
20 fim
21
22 MountainBike classe 22 < Bicicleta
23 #...
24 def default_tire_size # <- padrão da subclasse
25     '2.1'
26 fim
27 fim

```

Bicycle agora fornece estrutura, um algoritmo comum, se preferir, para suas subclasses.

Onde lhes permite influenciar o algoritmo, envia mensagens. As subclasses contribuem para o algoritmo implementando métodos de correspondência.

Todas as bicicletas agora compartilham o mesmo padrão para corrente, mas usam padrões diferentes para pneus tamanho, conforme mostrado abaixo:

```

1 road_bike = RoadBike.new(
2                         tamanho:           'M',

```

```

3           tape_color: 'vermelho' )
4
5 road_bike.tire_size 6           # => '23'
road_bike.chain             # => "10 velocidades"
7
8 mountain_bike = MountainBike.new(
9           tamanho:           'S',
10          front_shock: 'Manitou',
11          amortecedor traseiro: 'Raposa')
12
13 mountain_bike.tire_size # => '2.1'
14 road_bike.chain # => "10 velocidades"

```

Porém, é muito cedo para comemorar esse sucesso, porque ainda há algo errado com o código. Ele contém uma armadilha, aguardando os incautos.

Implementando cada método de modelo

O método de inicialização da bicicleta envia `default_tire_size`, mas a própria bicicleta envia não implementá-lo. Esta omissão pode causar problemas a jusante. Imagine isso FastFeed adiciona outro novo tipo de bicicleta, a reclinada. Os reclinados são baixos, longos bicicletas que colocam o ciclista numa posição reclinada e descontraída; essas bicicletas são rápidas e fácil nas costas e no pescoço do piloto.

O que acontece se algum programador criar inocentemente uma nova `RecumbentBike` subclasse, mas deixa de fornecer uma implementação `default_tire_size`? Ele encontra o seguinte erro.

```

1 classe Bicicleta Reclinada < Bicicleta
2 def default_chain
3     '9 velocidades'
4 fim
5 fim
6
7 dobrado = RecumbentBike.new
8 # NameError: variável ou método local indefinido
9 # 'default_tire_size'

```

O projetista original da hierarquia raramente encontra esse problema. Ela escreveu Bicicleta; ela entende os requisitos que as subclasses devem atender. O existente código funciona. Esses erros ocorrem no futuro, quando o aplicativo estiver sendo alterado para

atendem a um novo requisito e são encontrados por outros programadores, que entendem muito menos sobre o que está acontecendo.

A raiz do problema é que Bicycle impõe um requisito às suas subclasses que não é óbvio à primeira vista no código. À medida que Bicycle é escrita, as subclasses *devem* implementar default_tire_size. Subclasses inocentes e bem-intencionadas como RecumbentBike podem falhar porque não atendem aos requisitos dos quais são inconsciente.

Um mundo de possíveis danos pode ser amenizado antecipadamente seguindo uma regra simples. Qualquer classe que usa o padrão de método template deve fornecer uma implementação para cada mensagem que envia, mesmo que a única implementação razoável na classe remetente seja assim:

Bicicleta de 1 classe

```
2#...
3 def default_tire_size aumenta
4     NotImplemented
5 fim
6 fim
```

Afirmar explicitamente que as subclasses são necessárias para implementar uma mensagem fornece documentação útil para aqueles em quem se pode confiar para lê-la e mensagens de erro úteis para aqueles que não podem.

Depois que a Bicycle fornece essa implementação de default_tire_size, a criação de uma nova RecumbentBike falha com o seguinte erro.

```
1 dobrado = RecumbentBike.new
2# NotImplemented: NotImplemented
```

Embora seja perfeitamente aceitável simplesmente gerar esse erro e confiar no rastreamento de pilha para rastrear sua origem, você também pode fornecer explicitamente informações adicionais, conforme mostrado na linha 5 abaixo.

Bicicleta de 1 classe

```
2#...
3 def default_tire_size raise
4     NotImplemented, "Este #{self.class}
5         não pode responder a:"
6 fim
7 fim
```

Esta informação adicional torna o problema inevitavelmente claro. Como executar isso o código mostra, esta RecumbentBike precisa de acesso a uma implementação de tamanho_do_pneu_padrão.

```
1 dobrado = RecumbentBike.new
2 #NotImplementedError:
3 #      Esta RecumbentBike não pode responder a:
4 #          'default_tire_size'
```

Quer seja encontrado dois minutos ou dois meses depois de escrever a RecumbentBike classe, esse erro é inequívoco e facilmente corrigido.

Criar código que falha com mensagens de erro razoáveis exige pouco esforço no presente, mas fornece valor para sempre. Cada mensagem de erro é uma coisa pequena, mas pequena as coisas se acumulam para produzir grandes efeitos e é essa atenção aos detalhes que marca você como um programador sério. Sempre documente os requisitos do método de modelo por implementar métodos de correspondência que levantam erros úteis.

Gerenciando o acoplamento entre superclasses e subclasses

Bicycle agora contém a maior parte do comportamento abstrato da bicicleta. Tem código para gerenciar tamanho geral da bicicleta, corrente e tamanho do pneu, e sua estrutura convida as subclasses a fornecer padrões comuns para esses atributos. A superclasse está quase completa; está faltando apenas uma implementação de peças sobressalentes.

Essa implementação da superclasse sobressalente pode ser escrita de várias maneiras; o as alternativas variam na forma como elas acoplam as subclasses e superclasses. Gerenciar o acoplamento é importante; classes fortemente acopladas permanecem juntas e podem ser impossível mudar de forma independente.

Esta seção mostra duas implementações diferentes de peças sobressalentes – uma maneira fácil e óbvia um e outro um pouco mais sofisticado, mas também mais robusto.

Compreendendo o acoplamento

Esta primeira implementação de peças sobressalentes é mais simples de escrever, mas produz a mais precisa aulas acopladas.

Lembre-se de que a implementação atual do RoadBike é assim:

```

1 classe RoadBike < Bicicleta
2     #...
3 peças sobressalentes definitivas
4     {cadeia: '10 velocidades',
5         tamanho_do_pneu: '23',
6         cor_da_fita: cor_da_fita}
7 fim
8 fim

```

Este método é uma mistura de coisas diferentes e a última tentativa de promovê-lo fez um desvio para limpar o código. Esse desvio extraiu os valores codificados para cadeia e pneu em variáveis e mensagens, e promoveu apenas essas partes no Bicicleta. Os métodos que lidam com o tamanho da corrente e do pneu estão agora disponíveis no superclasse.

A implementação atual de peças sobressalentes da MountainBike é assim:

```

1 classe MountainBike < Bicicleta
2     #...
3 peças sobressalentes definitivas
4     super.merge({rear_shock: rear_shock})
5 fim
6 fim

```

O método de peças sobressalentes da MountainBike envia super; espera que uma de suas superclasses implementar peças sobressalentes. MountainBike mescla seu próprio hash de peças de reposição no resultado retornado por super, esperando claramente que o resultado também seja um hash.

Dado que a bicicleta agora pode enviar mensagens para obter o tamanho da corrente e do pneu e que seu implementação de peças sobressalentes deve retornar um hash, adicionando o seguinte método sobressalentes atende às necessidades da MountainBike .

Bicicleta de 1 classe

```

2 #...
3 peças sobressalentes definitivas
4     {tamanho_do_pneu:tamanho_do_pneu,
5         corrente:           corrente}
6 fim
7 fim

```

Uma vez que este método é colocado em Bicycle, todo o MountainBike funciona. Trazendo RoadBike é apenas uma questão de mudar sua implementação de peças sobressalentes para espelhar MountainBike's, ou seja, substituindo o código do tamanho da corrente e do pneu por um envio para super e adicionando as especializações para bicicletas de estrada ao hash resultante.

Supondo que esta alteração final no MountainBike tenha sido feita, a listagem a seguir mostra todo o código escrito até agora e completa a primeira implementação deste hierarquia.

Observe que o código segue um padrão discernível. Cada método de modelo enviado por A bicicleta é implementada na própria bicicleta , e tanto na MountainBike quanto na RoadBike envie super em seus métodos de inicialização e peças de reposição .

Bicicleta de 1 classe

```

2 attr_reader :tamanho, :chain, :tire_size
3
4 def inicializar(args={})
5     @tamanho      = args[:tamanho]
6     @corrente       = args[:chain]           || cadeia_padrão
7     @tire_size = args[:tire_size] || tamanho_do_pneu_padrão
8 fim
9
10 peças sobressalentes def
11     {tamanho_do_pneu:tamanho_do_pneu,
12      corrente:       corrente}
13 fim
14
15 def default_chain
16     '10 velocidades'
17 fim
18
19 def tamanho_do_pneu_padrão
20     aumentar NotImplementedError
21 fim
22 fim
23
24 RoadBike classe 24 < Bicicleta
25 attr_reader :tape_color
26
27 def inicializar(args)
28     @tape_color = args[:tape_color]
29     super(argumentos)
30 fim

```

```

31
32 peças sobressalentes def
33     super.merge({ tape_color: tape_color})
34 fim
35
36 def tamanho_do_pneu_padrão
37     '23'
38 fim
39 fim
40
41 classe MountainBike < Bicicleta
42 attr_reader :front_shock, :rear_shock
43
44 def inicializar(args)
45     @front_shock = args[:front_shock]
46     @rear_shock = args[:rear_shock]
47     super(argumentos)
48 fim
49
50 peças sobressalentes def
51     super.merge({rear_shock: rear_shock})
52 fim
53
54 def tamanho_do_pneu_padrão
55     '2.1'
56 fim
57 fim

```

Essa hierarquia de classes funciona e você pode ficar tentado a parar por aqui. No entanto, só porque funciona não garante que seja bom o suficiente. Ainda contém um peito armadilha que vale a pena remover.

Observe que as subclasses MountainBike e RoadBike seguem um padrão semelhante. Cada um deles sabe coisas sobre si mesmo (suas especializações em peças de reposição) e coisas sobre sua superclasse (que implementa peças sobressalentes para retornar um hash e que responde à inicialização).

Saber coisas sobre outras classes, como sempre, cria dependências e dependências acoplam objetos. As dependências no código acima também são o booby armadilhas; ambos são criados pelos envios de super nas subclasses.

Aqui está uma ilustração da armadilha. Se alguém criar uma nova subclass e esquecer de send super em seu método de inicialização , ele encontra este problema:

```

1 classe Bicicleta Reclinada < Bicicleta
2 attr_reader :flag
3
4 def inicializar (args)
5   @flag = args[:flag] # esqueci de enviar 'super'
6 fim
7
8 peças sobressalentes def
9   super.merge({bandeira: bandeira})
10 fim
11
12 def default_chain
13   '9 velocidades'
14 fim
15
16 def tamanho_do_pneu_padrão
17   '28'
18 fim
19 fim
20
21 dobrado = RecumbentBike.new(flag: 'alto e laranja')
22 peças sobressalentes dobradas
23 # -> {:tire_size => nulo, <- não foi inicializado
24 #:chain => nil, => "alto e laranja"} 25 #
      :bandeira

```

Quando a RecumbentBike não envia super durante a inicialização , ela perde o inicialização comum fornecida pela Bicycle e não obtém um tamanho, cadeia ou tamanho do pneu. Este erro pode manifestar-se num momento e local muito distante da sua causa, tornando é muito difícil depurar.

Um problema igualmente diabólico ocorre se a RecumbentBike se esquecer de enviar super seu método de reposição . Nada explode, em vez disso, o hash de peças sobressalentes está errado e esse erro pode não se tornar aparente até que um mecânico esteja parado na estrada com uma bicicleta quebrada, procurando em vão na caixa de peças de reposição.

Qualquer programador pode esquecer de enviar super e portanto causar esses erros, mas os principais culpados (e as principais vítimas) são programadores que não conhecem o codificam bem, mas têm a tarefa, no futuro, de criar novas subclasses de Bicycle.

O padrão de código nesta hierarquia exige que as subclasses não apenas saibam o que fazem, mas também como devem interagir com sua superclasse. Faz sentido de que as subclasses conhecem as especializações com as quais contribuem (elas são obviamente as

apenas classes que *podem* conhecê-los), mas forçando uma subclasse a saber como interagir com sua superclasse abstrata causa muitos problemas.

Ele empurra o conhecimento do algoritmo para as subclasses, forçando cada uma a envie explicitamente super para participar. Causa duplicação de código entre subclasses, exigindo que todos enviem super exatamente nos mesmos lugares. E isso aumenta a chance que futuros programadores criariam erros ao escrever novas subclasses, porque pode-se confiar que os programadores incluirão as especializações corretas, mas podem facilmente esqueça de enviar super.

Quando uma subclasse envia super, ela está efetivamente declarando que conhece o algoritmo; depende desse conhecimento. Se o algoritmo mudar, então as subclasses podem equilíbrio se suas próprias especializações não forem afetadas de outra forma.

Desacoplando subclasses usando mensagens de gancho

Todos esses problemas podem ser evitados com uma refatoração final. Em vez de permitir subclasses para conhecer o algoritmo e exigindo que enviem super, superclasses podem em vez disso, envie mensagens *de gancho*, aquelas que existem apenas para fornecer às subclasses um local para contribuir com informações, implementando métodos de correspondência. Esta estratégia remove conhecimento do algoritmo da subclasse e retorna o controle para a superclasse.

No exemplo a seguir, esta técnica é usada para fornecer às subclasses uma maneira de contribuir para a inicialização. O método de inicialização da bicicleta agora envia post_initialize e, como sempre, implementa o método correspondente, que neste caso não faz nada.

RoadBike fornece sua própria inicialização especializada, substituindo post_initialize, como você vê aqui:

Bicicleta de 1 classe	<pre> 2 3 def inicializar(args={}) 4 @tamanho = args[:tamanho] 5 @corrente = args[:chain] cadeia_padrão 6 @tire_size = args[:tire_size] tamanho_do_pneu_padrão 7 8 post_initialize(args) # Bicicleta envia ambos 9 fim 10 11 def post_initialize(args) # e implementa isso 12 nada 13 fim 14 #... </pre>
-----------------------	--

```

15 fim
16
RoadBike classe 17 < Bicicleta
18
19 def post_initialize(args) @tape_color = # RoadBike pode
20     args[:tape_color] # opcionalmente # substitui-o
21 fim
22 #...
23 fim

```

Essa mudança não apenas remove o envio de super do método de inicialização do RoadBike , mas também remove completamente o método de inicialização . RoadBike não controla mais a inicialização; em vez disso, contribui com especializações para um algoritmo abstrato maior. Esse algoritmo é definido na superclasse abstrata Bicycle, que por sua vez é responsável por enviar post_initialize.

RoadBike ainda é responsável pela *inicialização* necessária, mas não é mais responsável por *quando* sua inicialização ocorre. Esta mudança permite que a RoadBike conheça menos sobre a Bicicleta, reduzindo o acoplamento entre elas e tornando cada uma mais flexível diante de um futuro incerto. RoadBike não sabe quando seu método post_initialize será chamado e não se importa com qual objeto realmente envia a mensagem. A bicicleta (ou qualquer outro objeto) pode enviar esta mensagem a qualquer momento, não há necessidade de que ela seja enviada durante a inicialização do objeto.

Colocar o controle do tempo na superclasse significa que o algoritmo pode mudar sem forçar mudanças nas subclasses.

Essa mesma técnica pode ser usada para remover o envio de super do método de peças sobressalentes . Em vez de forçar a RoadBike a saber que a Bicycle implementa peças sobressalentes e que a implementação da Bicycle retorna um hash, você pode afrouxar o acoplamento implementando um gancho que devolve o controle à Bicycle.

O exemplo a seguir altera o método de peças de bicicleta para enviar local_spares. Bicycle fornece uma implementação padrão, que retorna um hash vazio. RoadBike aproveita esse gancho e o substitui para retornar sua própria versão de local_spares, adicionando peças sobressalentes específicas para bicicletas de estrada.

```

Bicicleta de 1 classe
2     #...
3 peças sobressalentes
4     def {tamanho_do_pneu:tamanho_do_pneu,
5         corrente:      cadeia}.merge(local_spares)
6 fim

```

```

7
8      # gancho para substituição de subclasses
9 def local_spares
10     {}
11 fim
12
13 fim
14
RoadBike classe 15 < Bicicleta
16 #...
17 def local_spares
18     {tape_color: fita_color}
19 fim
20
21 fim

```

A nova implementação de local_spares da RoadBike substitui sua implementação anterior de peças sobressalentes.

Esta mudança preserva a especialização fornecida pela RoadBike , mas

reduz seu acoplamento à Bicicleta. RoadBike não precisa mais saber que bicicleta implementa um método de reposição ; apenas espera que a sua própria implementação

local_spares será chamado por algum objeto, em algum momento.

Depois de fazer alterações semelhantes no MountainBike, a hierarquia final fica assim:

```

Bicicleta de 1 classe
2 attr_reader :tamanho, :cadeia, :tire_size
3
4 def inicializar(args={})
5     @tamanho = args[:tamanho]
6     @corrente      = args[:cadeia] || cadeia_padrão
7     @tamanho_do_pneu = args[:tamanho_do_pneu] || tamanho_do_pneu_padrão
8     post_initialize(args)
9 fim
10
11 peças sobressalentes def
12     {tamanho_do_pneu:tamanho_do_pneu,
13      corrente:      cadeia}.merge(local_spares)
14 fim
15
16 def tamanho_do_pneu_padrão
17     aumentar NotImplementedError
18 fim
19

```

```
20 # subclasses podem substituir 21 def post_initialize(args)
21
22
23 fim
24
25 def local_spares
26     {}
27 fim
28
29 def default_chain '10 velocidades'
30
31 fim
32
33 fim
34
35 classe RoadBike < Bicicleta 36
attr_reader :tape_color
37
38 def post_initialize(args) @tape_color =
39     args[:tape_color]
40 fim
41
42 def local_spares
43     {tape_color: fita_color}
44 fim
45
46 def tamanho_do_pneu_padrão
47     '23'
48 fim
49 fim
50
51 classe MountainBike < Bicicleta 52
attr_reader :front_shock, :rear_shock
52
53
54 def post_initialize(args) @front_shock =
55     args[:front_shock] @rear_shock = args[:rear_shock]
56
57 fim
58
59 def local_spares
60     {choque_traseiro: choque_traseiro}
61 fim
62
```

```
63 def tamanho_do_pneu_padrão
64     '2.1'
65 fim
66 fim
```

RoadBike e MountainBike são mais legíveis agora que contêm apenas especializações. Fica claro à primeira vista o que eles fazem e é claro que são especializações de Bicicleta.

Novas subclasses precisam apenas implementar os métodos do modelo. Este último exemplo ilustra como é simples criar uma nova subclasse, mesmo para alguém não familiarizado com a aplicação. Aqui está a classe RecumbentBike, uma nova especialização da Bicicleta:

```
1 classe Bicicleta Reclinada < Bicicleta
2 attr_reader :flag
3
4 def post_initialize(args)
5     @flag = args[:flag]
6 fim
7
8 def local_spares
9     {bandeira: bandeira}
10 fim
11
12 def default_chain
13     "9 velocidades"
14 fim
15
16 def tamanho_do_pneu_padrão
17     '28'
18 fim
19 fim
20
21 dobrado = RecumbentBike.new(flag: 'alto e laranja')
22 peças sobressalentes dobradas
23 # -> {tamanho_do_pneu => "28", 24 # :corrente
24 #             => "9 velocidades",
25#             :bandeira      => "alto e laranja"}
```

O código na RecumbentBike é transparentemente óbvio e é tão regular e previsível que poderia ter saído de uma linha de montagem. Ele ilustra a força e o valor de herança; quando a hierarquia está correta, qualquer pessoa pode criar uma nova subclasse com êxito.

Resumo A

herança resolve o problema de tipos relacionados que compartilham muitos comportamentos comuns, mas diferem em alguma dimensão. Ele permite isolar código compartilhado e implementar algoritmos comuns em uma classe abstrata, ao mesmo tempo que fornece uma estrutura que permite que subclasses contribuam com especializações.

A melhor maneira de criar uma superclasse abstrata é enviar código a partir de subclasses concretas. Identificar a abstração correta é mais fácil se você tiver acesso a pelo menos três classes concretas existentes. O exemplo simples deste capítulo baseou-se em apenas dois, mas no mundo real é melhor esperar pelas informações adicionais fornecidas por três casos.

Superclasses abstratas usam o padrão de método template para convidar herdeiros a fornecer especializações e usam métodos de gancho para permitir que esses herdeiros contribuam com essas especializações sem serem forçados a enviar super. Os métodos hook permitem que as subclasses contribuam com especializações sem conhecer o algoritmo abstrato. Eles eliminam a necessidade de as subclasses enviarem super e, portanto, reduzem o acoplamento entre as camadas da hierarquia e aumentam sua tolerância a mudanças.

Hierarquias de herança bem projetadas são fáceis de estender com novas subclasses, mesmo para programadores que sabem muito pouco sobre a aplicação. Essa facilidade de extensão é o maior ponto forte da herança. Quando o seu problema é a necessidade de inúmeras especializações de uma abstração comum e estável, a herança pode ser uma solução de custo extremamente baixo.

Esta página foi intencionalmente deixada em branco

CAPÍTULO 7

Compartilhando comportamento de função com módulos

O capítulo anterior terminou com uma nota alta, com um código que parecia tão promissor que você pode estar se perguntando onde ele esteve durante toda a sua vida. No entanto, antes de decidir usar a herança clássica para resolver todos os problemas de design imagináveis, considere o seguinte: o que acontecerá quando a FastFeet desenvolver a necessidade de bicicletas de montanha reclinadas?

Se a solução para este novo problema de design parece ilusória, isso é perfeitamente compreensível. A criação de uma subclasse de mountain bike reclinada requer a combinação das qualidades de duas subclasses existentes, algo que a herança não pode acomodar prontamente. Ainda mais angustiante é o facto de esta falha ilustrar apenas *uma* das várias maneiras pelas quais a herança pode correr mal.

Para colher os benefícios do uso da herança, você deve entender não apenas como escrever código herdável, mas também quando faz sentido fazê-lo. O uso da herança clássica é sempre opcional; cada problema que ele resolve pode ser resolvido de outra maneira.

Como nenhuma técnica de design é gratuita, a criação da aplicação com melhor relação custo-benefício requer a realização de compensações informadas entre os custos relativos e os prováveis benefícios das alternativas.

Este capítulo explora uma alternativa que utiliza as técnicas de herança para compartilhar um *papel*. Ele começa com um exemplo que usa um módulo Ruby para definir uma função comum e depois dá conselhos práticos sobre como escrever todo o código herdável.

Compreendendo as funções Alguns

problemas exigem o compartilhamento de comportamento entre objetos que de outra forma não seriam relacionados. Este comportamento comum é ortogonal à classe; é um *papel* que um objeto desempenha. Muitas das funções necessárias para um aplicativo serão óbvias em tempo de design, mas também é comum descobrir funções imprevistas à medida que você escreve o código.

Quando objetos anteriormente não relacionados começam a desempenhar um papel comum, eles entram em um relacionamento com os objetos para os quais desempenham o papel. Esses relacionamentos não são tão visíveis quanto aqueles criados pelos requisitos de subclasse/superclasse da herança clássica, mas ainda assim existem. O uso de uma função cria dependências entre os objetos envolvidos e essas dependências introduzem riscos que você deve levar em consideração ao decidir entre as opções de design.

Esta seção revela uma função oculta e cria código para compartilhar seu comportamento entre todos os intervenientes, minimizando ao mesmo tempo as dependências assim incorridas.

Encontrando Funções

O tipo de pato Preparador do Capítulo 5, Reduzindo Custos com a Digitização de Patos, é uma função. Objetos que implementam a interface do Preparer desempenham essa função. Mecânico, TripCoordinator e Driver implementam `prepare_trip`; portanto, outros objetos podem interagir com eles como se fossem Preparadores, sem se preocupar com sua classe subjacente.

A existência de uma função de Preparador sugere que há também uma função paralela de Preparável (essas coisas geralmente vêm em pares). A classe Trip atua como um Preparable no exemplo do Capítulo 5; ele implementa a interface `Preparable`. Essa interface inclui todas as mensagens que qualquer Preparador pode esperar enviar para um Preparable, ou seja, os métodos bicicletas, clientes e veículo. O papel do Preparable não é muito óbvio porque Trip é seu único participante, mas é importante reconhecer que ele existe. O Capítulo 9, Projetando testes custo-efetivos, sugere técnicas para testar e documentar a função Preparável de modo a distingui-la da classe Trip .

Embora a função de Preparador tenha múltiplos atores, é tão simples que é inteiramente definida pela sua interface. Para desempenhar esse papel, tudo o que um objeto precisa fazer é implementar sua própria versão pessoal de `prepare_trip`. Objetos que atuam como Preparadores possuem apenas esta interface em comum. Eles compartilham a assinatura do método, mas nenhum outro código.

Preparador e Preparável são tipos de pato perfeitamente legítimos. É muito mais comum, entretanto, descobrir funções mais sofisticadas, aquelas em que a função requer não apenas assinaturas de mensagens específicas, mas também um comportamento específico. Quando uma função precisa de comportamento compartilhado, você se depara com o problema de organizar o código compartilhado. Idealmente

esse código seria definido em um único local, mas poderia ser utilizado por qualquer objeto que desejasse atuar como o tipo `pato` e desempenhar a função.

Muitas linguagens orientadas a objetos fornecem uma maneira de definir um grupo nomeado de métodos que são independentes de classe e podem ser misturados a qualquer objeto. Em Ruby, esses `mix-ins` são chamados de *módulos*. Os métodos podem ser definidos em um módulo e então o módulo pode ser adicionado a qualquer objeto. Os módulos fornecem, portanto, uma maneira perfeita de permitir que objetos de diferentes classes desempenhem uma função comum usando um único conjunto de código.

Quando um objeto inclui um módulo, os métodos nele definidos ficam disponíveis via delegação automática. Se isso soa como herança clássica, também parece, pelo menos do ponto de vista do objeto incluído. Do ponto de vista desse objeto, as mensagens chegam, ele não as entende, elas são roteadas automaticamente para algum outro lugar, a implementação correta do método é encontrada magicamente, é executada e a resposta é retornada.

Depois de começar a colocar código em módulos e adicionar módulos a objetos, você expande o conjunto de mensagens às quais um objeto pode responder e entra em um novo domínio de complexidade de design. Um objeto que implementa diretamente poucos métodos ainda pode ter um conjunto de respostas muito grande. O conjunto total de mensagens às quais um objeto pode responder inclui

- Aqueles que implementa
- Aqueles implementados em todos os objetos acima dele na hierarquia
- Aqueles implementados em qualquer módulo que tenha sido adicionado a ele
- Aqueles implementados em todos os módulos adicionados a qualquer objeto acima dele na hierarquia

Se este parece ser um conjunto de respostas assustadoramente grande e potencialmente confuso, você tem uma compreensão clara do problema. Adquirir uma compreensão do comportamento de uma hierarquia profundamente aninhada é, na melhor das hipóteses, intimidante e, na pior, impossível.

Organizando responsabilidades Agora que você

tem uma visão suficientemente sombria das possibilidades, é hora de analisar um exemplo administrável. Assim como acontece com a herança clássica, antes de escolher se deseja criar um tipo de `pato` e colocar um comportamento compartilhado em um módulo, você precisa saber como fazê-lo corretamente. Felizmente, o exemplo clássico de herança apresentado no Capítulo 6, Adquirindo comportamento por meio da herança, está prestes a dar frutos; este exemplo baseia-se nessas técnicas e é significativamente mais curto.

Considere o problema de programar uma viagem. As viagens ocorrem em momentos específicos e envolvem bicicletas, mecânicos e veículos motorizados. Bicicletas, mecânicos e veículos são

coisas reais no mundo físico que não podem estar em dois lugares ao mesmo tempo. O FastFeet precisa de uma maneira de organizar todos esses objetos em um cronograma para que possa determinar, para qualquer momento, quais objetos estão disponíveis e quais já estão confirmados.

Determinar se uma bicicleta, mecânico ou veículo não programado está disponível para participar de uma viagem não é tão simples quanto verificar se ele está ocioso durante o intervalo em que a viagem está programada. Essas coisas do mundo real precisam de um pouco de inatividade entre as viagens, não podem terminar uma viagem em um dia e começar outra no dia seguinte. Bicicletas e veículos motorizados devem passar por manutenção, e os mecânicos precisam de um descanso para não serem gentis com os clientes e de uma chance de lavar suas roupas.

As exigências são que as bicicletas tenham um intervalo mínimo de um dia entre viagens, os veículos um mínimo de três dias e os mecânicos quatro dias.

O código para escalonar esses objetos pode ser escrito de várias maneiras e, como aconteceu ao longo do livro, este exemplo evoluirá. Ele começa com um código bastante alarmante e segue até uma solução satisfatória, tudo no interesse de expor prováveis antipadrões.

Suponha que exista uma classe `Schedule`. Sua interface já inclui estes três métodos:

`agendado?(destino, início, fim)` add(destino, início, fim)
remover(destino, início, fim)

Cada um dos métodos acima leva três argumentos: o objeto de destino e as datas de início e término do período de interesse. O `Schedule` é responsável por saber se o seu argumento alvo de entrada já está agendado e por adicionar e remover alvos do agendamento. Estas responsabilidades pertencem justamente ao próprio Anexo .

Esses métodos são bons, mas infelizmente há uma lacuna neste código. É verdade que saber se um objeto está agendado durante algum intervalo é toda a informação necessária para evitar o agendamento excessivo de um objeto já ocupado. Porém, saber que um objeto *não* está escalonado durante um intervalo não é informação suficiente para saber se ele *pode* ser escalonado durante esse mesmo intervalo. Para determinar corretamente se um objeto pode ser agendado, algum objeto, em algum lugar, deve levar em conta o lead time.

A Figura 7.1 mostra uma implementação onde o próprio Cronograma assume a responsabilidade de saber o lead time correto. O `agendável?` O método conhece todos os valores possíveis e verifica a classe de seu argumento de destino recebido para decidir qual lead time usar.

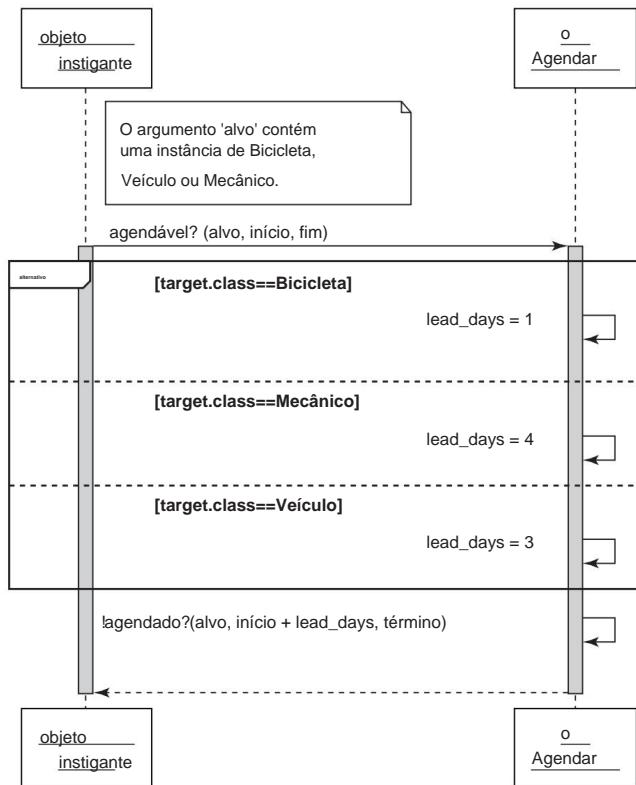


Figura 7.1 O cronograma conhece o lead time de outros objetos.

Você viu o padrão de verificar a classe para saber qual *mensagem* enviar; aqui a classe `Schedule` verifica para saber qual *valor* usar. Em ambos os casos, o `Schedule` sabe demais.

Este conhecimento não pertence ao `Schedule`, ele pertence às classes cujos nomes o `Schedule` está verificando.

Esta implementação clama por uma melhoria simples e óbvia, sugerida pelo padrão do código. Em vez de saber detalhes sobre outras turmas, a Agenda deveria enviar-lhes mensagens.

Removendo Dependências Desnecessárias

O fato de o `Schedule` verificar muitos nomes de classes para determinar qual valor colocar em uma variável sugere que o nome da variável deve ser transformado em uma mensagem, que por sua vez deve ser enviada para cada objeto recebido.

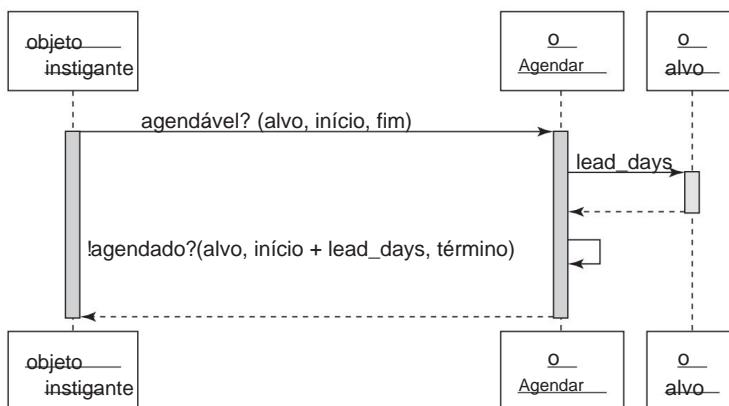


Figura 7.2 O cronograma espera que os alvos conheçam seu próprio lead time.

Descobrindo o tipo de pato programável A

Figura 7.2 mostra um diagrama de sequência para um novo código que remove a verificação da classe do programável? método e altera o método para enviar a mensagem lead_days para seu argumento de destino de entrada . Esta alteração substitui uma instrução if que verifica a classe de um objeto por uma mensagem enviada para esse mesmo objeto. Ele simplifica o código e transfere a responsabilidade de saber o número correto de dias de lead para o último objeto que poderia saber a resposta correta, que é exatamente onde essa responsabilidade pertence.

Uma análise mais detalhada da Figura 7.2 revela algo interessante. Observe que este diagrama contém uma caixa chamada “o alvo”. As caixas nos diagramas de seqüência destinam-se a representar objetos e são comumente nomeadas de acordo com classes, como em “o cronograma” ou “uma bicicleta”. Na Figura 7.2, o Schedule pretende enviar lead_days para seu alvo, mas o alvo pode ser uma instância de qualquer uma dentre diversas classes. Como a classe do alvo é desconhecida, não é óbvio como rotular a caixa para o destinatário desta mensagem.

A maneira mais fácil de desenhar o diagrama é contornar esse problema rotulando a caixa após o nome da variável e enviando a mensagem lead_days para esse “destino” sem ser preciso sobre sua classe. O Cronograma claramente não se preocupa com a classe do alvo ; em vez disso, apenas espera que ele responda a uma mensagem específica. Esta expectativa baseada em mensagens transcende a classe e expõe um papel desempenhado por todos os alvos e tornado explicitamente visível pelo diagrama de sequência.

O Cronograma espera que seu alvo se comporte como algo que entende lead_days, ou seja, como algo que é “agendável”. Você descobriu um tipo de pato.

No momento, esse novo tipo de pato tem um formato muito parecido com o tipo de pato Preparador do Capítulo 5; consiste apenas nesta interface. Os programáveis devem implementar `lead_days`, mas atualmente não têm outro código em comum.

Deixando os objetos falarem por si mesmos

Descobrir e usar esse tipo de pato melhora o código, removendo a dependência do Schedule de nomes de classes específicos, o que torna o aplicativo mais flexível e mais fácil de manter.

Entretanto, a Figura 7.2 ainda contém dependências desnecessárias que devem ser removidas.

É mais fácil ilustrar essas dependências com um exemplo extremo. Imagine uma classe `StringUtils` que implementa métodos utilitários para gerenciar strings. Você pode perguntar ao `StringUtils` se uma string está vazia enviando `StringUtils.empty?(some_string)`.

Se você escreveu muito código orientado a objetos, achará essa ideia ridícula.

Usar uma classe separada para gerenciar strings é evidentemente redundante; strings são objetos, têm comportamento próprio, gerenciam a si mesmas. Exigir que outros objetos conheçam um terceiro, `StringUtils`, para obter o comportamento de uma string complica o código ao adicionar uma dependência desnecessária.

Este exemplo específico ilustra a ideia geral de que os objetos devem gerenciar a si mesmos; eles devem conter seu próprio comportamento. Se o seu interesse estiver no objeto B, você não deveria ser forçado a saber sobre o objeto A se seu único uso for descobrir coisas sobre B.

O diagrama de sequência na Figura 7.2 viola esta regra. O instigador está tentando verificar se o objeto alvo é escalonável. Infelizmente, ele não faz essa pergunta ao alvo em si, mas sim a um terceiro, o `Schedule`. Perguntar ao `Schedule` se um destino é escalonável é como perguntar ao `StringUtils` se uma string está vazia. Obriga o instigador a conhecer e, portanto, a depender do Cronograma, mesmo que seu único interesse real esteja no alvo.

Assim como as strings respondem ao vazio? e podem falar por si, as metas devem responder ao programável?. O agendável? O método deve ser adicionado à interface da função Agendável .

Escrevendo o código concreto Da forma

como está atualmente, a função Agendável contém apenas uma interface. Adicionando o agendável? para esta função requer a escrita de algum código e não é imediatamente óbvio onde esse código deve residir. Você se depara com duas decisões; você deve decidir o que o código deve fazer e onde ele deve ficar.

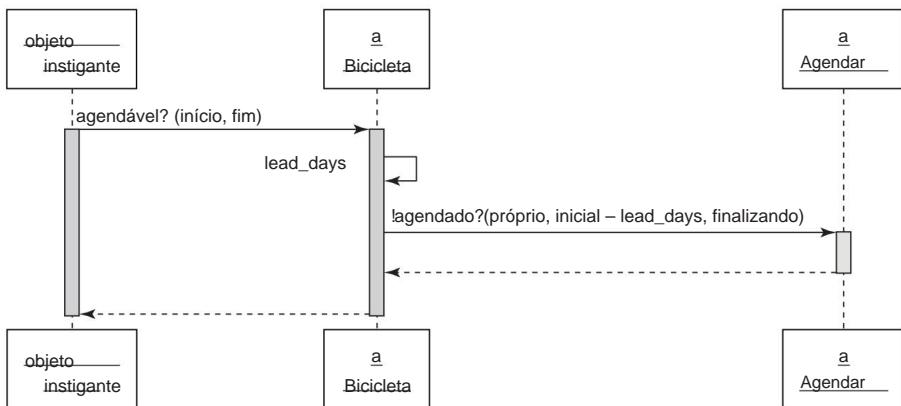


Figura 7.3 As aulas de bicicleta sabem se podem ser agendadas.

A maneira mais simples de começar é separar as duas decisões. Escolha uma classe concreta arbitrária (por exemplo, Bicicleta) e implemente o programável? método diretamente nessa classe. Depois de ter uma versão que funcione para Bicycle, você poderá refatorar seu caminho para um arranjo de código que permita que todos os Schedulables compartilhem o comportamento.

A Figura 7.3 mostra um diagrama de sequência onde esse novo código está em Bicycle. A bicicleta agora responde a mensagens sobre sua própria “programabilidade”.

Antes dessa mudança, todo objeto instigante precisava conhecer e, portanto, dependia do Cronograma. Essa mudança permite que as bicicletas falem por si, liberando objetos instigantes para interagir com elas sem o auxílio de terceiros.

O código para implementar este diagrama de sequência é simples. Aqui está uma programação muito simples. Claramente, esta não é uma implementação digna de produção, mas fornece um substituto suficientemente bom para o resto do exemplo.

Horário de 1 aula

```

2 def agendado?(agendável, data_inicial, data_final) coloca "Esta #{schedulable.class}
3   " "não está agendada\n" + " entre #{data_inicial} e      +
4     #{data_final}""
5
6   falso
7 fim
8 fim
  
```

O próximo exemplo mostra a implementação do programável? pela Bicycle . A bicicleta conhece seu próprio prazo de agendamento (definido na linha 23 e referenciado na linha 13 abaixo) e os delegados agendados? ao próprio Cronograma .

```
1 classe Bicicleta 2
2 attr_reader :schedule, :size, :chain, :tire_size
3
4     # Inje o Cronograma e forneça um padrão
5 def inicializar(args={}) @schedule =
6     args[:schedule] || Agenda.new # ...
7
8 fim
9
10 # Retorna verdadeiro se esta bicicleta estiver disponível 11 # durante este intervalo
11 # (agora específico da bicicleta). 12 def programável?(data_inicial, data_final)
12
13     !agendado?(data_inicial - dias_de_lead, data_final)
14 fim
15
16 # Retorne a resposta do cronograma
17 def agendado?(data_inicial, data_final) agenda.programado?
18     (self, data_inicial, data_final)
19 fim
20
21 # Retorna o número de lead_days antes que uma bicicleta 22 # possa ser agendada.
22
23 dias_de_lead_def
24     1
25 fim
26
27 #...
28 fim
29
30 requer 'data' 31 começando
31 = Date.parse("2015/09/04") 32 terminando = Date.parse("2015/09/10")
32
33
34 b = Bicicleta.nova 35
35 b.agendável?(início, fim)
36 # Esta Bicicleta não está programada
37 # entre 03/09/2015 e 10/09/2015
38 # => verdadeiro
```

A execução do código (linhas 30 a 35) confirma que a Bicycle ajustou corretamente a data de início para incluir os dias de antecedência específicos da bicicleta.

Este código esconde o conhecimento de quem é o Horário e o que o Horário faz dentro da Bicicleta. Os objetos que seguram uma bicicleta não precisam mais saber da existência ou do comportamento do cronograma.

Extraindo a Abstração O código acima resolve

a primeira parte do problema atual na medida em que decide qual é o agendável? método deveria servir, mas Bicicleta não é o único tipo de coisa “agendável”. Mecânico e Veículo também desempenham esse papel e por isso necessitam desse comportamento. É hora de reorganizar o código para que ele possa ser compartilhado entre objetos de classes diferentes.

O exemplo a seguir mostra um novo módulo Schedulable , que contém uma abstração extraída da classe Bicycle acima. O agendável? (linha 8) e agendado? (linha 12) os métodos são cópias exatas daqueles anteriormente implementados em Bicicleta.

```
1 módulo Agendável
2 attr_writer :agendar
3
4 cronograma definido
5     @schedule ||= ::Schedule.new
6 fim
7
8 def programável?(data_inicial, data_final)
9     !agendado?(data_inicial - dias_de lead, data_final)
10 fim
11
12 def agendado?(data_inicial, data_final) agendado.agendado?
13     (self, data_inicial, data_final)
14 fim
15
16 # incluers podem substituir 17 def
lead_days
18     0
19 fim
20
21 fim
```

Duas coisas mudaram no código que existia anteriormente em Bicycle. Primeiro, um método de agendamento (linha 4) foi adicionado. Este método retorna uma instância do cronograma geral.

De volta à Figura 7.2, o objeto instigante dependia do Cronograma, o que significava que poderia haver muitos locais na aplicação que precisavam do conhecimento do Cronograma. Na iteração seguinte, Figura 7.3, essa dependência foi transferida para Bicycle, reduzindo seu alcance na aplicação. Agora, no código acima, a dependência de Schedule foi removida de Bicycle e movida para o módulo Schedulable , isolando-o ainda mais.

A segunda alteração é no método lead_days (linha 17). A implementação anterior de Bicycle retornava um número específico de bicicleta, a implementação do módulo agora retorna um padrão mais genérico de zero dias.

Mesmo que não houvesse um padrão razoável do aplicativo para dias de entrega, o módulo Agendável ainda deverá implementar o método lead_days . As regras para módulos são as mesmas da herança clássica. Se um módulo enviar uma mensagem, ele deverá fornecer uma implementação, mesmo que essa implementação apenas gere um erro indicando que os usuários do módulo devem implementar o método.

Incluir este novo módulo na classe Bicycle original , conforme mostrado no exemplo abaixo, adiciona os métodos do módulo ao conjunto de respostas de Bicycle . O método lead_days é um gancho que segue o padrão do método template. Bicycle substitui este gancho (linha 4) para fornecer uma especialização.

A execução do código revela que Bicycle mantém o mesmo comportamento de quando implementou diretamente essa função.

Bicicleta de 1 classe

```

2 inclui programável
3
4 dias de lead_def
5     1
6 fim
7
8     ...
9 fim
10
11 requer 'data' 12 inicio =
Date.parse("2015/09/04") 13 final = Date.parse("2015/09/10")

```

```

15 b = Bicicleta.novo 16
b.agendável?(início, fim)
17# Esta Bicicleta não está programada
18#      entre 03/09/2015 e 10/09/2015
19# => verdadeiro
20

```

Mover os métodos para o módulo `Schedulable`, incluindo o módulo e substituir `lead_days`, permite que `Bicycle` continue a se comportar corretamente. Além disso, agora que você criou este módulo, outros objetos podem utilizá-lo para se tornarem Agendáveis. Eles podem desempenhar essa função sem duplicar o código.

O padrão de mensagens mudou daquele de envio `programável?` para uma `Bicicleta` para envio `agendável?` para um `Agendável`. Agora você está comprometido com o tipo `pato` e o diagrama de sequência mostrado na Figura 7.3 pode ser alterado para se parecer com o da Figura 7.4.

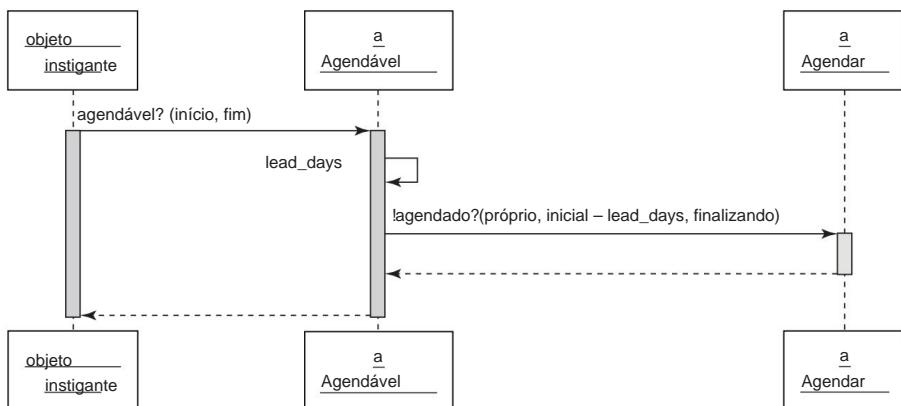


Figura 7.4 O tipo pato programável.

Uma vez incluído este módulo em todas as classes que podem ser escalonadas, o padrão de código se torna fortemente reminiscente de herança. O exemplo a seguir mostra `Veículo` e `Mecânico` incluindo o módulo `Agendável` e respondendo ao `programável?` mensagem.

Veículo de 1 classe

2 inclui **programável**

3

4 dias de **lead_def**

5 3

```

6 fim
7
8 #...
9 fim
10
11 classe Mecânico
12 incluem programável
13
14 dias de lead_def
15      4
16 fim
17
18 #...
19 fim
20
21 v = Veiculo.novo
22 v.agendável?(início, fim)
23 # Este Veículo não está programado
24 # entre 01/09/2015 e 10/09/2015
25 # => verdadeiro
26
27 m = Mecânico.novo
28 m.agendável?(início, fim)
29 # Este Mecânico não está agendado 30 # entre 29/02/2015 e
10/09/2015
31 # => verdadeiro

```

O código em Schedulable é a abstração e usa o padrão de método template para convidar objetos a fornecer especializações ao algoritmo que ele fornece. Os programáveis substituem lead_days para fornecer essas especializações. Quando programável? chega a qualquer Schedulable, a mensagem é automaticamente delegada ao método definido no módulo.

Isso pode não se enquadrar na definição estrita de herança clássica, mas em termos de como o código deve ser escrito e como as mensagens são resolvidas, certamente funciona como tal.

As técnicas de codificação são as mesmas porque a pesquisa do método segue o mesmo caminho.

Este capítulo teve o cuidado de manter uma distinção entre herança clássica e compartilhamento de código por meio de módulos. Essa diferença é versus *se comporta como uma* diferença definitivamente importa, cada escolha tem consequências distintas. No entanto, as técnicas de codificação para estas duas coisas são muito semelhantes e esta semelhança existe porque ambas as técnicas dependem da delegação automática de mensagens.

Métodos de pesquisa

Compreender as semelhanças entre herança clássica e inclusão de módulo é mais fácil se você entender como as linguagens orientadas a objetos, em geral, e Ruby, em particular, encontram a implementação do método que corresponde ao envio de uma mensagem.

Uma simplificação grosseira

Quando um objeto recebe uma mensagem, a linguagem OO primeiro procura na *classe* desse objeto uma implementação de método correspondente. Isso faz todo o sentido; caso contrário, as definições de método precisariam ser duplicadas em cada instância de cada classe. Armazenar os métodos conhecidos de um objeto dentro de sua classe significa que todas as instâncias de uma classe podem compartilhar o mesmo conjunto de definições de métodos; definições que precisam então existir em apenas um lugar.

Ao longo deste livro houve pouca preocupação em declarar explicitamente se o objeto em discussão é uma instância de uma classe ou a própria classe, esperando que a intenção fique clara no contexto e que você se sinta confortável com a noção de que as próprias classes são objetos por si só. Descrever como funciona a pesquisa de métodos exigirá um pouco mais de precisão.

Conforme dito acima, a busca por um método começa na classe do objeto receptor. Se esta classe não implementar a mensagem, a busca prossegue para sua superclasse. A partir daqui, apenas as superclasses importam, a busca prossegue subindo na cadeia de superclasses, procurando uma superclasse após a outra, até atingir o topo da hierarquia.

A Figura 7.5 mostra como uma linguagem genérica orientada a objetos procuraria o método *spares* da hierarquia Bicycle que você criou no Capítulo 6. Para os propósitos desta discussão, a classe Object fica no topo da hierarquia. Observe

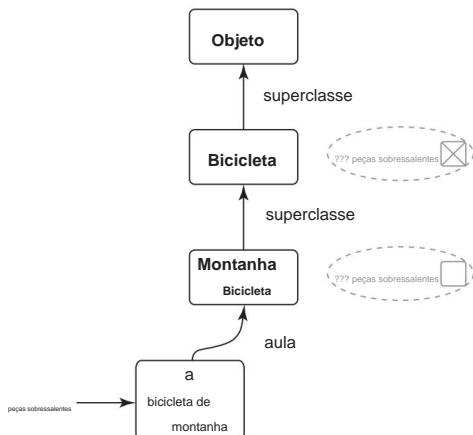


Figura 7.5 Uma generalização da pesquisa de métodos.

que as especificidades da pesquisa de métodos em Ruby serão mais envolventes, mas este é um primeiro modelo razoável.

Na Figura 7.5, a mensagem de peças sobressalentes é enviada para uma *instância* do MountainBike. A linguagem OO primeiro procura um método de peças de reposição correspondente na classe MountainBike . Ao não encontrar métodos sobressalentes nessa classe, a busca prossegue para a superclasse de MountainBike , Bicycle.

Como a Bicycle implementa peças sobressalentes, a pesquisa deste exemplo termina aqui. Porém, no caso em que não existe implementação de superclasse, a busca prossegue de uma superclasse para a próxima até atingir o topo da hierarquia e busca em Object.

Se todas as tentativas de encontrar um método adequado falharem, você pode esperar que a pesquisa pare, mas muitos idiomas fazem uma segunda tentativa para resolver a mensagem.

Ruby dá ao receptor original uma segunda chance, enviando-lhe uma nova mensagem, method_missing, e passando :spares como argumento. As tentativas de resolver esta nova mensagem reiniciam a pesquisa no mesmo caminho, exceto que agora a pesquisa é por method_missing em vez de peças de reposição.

Uma explicação mais precisa A seção

anterior explica apenas como os métodos são procurados para a herança clássica. A próxima seção expande a explicação para abranger métodos definidos em um módulo Ruby. A Figura 7.6 adiciona o módulo Schedulable ao caminho de pesquisa do método.

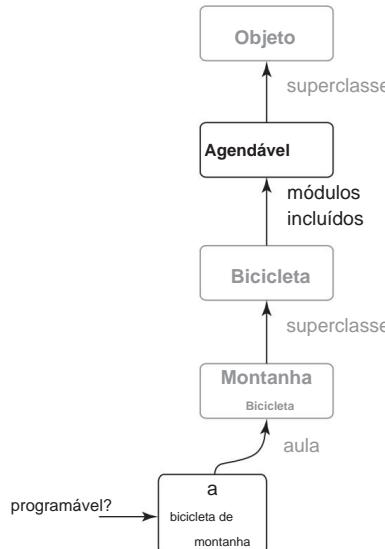


Figura 7.6 Uma explicação mais precisa da pesquisa de métodos.

A hierarquia de objetos na Figura 7.6 se parece muito com a da Figura 7.5. Ela difere apenas porque a Figura 7.6 mostra o módulo `Schedulable` destacado entre as classes `Bicycle` e `Object`.

Quando `Bicycle` inclui `Schedulable`, todos os métodos definidos no módulo tornam-se parte do conjunto de respostas de `Bicycle`. Os métodos do módulo vão para o caminho de pesquisa de métodos diretamente *acima* dos métodos definidos em `Bicycle`. A inclusão deste módulo não altera a superclasse de `Bicycle` (que ainda é `Object`), mas no que diz respeito à pesquisa de método, pode muito bem ter mudado. Qualquer mensagem recebida por uma instância de `MountainBike` agora tem chance de ser satisfeita por um método definido no módulo `Schedulable`.

Isto tem enormes implicações. Se `Bicycle` implementar um método que também é definido em `Schedulable`, a implementação de `Bicycle` substituirá a de `Schedulable`. Se `Schedulable` enviar métodos que não implementa, as instâncias de `MountainBike` poderão encontrar falhas confusas.

A Figura 7.6 mostra o programável? mensagem sendo enviada para uma instância do `MountainBike`. Para resolver esta mensagem, Ruby primeiro procura um método correspondente na classe `MountainBike`. A pesquisa então prossegue ao longo do caminho de pesquisa do método, que agora contém módulos e também superclasses. Uma implementação de agendável? é eventualmente encontrado em `Schedulable`, que fica no caminho de pesquisa entre `Bicycle` e `Object`.

Uma explicação quase completa Agora que você viu como os módulos se ajustam ao caminho de pesquisa do método, é hora de complicar ainda mais o quadro.

É perfeitamente possível que uma hierarquia contenha uma longa cadeia de superclasses, cada uma delas incluindo muitos módulos. Quando uma única classe inclui vários módulos diferentes, os módulos são colocados no caminho de pesquisa do método na ordem *inversa* da inclusão do módulo. Assim, os métodos do último módulo incluído são encontrados primeiro no caminho de pesquisa.

Esta discussão tem sido, até agora, sobre a inclusão de módulos em classes através da palavra-chave `include` do Ruby. Como você já viu, incluir um módulo em uma classe adiciona os métodos do módulo ao conjunto de respostas para todas as instâncias dessa classe. Por exemplo, na Figura 7.6 o módulo `Schedulable` foi incluído na classe `Bicycle` e, como resultado, instâncias de `MountainBike` ganham acesso aos métodos nele definidos.

No entanto, também é possível adicionar métodos de um módulo a um único objeto, usando a palavra-chave `extend` do Ruby. Como estender adiciona o comportamento do módulo diretamente a um objeto, estender uma classe com um módulo cria métodos de classe *nessa classe* e estender uma instância de uma classe com um módulo cria métodos de instância *nessa instância*. Esses

duas coisas são exatamente iguais; Afinal, as classes são apenas objetos antigos e a extensão se comporta da mesma forma para todos.

Finalmente, qualquer objeto também pode ter métodos ad hoc adicionados diretamente ao seu objeto pessoal. “Aula individual.” Esses métodos ad hoc são exclusivos para esse objeto específico.

Cada uma dessas alternativas aumenta o conjunto de respostas de um objeto, colocando definições de método em locais específicos e inequívocos ao longo do caminho de pesquisa do método. A Figura 7.7 ilustra a lista completa de possibilidades.

Antes de continuar, aqui vai uma palavra de advertência. A Figura 7.7 é precisa o suficiente para orientar o comportamento da maioria dos projetistas, mas não é a história completa. Para a maioria das aplicações

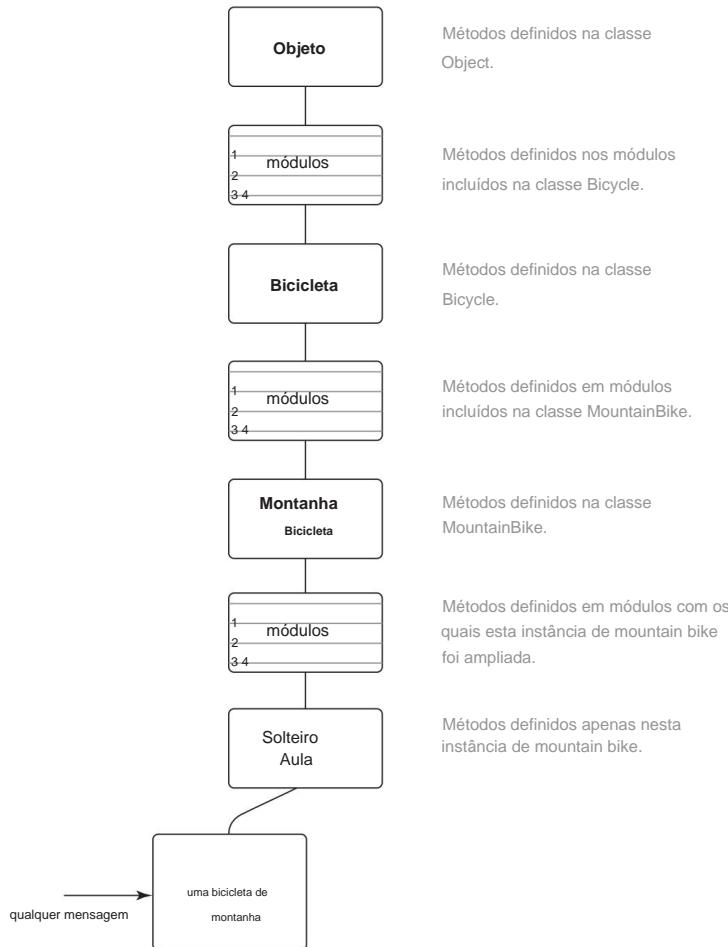


Figura 7.7 Uma explicação quase completa da pesquisa de métodos.

código, é perfeitamente adequado se comportar como se a classe Object estivesse no topo da hierarquia, mas, dependendo da sua versão do Ruby, isso pode não ser tecnicamente verdade. Se você estiver escrevendo um código para o qual acha que esse problema pode ser importante, certifique-se de compreender a hierarquia de objetos do Ruby em questão.

Herdando o comportamento do papel

Agora que você viu como definir o código compartilhado de um papel em um módulo e como o código de um módulo é inserido no caminho de pesquisa do método, você está preparado para escrever um código realmente assustador. Imagine as possibilidades. Você pode escrever módulos que incluem outros módulos. Você pode escrever módulos que substituem os métodos definidos em outros módulos. Você pode criar hierarquias de herança de classe profundamente aninhadas e depois incluir esses vários módulos em diferentes níveis da hierarquia.

Você pode escrever código impossível de entender, depurar ou estender.

Isso é algo poderoso e perigoso em mãos não treinadas. No entanto, como esse mesmo poder é o que permite criar estruturas simples de objetos relacionados que atendam elegantemente às necessidades de sua aplicação, sua tarefa não é evitar essas técnicas, mas aprender a usá-las pelos motivos certos, nos lugares certos., da maneira correta.

A primeira etapa nesse caminho é escrever um código herdável adequadamente.

Escrevendo código herdável

A utilidade e a capacidade de manutenção das hierarquias e módulos de herança são diretamente proporcionais à qualidade do código. Mais do que com outras estratégias de design, o compartilhamento do comportamento herdado requer técnicas de codificação muito específicas, que serão abordadas nas seções a seguir.

Reconheça os antipadrões Existem dois

antipadrões que indicam que seu código pode se beneficiar da herança.

Primeiro, um objeto que usa uma variável com um nome como tipo ou categoria para determinar qual mensagem enviar para si mesmo contém dois tipos altamente relacionados, mas ligeiramente diferentes. Este é um pesadelo de manutenção; o código deve mudar sempre que um novo tipo é adicionado. Código como esse pode ser reorganizado para usar herança clássica, colocando o código comum em uma superclasse abstrata e criando subclasses para os diferentes tipos. Esse rearranjo permite criar novos subtipos adicionando novas subclasses.

Essas subclasses estendem a hierarquia sem alterar o código existente.

Segundo, quando um objeto emissor verifica a classe de um objeto receptor para determinar qual mensagem enviar, você negligenciou um tipo pato. Este é outro pesadelo de manutenção; o código deve mudar toda vez que você introduz uma nova classe de receptor. Nesta situação todos os possíveis objetos receptores desempenham um papel comum.

Esta função deve ser codificada como um tipo pato e os receptores devem implementar a interface do tipo pato. Assim que o fizerem, o objeto original poderá enviar uma única mensagem para cada receptor, confiante de que, como cada receptor desempenha o papel, compreenderá a mensagem comum.

Além de compartilhar uma interface, os tipos pato também podem compartilhar comportamentos. Quando isso acontecer, coloque o código compartilhado em um módulo e inclua esse módulo em cada classe ou objeto que desempenha a função.

Insista na Abstração

Todo o código em uma superclasse abstrata deve ser aplicado a todas as classes que a herdam.

As superclasses não devem conter código que se aplique a algumas subclasses, mas não a todas.

Esta restrição também se aplica aos módulos: o código de um módulo deve ser aplicado a todos que o utilizam.

Abstrações defeituosas fazem com que objetos herdados contenham comportamento incorreto; tentativas de contornar esse comportamento errôneo farão com que seu código se deteriore. Ao interagir com esses objetos estranhos, os programadores são forçados a conhecer suas peculiaridades e a identificar dependências que devem ser evitadas.

Subclasses que substituem um método para gerar uma exceção como “não implementa” são um sintoma desse problema. Embora seja verdade que a conveniência compensa tudo e que às vezes é mais econômico organizar o código dessa maneira, você deve estar relutante em fazê-lo. Quando subclasses substituem um método para declarar que *não fazem aquilo*, elas chegam perigosamente perto de declarar que *não são aquilo*. Nada bom pode vir disso.

Se você não conseguir identificar corretamente a abstração, pode não haver uma, e se não existir nenhuma abstração comum, a herança não será a solução para o seu problema de design.

Honrar o contrato

As subclasses concordam com um *contrato*; eles prometem ser substituíveis por suas superclasses.

A substituibilidade só é possível quando os objetos se comportam conforme o esperado e se espera que as subclasses estejam em conformidade com a interface de sua superclasse. Eles devem responder a todas as mensagens nessa interface, recebendo os mesmos tipos de entradas e retornando os mesmos tipos de dados.

saídas. Não lhes é permitido fazer nada que obrigue os outros a verificar o seu tipo para saber como tratá-los ou o que esperar deles.

Onde as superclasses impõem restrições aos argumentos de entrada e aos valores de retorno, as subclasses podem se permitir um pouco de liberdade sem violar seu contrato.

As subclasses podem aceitar parâmetros de entrada com restrições mais amplas e retornar resultados com restrições mais restritas, ao mesmo tempo que permanecem perfeitamente substituíveis por suas superclasses.

Subclasses que não cumprem seu contrato são difíceis de usar. Eles são “especiais” e não podem ser substituídos livremente por suas superclasses. Essas subclasses estão declarando que não são realmente um *tipo* de sua superclasse e lançam dúvidas sobre a correção de toda a hierarquia.

Princípio de Substituição de Liskov (LSP)

Ao honrar o contrato, você está seguindo o Princípio de Substituição de Liskov, que leva o nome de sua criadora, Barbara Liskov, e fornece o “L” nos princípios de design SOLID.

Seu princípio afirma:

Seja $q(x)$ uma propriedade demonstrável sobre objetos x do tipo T . Então $q(y)$ deve ser verdadeiro para objetos y do tipo S onde S é um subtipo de T .

Os matemáticos compreenderão instantaneamente esta afirmação; todos os outros deveriam entender que, para que um sistema de tipos seja sensato, os subtipos devem ser substituíveis por seus supertipos.

Seguir esse princípio cria aplicativos onde uma subclasse pode ser usada em qualquer lugar que sua superclasse serviria, e onde objetos que incluem módulos podem ser confiáveis para desempenhar de forma intercambiável a função do módulo.

Use o padrão de método de modelo A técnica de

codificação fundamental para criar código herdável é o padrão de método de modelo. Esse padrão é o que permite separar o abstrato do concreto. O código abstrato define os algoritmos e os herdeiros concretos dessa abstração contribuem com especializações, substituindo esses métodos de modelo.

Os métodos de modelo representam as partes do algoritmo que variam e criá-los força você a tomar decisões explícitas sobre o que varia e o que não varia.

Desacoplar aulas preventivamente

Evite escrever código que exija que seus herdeiros enviem super; em vez disso, use mensagens de gancho para permitir que as subclasses participem enquanto as isenta da responsabilidade por conhecendo o algoritmo abstrato. A herança, por sua própria natureza, acrescenta dependências poderosas à estrutura e à organização do código. Escrever código que requer subclasses para enviar super adiciona uma dependência adicional; evite isso se puder.

Os métodos de gancho resolvem o problema de envio de super, mas, infelizmente, apenas para níveis adjacentes da hierarquia. Por exemplo, no Capítulo 6, Bicicleta enviou gancho método local_spares que MountainBike substituiu para fornecer especializações. Esse método hook serve ao seu propósito admiravelmente, mas o problema original reaparece se você adicione outro nível à hierarquia criando a subclasse MonsterMountainBike em Bicicleta de montanha. Para combinar suas próprias peças de reposição com as de seus pais, MonsterMountainBike seria forçado a substituir local_spares e, dentro dele, mande super.

Crie hierarquias superficiais

As limitações dos métodos de gancho são apenas uma das muitas razões para criar hierarquias.

Toda hierarquia pode ser pensada como uma pirâmide que tem profundidade e largura. Um a profundidade do objeto é o número de superclasses entre ele e o topo. Sua largura é a número de suas subclasses diretas. A forma de uma hierarquia é definida por sua amplitude geral e profundidade e é esse formato que determina facilidade de uso, manutenção e extensão. A Figura 7.8 ilustra algumas das possíveis variações de forma.

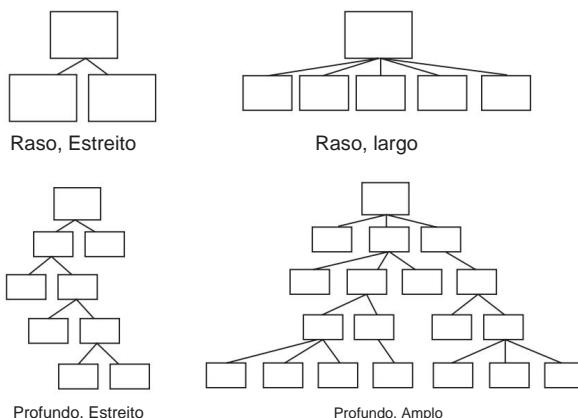


Figura 7.8 As hierarquias têm diferentes formatos.

Hierarquias superficiais e estreitas são fáceis de entender. Hierarquias superficiais e amplas são um pouco mais complicadas. Hierarquias profundas e estreitas são um pouco mais desafiadoras e, infelizmente, têm uma tendência natural de se tornarem mais amplas, estritamente como um efeito colateral de sua profundidade. Hierarquias amplas e profundas são difíceis de entender, caras de manter e devem ser evitadas.

O problema com hierarquias profundas é que elas definem um caminho de pesquisa muito longo para resolução de mensagens e fornecem inúmeras oportunidades para objetos nesse caminho adicionarem comportamento à medida que a mensagem passa. Como os objetos dependem de *tudo* que está acima deles, uma hierarquia profunda possui um grande conjunto de dependências integradas, cada uma das quais poderá mudar algum dia.

Outro problema com hierarquias profundas é que os programadores tendem a estar familiarizados apenas com as classes superiores e inferiores; isto é, eles tendem a compreender apenas o comportamento implementado nos limites do caminho de busca. As classes intermediárias recebem pouca atenção. Mudanças nessas classes médias vagamente compreendidas têm maiores chances de introduzir erros.

Resumo

Quando objetos que desempenham uma função comum precisam compartilhar comportamento, eles o fazem através de um módulo Ruby. O código definido em um módulo pode ser adicionado a qualquer objeto, seja uma instância de uma classe, a própria classe ou outro módulo.

Quando uma classe inclui um módulo, os métodos desse módulo são colocados no mesmo caminho de pesquisa que os métodos adquiridos por herança. Como os métodos de módulo e os métodos herdados se intercalam no caminho de pesquisa, as técnicas de codificação para módulos refletem aquelas de herança. Os módulos, portanto, devem usar o padrão de método template para convidar aqueles que os incluem a fornecer especializações, e devem implementar métodos de gancho para evitar forçar os incluidos a enviar super (e, assim, conhecer o algoritmo).

Quando um objeto adquire um comportamento que foi definido em outro lugar, independentemente de esse *outro lugar* ser uma superclasse ou um módulo incluído, o objeto adquirente assume o compromisso de honrar um contrato implícito. Este contrato é definido pelo Princípio de Substituição de Liskov, que em termos matemáticos diz que um subtipo deve ser substituível por seu supertipo, e em termos Ruby isso significa que um objeto deve agir como aquilo que afirma ser.

CAPÍTULO 8

Combinando objetos com composição

Composição é o ato de combinar partes distintas em um todo complexo, de modo que o todo se torne mais do que a soma de suas partes. A música, por exemplo, é composta.

Você pode não pensar no seu software como música, mas a analogia é adequada. A partitura musical da Quinta Sinfonia de Beethoven é uma longa lista de notas distintas e independentes. Você precisa ouvi-los apenas uma vez para entender que, embora *contenha* as notas, *não* são as notas. É algo mais.

Você pode criar software da mesma maneira, usando composição orientada a objetos para combinar objetos simples e independentes em conjuntos maiores e mais complexos. Na composição, o objeto maior está conectado às suas partes através de uma relação *tem-um*. Uma bicicleta tem peças. A bicicleta é o objeto que contém, as peças estão contidas dentro de uma bicicleta. Inerente à definição de composição está a ideia de que uma bicicleta não apenas possui peças, mas também se comunica com elas por meio de uma interface. A peça é um *papel* e as bicicletas ficam felizes em colaborar com qualquer objeto que desempenhe o papel.

Este capítulo ensina as técnicas de composição OO. Começa com um exemplo, passa para uma discussão dos pontos fortes e fracos relativos da composição e da herança e, em seguida, conclui com recomendações sobre como escolher entre técnicas alternativas de design.

Compondo uma bicicleta de peças

Esta seção começa onde o exemplo da Bicicleta no Capítulo 6, Adquirindo Comportamento Através da herança, terminou. Se esse código não estiver mais em sua mente, é vale a pena voltar ao final do Capítulo 6 e refrescar sua memória. Esta seção pega esse exemplo e o move através de várias refatorações, substituindo gradualmente herança com composição.

Atualizando a classe de bicicleta

A classe Bicycle é atualmente uma superclasse abstrata em uma hierarquia de herança e você gostaria de convertê-lo para usar composição. O primeiro passo é ignorar o código existente e pense em como uma bicicleta deve ser composta.

A classe Bicycle é responsável por responder à mensagem de peças sobressalentes . Esse A mensagem sobressalentes deve retornar uma lista de peças sobressalentes. As bicicletas têm peças, as peças da bicicleta o relacionamento parece naturalmente uma composição. Se você criou um objeto para conter todos peças de uma bicicleta, você poderia delegar a mensagem de peças sobressalentes a esse novo objeto.

É razoável nomear esta nova classe como Parts. O objeto Parts pode ser responsável para manter uma lista das peças da bicicleta e saber quais dessas peças precisam de peças sobressalentes. Observe que este objeto representa uma coleção de peças, não uma única peça.

O diagrama de sequência na Figura 8.1 ilustra essa ideia. Aqui, uma bicicleta manda a mensagem sobressalentes para seu objeto Parts .

Toda bicicleta precisa de um objeto Parts ; parte do que significa ser uma bicicleta é *tenho* peças. O diagrama de classes da Figura 8.2 ilustra esse relacionamento.

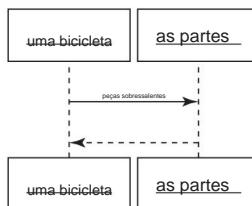


Figura 8.1A Bicicleta pergunta Peças para peças sobressalentes*



Figura 8.2A Bicicleta tem um Peças .

Este diagrama mostra as classes Bicicleta e Peças conectadas por uma linha. A linha atribui à bicicleta com um diamante negro; este diamante negro indica *composição*, significa que uma bicicleta é composta de peças. O lado Peças da linha tem o número 1." Isso significa que há apenas um objeto Parts por bicicleta.

É fácil converter a classe Bicycle existente para este novo design. Remova a maior parte seu código, adicione uma variável parts para conter o objeto Parts e delegue peças sobressalentes para peças. Aqui está a nova classe Bicicleta .

Bicicleta de 1 classe

```

2 attr_reader :tamanho, :partes
3
4 def inicializar(args={})
5   @tamanho      = args[:tamanho]
6   @peças        = args[:partes]
7 fim
8
9 peças sobressalentes def
10    peças.sobressalentes
11 fim
12 fim

```

A bicicleta é agora responsável por três coisas: saber o seu tamanho, segurar o seu Peças e respondendo às suas peças sobressalentes.

Criando uma hierarquia de peças

Isso foi fácil, mas apenas porque não havia muito comportamento relacionado com bicicletas no Aula de bicicleta para começar; a maior parte do código em Bicycle tratava de peças. Você ainda precisa do comportamento das peças que você acabou de remover da bicicleta e da maneira mais simples de fazer com que esse código funcione novamente é simplesmente lançá-lo em uma nova hierarquia de partes, como mostrado abaixo.

1 classe Peças

```

2 attr_reader :chain, :tire_size
3
4 def inicializar(args={})
5   @corrente      = args[:chain] @tire_size      || cadeia_padrão
6   = args[:tire_size] || tamanho_do_pneu_padrão
7   post_initialize(args)
8 fim

```

```
9
10 peças sobressalentes
11     def {tamanho_do_pneu:tamanho_do_pneu,
12         corrente:       cadeia}.merge(local_spares)
13 fim
14
15 def default_tire_size aumenta
16     NotImplemented
17 fim
18
19 # subclasses podem substituir 20 def post_initialize(args)
20 nil
21
22 fim
23
24 def local_spares
25     {}
26 fim
27
28 def default_chain '10 velocidades'
29
30 fim
31 fim
32
33 classe RoadBikeParts < Peças
34 attr_reader :tape_color
35
36 def post_initialize(args) @tape_color =
37     args[:tape_color]
38 fim
39
40 def local_spares
41     {tape_color: fita_color}
42 fim
43
44 def tamanho_do_pneu_padrão
45     '23'
46 fim
47 fim
48
49 classes MountainBikeParts < Peças
50 attr_reader :front_shock, :rear_shock
51
```

```

52 def post_initialize(args)
53     @front_shock = args[:front_shock]
54     @rear_shock = args[:rear_shock]
55 fim
56
57 def local_spares
58     {choque_traseiro: choque_traseiro}
59 fim
60
61 def tamanho_do_pneu_padrão
62     '2.1'
63 fim
64 fim

```

Este código é uma cópia quase exata da hierarquia Bicycle do Capítulo 6; as diferenças são que as classes foram renomeadas e a variável size foi removido.

O diagrama de classes na Figura 8.3 ilustra essa transição. Existe agora um classe de peças abstratas . A bicicleta é composta de peças. Parts tem duas subclasses, Peças de RoadBike e Peças de MountainBike.

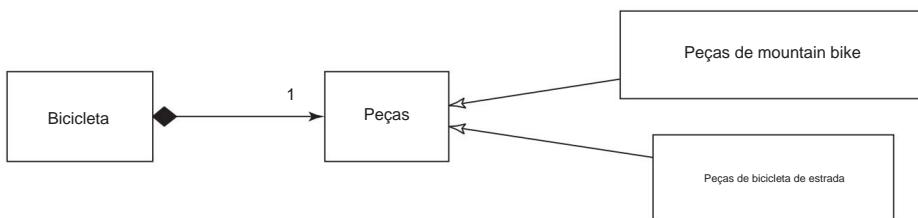


Figura 8.3 Uma hierarquia de Peças

Após essa refatoração, tudo ainda funciona. Como você pode ver abaixo, independentemente de quer tenha RoadBikeParts ou MountainBikeParts, uma bicicleta ainda pode corretamente responda seu tamanho e peças sobressalentes.

```

1 road_bike = 2
Bicicleta.new(
3     tamanho: 'L',
4     peças: RoadBikeParts.new(tape_color: 'red'))
5
6 road_bike.tamanho          # -> 'EU'
7

```

```

8 road_bike.spares 9 # ->
{:tire_size=>"23", :chain=>"10
10 #           velocidades", :tape_color=>"red"}
11 #
12
13 mountain_bike = 14
Bicicleta.nova(
15     tamanho: 'L',
16     peças: MountainBikeParts.new(rear_shock: 'Fox')
17
18 mountain_bike.size # -> 'L'
19
20 mountain_bike.spares 21 # ->
{:tire_size=>"2.1", :chain=>"10
22 #           velocidades", :rear_shock=>"Fox"}
23 #

```

Esta não foi uma grande mudança e não é uma grande melhoria. Contudo, esta refatoração revelou uma coisa útil; tornou incrivelmente óbvio o quanto pouco

Para começar, existia um código específico para bicicletas . A maior parte do código acima trata peças individuais; a hierarquia das Partes agora clama por outra refatoração.

Compondo o objeto Parts

Por definição, uma lista de peças contém uma lista de peças individuais. É hora de adicionar uma classe ao representam uma única parte. O nome da classe para uma parte individual claramente deveria ser, Papel mas introduzir uma classe Part quando você já tem uma classe Parts inicia uma conversa um desafio. É confuso usar a palavra “partes” para se referir a uma coleção de partes objetos, quando essa mesma palavra já se refere a um único objeto Parts . No entanto, o a frase anterior ilustra uma técnica que contorna o problema de comunicação; ao discutir Parte e Partes, você pode seguir o nome da classe com a palavra “objeto” e pluralize “objeto” conforme necessário.

Você também pode evitar o problema de comunicação desde o início, escolhendo nomes de classes diferentes, mas outros nomes podem não ser tão expressivos e podem introduzir seus próprios problemas de comunicação. Esta situação de peças/peças é comum o suficiente para que valha a pena lidar de frente. A escolha desses nomes de classe requer uma precisão de comunicação que é um objetivo digno por si só.

Assim, existe um objeto Parts , e ele pode conter muitos objetos Part – simples como que.

Compondo o objeto Parts

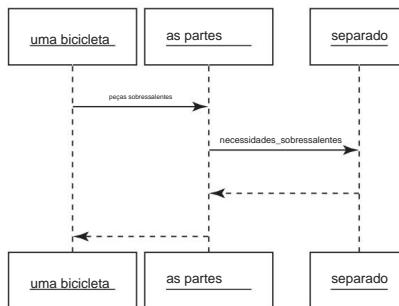


Figura 8.4 Bicicleta envia peças_sobressalentes para Peças , Peças envia necessidades_sobressalentes para cada Part .

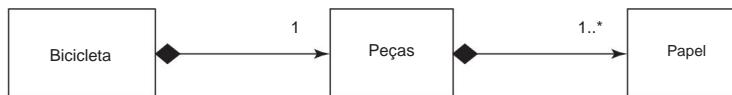


Figura 8.5 Bicicleta segura um Peças objeto, que por sua vez contém muitos Papel objetos.

Criando uma Peça

A Figura 8.4 mostra um novo diagrama de sequência que ilustra a conversa entre Bicycle e seu objeto Parts e entre um objeto Parts e seus objetos Part .

Bicycle envia peças sobressalentes para Parts e então o objeto Parts envia need_spare para cada parte.

Alterar o design desta forma requer a criação de um novo objeto Part . As partes object agora é composto de objetos Part , conforme ilustrado pelo diagrama de classes na Figura 8.5. O “1..**” na linha próxima à Peça indica que uma Parte terá uma ou mais Partes objetos.

A introdução desta nova classe Part simplifica a classe Parts existente , que agora torna-se um wrapper simples em torno de uma matriz de objetos Part . As peças podem filtrar sua lista de Separe objetos e devolva aqueles que precisam de peças sobressalentes. O código abaixo mostra três classes: a classe Bicycle existente, a classe Parts atualizada e a recém-introduzida Aula parcial .

```

Bicicleta de 1 classe
2 attr_reader :tamanho, :partes
3
4 def inicializar(args={})
5   @tamanho          = args[:tamanho]
6   @peças            = args[:partes]
  
```

```

7 fim
8
9 peças sobressalentes def
10     peças.sobressalentes
11 fim
12 fim
13
14 peças de classe
15 attr_reader :partes
16
17 def inicializar (partes)
18     @peças = peças
19 fim
20
21 peças sobressalentes def
22     peças.select { |parte| parte.needs_spare}
23 fim
24 fim
25
26a aula Parte
27 attr_reader :nome, :descrição, :needs_spare
28
29 def inicializar(args)
30     @nome           = args[:nome]
31     @descrição = args[:descrição]
32     @needs_spare = args.fetch(:needs_spare, verdadeiro)
33 fim
34 fim

```

Agora que essas três classes existem, você pode criar objetos Part individuais. A seguir o código cria várias partes diferentes e salva cada uma em uma variável de instância.

```

1 corrente =
2 Part.new(nome: 'corrente', descrição: '10 velocidades')
3
4 road_tire = 5
Part.new(nome: 'tamanho_do pneu', descrição: '23')
6
7 fita = 8
Part.new(nome: 'tape_color', descrição: 'red')
9
10 pneu_montanha =

```

Compondo o objeto Parts

```

11 Part.new(nome: 'tamanho_do_pneu', descrição: '2.1')
12
13 choque_traseiro =
14 Part.new (nome: 'rear_shock', descrição: 'Fox')
15
16 front_shock = 17
Parte.novo(
18     nome: 'front_shock',
19     descrição: 'Manitou',
20     necessidades_spare: falso)

```

Objetos de peças individuais podem ser agrupados em peças. O código abaixo combina os objetos Part da bicicleta de estrada em peças adequadas para bicicletas de estrada .

```

1 road_bike_parts = 2
Parts.new([corrente, road_tire, fita])

```

Claro, você pode pular esta etapa intermediária e simplesmente construir o objeto Parts dinamicamente ao criar uma bicicleta, conforme mostrado nas linhas 4–6 e 22–25 abaixo.

```

1 road_bike = 2
Bicicleta.new(
3     tamanho: 'L',
4     peças: Parts.new([cadeia,
5                         road_tire, fita]))
6
7
8 road_bike.tamanho          # -> 'EU'
9
10 road_bike.peças
11 # -> [<Parte:0x00000101036770
12 #           @name="chain",
13 #           @description="10 velocidades",
14 #           @needs_spare=true>,
15 #           <Parte:0x0000010102dc60
16 #           @name="tamanho_do_pneu",
17 #           etc...
18
19 mountain_bike = 20
Bicicleta.nova(
21     tamanho: 'L',

```

```

22     peças: Parts.new([cadeia,
23                         mountain_tire, choque
24                         frontal, choque
25                         traseiro]))
26
27 mountain_bike.tamanho          # -> 'EU'
28
29 mountain_bike.peças
30 # -> [#<Parte:0x00000101036770
31 #           @name="chain",
32 #           @description="10 velocidades",
33 #           @needs_spare=true>,
34 #           #<Parte:0x0000010101b678
35 #           @name="tamanho_do_pneu",
36 #           etc...

```

Como você pode ver nas linhas 8–17 e 27–34 acima, esse novo arranjo de código funciona muito bem e se comporta quase exatamente como a antiga hierarquia da bicicleta . Há um diferença: o antigo método de peças sobressalentes da bicicleta retornou um hash, mas esta nova peça sobressalente método retorna uma matriz de objetos Part .

Embora possa ser tentador pensar nesses objetos como instâncias de Parte, a composição lhe diz para pensar neles como objetos que desempenham o papel de Parte . Eles não precisam ser uma *espécie de classe* Part , eles apenas precisam agir como tal; isto é, eles devem responder a nome, descrição e necessidades_spare.

Tornando o objeto Parts mais parecido com um array

Este código funciona, mas definitivamente há espaço para melhorias. Dê um passo para trás por um minuto e pense nos métodos de peças e peças sobressalentes da bicicleta. Parece que essas mensagens deveriam retornar o mesmo tipo de coisa, mas os objetos que chegam custas não se comportam da mesma maneira. Veja o que acontece quando você pede a cada um seu tamanho.

Na linha 1 abaixo, a Spares tem o prazer de informar que seu tamanho é 3. No entanto, perguntando isso a mesma questão das partes não funciona tão bem, como você pode ver nas linhas 2–4.

```

1 mountain_bike.peças.tamanho # -> 3
2 mountain_bike.parts.size
3 # -> NoMethodError:
4 #           método indefinido 'tamanho' para #<Parts:...>

```

A linha 1 funciona porque Spares retorna um array (de objetos) e Array entende o tamanho. A linha 2 falha porque parts retorna instância de Parts, o que não acontece.

Falhas como essa irão perseguí-lo enquanto você possuir este código. Essas duas coisas parecem matrizes. Você inevitavelmente os tratará como se fossem, apesar do fato de que exatamente na metade das vezes o resultado será como pisar no proverbial ancinho no quintal. O objeto Parts não se comporta como um array e todas as tentativas de tratá-lo como tal falharão.

Você pode corrigir o problema imediato adicionando um método de tamanho a Parts. Esta é uma simples questão de implementar um método para delegar tamanho ao array real, conforme mostrado aqui:

```
1 tamanho definido
2     peças.tamanho
3 fim
```

No entanto, essa mudança coloca a classe Parts em um caminho escorregadio. Faça isso e não demorará muito para que você queira que as partes respondam a cada uma e depois classifiquem, e depois todo o resto no Array. Isso nunca acaba; quanto mais parecido com um array você criar Parts, mais parecido com um array você esperará que ele seja.

Talvez Parts seja um Array, embora com um pouco de comportamento extra. Você poderia fazer um; o próximo exemplo mostra uma nova versão da classe Parts , agora como uma subclasse de Array.

```
1 classe Peças < Matriz 2 def
peças sobressalentes
3     selecione {|part| parte.needs_spare}
4 fim
5 fim
```

O código acima é uma expressão muito direta da ideia de que Parts é uma especialização de Array; em uma linguagem orientada a objetos perfeita, esta solução seria exatamente correta. Infelizmente, a linguagem Ruby ainda não atingiu a perfeição e este design contém uma falha oculta.

O próximo exemplo ilustra o problema. Quando Parts subclassifica Array, ele herda todo o comportamento de Array . Esse comportamento inclui métodos como +, que soma dois arrays e retorna um terceiro. As linhas 3 e 4 abaixo mostram + combinando duas instâncias existentes de Parts e salvando o resultado na variável combo_parts .

Isso parece funcionar; `combo_parts` agora contém o número correto de peças (linha 7). No entanto, algo claramente não está certo. Como mostra a linha 12, `combo_parts` não pode responder às suas peças sobressalentes.

A causa raiz do problema é revelada nas linhas 15–17. Embora os objetos que foram `+`d juntos eram instâncias de `Parts`, o objeto que `+` retornou foi uma instância do `Array`, e o `Array` não entende peças sobressalentes.

```

1 # Parts herda '+' do Array, então você pode
2 #       adicione duas partes juntas.
3 combo_parts = 4
(mountain_bike.parts + road_bike.parts)
5
6 # '+' definitivamente combina as Partes
7 combo_parts.size                         # -> 7
8
9 # mas o objeto que '+' retorna
10 # não entende 'peças sobressalentes'
11 combo_parts.spares
12 # -> NoMethodError: método indefinido 'spares'
13 #       para #<Matriz:>
14
15 mountain_bike.parts.class # -> Peças
16 road_bike.parts.class # -> Peças
17 combo_parts.class                # -> Matriz!!!

```

Acontece que existem muitos métodos em `Array` que retornam novos arrays e, infelizmente, esses métodos retornam novas instâncias da classe `Array`, não novas instâncias de sua subclasse. A classe `Parts` ainda é enganosa e você acabou de trocar um problema para outro. Onde antes você ficou desapontado ao descobrir que `Parts` não tem tamanho do implemento, agora você pode se surpreender ao descobrir que adicionar duas partes juntas retorna um resultado que não comprehende peças sobressalentes.

Você viu três implementações diferentes de `Parts`. A primeira responde apenas à mensagens sobre peças sobressalentes e sobressalentes; não atua como um array, apenas contém um. O A implementação da segunda parte adiciona tamanho, uma pequena melhoria que apenas retorna o tamanho de sua matriz interna. As subclasses de implementação de peças mais recentes, `Array` e, portanto, dá a aparência de se comportar totalmente como uma matriz, mas como exemplo acima mostra, uma instância de `Parts` ainda exibe um comportamento inesperado.

Tornou-se claro que não existe uma solução perfeita; portanto, é hora de fazer uma decisão difícil. Mesmo que não possa responder ao tamanho, as peças originais

a implementação pode ser suficientemente boa; em caso afirmativo, você pode aceitar a falta de matrizes comportamento e reverter para essa versão. Se você precisar de tamanho e tamanho sozinho, pode ser é melhor adicionar apenas este método e então optar pela segunda implementação. Se você consegue tolerar a possibilidade de erros confusos ou sabe com absoluta certeza que você nunca os encontrará, pode fazer sentido subclassificar Array e caminhar silenciosamente.

Em algum lugar no meio termo entre complexidade e usabilidade está a seguinte solução. A classe Parts abaixo delega o tamanho e cada um para seu array @parts e inclui Enumerable para obter métodos comuns de travessia e pesquisa. Esse versão de Parts não possui todo o comportamento do Array, mas pelo menos tudo que afirma fazer realmente funciona.

```

1 requer 'encaminhamento'
2 peças de classe
3 estender Encaminhável
4 def_delegadores :@partes, :size, :each
5 incluem Enumerable
6
7 def inicializar (partes)
8     @peças = peças
9   fim
10
11 peças sobressalentes def
12     selecione ({parte| parte.needs_spare})
13   fim
14   fim

```

Enviar + para uma instância deste *Parts* resulta em uma exceção NoMethodError . No entanto, como Parts agora responde ao tamanho, cada um e todos os Enumerable gentilmente gera erros quando você o trata erroneamente como um array real, este código pode ser bom o suficiente. O exemplo a seguir mostra que peças sobressalentes e peças agora podem ambos respondem ao tamanho.

```

1 mountain_bike = 2
Bicicleta.new(
3     tamanho: 'L',
4     peças: Parts.new([cadeia,
5                         pneu_montanha,
6                         choque frontal,

```

```

7     amortecedor traseiro)])
8
9 mountain_bike.peças.tamanho # -> 3
10 mountain_bike.parts.size # -> 4

```

Você novamente tem uma versão funcional das classes Bicycle, Parts e Part . Está na hora para reavaliar o design.

Fabricação de peças

Reveja as linhas 4–7 acima. Os objetos Part mantidos na cadeia, mountain_tire, e assim por diante, as variáveis foram criadas há tanto tempo que você já deve ter esquecido eles. Pense no corpo de conhecimento que essas quatro linhas representam. Em algum lugar na sua aplicação, algum objeto precisava saber como criar esses objetos Part . E aqui, nas linhas 4–7 acima, este lugar tem que saber que esses quatro objetos específicos vão com bicicletas de montanha.

Isso é muito conhecimento e pode facilmente vaziar para toda a sua aplicação. Esse o vazamento é lamentável e desnecessário. Embora existam muitas peças individuais diferentes, existem apenas algumas combinações válidas de peças. Tudo seria mais fácil se você pudesse descrever as diferentes bicicletas e depois usar suas descrições para fabrique magicamente o objeto Parts correto para qualquer bicicleta.

É fácil descrever a combinação de peças que compõem uma bicicleta específica. O código abaixo faz isso com um array bidimensional simples, onde cada linha contém três colunas possíveis. A primeira coluna contém o nome da peça ('chain', 'tire_size', etc.), o segundo, a descrição da peça ('10 velocidades', '23', etc.) e o terceiro (que é opcional), um booleano que indica se esta peça precisa de reposição. Apenas 'front_shock' na linha 9 abaixo coloca um valor nesta terceira coluna, a outra as peças gostariam de ser padronizadas como verdadeiras, pois exigem peças sobressalentes.

```

1 road_config = [['chain',
2     ['tire_size'],
3     ['tape_color'],
4     ['vermelho']]
5
6 mountain_config = [['corrente',
7     '10 velocidades'],
8     ['tamanho_do_pneu', '2.1'],
9     ['front_shock', 'Manitou', falso],
10    ['rear_shock', 'Fox']]

```

Ao contrário de um hash, esta matriz bidimensional simples não fornece informações estruturais. No entanto, você entende como essa estrutura é organizada e pode codificar seu conhecimento em um novo objeto que fabrica peças.

Criando a PartsFactory

Conforme discutido no Capítulo 3, Gerenciando Dependências, um objeto que fabrica outros objetos é uma fábrica. Sua experiência anterior em outros idiomas pode predispor você recuar ao ouvir essa palavra, mas pense nisso como uma oportunidade para recuperá-la. A palavra *fábrica* não significa difícil, artificial ou excessivamente complicado; isso é apenas a palavra que os designers OO usam para comunicar de forma concisa a ideia de um objeto que cria outros objetos. As fábricas Ruby são simples e não há razão para evitar isso palavra reveladora de intenção.

O código abaixo mostra um novo módulo PartsFactory . Seu trabalho é pegar um array como um dos listados acima e fabricar um objeto Parts . Ao longo do caminho podemos criar objetos Part , mas esta ação é privada. Sua responsabilidade pública é criar uma Parte .

Esta primeira versão do PartsFactory leva três argumentos, uma configuração e os nomes das classes a serem usadas para Parte e Peças . A linha 6 abaixo cria o novo instância de Parts , inicializando-a com uma matriz de objetos Parte construídos a partir das informações na configuração .

```

1 módulo PartsFactory
2 def self.build(config,
3           part_class = Parte,
4           parts_class = Peças)
5
6   peças_class.new(
7     config.collect { |part_config|
8       parte_class.new(
9         nome:                 parte_config[0],
10        descrição: part_config[1],
11        precisa_spare: part_config.fetch(2, verdadeiro))})
12 fim
13 fim

```

Esta fábrica conhece a estrutura do array de configuração . Nas linhas 9–11 acima, ele espera nome estar na primeira coluna, descrição na segunda e necessidades_sobressalentes estar no terceiro.

Colocar o conhecimento da estrutura da configuração na fábrica tem duas consequências.

Primeiro, a configuração pode ser expressa de forma muito concisa. Porque a PartsFactory entende estrutura interna do config , config pode ser especificado como um array em vez de um hash.

Segundo, uma vez que você se compromete a manter a configuração em um array, você deve *sempre* criar novos objetos Parts usando a fábrica. Para criar novas peças através de qualquer outro mecanismo requer a duplicação do conhecimento codificado nas linhas 9–11 acima.

Agora que o PartsFactory existe, você pode usar as matrizes de configuração definidas acima para criar facilmente novas peças, conforme mostrado aqui:

```

1 road_parts = PartsFactory.build(road_config)
2 # -> [#<Parte:0x00000101825b70
3 #           @name="chain",
4 #           @description="10 velocidades",
5 #           @needs_spare=true>,
6 #           #<Parte:0x00000101825b20
7 #           @name="tamanho_do_pneu",
8 #           etc...
9 #
10 mountain_parts = PartsFactory.build(mountain_config)
11 # -> [#<Parte:0x0000010181ea28
12 #           @name="chain",
13 #           @description="10 velocidades",
14 #           @needs_spare=true>,
15 #           #<Parte:0x0000010181e9d8
16 #           @nome="tamanho_do_pneu", etc...
17 #

```

PartsFactory, combinado com as novas matrizes de configuração, isola todo o conhecimento necessário para criar peças válidas. Esta informação foi anteriormente dispersa em todo o aplicativo, mas agora está contido nesta classe e nessas duas matrizes.

Aproveitando o PartsFactory

Agora que o PartsFactory está instalado e funcionando, dê uma outra olhada na classe Part (repetido abaixo). Parte é simples. Não só isso, o único, mesmo *um pouco* complicado linha de código (a busca na linha 7 abaixo) é duplicada em PartsFactory. Se PartsFactory criou cada parte, a parte não precisaria desse código. E se você remover esse código da Part, não sobrou quase nada; você pode substituir toda a classe Part com um OpenStruct simples.

```

1 aula Parte
2 attr_reader :nome, :descrição, :needs_spare
3
4 def inicializar (args)
5   @nome           = args[:nome]
6   @descrição     = args[:descrição]
7   @needs_spare   = args.fetch(:needs_spare, verdadeiro)
8 fim
9 fim

```

A classe OpenStruct do Ruby é muito parecida com a classe Struct que você já viu, ela fornece uma maneira conveniente de agrupar vários atributos em um objeto. A diferença entre os dois é que Struct aceita argumentos de inicialização de ordem de posição enquanto o OpenStruct pega um hash para sua inicialização e então deriva atributos de o haxixe.

Existem bons motivos para remover a classe Part ; isso simplifica o código e você talvez nunca mais precise de algo tão complicado quanto o que você tem atualmente. Você pode remova todos os vestígios de Part excluindo a classe e alterando PartsFactory para use OpenStruct para criar um objeto que desempenhe a função de Parte . O seguinte código mostra uma nova versão do PartFactory onde a criação de peças foi refatorada em um método próprio (linha 9).

```

1 requer 'ostruct'
2 módulos PartsFactory
3 def self.build(config, parts_class = Parts)
4   peças_class.new(
5     config.collect {|part_config|
6       create_part(part_config)})
7 fim
8
9 def self.create_part(part_config)
10   OpenStruct.new(
11     nome:             parte_config[0],
12     descrição:       parte_config[1],
13     precisa_spare:  parte_config.fetch(2, verdadeiro))
14 fim
15 fim

```

A linha 13 acima é agora o único local no aplicativo cujo padrão é need_spare como verdade, então a PartsFactory deve ser a única responsável pela fabricação das peças.

Esta nova versão do PartsFactory funciona. Conforme mostrado abaixo, ele retorna um arquivo Parts que contém uma matriz de objetos OpenStruct , cada um dos quais desempenha a função Part .

```

1 mountain_parts = PartsFactory.build(mountain_config)
2 # -> <Partes:0x000001009ad8b8 @partes=
3 #
4 #           [#<OpenStruct name="chain", description="10-
5 #                           speed", need_spare=true>,
6 #           #<OpenStruct name="tire_size",
7 #           description="2.1",
8 #           etc...

```

A bicicleta composta

O código a seguir mostra que Bicycle agora usa composição. Mostra Bicicleta, Parts e PartsFactory e as matrizes de configuração para bicicletas de estrada e de montanha.

A bicicleta *possui* um Parts, que por sua vez *possui* uma coleção de objetos Part . Peças e Parte pode existir como classes, mas os objetos nos quais estão contidas pensam nelas como papéis. Parts é uma classe que desempenha a função Parts ; implementa peças sobressalentes. O papel de Parte é desempenhada por um OpenStruct, que implementa nome, descrição e necessidades_spare.

As 54 linhas de código a seguir substituem completamente a hierarquia de herança de 66 linhas do Capítulo 6.

Bicicleta de 1 classe

```

2 attr_reader :tamanho, :partes
3
4 def inicializar(args={})
5   @tamanho        = args[:tamanho]
6   @peças          = args[:partes]
7 fim
8
9 peças sobressalentes def
10    peças.sobressalentes
11 fim
12 fim
13
14 exigem 'encaminhamento'
15 peças de classe
16 estender Encaminhável

```

A bicicleta composta

```

17 def_delegadores :@partes, :size, :each
18 incluem Enumerável
19
20 def inicializar (partes)
21     @peças = peças
22 fim
23
24 peças sobressalentes def
25     selecione ({|parte| parte.needs_spare})
26 fim
27 fim
28
29 requerem 'ostruct'

Fábrica de peças de 30 módulos

31 def self.build(config, parts_class = Parts)
32     peças_class.new(
33         config.collect ({|part_config| create_part(part_config)}))
34
35 fim
36
37 def self.create_part(part_config)
38     OpenStruct.new(
39         nome:                 parte_config[0],
40         descrição: part_config[1],
41         precisa_spare: part_config.fetch(2, verdadeiro))
42 fim
43 fim
44
45 road_config = [['cadeia',
46     ['tamanho_do'           '10 velocidades'],
47     pneu, ['cor da fita',   '23'],
48                           'vermelho']]
49
50 mountain_config = [['corrente',
51     '10 velocidades'],
52     ['tamanho_do_pneu', '2.1'],
53     ['front_shock', 'Manitou', falso],
54     ['rear_shock', 'Fox']]
```

Este novo código funciona de forma muito semelhante à hierarquia anterior da Bicycle . A única diferença é que a mensagem de peças sobressalentes agora retorna uma matriz de objetos semelhantes a peças em vez de um hash, como você pode ver nas linhas 7 e 15 abaixo.

```
1 road_bike = 2
Bicicleta.new(
3         tamanho: 'L',
4         partes: PartsFactory.build(road_config))
5
6 road_bike.peças
7# -> [#<nome OpenStruct="cadeia", etc...
8
9 mountain_bike =
10 Bicicleta.nova(
11         tamanho: 'L',
12         partes: PartsFactory.build(mountain_config))
13
14 mountain_bike.peças
15# -> [#<Nome OpenStruct="cadeia", etc...
```

Agora que existem essas novas classes, é muito fácil criar um novo tipo de bicicleta.

Adicionar suporte para bicicletas reclinadas exigiu 19 novas linhas de código no Capítulo 6. A tarefa agora pode ser realizada com 3 linhas de configuração (linhas 2–4 abaixo).

Conforme mostrado nas linhas 11–23 acima, agora você pode criar uma nova bicicleta simplesmente descrevendo suas partes.

Agregação: um tipo especial de composição

Você já conhece o termo *delegação*; delegação é quando um objeto recebe uma mensagem e apenas a encaminha para outro. A delegação cria dependências; o objeto receptor deve reconhecer a mensagem e saber onde Envie isto.

A *composição* geralmente envolve delegação, mas o termo significa algo mais. Um objeto *composto* é composto de partes com as quais espera interagir através de interfaces bem definidas.

A composição descreve um relacionamento *tem-um*. As refeições têm aperitivos, as universidades têm departamentos, as bicicletas têm peças. Refeições, universidades e bicicletas são objetos compostos. Aperitivos, departamentos e peças são funções. O objeto composto depende da interface da função.

Como as refeições interagem com os aperitivos por meio de uma interface, novos objetos que desejam atuar como aperitivos precisam apenas implementar esta interface. Aperitivos inesperados se encaixam perfeitamente e de forma intercambiável em refeições.

O termo *composição* pode ser um pouco confuso porque é usado para duas conceitos ligeiramente diferentes. A definição acima é para o uso mais amplo do prazo. Na maioria dos casos, quando você vê a *composição*, não indicará mais nada do que este geral *tem - uma* relação entre dois objetos.

No entanto, tal como definido formalmente, significa algo um pouco mais específico; isto indica um relacionamento *tem-* a onde o objeto contido não tem vida independente de seu contêiner. Quando usado neste sentido mais estrito, você sabe não apenas que as refeições tenham aperitivos, mas também que, uma vez consumida a refeição, o aperitivo também se foi.

Isto deixa uma lacuna na definição que é preenchida pelo termo *agregação*. A agregação é exatamente como a composição, exceto que o objeto contido possui uma vida independente. As universidades possuem departamentos, que por sua vez contam com professores. Se sua aplicação gerencia muitas universidades e conhece milhares de professores, é bastante razoável esperar que, embora um departamento desapareça completamente quando a sua universidade extingue-se, os seus professores continue a existir.

A relação universidade-departamento é de composição (em sua sentido mais estrito) e a relação departamento-professor é de agregação.

Destruir um departamento não destrói seus professores; eles têm existência e vida próprias.

Esta distinção entre composição e agregação pode ter pouco efeito prático no seu código. Agora que você está familiarizado com ambos os termos, você pode usar *composição* para se referir a ambos os tipos de relacionamento e ser mais explícito somente se for necessário.

Decidindo entre herança e composição

Lembre-se de que a herança clássica é uma *técnica de organização de código*. O comportamento é disperso entre os objetos e esses objetos são organizados em relacionamentos de classe, de modo que a delegação automática de mensagens invoca o comportamento correto. Pense desta forma: pelo custo de organizar objetos em uma hierarquia, você obtém delegação de mensagens gratuitamente.

A composição é uma alternativa que reverte esses custos e benefícios. Na composição, o relacionamento entre objetos não é codificado na hierarquia de classes; em vez disso, os objetos são independentes e, como resultado, devem conhecer explicitamente e delegar mensagens uns aos outros. A composição permite que os objetos tenham independência estrutural, mas ao custo da delegação explícita de mensagens.

Agora que você viu exemplos de herança e composição, você pode começar a pensar em quando usá-los. A regra geral é que, diante de um problema que a composição pode resolver, você deve ter tendência a fazê-lo. Se você não pode defender explicitamente a herança como uma solução melhor, use a composição. A composição contém muito menos dependências integradas que a herança; muitas vezes é a melhor escolha.

A herança é uma solução melhor quando seu uso oferece altas recompensas por baixo risco. Esta seção examina os custos e benefícios da herança versus composição e fornece diretrizes para a escolha do melhor relacionamento.

Aceitar as consequências da herança Fazer escolhas sábias sobre a utilização da herança requer uma compreensão clara dos seus custos e benefícios.

Benefícios da herança

O Capítulo 2, Projetando classes com uma única responsabilidade, delineou quatro objetivos para o código: ele deve ser transparente, razoável, utilizável e exemplar. A herança, quando aplicada corretamente, se destaca no segundo, terceiro e quarto objetivos.

Os métodos definidos perto do topo das hierarquias de herança têm ampla influência porque a altura da hierarquia atua como uma alavancas que multiplica seus efeitos.

As alterações feitas nesses métodos afetam a árvore de herança. Hierarquias modeladas corretamente são, portanto, extremamente *razoáveis*; grandes mudanças no comportamento podem ser alcançadas por meio de pequenas mudanças no código.

O uso de herança resulta em código que pode ser descrito como *aberto-fechado*; as hierarquias estão abertas para extensão, mas permanecem fechadas para modificação. Adicionar uma nova subclasse a uma hierarquia existente não requer alterações no código existente. As hierarquias são, portanto, *utilizáveis*; você pode criar facilmente novas subclasses para acomodar novas variantes.

Hierarquias escritas corretamente são fáceis de estender. A hierarquia incorpora a abstração e cada nova subclass apresenta algumas diferenças concretas. O padrão existente é fácil de seguir e replicá-lo será a escolha natural de qualquer programador encarregado de criar novas subclasses. As hierarquias são, portanto, *exemplares*; por sua natureza, eles fornecem orientação para escrever o código para estendê-los.

Você não precisa ir além da fonte das próprias linguagens orientadas a objetos para ver o valor da organização do código usando herança. Em Ruby, a classe Numeric fornece um excelente exemplo. Integer e Float são modelados como subclasses de Numeric; este é *um* relacionamento é exatamente certo. Inteiros e flutuantes são fundamentalmente *números*. Permitir que essas duas classes compartilhem uma abstração comum é a maneira mais parcimoniosa de organizar o código.

Custos de herança

As preocupações sobre o uso da herança recaem em duas áreas diferentes. O primeiro medo é que você possa ser enganado e escolher a herança para resolver o tipo errado de problema. Se você cometer esse erro, chegará o dia em que você precisará adicionar comportamento, mas descobrirá que não há uma maneira fácil de fazer isso. Como o modelo está incorreto, o novo comportamento não se ajusta; neste caso, você será forçado a duplicar ou reestruturar o código.

Segundo, mesmo quando a herança faz sentido para o problema, você pode estar escrevendo código que será usado por outros para propósitos que você não previu. Esses outros programadores desejam o comportamento que você criou, mas podem não conseguir tolerar as dependências exigidas pela herança.

A seção anterior sobre os benefícios da herança teve o cuidado de qualificar as suas afirmações como aplicáveis apenas a uma “hierarquia corretamente modelada”. Imagine *razoável*, *utilizável* e *exemplar* como moedas de dois lados. O lado do benefício representa os ganhos maravilhosos que a herança proporciona. Se você aplicar a herança a um problema para o qual ela não é adequada, você efetivamente virará essas moedas e encontrará um prejuízo paralelo.

O outro lado da moeda *razoável* é o custo muito elevado de fazer alterações perto do topo de uma hierarquia modelada incorretamente. Neste caso, o efeito de alavancas funciona a seu favor; pequenas mudanças quebram tudo.

O lado oposto da moeda *utilizável* é a impossibilidade de adicionar comportamento quando novas subclasses representam uma mistura de tipos. A hierarquia das bicicletas no Capítulo 6 falhou quando surgiu a necessidade de bicicletas de montanha reclinadas. Esta hierarquia já contém subclasses para MountainBike e RecumbentBike; combinar as qualidades dessas duas classes em um único objeto não é possível na hierarquia que existe atualmente. Você não pode reutilizar o comportamento existente sem alterá-lo.

O outro lado da moeda *exemplar* é o caos que se segue quando programadores novatos tentam estender hierarquias modeladas incorretamente. Estas hierarquias inadequadas não devem ser estendidas, elas precisam ser refatoradas, mas os novatos não têm as habilidades para fazê-lo. Os novatos são forçados a duplicar o código existente ou a adicionar dependências nos nomes das classes, o que serve para exacerbar os problemas de design existentes.

A herança, portanto, é um lugar onde a pergunta “O que acontecerá quando eu estiver errado?” assume especial importância. A herança, por definição, vem com um conjunto profundamente enraizado de dependências. As subclasses dependem dos métodos definidos nas suas superclasses e da delegação automática de mensagens a essas superclasses.

Esta é a maior força e a maior fraqueza da herança clássica; as subclasses estão vinculadas, irrevogavelmente e por design, às classes acima delas na hierarquia. Essas dependências integradas amplificam os efeitos das modificações feitas nas superclasses.

Mudanças de comportamento enormes e abrangentes podem ser alcançadas com mudanças muito pequenas no código.

Isso é verdade, para o bem ou para o mal, quer você se arrependa ou não.

Finalmente, a sua consideração sobre o uso da herança deve ser moderada pelas suas expectativas sobre a população que utilizará o seu código. Se você estiver escrevendo código para um aplicativo interno em um domínio com o qual está intimamente familiarizado, poderá prever o futuro bem o suficiente para ter certeza de que seu problema de design é aquele para o qual a herança é uma solução econômica. À medida que você escreve código para um público mais amplo, sua capacidade de antecipar necessidades necessariamente diminui e a adequação de exigir herança como parte da interface diminui.

Evite escrever estruturas que exijam que os usuários do seu código criem subclasses de seus objetos para obter seu comportamento. Os objetos de sua aplicação já podem estar organizados em uma hierarquia; herdar de sua estrutura pode não ser possível.

Aceitando as consequências da composição

Os objetos construídos usando composição diferem daqueles construídos usando herança de duas maneiras básicas. Os objetos compostos não dependem da estrutura da hierarquia de classes e delegam suas próprias mensagens. Essas diferenças conferem um conjunto diferente de custos e benefícios.

Benefícios da composição

Ao usar a composição, a tendência natural é criar muitos objetos pequenos que contenham responsabilidades diretas e acessíveis por meio de interfaces claramente definidas. Esses objetos bem compostos se destacam quando comparados a vários dos objetivos do Capítulo 2 para código.

Esses pequenos objetos têm uma única responsabilidade e especificam seu próprio comportamento. Eles são *transparentes*; é fácil entender o código e fica claro o que acontecerá se ele mudar. Além disso, a independência do objeto composto da hierarquia significa que ele herda muito pouco código e, portanto, geralmente está imune a sofrer efeitos colaterais como resultado de alterações nas classes acima dele na hierarquia.

Como os objetos compostos lidam com suas partes por meio de uma interface, adicionar um novo tipo de peça é uma simples questão de conectar um novo objeto que honre a interface.

Do ponto de vista do objeto composto, adicionar uma nova variante de uma peça existente é *razoável* e não requer alterações no seu código.

Pela sua própria natureza, os objetos que participam da composição são pequenos, estruturalmente independentes e possuem interfaces bem definidas. Isso permite sua transição perfeita para componentes conectáveis e intercambiáveis. Objetos bem compostos são, portanto, facilmente *utilizáveis* em contextos novos e inesperados.

Na melhor das hipóteses, a composição resulta em aplicativos construídos com objetos simples e conectáveis, fáceis de estender e com alta tolerância a alterações.

Custos da Composição Os

pontos fortes da composição, como acontece com a maioria das coisas na vida, contribuem para os seus pontos fracos.

Um objeto composto depende de suas muitas partes. Mesmo que cada parte seja pequena e de fácil compreensão, a operação combinada do todo pode não ser tão óbvia. Embora cada parte individual possa de facto ser *transparente*, o todo pode não o ser.

Os benefícios da independência estrutural são obtidos às custas da delegação automática de mensagens. O objeto composto deve saber explicitamente quais mensagens delegar e para quem. Código de delegação idêntico pode ser necessário para muitos objetos diferentes; composição não oferece nenhuma maneira de compartilhar esse código.

Como ilustram esses custos e benefícios, a composição é excelente para prescrever regras para a montagem de um objeto feito de peças, mas não fornece tanta ajuda para o problema de organizar o código para uma coleção de peças que são quase idênticas.

Escolhendo relacionamentos A herança

clássica (Capítulo 6), o compartilhamento de comportamento por meio de módulos (Capítulo 7, Compartilhando comportamento de função com módulos) e a composição são soluções perfeitas para o problema que resolvem. O truque para reduzir os custos de aplicação é aplicar cada técnica ao problema certo.

Alguns dos grandes mestres do design orientado a objetos deram conselhos sobre usando herança e composição.

- “Herança é especialização.” — Bertrand Meyer, *Touch of Class: Aprendendo a programar bem com objetos e contratos*
- “A herança é mais adequada para adicionar funcionalidade a classes existentes quando você usará a maior parte do código antigo e adicionará quantidades relativamente pequenas de código novo.” — Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, *Padrões de Design: Elementos de Software Orientado a Objetos Reutilizáveis*
- “Use composição quando o comportamento for maior que a soma de suas partes.” — paráfrase de Grady Booch, *Análise e Design Orientados a Objetos*

Usar herança para relacionamentos *is-a*

Quando você seleciona herança em vez de composição, você está apostando que os benefícios assim acumulados superarão os custos. Algumas apostas têm maior probabilidade de dar resultado do que outras. Pequenos conjuntos de objetos do mundo real que se enquadram naturalmente em hierarquias de especialização estáticas e transparentes são candidatos a serem modelados usando herança clássica.

Imagine um jogo onde os jogadores correm de bicicleta. Os jogadores montam suas bicicletas “comprando” peças. Uma das peças que eles podem comprar é um choque. O jogo oferece seis choques quase idênticos; cada um difere ligeiramente em custo e comportamento.

Todos esses choques são, bem, *choques*. O seu “choque” está no cerne da sua identidade. Os choques não existem mais em nenhuma categoria atômica. As variantes dos choques são muito mais parecidas do que diferentes. A afirmação mais precisa e descritiva que você pode fazer sobre qualquer uma das variantes é que é *um* choque.

A herança é perfeita para esse problema. Os choques podem ser modelados como uma hierarquia estreita e rasa. O pequeno tamanho da hierarquia a torna compreensível, reveladora de intenções e facilmente extensível. Como esses objetos atendem aos critérios para o uso bem-sucedido da herança, o risco de estar errado é baixo, mas no caso improvável de você *estar* errado, o custo de mudar de ideia também é baixo. Você pode obter os benefícios da herança expondo-se a alguns de seus riscos.

Nos termos do exemplo deste Capítulo, cada choque diferente desempenha o papel de Parte. Ele herda o comportamento de choque comum e a função Part de sua superclasse abstrata Shock . O PartsFactory atualmente assume que cada peça pode ser representada pelo Part OpenStruct, mas você pode facilmente estender a matriz de configuração da peça para fornecer o nome da classe para um choque específico. Como você já pensa em Part como uma interface, é fácil conectar um novo tipo de parte, mesmo que essa parte use herança para obter parte de seu comportamento.

Se os requisitos mudarem de tal forma que haja uma explosão nos tipos de choques, reavalie esta decisão de projeto. Talvez ainda seja válido, talvez não. Se a modelação de um conjunto de novos choques exigir uma expansão drástica da hierarquia, ou se os novos choques não se enquadarem convenientemente no código existente, considere as alternativas *neste momento*.

Use tipos de pato para relacionamentos *que se comportam como*

um Alguns problemas exigem que muitos objetos diferentes desempenhem um papel comum. Além de suas responsabilidades principais, os objetos podem desempenhar funções como *agendáveis*, *preparáveis*, *imprimíveis* ou *persistentes*.

Existem duas maneiras principais de reconhecer a existência de uma função. Primeiro, embora um objeto o desempenhe, o papel não é a principal responsabilidade do objeto. Uma bicicleta se *comporta como uma* programável, mas é *uma* bicicleta. Em segundo lugar, a necessidade é generalizada; muitos objetos de outra forma não relacionados compartilham o desejo de desempenhar o mesmo papel.

A maneira mais esclarecedora de pensar sobre os papéis é do lado de fora, do ponto de vista de quem detém um papel, e não do ponto de vista de quem desempenha um papel. O detentor de um *programável* espera que ele implemente a interface do programável e honre o contrato do programável . Todos os *programáveis* são semelhantes no sentido de que devem atender a essas expectativas.

Sua tarefa de design é reconhecer que existe uma função, definir a interface do seu tipo de pato e fornecer uma implementação dessa interface para cada jogador possível. Algumas funções consistem apenas em sua interface, outras compartilham um comportamento comum. Defina o comportamento comum em um módulo Ruby para permitir que objetos desempenhem a função sem duplicar o código.

Use composição para relacionamentos *has-*

a Muitos objetos contêm diversas partes, mas são mais do que a soma dessas partes.

As bicicletas *têm* peças, mas a bicicleta em si é algo mais. Tem um comportamento separado e adicional ao comportamento de suas partes. Dados os requisitos atuais do exemplo da bicicleta, a maneira mais econômica de modelar o objeto Bicicleta é por meio da composição.

Esta distinção *é-um* versus *tem-um* está no cerne da decisão entre herança e composição.

Quanto mais partes um objeto tiver, maior será a probabilidade de ele ser modelado com composição.

Quanto mais você se aprofunda nas partes individuais, maior a probabilidade de descobrir uma parte específica que possui algumas variantes especializadas e, portanto, é uma candidata razoável à herança. Para cada problema, avalie os custos e benefícios de técnicas alternativas de projeto e use seu julgamento e experiência para fazer a melhor escolha.

A Composição

de Resumo permite combinar pequenas partes para criar objetos mais complexos, de modo que o todo se torne mais do que a soma de suas partes. Objetos compostos tendem a consistir em entidades simples e discretas que podem ser facilmente reorganizadas em novas combinações. Esses objetos simples são fáceis de entender, reutilizar e testar, mas como se combinam em um todo mais complicado, a operação da aplicação maior pode não ser tão fácil de entender quanto a das partes individuais.

Composição, herança clássica e compartilhamento de comportamento por meio de módulos são técnicas concorrentes para organizar código. Cada um tem custos e benefícios diferentes; essas diferenças os predispõem a resolver melhor problemas ligeiramente diferentes.

Essas técnicas são ferramentas, nada mais, e você se tornará um designer melhor se praticar cada uma delas. Aprender a usá-los corretamente é uma questão de experiência e julgamento, e uma das melhores maneiras de ganhar experiência é aprender com seus próprios erros. A chave para melhorar suas habilidades de design é tentar essas técnicas, aceitar seus erros com alegria, permanecer desapegado de decisões de design anteriores e refatorar impiedosamente.

À medida que você ganha experiência, você ficará melhor na escolha da técnica correta na primeira vez, seus custos diminuirão e suas aplicações melhorarão.

CAPÍTULO 9

Projetando com boa relação custo-benefício Testes

Escrever código mutável é uma arte cuja prática depende de três habilidades diferentes.

Primeiro, você deve entender o design orientado a objetos. Código mal projetado é naturalmente difícil de mudar. Do ponto de vista prático, a mutabilidade é o único projeto métrica que importa; o código fácil de alterar é bem projetado. Porque você leu até agora, é justo presumir que seus esforços serão recompensados e que você adquiriu uma base para começar a prática de projetar código mutável.

Em segundo lugar, você deve ter habilidade em refatorar código. Não no sentido casual de “vá no aplicativo e jogar algumas coisas por aí”, mas no sentido real, adulto e à prova de balas definido por Martin Fowler em *Refactoring: Improving the Design of Existing*

Código:

Refatoração é o processo de alterar um sistema de software de tal maneira que não altera o comportamento externo do código, mas melhora o estrutura interna.

Observe que a frase *não altera o comportamento externo do código*. Refatoração, como formalmente definido, não adiciona novo comportamento, melhora a estrutura existente. É um processo preciso que altera o código por meio de pequenas etapas semelhantes a caranguejos e de forma cuidadosa, incremental e infalível transforma um projeto em outro.

Um bom design preserva o máximo de flexibilidade a um custo mínimo, adiando decisões em todas as oportunidades, adiando compromissos até requisitos mais específicos

chegar. Quando esse dia chegar, *refatorar* é como você transforma a estrutura de código atual em uma que irá acomodar os novos requisitos. Novos recursos serão adicionados somente depois que você refatorar o código com sucesso.

Se suas habilidades de refatoração forem fracas, melhore-as. A necessidade de refatoração contínua é uma consequência de um bom design; seus esforços de design renderão todos os dividendos somente quando você puder refatorar com facilidade.

Finalmente, a arte de escrever código mutável requer a habilidade de escrever testes de alto valor. Os testes lhe dão confiança para refatorar constantemente. Testes eficientes comprovam que o código alterado continua a se comportar corretamente sem aumentar os custos gerais. Bons testes de refatorações de código climático com desenvoltura; eles são escritos de forma que alterações no código não forcem a reescrita dos testes.

Escrever testes que possam realizar esse truque é uma questão de design e é o tema deste capítulo.

Uma compreensão do design orientado a objetos, boas habilidades de refatoração e a capacidade de escrever testes eficientes formam um banco de três pernas sobre o qual repousa o código mutável. Código bem projetado é fácil de alterar, refatorar é como você muda de um design para outro e os testes liberam você para refatorar impunemente.

Testes Intencionais Os argumentos

mais comuns para a realização de testes são que eles reduzem bugs e fornecem documentação, e que escrever testes *primeiro* melhora o design da aplicação.

Esses benefícios, embora válidos, são indicadores de um objetivo mais profundo. O verdadeiro propósito dos testes, assim como o verdadeiro propósito do design, é reduzir custos. Se escrever, manter e executar testes consome mais tempo do que seria necessário para corrigir bugs, escrever documentação e projetar aplicativos, claramente não vale a pena escrever testes e nenhuma pessoa racional argumentaria o contrário.

É comum que os programadores que são novos em testes se encontrem no estado infeliz em que os testes que escrevem *custam* mais do que o valor que esses testes fornecem e que, portanto, queiram discutir sobre o valor dos testes. Esses são programadores que se consideravam altamente produtivos em suas vidas anteriores de teste, mas que colidiram com a parede do teste *primeiro* e pararam. Suas tentativas de programar *primeiro* o teste resultam em menos resultados, e seu desejo de recuperar a produtividade os leva a voltar aos velhos hábitos e renunciar à escrita de testes.

A solução para o problema dos testes dispendiosos, contudo, não é parar de testar, mas sim melhorar.

Obter um bom valor dos testes requer clareza de intenção e saber o que, quando e como testar.

Conhecendo suas intenções

Os testes têm muitos benefícios potenciais, alguns óbvios, outros mais obscuros. Uma compreensão completa desses benefícios aumentará sua motivação para alcançá-los.

Encontrando

Bugs Encontrar falhas, ou bugs, no início do processo de desenvolvimento gera grandes dividendos. Não apenas é mais fácil encontrar e corrigir um bug mais próximo de sua criação, mas acertar o código mais cedo ou mais tarde pode ter efeitos positivos inesperados no design resultante. Saber que você pode (ou não) fazer algo desde o início pode fazer com que você escolha alternativas no presente que alterem as opções de design disponíveis no futuro. Além disso, à medida que o código se acumula, os bugs incorporados adquirem dependências. A correção desses bugs no final do processo pode exigir a alteração de muitos códigos dependentes. Corrigir bugs antecipadamente sempre reduz custos.

Fornecimento de documentação

Os testes fornecem a única documentação confiável do projeto. A história que contam permanece verdadeira muito depois de os documentos em papel se tornarem obsoletos e a memória humana falhar. Escreva seus testes como se você esperasse que seu futuro eu tivesse amnésia. Lembre-se de que você esquecerá; escreva testes que o lembrem da história assim que o fizer.

Adiar decisões de projeto Os testes

permitem adiar decisões de projeto com segurança. À medida que suas habilidades de design melhoram, você começará a escrever aplicativos repletos de locais onde você sabe que o design precisa *de algo*, mas ainda não tem informações suficientes para saber exatamente o quê.

Esses são os lugares onde você aguarda informações adicionais, resistindo valentemente às forças que o obrigam a se comprometer com um projeto específico.

Esses pontos de decisão “pendentes” são frequentemente codificados como hacks extremamente embaralhados e extremamente concretos, escondidos atrás de interfaces totalmente apresentáveis. Esta situação ocorre quando se tem conhecimento de apenas um caso concreto no presente, mas espera-se que novos casos cheguem num futuro próximo. Você sabe que em algum momento será melhor atendido por um código que lide com esses muitos casos concretos como uma única abstração, mas no momento você não tem informações suficientes para antecipar qual será essa abstração.

Quando seus testes dependem de interfaces, você pode refatorar o código subjacente com abandono imprudente. Os testes verificam o bom comportamento contínuo da interface e as alterações no código subjacente não forçam a reescrita dos testes. Depender intencionalmente das interfaces permite que você use testes para adiar decisões de design com segurança e sem penalidades.

Suportando Abstrações

Quando mais informações finalmente chegarem e você tomar a próxima decisão de design, você alterará o código de forma a aumentar seu nível de abstração. Aqui reside outro dos benefícios dos testes de design.

Um bom design progride naturalmente em direção a pequenos objetos independentes que dependem de abstrações. O comportamento de uma aplicação bem projetada torna-se gradativamente o resultado de interações entre essas abstrações. As abstrações são componentes de design maravilhosamente flexíveis, mas as melhorias que elas fornecem têm um pequeno custo: embora cada abstração individual possa ser fácil de entender, não existe um único lugar no código que torne óbvio o comportamento do todo.

À medida que a base de código se expande e o número de abstrações aumenta, os testes tornam-se cada vez mais necessários. Existe um nível de abstração de design em que é quase impossível fazer qualquer alteração com segurança, a menos que o código tenha testes. Os testes são o seu registro da interface de cada abstração e, como tal, são a parede nas suas costas. Eles permitem adiar decisões de design e criar abstrações com qualquer profundidade útil.

Expondo Falhas de Design

O próximo benefício dos testes é que eles expõem falhas de design no código subjacente. Se um teste exigir uma configuração complicada, o código espera muito contexto. Se o teste de um objeto arrasta vários outros para a mistura, o código tem muitas dependências. Se o teste for difícil de escrever, outros objetos acharão o código difícil de reutilizar.

Os testes são o canário na mina de carvão; quando o design é ruim, os testes são difíceis.

O inverso, no entanto, não é garantido que seja verdadeiro. Testes caros não significam necessariamente que o aplicativo seja mal projetado. É tecnicamente possível escrever testes ruins para códigos bem projetados. Portanto, para que os testes reduzam seus custos, tanto o aplicativo subjacente quanto os testes devem ser bem projetados.

Seu objetivo é obter todos os benefícios dos testes pelo menor custo possível. A melhor maneira de atingir esse objetivo é escrever testes pouco acoplados apenas sobre as coisas que importam.

Sabendo o que testar A maioria dos

programadores escreve testes demais. Isto nem sempre é óbvio porque em muitos casos o custo destes testes desnecessários é tão elevado que os programadores envolvidos desistiram completamente dos testes. Não é que eles não tenham testes. Eles têm um conjunto de testes grande, mas desatualizado; simplesmente nunca funciona. Uma maneira simples de obter melhor valor dos testes é escrever menos deles. A maneira mais segura de fazer isso é testar tudo apenas uma vez e no local adequado.

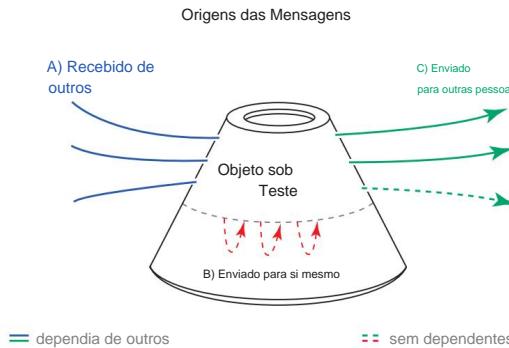


Figura 9.1 Os objetos em teste são como cápsulas espaciais, as mensagens ultrapassam os seus limites.

A remoção da duplicação dos testes reduz o custo de alteração dos testes em reação às alterações do aplicativo, e colocar os testes no lugar certo garante que eles serão forçados a mudar apenas quando for absolutamente necessário. Destilar seus testes até a essência requer ter uma ideia muito clara sobre o que você pretende testar, uma ideia que pode ser derivada de princípios de design que você já conhece.

Pense em uma aplicação orientada a objetos como uma série de mensagens passando entre um conjunto de caixas pretas. Lidar com cada objeto como uma caixa preta impõe restrições sobre o que os outros podem saber e limita o conhecimento público sobre qualquer objeto às mensagens que ultrapassam os seus limites.

Objetos bem projetados têm limites muito fortes. Cada um é como a cápsula espacial mostrada na Figura 9.1. Nada do lado de fora pode ver para dentro, nada do lado de dentro pode ver para fora e apenas algumas mensagens explicitamente acordadas podem passar pelas câmaras de descompressão predefinidas.

Essa ignorância intencional do interior de todos os outros objetos está no cerne do design. Lidar com objetos como se fossem apenas e exatamente as mensagens às quais eles respondem permite projetar uma aplicação mutável, e é a sua compreensão da importância dessa perspectiva que permite criar testes que forneçam o máximo benefício com o mínimo custo.

Os princípios de design que você está aplicando em seu aplicativo também se aplicam aos seus testes. Cada teste é apenas outro objeto de aplicação que precisa usar uma classe existente. Quanto mais o teste fica acoplado a essa classe, mais emaranhados os dois se tornam e mais vulnerável o teste fica a ser forçado a mudar desnecessariamente.

Você não deve apenas limitar os acoplamentos, mas os poucos permitidos devem ser para estabilizar as coisas. A coisa mais estável sobre qualquer objeto é sua interface pública; segue-se logicamente que os testes que você escreve devem ser para mensagens definidas em interfaces públicas. O



Figura 9.2 A mensagem enviada por um objeto é a mensagem recebida por outro.

os testes mais caros e menos úteis são aqueles que abrem buracos na contenção de um objeto paredes por acoplamento a detalhes internos instáveis. Esses testes ansiosos não provam nada sobre a exatidão geral de uma aplicação, mas ainda assim aumentam os custos porque romper com cada refatoração da classe subjacente.

Os testes devem concentrar-se nas mensagens recebidas ou enviadas que atravessam um limites do objeto. As mensagens recebidas constituem a interface pública do receptor objeto. As mensagens de saída, por definição, chegam a outros objetos e, portanto, fazem parte da interface de algum outro objeto, conforme ilustrado na Figura 9.2.

Na Figura 9.2, as mensagens que chegam em Foo constituem a interface pública de Foo . Foo é responsável por testar sua própria interface e faz isso fazendo afirmações sobre os resultados que essas mensagens retornam. Testes que fazem afirmações sobre o os valores retornados pelas mensagens são testes de estado. Tais testes geralmente afirmam que os resultados retornado por uma mensagem igual a um valor esperado.

A Figura 9.2 também mostra Foo enviando mensagens para Bar. Uma mensagem enviada por Foo para Bar está saindo de Foo , mas entrando em Bar. Esta mensagem faz parte da interface pública da Ordem dos Advogados e todos os testes de estado devem, portanto, ser confinados à Ordem dos Advogados. Foo não precisa e deveria não, teste essas mensagens de saída quanto ao estado. A regra geral é que os objetos devem fazer afirmações sobre o estado apenas para mensagens em suas próprias interfaces públicas. Seguindo esta regra confina os testes de valores de retorno de mensagens em um único local e remove duplicações desnecessárias, secando seus testes e reduzindo os custos de manutenção.

O fato de você não precisar testar o estado das mensagens de saída não significa as mensagens não precisam de nenhum teste. Existem dois tipos de mensagens enviadas, e um dos eles exigem um tipo diferente de teste.

Algumas mensagens enviadas não têm efeitos colaterais e, portanto, importam apenas para o seu remetentes. O remetente certamente se preocupa com o resultado que recebe (por que outro motivo enviar a mensagem?), mas nenhuma outra parte do aplicativo se importa se a mensagem for enviada. Extrovertido mensagens como esta são conhecidas como *consultas* e não precisam ser testadas pelo remetente objeto. As mensagens de consulta fazem parte da interface pública do seu receptor, que já implementa todos os testes de estado necessários.

No entanto, muitas mensagens enviadas têm efeitos colaterais (um arquivo é gravado, um registro do banco de dados é salvo, uma ação é executada por um observador) sobre os quais seu aplicativo

depende. Essas mensagens são *comandos* e é responsabilidade do objeto remetente provar que foram enviadas corretamente. Provar que uma mensagem foi enviada é um teste de comportamento, não de estado, e envolve afirmações sobre o número de vezes e com quais argumentos a mensagem é enviada.

Aqui estão, então, as diretrizes sobre o que testar: As mensagens recebidas devem ser testadas quanto ao estado que retornam. As mensagens de comando de saída devem ser testadas para garantir que sejam enviadas. As mensagens de consulta enviadas não devem ser testadas.

Contanto que os objetos do seu aplicativo tratem uns dos outros estritamente por meio de interfaces públicas, seus testes não precisarão saber mais nada. Ao testar esse conjunto mínimo de mensagens, nenhuma alteração no comportamento privado de qualquer objeto poderá afetar qualquer teste. Quando você testa mensagens de comando de saída apenas para provar que foram enviadas, seus testes fracamente acoplados podem tolerar alterações no aplicativo sem serem forçados a fazer alterações. Contanto que as interfaces públicas permaneçam estáveis, você poderá escrever testes uma vez e eles o manterão seguro para sempre.

Sabendo quando testar Você deve

escrever os testes primeiro, sempre que fizer sentido fazê-lo.

Infelizmente, julgar quando faz sentido fazer isso pode ser um desafio para designers novatos, tornando este conselho pouco útil. Os novatos geralmente escrevem códigos muito acoplados; eles combinam responsabilidades não relacionadas e vinculam muitas dependências em cada objeto. Suas aplicações são tapeçarias de código emaranhado, onde nenhum objeto vive isolado. É muito difícil testar retroativamente esses aplicativos porque os testes são reutilizados e esse código não pode ser reutilizado.

Escrever testes primeiro força um mínimo de capacidade de reutilização a ser incorporado em um objeto desde o seu início; caso contrário, seria impossível escrever testes. Portanto, os designers novatos são mais bem atendidos escrevendo o código de teste primeiro. Sua falta de habilidades de design pode tornar isso extremamente difícil, mas se eles perseverarem, pelo menos terão código testável, algo que de outra forma pode não ser verdade.

Esteja avisado, entretanto, que escrever testes primeiro não substitui e não garante uma aplicação bem projetada. A capacidade de reutilização resultante do teste primeiro é uma melhoria em relação a nada, mas o aplicativo resultante ainda pode ficar muito aquém de um bom design. Iniciantes bem-intencionados geralmente escrevem testes caros e duplicados em torno de códigos confusos e fortemente acoplados. É uma triste verdade que o código mais complexo geralmente seja escrito pela pessoa menos qualificada. Isto não reflete uma complexidade inata da tarefa subjacente, mas sim uma falta de experiência por parte do programador.

Os programadores novatos ainda não possuem as habilidades necessárias para escrever códigos simples.

As aplicações extremamente complicadas que esses novatos produzem devem ser vistas como triunfos da perseverança; é um milagre que esses aplicativos funcionem. O código é duro. As aplicações são difíceis de alterar e cada refatoração quebra todos os testes. Este elevado custo da mudança pode facilmente iniciar uma espiral descendente de produtividade que é desanimadora para todos os envolvidos. As alterações se espalham por todo o aplicativo e o custo de manutenção dos testes faz com que pareçam caros em relação ao seu valor.

Se você é um novato e está nesta situação, é importante manter a fé no valor de testes. Feito na hora certa e nas quantidades certas, testando e escrevendo código teste primeiro, reduzirá seus custos gerais. Obter esses benefícios requer a aplicação de princípios de design orientado a objetos em todos os lugares, tanto no código da sua aplicação quanto no código em seus testes. Seu novo conhecimento de design já torna mais fácil escrever código testável, a maior parte do restante deste capítulo ilustra como aplicar esses princípios de design durante a construção de testes. Porque bem desenhado os aplicativos são fáceis de alterar e testes bem projetados podem muito bem evitar alterações no geral, essas melhorias gerais de design compensam dramaticamente.

Designers experientes obtêm melhorias mais sutis desde os primeiros testes. Não é que eles não podem se beneficiar disso ou que nunca descobrirão algo inesperado seguindo seus ditames, em vez disso, os ganhos obtidos com a reutilização forçada são aqueles que eles já tem. Esses programadores já escrevem código reutilizável e fracamente acoplado; testes agregar valor de outras maneiras.

Não é incomum que designers experientes “agravem” um problema, isto é, façam experimentos onde eles apenas escrevem código. Esses experimentos são exploratórios, para problemas cuja solução são incertos. Depois que a clareza é obtida e um design é sugerido, esses programadores voltam a testar primeiro o código de produção.

Seu objetivo geral é criar aplicativos bem projetados que tenham testes aceitáveis cobertura. A melhor forma de atingir esse objetivo varia de acordo com os pontos fortes e a experiência do programador.

Esta licença para usar seu próprio julgamento não é permissão para pular o teste. Mal código projetado sem testes é apenas código legado que não pode ser testado. Não superestime seus pontos fortes e use uma visão inflada de si mesmo como desculpa para evitar testes. Embora às vezes faça sentido escrever um pouco de código à moda antiga, você deve errar no lado do teste primeiro.

Sabendo como testar

Qualquer um pode criar um novo framework de testes Ruby e às vezes parece que todos tem. A próxima estrutura novíssima pode conter um recurso sem o qual você simplesmente não consegue viver; se você entende os custos e benefícios, fique à vontade para escolher qualquer estrutura que mais lhe convier.

Teste Intencional

No entanto, existem muitas boas razões para permanecer dentro do padrão de testes. Os frameworks mais utilizados têm o melhor suporte. Eles são atualizados rapidamente para garantir a compatibilidade com novas versões do Ruby (e do Rails) e, portanto, não apresentam nenhum obstáculo para se manterem atualizados. Sua grande base de usuários os incentiva a manter a compatibilidade com versões anteriores; é improvável que eles mudem de forma a forçar a reescrita de todos os seus testes. E como são amplamente adotados, é fácil encontrar programadores que tenham experiência em usá-los.

No momento em que este livro foi escrito, as estruturas principais eram MiniTest, de Ryan Davis e seattle.rb e empacotado com Ruby a partir da versão 1.9, e RSpec, de David Chelimsky e a equipe RSpec. Essas estruturas têm filosofias diferentes e, embora você possa naturalmente inclinar-se para uma ou outra, ambas são escolhas excelentes.

Você não deve apenas escolher uma estrutura, mas também lidar com estilos alternativos de teste: Desenvolvimento Orientado a Testes (TDD) e Desenvolvimento Orientado a Comportamento (BDD). Aqui a decisão não é tão clara. TDD e BDD podem parecer estar em oposição, mas são melhor vistos como um continuum como a Figura 9.3, onde seus valores e experiência ditam a escolha de onde se posicionar.

Ambos os estilos criam código escrevendo testes primeiro. O BDD adota uma abordagem de fora para dentro, criando objetos nos limites de um aplicativo e trabalhando para dentro, zombando conforme necessário para fornecer objetos ainda não escritos. O TDD adota uma abordagem de dentro para fora, geralmente começando com testes de objetos de domínio e depois reutilizando esses objetos de domínio recém-criados nos testes de camadas adjacentes de código.

A experiência ou inclinação anterior pode tornar um estilo mais adequado para você do que o outro, mas ambos são completamente aceitáveis. Cada um tem custos e benefícios, alguns dos quais serão explorados nas próximas seções sobre redação de testes.

Ao testar, é útil pensar nos objetos do seu aplicativo divididos em duas categorias principais. A primeira categoria contém o objeto que você está testando,

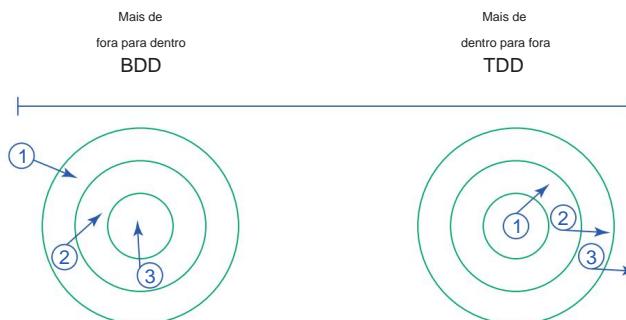


Figura 9.3 O BDD e o TDD devem ser vistos como um continuum.

referido a partir de agora como o *objeto em teste*. A segunda categoria contém todo o resto.

Seus testes obviamente devem saber coisas sobre a primeira categoria, ou seja, sobre objeto sob teste, mas eles devem permanecer tão ignorantes quanto possível sobre o segundo. Finja que o resto da aplicação é opaco, que a única informação disponível durante o teste é o que pode ser obtido olhando para o objeto sob teste.

Depois de direcionar seu foco de teste para o objeto específico em teste, você precisa escolher um ponto de vista de teste. Seus testes *podem* ficar completamente dentro do objeto em teste, com acesso efetivo a todos os seus internos. Esta é uma má ideia, no entanto, porque permite que o conhecimento que deveria ser privado do objeto vaze para o teste, aumentando o acoplamento entre eles e aumentando a probabilidade de que mudanças nos o código exigirão alterações nos testes. É melhor que os testes assumam um ponto de vista que visa ao longo das bordas do objeto em teste, onde eles podem saber apenas sobre mensagens que vem e vai.

Estrutura MiniTest

Os testes neste capítulo são escritos usando MiniTest. Isto não é um endosso de uma estrutura em detrimento de outra, mas sim um reconhecimento do fato de que exemplos escritos em MiniTest serão executados em qualquer lugar com Ruby 1.9 ou superior instalado. Você pode duplicar e experimentar estes exemplos sem instalar software adicional.

No momento em que você ler este capítulo, o MiniTest pode ter mudado. Perfeitos estranhos podem muito bem ter melhorado este software e oferecido essas melhorias gratuitamente; essa é a vida do desenvolvedor de código aberto. Sem considerar de como o MiniTest pode ter evoluído, os princípios ilustrados abaixo são verdadeiros. Não se distraia com mudanças na sintaxe; concentrar-se em compreender os objetivos subjacentes dos testes. Depois de entender esses objetivos, você pode alcançá-los por meio de qualquer estrutura de teste.

Testando mensagens recebidas

As mensagens recebidas constituem a interface pública de um objeto, a face que ele apresenta ao mundo. Essas mensagens precisam de testes porque outros objetos da aplicação dependem de sua assinaturas e nos resultados que eles retornam.

Esses primeiros testes usam código dos exemplos do Capítulo 3, Gerenciando Dependências. A seguir está um lembrete dessas classes Wheel and Gear , como eram quando emaranhadas junto. Gear cria uma instância da classe Wheel bem dentro de seu gear_inches método, na linha 24 abaixo.

Observação

O restante deste capítulo contém testes para código que apareceu anteriormente neste livro. Esses exemplos de código serviu anteriormente para explicar os princípios da orientação a objetos projeto; aqui eles ilustrarão como testar diferentes componentes do design. Os testes a seguir não cobrem todos linha de código que você viu, mas eles testam todos os conceitos você aprendeu neste livro.

Roda de 1 classe

```
2 attr_reader :rim, :tire
3 def inicializar (aro, pneu)
4     @aro = aro
5     @pneu = pneu
6 fim
7
8 diâmetro definido
9     aro + (pneu * 2)
10 fim
11 #...
12 fim
13
```

Equipamento de 14 classes

```
15 attr_reader :chainring, :cog, :rim, :tire
16 def inicializar (args)
17     @chainring = args[:chainring]
18     @cog = argumentos[:cog]
19     @rim = args[:rim]
20     @pneu = args[:pneu]
21 fim
22
23 def gear_inches
24     relação * Roda.nova(aro, pneu).diâmetro
25 fim
```

```

26
Razão de definição de 27
28 coroa / cog.to_f
29 fim
30 #...
31 fim

```

A Tabela 9.1 mostra as mensagens (além daquelas que retornam atributos simples) que cruzar os limites desses objetos. A roda responde a uma mensagem recebida, diâmetro (que por sua vez é enviado ou enviado pelo Gear) e o Gear responde a dois sinais recebidos mensagens, gear_inches e proporção.

O parágrafo inicial desta seção afirma que toda mensagem recebida faz parte da interface pública de um objeto e, portanto, deve ser testado. Agora é hora de adicionar um leve advertência a esta regra.

Excluindo interfaces não utilizadas

As mensagens recebidas devem ter dependentes. Como você pode ver na Tabela 9.1, isso é verdadeiro para diâmetro, gear_inches e proporção onde estão as mensagens recebidas. Algun objeto *diferente do implementador original* depende de cada uma dessas mensagens.

Se você desenhar esta tabela para o objeto em teste e encontrar uma suposta entrada mensagem que não tem dependentes, você deve encarar essa mensagem com muita desconfiança. Qual é o propósito de implementar uma mensagem que ninguém envia? Não é realmente *chegado*, é uma implementação especulativa que cheira a adivinhação sobre o futuro e antecipa claramente requisitos que não existem.

Não teste uma mensagem recebida que não tenha dependentes; delete isso. Seu aplicativo é melhorado eliminando impiedosamente o código que não está sendo usado ativamente. Tal código é fluxo de caixa negativo, acrescenta encargos de testes e manutenção, mas não fornece valor. Excluir o código não utilizado economiza dinheiro agora, se você não fizer isso, você deve testá-lo.

Tabela 9.1 Mensagens recebidas e enviadas por objeto.

Objeto	Mensagens recebidas	Mensagens de saída	Tem Dependentes?
Diâmetro da roda			Sim
Engrenagem		diâmetro	Não
	engrenagem_polegadas		Sim
razão			Sim

Supere qualquer relutância que você sinta; praticar esta poda lhe ensinará seu valor. Até o momento em que você estiver completamente convencido da correção dessa estratégia, você pode se consolar com o conhecimento de que, no limite, você pode recuperar o código excluído do controle de revisão. Independentemente de você fazer isso com alegria ou com dor, exclua o código. O código não utilizado custa mais para manter do que para recuperar.

Provando a Interface Pública As mensagens

recebidas são testadas fazendo afirmações sobre o valor, ou estado, que sua invocação retorna. O primeiro requisito para testar uma mensagem recebida é provar que ela retorna o valor correto em todas as situações possíveis.

O código a seguir mostra um teste do método do diâmetro da roda . A linha 4 cria um instância da Roda e a linha 6 afirma que esta Roda tem um diâmetro de 29.

```

1 classe WheelTest < MiniTest::Unit::TestCase
2
3 def test_calculates_diameter roda = Wheel.new(26,
4     1,5)
5
6     assert_in_delta(29,
7         diâmetro da roda,
8         0,01)
9 fim
10 fim

```

Este teste é extremamente simples e invoca muito pouco código. Wheel não tem dependências ocultas, portanto nenhum outro objeto de aplicativo é criado como efeito colateral da execução deste teste. O design do Wheel permite testá-lo independentemente de qualquer outra classe em sua aplicação.

Testar Gear é um pouco mais interessante. O Gear requer mais alguns argumentos do que o Wheel, mas mesmo assim a estrutura geral desses dois testes é muito semelhante. No teste gear_inches abaixo, a linha 4 cria uma nova instância do Gear e a linha 10 faz afirmações sobre os resultados do método.

```

1 classe GearTest < MiniTest::Unit::TestCase
2
3 def test_calcula_gear_inches
4     engrenagem =
5         Gear.new(coroa: 52,

```

```

6           11,
7   engrenagem: aro: 26,
8   pneu:      1.5)
9
10 assert_in_delta(137,1,
11     gear.gear_inches, 0,01)
12
13 fim
14 fim

```

Este novo teste gear_inches se parece muito com o teste de diâmetro da roda , mas não se preocupe enganado pelas aparências. Este teste tem emaranhados que o teste de diâmetro não ter. A implementação de gear_inches no Gear cria e usa incondicionalmente outro objeto, Roda. Gear e Wheel são acoplados no código e nos testes, embora não seja óbvio aqui.

O fato de o método gear_inches do Gear criar e usar outro objeto afeta a duração do teste e a probabilidade de sofrer consequências indesejadas como resultado de alterações em partes não relacionadas do aplicativo. O acoplamento que cria esse problema, entretanto, está escondido dentro do Gear e é totalmente invisível neste teste. O objetivo do teste é provar que gear_inches retorna o resultado correto e certamente atende a esse requisito, mas a forma como o código subjacente é estruturado adiciona risco oculto.

Se a criação de rodas for cara, o teste de equipamento paga esse custo, mesmo que não tem interesse na Wheel. Se a engrenagem estiver correta, mas a roda estiver quebrada, o teste de engrenagem pode falhar de forma enganosa, em um local muito distante do código que você está tentando testar.

Os testes são executados mais rapidamente quando executam o mínimo de código e o volume de recursos externos. o código invocado por um teste está diretamente relacionado ao seu design. Um aplicativo construído de objetos fortemente acoplados e carregados de dependência é como uma tapeçaria onde puxar um o fio arrasta todo o tapete. Quando objetos fortemente acoplados são testados, um teste de um objeto executa código em muitos outros. Se o código fosse tal que Wheel também fosse acoplado a outros objetos, esse problema é ampliado; executar o teste do Gear seria em seguida, crie uma grande rede de objetos, qualquer um dos quais pode quebrar de uma forma enlouquecedora maneira confusa.

Esses problemas se manifestam, mas não são exclusivos, dos testes. Porque os testes são a primeira reutilização de código, esse problema é apenas um prenúncio do que está por vir para o seu aplicação como um todo.

Isolando o objeto em teste

Gear é um objeto simples, mas as tentativas de testar seu método gear_inches já foram desenterrou complexidade oculta. O objetivo deste teste é garantir que as polegadas das engrenagens sejam calculadas corretamente, mas acontece que a execução de gear_inches depende do código dos objetos, além do Gear.

Isto expõe um problema de design mais amplo; quando você não pode testar o Gear isoladamente, é um mau presságio para o futuro. Esta dificuldade em isolar o Gear para testes revela que é vinculado a um contexto específico, que impõe limitações que irão interferir na reutilização.

O Capítulo 3 quebrou essa ligação removendo a criação de Wheel do Gear. Aqui está uma cópia do código que fez essa transição; Gear agora espera ser injetado com um objeto que entende o diâmetro.

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel
3 def inicializar (args)
4     @chainring = args[:chainring]
5     @cog       = argumentos[:cog]
6     @roda= args[:roda]
7 fim
8
9 def gear_inches
10    # O objeto na variável'wheel'
11    # desempenha o papel de 'Diametrizável'.
12    relação * diâmetro da roda
13 fim
14
Proporção de 15 defesas
16     coroa / cog.to_f
17 fim
18#...
19 fim

```

Esta transição de código é paralela a uma transição de pensamento. Gear não se importa mais sobre a classe do objeto injetado, ele apenas espera que implemente o diâmetro. O método do diâmetro faz parte da interface pública de uma função, que pode razoavelmente ser denominado Diametrizável.

Agora que o Gear está desacoplado do Wheel, você deve injetar uma instância do Diametrizável durante cada criação de equipamento . No entanto, como Wheel é o único

classe de aplicativo que desempenha essa função, suas opções de tempo de execução serão severamente limitadas. Na real vida, como o código existe atualmente, cada Gear que você criar será necessariamente injetado com uma instância de Wheel.

Por mais circular que pareça, injetar uma roda na engrenagem não é o mesmo que injetar um diâmetro. O código do aplicativo parece exatamente o mesmo, é verdade, mas seu significado lógico difere. A diferença não está nos caracteres que você digita, mas em seus pensamentos sobre o que eles significam. Liberar sua imaginação de um apego à classe do objeto recebido abre possibilidades de design e teste que caso contrário, não estarão disponíveis. Pensar no objeto injetado como uma instância de sua função oferece mais opções sobre que tipo de Diameterizable injetar no Gear durante seus testes.

Um diâmetro possível é, obviamente, Roda, porque implementa claramente a interface correta. O próximo exemplo faz esta escolha muito prosaica; ele atualiza o teste existente para acomodar as alterações no código, injetando uma instância de Roda (linha 6) durante o teste.

```

1 classe GearTest < MiniTest::Unit::TestCase
2 def test_calcula_gear_inches
3     engrenagem = engrenagem.new(
4         coroa: 52,
5         engrenagem:           11,
6         roda:                 Roda.novo(26, 1,5))
7
8     assert_in_delta(137,1,
9                     gear.gear_inches, 0,01)
10
11 fim
12 fim

```

Usar uma roda para o Diameterizable injetado resulta em um código de teste que reflete exatamente o aplicativo. É agora óbvio, tanto na aplicação como nos testes, que Gear está usando Wheel. O acoplamento invisível entre essas classes foi publicamente expor.

Este teste é rápido o suficiente, mas essa velocidade adequada é acidental. Não é que o teste gear_inches foi cuidadosamente isolado e, portanto, desacoplado de outro código; de jeito nenhum, só que todo o código acoplado a esse teste roda rapidamente também.

Observe também que não é óbvio aqui (ou em qualquer outro lugar) que a Roda esteja desempenhando o papel Diametrizável . O papel é virtual, está tudo na sua cabeça. Nada no código orienta futuros mantenedores a pensarem em Wheel como um Diameterizável.

Porém, apesar da invisibilidade do papel e desse acoplamento à Roda, a estruturação o teste desta forma tem uma vantagem muito real, como mostra a próxima seção.

Injetando Dependências Usando Classes Quando o código

do seu teste usa os mesmos objetos de colaboração que o código do seu aplicativo, seus testes sempre falham quando deveriam. O valor disto não pode ser subestimado.

Aqui está um exemplo simples. Imagine que a interface pública do Diameterizable mude. Outro programador entra na classe Wheel e altera o nome do método de diâmetro para largura, conforme mostrado na linha 8 abaixo.

Roda de 1 classe

```

2 attr_reader :aro,: pneu 3 def inicializar (aro,
pneu)
4      @aro          = aro
5      @pneu         = pneu
6 fim
7
8 def largura # <—— costumava ser o 'diâmetro' do aro + (pneu * 2)
9
10 fim
11#...
12 fim

```

Imagine ainda que este programador não conseguiu atualizar o nome da mensagem enviada no Gear. O Gear ainda envia o diâmetro em seu método gear_inches , como você pode ver neste lembrete do código atual do Gear :

```

1 classe de equipamento
2   #...
3 def gear_inches
4     proporção * wheel.diameter # <—— obsoleto
5 fim
6 fim

```

Como o teste Gear injeta uma instância de Wheel e Wheel implementa largura , mas A engrenagem envia o diâmetro, o teste agora falha:

```

1 engrenagem
2 ERRO test_calcula_gear_inches
3 método indefinido 'diâmetro'
```

Esta falha não é surpreendente, é exatamente o que deveria acontecer quando dois objetos concretos colaboram e o receptor de uma mensagem muda, mas o remetente não. A roda tem mudou e, como resultado, o Gear precisa mudar. Este teste falha como deveria.

O teste é simples e a falha óbvia porque o código é muito concreto, mas como todas as concreções funciona apenas para este caso específico. Aqui, para este código, o teste acima é bom o suficiente, mas há outras situações em que é melhor você localizar e teste a abstração.

Um exemplo mais extremo ilumina o problema. Se houver centenas de Diameterizáveis, como você decide qual é a maior intenção reveladora de injetar durante o teste? E se os Diameterizáveis forem extremamente caros, como você evita executando muitos códigos desnecessários e demorados? O bom senso sugere que se A roda é a única Diametrizável e é rápida o suficiente, o teste deve apenas injetar um Roda, mas se a sua escolha for menos óbvia?

Injetando Dependências como Funções

A classe Wheel e a função Diameterizable estão tão estreitamente alinhadas que é difícilvê-los como conceitos separados, mas entendendo o que aconteceu no teste anterior exige fazer uma distinção. Gear e Wheel têm relacionamentos com um terceiro coisa, o papel Diameterizável . Como você pode ver na Figura 9.4, Diameterizável é dependia do Gear e implementado pela Wheel.

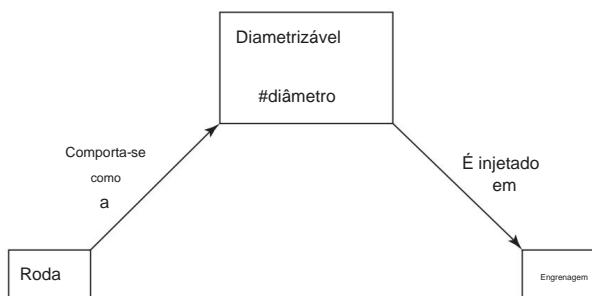


Figura 9.4 Engrenagem depende de Diametrizável ; Roda o implementa.

Este papel é uma abstração da ideia de que objetos díspares podem ter diâmetros.

Tal como acontece com todas as abstrações, é razoável esperar que este papel abstrato seja mais estável do que a concretização de onde veio. No entanto, no caso específico acima, o oposto é verdadeiro.

Existem dois lugares no código onde um objeto depende do conhecimento de `Diameterizable`. Primeiro, o `Gear` pensa que conhece a interface do `Diameterizable`; isto é, acredita que pode enviar diâmetro ao objeto injetado. Segundo, o código que criou o objeto a ser injetado acredita que `Wheel` implementa esta interface; isto é, espera que `Wheel` implemente o diâmetro. Agora que o `Diameterizable` mudou, há um problema. `Wheel` foi atualizado para implementar a nova interface, mas infelizmente o `Gear` ainda espera a antiga.

O objetivo da injeção de dependência é que ela permite substituir diferentes classes concretas sem alterar o código existente. Você pode montar um novo comportamento criando novos objetos que desempenhem funções existentes e injetando esses objetos onde essas funções são esperadas. O design orientado a objetos diz para você injetar dependências porque acredita que classes concretas específicas irão variar mais do que esses papéis ou, inversamente, os papéis serão mais estáveis do que as classes das quais foram abstraídos.

Infelizmente, o oposto simplesmente aconteceu. Neste exemplo não foi a classe do objeto injetado que mudou, foi a interface do role. Ainda é correto injetar uma roda, mas agora é incorreto enviar a mensagem do diâmetro para essa roda.

Quando um papel tem um único interveniente, esse interveniente concreto e o papel abstrato estão tão estreitamente alinhados que as fronteiras entre eles são facilmente confusas e é um facto prático que por vezes esta indefinição não importa. Neste caso, `Wheel` é o único player de `Diameterizable` e você não espera ter outros no momento. Se as rodas forem baratas, injetar uma roda real terá pouco efeito negativo nos seus testes.

Quando o código do aplicativo só pode ser escrito de uma maneira, espelhar esse arranjo costuma ser a maneira mais eficaz de escrever testes. Fazer isso permite que os testes falhem corretamente, independentemente de a concreção (o nome da classe `Wheel`) ou a abstração (a interface para o método do diâmetro) mudar.

No entanto, isso nem sempre é verdade. Às vezes, há forças em ação que levam você a querer renunciar ao uso do `Wheel` em seus testes. Se o seu aplicativo contém muitos `Diameterizables` diferentes, você pode querer criar um idealizado para que seus testes transmitam claramente a ideia dessa função. Se todos os `Diameterizables` forem caros, você pode falsificar um barato para fazer seus testes rodarem mais rápido. Se você estiver fazendo BDD, seu aplicativo pode ainda não conter nenhum objeto que desempenhe essa função; você pode ser forçado a fabricar *algo* apenas para escrever o teste.

Criando Duplas de Teste

O próximo exemplo explora a ideia de criar um objeto falso, ou *duplo de teste*, para jogar o Papel diametrizável . Para este teste, suponha que a interface do Diameterizable tenha revertido para o método do diâmetro original e que o diâmetro seja novamente implementado corretamente pelo Wheel e enviado pelo Gear. A linha 2 abaixo cria um DiameterDouble falso . A Linha 13 injeta essa falsificação no Gear.

```

1 # Crie um jogador com a função 'Diameterizável'
2 classes DiameterDouble
3 diâmetro definido
4     10
5 fim
6 fim
7
8 classes GearTest < MiniTest::Unit::TestCase
9 def test_calcula_gear_inches
10     engrenagem = engrenagem.new(
11         coroa: 52,
12         11,
13         engrenagem: roda:  DiâmetroDuplo.novo)
14
15     assert_in_delta(47,27,
16                     gear.gear_inches, 0,01)
17
18 fim
19 fim

```

Um teste duplo é uma instância estilizada de um role player usada exclusivamente para testes. Duplas como esta são muito fáceis de fazer; nada impede você de criar um para todas as situações possíveis. Cada variação é como um esboço de artista. Ele enfatiza um único característica interessante e permite que outros detalhes do objeto subjacente retrocedam ao fundo.

Este diâmetro *de pontas duplas* , ou seja, implementa uma versão de diâmetro que retorna uma resposta automática. DiameterDouble é bastante limitado, mas isso é tudo apontar. O fato de sempre retornar 10 para o diâmetro é perfeito. Este retorno esboçado value fornece uma base confiável sobre a qual construir o teste.

Muitas estruturas de teste possuem maneiras integradas de criar valores duplos e de retorno de stub. Esses mecanismos especializados podem ser úteis, mas para testes simples, é bom usá-los objetos Ruby simples e antigos, como no exemplo acima.

DiameterDouble *não* é uma simulação. É fácil adquirir o hábito de usar a palavra “mock” para descrever esse duplo, mas mocks são algo completamente diferente e serão abordados mais adiante neste capítulo, na seção Testando Mensagens de Saída.

Injetar esse duplo desacopla o teste Gear da classe Wheel . Já não importa se Wheel for lento porque DiameterDouble é sempre rápido. Este teste funciona apenas tudo bem, como mostra a execução:

1	Teste de Engrenagem
2	PASSAR test_calcula_gear_inches

Este teste utiliza um teste duplo e, portanto, é simples, rápido, isolado e revelador de intenção; O que poderia dar errado?

Vivendo o sonho

Imagine agora que o código sofre as mesmas alterações de antes:

A interface do Diameterizable muda de diâmetro para largura e a Roda fica atualizado, mas o Gear não. Essa mudança mais uma vez quebra o aplicativo.

Lembre-se de que o teste anterior do Gear (que injetou uma roda em vez de usar um double) percebeu esse problema imediatamente e começou a falhar com um método indefinido erro de ‘diâmetro’ .

Agora que você está injetando DiameterDouble, aqui está o que acontece quando você executa novamente o teste:

1	Teste de Engrenagem
2	PASSAR test_calcula_gear_inches

O teste *continua passando* mesmo que o aplicativo esteja definitivamente quebrado. Esta aplicação não pode funcionar; A engrenagem envia o diâmetro , mas a roda implementa a largura.

Você criou um universo alternativo, no qual os testes relatam alegremente que está tudo bem, apesar de a aplicação ser manifestamente incorreta. A possibilidade de criar este universo é o que faz com que alguns avisem que a repreensão (e a zombaria) faz testes frágeis. No entanto, como sempre é verdade, a culpa aqui é do programador e não da ferramenta. Escrever um código melhor requer a compreensão da causa raiz disso problema, o que por sua vez exige um olhar mais atento aos seus componentes.

O aplicativo contém uma função Diametrizable . Esta função originalmente tinha um jogador, Roda. Quando a GearTest criou o DiameterDouble, introduziu *um segundo jogador do papel*. Quando a interface de uma função muda, todos os participantes da função devem adotar

a nova interface. É fácil, no entanto, ignorar os atores que foram construídos especificamente para testes e foi exatamente isso que aconteceu aqui. A roda foi atualizada com a nova interface, mas o DiameterDouble não.

Usando testes para documentar funções

Não é de admirar que este problema ocorra; o papel é quase invisível. Não há lugar no aplicativo onde você pode apontar o dedo e dizer “Isso define Diameterizable”.

Ao lembrar que o papel existe é um desafio, esquecer que a prova dobra jogar é inevitável.

Uma forma de aumentar a visibilidade da função é afirmar que Wheel a desempenha. A linha 6 abaixo faz só isso; ele documenta a função e prova que o Wheel implementa corretamente sua interface.

```

1 classe WheelTest < MiniTest::Unit::TestCase
2 configuração de definição
3     @roda = Roda.new(26, 1,5)
4 fim
5
6 def test_implements_the_diameterizable_interface
7     assert_respond_to(@wheel,:diâmetro)
8 fim
9
10 def test_calcula_diâmetro
11     roda = Roda.new(26, 1,5)
12
13     assert_in_delta(29,
14                     diâmetro da roda,
15                     0,01)
16 fim
17 fim

```

O teste implements_the_diameterizable_interface introduz a ideia de testa funções, mas não é uma solução completamente satisfatória. Na verdade, é lamentavelmente incompleto. Primeiro, ele não pode ser compartilhado com outros Diameterizáveis. Outros jogadores de essa função teria que duplicar esse teste. Em seguida, não ajuda em nada com o problema de “viver o sonho” do teste do Gear . A afirmação de Wheel de que desempenha esse papel não impede que o DiameterDouble do Gear se torne obsoleto e permita o teste gear_inches para passar erroneamente.

Felizmente, o problema de documentar e testar funções tem uma solução simples, aquele que será completamente abordado na seção subsequente, Provando a

Correção dos Patos. Por enquanto, basta reconhecer que as funções precisam de testes próprios.

O objetivo desta seção foi provar interfaces públicas testando mensagens recebidas. A roda era barata para testar. O teste original do Gear era mais caro porque dependia de um acoplamento oculto ao Wheel. Substituir esse acoplamento por uma dependência injetada em Diameterizable isolou o objeto em teste, mas criou um dilema sobre injetar um objeto real ou falso.

Esta escolha entre injetar objetos reais ou falsos tem consequências de longo alcance.

Injetar no tempo de teste os mesmos objetos usados no tempo de execução garante que os testes sejam interrompidos corretamente, mas pode levar a testes de longa execução. Alternativamente, injetar duplos pode acelerar os testes, mas deixá-los vulneráveis à construção de um mundo de fantasia onde os testes funcionam, mas a aplicação falha.

Observe que o ato de testar não forçou, por si só, uma melhoria no design.

Nada nos testes fez você remover o acoplamento e injetar a dependência.

Embora seja verdade que a abordagem de fora para dentro do BDD fornece mais orientação do que o TDD, nenhuma das práticas impede um designer ingênuo de escrever Wheel e então incorporar a criação de uma Wheel profundamente dentro do Gear. Esse acoplamento não impossibilita os testes, apenas aumenta os custos. A redução do acoplamento depende de você e depende de sua compreensão dos princípios de design.

Testando Métodos Privados

Às vezes, o objeto em teste envia mensagens para si mesmo. Mensagens enviadas para métodos de auto-invocação definidos na interface privada do receptor. Estas mensagens privadas são como árvores proverbiais caindo em florestas vazias; eles não existem, pelo menos no que diz respeito ao restante do seu aplicativo. Como os envios de métodos privados não podem ser vistos fora da caixa preta do objeto em teste, no mundo primitivo do design idealizado eles não precisam ser testados.

No entanto, o mundo real não é tão simples e esta regra simples não resolve completamente satisfazer. Lidar com métodos privados requer julgamento e flexibilidade.

Ignorando métodos privados durante testes Existem muitas razões excelentes para omitir testes de métodos privados.

Primeiro, tais testes são redundantes. Os métodos privados ficam ocultos dentro do objeto em teste e seus resultados não podem ser vistos por outras pessoas. Esses métodos privados são invocados por métodos públicos *que já possuem testes*. Um bug em um método privado certamente pode quebrar

a aplicação geral, mas essa falha sempre será exposta por um teste existente.

Testar métodos privados nunca é necessário.

Em segundo lugar, os métodos privados são instáveis. Os testes de métodos privados são, portanto, acoplados ao código do aplicativo que provavelmente mudará. Quando o aplicativo muda, os testes serão forçados a mudar por sua vez. É fácil criar uma situação em que um tempo precioso é gasto realizando manutenção contínua em testes desnecessários.

Finalmente, testar métodos privados pode induzir outras pessoas a usá-los. Os testes fornecem documentação sobre o objeto em teste. Eles contam uma história sobre como espera interagir com o mundo em geral. A inclusão de métodos privados nesta história distrai os leitores de seu objetivo principal e os incentiva a quebrar o encapsulamento e a depender desses métodos. Seus testes devem ocultar métodos privados, e não expô-los.

Removendo métodos privados da classe em teste Uma maneira de contornar todo esse problema é evitar completamente os métodos privados. Se você não possui métodos particulares, não precisa se preocupar com os testes deles.

Um objeto com muitos métodos privados exala o cheiro de design de ter muitas responsabilidades. Se o seu objeto tiver tantos métodos privados que você não ousa deixá-los sem teste, considere extraí-los para um novo objeto. Os métodos extraídos constituem o núcleo das responsabilidades do novo objeto e, portanto, constituem sua interface pública, que é (teoricamente) estável e, portanto, segura para depender.

Esta estratégia é boa, mas infelizmente só é verdadeiramente útil se a nova interface for realmente estável. Às vezes, a nova interface não existe, e é nesse ponto que a teoria e a prática se separam. Esta nova interface pública será exatamente tão estável (ou instável) quanto era a interface privada original. Os métodos não se tornam magicamente mais confiáveis só porque foram movidos. É dispendioso associar-se a métodos instáveis – independentemente de serem retratados como públicos ou privados.

Escolhendo testar um método privado Tempos de grande incerteza exigem medidas drásticas. Portanto, ocasionalmente é defensável colocar um pouco de código fedorento no lugar e esconder a bagunça até que informações melhores cheguem. Esconder bagunça é fácil; basta agrupar o código incorreto em um método privado.

Se você criar uma bagunça e nunca consertar, seus custos acabarão aumentando, mas no curto prazo, para o problema certo, ter confiança suficiente para escrever códigos embarracados pode economizar dinheiro. Quando sua intenção é adiar uma decisão de design, faça a coisa mais simples que resolva o problema de hoje. Isole o código por trás da melhor interface que você puder conceber, acalme-se e aguarde por mais informações.

A aplicação desta estratégia pode resultar em métodos privados extremamente instáveis.

Depois de dar esse salto, é razoável considerar agravar seus pecados testando esses métodos instáveis. O código do aplicativo é feio e sofrerá alterações frequentes mudar; o risco de quebrar algo está sempre presente. Esses testes são caros e provavelmente será forçado a mudar em sincronia com o código subjacente, mas qualquer outra opção para manter as coisas funcionando pode ser mais cara.

Estes testes de métodos privados não são necessários para saber que uma mudança quebrou alguma coisa, os testes de interface pública ainda servem a esse propósito admiravelmente. Testes de métodos privados produzem mensagens de erro que identificam diretamente as partes com falha do código privado. Esses erros mais específicos são acoplamentos apertados que aumentam a manutenção custos, mas facilitam a compreensão dos efeitos das mudanças e, por isso, levam em conta um pouco da dor de refatorar códigos privados complexos.

Reducir as barreiras à refatoração é importante, porque você refatorará. Isso é o ponto inteiro. A bagunça é temporária, você pretende refatorá-la. À medida que mais informações de design chegam, esses métodos privados irão melhorar. Assim que a neblina se dissipar e um o design se revelar, os métodos se tornarão mais estáveis. À medida que a estabilidade melhora, o o custo de manutenção e a necessidade de testes diminuirão. Eventualmente será possível extraia os métodos privados em uma classe separada e exponha-os com segurança ao mundo.

As regras básicas para testar métodos privados são as seguintes: Nunca os escreva, e se você faz, nunca os teste, a menos que faça sentido fazê-lo. Portanto, seja tendencioso em não escrever esses testes, mas não tenha medo de fazê-lo se isso melhorar sua situação.

Testando mensagens de saída

As mensagens enviadas, como você sabe na seção “O que testar”, são *consultas* ou *comandos*. As mensagens de consulta são importantes apenas para o objeto que as envia, enquanto as mensagens de comando têm efeitos que são visíveis para outros objetos em seu aplicativo.

Ignorando mensagens de consulta

As mensagens que não apresentam efeitos colaterais são conhecidas como mensagens de consulta . Aqui está um exemplo simples, onde o método gear_inches do Gear envia o diâmetro.

```

1 classe de equipamento
2 ...
3 def gear_inches
4     relação * diâmetro da roda
5 fim
6 fim

```

Nada no aplicativo além do método gear_inches se importa com esse diâmetro é enviado. O método do diâmetro não tem efeitos colaterais, executá-lo não deixa vestígios visíveis, e nenhum outro objeto depende de sua execução.

Da mesma forma que os testes devem ignorar as mensagens enviadas para si mesmos, eles também devem ignorar mensagens de consulta de saída. As consequências do diâmetro de envio estão ocultas dentro do Gear. Como o aplicativo geral não precisa que esta mensagem seja enviada, seus testes não precisam de cuidados.

O método gear_inches do Gear depende do resultado que o diâmetro retorna, mas testes para comprovar a exatidão do diâmetro pertencem ao Wheel, não aqui ao Gear. É redundante que o Gear duplique esses testes; os custos de manutenção aumentarão se isso acontecer.

A única responsabilidade do Gear é provar que gear_inches funciona corretamente e pode fazer isso simplesmente testando que gear_inches sempre retorna resultados apropriados.

Provando Mensagens de Comando

Às vezes, porém, é *importante* que uma mensagem seja enviada; outras partes da sua aplicação dependem de algo que acontece como resultado. Neste caso, o objeto em teste é responsável pelo envio da mensagem e seus testes devem comprovar isso.

Ilustrar esse problema requer um novo exemplo. Imagine um jogo onde os jogadores correrem em bicicletas virtuais. Essas bicicletas, obviamente, têm marchas. A classe Gear agora é responsável por avisar ao aplicativo quando um jogador muda de marcha para que o aplicativo possa atualizar o comportamento da bicicleta.

No código a seguir, o Gear atende a esse novo requisito adicionando um observador. Quando um jogador muda de marcha, os métodos set_cog ou set_chainring são executados. Esses métodos salvam o novo valor e então invocam o método alterado do Gear (linha 20). Esse método então envia as alterações para o observador, passando a coroa e a engrenagem atuais.

```

1 classe de equipamento
2 attr_reader :chainring, :cog, :wheel, :observer
3 def inicializar(args)
4     #...
5     @observador = args[:observador]
6 fim
7
8     #...
9
10 def set_cog(nova_cog)
11     @cog = nova_cog
12     mudado

```

```

13 fim
14
15 def set_chainring(new_chainring) @chainring =
16     new_chainring alterado
17
18 fim
19
20 defesas alteradas
21     observador.changed(coroa, roda dentada)
22 fim
23 #...
24 fim

```

Gear tem uma nova responsabilidade; ele deve notificar o observador quando as engrenagens ou coroas mudarem. Esta nova responsabilidade é tão importante quanto a obrigação anterior de calcular polegadas de engrenagem. Quando um jogador troca de marcha o aplicativo estará correto somente se o Gear enviar a mudança para o observador. Seus testes devem provar que esta mensagem foi enviada.

Eles não apenas deveriam provar isso, mas também deveriam fazê-lo sem fazer afirmações sobre o resultado retornado pelo método alterado do observador . Assim como os testes de Wheel reivindicaram a responsabilidade exclusiva de fazer afirmações sobre os resultados do seu próprio método de diâmetro , os testes do observador são responsáveis por fazer afirmações sobre os resultados do seu método modificado . A responsabilidade de testar o valor de retorno de uma mensagem é do seu receptor. Fazer isso em qualquer outro lugar duplica os testes e aumenta os custos.

Para evitar duplicação, você precisa de uma maneira de provar que o Gear envia alterações ao observador, o que não o força a confiar na verificação do que retorna quando isso acontece.

Felizmente, isso é fácil; você precisa de uma *simulação*. Simulações são testes de comportamento, em oposição a testes de estado. Em vez de fazer afirmações sobre o que uma mensagem retorna, os mocks definem uma expectativa de que uma mensagem será enviada.

O teste abaixo prova que o Gear cumpre suas responsabilidades e o faz sem se vincular a detalhes sobre como o observador se comporta. O teste cria uma simulação (linha 4) que injeta no lugar do observador (linha 8). Cada método de teste diz ao mock para esperar receber a mensagem alterada (linhas 12 e 17) e então verifica se isso aconteceu (linhas 14 e 20).

```

1 classe GearTest < MiniTest::Unit::TestCase
2
3 configuração de definição
4     @observador = MiniTest::Mock.new

```

```

5      @engrenagem      = Engrenagem.novo()
6          coroa: 52,
7          11,
8          engrenagem: observador: @observador)
9 fim
10
11 def test_notifica_observadores_quando_cogs_change
12     @observer.expect(:alterado, verdadeiro, [52, 27])
13     @gear.set_cog(27)
14     @observer.verify
15 fim
16
17 def test_notifica_observers_when_chainrings_change
18     @observer.expect(:alterado, verdadeiro, [42, 11])
19     @gear.set_chainring(42)
20     @observer.verify
21 fim
22 fim

```

Este é o padrão de uso clássico para uma simulação. No `notify_observers_when_change` acima, a linha 12 informa ao mock qual mensagem esperar, a linha 13 aciona o comportamento que deve fazer com que essa expectativa seja atendida e, em seguida, a linha 14 pergunta a simulação para verificar se realmente era. O teste passa apenas se estiver enviando `set_chainring` a engrenagem faz algo que faz com que o observador seja alterado com o determinado argumentos.

Observe que tudo o que o mock fez com a mensagem foi lembrar que a recebeu. Se o objeto em teste depende do resultado que obtém quando o observador recebe `alterado`, o mock pode ser configurado para retornar um valor apropriado. Este retorno o valor, no entanto, não vem ao caso. As simulações têm como objetivo provar que as mensagens foram enviadas, elas retornam resultados somente quando necessário para executar os testes.

O fato de o Gear funcionar bem mesmo depois de você simular a mudança do observador `método` tal que não faz absolutamente nada prova que Gear não se importa com o que método realmente faz. A única responsabilidade do Gear é enviar a mensagem; esse teste deveria se restringir a provar que o Gear faz isso.

Em um aplicativo bem projetado, testar mensagens enviadas é simples. Se você tem dependências injetadas proativamente, você pode facilmente substituir simulações. Definir expectativas nessas simulações permite provar que o objeto em teste cumpre suas responsabilidades sem duplicar afirmações que pertencem a outro lugar.

Testando tipos de patos A seção

Testando mensagens recebidas neste capítulo entrou no território dos testes de funções, mas embora tenha introduzido o problema, não forneceu uma resolução satisfatória. É hora de voltar a esse tópico e examinar como testar tipos de patos. Esta seção mostra como criar testes que os participantes podem compartilhar e depois retornar ao problema original e usar testes compartilháveis para evitar que testes duplos se tornem obsoletos.

Testando funções

O código para este primeiro exemplo vem do tipo Preparer duck do Capítulo 5, Reduzindo custos com Duck Typing. Esses primeiros exemplos de código repetem parte da lição do Capítulo 5; sinta-se à vontade para passar para o primeiro teste se tiver uma memória clara do problema.

Aqui está um lembrete das classes originais de Mecânico, TripCoordinator e Driver :

```
1 classe Mecânico
2 def prepare_bicycle(bicicleta) #...
3
4 fim
5 fim
6
7 classe TripCoordinator 8 def
buy_food(clientes) #...
9
10 fim
11 fim
12
Motorista de 13ª classe
14 def gas_up(veículo)
15     #...
16 fim
17 def fill_water_tank(veículo) #...
18
19 fim
20 fim
```

Cada uma dessas classes possui uma interface pública razoável, mas quando o Trip usou essas interfaces para preparar uma viagem foi forçado a verificar a classe de cada objeto para determinar qual mensagem enviar, conforme mostrado aqui:

```

1 aula de viagem
2 attr_reader :bicicletas, :clientes, :veículo
3
4 def preparar (preparadores)
5     preparadores.each {|preparador|
6         preparador de caso
7         quando mecânico
8             preparer.prepare_bicycles(bicicletas)
9         quando TripCoordinator
10        preparer.buy_food(clientes)
11         quando motorista
12            preparer.gas_up(veículo)
13            preparer.fill_water_tank(veículo)
14         fim
15     }
16 fim
17 fim
```

A declaração de caso acima casais se preparam para três aulas concretas existentes.

Imagine tentar testar o método de preparação ou as consequências de adicionar um novo tipo do preparador nesta mistura. Este método é difícil de testar e caro de manter.

Se você encontrar um código que usa esse antipadrão, mas não possui testes, considere refatorar para um design melhor antes de escrevê-los. É sempre perigoso fazer mudanças na ausência de testes, mas essa pilha oscilante de código é tão frágil que refatorá-la primeiro pode muito bem ser a estratégia mais econômica. A refatoração que corrige isso O problema é simples e facilita todas as alterações subsequentes.

A primeira parte da refatoração é decidir sobre a interface do Preparador e implementar essa interface em cada participante da função. Se a interface pública do Preparer for prepare_trip, as alterações a seguir permitem Mechanic, TripCoordinator e Motorista para desempenhar o papel:

```

1 classe Mecânico
2 def prepare_trip(viagem)
3     trip.bicycles.each {|bicicleta|
4         prepare_bicycle(bicicleta)}
```

```

5 fim
6
7     #...
8 fim
9

Coordenador de viagem 10 turmas
11 def prepare_trip(viagem)
12     comprar_comida(viagem.clientes)
13 fim
14
15 #...
16 fim
17

Motorista classe 18
19 def prepare_trip(viagem)
20     veículo = viagem.veículo
21     gas_up(veículo)
22     fill_water_tank(veículo)
23 fim
24     #...
25 fim

```

Agora que existem Preparadores , o método de preparação do Trip pode ser bastante simplificado. O após a refatoração altera o método de preparação do Trip para colaborar com os Preparadores em vez de enviar mensagens exclusivas para cada classe específica:

```

1 aula de viagem
2 attr_reader :bicicletas, :clientes, :veículo
3
4 def preparar (preparadores)
5     preparadores.each {|preparador|
6         preparador.prepare_trip(self)}
7 fim
8 fim

```

Depois de fazer essas refatorações, você estará preparado para escrever testes. O código acima contém uma colaboração entre Preparadores e uma Viagem, que agora pode ser pensada como um preparável. Seus testes devem documentar a existência da função Preparador , provar que cada um de seus jogadores se comporta corretamente e mostrar que Trip interage com apropriadamente.

Como diversas classes diferentes atuam como Preparadores, o teste do papel deve ser escrito uma vez e compartilhado por todos os jogadores. MiniTest é um framework de testes de baixa cerimônia e suporta o compartilhamento de testes da maneira mais simples possível, através de módulos Ruby.

Aqui está um módulo que testa e documenta a interface do Preparer :

```

1 módulo PreparerInterfaceTest 2 def
2   test_implements_the_preparer_interface
3     assert_respond_to(@object, :prepare_trip)
4   fim
5   fim

```

Este módulo prova que @object responde a prepare_trip. O teste abaixo utiliza este módulo para provar que o Mecânico é um Preparador. Inclui o módulo (linha 2) e fornece um Mecânico durante a configuração através da variável @object (linha 5).

```

1 classe MechanicTest < MiniTest::Unit::TestCase
2 incluem PreparerInterfaceTest
3
4 configuração def
5   @mechanic = @object = Mechanic.new
6 fim
7
8   # outros testes que dependem de @mechanic
9 fim

```

Os testes TripCoordinator e Driver seguem esse mesmo padrão. Eles também incluem o módulo (linhas 2 e 10 abaixo) e inicializam @object em seus métodos de configuração (linhas 5 e 13).

```

1 classe TripCoordinatorTest < MiniTest::Unit::TestCase 2 inclui PreparerInterfaceTest
3
4 configuração def
5   @trip_coordinator = @object = TripCoordinator.new
6 fim
7 fim
8
9 classe DriverTest < MiniTest::Unit::TestCase
10 incluem PreparerInterfaceTest
11

```

```
Configuração de 12 definições
13      @driver = @object = Driver.new
14 fim
15 fim
```

A execução desses três testes produz um resultado satisfatório:

```
1 Teste de Motorista
2 PASSAR test_implements_the_preparger_interface
3
4 Teste Mecânico
5 PASSAR test_implements_the_preparger_interface
6
7 TripCoordinatorTest
8 PASSAR test_implements_the_preparger_interface
```

Definir o PreparerInterfaceTest como um módulo permite escrever o teste uma vez e então reutilizá-lo em cada objeto que desempenha a função. O módulo serve como teste e como documentação. Aumenta a visibilidade da função e torna mais fácil provar que qualquer O preparador recém-criado cumpre com sucesso suas obrigações.

O método test_implements_the_preparger_interface testa uma entrada mensagem e, como tal, pertence aos testes do objeto receptor, e é por isso que o módulo é incluído nas provas de Mecânico, TripCoordinator e Driver. Entrada mensagens, no entanto, andam de mãos dadas com mensagens enviadas e você deve testar ambos os lados desta equação. Você provou que todos os receptores implementam corretamente prepare_trip, agora você também deve provar que o Trip o envia corretamente.

Como você sabe, provar que uma mensagem de saída foi enviada é feito configurando expectativas em uma simulação. O teste a seguir cria uma simulação (linha 4), diz para esperar prepare_trip (linha 6), aciona o método prepare do Trip (linha 8) e depois verifica que o mock recebeu a mensagem adequada (linha 9).

```
1 classe TripTest < MiniTest::Unit::TestCase
2
3 def test_requests_trip_preparation
4      @preparer = MiniTest::Mock.new
5      @viagem = Viagem.new
6      @preparer.expect(:prepare_trip, nil, [@trip])
7
8      @trip.prepare([@preparer])
```

```

9      @preparer.verify
10     fim
11     fim

```

O teste `test_requests_trip_preparation` reside diretamente no `TripTest`. `Trip` é o único `Preparable` no aplicativo, portanto não há outro objeto com o qual compartilhar este teste. Se surgirem outros `Preparables`, o teste deverá ser extraído para um módulo e compartilhado entre os `Preparables` naquele momento.

A execução deste teste prova que o `Trip` colabora com os `Preparadores` usando a interface correta:

```

1 TripTest
2 PASSAR test_requests_trip_preparation

```

Isso conclui os testes da função `Preparador`. Agora é possível retornar ao problema da fragilidade ao usar dublês para desempenhar papéis em testes.

Usando testes de função para validar duplicatas

Agora que você sabe como escrever testes reutilizáveis que provam que um objeto desempenha corretamente uma função, você pode usar esta técnica para reduzir a fragilidade causada pelo stub.

No exemplo anterior, `Testando Mensagens Recebidas`, introduziu o problema de “viver o sonho”. O teste final naquela seção continha um falso positivo enganoso, no qual um teste que deveria ter falhado foi aprovado devido a um teste duplo que eliminou um método obsoleto. Aqui está um lembrete daquele teste que passou com falha:

```

1 classe DiameterDouble
2
3 def diâmetro # A interface mudou para 'largura', # mas este duplo e o Gear # ainda usam
4     10             'diâmetro'.
5     fim
6     fim
7
8 classes GearTest < MiniTest::Unit::TestCase
9 def test_calcula_gear_inches
10    engrenagem = Gear.new(coroa:
11        52,

```

Testando tipos de pato

```

12           engrenagem:
13           roda:      11,
14                           DiâmetroDuplo.novo)
15           assert_in_delta(47,27,
16                           gear.gear_inches, 0,01)
17
18 fim
19 fim

```

O problema com este teste é que DiameterDouble pretende reproduzir o

Função diametrizável , mas o faz incorretamente. Agora que a interface do Diameterizable mudou, o DiameterDouble está desatualizado. Este duplo obsoleto permite teste para se atrapalhar na crença equivocada de que o Gear funciona corretamente, quando na verdade na verdade, o GearTest só funciona quando combinado com seu duplo de teste igualmente confuso.

O aplicativo está quebrado, mas você não pode saber executando este teste.

Você viu o WheelTest pela última vez na seção Usando testes para documentar funções, onde estava tentando combater esse problema aumentando a visibilidade do Diameterizable interface. Aqui está um exemplo onde a linha 6 prova que Wheel atua como um Diameterizável que implementa largura:

```

1 classe WheelTest < MiniTest::Unit::TestCase
2 configuração de definição
3           @roda = Roda.new(26, 1,5)
4 fim
5
6 def test_implements_the_diameterizable_interface
7           assert_respond_to(@wheel, :largura)
8 fim
9
10 def test_calcula_diâmetro
11           #...
12 fim
13 fim

```

Com este teste, você agora segura todas as peças necessárias para resolver o problema de fragilidade. Você sabe como compartilhar testes entre os jogadores de uma função, você reconhece que tem dois jogadores da função Diameterizable , e você tem um teste que qualquer objeto pode usar para provar que ele desempenha corretamente o papel.

O primeiro passo para resolver o problema é extrair `test_implements_the_diameterizable_interface` do `Wheel` em um módulo próprio:

```

1 módulo DiameterizableInterfaceTest
2 def test_implements_the_diameterizable_interface
3     assert_respond_to(@objeto, :largura)
4 fim
5 fim

```

Assim que este módulo existir, reintroduzir o comportamento extraído de volta no `WheelTest` é uma simples questão de incluir o módulo (linha 2) e inicializar `@object` com um `Roda` (linha 5):

```

1 classe WheelTest < MiniTest::Unit::TestCase
2 incluem DiameterizableInterfaceTest
3
4 configuração de definição
5     @roda = @objeto = Roda.new(26, 1,5)
6 fim
7
8 def test_calcula_diâmetro
9     #...
10 fim
11 fim

```

Neste ponto, o `WheelTest` funciona exatamente como antes da extração, como você pode ver por executando o teste:

```

1 Teste de Roda
2 PASSAR test_implements_the_diameterizable_interface
3 PASSAR teste_calcula_diâmetro

```

É gratificante que o `WheelTest` ainda seja aprovado, mas essa refatoração serve a um propósito mais amplo do que apenas reorganizar o código. Agora que você tem um independente módulo que prova que um `Diameterizable` se comporta corretamente, você pode usar o módulo para evitar que testes duplos se tornem obsoletos silenciosamente.

O `GearTest` abaixo foi atualizado para usar este novo módulo. Linhas 9 a 15 definia uma nova classe de teste, `DiameterDoubleTest`. `DiameterDoubleTest` não é sobre

Equipamento em si, seu objetivo é evitar a fragilidade do teste, garantindo a solidez contínua do duplo.

```

1 classe DiameterDouble
2 diâmetro definido
3     10
4 fim
5 fim
6
7 # Prove que o teste duplo respeita a interface desta
8 # teste espera.
9 classe DiameterDoubleTest < MiniTest::Unit::TestCase
10 incluem DiameterizableInterfaceTest
11
Configuração de 12 definições
13     @objeto = DiameterDouble.new
14 fim
15 fim
16
17 classe GearTest < MiniTest::Unit::TestCase
18 def test_calcula_gear_inches
19     engrenagem = engrenagem.new(
20         coroa: 52,
21         engrenagem: 11,
22         roda: DiâmetroDuplo.novo)
23
24     assert_in_delta(47,27,
25         gear.gear_inches, 0,01)
26
27 fim
28 fim

```

O fato de DiameterDouble e Gear estarem incorretos permitiu que versões anteriores deste teste fossem aprovadas. Agora que o duplo está sendo testado para garantir que ele desempenhe honestamente seu papel, a execução do teste finalmente produz um erro:

```

1 DiâmetroDuploTeste
2 FALHA test_implements_the_diameterizable_interface
3     Esperado #<DiameterDouble:...> (DiameterDouble)
4     para responder a #largura.
5 Teste de Engrenagem
6 PASSAR test_calcula_gear_inches

```

O GearTest ainda passa erroneamente, mas isso não é um problema porque DiameterDoubleTest agora informa que DiameterDouble está errado. Esta falha faz com que você corrija o DiameterDouble para implementar a largura, conforme mostrado na linha 2 abaixo:

```

1 classe DiameterDouble
2 largura definida
3     10
4 fim
5 fim

```

Após esta alteração, reexecutar o teste produz uma falha no GearTest:

```

1 DiâmetroDuploTeste
2     PASSAR test_implements_the_diameterizable_interface
3
4 Teste de Engrenagem
5 ERRO test_calcula_gear_inches
6         método indefinido 'diâmetro'
7             para #<DiameterDouble:0x0000010090a7f8>
8                 gear_test.rb:35:em 'gear_inches'
9                 gear_test.rb:86:in 'test_calcula_gear_inches'
10

```

Agora que o DiameterDoubleTest foi aprovado, o GearTest falhou. Esta falha aponta diretamente à linha de código ofensiva no Gear. Os testes finalmente dizem para você mudar o Gear Método gear_inches para enviar largura em vez de diâmetro, como neste exemplo:

```

1 classe de equipamento
2
3 def gear_inches
4     # finalmente, 'largura' em vez de 'diâmetro'
5     proporção * roda.largura
6 fim
7
8 #...
9 fim

```

Depois de fazer essa alteração final, o aplicativo estará correto e todos os testes serão aprovados corretamente:

```

1 DiâmetroDuploTeste
2     PASSAR test_implements_the_diameterizable_interface

```

```

3
4 Teste de Engrenagem
5 PASSAR test_calcula_gear_inches

```

Este teste não apenas passa, mas continuará a passar (ou falhar) de forma adequada, não importa o que acontece com a interface Diameterizável . Quando você trata o teste duplica como você faria qualquer outro participante e testá-los para provar sua correção, você evita o teste fragilidade e pode quebrar sem medo das consequências.

O desejo de testar tipos de patos cria uma necessidade de testes compartilháveis para funções e, uma vez Ao adquirir essa perspectiva baseada em papéis, você poderá usá-la a seu favor em muitas situações. Do ponto de vista do objeto em teste, todos os outros objetos são uma função e lidar com objetos como se fossem representantes dos papéis que desempenham afrouxa o acoplamento e aumenta a flexibilidade, tanto na sua aplicação quanto nos seus testes.

Testando código herdado

Você finalmente chegou ao último desafio: testar o código herdado. Esta seção é muito como os anteriores, na medida em que recapitula um exemplo visto anteriormente e depois prossegue para testá-lo. O exemplo usado aqui é a hierarquia final da Bicicleta do Capítulo 6, Adquirindo comportamento por meio de herança. Mesmo que essa hierarquia eventualmente provou ser inadequado para herança, o código subjacente é bom e serve admiravelmente como uma base para esses testes.

Especificando a interface herdada

Aqui está a classe Bicycle como você a viu pela última vez no Capítulo 6:

	Bicicleta de 1 classe
--	------------------------------

```

2 attr_reader :tamanho, :cadeia, :tire_size
3
4 def inicializar(args={})
5     @tamanho          = args[:tamanho]
6     @corrente         = args[:cadeia] || cadeia_padrão
7     @tamanho_do_pneu = args[:tamanho_do_pneu] || tamanho_do_pneu_padrão
8     post_initialize(args)
9 fim
10
11 peças sobressalentes def
12     {tamanho_do_pneu:tamanho_do_pneu,
13      corrente:       cadeia}.merge(local_spares)

```

```

14 fim
15
16 def tamanho_do_pneu_padrão
17     aumentar NotImplementedError
18 fim
19
20 # subclasses podem substituir
21 def post_initialize(args)
22     nada
23 fim
24
25 def local_spares
26     {}
27 fim
28
29 def default_chain
30     '10 velocidades'
31 fim
32 fim

```

Aqui está o código para RoadBike, uma das subclasses de Bicycle :

```

1 classe RoadBike < Bicicleta
2 attr_reader :tape_color
3
4 def post_initialize(args)
5     @tape_color = args[:tape_color]
6 fim
7
8 def local_spares
9     {tape_color: fita_color}
10 fim
11
12 def tamanho_do_pneu_padrão
13     '23'
14 fim
15 fim

```

O primeiro objetivo do teste é provar que todos os objetos nesta hierarquia honram o seu contrato. O Princípio da Substituição de Liskov declara que os subtipos devem ser substituíveis pelos seus supertipos. As violações de Liskov resultam em objetos não confiáveis que não se comportam como esperado. A maneira mais fácil de provar que todos os objetos na hierarquia

obedece Liskov é escrever um teste compartilhado para o contrato comum e incluir esse teste em cada objeto.

O contrato está incorporado em uma interface compartilhada. O teste a seguir articula o interface e, portanto, define o que significa ser uma bicicleta:

```
1 módulo BicycleInterfaceTest 2 def
test_responds_to_default_tire_size
3     assert_respond_to(@object, :default_tire_size)
4 fim
5
6 def test_responds_to_default_chain
7     assert_respond_to(@object, :default_chain)
8 fim
9
10 def test_responds_to_chain
11     assert_respond_to(@object, :chain)
12 fim
13
14 def test_responds_to_size
15     assert_respond_to(@object, :size)
16 fim
17
18 def test_responds_to_tire_size
19     assert_respond_to(@object, :tire_size)
20 fim
21
22 def test_responds_to_spares
23     assert_respond_to(@object, :spares)
24 fim
25 fim
```

Qualquer objeto que passe no BicycleInterfaceTest pode ser confiável para agir como uma bicicleta. Todas as classes na hierarquia Bicycle devem responder a esta interface e ser capazes de passar neste teste. O exemplo a seguir inclui este teste de interface na superclasse abstrata BicycleTest (linha 2) e na subclasse concreta RoadBikeTest (linha 10):

```
1 classe BicycleTest < MiniTest::Unit::TestCase 2 inclui
BicycleInterfaceTest
3
4 configuração de definição
```

```

5      @bike = @object = Bicycle.new({tamanho_do_pneu: 0})
6 fim
7 fim
8
9 classe RoadBikeTest < MiniTest::Unit::TestCase
10 incluem BicycleInterfaceTest
11
Configuração de 12 definições
13      @bike = @object = RoadBike.new
14 fim
15 fim

```

Executar o teste conta uma história:

```

1 Teste de bicicleta
2 PASSAR test_responds_to_default_chain
3 PASSAR test_responds_to_size
4 PASSAR test_responds_to_tire_size
5 PASSAR test_responds_to_chain
6 PASSAR test_responds_to_spares
7 PASSAR test_responds_to_default_tire_size
8
9 Teste de bicicleta de estrada
10 PASSAR test_responds_to_chain
11 PASSAR test_responds_to_tire_size
12 PASSAR test_responds_to_default_chain
13 PASSAR test_responds_to_spares
14 PASSAR test_responds_to_default_tire_size
15 PASSAR test_responds_to_size

```

Observação

Não se assuste porque as partes do BicycleTest e RoadBikeTest é executado em ordens diferentes; a ordenação aleatória de testes é um recurso do MiniTest.

O BicycleInterfaceTest funcionará para todos os tipos de bicicletas e pode ser facilmente incluído em qualquer nova subclasse. Ele documenta a interface e evita regressões.

Especificando responsabilidades da subclasse

Não apenas todas as Bicicletas compartilham uma interface comum, como a superclasse abstrata Bicicleta impõe requisitos às suas subclasses.

Confirmado o comportamento da

subclasse Como existem muitas subclasses, elas devem compartilhar um teste comum para provar que cada uma atende aos requisitos. Aqui está um teste que documenta os requisitos para subclasses:

```

1 módulo BicycleSubclassTest 2 def
2   test_responds_to_post_initialize
3     assert_respond_to(@object, :post_initialize)
4   fim
5
6   def test_responds_to_local_spares
7     assert_respond_to(@object, :local_spares)
8   fim
9
10  def test_responds_to_default_tire_size
11    assert_respond_to(@object, :default_tire_size)
12  fim
13  fim

```

Este teste codifica os requisitos para subclasses de Bicicleta. Isso não força as subclasses a implementar esses métodos; na verdade, qualquer subclasse é livre para herdar post_initialize e local_spares. Este teste apenas prova que uma subclasse não faz nada tão maluco que cause falha nessas mensagens. O único método que deve ser implementado pelas subclasses é default_tire_size. A implementação da superclasse de default_tire_size gera um erro; este teste falhará a menos que a subclasse implemente sua própria versão especializada.

RoadBike atua como uma bicicleta , portanto seu teste já inclui o BicycleInterfaceTest. O teste abaixo foi alterado para incluir o novo BicycleSubclassTest; RoadBike também deve funcionar como uma subclasse de Bicycle.

```

1 classe RoadBikeTest < MiniTest::Unit::TestCase
2 incluem BicycleInterfaceTest 3 incluem
3   BicycleSubclassTest
4
5   configuração de
6     definição @bike = @object = RoadBike.new
7   fim
8   fim

```

A execução deste teste modificado conta uma história aprimorada:

```

1 Teste de bicicleta de estrada
2     PASSAR test_responds_to_default_tire_size
3     PASSAR test_responds_to_spares
4     PASSAR test_responds_to_chain
5     PASSAR test_responds_to_post_initialize
6     PASSAR test_responds_to_local_spares
7     PASSAR test_responds_to_size
8     PASSAR test_responds_to_tire_size
9     PASSAR test_responds_to_default_chain

```

Cada subclasse de Bicycle pode compartilhar esses mesmos dois módulos, porque cada subclasse deve agir tanto como uma bicicleta quanto como uma subclasse de bicicleta. Mesmo que tenha sido já há algum tempo que você viu a subclasse MountainBike , você certamente pode apreciar o capacidade de garantir que as MountainBikes sejam bons cidadãos, simplesmente adicionando estes dois módulos para seu teste, conforme mostrado aqui:

```

1 classe MountainBikeTest < MiniTest::Unit::TestCase
2 incluem BicycleInterfaceTest
3 incluem BicycleSubclassTest
4
5 configuração de definição
6     @bike = @object = MountainBike.new
7 fim
8 fim

```

O BicycleInterfaceTest e o BicycleSubclassTest, combinados, levam tudo da dor de testar o comportamento comum das subclasses. Esses testes dão a você confiança de que as subclasses não estão se afastando do padrão e permitem novatos para criar novas subclasses com total segurança. Programadores recém-chegados não têm que vasculhar as superclasses para descobrir requisitos, eles podem apenas incluir esses testes quando eles escrevem novas subclasses.

Confirmando a aplicação da superclasse

A classe Bicycle deve gerar um erro se uma subclasse não implementar tamanho_do_pneu_padrão. Embora este requisito se aplique a subclasses, o real o comportamento de fiscalização está em Bicicleta. Este teste é, portanto, colocado diretamente em BicycleTest, conforme mostrado na linha 8 abaixo:

```

1 classe BicycleTest < MiniTest::Unit::TestCase 2 inclui
BicycleInterfaceTest
3
4 configuração def
5   @bike = @object = Bicycle.new({tamanho_do_pneu: 0})
6 fim
7
8 def test_forces_subclasses_to_implement_default_tire_size
9   assert_raises(NotImplementedError) {@bike.default_tire_size}
10 fim
11 fim

```

Observe que a linha 5 do BicycleTest fornece um tamanho de pneu, ainda que estranho, no momento da criação de Bicycle . Se você olhar novamente para o método de inicialização do Bicycle, verá por quê. O método de inicialização espera receber um valor de entrada para tire_size ou poderá recuperá-lo enviando posteriormente a mensagem default_tire_size . Se você remover o argumento tire_size da linha 5, este teste morre em seu método de configuração durante a criação de uma bicicleta. Sem esse argumento, Bicycle não consegue passar pela inicialização do objeto com êxito.

O argumento tire_size é necessário porque Bicycle é uma classe abstrata que não espera receber a nova mensagem. A bicicleta não possui um protocolo de criação agradável e amigável. Não é necessário porque o aplicativo real nunca cria instâncias de Bicycle. Porém, o fato de o aplicativo não criar novas bicicletas não significa que isso nunca aconteça. Certamente que sim. A linha 5 do BicycleTest acima cria claramente uma nova instância desta classe abstrata.

Este problema é onipresente ao testar classes abstratas. O BicycleTest precisa de um objeto no qual executar testes e o candidato mais óbvio é uma instância de Bicycle. No entanto, criar uma nova instância de uma classe abstrata pode variar entre difícil e impossível. Este teste tem a sorte de que o protocolo de criação de Bicycle permite que o teste crie uma instância concreta de Bicycle passando tire_size , mas criar um objeto testável nem sempre é tão fácil e você pode achar necessário雇用 uma estratégia mais sofisticada. Felizmente, há uma maneira fácil de superar esse problema geral que será abordado abaixo na seção Testando o comportamento abstrato da superclasse.

Por enquanto, fornecer o argumento tire_size funciona perfeitamente. A execução de BicycleTest agora produz uma saída que se parece mais com uma superclasse abstrata:

```

1 Teste de bicicleta
2 PASSAR test_responds_to_default_tire_size
3 PASSAR test_responds_to_size

```

```

4 PASSAR test_responds_to_default_chain
5 PASSAR test_responds_to_tire_size
6 PASSAR test_responds_to_chain
7 PASSAR test_responds_to_spares
8 PASSAR test_forces_subclasses_to_implement_default_tire_size

```

Testando comportamento único

Os testes de herança até agora se concentraram em testar qualidades comuns. Maioria dos testes resultantes eram compartilháveis e acabaram sendo colocados em módulos (BicycleInterfaceTest e BicycleSubclassTest), embora um teste (forces_subclasses_to_implement_default_tire_size) foi colocado diretamente no BicycleTest.

Agora que você dispensou o comportamento comum, restam duas lacunas. Lá ainda não existem testes para especializações, nem para as fornecidas pelas subclasses concretas, nem para aquelas definidas na superclasse abstrata. A seção a seguir concentra no primeiro; ele testa especializações fornecidas por subclasses individuais. A seção depois move o foco para cima na hierarquia e testa o comportamento exclusivo de Bicycle.

Testando o comportamento da subclasse concreta

Agora é a hora de renovar seu compromisso de escrever o número mínimo absoluto de testes. Relembre a aula RoadBike . Os módulos compartilhados já comprovam a maior parte seu comportamento. A única coisa que resta testar são as especializações que a RoadBike fornece.

É importante testar essas especializações sem incorporar conhecimento do superclasse no teste. Por exemplo, RoadBike implementa local_spares e também responde a peças sobressalentes. O RoadBikeTest deve garantir que local_spares funcione enquanto mantendo ignorância deliberada sobre a existência do método de reposição . O compartilhado O BicycleInterfaceTest já comprova que a RoadBike responde corretamente às peças sobressalentes , é redundante e, em última análise, limitando a referência direta a esse método neste teste.

O método local_spares , entretanto, é claramente de responsabilidade da RoadBike . Linha 9 abaixo testa esta especialização diretamente no RoadBikeTest:

```

1 classe RoadBikeTest < MiniTest::Unit::TestCase
2 incluem BicycleInterfaceTest
3 incluem BicycleSubclassTest
4
5 configuração de definição
6     @bike = @object = RoadBike.new(tape_color: 'vermelho')
7 fim

```

```

8
9 def test_puts_tape_color_in_local_spares
10     assert_equal 'vermelho', @bike.local_spares[:tape_color]
11 fim
12 fim

```

A execução do RoadBikeTest mostra agora que cumpre as suas responsabilidades comuns e também fornece especializações próprias:

```

1 Teste de bicicleta de estrada
2 PASSAR test_responds_to_default_chain
3 PASSAR test_responds_to_default_tire_size
4 PASSAR test_puts_tape_color_in_local_spares
5 PASSAR test_responds_to_spares
6 PASSAR test_responds_to_size
7 PASSAR test_responds_to_local_spares
8 PASSAR test_responds_to_post_initialize
9 PASSAR test_responds_to_tire_size
10 PASSAR test_responds_to_chain

```

Testando o comportamento abstrato da superclasse

Agora que você testou as especializações de subclasse, é hora de voltar atrás e terminar testando a superclasse. Mover seu foco para cima na hierarquia para Bicicleta reintroduz uma problema encontrado anteriormente; Bicicleta é uma superclasse abstrata. Criando um instância de Bicycle não é apenas difícil, mas a instância pode não ter todo o comportamento você precisa fazer o teste.

Felizmente, suas habilidades de design fornecem uma solução. Como a Bicycle usou métodos de modelo para adquirir especializações concretas, você pode eliminar o comportamento que normalmente seria fornecido por subclasses. Melhor ainda, porque você entende o Princípio de Substituição de Liskov, você pode facilmente fabricar uma instância testável de Bicycle criando uma nova subclasse para uso exclusivo neste teste.

O teste abaixo segue exatamente essa estratégia. A linha 1 define uma nova classe, StubbedBike, como uma subclasse de Bicycle. O teste cria uma instância desta classe (linha 15) e usa-o para provar que Bicycle inclui corretamente os locais_spares da subclasse contribuição em peças sobressalentes (linha 23).

Às vezes, continua sendo conveniente criar uma instância do abstrato Bicycle classe, mesmo que isso exija a passagem do argumento tire_size , como na linha 14. Isso instância de Bicycle continua a ser usada no teste na linha 18 para provar que o a classe abstrata força as subclasses a implementar default_tire_size.

Esses dois tipos de Bicicletas coexistem pacificamente no teste, como você pode ver aqui:

```

1 classe StubbedBike < Bicicleta
2 def tamanho_do_pneu_padrão
3     0
4 fim
5 def local_spares
6     {selim: 'doloroso'}
7 fim
8 fim
9
10 classes BicycleTest < MiniTest::Unit::TestCase
11 incluem BicycleInterfaceTest
12
13 configuração de definição
14     @bike = @object = Bicycle.new({tamanho_do_pneu: 0})
15     @stubbed_bike = StubbedBike.new
16 fim
17
18 def test_forces_subclasses_to_implement_default_tire_size
19     assert_raises(NotImplementedError) {
20         @bike.default_tire_size}
21 fim
22
23 def test_includes_local_spares_in_spares
24     assert_equal @stubbed_bike.spares,
25             {tamanho_do_pneu: 0,
26              corrente:      '10 velocidades',
27              selim:        'doloroso'}
28 fim
29 fim

```

A ideia de criar uma subclasse para fornecer stubs pode ser útil em muitas situações. Como contanto que sua nova subclasse não viole Liskov, você pode usar esta técnica em qualquer teste você gosta.

A execução do BicycleTest agora prova que inclui contribuições de subclasses no

lista de peças sobressalentes :

1	Teste de bicicleta
2	PASSAR test_responds_to_spares
3	PASSAR test_responds_to_tire_size

Testando código herdado

```

4 PASSAR test_responds_to_default_chain
5 PASSAR test_responds_to_default_tire_size
6 PASSAR test_forces_subclasses_to_implement_default_tire_size
7 PASSAR test_responds_to_chain
8 PASSAR test_includes_local_spares_in_spares
9 PASSAR test_responds_to_size

```

Um último ponto: se você teme que a StubbedBike se torne obsoleta e permita BicycleTest para passar quando deveria falhar, a solução está próxima. Há já é um BicycleSubclassTest comum . Assim como você usou o Diameterizable InterfaceTest para garantir o bom comportamento contínuo do DiameterDouble , você pode use BicycleSubclassTest para garantir a correção contínua de StubbedBike. Adicionar o seguinte código para BicycleTest:

```

1 # Prove que o teste duplo respeita a interface desta
2 # teste espera.
3 classes StubbedBikeTest < MiniTest::Unit::TestCase
4 incluem BicycleSubclassTest
5
6 configuração de definição
7     @object = StubbedBike.new
8 fim
9 fim

```

Depois de fazer essa alteração, a execução do BicycleTest produz esta saída adicional:

```

1 StubbedBikeTest
2     PASSAR test_responds_to_default_tire_size
3     PASSAR test_responds_to_local_spares
4     PASSAR test_responds_to_post_initialize

```

Hierarquias de herança cuidadosamente escritas são fáceis de testar. Escreva um teste compartilhável para a interface geral e outra para as responsabilidades da subclasse. Isolar diligentemente responsabilidades. Tenha especial cuidado ao testar especializações de subclasse para evitar conhecimento da superclasse vaze para o teste da subclasse.

Testar superclasses abstratas pode ser um desafio; use a substituição de Liskov Princípio a seu favor. Se você aproveitar Liskov e criar novas subclasses que sejam usado exclusivamente para teste, considere exigir que essas subclasses sejam aprovadas em sua subclasse teste de responsabilidade para garantir que não se tornem obsoletos acidentalmente.

Resumo

Os testes são indispensáveis. Aplicativos bem projetados são altamente abstratos e estão sob constante pressão para evoluir; sem testes essas aplicações não podem ser compreendidas nem alterado com segurança. Os melhores testes são fracamente acoplados ao código e teste subjacentes. tudo de uma vez e no lugar certo. Eles agregam valor sem aumentar custos.

Um aplicativo bem projetado com um conjunto de testes cuidadosamente elaborado é uma alegria de se ver e um prazer estender. Ele pode se adaptar a cada nova circunstância e atender a qualquer necessidade inesperada.

Posfácio

Responsabilidades, dependências, interfaces, patos, herança, compartilhamento de comportamento, composição e testes – você aprendeu tudo. Você mergulhou em um mundo de objetos e, se este livro atingiu seu objetivo, você pensa sobre os objetos de maneira diferente agora do que quando começou.

O Capítulo 1, Design Orientado a Objetos, afirmou que o design orientado a objetos trata do gerenciamento de dependências; essa afirmação ainda é verdadeira, mas é apenas uma verdade sobre o design. Uma verdade mais profunda é que existe uma maneira pela qual todos os objetos são idênticos, independentemente de representarem aplicações inteiras, subsistemas principais, classes individuais ou métodos simples. Um único objeto nunca está sozinho; os aplicativos consistem em objetos relacionados entre si. Assim como uma chave e sua fechadura, uma mão e sua luva, ou uma chamada e sua resposta, os objetos são definidos, não pelo que fazem, mas pelas mensagens que passam entre eles. O design orientado a objetos é fractal; o problema central é definir uma forma extensível de comunicação dos objetos e, em todos os níveis de ampliação, esse problema parece o mesmo.

Este livro está cheio de regras sobre como escrever código – regras para gerenciar dependências e criar interfaces. Agora que você conhece essas regras, pode adaptá-las aos seus próprios propósitos. As tensões inerentes ao design significam que estas regras devem ser quebradas; aprender a quebrá-los bem é a maior força de um designer.

Os princípios do design são ferramentas e com a prática eles virão naturalmente às suas mãos, permitindo que você crie aplicativos mutáveis que atendam ao seu propósito e lhe tragam alegria. Suas aplicações não serão perfeitas, mas não desanime. A perfeição é ilusória, talvez até inalcançável; isso não deve impedir o seu desejo de alcançá-lo.

Persistir. Prática. Experimentar. Imagine. Faça o seu melhor trabalho e tudo o mais se seguirá.

Esta página foi intencionalmente deixada em branco

Índice

= operador, 43, 48–49	<i>Comporta-se como um relacionamento</i> , 189	código, organizando para permitir mudanças, 16–17
Comportamento abstrato, promoção, 120–23 classes, 117–20, 235, 237 definição de, 54 superclasse, criação, 117–20	Comportamento adquirido por herança, 105–39 confirmando, 233–36 estruturas de dados, ocultando, 26–29 dependendo de, em vez de dados, 24–29 variáveis de instância, 19 subclasse, 233–39 superclasse, 234–39 teste, 236–39	concreto, 106–9, 209 decidindo o que pertence, 16–17 dissociando, escrevendo código herdável, 161 carregado de dependência, evitando, 55 agrupando métodos em, 16 referências a (<i>Ver Código fraca</i> mente acoplado, escrita) responsabilidades isoladas em, 31–33 baseado em Ruby <i>versus</i> estrutura, 53–54 <i>tipo e categoria</i> usada em, 111 virtual, 61 linguagens OO baseadas em classe, 12–13 classe de classe, 14 herança clássica, 105–6 Classe de ambiguidade de objeto, 94–95 verificação, 97, 111, 146 Classe em teste, remoção de métodos privados de, 214 Código.
Abstrações extraír, 150–53 encontrar, 116–29 insistir em, ao escrever código herdável, 159 reconhecer, 54–55 separar de concreções, 123–25 apoiando, em testes intencionais, padrão de método de modelo 194, 125–29 Tipos entre classes, 86 Ao <i>infinito</i> , 3 Agregação, 183–84 Ágil, 8–10 Definição antipadrão de, 109, 111 reconhecendo, 158–59 Dependências de ordem de argumento, remoção, 46–51 padrões, definição explícita, 48–49 hashes para inicialização argumentos, usando, 46–48 inicialização multiparâmetro, isolando, 49–51 Delegação automática de mensagens, 105–6	Desenvolvimento Orientado a Comportamento (BDD), 199, 213 Design frontal grande (BUFD), 8–9 Booch, Grady, 188 Ponto de equilíbrio, 11 Insetos, descoberta, 193 Declarações de caso que ativam a classe, 96–98 kind_of? e is_a?, 97 responde_to?, 97 Categoría usada na aula, 111 Classe. <i>Consulte também Responsabilidade única, classes</i> com resumo, 117–20, 235, 237 evitando carga de dependência, 55 bicicleta, atualização, 164–65 declarações de caso que ligam, 96–98	herdável, escrita; Código herdado, teste concreto, escrita, 147–50 injecção de dependência na forma, 41–42 dependendo do comportamento em vez de dados, 24–29 abraçando mudanças, escrita,

- inicialização, 121
 fracalemente acoplado, escrita, 39–51
 aberto-fechado, 185
 organização para permitir mudanças, 16–17
 apresentando sua melhor
 (inter)face, escrita, 76–79
 confiando na digitação de pato, escrita, 95–100
 simples responsabilidade, aplicação, 29–33
 verdades sobre, 53
 Técnica de organização de código, 184
 Coesão, 22
 Mensagens de comando, 197, 216–18
 Ciclo de compilação/criação, 102, 103, 104
 Compilador, 54, 101, 103–4, 118
 Agregação de
 composição e, 183–84 benefícios
 de, 187 de bicicleta,
 180–84 de bicicleta de
 peças, 164–68 consequências de,
 aceitação, 187–88 custos de, 187–88 para
 relacionamentos *tem-*
 um, 183, 190 herança e , decidindo entre,
 184–90 peças de fabricação, 176–80 objetos combinados
 com, 163–90 de objeto de peças, 168–76 resumo, 190 uso do termo, 183–84
- Classe de concreto, 106–9, 209
 Abstrações de
 concreções separadas de, 123–25
 herança
 e, 106–9 reconhecendo, 54–55
 escrita, 147–50
- Independência de contexto, busca, 71–73 minimização, 79
 Contrato, honra, 159–60
 Custos
 de composição, 187–88 de
 tipagem de pato, 85–104 de
 herança, 185–86 de teste,
 191–240
- Acoplamento
 de classes de desacoplamento na
 escrita de código herdável, 161
- desacoplamento de subclasses usando
 mensagens de gancho,
 134–38 entre superclasses e
 subclasses, gerenciamento,
 129–38
 compreensão, 129–34
 Acoplamento entre objetos (CBO), 37–38
 C++, 102
- Dados
 dependendo do comportamento em
 vez de, 24–
 29 variáveis de instância, ocultação, 24–26 estruturas, ocultação, 26–29 tipos, 12, 13
- Desacoplando
 classes na escrita de código
 herdável,
 161 subclasses usando mensagens de
 gancho, 134–38 Padrões, definindo
 explicitamente , 48–49
 Delegação, 82, 183 Deméter. Veja
 Lei de Deméter (LoD)
 Falha de rebaixamento, 123
 ordem de
 argumento de dependências,
 remoção,
 46–51 acoplamento entre objetos, 37–38
 direção de (*consulte* direção de
 dependência)
 descoberta, 55–57
 injeção (*consulte* injeção de
 dependência)
 interfaces e, 62–63
 isolamento, 42–45
 código fracalemente acoplado, escrita,
 39–51
 gerenciamento, 35–57
 objetos falando por si mesmos, 147 outros,
 38–
 39 reconhecimento,
 37 remoção
 desnecessária, 145–47 reversão, 51–53
 agendamento tipo
 pato, descoberta, 146–47
 resumo, 57 compreensão,
 36–39
- Abstrações de direção de
 dependência, reconhecendo, 54–55
- mudança, probabilidade de (Ver
 Probabilidade de mudança)
 escolher, 53–57
 concreções, reconhecer, 54–55 classes
 carregadas de dependência,
 evitar, 55
 encontrar, 55–57
 gerenciar, 51–57
 reverte, 51–53
 Falha de injeção de
 dependência de,
 60 em código fracalemente acoplado, 39–42 como funções,
 208–13 para moldar código,
 41–42 usando classes, 207–8
 Princípio de Inversão de Dependência, 5
 Classes carregadas de dependência, evitando,
 55
 Ato de
 design de, 7–
 11 definição de, 4
 falhas em, 7–8
 julgamento, 10–11
 padrões, 6–7
 princípios, 5–6
 problemas resolvidos por, 2–3
 ferramentas,
 4–7 quando projetar, 8–10
 Adiamento de decisões
 de design, 193
 quando tomar, 22–23
 Falhas de design, exposição, 194
 Padrões de projeto, 6–7
 Padrões de projeto: elementos de
 Orientado a objetos reutilizáveis
 Software (Gama, Helm,
 Johnson e Vlissides), 6, 188
- Princípios de design, 5–6
 Ferramentas de design, 4–7
 Documentação
 de tipos de pato, 98 de
 funções, testes usados em, 212–13
 fornecimento, em testes, 193
 Objetos de domínio, 64, 83, 96, 199
 Duplas, testes de função para validar,
 224–29
 SECO (não se repita), 5, 24, 27, 28, 45, 50,
 196
 Tipos de pato, 219–29
 definição, 85
 documentação, 98

- encontrar, 90–94
 oculto, reconhecer, 96–98 ignorar,
 87 compartilhar
 entre, 99 testar papéis,
 219–24 confiar em, colocar,
 98 usar testes de papéis
 para validar duplas, 224–29 Duck
 digitação para
 relacionamentos
que se comportam como um, 189
 declarações de caso que ativam a
 classe, 96–98
 escolhendo patos com sabedoria, 99–
 100 código que depende de, escrita, 95–
 100 consequências de, 94–95
 custos reduzidos com, 85–104
 digitação dinâmica e, 100–104 medo
 de, conquista, 100–104 problema,
 composição, 87–90 agendamento,
 descoberta, 146–47 digitação estática e,
 100–102 resumo, 104 compreensão,
 85–95 digitação
 dinâmica abrangendo, 102–
4 digitação estática
 vs., 100– 102
- Tipos incorporados de herança**
 achado, 111–12
 múltiplo, 109–11
Interfaces explícitas, criação, 76–78
Mensagens externas, isolamento, 44–45
Responsabilidades extras
 extraídas de métodos, 29–31 isolados
 em aulas, 31–33
- Fábricas**, 51
tipo de dados de
 arquivo, 12 argumentos de ordem fixa,
 46–51 classe Fixnum, 13,
 14 Fowler, Martin, 191
classe Framework, vs. classe baseada
 em Ruby, 53–54
- Gama**, Erich, 6, 188
Gangue dos Quatro (Gof), 6, 188
Polegadas de engrenagem, 20–21
- tem* relacionamentos, 183, 190
 vs. *relacionamentos é-um*, 188–89
Hashes usados para argumentos de
 inicialização, 46–48
 Helm, Ricardo, 6, 188
- Patos escondidos**
 encontrando, 90–94
 reconhecendo, 96–98
Turma altamente coesa, 22
Mensagens de gancho, 134–38
 Caça, Andy, 5
- Mensagens recebidas, testes**,
 200–213
 injetando dependências como
 funções,
 208–13 injetando dependências
 usando classes, 207–8
interfaces, exclusão não
 utilizada, 202–3 isolando objeto em
 teste, 205–7 provando interface pública, 203
Código herdável, escrita, 158–62
 abstração, insistência, 159
 antipadrões, reconhecimento, 158–59
 classes, desacoplamento preventivo,
 161
 contrato, honrando, 159–60
 hierarquias superficiais, criando, 161–
 62 padrão
 de método de modelo, usando, 160
- Classe abstrata**
 de herança, descoberta, 116–29
 comportamento adquirido por meio
 de, 105–
 39 benefícios de, 184–
 85 escolha, 112–14
 clássico, 105–6
 composição e, decidindo entre,
 184–90 concreções e,
 106–9 consequências de,
 aceitando, 184–86 custos de, 185–
 86 tipos
 incorporados de,
 109–12 imagem da árvore
 genealógica de, 112 implicando,
 117 para
 relacionamentos é-a, 188–89
 aplicando incorretamente,
 114–16 múltiplo,
 112 problema resolvido por,
 112 reconhecendo onde usar, 106–14
 relacionamentos, desenho, 114
 regras de, 117
 simples, 112
 resumo, 139
- superclasses e subclasses**,
 acoplamento entre, 129–38
Código herdado, teste, 229–39
 comportamento, teste exclusivo, 236–
 39 interface herdada, especificação,
 229–32
 responsabilidades de
 subclasse,
 especificação, 233–36 Interface
 herdada,
 especificação, 229–32 Herdando
 o comportamento da função, 158
 Argumentos de inicialização,
 41–43 hashes usados para, 46–48 no
 isolamento da criação de
 instância, 42–43 Código de
 inicialização, 121 Injeção
 de dependências. Consulte Injeção de
 dependência Variáveis de instância,
 ocultação, 24–26 Intenção,
 construção,
 64–65 Teste intencional, 192–200 Interface herdada, ex-
 Interfaces. Veja também Privado
 interfaces; Interfaces públicas
 código apresentando sua melhor
 (interface, escrevendo, 76–79
 definindo, 61–63
 excluindo não utilizado, 202–
 3 dependências e, 62–63
 explícito, 76–78
 flexível, 59–83 Lei
 de Deméter e, 80–83
 responsabilidades e, 62–63
 resumo, 83
 compreensão, 59–61
 Princípio de segregação de interface, 5
 é_a?, 97
 relacionamentos é-a, 188–89 vs.
 relacionamentos tem-a, 188–89
Isolamento
 de dependências, 42–45 de
 externos mensagens, 44–45 de
 criação de instância, 42–43, 42–44 de
 inicialização multiparâmetro, 49–51 de
 objeto
 em teste, 205–7 de responsabilidades
 em classes, 31–33
- Java, 102, 118
 JavaScript, 106
 Johnson, Ralph, 6, 188

- Palavras-chave, 77–
78 *tipo_de?*, 97
- Lei de Demeter (LoD), 5, 80–83 definindo
Demeter, 80 projeto
Demeter, 5, 80 ouvindo
Demeter, 82–83 violações, 80–82
Probabilidade de
mudança, 53–57 em referências
incorporadas a mensagens, 45
vs. número de
dependentes, 55–57 compreensão, 53–54
- Liskov, Bárbara, 160
- Princípio de Substituição de Liskov (LSP), 5,
160, 230–31, 237, 239
- Código fracamente acoplado, escrita,
39–51
injetar dependências, 39–42 isolar
dependências, 42–45 remover
dependências de ordem de
argumento, 46–51
- Gerenciando dependências, 3
mensagens,
15 encadeamento de mensagens, 38–39,
80–83 mensagens. Consulte
também Mensagens
recebidas, testes de aplicativos,
criação, 76 delegação automática de
mensagens, 105–6 comando,
prova, 216–18
delegação, 82 externo,
isolamento, 44–45 entrada, teste,
200–213 probabilidade de
alteração, referências
incorporadas para, 45 encaminhamento
de mensagens via herança clássica, 112
objetos descobertos por, 74–76
consulta, ignorando, 215–16
testando saída, 215–18
- Metaprogramação, 102–3
Métodos
de responsabilidades extras extraídas
de, 29–31
agrupamento em classes, 16
wrapper, 24–25, 82
Métodos, olhando para cima, 154–58
simplificação grosseira, 154–55
explicação mais precisa, 155–56
- Métricas, 5, 10–11
- Meyer, Bertrand, 188
- MiniTeste, 200
- Definição
de módulos de
comportamento de função 143
compartilhado com, 141–62
- Remendos de macaco, 100
- Inicialização multiparâmetro,
isolamento, 49–51
- Herança múltipla, 112
- Voo espacial Goddard da NASA
Aplicações centrais, 6
- Nada, 48–49, 113
- Classe Zero, 113
- Ambiguidade
de classe de objeto sobre, 94–
95 verificação, 97, 111, 146
- Análise e Design Orientado a Objetos
(Booch), 188
- Design Orientado a Objetos (OOD), 1–
14. Consulte também
Dependências de design
gerenciadas por, 3
 mestres de, 188
visão geral de, 1 Linguagens orientadas a
objetos, 12–14 Programação orientada a objetos,
11–14
linguagens orientadas a objetos em,
12–14
visão geral de 11
linguagens procedurais em 12
objetos. Veja *também* objeto Peças
Combinado com composição,
domínio
163–90, 64, 83, 96, 199
mensagens usadas para descobrir, 74–
76 falando por si, 147 confiando nos
outros, 73–74
- Objeto em teste, 200, 202, 205–7
- Código aberto-fechado, 185
- Princípio Aberto-Fechado, 5, 185
- Métodos substituídos, 115
- Partes da
composição do objeto, 168–
76 criação, 169–72
- criando PartsFactory, 177–78
hierarquia, criando, 165–68
aproveitando PartsFactory, 178–80
tornando mais parecido com array, 172–
76 fabricação, 176–80
- Polimorfismo, 95
- Interfaces privadas
definindo, 61, 62
dependendo de, cautela em, 79
- Palavra-chave privada, 77–78
- Métodos privados, testes, 213–15
escolher, 214–15
ignorar, 213–14
remover da classe em teste,
214
- Linguagens de programação
digitadas estaticamente ou
dinamicamente,
100–104 sintaxe in, 118 *tipo* usado em, 85–86
- Falha na promoção, 122–23
- Palavra-chave protegida, 77–78
- Interfaces públicas
independência de contexto, busca,
71–73
- definição, 61, 62
exemplo de aplicação: empresa
de passeios de bicicleta, 63–
64 descoberta,
63–76 intenção, construção, 64–65
aplicação baseada em
mensagens,
criação, 76 mensagens usadas para
descobrir objetos, 74–76 de
outros, honrando,
78–79 provando, 203–4 diagramas de
sequência, usando, 65–69
confiando em outros objetos, 73–74
“o que”
vs. “como”, importância de, 69–71 Palavra-chave pública
- Mensagens de consulta, 196, 197, 215–16
- Refatorando
barreiras para, reduzindo, 215
definição de, 191 na
extração de responsabilidades extras de
métodos, 29–31
regra para,
123 estratégias, decidindo entre,
122–23

- testando funções e, 220–21, 226 na escrita de código mutável, 191–92
- Refatoração: Melhorando o Design de Código Existente* (Fowler), 191
- Relacionamentos, 188–90
- agregação e, 183–84 usam
 - composição para relacionamentos *tem-a*, 190 usam tipos
 - pato para relacionamentos *que se comportam como um*, 189 usam herança para é-a
 - relacionamentos, 188–89
- responde_to?, 97
- Responsabilidades, organização, 143–45
- Design Orientado a Responsabilidade (RDD), 22
- Invertendo a direção da dependência, 51–53
- Funções código concreto, escrita, 147–50
- descoberta, 142–43
 - código herdável, escrita, 158–62 injeção de dependências como, 208–13
 - comportamento de função
 - compartilhado com módulos, 141–62
 - resumo, 162
 - testes, em digitação de pato, 219–24
 - testes para documentar, 212–13
 - testes para validar duplas, 224–29
 - compreensão, 142–58 classe
 - baseada em Ruby vs. classe de estrutura, 53–
 - 54 erros de tipo de tempo de execução, 101, 103
- Diagramas de sequência, usando, 65–69
- Hierarquias superficiais na escrita de código herdável, 161–62
- Herança única, 112
- Responsabilidade única, classes com benefícios de, 31 código abraçando mudanças, escrita, 24–33 criação, 17–23 decisões de design, quando tomar, 22–23 design, 15–34 determinação, 22
- aplicação, 29–33 exemplo
- de aplicação: bicicletas e engrenagens, 17–21 responsabilidades
- extras e, 29–33 importância de, 21 roda real,
- 33–34 resumo, 34 Princípio de Responsabilidade Única, 5 classes de
- design com, 15–34 princípios de design SÓLIDOS , 5, 160 Repositório de código-fonte, 59 Linhas de código-fonte (SLOC), 10–11 Especializações, 117 Spike a problem, 198 Tipos de pato de digitação estática e, subvertendo com, 100–101
- vs. digitação dinâmica, 100–102 Classe String , 13–14 Tipo de dados String,
- 12, 13 Objetos String, 13–14 Confirmação de comportamento
- de subclasse, 233–34 testes, 236–37 Desacoplamento de subclasses usando mensagens de gancho, 134–38 superclasses e, acoplamento entre, 129–38 Comportamento de superclasse
- confirmando, 234–36 testando, 237–39
- Criação de superclasses, 117–20 subclasses e, acoplamento entre, 129–38
- Sintaxe, 118
- Dívida técnica, 11, 79 Padrão de método de modelo implementando cada, 127–29 usando, 125–27 na escrita de código herdável, 160
- Desenvolvimento Orientado a Testes (TDD), 199, 213
- Testando abstrações, suporte, 194
- bugs, descoberta, 193 custo-benefício, projeto, 191–240 criação de testes duplos, 210–11 decisões de projeto, adiamento, 193 falhas de projeto, exposição, 194 documentação, fornecimento, 193 tipos de patos, 219–29 mensagens recebidas, 200–213 código herdado, 229–39 testes intencionais, 192–200 saber como testar, 198–200 saber o que testar, 194–97 saber quando testar, 197–98 conhecer suas intenções, 193–94 mensagens enviadas, 215–18 privadas métodos, resumo 213–15, 240 para documentar funções, 212–13
- Testando mensagens de saída, 215–18 mensagens de comando, provando, 216–18 mensagens de consulta, ignorando, 215–16 Thomas, Dave, 5
- Toque de Classe: Aprendendo a Programar Bem com objetos e Contratos* (Meyer), 188 Acidente de trem, 80, 82, 83 código TRUE, 17 Tipos, 85. Veja também Duck digitação entre classes, 86 estático vs. dinâmico, 100–102 dentro da classe, 63 Tipo usado na classe, 111
- Linguagem de modelagem unificada (UML), 65–66, 114 Caso de uso, 64, 65, 66, 67, 69, 74
- Variáveis, definidoras, 12 Aula virtual, 61 Vlissides, Jon, 6, 188
- “O que” versus “como”, importância de, 69–71
- Wilkerson, Brian, 22 Wirs-Brock, Rebecca, 22 Tipos dentro da classe, 63 Método Wrapper, 24–25, 82