

Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship



Código Limpo

Série Robert C. Martin

A missão desta série é melhorar o estado da arte do artesanato de software.

Os livros desta série são técnicos, pragmáticos e substanciais. Os autores são artesãos e profissionais altamente experientes dedicados a escrever sobre o que realmente funciona na prática, em oposição ao que pode funcionar na teoria. Você lerá sobre o que o autor fez, não o que ele acha que você deveria fazer. Se o livro for sobre programação, haverá muito código. Se o livro for sobre gerenciamento, haverá muitos estudos de caso de projetos reais.

Estes são os livros que todos os praticantes sérios terão em suas estantes.

Estes são os livros que serão lembrados por fazerem a diferença e por orientarem os profissionais a se tornarem verdadeiros artesãos.

Gerenciando Projetos Ágeis

Agostinho sonhador

Estimativa e planejamento ágil

Mike Cohn

Trabalhando Efetivamente com Código Legado

Michael C. Feathers

Agile Java™: Criando Código com Desenvolvimento Orientado a Testes

Jeff Longer

Princípios, padrões e práticas ágeis em C#

Robert C. Martin e Micah Martin

Desenvolvimento Ágil de Software: Princípios, Padrões e Práticas

Robert C. Martin

Código Limpido: Um Manual de Artesanato de Software Ágil

Robert C. Martin

UML para programadores Java™

Robert C. Martin

Adequado para desenvolvimento de software: Framework para testes integrados

Rick Mugridge e Ward Cunningham

Desenvolvimento Ágil de Software com SCRUM

Ken Schwaber e Mike Beedle

Engenharia de software extrema: uma abordagem prática

Daniel H. Steinberg e Daniel W. Palmer

Para obter mais informações, visite informit.com/martinseries

Código Limpo

Um manual de agilidade **Artesanato de Software**

Os Mentores de Objetos:

Robert C. Martin

Michael C. Feathers Timothy R. Ottinger

Jeffrey J. Langr Brett L. Schuchert

James W. Grenning Kevin Dean Wampler

Object Mentor Inc.

Escrever código limpo é o que você deve fazer para se considerar um profissional.

Não há desculpa razoável para fazer algo menos do que o seu melhor.



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

PRENTICE
HALL

Cidade do Cabo • Sydney • Tóquio • Cingapura • Cidade do México

Muitas das designações usadas por fabricantes e vendedores para distinguir seus produtos são reivindicadas como marcas registradas. Onde 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 todas em maiúsculas.

Os autores e a editora 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 é assumida por danos acidentais ou consequenciais relacionados ou decorrentes do uso das informações ou programas aqui contidos.

A editora oferece excelentes descontos neste livro quando encomendado em quantidade para compras em massa 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@pearsoned.com

Inclui referências bibliográficas e índice.

ISBN 0-13-235088-2 (pbk.: papel alk.)

1. Desenvolvimento ágil de software. 2. Software de computador — Confiabilidade. I. Título.

2008 QA76.76.D47M3652 005.1

—dc22

2008024750

Direitos autorais © 2009 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 informações sobre permissões, escreva para:

Pearson Education, Inc.
Departamento de Direitos e Contratos
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-13-235088-4

ISBN-10: 0-13-235088-2 Texto

impresso nos Estados Unidos em papel reciclado na Courier em Stoughton, Massachusetts.

Primeira impressão julho de 2008

Para Ann Marie: O eterno amor da minha vida.

Esta página foi intencionalmente deixada em branco

Conteúdo

Prefácio.....	xix
Introdução	xxxv
Na capa.....	xxix
Capítulo 1: Código Limpo.....	1
Haverá Código	2
Código Inválido	3
O custo total de possuir uma bagunça.....	4
O Grande Redesign no Céu.....	
5 Atitude	5
O Enigma Primordial.....	6 A
Arte do Código Limpo?.....	6 O
que é Código Limpo?.....	7
Escolas de Pensamento	12
Somos Autores.....	
13 A regra do escoteiro	14
Prequela e Princípios.....	15
Conclusão.....	15
Bibliografia.....	15
Capítulo 2: Nomes Significativos.....	17
Introdução.....	17 Use
nomes que revelam intenções	18 Evite a
Desinformação.....	19 Faça
distinções significativas.....	20 Use
Nomes Pronunciáveis.....	21 Usar
nomes pesquisáveis.....	22

Evite Codificações.....	23
Notação húngara	23
Prefixos de membro.....	24
Interfaces e Implementações ..	24
Evite o Mapeamento Mental	25
Nomes de classe	25
Nomes de métodos	25
Não Seja Bonito	26
Escolha uma palavra por conceito.....	26
Não faça trocadilhos	26
Usar nomes de domínio da solução.....	27
Usar Nomes de Domínio de Problema.....	
27 Adicione um contexto significativo.....	27
Não Adicione Contexto Gratuito.....	29
Palavras Finais	30
 Capítulo 3: Funções.....	31
Pequeno!.....	34
Blocos e recuos.....	35 Faça
Uma Coisa.....	35
Seções dentro Funções	36 Um
nível de abstração por função	36 Lendo o
código de cima para baixo: a regra <i>Stepdown</i>	37 Instruções
Switch	37 Use nomes
descritivos.....	39 Argumentos de
função	40 Formas Monádicas
Comuns.....	41 Argumentos do
sinalizador	41 Funções
diádicas.....	42
Tríades.....	42
Objetos de Argumento.....	43
Listas de Argumentos.....	43
Verbos e palavras-chave.....	43
Não Tem Efeitos Colaterais.....	44
Argumentos de Saída	45
Separação de consulta de comando	45

Prefira Exceções a Retornar Códigos de Erro.....	46
Extrair Blocos Try/Catch.....	46
tratamento de erros é uma coisa.....	47
O ímã de dependência Error.java	47
Não se repita	48 Programação
Estruturada	48 Como você escreve funções assim?
Conclusão.....	49
SetupTeardownIncluder	50
Bibliografia.....	52
 Capítulo 4: Comentários.....	53
Comentários não compensam códigos ruins.....	55
Explique-se no código	55 Bons comentários.
Comentários Legais.....	55
Comentários Informativos.....	56
Explicação da Intenção.....	56
Esclarecimento.....	57
Aviso de Consequências.....	58 TODO
Comentários.....	58
Amplificação.....	59
Javadoc em APIs públicas.....	
59 Comentários Ruins	59
Resmungando	59
Comentários redundantes	60
Comentários enganosos.....	63
Comentários obrigatórios.....	63
Comentários do Diário.....	63
Comentários de Ruído	64
Ruído Assustador	66
Não Use um Comentário Quando Você Pode Usar uma Função ou uma Variável.....	67
Marcadores de Posição.....	67
Fechamento Comentários de colchetes.....	67 Atribuições e autorias.....

Código Comentado.....	68
Comentários HTML	69
Informações Não Locais	69
Demasiada Informação	
70 Conexão Inóvia.....	70
Cabeçalhos de Função.....	70
Javadoc em código não público	71
Exemplo.....	71
Bibliografia	74
Capítulo 5: Formatação	75 A
Finalidade da Formatação	76
Formatação Vertical	76
A Metáfora do Jornal	77
Abertura Vertical Entre Conceitos	78
Densidade Vertical.....	79
Distância Vertical	80
Ordenação Vertical.....	84 Formatação
horizontal	85 Abertura
horizontal e densidade	86 Alinhamento
horizontal.....	87
Recuo.....	88
Telescópios Fictícios.....	90
Regras da equipe	90
Regras de Formatação do Tio Bob	90
Capítulo 6: Objetos e Estruturas de Dados.....	93 Abstração
de Dados	93 Anti-
simetria de Dados/Objetos	95 A
Lei de Deméter	97 Destroços
de Trem	98
Híbridos	99
Escondendo a Estrutura	99
Objetos de Transferência de	
Dados	100 Registro
Ativo	101
Conclusão.....	101 Bibliografia.....

Capítulo 7: Tratamento de erros	103
Use exceções em vez de códigos de retorno.....	104
Escreva sua instrução Try-Catch-Finally primeiro105 Use	
exceções não verificadas.....106	
Forneça Contexto com Exceções.....107 Definir	
classes de exceção em termos das necessidades de um	
chamador.....107 Definir o fluxo	
normal109 Não Retorne	
Nulo.....110 Não Passe	
Nulo111	
Conclusão.....112 Bibliografia.....	
Capítulo 8: Limites.....113 Usando	
Código de Terceiros	114
Explorando e aprendendo Limites.....116	
Aprendizagem log4j.....116	
Testes de aprendizado são melhores que	
gratuitos.....118 Usando código que ainda	
não existe.....118 Limites	
Limpos120 Bibliografia.....	
Capítulo 9: Testes de unidade.....121	
As Três Leis de TDD	122
Mantendo os Testes Limpos	123
Testes Habilitam as -ilidades.....	
124 Testes de limpeza	
124 Linguagem de teste específica de domínio..	
127 A Padrão Duplo	127
Um Assert por Teste	130
Conceito Único por Teste.....131	
PRIMEIRO.....132	
Conclusão.....133	
Bibliografia133	
Capítulo 10: Aulas.....135	
Organização de classe	136
Encapsulamento136	

As Turmas Devem Ser Pequenas!	136
O Princípio da Responsabilidade Única.....	138
Coesão.....	140
Mantendo os Resultados da Coesão em Muitas Turmas Pequenas.....	141
Organizando para a Mudança	147
Isolando-se da Mudança.....	149
Bibliografia.....	151
Capítulo 11: Sistemas	153
Como você construiria uma cidade?	154
Separar a construção de um sistema de usá- lo	154
Separação do principal	155
Fábricas	155
Injeção de Dependência.....	157
Ampliação	157
Preocupações Transversais	160
Proxies Java	
161 Estruturas AOP Java puras	163
Aspectos J Aspectos	
166 Testar a arquitetura do sistema.....	166
Otimize a Tomada de Decisões	167
Use os padrões com sabedoria, quando agregarem valor demonstrável	168
168 Os sistemas precisam de linguagens específicas de domínio.....	168
Conclusão.....	169
Bibliografia	
Capítulo 12: Emergência	171
Limpando através do Design Emergente	171
Regra de Design Simples 1: Executa Todos os Testes	172
Regras de Design Simples 2–4:	
172 Refatoração	172
Sem Duplicação.....	173
Sem Expressivo.....	175
172 Classes Mínimas e Métodos.....	176
Conclusão	176
Bibliografia	
Capítulo 13: Simultaneidade	177
Por que simultaneidade?	178
Mitos e Equívocos.....	179

Desafios.....	180	Princípios
de defesa de simultaneidade	180	Princípio da
Responsabilidade Única	181	Corolário: Limite o
Escopo dos Dados.....	181	Corolário: Use cópias de
dados	181	Corolário: Threads devem ser tão
independentes quanto possível	182	Conheça o seu
Biblioteca	182	Coleções seguras para
threads.....	182	Conheça sua Execução
Modelos.....	183	Produtor-
Consumidor.....	184	Leitores-
Escritores.....	184	Jantares com
Filósofos	184	Cuidado com as
Dependências entre os Métodos Sincronizados	185	Mantenha as
seções sincronizadas pequenas.....	185	Escrever Código
de Desligamento Correto é Difícil.....	186	Testando
código encadeado	186	186 Trate falhas
espúrias como problemas de threading candidatos	187	Faça seu
código não encadeado funcionar primeiro.....	187	código não encadeado funcionar primeiro.....
código encadeado conectável	187	Tornar seu código
sintonizável.....	187	encadeado sintonizável.....
do que processadores.....	188	Executar com mais threads
plataformas	188	do que processadores.....
tentar e forçar falhas.....	188	Executar em diferentes
Codificado		plataformas
manualmente	189	Instrumente seu código para
Automatizado	189	tentar e forçar falhas.....
Conclusão.....	190	188 Codificado
Bibliografia.....	191	manualmente
Capítulo 14: Refinamento Sucessivo.....	193	Implementação
Args	194	Como eu fiz
isso?	200	Args: O
Rascunho	201	Então eu
parei	212	Sobre o
Incrementalismo	212	Incrementalismo
de String	214	Argumentos
Conclusão.....	250	

Capítulo 15: Interiores do JUnit.....	251	A
Estrutura JUnit	252	
Conclusão.....	265	
Capítulo 16: Refatorando SerialDate	267	
Primeiro, Faça Funciona	268	
Então faça certo.....	270	
Conclusão.....	284	
Bibliografia.....	284	
Capítulo 17: Cheiros e Heurísticas.....	285	
Comentários.....	286	
C1: Informações <i>Inapropriadas</i> ..	286	
C2: Comentário <i>obsoleto</i>	286	C3:
Comentário <i>redundante</i>	286	
C4: Comentário <i>mal escrito</i>	287	
C5: Código <i>Comentado</i>	287	Meio
Ambiente	287	
E1: <i>Construir requer mais de uma etapa</i>	287	
E2: <i>Testes Requerem Mais de uma Etapa</i>	287	
Funções.....	288	
F1: <i>Demasiados Argumentos</i>	288	
F2: <i>Argumentos de Saída</i>	288	
F3: <i>Argumentos do sinalizador</i>	288	
F4: <i>Função Morta</i>	288	
Geral.....	288	
G1: <i>Múltiplos idiomas em um arquivo de origem</i>	288	G2:
Comportamento óbvio Não está implementado.....	288	
G3: <i>Comportamento incorreto nos limites..</i>	289	G4:
Seguranças Substituídas	289	
G5: <i>Duplicação</i>	289	G6:
Código em nível de abstração incorreto.....	290	
G7: <i>Classes Base Dependentes de Seus Derivados</i>	291	
G8: <i>Demasiada Informação</i>	291	
G9: <i>Código Morto</i>	292	
G10: <i>Separação Vertical</i>	292	
G11: <i>Inconsistência</i>	292	
G12: Desorganização	293	

G13: Acoplamento Artificial	293
G14: Inveja de recursos	293
G15 : Argumentos do Seletor	294
G16: Intenção Obscura	295
G17: Responsabilidade Desviada	
.295 G18: Estática Inapropriada	296
G19: Usar Variáveis Explicativas	296
G20: Os nomes das funções devem dizer o que fazem	297
G21: Entenda o Algoritmo	297
G22: Tornar as Dependências Lógicas Físicas.....	298
G23: Prefira Polimorfismo a If/Else ou Switch/Case	299
G24: Siga as Convenções Padrão.....	299
Números Mágicos por Constantes Nomeadas.....	300
G26: Seja Preciso.....	301
G27: Estrutura acima da Convenção.....	301
G28: Encapsular Condicionais	301
G29: Evite Condicionais Negativos.....	302
G30: Funções Devem Fazer Uma Coisa	302
G31: Acoplamentos temporais ocultos	302
G32: Não seja arbitrário..	303
G33: Encapsular Condições de Fronteira.....	304
G34: As funções devem descer apenas um nível de abstração	304
G35: Mantenha os dados configuráveis em níveis altos.....	306
G36: Evitar Navegação Transitiva.....	306
Java	307
J1: Evite listas de importação longas usando curingas.....	307
J2: Não Herdar Constantes	307
J3: Constantes versus Enums.....	308
Nomes.....	309
N1: Escolher nomes descritivos.....	309
N2: Escolha Nomes no Nível Apropriado de Abstração.....	311
N3: Use Nomenclatura Padrão Sempre que Possível.....	311
N4: Nomes inequívocos	312
N5: Use nomes longos para escopos longos..	312
N6: Evite C.....	

Testes.....	.313 T1:
Testes <i>Insuficientes</i>313 T2: <i>Use uma Ferramenta de Cobertura!</i>313
T3 : <i>Não Ignore Testes Triviais</i>313 T4: <i>Um teste ignorado é uma pergunta sobre uma ambigüidade.</i>313 T5:
Condições de Limite de Teste314 T6:
Teste Exaustivo Perto de Bugs.....314 T7: <i>Padrões de falha são reveladores</i>
<i>Cobertura de Teste Podem Ser Reveladores</i>314 T8: <i>Padrões de testes devem ser rápidos.</i>314
Conclusão.....314
Bibliografia.....315
Apêndice A: Simultaneidade II.....317 Exemplo de
Cliente/Servidor317 O
Servidor317 Adicionando
encadeamento.....319 Observações
do servidor319
Conclusão.....321 Possíveis
Caminhos de Execução321 Número de
Caminhos.....322 Indo Mais
Fundo323
Conclusão...326
Conhecendo sua biblioteca.....326
Estrutura do Executor326 Soluções
sem bloqueio.....327 Classes não seguras
para thread.....328
Dependências Entre Métodos Podem	
Quebrar Códigos Simultâneos329 Tolerar
a falha.....330 Bloqueio Baseado no
Cliente..330 Bloqueio baseado em
servidor332 Aumentando o
Rendimento333 Cálculo de Thread
Único de Rendimento.....334 Cálculo multithread de taxa de
transferência.....335
Impasse.....335 Exclusão
mútua336 Bloquear e
aguardar337

Sem Preempção	337
Circular	337
a Exclusão Mútua.....	337
Bloqueio e Espera.....	337
Preempção de quebra.....	338
Espera Circular.....	338
Teste Código	
multithread	339 Suporte de ferramenta
para teste de código baseado em thread	342
Conclusão.....	342
Tutorial: Completo Exemplos de código	343
Cliente/Servidor Não Encadeado.....	343
Cliente/Servidor Usando Threads.....	346
Apêndice B: org.jfree.date.SerialDate	349
Apêndice C: Referências Cruzadas de Heurísticas	409
Epílogo.....	411
Índice	413

Esta página foi intencionalmente deixada em branco

Prefácio

Um dos nossos doces favoritos aqui na Dinamarca é o Ga-Jol, cujos fortes vapores de alcaçuz são um complemento perfeito para o nosso clima úmido e muitas vezes frio. Parte do charme de Ga-Jol para nós, dinamarqueses, são os ditos sábios ou espirituosos impressos na aba de cada caixa. Comprei dois pacotes da iguaria esta manhã e descobri que trazia esta velha serra dinamarquesa:

Honestidade nas pequenas coisas não é pouca coisa.

"Honestidade nas pequenas coisas não é uma coisa pequena." Foi um bom presságio condizente com o que eu já queria dizer aqui. Pequenas coisas importam. Este é um livro sobre preocupações humildes cujo valor, no entanto, está longe de ser pequeno.

Deus está nos detalhes, disse o arquiteto Ludwig Mies van der Rohe. Esta citação lembra argumentos contemporâneos sobre o papel da arquitetura no desenvolvimento de software e, particularmente, no mundo Ágil. Bob e eu ocasionalmente nos encontramos apaixonadamente envolvidos nesse diálogo. E sim, Mies van der Rohe estava atento à utilidade e às formas atemporais de construção que fundamentam a grande arquitetura. Por outro lado, ele também selecionou pessoalmente cada maçaneta para cada casa que projetou. Por que? Porque as pequenas coisas importam.

Em nosso "debate" contínuo sobre TDD, Bob e eu descobrimos que concordamos que a arquitetura de software tem um lugar importante no desenvolvimento, embora provavelmente tenhamos visões diferentes sobre exatamente o que isso significa. Tais sofismas são relativamente sem importância, no entanto, porque podemos aceitar como certo que profissionais responsáveis dedicuem *algum* tempo para pensar e planejar no início de um projeto. As noções de design do final da década de 1990 impulsionadas *apenas* pelos testes e pelo código já se foram. No entanto, a atenção aos detalhes é uma base ainda mais crítica do profissionalismo do que qualquer grande visão. Primeiro, é por meio da prática em pequena escala que os profissionais ganham proficiência e confiança para a prática em grande escala. Em segundo lugar, a menor construção desleixada, da porta que não fecha bem ou do ladrilho levemente torto no chão, ou mesmo da mesa bagunçada, dissipam completamente o charme do todo maior. É disso que se trata o código limpo.

Ainda assim, a arquitetura é apenas uma metáfora para o desenvolvimento de software e, em particular, para a parte do software que entrega o *produto* inicial da mesma forma que um arquiteto entrega um edifício impecável. Nestes dias de Scrum e Agile, o foco está em trazer *produtos* rapidamente para o mercado. Queremos que a fábrica funcione em alta velocidade para produzir software. Estas são fábricas humanas: codificadores que pensam e sentem que estão trabalhando a partir de um registro de produto ou história do usuário para criar o *produto*. A metáfora da manufatura parece cada vez mais forte em tal pensamento. Os aspectos de produção da fabricação de automóveis japonesa, de um mundo de linha de montagem, inspiram muito do Scrum.

No entanto, mesmo na indústria automobilística, a maior parte do trabalho não está na fabricação, mas na manutenção - ou em sua prevenção. Em software, 80% ou mais do que fazemos é curiosamente chamado de "manutenção": o ato de reparar. Em vez de adotar o típico foco ocidental na *produção* de um bom software, deveríamos pensar mais como reparadores domésticos na indústria da construção ou mecânicos de automóveis no campo automotivo. O que a administração japonesa tem a dizer sobre isso?

Por volta de 1951, uma abordagem de qualidade chamada Total Productive Maintenance (TPM) surgiu no cenário japonês. Seu foco está na manutenção e não na produção. Um dos grandes pilares do TPM é o conjunto dos chamados princípios 5S. 5S é um conjunto de disciplinas – e aqui uso o termo “disciplina” de forma instrutiva. Esses princípios 5S são, de fato, os fundamentos do Lean – outra palavra da moda no cenário ocidental e uma palavra da moda cada vez mais proeminente nos círculos de software. Esses princípios não são uma opção. Como Uncle Bob relata em seu material inicial, a boa prática de software requer tal disciplina: foco, presença de espírito e pensamento. Nem sempre se trata apenas de fazer, de forçar o equipamento da fábrica a produzir na velocidade ideal. A filosofia 5S comprehende estes conceitos:

- *Seiri*, ou organização (pense em “classificar” em inglês). Saber onde estão as coisas – usando abordagens como nomes adequados – é crucial. Você acha que nomear identificadores não é importante? Leia nos próximos capítulos.
- *Seiton*, ou arrumação (pense em “sistematizar” em inglês). Há um velho ditado americano: *Um lugar para cada coisa e cada coisa em seu lugar*. Um pedaço de código deve estar onde você espera encontrá-lo - e, se não estiver, você deve refatorá-lo para chegar lá. •
- Seiso*, ou limpeza (pense em “brilhar” em inglês): mantenha o local de trabalho livre de fios pendurados, graxa, restos e resíduos. O que os autores aqui dizem sobre encher seu código com comentários e linhas de código comentadas que capturam o histórico ou desejos para o futuro? Livrar-se deles.
- *Seiketsu*, ou padronização: O grupo concorda em como manter o local de trabalho limpo. Você acha que este livro diz algo sobre ter um estilo de codificação consistente e um conjunto de práticas dentro do grupo? De onde vêm esses padrões? Leia. • *Shitsuke*, ou disciplina (autodisciplina). Isso significa ter disciplina para seguir as práticas e refletir frequentemente sobre o próprio trabalho e estar disposto a mudar.

Se você aceitar o desafio — sim, o desafio — de ler e aplicar este livro, compreenderá e apreciará o último ponto. Aqui, estamos finalmente indo às raízes do profissionalismo responsável em uma profissão que deve se preocupar com o ciclo de vida de um produto. À medida que mantemos automóveis e outras máquinas sob TPM, a manutenção avariada – esperando que os bugs apareçam – é a exceção. Em vez disso, subimos um nível: inspecionamos as máquinas todos os dias e consertamos as peças de desgaste antes que quebrem, ou fazemos o equivalente à proverbial troca de óleo de 10.000 milhas para evitar o desgaste. No código, refatore impiedosamente. Você pode melhorar ainda mais um nível, já que o movimento TPM inovou há mais de 50 anos: construir máquinas que são mais fáceis de manter em primeiro lugar. Tornar seu código legível é tão importante quanto torná-lo executável. A prática definitiva, introduzida nos círculos de TPM por volta de 1960, é focar na introdução de máquinas totalmente novas ou

substituindo os antigos. Como Fred Brooks nos adverte, provavelmente deveríamos refazer os principais pedaços de software do zero a cada sete anos ou mais para varrer a sujeira rastejante. Talvez devêssemos atualizar a constante de tempo de Brooks para uma ordem de semanas, dias ou horas em vez de anos. É aí que reside o detalhe.

Há um grande poder nos detalhes, mas há algo de humilde e profundo nessa abordagem da vida, como poderíamos esperar estereotipadamente de qualquer abordagem que reivindique raízes japonesas. Mas esta não é apenas uma visão oriental da vida; A sabedoria popular inglesa e americana está cheia de tais admoestações. A citação de Seiton acima fluiu da pena de um ministro de Ohio que literalmente via a limpeza “como um remédio para todos os graus de mal”. Que tal Seiso? A *limpeza está próxima da divindade*. Por mais bonita que seja uma casa, uma mesa bagunçada rouba seu esplendor. Que tal Shutsuke nessas pequenas questões? *Quem é fiel no pouco, é fiel no muito*. Que tal estar ansioso para refatorar no momento responsável, fortalecendo sua posição para as “grandes” decisões subseqüentes, em vez de adiar? *Um ponto no tempo salva nove. Deus ajuda quem cedo madruga. Não deixe para amanhã o que você pode fazer hoje.* (Tal era o sentido original da frase “o último momento responsável” em Lean até que caiu nas mãos de consultores de software.) Que tal calibrar o lugar de pequenos esforços individuais em um grande todo? *Grandes carvalhos crescem de pequenas bolotas*. Ou que tal integrar o trabalho preventivo simples à vida cotidiana? *Um grama de prevenção vale um quilo de cura. Uma maçã por dia mantém o médico longe.* O código limpo honra as raízes profundas da sabedoria sob nossa cultura mais ampla, ou nossa cultura como era, ou deveria ser, e *pode* ser com atenção aos detalhes.

Mesmo na grande literatura arquitetônica encontramos serras que remontam a esses supostos detalhes. Pense nas maçanetas de Mies van der Rohe. Isso é *seiri*. Isso é estar atento a cada nome de variável. Você deve nomear uma variável com o mesmo cuidado com que nomeia um primogênito.

Como todo proprietário sabe, esse cuidado e refinamento contínuo nunca terminam. O arquiteto Christopher Alexander - pai dos padrões e linguagens de padrões - vê cada ato de design em si como um pequeno ato local de reparo. E ele vê o artesanato da estrutura fina como sendo da competência exclusiva do arquiteto; as formas maiores podem ser deixadas para padrões e sua aplicação pelos habitantes. O design está sempre em andamento, não apenas quando adicionamos um novo cômodo a uma casa, mas também quando estamos atentos à repintura, substituição de tapetes gastos ou reforma da pia da cozinha. A maioria das artes ecoa sentimentos análogos. Em nossa busca por outros que atribuem a morada de Deus aos detalhes, nos encontramos na boa companhia do autor francês do século XIX, Gustav Flaubert. O poeta francês Paul Valéry nos adverte que um poema nunca está pronto e é continuamente retrabalhado, e parar de trabalhá-lo é abandono. Tal preocupação com o detalhe é comum a todos os empreendimentos de excelência. Portanto, talvez haja pouca novidade aqui, mas ao ler este livro você será desafiado a adotar boas disciplinas que há muito tempo se rendeu à apatia ou ao desejo de espontaneidade e apenas “responder à mudança”.

Infelizmente, geralmente não vemos essas preocupações como pilares fundamentais da arte da programação. Abandonamos nosso código cedo, não porque está pronto, mas porque nosso sistema de valores se concentra mais na aparência externa do que na substância do que entregamos.

Essa desatenção nos custa no final: *sempre aparece uma moeda ruim*. A pesquisa, nem na indústria nem na academia, se humilha à posição humilde de manter o código limpo. Nos meus dias de trabalho na organização Bell Labs Software Production Research (*Produção*, de fato!), tivemos algumas descobertas no verso do envelope que sugeriam que o estilo de indentação consistente era um dos indicadores estatisticamente mais significativos de baixa densidade de bugs.

Queremos que a arquitetura ou linguagem de programação ou alguma outra noção elevada seja a causa da qualidade; como pessoas cujo suposto profissionalismo se deve ao domínio de ferramentas e métodos de design elevados, nos sentimos insultados pelo valor que essas máquinas de chão de fábrica, os codificadores, agregam por meio da aplicação simples e consistente de um estilo de indentação. Para citar meu próprio livro de 17 anos atrás, esse estilo distingue a excelência da mera competência. A cosmovisão japonesa comprehende o valor crucial do trabalhador cotidiano e, mais ainda, dos sistemas de desenvolvimento que se devem às ações simples e cotidianas desses trabalhadores. A qualidade é o resultado de um milhão de atos altruístas de cuidado - não apenas de qualquer grande método que desce dos céus. O fato de esses atos serem simples não significa que sejam simplistas e dificilmente significa que sejam fáceis. Eles são, no entanto, o tecido da grandeza e, mais ainda, da beleza, em qualquer empreendimento humano. Ignorá-los ainda não é ser totalmente humano.

Claro, ainda sou um defensor de pensar em um escopo mais amplo e, particularmente, do valor das abordagens arquitetônicas enraizadas no conhecimento profundo do domínio e na usabilidade do software. O livro não é sobre isso - ou, pelo menos, não é obviamente sobre isso. Este livro tem uma mensagem mais sutil cuja profundidade não deve ser subestimada. Ele se encaixa na visão atual de pessoas realmente baseadas em código, como Peter Sommerlad, Kevlin Henney e Giovanni Asproni. "O código é o design" e "Código simples" são seus mantras. Embora devamos ter o cuidado de lembrar que a interface é o programa e que suas estruturas têm muito a dizer sobre a estrutura do nosso programa, é crucial adotar continuamente a postura humilde de que o design vive no código. E enquanto o retrabalho na metáfora da manufatura leva ao custo, o retrabalho no design leva ao valor. Devemos ver nosso código como a bela articulação de nobres esforços de design - design como um processo, não um ponto final estático. É no código que as métricas arquitetônicas de acoplamento e coesão funcionam. Se você ouvir Larry Constantine descrever o acoplamento e a coesão, verá que ele fala em termos de código — não em conceitos abstratos grandiosos que podem ser encontrados na UML. Richard Gabriel nos aconselha em seu ensaio, "Abstração Descant" que a abstração é má. O código é anti-mal e o código limpo talvez seja divino.

Voltando à minha caixinha de Ga-Jol, acho importante ressaltar que a sabedoria dinamarquesa nos aconselha não apenas a prestar atenção nas pequenas coisas, mas também a ser honesto nas pequenas coisas. Isso significa ser honesto com o código, honesto com nossos colegas sobre o estado de nosso código e, acima de tudo, ser honesto conosco sobre nosso código. Fizemos o possível para "deixar o acampamento mais limpo do que o encontramos"? Refatoramos nosso código antes de fazer o check-in? Essas não são preocupações periféricas, mas preocupações que estão exatamente no centro dos valores ágeis. É uma prática recomendada no Scrum que a refatoração seja parte do conceito de "Pronto". Nem a arquitetura nem o código limpo insistem na perfeição, apenas na honestidade e em fazer o melhor que podemos. *Errar é humano; perdoar, divino*. No Scrum, tornamos tudo visível. Lavamos nossa roupa suja. Somos honestos sobre o estado do nosso código porque

código nunca é perfeito. Tornamo-nos mais plenamente humanos, mais dignos do divino e mais próximos dessa grandeza nos detalhes.

Em nossa profissão, precisamos desesperadamente de toda a ajuda possível. Se um chão de fábrica limpo reduz acidentes e ferramentas de oficina bem organizadas aumentam a produtividade, então eu sou totalmente a favor. Quanto a este livro, é a melhor aplicação pragmática dos princípios Lean ao software que já vi impressa. Eu não esperava menos desse pequeno grupo prático de indivíduos pensantes que há anos lutam juntos não apenas para se tornarem melhores, mas também para doar seus conhecimentos à indústria em obras como as que você agora encontra em suas mãos. Deixa o mundo um pouco melhor do que eu o encontrei antes de o tio Bob me enviar o manuscrito.

Tendo concluído este exercício de insights elevados, vou limpar minha mesa.

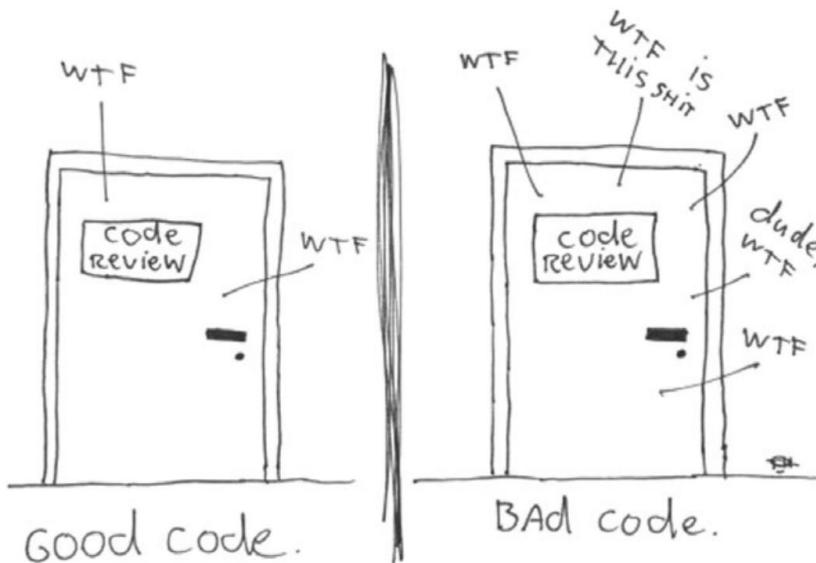
James O. Coplien

Mørdrup, Dinamarca

Esta página foi intencionalmente deixada em branco

Introdução

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



Reproduzido com a gentil permissão de Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Qual porta representa seu código? Qual porta representa sua equipe ou sua empresa?
Por que estamos naquela sala? Esta é apenas uma revisão de código normal ou encontramos um fluxo de problemas horríveis logo após a ativação? Estamos depurando em pânico, examinando o código que pensávamos funcionar? Os clientes estão saindo em massa e os gerentes respirando fundo

nossos pescoços? Como podemos ter certeza de que acabamos atrás da porta *certa* quando as coisas ficam difíceis? A resposta é: artesanato.

Há duas partes para aprender artesanato: conhecimento e trabalho. Você deve obter o conhecimento de princípios, padrões, práticas e heurísticas que um artesão conhece, e também deve cravar esse conhecimento em seus dedos, olhos e intestino trabalhando duro e praticando.

Eu posso te ensinar a física de andar de bicicleta. De fato, a matemática clássica é relativamente direta. Gravidade, fricção, momento angular, centro de massa e assim por diante, podem ser demonstrados com menos de uma página cheia de equações. Com essas fórmulas pude provar a você que andar de bicicleta é prático e dar a você todo o conhecimento necessário para fazê-la funcionar. E você ainda cairia na primeira vez que subisse naquela bicicleta.

A codificação não é diferente. Poderíamos escrever todos os princípios de "sentir-se bem" do código limpo e confiar em você para fazer o trabalho (em outras palavras, deixá-lo cair quando subir na bicicleta), mas que tipo de professores isso nos tornaria, e que tipo de aluno isso faria de você?

Não. Não é assim que este livro vai funcionar.

Aprender a escrever código limpo é um *trabalho árduo*. Requer mais do que apenas o conhecimento de princípios e padrões. Você deve *suar* por isso. Você deve praticá-lo sozinho e observar a si mesmo falhar. Você deve observar os outros praticando e falhando. Você deve vê-los tropeçar e refazer seus passos. Você deve vê-los agonizar com as decisões e ver o preço que pagam por tomar essas decisões da maneira errada.

Esteja preparado para trabalhar duro enquanto lê este livro. Este não é um livro de "sentir-se bem" que você pode ler em um avião e terminar antes de pousar. Este livro vai fazer você trabalhar, e *trabalhar duro*. Que tipo de trabalho você vai fazer? Você estará lendo código — muito código. E você será desafiado a pensar sobre o que há de certo nesse código e o que há de errado com ele. Você será solicitado a acompanhar enquanto desmontamos os módulos e os juntamos novamente. Isso levará tempo e esforço; mas achamos que valerá a pena.

Dividimos este livro em três partes. Os primeiros capítulos descrevem os princípios, padrões e práticas de escrever código limpo. Há bastante código nesses capítulos, e eles serão difíceis de ler. Eles vão prepará-lo para a segunda seção que está por vir. Se você largou o livro depois de ler a primeira seção, boa sorte para você!

A segunda parte do livro é o trabalho mais difícil. Consiste em vários estudos de caso de complexidade cada vez maior. Cada estudo de caso é um exercício de limpeza de algum código — de transformação de código que apresenta alguns problemas em código com menos problemas. O detalhe nesta seção é *intenso*. Você terá que alternar entre a narrativa e as listagens de código. Você terá que analisar e entender o código com o qual estamos trabalhando e percorrer nosso raciocínio para fazer cada mudança que fazemos. Reserve algum tempo porque *isso deve levar dias*.

A terceira parte deste livro é a recompensa. É um único capítulo contendo uma lista de heurísticas e cheiros coletados durante a criação dos estudos de caso. À medida que examinamos e limpamos o código nos estudos de caso, documentamos todos os motivos de nossas ações como um

heurística ou olfativa. Tentamos entender nossas próprias reações ao código que estávamos lendo e alterando e trabalhamos duro para entender por que sentimos o que sentimos e fizemos o que fizemos. O resultado é uma base de conhecimento que descreve a maneira como pensamos quando escrevemos, lemos e limpamos o código.

Esta base de conhecimento é de valor limitado se você não fizer o trabalho de ler cuidadosamente os estudos de caso na segunda parte deste livro. Nesses estudos de caso, anotamos cuidadosamente cada alteração que fizemos com referências futuras às heurísticas. Essas referências futuras aparecem entre colchetes assim: [H22]. Isso permite que você veja o *contexto* em que essas heurísticas foram aplicadas e escritas! Não são as próprias heurísticas que são tão valiosas, é a *relação entre essas heurísticas e as decisões discretas que tomamos enquanto limpamos o código nos estudos de caso*.

Para ajudá-lo ainda mais com essas relações, colocamos uma referência cruzada no final do livro que mostra o número da página para cada referência futura. Você pode usá-lo para pesquisar cada local onde uma determinada heurística foi aplicada.

Se você ler a primeira e a terceira seções e pular os estudos de caso, terá lido mais um livro do tipo “sinta-se bem” sobre como escrever um bom software. Mas se você reservar um tempo para trabalhar com os estudos de caso, seguindo cada pequeno passo, cada decisão minuciosa - se você se colocar em nosso lugar e se迫使 a pensar nos mesmos caminhos que pensamos, então você ganhará uma experiência muito mais rica. compreensão desses princípios, padrões, práticas e heurísticas. Eles não serão mais conhecimentos de “sentir-se bem”. Eles terão sido esmagados em seu intestino, dedos e coração. Eles se tornarão parte de você da mesma forma que uma bicicleta se torna uma extensão de sua vontade quando você domina como andar nela.

Agradecimentos

Obra de arte

Obrigado aos meus dois artistas, Jeniffer Kohnke e Angela Brooks. Jennifer é responsável pelas fotos impressionantes e criativas no início de cada capítulo e também pelos retratos de Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers e eu.

Ângela é a responsável pelas imagens inteligentes que enfeitam o interior de cada capítulo. Ela fez algumas fotos para mim ao longo dos anos, incluindo muitas das fotos internas de *Agile Software Development: Principles, Patterns, and Practices*. Ela também é minha primogênita, em quem me comproazo.

Esta página foi intencionalmente deixada em branco

Na cobertura

A imagem na capa é M104: The Sombrero Galaxy. M104 está localizado em Virgem e fica a pouco menos de 30 milhões de anos-luz de nós. Em seu núcleo está um buraco negro supermassivo pesando cerca de um bilhão de massas solares.

A imagem lembra a explosão da lua poderosa Klingon *Praxis*? Lembro-me vividamente da cena em *Star Trek VI* que mostrava um anel equatorial de detritos voando para longe daquela explosão. Desde aquela cena, o anel equatorial tem sido um artefato comum em explosões de filmes de ficção científica. Foi até adicionado à explosão de Alderaan em edições posteriores do primeiro filme de *Star Wars*.

O que causou a formação deste anel em torno de M104? Por que tem uma protuberância central tão grande e um núcleo tão brilhante e minúsculo? Parece-me que o buraco negro central perdeu a calma e abriu um buraco de 30.000 anos-luz no meio da galáxia. Ai se abateu sobre qualquer civilização que pudesse estar no caminho dessa ruptura cósmica.

Buracos negros supermassivos engolem estrelas inteiras como almoço, convertendo uma fração considerável de sua massa em energia. $E = MC^2$ é alavancagem suficiente, mas quando M é uma massa estelar: Cuidado! Quantas estrelas caíram de cabeça naquela boca antes que o monstro estivesse saciado? O tamanho do vazio central poderia ser uma dica?

A imagem de M104 na capa é uma combinação da famosa fotografia de luz visível do Hubble (à direita) e a imagem infravermelha recente do observatório orbital Spitzer (abaixo, à direita). É a imagem infravermelha que nos mostra claramente a natureza anelar da galáxia. Na luz visível, vemos apenas a borda frontal do anel em silhueta. A protuberância central obscurece o resto do anel.



Mas no infravermelho, as partículas quentes no anel brilham através da protuberância central. As duas imagens combinadas nos dão uma visão que nunca vimos antes e implicam que há muito tempo atrás era um inferno de atividade.

Imagen da capa: © Telescópio Espacial Spitzer

Esta página foi intencionalmente deixada em branco

1

Código Limpo



Você está lendo este livro por dois motivos. Primeiro, você é um programador. Em segundo lugar, você quer ser um programador melhor. Bom. Precisamos de programadores melhores.

Este é um livro sobre boa programação. Ele é preenchido com código. Vamos olhar para o código de todas as direções diferentes. Vamos olhar de cima para baixo, de baixo para cima e de dentro para fora. Quando terminarmos, saberemos muito sobre código. Além do mais, seremos capazes de dizer a diferença entre código bom e código ruim. Nós saberemos como escrever um bom código. E saberemos como transformar código ruim em código bom.

Haverá Código

Pode-se argumentar que um livro sobre código está de alguma forma atrasado - esse código não é mais o problema; que devemos nos preocupar com modelos e requisitos.

Na verdade, alguns sugeriram que estamos perto do fim do código. Que em breve todo código será gerado ao invés de escrito. Que os programadores simplesmente não serão necessários porque os empresários irão gerar programas a partir de especificações.

Absurdo! Nunca nos livraremos do código, porque o código representa os detalhes dos requisitos. Em algum nível, esses detalhes não podem ser ignorados ou abstraídos; eles devem ser especificados. E especificar os requisitos com tantos detalhes que uma máquina possa executá-los é *programação*. Tal especificação é *código*.

Espero que o nível de abstração de nossas linguagens continue aumentando. Também espero que o número de idiomas específicos de domínio continue a crescer. Isso será uma coisa boa. Mas não eliminará o código. De fato, todas as especificações escritas nessas linguagens de nível superior e específicas de domínio serão *códigos*! Ele ainda precisará ser rigoroso, preciso e tão formal e detalhado que uma máquina possa entendê-lo e executá-lo.

As pessoas que pensam que o código um dia vai desaparecer são como os matemáticos que esperam um dia descobrir uma matemática que não precisa ser formal. Eles esperam que um dia descubramos uma maneira de criar máquinas que possam fazer o que queremos, e não o que dizemos. Essas máquinas terão que ser capazes de nos entender tão bem que possam traduzir necessidades vagamente especificadas em programas de execução perfeita que atendam precisamente a essas necessidades.

Isso nunca vai acontecer. Nem mesmo os humanos, com toda a sua intuição e criatividade, conseguiram criar sistemas de sucesso a partir dos vagos sentimentos de seus clientes.

De fato, se a disciplina de especificação de requisitos nos ensinou alguma coisa, é que requisitos bem especificados são tão formais quanto o código e podem atuar como testes executáveis desse código!

Lembre-se de que o código é realmente a linguagem na qual expressamos os requisitos. Podemos criar linguagens mais próximas dos requisitos. Podemos criar ferramentas que nos ajudem a analisar e montar esses requisitos em estruturas formais. Mas nunca eliminaremos a precisão necessária - então sempre haverá código.

código ruim

Recentemente, li o prefácio do livro de Kent Beck, *Implementation Patterns*.¹ Ele diz: “... este livro é baseado em uma premissa bastante frágil: que um bom código é importante. . .” Uma premissa frágil? Discordo! Acho que essa premissa é uma das mais robustas, sustentadas e sobreacarregadas de todas as premissas de nosso ofício (e acho que Kent sabe disso). Sabemos que um bom código é importante porque tivemos que lidar por muito tempo com sua falta.

Conheço uma empresa que, no final dos anos 80, criou um aplicativo *matador*. Foi muito popular e muitos profissionais compraram e usaram. Mas então os ciclos de lançamento começaram a se estender. Os bugs não foram reparados de uma versão para outra. Os tempos de carregamento aumentaram e as falhas aumentaram. Lembro-me do dia em que desliguei o produto frustrado e nunca mais o usei. A empresa faliu pouco tempo depois disso.

Duas décadas depois, conheci um dos primeiros funcionários daquela empresa e perguntei o que havia acontecido. A resposta confirmou meus temores. Eles apressaram o produto para o mercado e fizeram uma grande bagunça no código. À medida que adicionavam mais e mais recursos, o código ficava cada vez pior até que eles simplesmente não conseguiam mais administrá-lo. *Foi o código ruim que derrubou a empresa.*

Você já foi significativamente impedido por um código ruim? Se você é um programador com alguma experiência, já sentiu esse impedimento muitas vezes. De fato, temos um nome para isso. Chamamos de *vadear*. Nós passamos por códigos ruins. Caminhamos por um pântano de arbustos emaranhados e armadilhas escondidas. Lutamos para encontrar nosso caminho, esperando por alguma dica, alguma pista do que está acontecendo; mas tudo o que vemos é um código cada vez mais sem sentido.

Claro que você foi impedido por um código ruim. Então, por que você o escreveu?

Você estava tentando ir rápido? Você estava com pressa? Provavelmente sim. Talvez você tenha sentido que não tinha tempo para fazer um bom trabalho; que seu chefe ficaria zangado com você se você se desse ao trabalho de limpar seu código. Talvez você estivesse apenas cansado de trabalhar neste programa e queria que acabasse. Ou talvez você tenha olhado para o acúmulo de outras coisas que havia prometido fazer e percebeu que precisava fechar esse módulo para poder passar para o próximo. Todos nós já fizemos isso.

Todos nós olhamos para a bagunça que acabamos de fazer e decidimos deixar para outro dia. Todos nós sentimos o alívio de ver nosso programa confuso funcionar e decidir que um



1. [Beck07].

bagunça de trabalho é melhor do que nada. Todos nós dissemos que iríamos voltar e limpá-lo mais tarde. Claro, naquela época não conhecíamos a lei de LeBlanc: *depois é igual a nunca*.

O custo total de possuir uma bagunça

Se você é programador há mais de dois ou três anos, provavelmente foi significativamente retardado pelo código confuso de outra pessoa. Se você é programador há mais de dois ou três anos, provavelmente foi prejudicado por códigos confusos.

O grau de desaceleração pode ser significativo. Ao longo de um ou dois anos, as equipes que estavam se movendo muito rapidamente no início de um projeto podem se mover a passo de tartaruga. Cada mudança que eles fazem no código quebra duas ou três outras partes do código. Nenhuma mudança é trivial. Cada adição ou modificação no sistema requer que os emaranhados, torções e nós sejam “compreendidos” para que mais emaranhados, torções e nós possam ser adicionados.

Com o tempo, a bagunça se torna tão grande, tão profunda e tão alta que eles não conseguem limpá-la. Não há como.

À medida que a bagunça aumenta, a produtividade da equipe continua diminuindo, aproximando-se assintoticamente de zero. À medida que a produtividade diminui, a gerência faz a única coisa que pode; eles adicionam mais funcionários ao projeto na esperança de aumentar a produtividade. Mas essa nova equipe não é versada no design do sistema. Eles não sabem a diferença entre uma mudança que corresponde à intenção do projeto e uma mudança que frustra a intenção do projeto. Além disso, eles e todos os outros membros da equipe estão sob pressão terrível para aumentar a produtividade. Então, todos eles fazem mais e mais bagunça, levando a produtividade cada vez mais para zero. (Veja a Figura 1-1.)

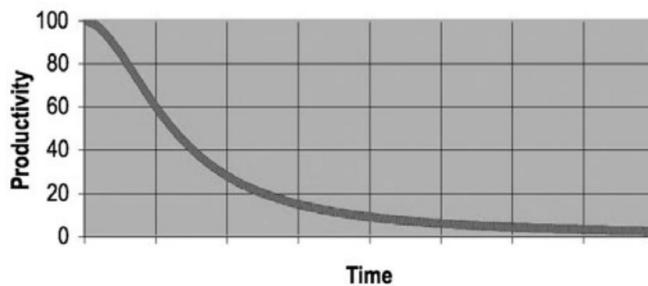


Figura 1-1
Produtividade vs. tempo

O custo total de possuir uma bagunça

O Grande Redesenho no Céu

Eventualmente, a equipe se rebela. Eles informam à administração que não podem continuar a se desenvolver nessa base de código odiosa. Eles exigem um redesenho. A gerência não quer gastar os recursos em uma nova reformulação do projeto, mas não pode negar que a produtividade é terrível. Eventualmente, eles se curvam às demandas dos desenvolvedores e autorizam o grande redesenho no céu.

Uma nova equipe de tigres é selecionada. Todo mundo quer estar nesta equipe porque é um projeto de campo verde. Eles começam de novo e criam algo verdadeiramente bonito. Mas apenas os melhores e mais brilhantes são escolhidos para o time tigre. Todos os outros devem continuar a manter o sistema atual.

Agora as duas equipes estão em uma corrida. A equipe tigre deve construir um novo sistema que faça tudo o que o sistema antigo faz. Além disso, eles precisam acompanhar as mudanças que estão sendo feitas continuamente no sistema antigo. A administração não substituirá o sistema antigo até que o novo sistema possa fazer tudo o que o sistema antigo faz.

Esta corrida pode durar muito tempo. Eu vi isso levar 10 anos. E quando termina, os membros originais da equipe tigre já se foram há muito tempo, e os membros atuais estão exigindo que o novo sistema seja redesenho porque é uma bagunça.

Se você experimentou pelo menos uma pequena parte da história que acabei de contar, já sabe que gastar tempo mantendo seu código limpo não é apenas econômico; é uma questão de sobrevivência profissional.

Atitude

Você já se deparou com uma confusão tão grave que levou semanas para fazer o que deveria ter levado horas? Você viu o que deveria ter sido uma mudança de uma linha, feita em centenas de módulos diferentes? Esses sintomas são muito comuns.

Por que isso acontece com o código? Por que um bom código apodrece tão rapidamente em um código ruim? Temos muitas explicações para isso. Reclamamos que os requisitos mudaram de forma a frustrar o projeto original. Lamentamos os horários que eram muito apertados para fazer as coisas direito.

Nós tagarelamos sobre gerentes estúpidos e clientes intolerantes e tipos de marketing inúteis e desinfetantes de telefone. Mas a culpa, querido Dilbert, não está em nossas estrelas, mas em nós mesmos.

Nós não somos profissionais.

Esta pode ser uma pílula amarga de engolir. Como essa bagunça pode ser *nossa* culpa? E os requisitos? E o cronograma? E os gerentes estúpidos e os tipos de marketing inúteis? Eles não carregam parte da culpa?

Não. Os gerentes e profissionais de marketing *nós* procuram para obter as informações necessárias para fazer promessas e compromissos; e mesmo quando eles não olham para nós, não devemos ter vergonha de dizer a eles o que pensamos. Os usuários nos procuram para validar a forma como os requisitos se encaixam no sistema. Os gerentes de projeto nos procuram para ajudar a elaborar o cronograma. Nós

são profundamente cúmplices do planejamento do projeto e compartilham grande parte da responsabilidade por eventuais falhas; especialmente se essas falhas tiverem a ver com código ruim!

"Mas espere!" você diz. "Se eu não fizer o que meu gerente diz, serei demitido." Provavelmente não. A maioria dos gerentes quer a verdade, mesmo quando não age como tal. A maioria dos gerentes quer um bom código, mesmo quando estão obcecados com o cronograma. Eles podem defender o cronograma e os requisitos com paixão; mas esse é o trabalho deles. É seu trabalho defender o código com a mesma paixão.

Para esclarecer esse ponto, e se você fosse um médico e tivesse um paciente que exigisse que você parasse com toda a bobagem de lavar as mãos na preparação para a cirurgia porque estava tomando muito tempo?² Claramente, o paciente é o chefe ; e, no entanto, o médico deve se recusar absolutamente a obedecer. Por que? Porque o médico sabe mais do que o paciente sobre os riscos de doenças e infecções. Seria antiprofissional (muito menos criminoso) o médico concordar com o paciente.

Da mesma forma, não é profissional que os programadores se submetam à vontade de gerentes que não o fazem. entender os riscos de fazer bagunça.

O enigma primordial

Os programadores enfrentam um enigma de valores básicos. Todos os desenvolvedores com mais de alguns anos de experiência sabem que as bagunças anteriores os atrasam. E, no entanto, todos os desenvolvedores sentem a pressão de fazer bagunça para cumprir os prazos. Resumindo, eles não demoram para ir rápido!

Os verdadeiros profissionais sabem que a segunda parte do enigma está errada. Você *não* vai cumprir o prazo fazendo bagunça. Na verdade, a confusão vai atrasá-lo instantaneamente e forçá-lo a perder o prazo. A *única* maneira de cumprir o prazo - a única maneira de ir rápido - é manter o código o mais limpo possível o tempo todo.

A arte do código limpo?

Digamos que você acredite que um código confuso é um impedimento significativo. Digamos que você aceite que a única maneira de ir rápido é manter seu código limpo. Então você deve se perguntar: "Como faço para escrever um código limpo?" Não adianta tentar escrever código limpo se você não sabe o que significa código limpo!

A má notícia é que escrever um código limpo é muito parecido com pintar um quadro. A maioria de nós sabe quando um quadro é bem ou mal pintado. Mas ser capaz de distinguir a arte boa da ruim não significa que saibamos pintar. Da mesma forma, ser capaz de reconhecer código limpo de código sujo não significa que sabemos como escrever código limpo!

2. Quando a lavagem das mãos foi recomendada pela primeira vez aos médicos por Ignaz Semmelweis em 1847, ela foi rejeitada com base no fato de que os médicos estavam muito ocupados e não teriam tempo para lavar as mãos entre as visitas aos pacientes.

O custo total de possuir uma bagunça

Escrever um código limpo requer o uso disciplinado de uma miríade de pequenas técnicas aplicadas por meio de um senso de "limpeza" adquirido meticulosamente. Esse "senso de código" é a chave.

Alguns de nós nascemos com isso. Alguns de nós têm que lutar para adquiri-lo. Não apenas nos permite ver se o código é bom ou ruim, mas também nos mostra a estratégia para aplicar nossa disciplina para transformar código ruim em código limpo.

Um programador sem "senso de código" pode olhar para um módulo bagunçado e reconhecer a bagunça, mas não terá ideia do que fazer a respeito. Um programador com "senso de código" olhará para um módulo confuso e verá opções e variações. O "code-sense" vai ajudar esse programador a escolher a melhor variação e guiá-lo a traçar uma sequência de transformações de preservação de comportamento para ir daqui para lá.

Resumindo, um programador que escreve código limpo é um artista que pode pegar uma tela em branco através de uma série de transformações até que seja um sistema elegantemente codificado.

O que é código limpo?

Provavelmente existem tantas definições quanto programadores. Então, perguntei a alguns programadores muito conhecidos e profundamente experientes o que eles achavam.

**Bjarne Stroustrup, inventor do C++ e autor de
*The C++ Programming
Linguagem***

*Gosto que meu código seja elegante e eficiente.
A lógica deve ser direta para dificultar a ocultação
de bugs, as dependências mínimas para facilitar
a manutenção, o tratamento de erros completo
de acordo com uma estratégia articulada e o
desempenho próximo do ideal para não tentar as
pessoas a bagunçar o código com códigos sem
princípios, otimizações. O código limpo faz uma
coisa bem.*



Bjarne usa a palavra "elegante". Essa é uma palavra e tanto! O dicionário do meu MacBook® fornece as seguintes definições: *agradavelmente gracioso e elegante na aparência ou nas maneiras; agradavelmente engenhoso e simples*. Observe a ênfase na palavra "agradável". Aparentemente, Bjarne acha que código limpo é *agradável* de ler. Lê-lo deve fazer você sorrir da mesma forma que uma caixa de música bem trabalhada ou um carro bem projetado faria.

Bjarne também menciona a eficiência — duas vezes. Talvez isso não deva nos surpreender vindo do inventor do C++; mas acho que há mais do que o simples desejo de velocidade. Ciclos desperdiçados são deselegantes, não agradam. E agora observe a palavra que Bjarne usa

para descrever a consequência dessa desleigânciia. Ele usa a palavra “tentar”. Há uma verdade profunda aqui. Código ruim tenta a bagunça para crescer! Quando outros alteram um código ruim, eles tendem a piorá-lo.

O pragmático Dave Thomas e Andy Hunt disseram isso de uma maneira diferente. Eles usaram a metáfora das janelas quebradas.³ Um prédio com janelas quebradas parece que ninguém se importa com ele. Então outras pessoas param de se importar. Eles permitem que mais janelas sejam quebradas. Eventualmente, eles os quebram ativamente. Despojam a fachada com pichações e permitem a coleta de lixo. Uma janela quebrada inicia o processo de decadênciia.

Bjarne também menciona que a manipulação de erros deve ser completa. Isso vai para a disciplina de prestar atenção aos detalhes. O tratamento abreviado de erros é apenas uma maneira de os programadores passarem por cima dos detalhes. Vazamentos de memória são outra, condições de corrida ainda outra. Nomenclatura inconsistente ainda outro. O resultado é que o código limpo exibe muita atenção aos detalhes.

Bjarne fecha com a afirmação de que o código limpo faz uma coisa bem. Não é por acaso que existem tantos princípios de design de software que podem ser resumidos a esta simples adverténcia. Escritor após escritor tentou comunicar esse pensamento. Código ruim tenta fazer demais, tem intenção confusa e ambigüidade de propósito. Código limpo é *focado*. Cada função, cada classe, cada módulo expõe uma atitude obstinada que permanece totalmente sem distrações e sem poluição pelos detalhes circundantes.

Grady Booch, autor de *Objeto Análise e Projeto Orientados com Formulários*

O código limpo é *simples e direto*. O código limpo é lido como uma prosa bem escrita. O código limpo nunca obscurece a intenção do designer, mas está cheio de abstrações nítidas e linhas diretas de controle.

Grady faz alguns dos mesmos pontos que Bjarne, mas ele adota uma perspectiva de *legibilidade*. Gosto especialmente de sua visão de que o código limpo deve ser lido como uma prosa bem escrita. Pense em um livro realmente bom que você leu. Lembre-se de como as palavras desapareceram para serem substituídas por imagens! Foi como assistir a um filme, não foi? Melhorar! Você viu os personagens, ouviu os sons, experimentou o pathos e o humor.

Ler código limpo nunca será como ler *O Senhor dos Anéis*. Ainda assim, a metáfora literária não é ruim. Como um bom romance, o código limpo deve expor claramente as tensões do problema a ser resolvido. Deve construir essas tensões até um clímax e, em seguida, dar



3. <http://www.pragmaticprogrammer.com/booksellers/2004-12.html>

O custo total de possuir uma bagunça

o leitor que “Aha! Claro!” como as questões e tensões são resolvidas na revelação de uma solução óbvia.

Acho que o uso de Grady da frase “abstração nítida” é um oxímoro fascinante! Afinal, a palavra “nítido” é quase um sinônimo de “concreto”. O dicionário do meu MacBook contém a seguinte definição de “crisp”: *rapidamente decisivo e prático, sem hesitação ou detalhes desnecessários*. Apesar dessa aparente justaposição de significado, as palavras carregam uma mensagem poderosa. Nosso código deve ser prático em vez de especulativo. Deve conter apenas o necessário. Nossos leitores devem perceber que fomos decisivos.

“Big” Dave Thomas, fundador da OTI, padrinho da estratégia Eclipse

O código limpo pode ser lido e aprimorado por um desenvolvedor que não seja seu autor original. Possui testes unitários e de aceitação. Tem nomes significativos. Ele fornece uma maneira em vez de muitas maneiras de fazer uma coisa. Ele tem dependências mínimas, que são definidas explicitamente, e fornece uma API clara e mínima. O código deve ser alfabetizados, pois, dependendo do idioma, nem todas as informações necessárias podem ser expressas claramente apenas em código.



Big Dave compartilha o desejo de Grady por legibilidade, mas com uma reviravolta importante.

Dave afirma que o código limpo torna mais fácil para *outras* pessoas aprimorá-lo. Isso pode parecer óbvio, mas não pode ser enfatizado demais. Afinal, existe uma diferença entre código fácil de ler e código fácil de alterar.

Dave associa a limpeza aos testes! Dez anos atrás, isso teria levantado muitas sobrancelhas. Mas a disciplina de Test Driven Development teve um impacto profundo em nosso setor e se tornou uma de nossas disciplinas mais fundamentais. Davi está certo. Código, sem testes, não é limpo. Por mais elegante que seja, por mais legível e acessível, se não tiver testes, será impuro.

Dave usa a palavra *mínimo* duas vezes. Aparentemente, ele valoriza o código pequeno, em vez do código grande. Na verdade, esse tem sido um refrão comum em toda a literatura de software desde o seu início. Menor é melhor.

Dave também diz que o código deve ser *alfabetizado*. Esta é uma referência suave à programação alfabetizada de Knuth.⁴ O resultado é que o código deve ser composto de forma a torná-lo legível por humanos.

4. [Knuth92].

Michael Feathers, autor de *Trabalhando Efetivamente com código legado*

Eu poderia listar todas as qualidades que noto no código limpo, mas há uma qualidade abrangente que leva a todas elas. Código limpo sempre parece ter sido escrito por alguém que se importa.

Não há nada óbvio que você possa fazer para torná-lo melhor. Todas essas coisas foram pensadas pelo autor do código e, se você tentar imaginar melhorias, será levado de volta para onde está, apreciando o código que alguém deixou para você - código deixado por alguém que se preocupa profundamente com o ofício.

Uma palavra: cuidado. Esse é realmente o tema deste livro. Talvez um subtítulo apropriado seja *How to Care for Code*.

Michael acertou na cabeça. Código limpo é código que foi cuidado. Alguém se preocupou em mantê-lo simples e organizado. Eles prestaram a devida atenção aos detalhes. Eles se importaram.



Ron Jeffries, autor de *Extreme Programming: Programação Instalada e Extrema Aventuras em C#*

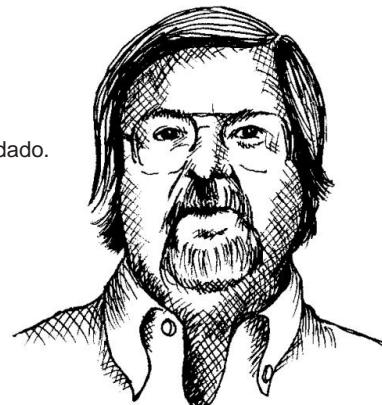
Ron começou sua carreira programando em Fortran no Comando Aéreo Estratégico e escreveu código em quase todas as linguagens e em quase todas as máquinas. Vale a pena considerar suas palavras com cuidado.

Nos últimos anos, comecei e quase terminei com as regras de código simples de Beck. Em ordem de prioridade, código simples:

- Executa todos os testes;
- Não contém duplicação;
- Expressa todas as ideias de design que estão no sistema;
- Minimiza o número de entidades como classes, métodos, funções e similares.

Destes, concentro-me principalmente na duplicação. Quando a mesma coisa é feita repetidamente, é sinal de que existe uma ideia em nossa mente que não está bem representada no código. Eu tento descobrir o que é. Então tento expressar essa ideia com mais clareza.

A expressividade para mim inclui nomes significativos e é provável que eu mude os nomes das coisas várias vezes antes de me estabelecer. Com ferramentas de codificação modernas, como o Eclipse, renomear é bastante barato, então não me incomoda mudar. A expressividade vai



O custo total de possuir uma bagunça

11

além de nomes, no entanto. Também observo se um objeto ou método está fazendo mais de uma coisa. Se for um objeto, provavelmente precisa ser dividido em dois ou mais objetos. Se for um método, sempre usarei a refatoração Extract Method nele, resultando em um método que diz mais claramente o que faz e alguns submétodos dizendo como isso é feito.

Duplicação e expressividade me levam muito longe no que considero código limpo, e melhorar o código sujo com apenas essas duas coisas em mente pode fazer uma grande diferença. Há, no entanto, uma outra coisa que estou ciente de fazer, que é um pouco mais difícil de explicar.

Depois de anos fazendo este trabalho, parece-me que todos os programas são compostos de elementos muito semelhantes. Um exemplo é “encontrar coisas em uma coleção”. Quer tenhamos um banco de dados de registros de funcionários, um mapa de hash de chaves e valores ou uma matriz de itens de algum tipo, muitas vezes nos encontramos querendo um determinado item dessa coleção. Quando descubro que isso está acontecendo, geralmente envolvo a implementação específica em um método ou classe mais abstrato. Isso me dá algumas vantagens interessantes.

Possuo implementar a funcionalidade agora com algo simples, digamos um mapa de hash, mas como agora todas as referências a essa pesquisa são cobertas por minha pequena abstração, posso alterar a implementação sempre que quiser. Posso avançar rapidamente, preservando minha capacidade de mudar mais tarde.

Além disso, a abstração da coleção geralmente chama minha atenção para o que “realmente” está acontecendo e me impede de seguir o caminho da implementação de um comportamento arbitrário de coleção quando tudo o que eu realmente preciso são algumas maneiras bastante simples de encontrar o que quero.

*Duplicação reduzida, alta expressividade e construção precoce de abstrações simples.
Isso é o que torna o código limpo para mim.*

Aqui, em alguns parágrafos curtos, Ron resumiu o conteúdo deste livro. Não duplicação, uma coisa, expressividade, pequenas abstrações. Tudo está lá.

**Ward Cunningham, inventor do Wiki,
inventor do Fit, co-inventor da eXtreme
Programming. Força motriz por trás dos
Padrões de Projeto. Líder de pensamento
Smalltalk e OO. O padrinho de todos
aqueles que se preocupam com o código.**

Você sabe que está trabalhando em um código limpo quando cada rotina que você lê acaba sendo exatamente o que você esperava. Você pode chamá-lo de código bonito quando o código também faz parecer que a linguagem foi feita para o problema.

Declarções como essa são características de Ward. Você lê, acena com a cabeça e passa para o próximo tópico. Soa tão razoável, tão óbvio, que mal se registra como algo profundo. Você pode pensar que foi exatamente o que você esperava. Mas vamos dar uma olhada mais de perto.



“... praticamente o que você esperava.” Quando foi a última vez que você viu um módulo que era exatamente o que você esperava? Não é mais provável que os módulos que você vê sejam intrigantes, complicados, emaranhados? Desorientação não é a regra? Você não está acostumado a se debater tentando agarrar e segurar os fios do raciocínio que brotam de todo o sistema e tecem seu caminho através do módulo que você está lendo? Quando foi a última vez que você leu algum código e acenou com a cabeça do jeito que você poderia ter acenado com a declaração de Ward?

Ward espera que, ao ler um código limpo, você não se surpreenda. Na verdade, você nem vai gastar muito esforço. Você o lerá e será exatamente o que você esperava. Será óbvio, simples e atraente. Cada módulo definirá o cenário para o próximo. Cada um diz a você como o próximo será escrito. Programas tão *limpos* são tão profundamente bem escritos que você nem percebe. O designer faz com que pareça ridiculamente simples como todos os designs excepcionais.

E a noção de beleza de Ward? Todos nós protestamos contra o fato de que nossas línguas não foram projetadas para nossos problemas. Mas a declaração de Ward coloca o ônus de volta em nós. Ele diz que um código bonito *faz a linguagem parecer que foi feita para o problema!* Portanto, é nossa responsabilidade fazer com que a linguagem pareça simples! Intolerantes da linguagem em todos os lugares, cuidado! Não é a linguagem que faz os programas parecerem simples. É o programador que faz a linguagem parecer simples!

Escolas de pensamento

E quanto a mim (tio Bob)? O que eu acho que é código limpo? Este livro contará a você, em detalhes hediondos, o que eu e meus compatriotas pensamos sobre código limpo. Diremos a você o que achamos que torna um nome de variável limpo, uma função limpa, uma classe limpa, etc. Apresentaremos essas opiniões como absolutas e não pediremos desculpas por nossa estridência. Para nós, neste ponto de nossas carreiras, eles são absolutos. Eles são *nossa escola de pensamento* sobre código limpo.

Os artistas marciais nem todos concordam sobre a melhor arte marcial ou a melhor técnica dentro de uma arte marcial. Freqüentemente, os mestres marciais formarão suas próprias escolas de pensamento e reunirão alunos para aprender com eles. Assim vemos o *Gracie Jiu Jitsu*, fundado e ensinado pela família Gracie no Brasil. Vemos o *Hakkoryu Jiu Jitsu*, fundado e ensinado por Okuyama Ryuho em Tóquio. Vemos o *Jeet Kune Do*, fundado e ensinado por Bruce Lee nos Estados Unidos.



Os alunos dessas abordagens mergulham nos ensinamentos do fundador. Eles se dedicam a aprender o que aquele mestre em particular ensina, muitas vezes excluindo o ensino de qualquer outro mestre. Mais tarde, à medida que os alunos crescem em sua arte, eles podem se tornar alunos de um mestre diferente para que possam ampliar seus conhecimentos e práticas. Alguns acabam refinando suas habilidades, descobrindo novas técnicas e fundando suas próprias escolas.

Nenhuma dessas diferentes escolas está absolutamente *certa*. No entanto, dentro de uma escola particular, agimos *como* se os ensinamentos e técnicas *fossem* corretos. Afinal, existe um jeito certo de praticar o Hakkoryu Jiu Jitsu, ou Jeet Kune Do. Mas essa correção dentro de uma escola não invalida os ensinamentos de uma escola diferente.

Considere este livro uma descrição da *Object Mentor School of Clean Code*. As técnicas e ensinamentos contidos são a forma como *praticamos nossa* arte. Estamos dispostos a afirmar que, se você seguir esses ensinamentos, desfrutará dos benefícios que desfrutamos e aprenderá a escrever códigos limpos e profissionais. Mas não cometa o erro de pensar que de alguma forma estamos “certos” em qualquer sentido absoluto. Existem outras escolas e outros mestres que reivindicam tanto profissionalismo quanto nós. Caberia a você aprender com eles também.

De fato, muitas das recomendações deste livro são controversas. Você provavelmente não concordará com todos eles. Você pode discordar violentemente de alguns deles. Isso é bom. Não podemos reivindicar a autoridade final. Por outro lado, as recomendações deste livro são coisas sobre as quais pensamos muito e muito. Nós os aprendemos através de décadas de experiência e repetidas tentativas e erros. Então, quer você concorde ou discorde, seria uma pena se você não visse e respeitasse nosso ponto de vista.

Nós somos autores

O campo @autor de um Javadoc nos diz quem somos. Somos autores. E uma coisa sobre os autores é que eles têm leitores. De fato, os autores são *responsáveis* por se comunicar bem com seus leitores. Da próxima vez que você escrever uma linha de código, lembre-se de que você é um autor, escrevendo para leitores que julgarão seu esforço.

Você pode perguntar: Quanto de código é realmente lido? A maior parte do esforço não vai para escrevê-lo?

Você já reproduziu uma sessão de edição? Nos anos 80 e 90, tínhamos editores como o Emacs que registravam cada tecla digitada. Você pode trabalhar por uma hora e depois reproduzir toda a sua sessão de edição como um filme de alta velocidade. Quando fiz isso, os resultados foram fascinantes.

A grande maioria da reprodução foi rolando e navegando para outros módulos!

Bob entra no módulo.

Ele rola para baixo até a função que precisa ser alterada.

Ele faz uma pausa, considerando suas opções.

Oh, ele está rolando até o topo do módulo para verificar a inicialização de uma variável.

Agora ele rola para baixo e começa a digitar.

Ops, ele está apagando o que digitou!

Ele digita novamente.

Ele apaga de novo!

Ele digita metade de outra coisa, mas apaga isso!

Ele rola para baixo até outra função que chama a função que está alterando para ver como ela é chamada.

Ele volta para cima e digita o mesmo código que acabou de apagar.

Ele faz uma pausa.

Ele apaga esse código novamente!

Ele abre outra janela e olha para uma subclasse. Essa função é substituída?

...

Você começa a deriva. De fato, a proporção de tempo gasto lendo versus escrevendo é bem superior a 10:1.

Estamos *constantemente* lendo códigos antigos como parte do esforço de escrever novos códigos.

Como essa proporção é tão alta, queremos que a leitura do código seja fácil, mesmo que dificulte a escrita. É claro que não há como escrever código sem lê-lo, portanto, *torná-lo fácil de ler na verdade facilita a escrita*.

Não há como fugir dessa lógica. Você não pode escrever código se não puder ler o código circundante. O código que você está tentando escrever hoje será difícil ou fácil de escrever, dependendo de quão difícil ou fácil é ler o código circundante. Portanto, se você quiser ir rápido, se quiser terminar rapidamente, se quiser que seu código seja fácil de escrever, torne-o fácil de ler.

A regra do escoteiro

Não basta escrever bem o código. O código deve ser *mantido limpo* ao longo do tempo. Todos nós já vimos o código apodrecer e degradar com o passar do tempo. Portanto, devemos assumir um papel ativo na prevenção dessa degradação.

Os Boy Scouts of America têm uma regra simples que podemos aplicar à nossa profissão.

Deixe o acampamento mais limpo do que o encontrou.⁵

Se todos nós fizéssemos check-in de nosso código um pouco mais limpo do que quando fizemos check-out, o código simplesmente não poderia apodrecer. A limpeza não precisa ser algo grande. Mude um nome de variável para melhor, quebre uma função que é um pouco grande demais, elimine um pequeno pedaço de duplicação, limpe uma declaração if composta .

Você consegue se imaginar trabalhando em um projeto em que o código *simplesmente melhorou* com o passar do tempo? Você acredita que alguma outra opção é profissional? De fato, a melhoria contínua não é uma parte intrínseca do profissionalismo?

5. Isso foi adaptado da mensagem de despedida de Robert Stephenson Smyth Baden-Powell aos escoteiros: "Tente deixar este mundo um pouco melhor do que o encontrou..."

Prequela e Princípios

De muitas maneiras, este livro é uma “prequela” de um livro que escrevi em 2002, intitulado *Agile Software Development: Principles, Patterns, and Practices* (PPP). O livro PPP se preocupa com os princípios do projeto orientado a objetos e muitas das práticas usadas por desenvolvedores profissionais. Se você ainda não leu o PPP, pode descobrir que ele continua a história contada por este livro. Se você já o leu, encontrará muitos dos sentimentos desse livro ecoados neste nível do código.

Neste livro você encontrará referências esporádicas a vários princípios de design. Estes incluem o Princípio de Responsabilidade Única (SRP), o Princípio Aberto e Fechado (OCP) e o Princípio de Inversão de Dependência (DIP), entre outros. Esses princípios são descritos em profundidade no PPP.

Conclusão

Livros sobre arte não prometem fazer de você um artista. Tudo o que eles podem fazer é fornecer algumas das ferramentas, técnicas e processos de pensamento que outros artistas usaram. Da mesma forma, este livro não promete fazer de você um bom programador. Não pode prometer dar a você “senso de código”. Tudo o que ele pode fazer é mostrar os processos de pensamento de bons programadores e os truques, técnicas e ferramentas que eles usam.

Assim como um livro sobre arte, este livro será cheio de detalhes. Haverá muitos códigos. Você verá um código bom e um código ruim. Você verá um código ruim transformado em um código bom. Você verá listas de heurísticas, disciplinas e técnicas. Você verá exemplo após exemplo. Depois disso, cabe a você.

Lembra-se da velha piada sobre o violinista concertista que se perdeu a caminho de uma apresentação? Ele parou um velho na esquina e perguntou como chegar ao Carnegie Hall. O velho olhou para o violinista e o violino debaixo do braço e disse: “Pratique, filho. Prática!”

Bibliografia

[Beck07]: *Padrões de Implementação*, Kent Beck, Addison-Wesley, 2007.

[Knuth92]: *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

Esta página foi intencionalmente deixada em branco

2

Nomes Significativos

por Tim Ottinger



Introdução

Os nomes estão por toda parte no software. Nomeamos nossas variáveis, nossas funções, nossos argumentos, classes e pacotes. Nomeamos nossos arquivos de origem e os diretórios que os contêm. Nomeamos nossos arquivos jar, arquivos war e arquivos ear. Nós nomeamos e nomeamos e nomeamos. Porque nós fazemos

tanto, é melhor fazê-lo bem. O que segue são algumas regras simples para criar bons nomes.

Use nomes reveladores de intenções

É fácil dizer que os nomes devem revelar a intenção. O que queremos mostrar a você é que levamos isso a sério . Escolher bons nomes leva tempo, mas economiza mais do que leva.

Portanto, tome cuidado com seus nomes e troque-os quando encontrar nomes melhores. Todos que lerem seu código (incluindo você) ficarão mais felizes se você o fizer.

O nome de uma variável, função ou classe deve responder a todas as grandes questões. Deve dizer por que existe, o que faz e como é usado. Se um nome requer um comentário, então o nome não revela sua intenção.

```
int d; // tempo decorrido em dias
```

O nome d não revela nada. Não evoca uma sensação de tempo decorrido, nem de dias. Devemos escolher um nome que especifique o que está sendo medido e a unidade dessa medida:

```
int elapsedTimeInDays; int
daysSinceCreation; int
daysSinceModification; int
arquivoAgelInDays;
```

Escolher nomes que revelem a intenção pode facilitar muito a compreensão e a mudança código. Qual é a finalidade deste código?

```
public List<int[]> getThem() { List<int[]>
    list1 = new ArrayList<int[]>(); for (int[] x : theList) if (x[0] ==
    4) list1.add(x); lista de retorno1;

}
```

Por que é difícil dizer o que esse código está fazendo? Não há expressões complexas. Espaçamento e recuo são razoáveis. Existem apenas três variáveis e duas constantes mencionadas. Não existem classes sofisticadas ou métodos polimórficos, apenas uma lista de arrays (ou assim parece).

O problema não é a simplicidade do código, mas a *implícita* do código (para cunhar uma frase): o grau em que o contexto não é explícito no próprio código. O código exige implicitamente que saibamos as respostas para perguntas como:

1. Que tipos de coisas estão na lista?
2. Qual é o significado do subscrito zero de um item na Lista?
3. Qual é o significado do valor 4?
4. Como eu usaria a lista que está sendo retornada?

As respostas a essas perguntas não estão presentes no exemplo de código, mas poderiam estar. Digamos que estamos trabalhando em um jogo de vassoura de minas. Descobrimos que o quadro é uma lista de células chamada theList. Vamos renomeá-lo para gameBoard.

Cada célula no tabuleiro é representada por uma matriz simples. Além disso, descobrimos que o subscrito zero é a localização de um valor de status e que um valor de status de 4 significa “sinalizado”. Apenas dando nomes a esses conceitos podemos melhorar consideravelmente o código:

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>(); for (int[] célula : gameBoard) if (célula[STATUS_VALUE] == FLAGGED) flaggedCells.add(cell); return células marcadas;
}
```

Observe que a simplicidade do código não mudou. Ele ainda tem exatamente o mesmo número de operadores e constantes, com exatamente o mesmo número de níveis de aninhamento. Mas o código se tornou muito mais explícito.

Podemos ir além e escrever uma classe simples para células em vez de usar um array de ints. Ele pode incluir uma função reveladora de intenções (chame-a de isFlagged) para ocultar os números mágicos. Isso resulta em uma nova versão da função:

```
public List<Cell> getFlaggedCells() { List<Cell> flaggedCells = new ArrayList<Cell>(); for (Célula: gameBoard) if (cell.isFlagged())
    flaggedCells.add(célula); return células marcadas;
}
```

Com essas simples mudanças de nome, não é difícil entender o que está acontecendo. Este é o poder de escolher bons nomes.

Evite Desinformação

Os programadores devem evitar deixar pistas falsas que obscureçam o significado do código. Devemos evitar palavras cujos significados arraigados variam de nosso significado pretendido. Por exemplo, hp, aix e sco seriam nomes de variáveis ruins porque são nomes de plataformas ou variantes do Unix. Mesmo que você esteja codificando uma hipotenusa e hp pareça uma boa abreviação, ela pode ser desinformativa.

Não se refira a um agrupamento de contas como uma accountList , a menos que seja realmente uma lista. A palavra lista significa algo específico para programadores. Se o contêiner que contém as contas não for realmente uma lista, isso pode levar a conclusões falsas.¹ Portanto, accountGroup ou bundleOfAccounts ou apenas contas simples seriam melhores.

¹. Como veremos mais adiante, mesmo que o contêiner seja uma lista, provavelmente é melhor não codificar o tipo de contêiner no nome.

Cuidado com o uso de nomes que variam em pequenas formas. Quanto tempo leva para detectar a diferença sutil entre um XYZControllerForEfficientHandlingOfStrings em um módulo e, em algum lugar um pouco mais distante, XYZControllerForEfficientStorageOfStrings? As palavras têm formas assustadoramente semelhantes.

Escrever conceitos semelhantes de maneira semelhante é *informação*. Usar grafias inconsistentes é *desinformação*. Com ambientes Java modernos, desfrutamos da conclusão automática de código. Escrevemos alguns caracteres de um nome e pressionamos alguma combinação de teclas de atalho (se houver) e somos recompensados com uma lista de possíveis conclusões para esse nome. É muito útil se os nomes para coisas muito semelhantes forem classificados em ordem alfabética e se as diferenças forem muito óbvias, porque o desenvolvedor provavelmente escolherá um objeto pelo nome sem ver seus comentários abundantes ou mesmo a lista de métodos fornecidos por essa classe.

Um exemplo realmente terrível de nomes desinformativos seria o uso de L minúsculo ou O maiúsculo como nomes de variáveis, especialmente em combinação. O problema, é claro, é que eles se parecem quase inteiramente com as constantes um e zero, respectivamente.

```
int a = l; se
( O == l ) a =
O1;
senão l = 01;
```

O leitor pode pensar que isso é um artifício, mas examinamos o código onde tais coisas eram abundantes. Em um caso, o autor do código sugeriu o uso de uma fonte diferente para que as diferenças fossem mais óbvias, uma solução que teria de ser transmitida a todos os futuros desenvolvedores como tradição oral ou em um documento escrito. O problema é resolvido definitivamente e sem criar novos produtos de trabalho por uma simples renomeação.

Tornar Significativo Distinções

Os programadores criam problemas para si mesmos quando escrevem código apenas para satisfazer um compilador ou interpretador. Por exemplo, porque você não pode usar o mesmo nome para se referir a duas coisas diferentes no mesmo escopo, você pode ser tentado a mudar um nome de forma arbitrária. Às vezes, isso é feito com erros de ortografia, levando a uma situação surpreendente em que corrigir erros de ortografia leva à incapacidade de compilar.²



Não é suficiente adicionar séries de números ou palavras de ruído, mesmo que o compilador seja satisfeito. Se os nomes devem ser diferentes, eles também devem significar algo diferente.

2. Considere, por exemplo, a prática verdadeiramente hedionda de criar uma variável chamada klass apenas porque o nome class foi usado para outra coisa.

A nomenclatura de séries numéricas (a1, a2, .. aN) é o oposto da nomenclatura intencional. Esses nomes não são desinformativos - eles não são informativos; eles não fornecem nenhuma pista sobre a intenção do autor. Considerar:

```
public static void copyChars(char a1[], char a2[]) { for (int i = 0; i <
a1.length; i++) { a2[i] = a1[i]; }

}
```

Esta função lê muito melhor quando a origem e o destino são usados para o argumento nomes.

As palavras barulhentas são outra distinção sem sentido. Imagine que você tenha uma classe Produto . Se você tiver outro chamado ProductInfo ou ProductData, você tornou os nomes diferentes sem fazê-los significar algo diferente. Informações e dados são palavras de ruído indistintas como a, an e the.

Observe que não há nada de errado em usar convenções de prefixo como a e the , desde que façam uma distinção significativa. Por exemplo, você pode usar a para todas as variáveis locais e the para todos os argumentos de função.³ O problema surge quando você decide chamar uma variável de theZork porque já tem outra variável chamada zork.

Palavras de ruído são redundantes. A palavra variável nunca deve aparecer em um nome de variável. A palavra tabela nunca deve aparecer no nome de uma tabela. Como NameString é melhor que Name? Um nome seria um número de ponto flutuante? Nesse caso, isso quebra uma regra anterior sobre desinformação. Imagine encontrar uma classe chamada Customer e outra chamada CustomerObject. O que você deve entender como a distinção? Qual representará o melhor caminho para o histórico de pagamentos de um cliente?

Existe um aplicativo que conhecemos onde isso é ilustrado. nós mudamos os nomes para proteger os culpados, mas aqui está a forma exata do erro:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Como os programadores neste projeto devem saber qual dessas funções chamar?

Na ausência de convenções específicas, a variável moneyAmount é indistinguível de money, customerInfo é indistinguível de customer, accountData é indistinguível de account e theMessage é indistinguível de message. Distinga os nomes de forma que o leitor saiba o que as diferenças oferecem.

Use nomes pronunciáveis

Os seres humanos são bons em palavras. Uma parte significativa de nossos cérebros é dedicada ao conceito de palavras. E as palavras são, por definição, pronunciáveis. Seria uma pena não aceitar

3. Uncle Bob costumava fazer isso em C++, mas desistiu da prática porque os IDEs modernos a tornam desnecessária.

vantagem dessa enorme porção de nossos cérebros que evoluiu para lidar com a linguagem falada. Portanto, torne seus nomes pronunciáveis.

Se você não consegue pronunciá-lo, não pode discuti-lo sem parecer um idiota. "Bem, aqui no bee cee arr three cee enn tee temos um pee ess zee kyew int, viu?" Isso é importante porque a programação é uma atividade social.

Uma empresa que conheço tem genymdhms (data de geração, ano, mês, dia, hora, minuto e segundo), então eles andavam por aí dizendo "gen why emm dee aich emm ess". Tenho o hábito irritante de pronunciar tudo como está escrito, então comecei a dizer "gen-yah-mudda hims". Mais tarde, foi chamado assim por uma série de designers e analistas, e ainda parecíamos bobos. Mas nós estávamos na brincadeira, então foi divertido. Divertido ou não, estávamos tolerando nomes ruins. Os novos desenvolvedores precisavam ter as variáveis explicadas a eles, e então eles falavam sobre isso com palavras bobas inventadas, em vez de usar os termos apropriados em inglês. Comparar

```
class DtaRcrd102
{
    private Date genymdhms;
    data privada modymdhms;
    string final privada pszqint = "102"; /* ... */};
```

para

```
class Cliente
{
    private Date generationTimestamp;
    Private Data modificaçãoTimestamp;; private
    final String recordId = "102"; /* ... */};
```

A conversa inteligente agora é possível: "Ei, Mikey, dê uma olhada neste disco! O timestamp de geração está definido para a data de amanhã! Como pode ser?"

Use nomes pesquisáveis

Nomes de uma única letra e constantes numéricas têm um problema específico, pois não são fáceis de localizar em um corpo de texto.

Pode-se facilmente grep para MAX_CLASSES_PER_STUDENT, mas o número 7 pode ser mais problemático. As pesquisas podem revelar o dígito como parte de nomes de arquivos, outras definições de constantes e em várias expressões em que o valor é usado com intenções diferentes. É ainda pior quando uma constante é um número longo e alguém pode ter transposto dígitos, criando assim um bug ao mesmo tempo em que evita a busca do programador.

Da mesma forma, o nome e é uma má escolha para qualquer variável que um programador precise pesquisar. É a letra mais comum no idioma inglês e provavelmente aparece em todas as passagens de texto em todos os programas. A esse respeito, nomes mais longos superam nomes mais curtos e qualquer nome pesquisável supera uma constante no código.

Minha preferência pessoal é que nomes de uma única letra SÓ possam ser usados como variáveis locais dentro de métodos curtos. O comprimento de um nome deve corresponder ao tamanho de seu escopo

[N5]. Se uma variável ou constante puder ser vista ou usada em vários lugares em um corpo de código, é imperativo dar a ela um nome de fácil pesquisa. mais uma vez comparar

```
for (int j=0; j<34; j++) { s += (tjj)*4)/5;
}

para

int realDaysPerIdealDay = 4; const int
WORK_DAYS_PER_WEEK = 5; int soma =
0; for (int j=0; j < NUMBER_OF_TASKS; j++) { int realTaskDays =
taskEstimate[j] * realDaysPerIdealDay; int realTaskWeeks = (realdays /
WORK_DAYS_PER_WEEK); soma += realTaskWeeks;

}
```

Observe que sum, acima, não é um nome particularmente útil, mas pelo menos é pesquisável. O código nomeado intencionalmente cria uma função mais longa, mas considere como será mais fácil encontrar WORK_DAYS_PER_WEEK do que encontrar todos os lugares onde 5 foi usado e filtrar a lista apenas para as instâncias com o significado pretendido.

Evite Codificações

Temos codificações suficientes para lidar sem adicionar mais ao nosso fardo. Codificar informações de tipo ou escopo em nomes simplesmente adiciona uma carga extra de decifração. Dificilmente parece razoável exigir que cada novo funcionário aprenda outra "linguagem" de codificação, além de aprender o (geralmente considerável) corpo de código no qual trabalhará. É uma carga mental desnecessária ao tentar resolver um problema problema. Nomes codificados raramente são pronunciáveis e são fáceis de digitar incorretamente.

notação húngara

Antigamente, quando trabalhávamos em idiomas desafiados pelo tamanho do nome, violávamos essa regra por necessidade e com pesar. Fortran codificações forçadas tornando a primeira letra um código para o tipo. As primeiras versões do BASIC permitiam apenas uma letra mais um dígito. A notação húngara (HN) levou isso a um nível totalmente novo.

O HN era considerado muito importante na API do Windows C, quando tudo era um identificador inteiro ou um ponteiro longo ou um ponteiro nulo , ou uma das várias implementações de "string" (com diferentes usos e atributos). O compilador não verificava os tipos naquela época, então os programadores precisavam de uma muleta para ajudá-los a lembrar os tipos.

Nas linguagens modernas, temos sistemas de tipos muito mais ricos, e os compiladores lembram e impõem os tipos. Além do mais, há uma tendência para classes menores e funções mais curtas para que as pessoas geralmente possam ver o ponto de declaração de cada variável que estão usando.

Os programadores Java não precisam de codificação de tipo. Os objetos são fortemente tipados e os ambientes de edição avançaram tanto que detectam um erro de tipo muito antes que você possa executar uma compilação! Portanto, hoje em dia, HN e outras formas de codificação de tipo são simplesmente impedimentos. Eles tornam mais difícil alterar o nome ou o tipo de uma variável, função ou classe. Eles dificultam a leitura do código. E criam a possibilidade de que o sistema de codificação engane o leitor.

```
PhoneNumber phoneString; //  
nome não alterado quando o tipo é alterado!
```

Prefixos de membro

Você também não precisa mais prefixar as variáveis de membro com m_. Suas classes e funções devem ser pequenas o suficiente para que você não precise delas. E você deve usar um ambiente de edição que destaque ou colorize os membros para torná-los distintos.

```
classe pública Parte  
{ private String m_dsc; // A descrição textual void setName(String  
name){  
    m_dsc = nome;  
}  
}
```

```
classe pública Parte {  
    Descrição da string; void  
    setDescription(String descrição) { this.description =  
        descrição;  
    }  
}
```

Além disso, as pessoas aprendem rapidamente a ignorar o prefixo (ou sufixo) para ver a parte significativa do nome. Quanto mais lemos o código, menos vemos os prefixos. Eventualmente, os prefixos tornam-se uma desordem invisível e um marcador de código mais antigo.

Interfaces e Implementações

Às vezes, esses são um caso especial para codificações. Por exemplo, digamos que você esteja construindo uma FÁBRICA ABSTRATA para a criação de formas. Esta fábrica será uma interface e será implementada por uma classe concreta. O que você deve nomeá-los? IShapeFactory e ShapeFactory? Prefiro deixar as interfaces sem adornos. O eu anterior, tão comum nos maços legados de hoje, é, na melhor das hipóteses, uma distração e, na pior, muita informação. Não quero que meus usuários saibam que estou entregando a eles uma interface. Eu só quero que eles saibam que é uma ShapeFactory. Portanto, se devo codificar a interface ou a implementação, escolho a implementação. Chamá-lo de ShapeFactoryImp, ou mesmo o hediondo CShapeFactory, é preferível a codificar a interface.

Evite mapas mentais

Os leitores não deveriam ter que traduzir mentalmente seus nomes para outros nomes que eles já conhecem. Esse problema geralmente surge de uma escolha de não usar termos de domínio de problema nem termos de domínio de solução.

Este é um problema com nomes de variáveis de uma única letra. Certamente um contador de loop pode ser nomeado i ou j ou k (embora nunca !!) se seu escopo for muito pequeno e nenhum outro nome puder entrar em conflito com ele. Isso ocorre porque os nomes de uma única letra para contadores de loop são tradicionais. No entanto, na maioria dos outros contextos, um nome com uma única letra é uma escolha ruim; é apenas um espaço reservado que o leitor deve mapear mentalmente para o conceito real. Não pode haver pior razão para usar o nome c do que porque a e b já foram usados.

Em geral, os programadores são pessoas muito inteligentes. Pessoas inteligentes às vezes gostam de mostrar sua inteligência demonstrando suas habilidades de malabarismo mental. Afinal, se você puder lembrar com segurança que r é a versão em minúsculas da url com o host e o esquema removidos, então você deve ser muito esperto.

Uma diferença entre um programador inteligente e um programador profissional é que o profissional entende que a clareza é fundamental. Os profissionais usam seus poderes para o bem e escrevem códigos que outros possam entender.

Nomes de classe

Classes e objetos devem ter nomes de substantivos ou frases de substantivos como Customer, WikiPage, Account e AddressParser. Evite palavras como gerente, processador, dados ou informações no nome de uma classe. Um nome de classe não deve ser um verbo.

Nomes de métodos

Os métodos devem ter nomes de verbos ou frases verbais como postPayment, deletePage ou save. Acessadores, modificadores e predicados devem ser nomeados por seus valores e prefixados com get, set e estão de acordo com o padrão javabean.⁴

```
string nome = funcionário.getName();
cliente.setName("mike"); if
(salário.isPosted())...
```

Quando os construtores estiverem sobrecarregados, use métodos de fábrica estáticos com nomes que descreva os argumentos. Por exemplo,

```
Ponto de fulcro complexo = Complex.FromRealNumber(23.0);
geralmente é melhor do que
Ponto de fulcro complexo = new Complex(23.0);
Considere impor seu uso tornando privados os construtores correspondentes.
```

4. <http://java.sun.com/products/javabeans/docs/spec.html>

Não seja bonito

Se os nomes forem inteligentes demais, serão memoráveis apenas para as pessoas que compartilham o senso de humor do autor, e apenas enquanto essas pessoas se lembrarem da piada. Eles saberão o que a função chamada `HolyHandGrenade` deve fazer? Claro, é fofo, mas talvez neste caso `DeleteItems` seja um nome melhor.



Escolha clareza ao invés de valor de entretenimento.

A fofura no código geralmente aparece na forma de coloquialismos ou gírias. Por exemplo, não use o nome `whack()` para significar `kill()`. Não conte pequenas piadas dependentes de cultura como `eatMyShorts()` para significar `abort()`.

Diga o que você quer dizer. Significa o que você diz.

Escolha uma palavra por conceito

Escolha uma palavra para um conceito abstrato e fique com ela. Por exemplo, é confuso ter `fetch`, `retrieve` e `get` como métodos equivalentes de diferentes classes. Como você lembra qual nome de método combina com qual classe? Infelizmente, muitas vezes você precisa lembrar qual empresa, grupo ou indivíduo escreveu a biblioteca ou classe para lembrar qual termo foi usado. Caso contrário, você gastará muito tempo navegando pelos cabeçalhos e amostras de código anteriores.

Ambientes de edição modernos, como Eclipse e IntelliJ, fornecem pistas sensíveis ao contexto, como a lista de métodos que você pode chamar em um determinado objeto. Mas observe que a lista geralmente não fornece os comentários que você escreveu sobre os nomes de suas funções e listas de parâmetros.

Você terá sorte se fornecer os *nomes* dos parâmetros das declarações de função. Os nomes das funções devem ser independentes e devem ser consistentes para que você escolha o método correto sem nenhuma exploração adicional.

Da mesma forma, é confuso ter um controlador , um gerenciador e um driver na mesma base de código. Qual é a diferença essencial entre um `DeviceManager` e um `Protocol Controller`? Por que ambos não são controladores ou ambos não são gerentes? Ambos são Drivers mesmo? O nome leva você a esperar dois objetos que têm tipos muito diferentes, bem como classes diferentes.

Um léxico consistente é um grande benefício para os programadores que devem usar seu código.

não trocadilho

Evite usar a mesma palavra para dois propósitos. Usar o mesmo termo para duas ideias diferentes é essencialmente um trocadilho.

Se você seguir a regra “uma palavra por conceito”, poderá acabar com muitas classes que possuem, por exemplo, um método add . Desde que as listas de parâmetros e os valores de retorno dos vários métodos add sejam semanticamente equivalentes, está tudo bem.

No entanto, alguém pode decidir usar a palavra adicionar para “consistência” quando não está de fato adicionando no mesmo sentido. Digamos que temos muitas classes nas quais add criará um novo valor adicionando ou concatenando dois valores existentes. Agora, digamos que estamos escrevendo uma nova classe que possui um método que coloca seu único parâmetro em uma coleção. Devemos chamar esse método de add? Pode parecer consistente porque temos tantos outros métodos add , mas neste caso, a semântica é diferente, então devemos usar um nome como insert ou append . Chamar o novo método add seria um trocadilho.

Nosso objetivo, como autores, é tornar nosso código o mais fácil possível de entender. Queremos que nosso código seja uma leitura rápida, não um estudo intenso. Queremos usar o modelo popular de brochura, segundo o qual o autor é responsável por se tornar claro, e não o modelo acadêmico, em que o trabalho do estudioso é extrair o significado do artigo.

Usar nomes de domínio da solução

Lembre-se de que as pessoas que lerão seu código serão programadores. Portanto, vá em frente e use termos de ciência da computação (CS), nomes de algoritmos, nomes de padrões, termos matemáticos e assim por diante. Não é sensato tirar todos os nomes do domínio do problema porque não queremos que nossos colegas de trabalho tenham que ir e voltar do cliente perguntando o que cada nome significa quando eles já conhecem o conceito por um nome diferente.

O nome AccountVisitor significa muito para um programador familiarizado com o padrão VISITOR . Que programador não saberia o que era uma JobQueue ? Há muitas coisas muito técnicas que os programadores precisam fazer. Escolher nomes técnicos para essas coisas geralmente é o caminho mais apropriado.

Use nomes de domínio problemáticos

Quando não houver “programmer-eese” para o que você está fazendo, use o nome do domínio do problema. Pelo menos o programador que mantém seu código pode perguntar a um especialista de domínio o que isso significa.

Separar conceitos de solução e domínio de problema faz parte do trabalho de um bom programador e designer. O código que tem mais a ver com os conceitos do domínio do problema deve ter nomes extraídos do domínio do problema.

Adicionar contexto significativo

Existem alguns nomes que são significativos por si mesmos - a maioria não é. Em vez disso, você precisa colocar os nomes no contexto para o seu leitor, colocando-os em classes, funções ou namespaces bem nomeados. Quando tudo mais falhar, prefixar o nome pode ser necessário como último recurso.

Imagine que você tenha variáveis denominadas firstName, lastName, street, houseNumber, city, state e zipcode. Juntos, fica bem claro que eles formam um endereço. Mas e se você acabasse de ver a variável de estado sendo usada sozinha em um método? Você inferiria automaticamente que fazia parte de um endereço?

Você pode adicionar contexto usando prefixos: addrFirstName, addrLastName, addrState e assim por diante. Pelo menos os leitores entenderão que essas variáveis fazem parte de uma estrutura maior. Obviamente, uma solução melhor é criar uma classe chamada Address. Então, até o compilador sabe que as variáveis pertencem a um conceito maior.

Considere o método da Listagem 2-1. As variáveis precisam de um contexto mais significativo? O nome da função fornece apenas parte do contexto; o algoritmo fornece o resto. Depois de ler a função, você verá que as três variáveis, número, verbo e pluralModifier, fazem parte da mensagem “estatísticas de palpite”. Infelizmente, o contexto deve ser inferido. Quando você olha o método pela primeira vez, os significados das variáveis são opacos.

Listagem

2-1 Variáveis com contexto pouco claro.

```
private void printGuessStatistics(char candidate, int count) { String number; Verbo de
cadeia; String
pluralModifier; if
(contagem == 0) { número =
"não"; verbo = "são";
pluralModifier =
"s"; } else if (count
== 1) { número = "1"; verbo
= "é"; pluralModifier = ""; } else
{ número =

Integer.toString(contagem);
verbo =
"são"; pluralModifier = "s"; }

String palpiteMensagem = String.format("Há %s
%s %s%s", verbo, número, candidato, pluralModifier ); print(mensagem de palpite);

}
```

A função é um pouco longa demais e as variáveis são usadas por toda parte. Para dividir a função em pedaços menores, precisamos criar uma classe GuessStatisticsMessage e fazer os três campos variáveis dessa classe. Isso fornece um contexto claro para as três variáveis. Eles são *definitivamente* parte do GuessStatisticsMessage. A melhoria do contexto também permite que o algoritmo seja muito mais limpo, dividindo-o em várias funções menores. (Consulte a Listagem 2-2.)

Listagem 2-2**As variáveis têm um contexto.**

```

public class GuessStatisticsMessage { private String
    number; verbo de String
    privada; private String
    pluralModifier;

    public String make(char candidate, int count)
        { createPluralDependentMessageParts(count); return
        String.format("Há %s %s
        %s%s", verbo, número,
        candidato, pluralModifier );
    }

    private void createPluralDependentMessageParts(int count) { if (count == 0)
        { thereAreNoLetters(); }
        else if (count == 1)
        { therelsOneLetter(); } else
        { thereAreManyLetters(count);

    }

    private void thereAreManyLetters(int count) { number =
        Integer.toString(count); verbo = "são";
        pluralModifier =
        "s";
    }

    private void therelsOneLetter() { number =
        "1"; verbo = "é";
        pluralModifier =
        "";
    }

    private void thereAreNoLetters() { número =
        "não"; verbo =
        "são";
        pluralModifier = "s";
    }
}

```

Não adicione contexto gratuito

Em um aplicativo imaginário chamado “Gas Station Deluxe”, é uma má ideia prefixar todas as classes com GSD. Francamente, você está trabalhando contra suas ferramentas. Você digita G e pressiona a tecla de conclusão e é recompensado com uma lista de uma milha de cada classe no sistema. Isso é sábio? Por que dificultar a ajuda do IDE?

Da mesma forma, digamos que você inventou uma classe MailingAddress no módulo de contabilidade do GSD e a nomeou GSDAccountAddress. Posteriormente, você precisará de um endereço de correspondência para o aplicativo de contato com o cliente. Você usa GSDAccountAddress? Parece o nome certo? Dez dos 17 caracteres são redundantes ou irrelevantes.

Nomes mais curtos geralmente são melhores do que nomes mais longos, desde que sejam claros. Adicionar não mais contexto para um nome do que o necessário.

Os nomes `accountAddress` e `customerAddress` são bons nomes para instâncias da classe `Address`, mas podem ser nomes ruins para classes. `Endereço` é um bom nome para uma classe. Se eu precisar diferenciar entre endereços MAC, endereços de porta e endereços da Web, posso considerar `PostalAddress`, MAC e URI. Os nomes resultantes são mais precisos, que é o objetivo de toda nomeação.

Palavras Finais

A coisa mais difícil na escolha de bons nomes é que isso requer boas habilidades descritivas e um histórico cultural compartilhado. Esta é uma questão de ensino, e não uma questão técnica, de negócios ou de gerenciamento. Como resultado, muitas pessoas neste campo não aprendem a fazê-lo muito bem.

As pessoas também têm medo de renomear as coisas por medo de que outros desenvolvedores se oponham. Não compartilhamos desse medo e descobrimos que somos realmente gratos quando os nomes mudam (para melhor). Na maioria das vezes, não memorizamos os nomes das classes e métodos. Usamos as ferramentas modernas para lidar com detalhes como esse, para que possamos nos concentrar se o código é lido como parágrafos e sentenças ou, pelo menos, como tabelas e estrutura de dados (uma sentença nem sempre é a melhor maneira de exibir dados). Você provavelmente acabará surpreendendo alguém ao renomear, assim como faria com qualquer outra melhoria de código. Não deixe que isso pare você em suas trilhas.

Siga algumas dessas regras e veja se você não melhora a legibilidade do seu código. Se você estiver mantendo o código de outra pessoa, use ferramentas de refatoração para ajudar a resolver esses problemas. Vai pagar no curto prazo e continuar a pagar no longo prazo.

3

Funções



Nos primórdios da programação, compusemos nossos sistemas de rotinas e sub-rotinas. Então, na era do Fortran e PL/1, compusemos nossos sistemas de programas, subprogramas e funções. Hoje em dia apenas a função sobrevive daqueles primeiros dias. Funções são a primeira linha de organização em qualquer programa. Escrevê-los bem é o tópico deste capítulo.

Considere o código na Listagem 3-1. É difícil encontrar uma função longa no FitNesse,¹ mas depois de pesquisar um pouco encontrei esta. Não só é longo, mas tem código duplicado, muitas strings estranhas e muitos tipos de dados e APIs estranhos e inóbvios. Veja quanto sentido você consegue entender nos próximos três minutos.

Listagem

3-1 HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup ) lança
exceção {
    WikiPage wikiPage = pageData.getWikiPage(); Buffer
    StringBuffer = new StringBuffer(); if (pageData.
    hasAttribute("Test")) { if (includeSuiteSetup)
        { WikiPage suiteSetup =
            PageCrawlerImpl.
            getPageCrawler().getInheritedPage( SuiteResponder.
            SUITE_SETUP_NAME, wikiPage
        );
        if (suiteSetup != null) { WikiPagePath
            pagePath =
                suiteSetup.getPageCrawler().getFullPath(suiteSetup);
            String pagePathName = PathParser.render(pagePath); buffer.append("!
            include
                -setup .") .append(pagePathName) .append("\n");
        }
    }
    Configuração da
    WikiPage = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage); if
    (setUp != null) { WikiPagePath
        setupPath =
            wikiPage.getPageCrawler().getFullPath(setup); String
        setupPathName = PathParser.render(setupPath); buffer.append("\ninclude
            -setup .") .append(setupPathName) .append("\n");
    }

}
} buffer.append(pageData.getContent()); if
(pageData.hasAttribute("Test")) {
    Desmontagem da WikiPage =
        PageCrawlerImpl.getInheritedPage("TearDown", wikiPage); if (tearDown !=
    null) { WikiPagePath
        tearDownPath =
            wikiPage.getPageCrawler().getFullPath(teardown); String
        tearDownPathName = PathParser.render(tearDownPath);
        buffer.append("\n") .append("!
            include
                -teardown .") .append(tearDownPathName) .append("\n");
    }
}
```

1. Uma ferramenta de teste de código aberto. www.fitnesse.org

Listagem 3-1 (continuação)**HtmlUtil.java (FitNesse 20070619)**

```

if (includeSuiteSetup) {
    WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage( SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage
    );
    if (suiteTeardown != null) { WikiPagePath
        pagePath =
            suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath); buffer.append("!
            include -teardown .") .append(pagePathName) .append("\n");
    }
}

} pageData.setContent(buffer.toString()); return
pageData.getHtml();
}

```

Você entende a função após três minutos de estudo? Provavelmente não. Há muita coisa acontecendo lá em muitos níveis diferentes de abstração. Existem strings estranhas e chamadas de função estranhas misturadas com instruções if duplamente aninhadas controladas por sinalizadores.

No entanto, com apenas algumas extrações de método simples, algumas renomeações e uma pequena reestruturação, consegui capturar a intenção da função nas nove linhas da Listagem 3-2. Veja se você consegue entender isso nos próximos 3 minutos.

Listagem**3-2 HtmlUtil.java (refatorado)**

```

public static String renderPageWithSetupsAndTeardowns(PageData
    pageData, boolean isSuite) throws Exception
{ boolean isTestPage =
    pageData.hasAttribute("Test"); if (isTestPage) { WikiPage testPage =
    pageData.getWikiPage();
    StringBuffer newPasswordContent = new StringBuffer();
    includeSetupPages(testPage, newPasswordContent, isSuite);
    newPasswordContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPasswordContent, isSuite);
    pageData.setContent(newPageContent.toString()); }

    return pageData.getHtml();
}

```

A menos que você seja um aluno do FitNesse, provavelmente não entende todos os detalhes. Ainda assim, você provavelmente entende que essa função executa a inclusão de algumas páginas de configuração e desmontagem em uma página de teste e, em seguida, renderiza essa página em HTML. Se você estiver familiarizado com o JUnit,² provavelmente percebeu que essa função pertence a algum tipo de estrutura de teste baseada na Web. E, claro, isso está correto. Adivinhar essas informações da Listagem 3-2 é muito fácil, mas é bastante obscurecido pela Listagem 3-1.

Então, o que torna uma função como a Listagem 3-2 fácil de ler e entender? Como podemos fazer uma função comunicar sua intenção? Que atributos podemos dar às nossas funções que permitirão a um leitor casual intuir o tipo de programa em que vivem?

Pequeno!

A primeira regra das funções é que elas devem ser pequenas. A segunda regra das funções é que *elas devem ser menores que isso*. Esta não é uma afirmação que eu possa justificar. Não posso fornecer referências a pesquisas que mostrem que funções muito pequenas são melhores. O que posso dizer é que, por quase quatro décadas, escrevi funções de todos os tamanhos diferentes. Escrevi dez abominações desagradáveis de 3.000 linhas. Eu escrevi várias funções no intervalo de 100 a 300 linhas. E escrevi funções com 20 a 30 linhas. O que essa experiência me ensinou, por meio de tentativas e erros, é que as funções devem ser muito pequenas.

Nos anos 80 costumávamos dizer que uma função não deveria ser maior que uma tela inteira. É claro que dissemos isso em uma época em que as telas do VT100 tinham 24 linhas por 80 colunas e nossos editores usavam 4 linhas para fins administrativos. Hoje em dia, com uma fonte reduzida e um bom monitor grande, você pode colocar 150 caracteres em uma linha e 100 linhas ou mais em uma tela. As linhas não devem ter 150 caracteres. As funções não devem ter 100 linhas. As funções dificilmente devem ter 20 linhas.

Quão curta deve ser uma função? Em 1999 fui visitar Kent Beck em sua casa em Oregon. Nós nos sentamos e fizemos alguma programação juntos. A certa altura, ele me mostrou um programinha Java/Swing bonitinho que chamou de *Sparkle*. Produziu um efeito visual na tela muito parecido com a varinha mágica da fada madrinha do filme Cinderela. À medida que você movia o mouse, os brilhos pingavam do cursor com uma cintilação satisfatória, caindo na parte inferior da janela por meio de um campo gravitacional simulado. Quando Kent me mostrou o código, fiquei impressionado com o quão pequenas todas as funções eram. Eu estava acostumado a funções em programas Swing que ocupavam quilômetros de espaço vertical. Cada função *neste* programa tinha apenas duas, três ou quatro linhas. Cada um era transparentemente óbvio. Cada um contou uma história. E cada um levou você ao próximo em uma ordem convincente. É assim que suas funções devem ser curtas!³

2. Uma ferramenta de teste de unidade de código aberto para Java.

www.junit.org 3. Perguntei a Kent se ele ainda tinha uma cópia, mas ele não conseguiu encontrar. Também procurei em todos os meus computadores antigos, mas sem sucesso.

Tudo o que resta agora é a minha memória daquele programa.

Quão curtas devem ser suas funções? Eles geralmente devem ser mais curtos do que a Listagem 3-2! De fato, a Listagem 3-2 deveria ser abreviada para a Listagem 3-3.

Listagem 3-3

HtmlUtil.java (refatorado)

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, booleano isSuite) lança Exception { if
        (isTestPage(pageData))
            includeSetupAndTeardownPages(pageData, isSuite); return
        pageData.getHtml();
    }
```

Blocos e recuo

Isso implica que os blocos dentro das instruções if , else , while e assim por diante devem ter o comprimento de uma linha. Provavelmente essa linha deve ser uma chamada de função. Isso não apenas mantém a função delimitadora pequena, mas também agrupa valor documental porque a função chamada dentro do bloco pode ter um nome bem descriptivo.

Isso também implica que as funções não devem ser grandes o suficiente para conter estruturas aninhadas. Portanto, o nível de recuo de uma função não deve ser maior que um ou dois. Isso, é claro, torna as funções mais fáceis de ler e entender.

Faça uma coisa

Deve ficar muito claro que a Listagem 3-1 está fazendo muito mais do que uma coisa. Ele está criando buffers, buscando páginas, procurando por páginas herdadas, renderizando caminhos, anexando strings misteriosas e gerando HTML, entre outras coisas. A Listagem 3-1 está muito ocupada fazendo muitas coisas diferentes. Por outro lado, a Listagem 3-3 está fazendo uma coisa simples. Está incluindo configurações e desmontagens em páginas de teste.

O conselho a seguir apareceu de uma forma ou de outra por 30 anos ou mais.

**AS FUNÇÕES DEVEM FAZER UMA COISA. ELES DEVEM FAZER BEM.
SÓ DEVEM FAZER ISSO.**



O problema com essa afirmação é que é difícil saber o que é “uma coisa”. Faz Listagem 3-3 faz uma coisa? É fácil argumentar que ele está fazendo três coisas:

1. Determinar se a página é uma página de teste.
2. Em caso afirmativo, incluindo configurações e desmontagens.
3. Renderizando a página em HTML.

Então qual é? A função está fazendo uma coisa ou três coisas? Observe que as três etapas da função estão um nível de abstração abaixo do nome declarado da função. Podemos descrever a função descrevendo-a como um breve parágrafo *TO4*:

TO RenderPageWithSetupsAndTeardowns, verificamos se a página é uma página de teste e, em caso afirmativo, incluímos as configurações e desmontagens. Em ambos os casos, renderizamos a página em HTML.

Se uma função executa apenas as etapas que estão um nível abaixo do nome declarado da função, a função está fazendo uma coisa. Afinal, a razão pela qual escrevemos funções é decompor um conceito maior (em outras palavras, o nome da função) em um conjunto de etapas no próximo nível de abstração.

Deve ficar bem claro que a Listagem 3-1 contém etapas em muitos níveis diferentes de abstração. Portanto, está claramente fazendo mais de uma coisa. Mesmo a Listagem 3-2 tem dois níveis de abstração, conforme comprovado por nossa capacidade de reduzi-la. Mas seria muito difícil reduzir significativamente a Listagem 3-3. Poderíamos extrair a instrução `if` em uma função chamada `includeSetupsAndTeardownsIfTestPage`, mas isso simplesmente reafirma o código sem alterar o nível de abstração.

Então, outra forma de saber que uma função está fazendo mais do que “uma coisa” é se você pode extrair outra função dela com um nome que não seja meramente uma reafirmação de sua implementação [G34].

Seções dentro de Funções

Veja a Listagem 4-7 na página 71. Observe que a função `generatePrimes` é dividida em seções como *declarações, inicializações e peneira*. Este é um sintoma óbvio de fazer mais de uma coisa. Funções que fazem uma coisa não podem ser razoavelmente divididas em seções.

Um nível de abstração por função

Para garantir que nossas funções estejam fazendo “uma coisa”, precisamos garantir que as instruções dentro de nossa função estejam todas no mesmo nível de abstração. É fácil ver como a Listagem 3-1 viola essa regra. Existem conceitos que estão em um nível muito alto de abstração, como `getHtml()`; outras que estão em um nível intermediário de abstração, como: `String pagePathName = PathParser.render(pagePath)`; e ainda outros que são notavelmente de baixo nível, como: `.append("\n")`.

Misturar níveis de abstração dentro de uma função é sempre confuso. Os leitores podem não ser capazes de dizer se uma determinada expressão é um conceito essencial ou um detalhe. Pior,

4. A linguagem LOGO usava a palavra-chave “TO” da mesma forma que Ruby e Python usam “def”. Assim, cada função começava com a palavra “TO”. Isso teve um efeito interessante na maneira como as funções foram projetadas.

como janelas quebradas, uma vez que detalhes são misturados com conceitos essenciais, mais e mais detalhes tendem a se acumular dentro da função.

Lendo o código de cima para baixo: a regra Stepdown

Queremos que o código seja lido como uma narrativa de cima para baixo.⁵ Queremos que cada função seja seguida por aquelas no próximo nível de abstração para que possamos ler o programa, descendo um nível de abstração por vez enquanto lemos a lista de funções. Eu chamo isso de *Regra do Degrau*.

Em outras palavras, queremos ser capazes de ler o programa como se fosse um conjunto de parágrafos *TO*, cada um dos quais descrevendo o nível atual de abstração e referenciando os parágrafos *TO* subsequentes no próximo nível abaixo.

Para incluir as configurações e desmontagens, incluímos as configurações, depois incluímos o conteúdo da página de teste e depois incluímos as desmontagens.

Para incluir as configurações, incluímos a configuração da suíte, se for uma suíte, então incluímos a configuração regular.

Para incluir a configuração do conjunto, procuramos na hierarquia pai a página "SuiteSetUp" e adicione uma declaração de inclusão com o caminho dessa página.

Para procurar o pai. . .

Acontece que é muito difícil para os programadores aprender a seguir essa regra e escrever funções que ficam em um único nível de abstração. Mas aprender esse truque também é muito importante. É a chave para manter as funções curtas e garantir que elas façam "uma coisa". Fazer o código ser lido como um conjunto de parágrafos *TO* de cima para baixo é uma técnica eficaz para manter o nível de abstração consistente.

Dê uma olhada na Listagem 3-7 no final deste capítulo. Ele mostra toda a função `testableHtml` refatorada de acordo com os princípios descritos aqui. Observe como cada função introduz a próxima e cada função permanece em um nível consistente de abstração.

Declarações de troca

É difícil fazer uma instrução `switch` pequena.⁶ Mesmo uma instrução `switch` com apenas dois casos é maior do que eu gostaria que fosse um único bloco ou função. Também é difícil fazer uma declaração `switch` que faça uma coisa. Por sua natureza, as instruções `switch` sempre fazem *N* coisas. Infelizmente, nem sempre podemos evitar as instruções `switch`, mas podemos garantir que cada instrução `switch` seja enterrada em uma classe de baixo nível e nunca seja repetida. Fazemos isso, é claro, com polimorfismo.

5. [KP78], pág. 37.

6. E, é claro, incluo cadeias `if/else` nisso.

Considere a Listagem 3-4. Mostra apenas uma das operações que podem depender do tipo de empregado.

Listagem

3-4 Payroll.java

```
public Money calculatePay(Employee e) throws
InvalidEmployeeType { switch
(e.type) {
caso COMISSIONADO:
    return calcularCommissionedPay(e);
caso HORA:
    return calcularHoraPag(e);
case SALARIED:
    return calculaSalarioPag(e); padrão: lançar
novo
    InvalidEmployeeType(e.type);
}
}
```

Existem vários problemas com esta função. Primeiro, é grande e, quando novos tipos de funcionários forem adicionados, ele crescerá. Em segundo lugar, ele claramente faz mais de uma coisa.

Em terceiro lugar, viola o Princípio da Responsabilidade Única⁷ (SRP) porque há mais de um motivo para sua mudança. Quarto, ele viola o Open Closed Principle⁸ (OCP) porque deve mudar sempre que novos tipos são adicionados. Mas possivelmente o pior problema com esta função é que existe um número ilimitado de outras funções que terão a mesma estrutura. Por exemplo poderíamos ter

isPayday(Empregado e, Data data),

ou

entregaPagamento(Empregado e, Pagamento em dinheiro),

ou uma série de outros. Todos os quais teriam a mesma estrutura deletéria.

A solução para esse problema (consulte a Listagem 3-5) é enterrar a instrução switch no e nunca Porão de uma FÁBRICA ABSTRATA ,⁹ permitir que ninguém a veja. A fábrica usará o instrução switch para criar instâncias apropriadas das derivadas de Employee, e as várias funções, como calculatePay, isPayday e deliveryPay, serão despachadas polimorficamente através da interface Employee .

Minha regra geral para instruções switch é que elas podem ser toleradas se aparecerem apenas uma vez, forem usadas para criar objetos polimórficos e estiverem ocultas atrás de uma herança

7. a. http://en.wikipedia.org/wiki/Single_responsibility_principle
b. <http://www.objectmentor.com/resources/articles/srp.pdf>

8. a. http://en.wikipedia.org/wiki/Open/closed_principle
b. <http://www.objectmentor.com/resources/articles/ocp.pdf> 9. [GOF].

Listagem 3-5**Employee and Factory public**

```

abstract class Employee {
    public abstract booleano isPayday(); public abstract
    Dinheiro calcularPagamento(); public abstract void
    entregarPagar(Dinheiro pagar);
}

-----
interface pública FuncionárioFábrica {
    public Employee makeEmployee(EmployeeRecord r) lança InvalidEmployeeType;
}
-----

public class EmployeeFactoryImpl implementa EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType { switch (r.type) {

        caso COMISSONADO:
            return new ComissionadoEmployee(r) ;
        caso HORA:
            return new HourlyEmployee(r);
        caso SALÁRIO:
            return new SalariedEmployee(r); padrão: lançar
        novo
            InvalidEmployeeType(r.type);
    }
}
}

```

relacionamento para que o resto do sistema não possa vê-los [G23]. É claro que cada circunstância é única e há momentos em que viola uma ou mais partes dessa regra.

Use nomes descritivos

Na Listagem 3-7, alterei o nome de nossa função de exemplo de testableHtml para SetupTeardownIncluder.render. Este é um nome muito melhor porque descreve melhor o que a função faz. Também dei a cada um dos métodos privados um nome igualmente descritivo, como isTestable ou includeSetupAndTeardownPages. É difícil superestimar o valor de bons nomes. Lembre-se do princípio de Ward: “*Você sabe que está trabalhando em código limpo quando cada rotina acaba sendo exatamente o que você esperava*”. Metade da batalha para alcançar esse princípio é escolher bons nomes para pequenas funções que fazem uma coisa.

Quanto menor e mais focada for uma função, mais fácil será escolher um descritivo nome.

Não tenha medo de fazer um nome longo. Um nome longo e descritivo é melhor do que um nome curto e enigmático. Um nome descritivo longo é melhor do que um comentário descritivo longo. Use uma convenção de nomenclatura que permita que várias palavras sejam facilmente lidas nos nomes das funções e, em seguida, use essas várias palavras para dar à função um nome que diga o que ela faz.

Não tenha medo de gastar tempo escolhendo um nome. Na verdade, você deve tentar vários nomes diferentes e ler o código com cada um no lugar. IDEs modernos como Eclipse ou IntelliJ tornam trivial a mudança de nomes. Use um desses IDEs e experimente nomes diferentes até encontrar um que seja o mais descritivo possível.

A escolha de nomes descritivos esclarecerá o design do módulo em sua mente e o ajudará a melhorá-lo. Não é incomum que a busca por um bom nome resulte em uma reestruturação favorável do código.

Seja consistente em seus nomes. Use as mesmas frases, substantivos e verbos nos nomes das funções que você escolher para seus módulos. Considere, por exemplo, os nomes `includeSetup AndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. A fraseologia semelhante nesses nomes permite que a sequência conte uma história. Na verdade, se eu mostrasse apenas a sequência acima, você se perguntaria: “O que aconteceu com `includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`?” Como é isso por ser “... praticamente o que você esperava.”

Argumentos da função

O número ideal de argumentos para uma função é zero (niládico). Em seguida vem um (monádico), seguido de perto por dois (diádico). Três argumentos (triádicos) devem ser evitados sempre que possível. Mais de três (poliádico) requer uma justificação muito especial - e então não deve ser usado de qualquer maneira.

Argumentos são difíceis. Eles exigem muito poder conceitual. É por isso que me livrei de quase todos eles do exemplo. Considere, por exemplo, o `StringBuffer` no exemplo. Poderíamos tê-lo passado como um argumento em vez de torná-lo uma variável de instância, mas nossos leitores teriam que interpretá-lo cada vez que o vissem. Quando você está lendo a história contada pelo módulo, `includeSetupPage()` é mais fácil de entender do que `includeSetupPageInto(newPage Content)`. O argumento está em um nível de abstração diferente do nome da função e força você a conhecer um detalhe (em outras palavras, `StringBuffer`) que não é particularmente importante nesse ponto.



Os argumentos são ainda mais difíceis do ponto de vista do teste. Imagine a dificuldade de escrever todos os casos de teste para garantir que todas as várias combinações de argumentos funcionem corretamente. Se não houver argumentos, isso é trivial. Se houver um argumento, não é muito difícil. Com dois argumentos, o problema fica um pouco mais desafiador. Com mais de dois argumentos, testar cada combinação de valores apropriados pode ser assustador.

Os argumentos de saída são mais difíceis de entender do que os argumentos de entrada. Quando lemos uma função, estamos acostumados com a ideia de que as informações entram *na* função por meio de argumentos e *saiem* pelo valor de retorno. Normalmente, não esperamos que as informações saiam por meio dos argumentos. Portanto, os argumentos de saída geralmente nos levam a pensar duas vezes.

Um argumento de entrada é a próxima melhor coisa para nenhum argumento. `SetupTeardownIncluder.render(pageData)` é muito fácil de entender. Claramente, vamos *renderizar* os dados no objeto `pageData`.

Formas Monádicas Comuns

Existem duas razões muito comuns para passar um único argumento para uma função. Você pode estar fazendo uma pergunta sobre esse argumento, como em `boolean fileExists("MyFile")`. Ou você pode estar operando com base nesse argumento, transformando-o em outra coisa e *devolvendo-o*. Por exemplo, `InputStream fileOpen("MyFile")` transforma um nome de arquivo `String` em um valor de retorno `InputStream`. Esses dois usos são o que os leitores esperam quando veem uma função. Você deve escolher nomes que tornem clara a distinção e sempre usar as duas formas em um contexto consistente. (Consulte Separação de consulta de comando abaixo.)

Uma forma um pouco menos comum, mas ainda muito útil para uma única função de argumento, é um *evento*. Nesta forma, há um argumento de entrada, mas nenhum argumento de saída. O programa geral destina-se a interpretar a chamada de função como um evento e usar o argumento para alterar o estado do sistema, por exemplo, `void passwordAttemptFailedNtimes(int tries)`. Use este formulário com cuidado. Deve ficar bem claro para o leitor que se trata de um evento. Escolha nomes e contextos com cuidado.

Tente evitar quaisquer funções monádicas que não sigam esses formulários, por exemplo, `void includeSetupPageInto(StringBuffer pageText)`. Usar um argumento de saída em vez de um valor de retorno para uma transformação é confuso. Se uma função vai transformar seu argumento de entrada, a transformação deve aparecer como o valor de retorno. De fato, `StringBuffer transform(StringBuffer in)` é melhor que `void transform-(StringBuffer out)`, mesmo que a implementação no primeiro caso simplesmente retorne o argumento de entrada. Pelo menos ainda segue a forma de uma transformação.

Argumentos sinalizadores

Argumentos sinalizadores são feios. Passar um booleano para uma função é uma prática realmente terrível. Isso complica imediatamente a assinatura do método, proclamando em voz alta que essa função faz mais de uma coisa. Ele faz uma coisa se a bandeira for verdadeira e outra se a bandeira for falsa!

Na Listagem 3-7, não tínhamos escolha porque os chamadores já estavam passando esse sinalizador e eu queria limitar o escopo da refatoração para a função e abaixo. Ainda assim, a chamada do método `render(true)` é simplesmente confusa para um leitor ruim. Passar o mouse sobre a chamada e ver `render(boolean isSuite)` ajuda um pouco, mas não tanto. Deveríamos ter dividido a função em duas: `renderForSuite()` e `renderForSingleTest()`.

Funções diádicas

Uma função com dois argumentos é mais difícil de entender do que uma função monádica. Para a prova ¹⁰
Por exemplo, writeField(name) é mais fácil de entender do que writeField(output-Stream, name).

Embora o significado de ambos seja claro, o primeiro desliza além do olho, depositando facilmente seu significado. O segundo requer uma pequena pausa até aprendermos a ignorar o primeiro parâmetro. E isso, é claro, acaba resultando em problemas porque nunca devemos ignorar nenhuma parte do código. As partes que ignoramos são onde os insetos se esconderão.

Há momentos, é claro, em que dois argumentos são apropriados. Por exemplo, Ponto p = new Ponto(0,0); é perfeitamente razoável. Os pontos cartesianos naturalmente levam dois argumentos. De fato, ficaríamos muito surpresos em ver o novo Point(0). No entanto, os dois argumentos neste caso são componentes ordenados de um único valor! Considerando que output-Stream e name não possuem nem uma coesão natural, nem uma ordenação natural.

Mesmo funções diádicas óbvias como assertEquals(esperado, real) são problemáticas. Quantas vezes você colocou o real onde deveria estar o esperado? Os dois argumentos não têm ordenação natural. A ordem esperada e real é uma convenção que requer prática para aprender.

Díades não são más, e você certamente terá que escrevê-las. No entanto, você deve estar ciente de que eles têm um custo e devem aproveitar os mecanismos disponíveis para convertê-los em mônadas. Por exemplo, você pode tornar o método writeField um membro de outputStream para poder dizer outputStream. escrevaCampo(nome). Ou você pode tornar o outputStream uma variável de membro da classe atual para que não precise transmiti-lo. Ou você pode extrair uma nova classe como FieldWriter que usa o outputStream em seu construtor e tem um método de gravação .

tríades

Funções que levam três argumentos são significativamente mais difíceis de entender do que díades. Os problemas de ordenar, pausar e ignorar são mais do que duplicados. Sugiro que pense bem antes de criar uma tríade.

Por exemplo, considere a sobrecarga comum de assertEquals que usa três argumentos: assertEquals(mensagem, esperado, real). Quantas vezes você já leu a mensagem e achou que era o esperado? Muitas vezes tropecei e parei nessa tríade em particular. Na verdade, *toda vez que vejo*, dou uma olhada dupla e aprendo a ignorar a mensagem.

Por outro lado, aqui está uma tríade que não é tão insidiosa: assertEquals(1.0, amount, .001). Embora isso ainda exija uma olhada dupla, vale a pena. É sempre bom lembrar que a igualdade de valores de ponto flutuante é algo relativo.

10. Acabei de refatorar um módulo que usava a forma diádica. Conseguir fazer do outputStream um campo da classe e converter todas as chamadas writeField para a forma monádica. O resultado foi muito mais limpo.

Objetos de Argumento

Quando uma função parece precisar de mais de dois ou três argumentos, é provável que alguns desses argumentos devam ser agrupados em uma classe própria. Considere, por exemplo, a diferença entre as duas declarações a seguir:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Centro do ponto, raio duplo);
```

Reducir o número de argumentos criando objetos a partir deles pode parecer trapaça, mas não é. Quando grupos de variáveis são passados juntos, como x e y no exemplo acima, eles provavelmente fazem parte de um conceito que merece o nome de ter.

Listas de argumentos

Às vezes, queremos passar um número variável de argumentos para uma função. Considere, por exemplo, o método String.format :

```
String.format("%s trabalhou %.2f horas.", nome, horas);
```

Se os argumentos variáveis forem todos tratados de forma idêntica, como no exemplo acima, eles serão equivalentes a um único argumento do tipo List. Por esse raciocínio, String.format é realmente diádico. De fato, a declaração de String.format mostrado abaixo é claramente diádica.

```
public String format(String format, Object... args)
```

Portanto, todas as mesmas regras se aplicam. As funções que recebem argumentos variáveis podem ser mônadas, diádes ou mesmo tríades. Mas seria um erro dar-lhes mais argumentos do que isso.

```
void monad(Integer... args); void
dyad(String nome, Integer... args); void triad(String
name, int count, Integer... args);
```

Verbos e palavras-chave

Escolher bons nomes para uma função pode ajudar muito a explicar a intenção da função e a ordem e intenção dos argumentos. No caso de uma mònada, a função e o argumento devem formar um belo par verbo/substantivo. Por exemplo, write(name) é muito sugestivo. Qualquer que seja essa coisa de "nome", ela está sendo "escrita". Um nome ainda melhor pode ser writeField(name), que nos diz que o "nome" é um "campo".

Este último é um exemplo da forma *de palavra-chave* de um nome de função. Usando esta forma, codificamos os nomes dos argumentos no nome da função. Por exemplo, assertEquals pode ser melhor escrito como assertEqualsActual(esperado, real). Isso atenua fortemente o problema de ter que lembrar a ordem dos argumentos.

Não Tem Efeitos Colaterais

Os efeitos colaterais são mentiras. Sua função promete fazer uma coisa, mas também faz outras coisas ocultas . Às vezes, ele fará alterações inesperadas nas variáveis de sua própria classe.

Às vezes, ele os fará para os parâmetros passados para a função ou para o sistema global. Em ambos os casos, são inverdades tortuosas e prejudiciais que muitas vezes resultam em estranhos acoplamentos temporais e dependências de ordem.

Considere, por exemplo, a função aparentemente inócuia da Listagem 3-6. Esta função usa um algoritmo padrão para corresponder um nome de usuário a uma senha. Ele retorna verdadeiro se eles corresponderem e falso se algo der errado. Mas também tem um efeito colateral. Você consegue identificar?

Listagem 3-6

UserValidator.java public class

```
UserValidator { private Cryptographer
cryptographer;

public boolean checkPassword(String userName, String password) {
    Usuário usuário = UserGateway.findByName(userName); if
(usuário != Usuário.NULL) {
        String codedPhrase = user.getPhraseEncodedByPassword(); Frase de string
= cryptographer.decrypt(codedPhrase, senha); if ("Senha válida".equals(frase)) {

            Session.initialize();
            retornar verdadeiro;
        }

    } retorna falso;
}
}
```

O efeito colateral é a chamada para `Session.initialize()`, é claro. A função `checkPassword` , por seu nome, diz que verifica a senha. O nome não implica que ele inicializa a sessão. Portanto, um chamador que acredita no que diz o nome da função corre o risco de apagar os dados da sessão existente quando decidir verificar a validade do

do utilizador.

Esse efeito colateral cria um acoplamento temporal. Ou seja, `checkPassword` só pode ser chamado em determinados momentos (ou seja, quando é seguro inicializar a sessão). Se for chamado fora de ordem, os dados da sessão podem ser perdidos inadvertidamente. Os acoplamentos temporais são confusos, especialmente quando ocultos como efeito colateral. Se você deve ter um acoplamento temporal, deve deixar claro no nome da função. Nesse caso, podemos renomear a função `checkPasswordAndInitializeSession`, embora isso certamente viole "Faça uma coisa".

Argumentos de saída

Os argumentos são mais naturalmente interpretados como *entradas* para uma função. Se você tem programado por mais de alguns anos, tenho certeza de que pensou duas vezes em um argumento que era na verdade uma *saída* em vez de uma entrada. Por exemplo:

anexar Rodapé(s);

Esta função acrescenta s como rodapé a alguma coisa? Ou acrescenta algum rodapé a s? É uma entrada ou uma saída? Não demora muito para olhar a assinatura da função e ver:

```
public void appendFooter(relatório StringBuffer)
```

Isso esclarece o problema, mas apenas à custa de verificar a declaração da função.

Qualquer coisa que o force a verificar a assinatura da função é equivalente a uma segunda olhada. É uma quebra cognitiva e deve ser evitada.

Nos dias anteriores à programação orientada a objetos, às vezes era necessário ter argumentos de saída. No entanto, grande parte da necessidade de argumentos de saída desaparece em linguagens OO, porque isso se *destina* a atuar como um argumento de saída. Em outras palavras, seria melhor que `appendFooter` fosse invocado como

```
report.appendFooter();
```

Em geral, argumentos de saída devem ser evitados. Se sua função deve mudar o estado de algo, faça com que mude o estado de seu próprio objeto.

Separação de consulta de comando

As funções devem fazer algo ou responder a algo, mas não ambos. Sua função deve alterar o estado de um objeto ou deve retornar algumas informações sobre esse objeto. Fazer as duas coisas geralmente leva à confusão. Considere, por exemplo, a seguinte função:

```
public boolean set(String attribute, String value);
```

Esta função define o valor de um atributo nomeado e retorna verdadeiro se for bem-sucedido e falso se tal atributo não existir. Isso leva a declarações estranhas como esta:

```
if (set("nome de usuário", "unclebob"))...
```

Imagine isso do ponto de vista do leitor. O que isso significa? Está perguntando se o atributo "username" foi definido anteriormente como "unclebob"? Ou está perguntando se o atributo "username" foi definido com sucesso como "unclebob"? É difícil inferir o significado da chamada porque não está claro se a palavra "conjunto" é um verbo ou um adjetivo.

O autor pretendia definir como um verbo, mas no contexto da instrução `if` parece um adjetivo. Portanto, a declaração é lida como "Se o atributo de nome de usuário foi definido anteriormente como tiobob" e não "defina o atributo de nome de usuário como tiobob e se isso funcionou, então. . ." Nós

poderia tentar resolver isso renomeando a função set para setAndCheckIfExists, mas isso não ajuda muito na legibilidade da instrução if . A solução real é separar o comando da consulta para que a ambigüidade não ocorra.

```
if (attributeExists("nome de usuário")) {
    setAttribute("nome de usuário", "unclebob");
    ...
}
```

Prefira Exceções a Retornar Códigos de Erro

Retornar códigos de erro de funções de comando é uma violação sutil da separação de consulta de comando. Ele promove comandos sendo usados como expressões nos predicados de declarações if .

```
if (deletePage(page) == E_OK)
```

Isso não sofre de confusão verbo/adjetivo, mas leva a estruturas profundamente aninhadas. Ao retornar um código de erro, você cria o problema de que o chamador deve lidar com o erro imediatamente.

```
if (deletePage(page) == E_OK) { if
    (registry.deleteReference(page.name) == E_OK) {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
            logger.log("página apagada");
        }
        else
            { logger.log("configKey não excluído");
        }
    } } else
        { logger.log("deleteReference do registro falhou");
    } } else
        { logger.log("excluir falhou"); return
    E_ERR;
}
```

Por outro lado, se você usar exceções em vez de códigos de erro retornados, o erro o código de processamento pode ser separado do código do caminho feliz e pode ser simplificado:

```
tente
{ deletePage(page);
registro.deleteReference(page.name);
configKeys.deleteKey(page.name.makeKey());

} catch (Exception e)
{ logger.log(e.getMessage());
}
```

Extrair blocos Try/Catch

Os blocos try/catch são feios por si só. Eles confundem a estrutura do código e misturam o processamento de erros com o processamento normal. Portanto, é melhor extraí os corpos dos blocos try e catch em funções próprias.

Prefira Exceções a Retornar Códigos de Erro

```
public void delete(página da página) { try
    { deletePageAndAllReferences(page);
    } catch (Exception e) { logError(e);
    }
}

private void deletePageAndAllReferences(Página da página) lança exceção {
    deletarPágina(página);
    registro.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e)
{
    logger.log(e.getMessage());
}
```

Acima, a função de exclusão é sobre o processamento de erros. É fácil entender e depois ignorar. A função `deletePageAndAllReferences` trata dos processos de exclusão total de uma página. O tratamento de erros pode ser ignorado. Isso fornece uma boa separação que torna o código mais fácil de entender e modificar.

Tratamento de erros é uma coisa

As funções devem fazer uma coisa. A manipulação de erros é uma coisa. Assim, uma função que lida com erros não deve fazer mais nada. Isso implica (como no exemplo acima) que se a palavra-chave `try` existe em uma função, ela deve ser a primeira palavra da função e não deve haver nada após os blocos `catch/finally`.

O ímã de dependência Error.java

Retornar códigos de erro geralmente implica que existe alguma classe ou enum na qual todos os códigos de erro são definidos.

```
Erro de enumeração pública {
    OK,
    INVÁLIDO,
    NÃO DESTA MANEIRA,
    BLOQUEADO,
    OUT_OF_RESOURCES,
    AGUARDANDO_PARA_EVENTO;
}
```

Classes como essa são um *ímã de dependência*; muitas outras classes devem importá-los e usá-los. Assim, quando a enumeração `Error` é alterada, todas as outras classes precisam ser recompiladas e reimplementadas.¹¹ Isso coloca uma pressão negativa na classe `Error`. Os programadores não querem

¹¹. Aqueles que achavam que poderiam escapar sem recompilar e redistribuir foram encontrados — e tratados.

para adicionar novos erros porque eles têm que reconstruir e reimplantar tudo. Assim, eles reutilizam códigos de erro antigos em vez de adicionar novos.

Quando você usa exceções em vez de códigos de erro, novas exceções são *derivadas* da classe de exceção. Eles podem ser adicionados sem forçar nenhuma recompilação ou reimplantação.¹²

Não se repita¹³

Examine cuidadosamente a Listagem 3-1 e você notará que há um algoritmo que é repetido quatro vezes, uma vez para cada um dos casos SetUp, SuiteSetUp, TearDown e SuiteTearDown . Não é fácil identificar essa duplicação porque as quatro instâncias estão misturadas com outro código e não são duplicadas uniformemente. Ainda assim, a duplicação é um problema porque incha o código e exigirá modificações em quatro vezes caso o algoritmo precise mudar. É também uma oportunidade quádrupla para um erro de omissão.



Essa duplicação foi corrigida pelo método `include` na Listagem 3-7. Leia esse código novamente e observe como a legibilidade de todo o módulo é aprimorada pela redução dessa duplicação.

A duplicação pode ser a raiz de todos os males do software. Muitos princípios e práticas foram criados com o objetivo de controlá-lo ou eliminá-lo. Considere, por exemplo, que todas as formas normais do banco de dados de Codd servem para eliminar a duplicação de dados. Considere também como a programação orientada a objetos serve para concentrar o código em classes básicas que, de outra forma, seriam redundantes. Programação Estruturada, Programação Orientada a Aspectos, Programação Orientada a Componentes, são todas, em parte, estratégias para eliminar a duplicação. Parece que, desde a invenção da sub-rotina, as inovações no desenvolvimento de software têm sido uma tentativa contínua de eliminar a duplicação de nosso código-fonte.

Programação Estruturada

Alguns programadores seguem as regras de programação estruturada de Edsger Dijkstra.¹⁴ Dijkstra disse que toda função, e todo bloco dentro de uma função, deveria ter uma entrada e uma saída. Seguir essas regras significa que deve haver apenas uma instrução `return` em uma função, nenhuma instrução `break` ou `continue` em um loop e nunca, *jamais*, qualquer instrução `goto`.

12. Este é um exemplo do Princípio Aberto e Fechado (OCP) [PPP02].

13. O princípio DRY. [PRAG].

14. [SP72].

Embora simpatizemos com os objetivos e disciplinas da programação estruturada, essas regras trazem poucos benefícios quando as funções são muito pequenas. É apenas em funções maiores que tais regras fornecem benefícios significativos.

Portanto, se você mantiver suas funções pequenas, as instruções múltiplas ocasionais de retorno, quebra ou continuação não causam danos e às vezes podem até ser mais expressivas do que a regra de entrada única e saída única. Por outro lado, goto só faz sentido em funções grandes, portanto deve ser evitado.

Como você escreve funções assim?

Escrever software é como qualquer outro tipo de escrita. Quando você escreve um artigo ou um artigo, você primeiro coloca seus pensamentos no papel, depois massageia até que fique bem lido. O primeiro rascunho pode ser desajeitado e desorganizado, então você o redige, o reestrutura e o refina até que fique do jeito que você quer.

Quando escrevo funções, elas ficam longas e complicadas. Eles têm muitos recuos e loops aninhados. Eles têm longas listas de argumentos. Os nomes são arbitrários e há código duplicado. Mas também tenho um conjunto de testes de unidade que abrangem cada uma dessas linhas de código desajeitadas.

Então eu massageio e refino esse código, separando funções, alterando nomes, eliminando a duplicação. Eu reduzo os métodos e os reordeno. Às vezes, dou aulas inteiras, enquanto mantenho os testes aprovados.

No final, termino com funções que seguem as regras que estabeleci neste capítulo. Eu não os escrevo dessa maneira para começar. Acho que ninguém poderia.

Conclusão

Todo sistema é construído a partir de uma linguagem específica de domínio projetada pelos programadores para descrever esse sistema. As funções são os verbos dessa linguagem e as classes são os substantivos. Isso não é um retrocesso à velha e hedionda noção de que os substantivos e verbos em um documento de requisitos são o primeiro palpite das classes e funções de um sistema. Em vez disso, esta é uma verdade muito mais antiga. A arte da programação é, e sempre foi, a arte do design de linguagem.

Os programadores mestres pensam nos sistemas como histórias a serem contadas, em vez de programas a serem escritos. Eles usam as facilidades de sua linguagem de programação escolhida para construir uma linguagem muito mais rica e expressiva que pode ser usada para contar essa história. Parte dessa linguagem específica de domínio é a hierarquia de funções que descrevem todas as ações que ocorrem nesse sistema. Em um ato engenhoso de recursão, essas ações são escritas para usar a própria linguagem específica do domínio que definem para contar sua própria pequena parte da história.

Este capítulo tratou bem da mecânica das funções de escrita. Se você seguir as regras aqui contidas, suas funções serão curtas, bem nomeadas e bem organizadas. Mas

nunca esqueça que seu objetivo real é contar a história do sistema e que as funções que você escreve precisam se encaixar perfeitamente em uma linguagem clara e precisa para ajudá-lo nessa narrativa.

SetupTeardownIncluder

Listagem

3-7 SetupTeardownIncluder.java

```
package fitness.html;

import fitness.responders.run.SuiteResponder; importar
fitness.wiki..*;

public class SetupTeardownIncluder { private
    PageData pageData; privado
    booleano isSuite; WikiPage
    privada testPage; stringBuffer
    privado newPageContent; privado PageCrawler
    pageCrawler;

    public static String render(PageData pageData) throws Exception { return render(pageData,
        false);
    }

    public static String render(PageData pageData, booleano isSuite) throws Exception
    { return new
        SetupTeardownIncluder(pageData).render(isSuite);
    }

    private SetupTeardownIncluder(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage();
        pageCrawler = testPage.getPageCrawler();
        newPageContent = new StringBuffer();
    }

    private String render(booleano isSuite) lança Exception { this.isSuite = isSuite; if
        (isTestPage())

        includeSetupAndTeardownPages(); return
        pageData.getHtml();
    }

    private booleano isTestPage() lança Exception { return
        pageData.hasAttribute("Test");
    }

    private void includeSetupAndTeardownPages() lança exceção {
        includeSetupPages();
        includePageContent();
        includeTeardownPages();
        atualizarPageContent();
    }
}
```

Listagem 3-7 (continuação)**SetupTeardownIncluder.java** private void

```

includeSetupPages() throws Exception { if (isSuite)

    includeSuiteSetupPage();
    includeSetupPage();
}

private void includeSuiteSetupPage() lança exceção
{ include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage() gera exceção {
    include("SetUp", "-setup");
}

private void includePageContent() lança exceção
{ newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() lança exceção { includeTeardownPage();
if (isSuite)

    includeSuiteTeardownPage();
}

private void includeTeardownPage() lança exceção { include("TearDown",
"-teardown");
}

private void includeSuiteTeardownPage() lança exceção
{ include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent() gera exceção
{ pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) lança Exception {
    WikiPage heredPage = findInheritedPage(pageName); if (herdadaPágina!
= nulo) {
        String pagePathName = getPathNameForPage(heredPage);
        buildIncludeDirective(pagePathName, arg); }

}

WikiPage privada findInheritedPage(String pageName) lança exceção { return
    PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

string privada getPathNameForPage(página WikiPage) lança exceção {
    WikiPagePath pagePath = pageCrawler.getFullPath(page); return
    PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {

    newPageContent.append("\n\ninclude "
}

```

Listagem 3-7 (continuação)

```
SetupTeardownIncluder.java .append(arg) .append(" .") .append(pagePathName) .append("\n");  
    }  
}
```

Bibliografia

[KP78]: Kernighan e Plaugher, *Os Elementos do Estilo de Programação*, 2d. ed., McGraw Hill, 1978.

[PPP02]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: *Programação Estruturada*, O.-J. Dahl, EW Dijkstra, CAR Hoare, Academic Press, Londres, 1972.

4

Comentários



“Não comente códigos ruins – reescreva-os.”

—Brian W. Kernighan e PJ Plaugher¹

Nada pode ser tão útil quanto um comentário bem colocado. Nada pode atrapalhar mais um módulo do que comentários dogmáticos frívolos. Nada pode ser tão prejudicial quanto um velho comentário grosseiro que propaga mentiras e desinformação.

Comentários não são como a Lista de Schindler. Eles não são “puro bem”. De fato, os comentários são, na melhor das hipóteses, um mal necessário. Se nossas linguagens de programação fossem expressivas o suficiente, ou se

1. [KP78], p. 144.

tivéssemos o talento para manejar sutilmente essas linguagens para expressar nossa intenção, não precisaríamos muito de comentários - talvez nem um pouco.

O uso adequado de comentários é para compensar nossa falha em nos expressarmos no código. Observe que usei a palavra *fracasso*. Eu quis dizer isso. Comentários são sempre falhas. Devemos tê-los porque nem sempre podemos descobrir como nos expressar sem eles, mas seu uso não é motivo de comemoração.

Portanto, quando você se encontrar em uma posição em que precisa escrever um comentário, pense bem e veja se não há alguma maneira de virar a mesa e se expressar em código. Toda vez que você se expressar em código, você deve dar um tapinha nas costas. Cada vez que você escreve um comentário, deve fazer uma careta e sentir o fracasso de sua capacidade de expressão.

Por que estou tão desanimado com os comentários? Porque eles mentem. Nem sempre, e não intencionalmente, mas com muita frequência. Quanto mais antigo for um comentário e quanto mais distante estiver do código que descreve, maior a probabilidade de estar simplesmente errado. A razão é simples. Os programadores não podem mantê-los de forma realista.

O código muda e evolui. Pedaços dele se movem daqui para lá. Esses pedaços se bifurcam e se reproduzem e se juntam novamente para formar quimeras. Infelizmente, os comentários nem sempre os seguem - nem sempre podem acompanhá-los. E com muita frequência os comentários são separados do código que descrevem e se tornam sinopses órfãos de precisão cada vez menor. Por exemplo, veja o que aconteceu com este comentário e a linha que ele pretendia descrever:

```
solicitação MockRequest;
string final privada HTTP_DATE_REGEX = 
    "[SMTWF]{2}\\,\\s[0-9]{2}\\s[JFMASOND]{2}\\s"+ "[0-9]{4} \\s[0-9]{2}\\
    \\:[0-9]{2}\\:[0-9]{2}\\sGMT"; resposta de resposta privada; contexto
FitNesseContext privado;
respondente FileResponder privado; local
privado saveLocale;
```

// Exemplo: "Tue, 02 de abril de 2003 22:18:49 GMT"

Outras variáveis de instância que provavelmente foram adicionadas posteriormente foram interpostas entre a constante HTTP_DATE_REGEX e seu comentário explicativo.

É possível afirmar que os programadores devem ser disciplinados o suficiente para manter os comentários em alto estado de conservação, relevância e precisão. Eu concordo, eles deveriam.

Mas prefiro que essa energia seja direcionada para tornar o código tão claro e expressivo que não precise de comentários em primeiro lugar.

Comentários imprecisos são muito piores do que nenhum comentário. Eles iludem e enganam. Eles estabelecem expectativas que nunca serão cumpridas. Eles estabelecem regras antigas que não precisam, ou não devem mais ser seguidas.

A verdade só pode ser encontrada em um lugar: o código. Somente o código pode realmente dizer o que ele faz. É a única fonte de informações verdadeiramente precisas. Portanto, embora os comentários às vezes sejam necessários, gastaremos muita energia para minimizá-los.

Comentários não compensam código ruim

Uma das motivações mais comuns para escrever comentários é código ruim. Escrevemos um módulo e sabemos que é confuso e desorganizado. Sabemos que é uma confusão. Então dizemos a nós mesmos: "Ooh, é melhor eu comentar isso!" Não! É melhor limpá-lo!

Um código claro e expressivo com poucos comentários é muito superior a um código confuso e complexo com muitos comentários. Em vez de gastar seu tempo escrevendo os comentários que explicam a bagunça que você fez, gaste-o limpando essa bagunça.

Explique-se em código

Certamente há momentos em que o código é um veículo ruim para explicação. Infelizmente, muitos programadores entendem que isso significa que o código raramente, ou nunca, é um bom meio de explicação. Isso é patentemente falso. Qual você prefere ver? Esse:

```
// Verifique se o funcionário é elegível para benefícios completos if ((employee.flags
& HOURLY_FLAG) && (employee.age > 65))
```

Ou isto?

```
if (employee.isEligibleForFullBenefits())
```

Leva apenas alguns segundos de reflexão para explicar a maior parte de sua intenção no código. Em muitos casos, basta criar uma função que diga a mesma coisa que o comentário que você deseja escrever.

Bons comentários

Alguns comentários são necessários ou benéficos. Veremos alguns que considero dignos dos bits que consomem. Tenha em mente, no entanto, que o único comentário realmente bom é aquele que você encontrou uma maneira de não escrever.

Comentários legais

Às vezes, nossos padrões de codificação corporativos nos obrigam a escrever certos comentários por motivos legais. Por exemplo, declarações de direitos autorais e autoria são coisas necessárias e razoáveis para colocar em um comentário no início de cada arquivo de origem.

Aqui, por exemplo, está o cabeçalho de comentário padrão que colocamos no início de cada arquivo de origem no FitNesse. Fico feliz em dizer que nosso IDE impede que esse comentário aja como desordem ao recolhê-lo automaticamente.

```
// Copyright (C) 2003,2004,2005 da Object Mentor, Inc. Todos os direitos reservados.
// Lançado sob os termos da GNU General Public License versão 2 ou posterior.
```

Comentários como este não devem ser contratos ou termos legais. Sempre que possível, consulte uma licença padrão ou outro documento externo em vez de colocar todos os termos e condições no comentário.

Comentários informativos

Às vezes é útil fornecer informações básicas com um comentário. Por exemplo, considere este comentário que explica o valor de retorno de um método abstrato:

```
// Retorna uma instância do Responder sendo testado. Protected
abstract Responder responderInstance();
```

Às vezes, um comentário como esse pode ser útil, mas é melhor usar o nome da função para transmitir as informações sempre que possível. Por exemplo, neste caso, o comentário pode se tornar redundante renomeando a função: responderBeingTested.

Aqui está um caso que é um pouco melhor:

```
// formato correspondente kk:mm:ss EEE, MMM dd, aaaa
Padrão timeMatcher = Pattern.compile(
    "\\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Nesse caso, o comentário nos informa que a expressão regular destina-se a corresponder a uma hora e data que foram formatadas com a função SimpleDateFormat.format usando a string de formato especificada. Ainda assim, poderia ter sido melhor e mais claro se esse código tivesse sido movido para uma classe especial que convertesse os formatos de datas e horas. Então o comentário provavelmente teria sido supérfluo.

Explicação da Intenção

Às vezes, um comentário vai além de apenas informações úteis sobre a implementação e fornece a intenção por trás de uma decisão. No caso a seguir, vemos uma decisão interessante documentada por um comentário. Ao comparar dois objetos, o autor decidiu que queria classificar os objetos de sua classe acima dos objetos de qualquer outra.

```
public int compareTo(Objeto o) {
    if(o exemplo de WikiPagePath) {
        WikiPagePath p = (WikiPagePath) o; String
        NomeComprimido = StringUtil.join(nomes, ""); String
        compressedArgumentName = StringUtil.join(p.names, ""); return
        compressedName.compareTo(compressedArgumentName);

    } retorna 1; // somos maiores porque somos do tipo certo.
}
```

Aqui está um exemplo ainda melhor. Você pode não concordar com a solução do programador para o problema, mas pelo menos você sabe o que ele estava tentando fazer.

```
public void testConcurrentAddWidgets() lança exceção {
    WidgetBuilder widgetBuilder = new
    WidgetBuilder(new Class[]{BoldWidget.class});
```

```

String texto = """texto em negrito""";
ParentWidget pai = new
    BoldWidget(new MockWidgetRoot(), """texto em negrito"""); AtomicBoolean
failFlag = new AtomicBoolean(); failFlag.set(false);

//Esta é a nossa melhor tentativa de obter uma condição de
corrida //criando um grande número de
threads. for (int i = 0; i < 25000; i++)
{
    WidgetBuilderThread widgetBuilderThread =
        novo WidgetBuilderThread(widgetBuilder, texto, pai, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
}

} assertEquals(false, failFlag.get());
}

```

Esclarecimento

Às vezes, é útil traduzir o significado de algum argumento obscuro ou valor de retorno em algo legível. Em geral, é melhor encontrar uma maneira de tornar esse argumento ou valor de retorno claro por si só; mas quando faz parte da biblioteca padrão ou no código que você não pode alterar, um comentário esclarecedor útil pode ser útil.

```

public void testCompareTo() lança exceção {

    WikiPagePath a = PathParser.parse("PáginaA");
    WikiPagePath ab = PathParser.parse("PáginaA.PáginaB");
    WikiPagePath b = PathParser.parse("PáginaB");
    WikiPagePath aa = PathParser.parse("PáginaA.PáginaA");
    WikiPagePath bb = PathParser.parse("PáginaB.PáginaB");
    WikiPagePath ba = PathParser.parse("PáginaB.PáginaA");

    assertTrue(a.compareTo(a) == 0); // a == a
    assertTrue(a.compareTo(b) != 0); // a != b
    assertTrue(ab.compareTo(ab) == 0); // ab == ab
    assertTrue(a.compareTo(b) == -1); // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1); // b > a
    assertTrue(ab.compareTo(aa) == 1); // ab > aa
    assertTrue(bb.compareTo(ba) == 1); // bb > ba
}

```

Existe um risco substancial, é claro, de que um comentário esclarecedor esteja incorreto. Percorra o exemplo anterior e veja como é difícil verificar se eles estão corretos. Isso explica por que o esclarecimento é necessário e por que é arriscado. Portanto, antes de escrever comentários como este, certifique-se de que não há maneira melhor e, em seguida, cuide ainda mais para que sejam precisos.

Aviso de Consequências

Às vezes é útil alertar outros programadores sobre certas consequências. Por exemplo, aqui está um comentário que explica por que um determinado caso de teste está desativado:

```
// Não corra a menos que //
tenha algum tempo para matar.
public void _testWithReallyBigFile() {
    writeLinesToFile(10000000);
    resposta.setBody(testFile);
    resposta.readyToSend(this); String
    responseString = output.toString(); assertSubString("Content-
Length: 1000000000", responseString); assertTrue(bytesSent > 1000000000);
}
```



Hoje em dia, é claro, desligaríamos o caso de teste usando o atributo `@Ignore` com uma string explicativa apropriada. `@Ignore("Demora muito para rodar")`. Mas nos dias anteriores ao JUnit 4, colocar um sublinhado na frente do nome do método era uma convenção comum. O comentário, embora irreverente, mostra muito bem o ponto.

Aqui está outro exemplo mais comovente:

```
public static SimpleDateFormat makeStandardHttpDateFormat() {
    //SimpleDateFormat não é thread-safe, //portanto,
    //precisamos criar cada instância independentemente.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM aaaa HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT")); retorno df;
}
```

Você pode reclamar que existem maneiras melhores de resolver esse problema. Eu posso concordar com você. Mas o comentário, conforme apresentado aqui, é perfeitamente razoável. Isso impedirá que algum programador ansioso demais use um inicializador estático em nome da eficiência.

TODO Comments

Às vezes, é razoável deixar notas de “coisas a fazer” na forma de comentários `//TODO` . No caso a seguir, o comentário `TODO` explica por que a função tem uma implementação degenerada e qual deve ser o futuro dessa função.

```
//TODO-MdM estes não são necessários
// Esperamos que isso desapareça quando fizermos o modelo de checkout
protected VersionInfo makeVersion() throws Exception {
    retornar nulo;
}
```

TODOs são trabalhos que o programador acha que devem ser feitos, mas por algum motivo não podem fazer no momento. Pode ser um lembrete para excluir um recurso obsoleto ou um apelo para que outra pessoa analise um problema. Pode ser um pedido para que outra pessoa pense em um nome melhor ou um lembrete para fazer uma alteração que depende de um evento planejado. O que quer que um TODO possa ser, *não é uma desculpa para deixar um código ruim no sistema.*

Hoje em dia, a maioria dos bons IDEs fornece gestos e recursos especiais para localizar todos os comentários TODO , portanto, não é provável que eles se percam. Ainda assim, você não quer que seu código seja repleto de TODOs. Portanto, examine-os regularmente e elimine aqueles que você pode.

Amplificação

Um comentário pode ser usado para ampliar a importância de algo que, de outra forma, pode parecer irrelevante.

```
String listItemContent = match.group(3).trim(); // o trim é  
muito importante. Ele remove os // espaços iniciais que podem  
fazer com que o item seja reconhecido // como outra lista.  
new ListItemWidget(this,  
listItemContent, this.level + 1); return buildList(text.substring(match.end()));
```

Javadoc em APIs públicas

Não há nada tão útil e satisfatório quanto uma API pública bem descrita. Os documentos java para a biblioteca Java padrão são um exemplo. Seria difícil, na melhor das hipóteses, escrever programas Java sem eles.

Se você estiver escrevendo uma API pública, certamente deve escrever bons javadocs para ela. Mas tenha em mente o restante dos conselhos deste capítulo. Javadoc podem ser tão enganosos, não locais e desonestos quanto qualquer outro tipo de comentário.

Comentários ruins

A maioria dos comentários se enquadra nessa categoria. Geralmente são muletas ou desculpas para códigos ruins ou justificativas para decisões insuficientes, chegando a pouco mais do que o programador falando sozinho.

resmungando

Colocar um comentário apenas porque você acha que deveria ou porque o processo exige isso é um truque. Se você decidir escrever um comentário, gaste o tempo necessário para garantir que seja o melhor comentário que você pode escrever.

Aqui, por exemplo, está um caso que encontrei no FitNesse, onde um comentário poderia de fato ter sido útil. Mas o autor estava com pressa ou simplesmente não prestava muita atenção. Sua murmurção deixou para trás um enigma:

```
public void loadProperties() {
    try {
        String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE; FileInputStream
        propertiesStream = new FileInputStream(propertiesPath); carregadoProperties.load(propertiesStream); }
        catch(IOException e) {

            // Nenhum arquivo de propriedades significa que todos os padrões são carregados
        }
    }
```

O que esse comentário no bloco catch significa? Claramente significava algo para o autor, mas o significado não é tão bom assim. Aparentemente, se obtivermos uma IOException, significa que não havia arquivo de propriedades; e nesse caso todos os padrões são carregados. Mas quem carrega todos os padrões? Eles foram carregados antes da chamada para loadProperties.load? Ou loadProperties.load capturou a exceção, carregou os padrões e passou a exceção para nós ignorarmos? Ou loadProperties.load carregou todos os padrões antes de tentar carregar o arquivo? O autor estava tentando se consolar com o fato de estar deixando o bloco catch vazio? Ou — e essa é a possibilidade assustadora — o autor estava tentando dizer a si mesmo para voltar aqui mais tarde e escrever o código que carregaria os padrões?

Nosso único recurso é examinar o código em outras partes do sistema para descobrir o que está acontecendo. Qualquer comentário que o force a procurar em outro módulo o significado desse comentário falhou em se comunicar com você e não vale os bits que consome.

Comentários redundantes

A Listagem 4-1 mostra uma função simples com um comentário de cabeçalho completamente redundante. O comentário provavelmente leva mais tempo para ser lido do que o próprio código.

Listagem 4-1 waitForClose
<pre>// Método utilitário que retorna quando this.closed é verdadeiro. Lança uma exceção // se o tempo // limite for atingido. público void sincronizado waitForClose(final longo timeoutMillis) lança exceção { if(!fechado) { wait(timeoutMillis); if(! closed) throw new Exception("MockResponseSender não pôde ser fechado"); } }</pre>

A que propósito serve este comentário? Certamente não é mais informativo do que o código. Não justifica o código, nem fornece intenção ou justificativa. Não é mais fácil de ler do que o código. Na verdade, é menos preciso do que o código e induz o leitor a aceitar essa falta de precisão em vez de um verdadeiro entendimento. É como um vendedor de carros usados alegre garantindo que você não precisa olhar sob o capô.

Agora considere a legião de javadocs inúteis e redundantes na Listagem 4-2 retirados do Tomcat. Esses comentários servem apenas para confundir e obscurecer o código. Eles não servem a nenhum propósito documental. Para piorar as coisas, mostrei apenas os primeiros. Há muitos mais neste módulo.

Listagem**4-2 ContainerBase.java (Tomcat) classe**

```
abstrata pública ContainerBase
    implementa Container, Ciclo de Vida, Pipeline,
    MBeanRegistration, Serializável {

    /**
     * O atraso do processador para este componente.
     */
    protegido int backgroundProcessorDelay = -1;

    /**
     * O suporte de evento de ciclo de vida para este componente.
     */
    protegido LifecycleSupport lifecycle = new
        LifecycleSupport(this);

    /**
     * Os ouvintes de eventos do contêiner para este contêiner. */
    listeners ArrayList protegidos = new ArrayList();

    /**
     * A implementação do Loader com a qual este Contêiner está *
     * associado. */

    Protected Loader loader = null;

    /**
     * A implementação do Logger com a qual este Contêiner está *
     * associado. */
    logger protegido = nulo;

    /**
     * Nome do criador de logs
     * associado. */ protegido String logName = null;
```

Listagem 4-2 (continuação)**ContainerBase.java (Tomcat)**

```
/**  
 * A implementação do Manager com a qual este Container está * associado.  
 */ gerenciador  
  
 protegido gerenciador = nulo;  
  
/**  
 * O cluster ao qual este Container está associado. */ cluster cluster  
  
 protegido = nulo;  
  
/**  
 * O nome legível deste Container. */ nome da string  
  
 protegida = nulo;  
  
/**  
 * O Container pai do qual este Container é filho. */ pai do contêiner protegido  
 =  
 nulo;  
  
/**  
 * O carregador de classes pai a ser configurado quando instalamos um *  
 Loader. */  
  
 ClassLoader protegido parentClassLoader = nulo;  
  
/**  
 * O objeto Pipeline ao qual este Container está * associado. */  
 pipeline protegido  
  
 pipeline = new StandardPipeline(this);  
  
/**  
 * O Domínio ao qual este Contêiner está associado. */ domínio  
  
 protegido domínio = nulo;  
  
/**  
 * O objeto DirContext de recursos ao qual este Container * está associado. */  
 recursos protegidos  
  
 do DirContext = nulo;
```

Comentários enganosos

Às vezes, com as melhores intenções, um programador faz uma declaração em seus comentários que não é precisa o suficiente para ser exata. Considere por outro momento o comentário bastante redundante, mas também sutilmente enganoso, que vimos na Listagem 4-1.

Você descobriu como o comentário foi enganoso? O método não retorna *when this.closed* se torna verdadeiro. Ele retorna *se this.closed* for verdadeiro; caso contrário, ele espera por um tempo limite cego e lança uma exceção se *this.closed* ainda não for verdadeiro.

Essa sutil desinformação, expressa em um comentário que é mais difícil de ler do que o corpo do código, pode fazer com que outro programador chame alegremente essa função na expectativa de que ela retornará assim que *this.closed* se tornar verdadeiro. Esse pobre programador se encontraria em uma sessão de depuração tentando descobrir por que seu código foi executado tão lentamente.

Comentários obrigatórios

É simplesmente bobo ter uma regra que diga que toda função deve ter um javadoc ou toda variável deve ter um comentário. Comentários como esse apenas bagunçam o código, propagam mentiras e levam à confusão e desorganização geral.

Por exemplo, javadocs necessários para cada função levam a abominações como a Listagem 4-3. Essa confusão não acrescenta nada e serve apenas para ofuscar o código e criar potencial para mentiras e desorientação.

Listagem
<pre>4-3 /** * * @param title O título do CD * @param author O autor do CD * @param tracks O número de faixas no CD * @param durationInMinutes A duração do CD em minutos */ public void addCD(String title, String autor, int faixas, int duraçãoEmMinutos) { CD cd = new CD(); cd.title = título; cd.autor = autor; cd.tracks = faixas; cd.duration = duração; cdList.add(cd); }</pre>

Comentários do diário

Às vezes, as pessoas adicionam um comentário ao início de um módulo toda vez que o editam. Esses comentários se acumulam como uma espécie de diário, ou log, de todas as alterações que já foram feitas. Eu vi alguns módulos com dezenas de páginas dessas entradas de diário.

```
* Alterações (a partir de 11-Out-2001)
* -----
* 11-Out-2001 : Reorganizou a classe e mudou-a para o novo pacote com.jrefinery.date
(DG);
* 05-Nov-2001 : Adicionado um método getDescription() e eliminada a classe NotableDate (DG);
* 12-Nov-2001: IBD requer o método setDescription(), agora que a classe NotableDate se foi (DG);
    Alterado getPreviousDayOfWeek(), getFollowingDayOfWeek() e
    getNearestDayOfWeek() para corrigir bugs (DG);
* 05-Dez-2001 : Corrigido bug na classe SpreadsheetDate (DG);
* 29 de maio de 2002: Moveu as constantes do mês para uma interface separada
(Constantes do Mês) (DG);
* 27-Ago-2002 : Corrigido bug no método addMonths(), graças a N???levka Petr (DG); * 03-Out-2002 :
Corrigidos os erros reportados pela Checkstyle (DG); * 13-Mar-2003 :
Implementado Serializável (DG); * 29-Mai-2003 : Corrigido bug
no método addMonths (DG); * 04-Set-2003 : Comparável
implementado. Atualizado o isInRange javadocs (DG); * 05-Jan-2005 : Corrigido bug no método
addYears() (1096282) (DG);
```

Há muito tempo, havia um bom motivo para criar e manter essas entradas de log no início de cada módulo. Não tínhamos sistemas de controle de código-fonte que fizessem isso por nós. Hoje em dia, no entanto, esses diários longos são apenas mais confusos para obfuscate o módulo. Eles devem ser completamente removidos.

Comentários de Ruído

Às vezes você vê comentários que não passam de ruído. Eles reafirmam o óbvio e não fornecem nenhuma informação nova.

```
/**
 * Construtor padrão. */

protectedAnualDateRule() { }
```

Não, sério? Ou que tal isso:

```
/** O dia do mês. */ private int
diaDoMês;
```

E então há este modelo de redundância:

```
/**
 * Retorna o dia do mês.
 * @return o dia do mês. */ public int
getDayOfMonth() { return dayOfMonth;
}
```

Esses comentários são tão barulhentos que aprendemos a ignorá-los. À medida que lemos o código, nossos olhos simplesmente passam por cima deles. Eventualmente, os comentários começam a mentir conforme o código em torno deles muda.

O primeiro comentário na Listagem 4-4 parece apropriado.² Ele explica por que o bloco catch está sendo ignorado. Mas o segundo comentário é puro ruído. Aparentemente, o programador estava tão frustrado em escrever blocos try/catch nessa função que precisava desabafar.

Listagem

4-4 startSending

```
private void iniciarEnviando() {

    tentar
    {
        doEnviando();

    } catch(SocketException e) {

        // normal. alguém interrompeu o pedido.

    } catch(Exceção e) {

        tentar
        {
            response.add(ErrorResponder.makeExceptionString(e)); resposta.closeAll();

        } catch(Exceção e1) {

            //Me dá um tempo!
        }
    }
}
```

Em vez de desabafar em um comentário inútil e ruidoso, o programador deveria ter reconhecido que sua frustração poderia ser resolvida melhorando a estrutura de seu código. Ele deveria ter direcionado sua energia para extrair o último bloco try/catch em uma função separada, conforme mostrado na Listagem 4-5.

Listagem

4-5 startSending (refatorado) private

```
void startSending() {

    tentar
    {
        doEnviando();
    }
}
```

2. A tendência atual dos IDEs de verificar a ortografia nos comentários será um bálsamo para aqueles de nós que leem muito código.

Listagem 4-5 (continuação)**startSending (refatorada)**

```

startSending (refatorada)

    catch(SocketException e) {

        // normal. alguém interrompeu o pedido.

        } catch(Exceção e) {

            addExceptionAndCloseResponse(e);
        }
    }

    private void addExceptionAndCloseResponse(Exception e) {

        tentar
        {
            response.add(ErrorResponder.makeExceptionString(e)); resposta.closeAll();

        } catch(Exceção e1) { }

    }
}

```

Substitua a tentação de criar ruído pela determinação de limpar seu código. Você descobrirá que isso o torna um programador melhor e mais feliz.

barulho assustador

Javadoc também podem ser barulhentos. Qual é a finalidade dos seguintes Javadoc (de uma conhecida biblioteca de código aberto)? Resposta: nada. Eles são apenas comentários ruidosos redundantes escritos a partir de algum desejo equivocado de fornecer documentação.

```

/** O nome. */ private
String nome;

/** A versão. */ versão
privada da String;

/** O nome da licença. */ private
String nome da licença;

/** A versão. */ informações
privadas da string;

```

Leia esses comentários novamente com mais atenção. Você vê o erro de recortar e colar? Se os autores não estão prestando atenção quando os comentários são escritos (ou colados), por que esperar que os leitores lucrem com eles?

Não use um comentário quando puder usar uma função ou uma variável

Considere o seguinte trecho de código:

```
// o módulo da lista global <mod> depende do // subsistema do qual fazemos
parte? if
(smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Isso poderia ser reformulado sem o comentário como

```
ArrayList moduleDependees = smodule.getDependSubsystems(); String
ourSubSystem = subSysMod.getSubSystem(); if
(moduleDependees.contains(ourSubSystem))
```

O autor do código original pode ter escrito primeiro o comentário (improvável) e depois escrito o código para preencher o comentário. No entanto, o autor deveria ter refatorado o código, como eu fiz, para que o comentário pudesse ser removido.

Marcadores de Posição

Às vezes, os programadores gostam de marcar uma posição específica em um arquivo de origem. Por exemplo, recentemente encontrei isso em um programa que estava procurando:

```
// Ações /////////////////////////////////
```

São raros os momentos em que faz sentido reunir certas funções sob um banner como este. Mas, em geral, eles são uma confusão que deve ser eliminada - especialmente o barulhento trem de cortes no final.

Pense desta maneira. Um banner é surpreendente e óbvio se você não vê banners com muita frequência. Portanto, use-os com moderação e somente quando o benefício for significativo. Se você usar banners demais, eles cairão no ruído de fundo e serão ignorados.

Comentários de chaves de fechamento

Às vezes, os programadores colocam comentários especiais nas chaves de fechamento, como na Listagem 4-6. Embora isso possa fazer sentido para funções longas com estruturas profundamente aninhadas, serve apenas para confundir o tipo de funções pequenas e encapsuladas que preferimos. Portanto, se você quiser marcar suas chaves de fechamento, tente encurtar suas funções.

Listagem 4-6**wc.java**

```
public class wc { public
static void main(String[] args) { BufferedReader in =
new BufferedReader(new InputStreamReader(System.in)); Linha de corda; int ContaLinhas = 0; int
charCont = 0;
int ContaPalavras = 0;
tentar {
```

Listagem 4-6 (continuação)**wc.java**

```

while ((line = in.readLine()) != null) { lineCount++;
    charCount +=
        linha.comprimento(); String palavras[]
        = line.split("\\W"); wordCount +=
        palavras.comprimento(); } //while

    System.out.println("wordCount = " + wordCount);
    System.out.println("lineCount = " + lineCount);
    System.out.println("charCount = " + charCount); // tenta
    pegar
    (IOException e) {
        System.err.println("Erro: " + e.getMessage()); } //pega } //principal

}

```

Atribuições e assinaturas

/* Adicionado por Rick */

Os sistemas de controle de código-fonte são muito bons em lembrar quem adicionou o quê e quando. Não há necessidade de poluir o código com pequenas assinaturas. Você pode pensar que tais comentários seriam úteis para ajudar outras pessoas a saber com quem falar sobre o código. Mas a realidade é que eles tendem a permanecer por anos e anos, tornando-se cada vez menos precisos e relevantes.

Novamente, o sistema de controle de código-fonte é um lugar melhor para esse tipo de informação.

Código Comentado

Poucas práticas são tão odiosas quanto comentar código. Não faça isso!

```

Resposta InputStreamResponse = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream(); // leitor
StreamReader = new StreamReader(resultsStream); //
response.setContent(reader.read(formatter.getByteCount()));

```

Outros que virem esse código comentado não terão coragem de excluí-lo. Eles vão pensar que está lá por um motivo e é importante demais para deletar. Portanto, o código comentado se acumula como resíduos no fundo de uma garrafa de vinho ruim.

Considere isso do Apache Commons:

```

this.bytePos = writeBytes(pngIdBytes, 0); //hdrPos =
bytePos;
escrevaCabeçalho();
escreverResolução(); //
dataPos = bytePos; if
(writelImageData()) { writeEnd();
this.pngBytes
= resizeByteArray(this.pngBytes, this.maxPos);
}

```

```

else
    { this.pngBytes = null;

} return this.pngBytes;

```

Por que essas duas linhas de código estão comentadas? Eles são importantes? Eles foram deixados como lembretes de alguma mudança iminente? Ou eles são apenas lixo que alguém comentou anos atrás e simplesmente não se preocupou em limpar.

Houve um tempo, nos anos 60, em que comentar o código poderia ser útil. Mas temos bons sistemas de controle de código-fonte há muito tempo. Esses sistemas lembrarão o código para nós. Não precisamos mais comentar. Basta deletar o código. Não vamos perdê-lo. Promessa.

Comentários HTML

O HTML nos comentários do código-fonte é uma abominação, como você pode perceber lendo o código abaixo. Isso torna os comentários difíceis de ler no único lugar onde deveriam ser fáceis de ler - o editor/IDE. Se os comentários forem extraídos por alguma ferramenta (como Javadoc) para aparecer em uma página da Web, então deve ser responsabilidade dessa ferramenta, e não do programador, adornar os comentários com o HTML apropriado.

```

/**
 * Tarefa para executar testes de ajuste.
 * Esta tarefa executa testes de aptidão e publica os resultados. * <p/>
 *
 * <pré>
 * Uso:
 * <taskdef name="execute-fitness-tests";
 *   classname="fitness.ant.ExecuteFitnessTestsTask";
 *   classpathref="classpath" />;
 * OU
 * <taskdef classpathref="classpath";
 *   resource="tasks.properties" />;
 * <p/>
 * <execute-fitness-tests
 *   suitepage="FitNesse.SuiteAcceptanceTests";
 *   fitnesseport="8082";
 *   resultsdir="${results.dir}";
 *   resultsthmlpage="fit-results.html";
 *   classpathref="classpath" /> * </pre> */

```

Informações não locais

Se você precisar escrever um comentário, certifique-se de que ele descreva o código próximo ao qual aparece. Não ofereça informações de todo o sistema no contexto de um comentário local. Considere, por exemplo, o comentário javadoc abaixo. Além do fato de ser terrivelmente redundante, também oferece informações sobre a porta padrão. E, no entanto, a função não tem absolutamente nenhum controle sobre qual é esse padrão. O comentário não está descrevendo a função, mas alguma outra parte muito distante do sistema. Claro que não há garantia de que este comentário será alterado quando o código que contém o padrão for alterado.

```
/*
 * Porta na qual fitnessse seria executado. O padrão é <b>8082</b>.
 * @param fitnesssePort */
public void setFitnesssePort(int fitnesssePort) {
    this.fitnesssePort = fitnesssePort;
}
```

Muita informação

Não coloque discussões históricas interessantes ou descrições irrelevantes de detalhes em seus comentários. O comentário abaixo foi extraído de um módulo projetado para testar se uma função pode codificar e decodificar base64. Além do número RFC, alguém lendo este código não precisa das informações misteriosas contidas no comentário.

```
/*
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
Parte Um: Formato dos Corpos de Mensagens da Internet
seção 6.8. Codificação de transferência de conteúdo Base64 O
processo de codificação representa grupos de bits de entrada de 24 bits como strings de saída
de 4 caracteres codificados. Da esquerda para a direita, um grupo de entrada de 24 bits
é formado pela concatenação de 3 grupos de entrada de 8 bits.
Esses 24 bits são então tratados como 4 grupos concatenados de 6 bits, cada um dos quais
é traduzido em um único dígito no alfabeto base64.
Ao codificar um fluxo de bits por meio da codificação base64, deve-se presumir que o fluxo
de bits seja ordenado com o bit mais significativo primeiro.
Ou seja, o primeiro bit no fluxo será o bit de ordem superior no primeiro byte de 8 bits, e o
oitavo bit será o bit de ordem inferior no primeiro byte de 8 bits e assim por diante. */
```

Conexão Inóvia

A conexão entre um comentário e o código que ele descreve deve ser óvia. Se você está se dando ao trabalho de escrever um comentário, pelo menos gostaria que o leitor pudesse ver o comentário e o código e entender do que o comentário está falando.

Considere, por exemplo, este comentário extraído do Apache Commons:

```
/*
 * começa com uma matriz que é grande o suficiente para conter todos os pixels * (mais
bytes de filtro) e 200 bytes extras para informações de cabeçalho */ this.pngBytes =
new byte[((this.width + 1) * this .altura * 3) + 200];
```

O que é um byte de filtro? Relaciona-se com o +1? Ou para o *3? Ambos? Um pixel é um byte? Por que 200? O objetivo de um comentário é explicar o código que não explica a si mesmo. É uma pena quando um comentário precisa de sua própria explicação.

Cabeçalhos de Função

Funções curtas não precisam de muita descrição. Um nome bem escolhido para uma pequena função que faz uma coisa geralmente é melhor do que um cabeçalho de comentário.

Javadocs em código não público

Por mais úteis que sejam os javadocs para APIs públicas, eles são um anátema para o código que não se destina ao consumo público. Gerar páginas javadoc para as classes e funções dentro de um sistema geralmente não é útil, e a formalidade extra dos comentários javadoc equivale a pouco mais do que lixo e distração.

Exemplo

Escrevi o módulo na Listagem 4-7 para a primeira *Imersão XP*. A intenção era ser um exemplo de código ruim e estilo de comentário. Kent Beck então refatorou esse código em uma forma muito mais agradável na frente de várias dezenas de alunos entusiasmados. Mais tarde, adaptei o exemplo para meu livro *Agile Software Development, Principles, Patterns, and Practices* e o primeiro de meus artigos *Craftsman* publicado na revista *Software Development*.

O que acho fascinante sobre este módulo é que houve um tempo em que muitos de nós o consideraríamos “bem documentado”. Agora vemos isso como uma pequena bagunça. Veja quantos problemas de comentários diferentes você pode encontrar.

Listagem

4-7 GeneratePrimes.java /** *

```
Esta classe Gera números primos até um * máximo especificado pelo
usuário. O algoritmo utilizado é o Peneiro de Eratóstenes.
<p>
* Eratóstenes de Cirene, bc 276 aC, Cirene, Líbia -- * dc 194, Alexandria.
O primeiro homem a calcular a *circunferência da Terra. Também conhecido
por trabalhar em * calendários com anos bissextos e dirigia a biblioteca de
Alexandria.
<p>
* O algoritmo é bastante simples. Dado um array de inteiros * começando em 2.
Risque todos os múltiplos de 2. Encontre o próximo * inteiro não cruzado e risque
todos os seus múltiplos.
* Repita até passar a raiz quadrada do valor * máximo.
*
* @autor Alphonse *
@version 13 de fevereiro de 2002

atp */ import java.util.*;

public class GeneratePrimes {

    /**
     * @param maxValue é o limite de geração. */
    public static int[] generatePrimes(int maxValue) {

        if (maxValue >= 2) // o único caso válido {

            // declarações int s
            = maxValue + 1; // tamanho do array boolean[] f =
            new boolean[s]; int eu;
```

Listagem 4-7 (continuação)**GeneratePrimes.java // inicializa**

```

o array como true. for (i = 0; i < s; i+
+) f[i] = verdadeiro;

// elimina os não primos conhecidos
f[0] = f[1] = false;

// peneira
int j;
for (i = 2; i < Math.sqrt(s) + 1; i++) {

    if (f[i]) // se i não for cruzado, cruza seus múltiplos. {

        para (j = 2 * i; j < s; j += i) f[j] = falso; //
múltiplo não é primo
    }
}

// quantos primos existem? contagem
int = 0; for (i = 0;
i < s; i++) {

    if (f[i])
        conta++; // contagem de batidas.
}

int[] primos = new int[contagem];

// move os primos para o resultado for (i =
0, j = 0; i < s; i++) {

    if (f[i]) // se primo
        primos[j++] = i;
}

retorna primos; // retorna os primos

} else // maxValue < 2
    return new int[0]; // retorna array nulo se entrada incorreta.
}
}

```

Na Listagem 4-8 você pode ver uma versão refatorada do mesmo módulo. Observe que o uso de comentários é significativamente restrito. Existem apenas dois comentários em todo o módulo. Ambos os comentários são de natureza explicativa.

Listagem 4-8**PrimeGenerator.java (refatorado) /** * Esta classe Gera**

números primos até um * máximo especificado pelo usuário. O algoritmo utilizado é o Peneiro de Eratóstenes.

* Dado um array de inteiros começando em 2: * Encontre o primeiro inteiro não cruzado, e risque todos os seus

Listagem 4-8 (continuação)**PrimeGenerator.java (refeito)**

* múltiplos. Repita até que não haja mais múltiplos * na matriz. */

```
public class PrimeGenerator {

    private static boolean[] crossOut; resultado int[]
    estático privado;

    public static int[] generatePrimes(int maxValue) {

        if (maxValor < 2)
            retornar novo int[0];
        else

            { uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult(); resultado
            de retorno;
            }
    }

    private static void uncrossIntegersUpTo(int maxValue) {

        cruzadoOut = new boolean[maxValue + 1]; for (int i
        = 2; i < crossOut.length; i++)
        cruzadoOut[i] = falso;
    }

    private static void crossOutMultiples() {

        limite int = determinelterationLimit(); for (int i = 2;
        i <= limite; i++) if (notCrossed(i))
        crossOutMultiplesOf(i);

    }

    private static int determinelterationLimit() {

        // Cada múltiplo no array tem um fator primo que // é menor ou igual à
        // raiz do tamanho do array, // então não temos que riscar múltiplos de
        // números // maiores que essa raiz. double iterationLimit =
        Math.sqrt(crossedOut.length);
        return (int) iteraçãoLimite;

    }

    private static void crossOutMultiplesOf(int i) {

        for (int multiple = 2*i; multiple <
        crossOut.length; multiple += i)
        crossOut[multiple]
        = true;
    }
}
```

Listagem 4-8 (continuação)

PrimeGenerator.java (refatorado) private static

```
boolean notCrossed(int i) {  
    return crossOut[i] == false;  
}  
  
private static void putUncrossedIntegersIntoResult() {  
  
    resultado = new int[numberOfUncrossedIntegers()]; for (int j = 0, i = 2; i < crossOut.length; i++)  
        if (nãoCruzado(i))  
            resultado[j++] = i;  
    }  
  
private static int numberOfUncrossedIntegers() {  
  
    contagem int = 0;  
    for (int i = 2; i < crossOut.length; i++) if (notCrossed(i))  
        count++;  
  
    contagem de retorno;  
    }  
}
```

É fácil argumentar que o primeiro comentário é redundante porque parece muito com a própria função `generatePrimes`. Ainda assim, acho que o comentário serve para facilitar o leitor a entrar no algoritmo, então estou inclinado a deixá-lo.

O segundo argumento é quase certamente necessário. Ele explica o raciocínio por trás do uso da raiz quadrada como o limite do loop. Não consegui encontrar nenhum nome de variável simples, nem qualquer estrutura de codificação diferente que esclarecesse esse ponto. Por outro lado, o uso da raiz quadrada pode ser um conceito. Estou realmente economizando tanto tempo limitando a iteração à raiz quadrada? O cálculo da raiz quadrada pode levar mais tempo do que estou economizando?

Vale a pena pensar. Usar a raiz quadrada como limite de iteração satisfaz o antigo C e o hacker da linguagem assembly em mim, mas não estou convencido de que valha a pena o tempo e o esforço que todo mundo gastará para entendê-la.

Bibliografia

[KP78]: Kernighan e Plaugher, *Os Elementos do Estilo de Programação*, 2d. ed., McGraw Hill, 1978.

5

Formatação



Quando as pessoas olham sob o capô, queremos que fiquem impressionadas com a limpeza, consistência e atenção aos detalhes que percebem. Queremos que eles fiquem impressionados com a ordem. Queremos que suas sobrancelhas se levantem enquanto percorrem os módulos. Queremos que eles percebam que os profissionais estão trabalhando. Se, em vez disso, virem uma massa de código embaralhada que parece ter sido escrita por um bando de marinheiros bêbados, é provável que concluam que a mesma falta de atenção aos detalhes permeia todos os outros aspectos do projeto.

Você deve tomar cuidado para que seu código esteja bem formatado. Você deve escolher um conjunto de regras simples que regem o formato do seu código e, em seguida, aplicar essas regras de forma consistente. Se você estiver trabalhando em uma equipe, a equipe deve concordar com um único conjunto de regras de formatação e todos os membros devem obedecer. Ajuda ter uma ferramenta automatizada que pode aplicar essas regras de formatação para você.

O objetivo da formatação

Em primeiro lugar, vamos ser claros. A formatação do código é *importante*. É muito importante para ignorar e é muito importante para tratar religiosamente. A formatação de código é sobre comunicação, e a comunicação é a primeira ordem de trabalho do desenvolvedor profissional.

Talvez você tenha pensado que “fazer funcionar” era a primeira tarefa de um desenvolvedor profissional. Espero que agora, no entanto, que este livro tenha tirado você dessa ideia. A funcionalidade que você cria hoje tem uma boa chance de mudar na próxima versão, mas a legibilidade do seu código terá um efeito profundo em todas as mudanças que serão feitas. O estilo de codificação e a legibilidade estabelecem precedentes que continuam a afetar a capacidade de manutenção e a extensibilidade muito depois de o código original ter sido alterado além do reconhecimento. Seu estilo e disciplina sobrevivem, mesmo que seu código não.

Então, quais são os problemas de formatação que nos ajudam a nos comunicar melhor?

Formatação vertical

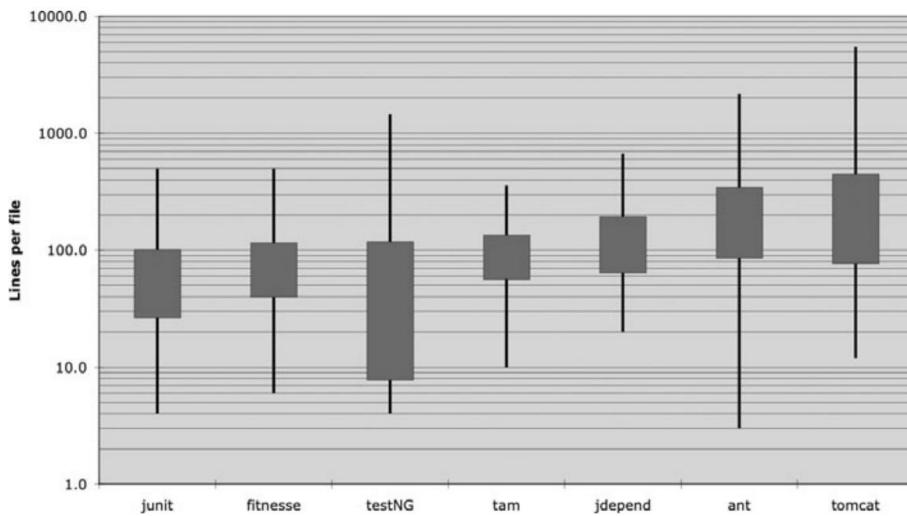
Vamos começar com o tamanho vertical. Qual deve ser o tamanho de um arquivo de origem? Em Java, o tamanho do arquivo está intimamente relacionado ao tamanho da classe. Falaremos sobre o tamanho da classe quando falarmos sobre classes. Por enquanto, vamos apenas considerar o tamanho do arquivo.

Qual é o tamanho da maioria dos arquivos de origem Java? Acontece que existe uma enorme variedade de tamanhos e algumas notáveis diferenças de estilo. A Figura 5-1 mostra algumas dessas diferenças.

Sete projetos diferentes são descritos. Junit, FitNesse, testNG, Time and Money, JDepend, Ant e Tomcat. As linhas nas caixas mostram os tamanhos mínimo e máximo de arquivo em cada projeto. A caixa mostra aproximadamente um terço (um desvio padrão¹) dos arquivos. O meio da caixa é a média. Portanto, o tamanho médio do arquivo no projeto FitNesse é de cerca de 65 linhas, e cerca de um terço dos arquivos tem entre 40 e 100 linhas. O maior arquivo no FitNesse tem cerca de 400 linhas e o menor tem 6 linhas.

Observe que esta é uma escala logarítmica, portanto, a pequena diferença na posição vertical implica uma diferença muito grande no tamanho absoluto.

1. A caixa mostra $\sigma/2$ acima e abaixo da média. Sim, eu sei que a distribuição do tamanho do arquivo não é normal e, portanto, o desvio padrão não é matematicamente preciso. Mas não estamos tentando precisão aqui. Estamos apenas tentando ter uma ideia.

**Figura 5-1**

Escala LOG de distribuições de comprimento de arquivo (altura da caixa = sigma)

Junit, FitNesse e Time and Money são compostos de arquivos relativamente pequenos. Nenhum tem mais de 500 linhas e a maioria desses arquivos tem menos de 200 linhas. O Tomcat e o Ant, por outro lado, têm alguns arquivos com vários milhares de linhas e quase a metade com mais de 200 linhas.

o que aquilo significa para nós? Parece ser possível construir sistemas significativos (o FitNesse tem cerca de 50.000 linhas) a partir de arquivos que normalmente têm 200 linhas, com um limite máximo de 500. Embora essa não deva ser uma regra rígida e rápida, deve ser considerada muito desejável. Arquivos pequenos geralmente são mais fáceis de entender do que arquivos grandes.

A Metáfora do Jornal

Pense em um artigo de jornal bem escrito. Você lê verticalmente. No topo, você espera uma manchete que lhe diga do que se trata a história e permita que você decida se é algo que deseja ler. O primeiro parágrafo fornece uma sinopse de toda a história, ocultando todos os detalhes enquanto fornece conceitos gerais. À medida que você continua descendo, os detalhes aumentam até que você tenha todas as datas, nomes, citações, reclamações e outras minúcias.

Gostaríamos que um arquivo de origem fosse como um artigo de jornal. O nome deve ser simples, mas explicativo. O nome, por si só, deve ser suficiente para nos dizer se estamos no módulo certo ou não. As partes superiores do arquivo de origem devem fornecer o nível superior

conceitos e algoritmos. O detalhe deve aumentar à medida que descemos, até que no final encontramos as funções e detalhes de nível mais baixo no arquivo de origem.

Um jornal é composto de muitos artigos; a maioria é muito pequena. Alguns são um pouco maiores. Muito poucos contêm tanto texto quanto uma página pode conter. Isso torna o jornal *utilizável*. Se o jornal fosse apenas uma longa história contendo um aglomerado desordenado de fatos, datas e nomes, simplesmente não o leríamos.

Abertura Vertical Entre Conceitos

Quase todo o código é lido da esquerda para a direita e de cima para baixo. Cada linha representa uma expressão ou cláusula, e cada grupo de linhas representa um pensamento completo. Esses pensamentos devem ser separados uns dos outros com linhas em branco.

Considere, por exemplo, a Listagem 5-1. Existem linhas em branco que separam a declaração do pacote, a(s) importação(ões) e cada uma das funções. Essa regra extremamente simples tem um efeito profundo no layout visual do código. Cada linha em branco é uma sugestão visual que identifica um conceito novo e separado. Conforme você examina a listagem, seu olhar é atraído para a primeira linha que segue uma linha em branco.

Listagem 5-1

BoldWidget.java package

```
fitnesse.wikitext.widgets;  
  
importar java.util.regex.*;  
  
public class BoldWidget extends ParentWidget {  
    public static final String REGEXP = "^.+?"; private static final  
    Pattern pattern = Pattern.compile("^(.+?)",  
        Padrão.MULTILINE + Padrão.DOTALL);  
  
    public BoldWidget(ParentWidget pai, String text) lança Exception {  
        super(pai); Matcher  
        match = pattern.matcher(text); match.find();  
  
        addChildWidgets(match.group(1));  
    }  
  
    public String render() lança exceção {  
        StringBuffer html = new StringBuffer("<b>");  
        html.append(childHtml()).append("</b>"); return  
        html.toString();  
    }  
}
```

Tirar essas linhas em branco, como na Listagem 5-2, tem um efeito notavelmente obscuro na legibilidade do código.

Formatação vertical**Listagem****5-2 BoldWidget.java**

```
package fitnessse.wikitext.widgets; importar
java.util.regex.*; public class
BoldWidget extends ParentWidget {
    public static final String REGEXP = ".+?"; private static final
    Pattern pattern = Pattern.compile("(.)",
        Padrão.MULTILINE + Padrão.DOTALL);
    public BoldWidget(ParentWidget pai, String text) lança Exception {
        super(pai); Matcher
        match = pattern.matcher(text); match.find();

        addChildWidgets(match.group(1));} public
    String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>"); return
        html.toString();
    }
}
```

Esse efeito é ainda mais pronunciado quando você desfoca os olhos. No primeiro exemplo, os diferentes agrupamentos de linhas aparecem em você, enquanto o segundo exemplo parece uma confusão. A diferença entre essas duas listas é um pouco de abertura vertical.

Densidade Vertical

Se a abertura separa os conceitos, a densidade vertical implica uma associação próxima. Portanto, linhas de código estreitamente relacionadas devem aparecer verticalmente densas. Observe como os comentários inúteis na Listagem 5-3 quebram a estreita associação das duas variáveis de instância.

Listagem

```
5-3 public class ReporterConfig {

    /**
     * O nome da classe do ouvinte do repórter */ private
    String m_className;

    /**
     * As propriedades do ouvinte do repórter */ private
    List<Propriedade> m_properties = new ArrayList<Propriedade>();

    public void addProperty(propriedade de propriedade)
        { m_properties.add(propriedade);
    }
```

A Listagem 5-4 é muito mais fácil de ler. Cabe em um “olho cheio”, ou pelo menos para mim. Posso olhar e ver que se trata de uma classe com duas variáveis e um método, sem precisar mexer muito a cabeça ou os olhos. A listagem anterior me obriga a usar muito mais movimento dos olhos e da cabeça para atingir o mesmo nível de compreensão.

Listagem 5-4

```
public class ReporterConfig { private
    String m_className; private
    List<Propriedade> m_properties = new ArrayList<Propriedade>();

    public void addProperty(propriedade de propriedade)
        { m_properties.add(propriedade);
    }
```

Distância Vertical

Você já perseguiu seu rabo através de uma classe, pulando de uma função para outra, rolando para cima e para baixo no arquivo de origem, tentando adivinhar como as funções se relacionam e operam, apenas para se perder em um ninho de rato de confusão? Você já procurou na cadeia de herança para a definição de uma variável ou função? Isso é frustrante porque você está tentando entender *o que* o sistema faz, mas está gastando seu tempo e energia mental tentando localizar e lembrar *onde* estão as peças.

Conceitos que estão intimamente relacionados devem ser mantidos verticalmente próximos uns dos outros [G10]. Claramente esta regra não funciona para conceitos que pertençam a arquivos separados. Mas conceitos intimamente relacionados não devem ser separados em arquivos diferentes, a menos que você tenha um motivo muito bom. Na verdade, esta é uma das razões pelas quais as variáveis protegidas devem ser evitadas.

Para aqueles conceitos que estão tão intimamente relacionados que pertencem ao mesmo arquivo de origem, sua separação vertical deve ser uma medida de quanto importante cada um é para a capacidade de compreensão do outro. Queremos evitar forçar nossos leitores a percorrer nossos arquivos de origem e classes.

Declarações de variáveis. As variáveis devem ser declaradas o mais próximo possível de seu uso. Como nossas funções são muito curtas, as variáveis locais devem aparecer no topo de cada função, como nesta função longa do Junit4.3.1.

```
private static void readPreferences() {
    InputStream é = nulo; tente
    { is=
        new FileInputStream(getPreferencesFile()); setPreferences(new
        Properties(getPreferences())); getPreferences().load(is);

    } catch (IOException e) { try { if (is !
        = null)
            is.close(); } catch
            (IOException
            e1) { }

    }
}
```

As variáveis de controle para loops geralmente devem ser declaradas dentro da instrução de loop, como nesta pequena função bonitinha da mesma fonte.

```
public int countTestCases() { int
    count= 0; for
        (Testar cada : testes)
            contagem += cada.countTestCases();
            contagem de retorno;
    }
```

Em casos raros, uma variável pode ser declarada no início de um bloco ou logo antes de um loop em uma função longa. Você pode ver essa variável neste snippet no meio de uma função muito longa em TestNG.

```
...
for (teste XmlTest: m_suite.getTests()) { TestRunner
    tr = m_runnerFactory.newTestRunner(this, test);
    tr.addListener(m_textReporter);
    m_testRunners.add(tr);

    invocador = tr.getInvoker();

    for (ITestNGMethod m : tr.getBeforeSuiteMethods())
        { beforeSuiteMethods.put(m.getMethod(), m);
    }

    for (ITestNGMethod m : tr.getAfterSuiteMethods())
        { afterSuiteMethods.put(m.getMethod(), m);
    }
}
...
```

As variáveis de instância, por outro lado, devem ser declaradas no início da classe. Isso não deve aumentar a distância vertical dessas variáveis, porque em uma classe bem projetada, elas são usadas por muitos, senão todos, dos métodos da classe.

Tem havido muitos debates sobre onde as variáveis de instância devem ir. Em C++, comumente praticamos a chamada *regra da tesoura*, que coloca todas as variáveis de instância na parte inferior. A convenção comum em Java, no entanto, é colocá-los todos no topo da classe. Não vejo razão para seguir qualquer outra convenção. O importante é que as variáveis de instância sejam declaradas em um local bem conhecido. Todos devem saber onde ir para ver as declarações.

Considere, por exemplo, o estranho caso da classe TestSuite no JUnit 4.3.1. Eu atenuei muito esta classe para fazer o ponto. Se você olhar na metade da listagem, verá duas variáveis de instância declaradas lá. Seria difícil escondê-los em um lugar melhor. Alguém lendo este código teria que tropeçar nas declarações por acidente (como eu fiz).

```
public class TestSuite implementa Teste {
    static public Test createTest(Class<? extends TestCase> theClass, String name) {
        ...
    }
```

```

Construtor estático público <? extends TestCase>
getTestConstructor(Class<? extends TestCase> theClass) throws
NoSuchMethodException {
    ...
}

aviso de teste estático público (mensagem de string final) {
    ...
}

string estática privada exceçãoToString(Throwable t) {
    ...
}

string privada fName;

private Vector<Test> fTests= new Vector<Test>(10);

public TestSuite() { }

public TestSuite(final Class<? extends TestCase> theClass) {
    ...
}

public TestSuite(Class<? extends TestCase> theClass, String name) {
    ...
}
...
}

```

Funções dependentes. Se uma função chamar outra, elas devem estar próximas verticalmente e o chamador deve estar acima do chamado, se possível. Isso dá ao programa um fluxo natural. Se a convenção for seguida de forma confiável, os leitores poderão confiar que as definições de função seguirão logo após seu uso. Considere, por exemplo, o trecho de FitNesse na Listagem 5-5. Observe como a função superior chama as que estão abaixo dela e como elas, por sua vez, chamam as que estão abaixo delas. Isso facilita a localização das funções chamadas e aumenta muito a legibilidade de todo o módulo.

Listagem

5-5 WikiPageResponder.java

```

public class WikiPageResponder implements SecureResponder
{ página WikiPage protegida;
  PageData protegido pageData;
  string protegida pageTitle;
  solicitação de solicitação
  protegida; rastreador PageCrawler protegido;

  public Response makeResponse(contexto FitNesseContext, solicitação de solicitação)
    lança exceção {
      String pageName = getPageNameOrDefault(request, "FrontPage");

```

Listagem 5-5 (continuação)**WikiPageResponder.java**

```

loadPage(pageName, contexto); if
(page == null) return
    notFoundResponse(context, request); caso contrário,
        retorno makePageResponse(contexto);
}

private String getPageNameOrDefault(Solicitação de solicitação, String defaultPageName) {

    String pageName = request.getResource(); if
    (StringUtil.isBlank(pageName)) pageName =
        defaultPageName;

    return pageName;
}

Protected void loadPage(String resource, FitNesseContext context) throws Exception
{
    WikiPagePath path =
    PathParser.parse(resource); rastreador =
    context.root.getPageCrawler();
    crawler.setDeadEndStrategy(novo VirtualEnabledPageCrawler()); página =
    crawler.getPage(context.root, caminho); if (page != null)
        pageData =
            page.getData();
}

resposta privada notFoundResponse(contexto FitNesseContext, solicitação de solicitação) throws
Exception { return new
    NotFoundResponder().makeResponse(contexto, solicitação);
}

private SimpleResponse makePageResponse(FitNesseContext context) lança exceção
{ pageTitle =
    PathParser.render(crawler.getFullPath(page)); String html = makeHtml(contexto);

    Resposta SimpleResponse = new SimpleResponse();
    resposta.setMaxAge(0);
    resposta.setContent(html); resposta
    de retorno;
}
...

```

Como um aparte, este trecho fornece um bom exemplo de como manter as constantes no nível apropriado [G35]. A constante "FrontPage" poderia ter sido ocultada na função getPageNameOrDefault , mas isso teria ocultado uma constante conhecida e esperada em uma função inadequada de baixo nível. Era melhor passar essa constante do lugar onde faz sentido conhecê-la para o lugar que realmente a usa.

Afinidade Conceitual. Certos bits de código *querem* estar perto de outros bits. Eles têm uma certa afinidade conceitual. Quanto mais forte essa afinidade, menos distância vertical deve haver entre eles.

Como vimos, essa afinidade pode ser baseada em uma dependência direta, como uma função chamando outra ou uma função usando uma variável. Mas há outras causas possíveis de afinidade. A afinidade pode ser causada porque um grupo de funções executa uma operação semelhante. Considere este trecho de código do Junit 4.3.1:

```
classe pública Asserção {
    static public void assertTrue(String message, boolean condition) { if (!condition)
        fail(message);
    }

    static public void assertTrue(condição booleana) {
        assertTrue(nulo, condição);
    }

    static public void assertFalse(String message, boolean condition) { assertTrue(message, !
        condition);
    }

    static public void assertFalse(condição booleana) {
        assertFalse(nulo, condição);
    }
    ...
}
```

Essas funções têm uma forte afinidade conceitual porque compartilham um esquema de nomenclatura comum e executam variações da mesma tarefa básica. O fato de eles ligarem um para o outro é secundário. Mesmo que não o fizessem, eles ainda iriam querer ficar juntos.

Ordenação vertical

Em geral, queremos que as dependências de chamada de função apontem na direção descendente. Ou seja, uma função que é chamada deve estar abaixo de uma função que faz a chamada.² Isso cria um bom fluxo no módulo de código-fonte do nível superior para o nível inferior.

Como nos artigos de jornal, esperamos que os conceitos mais importantes venham primeiro e que sejam expressos com o mínimo de detalhes poluidores. Esperamos que os detalhes de baixo nível venham por último. Isso nos permite examinar os arquivos de origem, obtendo a essência do



². Isso é exatamente o oposto de linguagens como Pascal, C e C++, que obrigam funções a serem definidas, ou pelo menos declaradas, antes de serem usados.

primeiras funções, sem ter que mergulhar nos detalhes. A listagem 5-5 é organizada dessa maneira. Talvez exemplos ainda melhores sejam a Listagem 15-5 na página 263 e a Listagem 3-7 na página 50.

Formatação Horizontal

Qual deve ser a largura de uma linha? Para responder a isso, vamos ver a largura das linhas em programas típicos. Mais uma vez, examinamos os sete projetos diferentes. A Figura 5-2 mostra a distribuição dos comprimentos de linha de todos os sete projetos. A regularidade é impressionante, especialmente em torno de 45 caracteres. De fato, cada tamanho de 20 a 60 representa cerca de 1% do número total de linhas. Isso é 40 por cento! Talvez outros 30% tenham menos de 10 caracteres de largura. Lembre-se de que esta é uma escala logarítmica, portanto, a aparência linear da queda acima de 80 caracteres é realmente muito significativa. Os programadores claramente preferem linhas curtas.

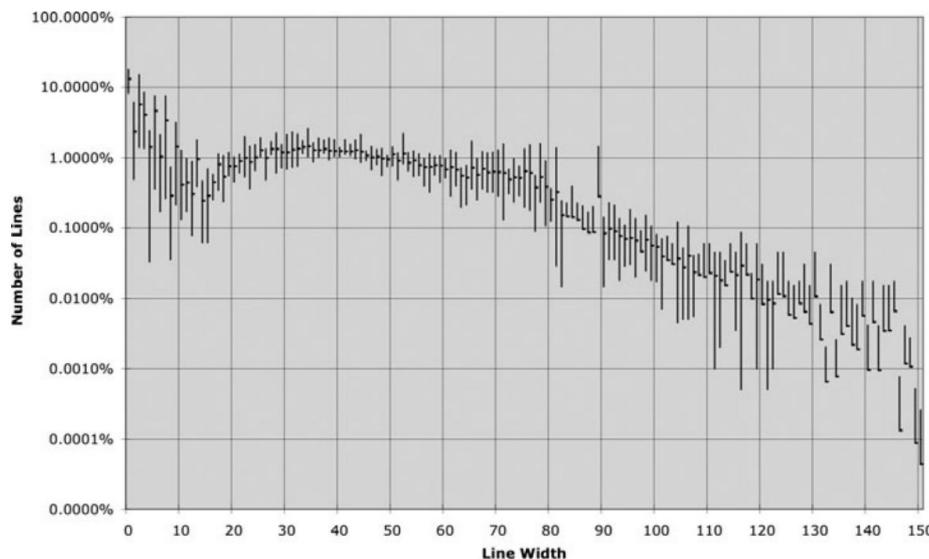


Figura 5-2

Distribuição de largura de linha Java

Isso sugere que devemos nos esforçar para manter nossas falas curtas. O antigo limite Hollerith de 80 é um pouco arbitrário e não me oponho a linhas que cheguem a 100 ou mesmo a 120. Mas, além disso, provavelmente é apenas descuido.

Eu costumava seguir a regra de que você nunca deveria ter que rolar para a direita. Mas os monitores são muito largos para isso hoje em dia, e os programadores mais jovens podem encolher a fonte tão pequena

que eles podem obter 200 caracteres na tela. Não faça isso. Pessoalmente, defino meu limite em 120.

Abertura horizontal e densidade

Usamos espaço em branco horizontal para associar coisas fortemente relacionadas e desassociar coisas que são menos relacionadas. Considere a seguinte função:

```
private void measureLine(String line) { lineCount++;
    int lineSize =
        line.length(); totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize,
        lineCount); recordWidestLine(lineSize);

}
```

Circulei os operadores de atribuição com espaços em branco para acentuá-los. As instruções de atribuição têm dois elementos principais e distintos: o lado esquerdo e o lado direito. Os espaços tornam essa separação óbvia.

Por outro lado, não coloquei espaços entre os nomes das funções e os parênteses de abertura. Isso ocorre porque a função e seus argumentos estão intimamente relacionados. Separá-los faz com que pareçam separados em vez de unidos. Eu separo os argumentos dentro dos parênteses de chamada de função para acentuar a vírgula e mostrar que os argumentos estão separados.

Outro uso para o espaço em branco é acentuar a precedência dos operadores.

```
public class Quadrática {
    public static double root1(double a, double b, double c) { double determinante =
        determinante(a, b, c); return (-b + Math.sqrt(determinante)) /
        (2*a);
    }

    public static double root2(int a, int b, int c) { double determinante =
        determinante(a, b, c); return (-b - Math.sqrt(determinante)) /
        (2*a);
    }

    determinante duplo estático privado(duplo a, duplo b, duplo c) { return b*b - 4*a*c;
}
```

Observe como as equações são bem lidas. Os fatores não têm espaço em branco entre eles porque são de alta precedência. Os termos são separados por espaços em branco porque a adição e a subtração têm menor precedência.

Infelizmente, a maioria das ferramentas para reformatar o código é cega para a precedência dos operadores e impõe o mesmo espaçamento por toda parte. Portanto, espaçamentos sutis como os mostrados acima tendem a se perder depois que você reformata o código.

Alinhamento horizontal

Quando eu era um programador em linguagem assembly,³ eu usava o alinhamento horizontal para acentuar certas estruturas. Quando comecei a codificar em C, C++ e, eventualmente, em Java, continuei tentando alinhar todos os nomes de variáveis em um conjunto de declarações ou todos os rvalues em um conjunto de instruções de atribuição. Meu código poderia ter ficado assim:

```
public class FitNesseExpediter implements ResponseSender {

    private Socket private          soquete;
    InputStream private           entrada;
    OutputStream private Request  saída;
    private Response             solicitar;
    private FitNesseContext      resposta;
    context; longo protegido requestParsingTimeLimit;
    private long requestProgress; pedido longo privadoParsingDeadline;
    private boolean hasError;

    public FitNesseExpediter(Socket           s,
                           Contexto FitNesseContext) lança exceção
    {
        this.context =           contexto;
        soquete = s; entrada =
        s.getInputStream(); s.getOutputStream(); saída =
        10000;           requestParsingTimeLimit =
    }
}
```

Descobri, no entanto, que esse tipo de alinhamento não é útil. O alinhamento parece enfatizar as coisas erradas e desviar meu olhar da verdadeira intenção. Por exemplo, na lista de declarações acima, você fica tentado a ler a lista de nomes de variáveis sem examinar seus tipos. Da mesma forma, na lista de instruções de atribuição, você é tentado a examinar a lista de rvalues sem nunca ver o operador de atribuição. Para piorar a situação, as ferramentas de reformatação automática geralmente eliminam esse tipo de alinhamento.

Então, no final, eu não faço mais esse tipo de coisa. Hoje em dia prefiro declarações e atribuições desalinhadas, como mostrado abaixo, porque apontam uma deficiência importante. Se eu tiver listas longas que precisam ser alinhadas, *o problema é o tamanho das listas*, não a falta de alinhamento. O comprimento da lista de declarações no FitNesseExpediter abaixo sugere que esta classe deve ser dividida.

```
public class FitNesseExpediter implements ResponseSender {

    Tomada de tomada privada;
    entrada InputStream privada; saída
    OutputStream privada; solicitação de
    solicitação privada;
```

³. Quem estou enganando? Eu ainda sou um programador de linguagem assembly. Você pode tirar o menino do metal, mas não pode tirar o metal do menino!

```

    resposta de resposta privada;
    contexto FitNesseContext privado; longo
    protegido requestParsingTimeLimit; private long
    requestProgress; pedido longo
    privadoParsingDeadline; private boolean hasError;

    public FitNesseExpediter(Socket s, contexto FitNesseContext) lança Exception {
        this.context = contexto; soquete
        = s; entrada =
        s.getInputStream(); saída =
        s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}

```

Recuo

Um arquivo de origem é uma hierarquia, como um esboço. Há informações que pertencem ao arquivo como um todo, às classes individuais dentro do arquivo, aos métodos dentro das classes, aos blocos dentro dos métodos e recursivamente aos blocos dentro dos blocos. Cada nível dessa hierarquia é um escopo no qual os nomes podem ser declarados e no qual as declarações e comandos executáveis são interpretados.

Para tornar essa hierarquia de escopos visível, recuamos as linhas do código-fonte proporcionalmente à sua posição na hierarquia. As instruções no nível do arquivo, como a maioria das declarações de classe, não são recuadas. Os métodos dentro de uma classe são recuados um nível à direita da classe. As implementações desses métodos são implementadas um nível à direita da declaração do método. As implementações de bloco são implementadas um nível à direita do bloco que as contém e assim por diante.

Os programadores dependem muito desse esquema de indentação. Eles alinham visualmente as linhas à esquerda para ver em qual escopo aparecem. Isso permite que eles saltem rapidamente sobre os escopos, como implementações de instruções if ou while , que não são relevantes para sua situação atual. Eles examinam a esquerda em busca de novas declarações de método, novas variáveis e até mesmo novas classes. Sem indentação, os programas seriam virtualmente ilegíveis por humanos.

Considere os seguintes programas que são sintáticamente idênticos:

```

public class FitNesseServer implementa SocketServer { private FitNesseContext context; public
FitNesseServer(contexto FitNesseContext) { this.context = contexto; } public void serve(Socket s)
{ serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter
sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout);
sender.start(); } catch(Exception e)
{ e.printStackTrace(); } }

```

```

public class FitNesseServer implementa SocketServer { private
FitNesseContext context;

```

```

public FitNesseServer(contexto FitNesseContext) { this.context = 
    contexto;
}

public void serve(Socket s) { serve(s,
    10000);
}

public void serve(Socket s, requestTimeout longo) {
    try
    { FitNesseExpediter sender = new FitNesseExpediter(s, contexto);
        sender.setRequestParsingTimeLimit(requestTimeout); sender.start();

    } catch (Exception e)
    { e.printStackTrace();
    }
}
}

```

Seu olho pode discernir rapidamente a estrutura do arquivo recuado. Você pode identificar quase instantaneamente as variáveis, construtores, acessadores e métodos. Leva apenas alguns segundos para perceber que este é algum tipo de front-end simples para um soquete, com um tempo limite. A versão sem recuos, no entanto, é virtualmente impenetrável sem um estudo intenso.

Quebra de indentação. Às vezes, é tentador quebrar a regra de indentação para instruções if curtas , loops while curtos ou funções curtas. Sempre que sucumbi a essa tentação, quase sempre voltei e coloquei o recuo de volta. Portanto, evito reduzir os escopos a uma linha como esta:

```

public class CommentWidget extends TextWidget {

    public static final String REGEXP = "^#[^\r\n]*(?:\r\n|\n\r)?";
    public CommentWidget(ParentWidget parent, String text){super(parent, text);}
    public String render() throws Exception {return "";}
}

```

Prefiro expandir e recuar os escopos, assim:

```

public class CommentWidget extends TextWidget { public static
    final String REGEXP = "^#[^\r\n]*(?:\r\n|\n\r)?";

    public CommentWidget(ParentWidget pai, String texto) {
        super(pai, texto);
    }

    public String render() lança exceção {
        retornar "";
    }
}

```

Escopos fictícios

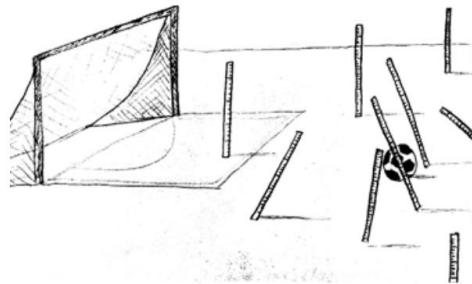
Às vezes, o corpo de um while ou for é uma declaração fictícia, como mostrado abaixo. Não gosto desse tipo de estrutura e tento evitá-las. Quando não posso evitá-las, certifico-me de que o corpo fictício esteja devidamente recortado e cercado por chaves. Não sei dizer quantas vezes fui enganado por um ponto-e-vírgula sentado silenciosamente no final de um loop while na mesma linha. A menos que você torne esse ponto e vírgula visível recuando-o em sua própria linha, é muito difícil de ver.

```
while (dis.read(buf, 0, readBufferSize) != -1)
;
```

Regras da equipe

O título desta seção é um jogo de palavras. Todo programador tem suas próprias regras de formatação favoritas, mas se ele trabalha em equipe, a equipe manda.

Uma equipe de desenvolvedores deve concordar com um único estilo de formatação e todos os membros dessa equipe devem usar esse estilo. Queremos que o software tenha um estilo consistente. Não queremos que pareça ter sido escrito por um grupo de indivíduos discordantes.



Quando comecei o projeto FitNesse em 2002, sentei-me com a equipe para elaborar um estilo de codificação. Isso levou cerca de 10 minutos. Decidimos onde colocaríamos nossas chaves, qual seria nosso tamanho de recuo, como nomearíamos classes, variáveis e métodos e assim por diante. Em seguida, codificamos essas regras no formatador de código do nosso IDE e as mantemos desde então. Essas não eram as regras que eu prefiro; eram regras decididas pela equipe. Como membro dessa equipe, eu os acompanhei ao escrever o código do projeto FitNesse.

Lembre-se, um bom sistema de software é composto de um conjunto de documentos que podem ser lidos de maneira agradável. Eles precisam ter um estilo consistente e suave. O leitor precisa ser capaz de confiar que os gestos de formatação que ele viu em um arquivo de origem significarão a mesma coisa em outros. A última coisa que queremos fazer é adicionar mais complexidade ao código-fonte escrevendo-o em uma mistura de diferentes estilos individuais.

Regras de formatação do tio Bob

As regras que eu uso pessoalmente são muito simples e são ilustradas pelo código na Listagem 5-6. Considere isso como um exemplo de como o código torna o melhor documento padrão de codificação.

Listagem**5-6 CodeAnalyzer.java**

```
public class CodeAnalyzer implements JavaFileAnalysis
    { private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }

    public static List<File> findJavaFiles(File parentDirectory) {
        List<Arquivo> arquivos = new ArrayList<Arquivo>();
        findJavaFiles(parentDirectory, arquivos); retornar
        arquivos;
    }

    private static void findJavaFiles(File parentDirectory, List<File> files) { for (File file : parentDirectory.listFiles())
        { if (file.getName().endsWith(".java")) files.add(arquivo); else if
            (arquivo.isDirectory()) findJavaFiles(arquivo,
            arquivos);

        }
    }

    public void analysisFile(Arquivo javaFile) lança exceção {
        BufferedReader br = new BufferedReader(new FileReader(javaFile)); Linha de corda; while
        ((linha =
        br.readLine()) != nulo) medidaLinha(linha);

    }

    private void measureLine(String line) { lineCount++;
        int lineSize =
        line.length(); totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize,
        lineCount); recordWidestLine(lineSize);

    }

    private void recordWidestLine(int lineSize) {
        if (lineSize > maxLineWidth)
        { maxLineWidth = lineSize;
        maislarguraLineNumber = lineCount; }

    }

    public int getLineCount() { return
        lineCount;
    }

    public int getMaxLineWidth() { return
        maxLineWidth;
    }
```

Listagem 5-6 (continuação)**CodeAnalyzer.java**

```
public int getWidestLineNumber() { return
    widestLineNumber;
}

public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
}

public double getMeanLineWidth() {
    return (duplo)totalChars/lineCount;
}

public int getMedianLineWidth() {
    Integer[]sortedWidths = getSortedWidths(); int
    cumulativoLineCount = 0; for (int
    width : sortedWidths) {
        cumulativoLineCount += lineCountForWidth(largura); if
        (cumulativeLineCount > lineCount/2) largura de
        retorno;
    } throw new Error("Não é possível chegar aqui");
}

private int lineCountForWidth(int width) { return
    lineWidthHistogram.getLinesforWidth(width).size();
}

private Integer[] getSortedWidths() { Set<Integer>
    larguras = lineWidthHistogram.getWidths(); Integer[] sortedWidths =
    (widths.toArray(new Integer[0])); Arrays.sort(sortedWidths); return larguras
    classificadas;
}
```

6

Objetos e Estruturas de Dados



Há uma razão para mantermos nossas variáveis privadas. Não queremos que mais ninguém dependa deles. Queremos manter a liberdade de mudar seu tipo ou implementação por capricho ou impulso. Por que, então, tantos programadores adicionam getters e setters automaticamente a seus objetos, expondo suas variáveis privadas como se fossem públicas?

Abstração de dados

Considere a diferença entre a Listagem 6-1 e a Listagem 6-2. Ambos representam os dados de um ponto no plano cartesiano. E, no entanto, um expõe sua implementação e o outro o esconde completamente.

Listagem 6-1**Ponto de Concreto**

```
public class Point { public
    double x; público
    duplo y;
}
```

Listagem 6-2**Ponto Abstrato**

```
ponto de interface pública
{ double getX();
double getY(); void
setCartesian(duplo x, duplo y); duplo getR(); double
getTheta(); void
setPolar(duplo r, duplo
teta);
}
```

A beleza da Listagem 6-2 é que não há como saber se a implementação está em coordenadas retangulares ou polares. Pode ser nenhum dos dois! E, no entanto, a interface ainda representa inequivocamente uma estrutura de dados.

Mas representa mais do que apenas uma estrutura de dados. Os métodos impõem uma política de acesso. Você pode ler as coordenadas individuais independentemente, mas deve definir as coordenadas juntas como uma operação atômica.

A Listagem 6-1, por outro lado, é claramente implementada em coordenadas retangulares e nos força a manipular essas coordenadas de forma independente. Isso expõe a implementação. Na verdade, isso exporia a implementação mesmo se as variáveis fossem privadas e estivéssemos usando getters e setters de variável única.

Ocultar a implementação não é apenas uma questão de colocar uma camada de funções entre as variáveis. Ocultar a implementação é sobre abstrações! Uma classe não simplesmente envia suas variáveis por meio de getters e setters. Em vez disso, expõe interfaces abstratas que permitem que seus usuários manipulem a *essência* dos dados, sem precisar conhecer sua implementação.

Considere a Listagem 6-3 e a Listagem 6-4. O primeiro usa termos concretos para comunicar o nível de combustível de um veículo, enquanto o segundo o faz com a abstração de porcentagem. No caso concreto, você pode ter certeza de que são apenas acessadores de variáveis. No caso abstrato, você não tem nenhuma pista sobre a forma dos dados.

Listagem 6-3**veículo de concreto**

```
veículo de interface pública {
    double getFuelTankCapacityInGallons(); double
    getGallonsOfGasoline();
}
```

Listagem 6-4
veículo abstrato

```
public interface Vehicle { double
    getPercentFuelRemaining();
}
```

Em ambos os casos acima, a segunda opção é preferível. Não queremos expor os detalhes de nossos dados. Em vez disso, queremos expressar nossos dados em termos abstratos. Isso não é feito simplesmente usando interfaces e/ou getters e setters. Pensamento sério precisa ser colocado na melhor maneira de representar os dados que um objeto contém. A pior opção é adicionar getters e setters alegremente.

Anti-simetria de dados/objetos

Esses dois exemplos mostram a diferença entre objetos e estruturas de dados. Os objetos escondem seus dados atrás de abstrações e expõem funções que operam nesses dados. A estrutura de dados expõe seus dados e não possui funções significativas. Volte e leia isso de novo.

Observe a natureza complementar das duas definições. Eles são opostos virtuais. Essa diferença pode parecer trivial, mas tem implicações de longo alcance.

Considere, por exemplo, o exemplo de formato procedural na Listagem 6-5. A classe Geometry opera nas três classes de forma. As classes de formas são estruturas de dados simples sem nenhum comportamento. Todo o comportamento está na classe Geometry .

Listagem 6-5

Processual Forma public

```
class Quadrado {
    público Ponto superior
    esquerdo; dupla face pública;
}

public class Rectangle { public
    Point topLeft; altura dupla
    pública; largura dupla pública;
}

public class Circle { public
    Point center; duplo raio
    público;
}

public class Geometry { public
    final double PI = 3,141592653589793;

    área dupla pública (formato do objeto) lança NoSuchShapeException {

        if (forma instância de Square) { Square s
            = (Square)shape; return s.side *
            s.side;
        }
    }
}
```

Listagem 6-5 (continuação)

Forma processual

```
else if (instância de forma de Retângulo) { Retângulo  
    r = (Retângulo) forma; return r.height *  
    r.width;  
  
} else if (instância de forma de Círculo) { Círculo  
    c = (Círculo) forma; return PI *  
    c.radius * c.radius;  
  
} lance o novo NoSuchShapeException();  
}  
}
```

Os programadores orientados a objetos podem torcer o nariz para isso e reclamar que é processual - e eles estariam certos. Mas o escârnio pode não ser justificado. Considere o que aconteceria se uma função perimetral() fosse adicionada a Geometry. As classes de forma não seriam afetadas! Quaisquer outras classes que dependessem das formas também não seriam afetadas!

Por outro lado, se eu adicionar uma nova forma, devo alterar todas as funções em Geometria para lidar com ela. Mais uma vez, leia isso. Observe que as duas condições são diametralmente opostas.

Agora considere a solução orientada a objetos na Listagem 6-6. Aqui o método `area()` é polimórfico. Nenhuma aula de Geometria é necessária. Portanto, se eu adicionar uma nova forma, nenhuma das *funções* existentes será afetada, mas se eu adicionar uma nova função, todas as *formas* deverão ser alteradas!

Listagem 6-6

Formas Polimórficas public

```
class Square implements Shape { private Point  
    topLeft; lado duplo privado;  
  
    public double area() { return  
        lado*lado;  
    }  
}  
  
public class Retângulo implementa Forma {  
    ponto privado superior  
    esquerdo; altura dupla privada;  
    largura dupla privada;  
  
    public double area() {  
        altura de retorno * largura;  
    }  
}
```

1. Existem maneiras de contornar isso que são bem conhecidas de designers experientes em orientação a objetos: VISITANTE ou despacho duplo, por exemplo. Mas essas técnicas carregam seus próprios custos e geralmente retornam a estrutura à de um programa processual.

Listagem 6-6 (continuação)**Formas Polimórficas public**

```
class Circle implements Shape {
    centro de ponto privado; raio
    duplo privado; PI duplo final
    público = 3,141592653589793;

    public double area() { return
        PI * radius * radius;
    }
}
```

Novamente, vemos a natureza complementar dessas duas definições; eles são virtuais opostos! Isso expõe a dicotomia fundamental entre objetos e estruturas de dados:

O código procedural (código que usa estruturas de dados) facilita a adição de novas funções sem alterar as estruturas de dados existentes. O código OO, por outro lado, facilita a adição de novas classes sem alterar as funções existentes.

O complemento também é verdadeiro:

O código processual torna difícil adicionar novas estruturas de dados porque todas as funções devem mudar. O código OO torna difícil adicionar novas funções porque todas as classes devem mudar.

Então, as coisas que são difíceis para OO são fáceis para procedimentos, e as coisas que são difíceis para procedimentos são fáceis para OO!

Em qualquer sistema complexo, haverá momentos em que queremos adicionar novos tipos de dados em vez de novas funções. Para esses casos, objetos e OO são os mais apropriados. Por outro lado, também haverá momentos em que desejaremos adicionar novas funções em vez de tipos de dados. Nesse caso, o código processual e as estruturas de dados serão mais apropriados.

Programadores maduros sabem que a ideia de que tudo é um objeto é *um mito*. Alguns vezes você realmente quer estruturas de dados simples com procedimentos operando nelas.

A Lei de Deméter

Existe uma heurística conhecida chamada *Lei de Deméter*² que diz que um módulo não deve conhecer as entranhas dos *objetos* que manipula. Como vimos na seção anterior, os objetos ocultam seus dados e expõem as operações. Isso significa que um objeto não deve expor sua estrutura interna por meio de acessadores porque fazer isso é expor, em vez de ocultar, sua estrutura interna.

Mais precisamente, a Lei de Deméter diz que um método *f* de uma classe *C* só deve chamar os métodos destes:

- *C*
- Um objeto criado por *f*

2. http://en.wikipedia.org/wiki/Law_of_Demeter

- Um objeto passado como um argumento para f •

Um objeto mantido em uma variável de instância de C

O método *não* deve invocar métodos em objetos que são retornados por qualquer um dos funções permitidas. Em outras palavras, converse com amigos, não com estranhos.

O seguinte code3 parece violar a Lei de Demeter (entre outras coisas) porque chama a função `getScratchDir()` no valor de retorno de `getOptions()` e então chama `getAbsolutePath()` no valor de retorno de `getScratchDir()`.

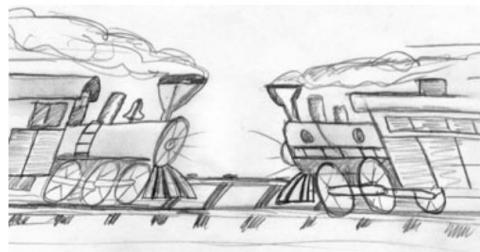
```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Destroços de trem

Esse tipo de código costuma ser chamado de *acidente de trem* porque se parece com um monte de vagões de trem acoplados. Cadeias de chamadas como esta são geralmente consideradas estilo desleixado e devem ser evitadas [G36]. Geralmente, é melhor dividi-los da seguinte maneira:

```
Opções opts = ctxt.getOptions(); Arquivo
scratchDir = opts.getScratchDir(); final String
outputDir = scratchDir.getAbsolutePath();
```

Esses dois fragmentos de código violam a Lei de Deméter? Certamente o módulo recipiente sabe que o objeto ctxt contém opções, que contêm um diretório temporário, que possui um caminho absoluto. É muito conhecimento para uma função saber. A função de chamada sabe como navegar por vários objetos diferentes.



Se isso é uma violação de Demeter depende se ctxt, Options e ScratchDir são ou não objetos ou estruturas de dados. Se eles são objetos, então sua estrutura interna deve ser escondida em vez de exposta, e assim o conhecimento de suas entranhas é uma clara violação da Lei de Deméter. Por outro lado, se ctxt, Options e ScratchDir são apenas estruturas de dados sem comportamento, então eles expõem naturalmente sua estrutura interna e, portanto, Demeter não se aplica.

O uso de funções de acesso confunde o assunto. Se o código tivesse sido escrito como segue baixos, provavelmente não estariamos perguntando sobre as violações de Deméter.

```
final String outputDir = ctxt.options.scratchDir.getAbsolutePath;
```

Esse problema seria muito menos confuso se as estruturas de dados simplesmente tivessem variáveis públicas e nenhuma função, enquanto os objetos tivessem variáveis privadas e funções públicas. No entanto,

3. Encontrado em algum lugar na estrutura do apache.

existem estruturas e padrões (por exemplo, “beans”) que exigem que mesmo estruturas de dados simples tenham acessadores e modificadores.

Híbridos

Essa confusão às vezes leva a estruturas híbridas infelizes que são metade objeto e metade estrutura de dados. Eles têm funções que fazem coisas significativas e também têm variáveis públicas ou acessadores e modificadores públicos que, para todos os efeitos, tornam públicas as variáveis privadas, tentando outras funções externas a usar essas variáveis da mesma forma que um programa procedural usaria um estrutura de dados.⁴

Esses híbridos dificultam a adição de novas funções, mas também dificultam a adição de novas estruturas de dados. Eles são o pior dos dois mundos. Evite criá-los. Eles são indicativos de um design confuso cujos autores não têm certeza - ou pior, ignoram - se precisam de proteção contra funções ou tipos.

Estrutura oculta

E se ctxt, options e scratchDir forem objetos com comportamento real? Então, como os objetos devem esconder sua estrutura interna, não devemos ser capazes de navegar por eles. Como então obteríamos o caminho absoluto do diretório temporário?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
ou
ctx.getScratchDirectoryOption().getAbsolutePath()
```

A primeira opção pode levar a uma explosão de métodos no objeto ctxt. A segunda supõe que getScratchDirectoryOption() retorne uma estrutura de dados, não um objeto. Nenhuma das opções parece boa.

Se ctxt for um objeto, devemos dizer a ele para *fazer algo*; não deveríamos estar perguntando sobre seus componentes internos. Então, por que queremos o caminho absoluto do diretório temporário? O que faríamos com isso? Considere este código (muitas linhas abaixo) do mesmo módulo:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class"; FileOutputStream fout =
new FileOutputStream(outFile); BufferedOutputStream bos = new
BufferedOutputStream(fout);
```

A mistura de diferentes níveis de detalhe [G34][G6] é um pouco preocupante. Pontos, barras, extensões de arquivo e objetos de arquivo não devem ser misturados de forma tão descuidada e misturados com o código anexo. Ignorando isso, no entanto, vemos que a intenção de obter o caminho absoluto do diretório temporário era criar um arquivo temporário com um determinado nome.

4. Isso às vezes é chamado de Feature Envy de [Refactoring].

Então, e se disséssemos ao objeto ctxt para fazer isso?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Isso parece uma coisa razoável para um objeto fazer! Isso permite que o ctxt oculte seus componentes internos e evita que a função atual tenha que violar a Lei de Deméter ao navegar por objetos que não deveria conhecer.

Objetos de transferência de dados

A forma quintessencial de uma estrutura de dados é uma classe com variáveis públicas e sem funções. Às vezes, isso é chamado de objeto de transferência de dados ou DTO. DTOs são estruturas muito úteis, especialmente ao se comunicar com bancos de dados ou analisar mensagens de soquetes e assim por diante. Eles geralmente se tornam os primeiros de uma série de estágios de conversão que convertem dados brutos em um banco de dados em objetos no código do aplicativo.

Um pouco mais comum é a forma “bean” mostrada na Listagem 6-7. Beans possuem variáveis privadas manipuladas por getters e setters. O quase encapsulamento de beans parece fazer alguns puristas OO se sentirem melhor, mas geralmente não oferece nenhum outro benefício.

Listagem 6-7

```
address.java public
```

```
class Address {  
    rua String privada; private  
    String streetExtra; cidade privada de  
    String; estado String  
    privado; zip de string privado;  
  
    public Address(String rua, String ruaExtra, String cidade, String  
        estado, String zip) { this.street = street; esta.ruaExtra =  
        ruaExtra; esta.cidade =  
        cidade; this.state = estado; this.zip = zip;  
  
    }  
    public String getStreet() {  
        rua de retorno;  
    }  
    public String getStreetExtra() { return  
        streetExtra;  
    }  
    public String getCity() { return  
        cidade;  
    }
```

Listagem 6-7 (continuação)

```
address.java public
String getState() {
    estado de retorno;
}

public String getZip() { return zip;

}
}
```

registro ativo

Active Records são formas especiais de DTOs. São estruturas de dados com variáveis públicas (ou acessadas por bean); mas eles normalmente têm métodos de navegação como salvar e localizar. Normalmente, esses registros ativos são traduções diretas de tabelas de banco de dados ou outros dados fontes.

Infelizmente, muitas vezes descobrimos que os desenvolvedores tentam tratar essas estruturas de dados como se fossem objetos, colocando métodos de regras de negócios nelas. Isso é estranho porque cria um híbrido entre uma estrutura de dados e um objeto.

A solução, claro, é tratar o Active Record como uma estrutura de dados e criar objetos separados que contenham as regras de negócio e que escondam seus dados internos (que provavelmente são apenas instâncias do Active Record).

Conclusão

Os objetos expõem o comportamento e ocultam os dados. Isso facilita a adição de novos tipos de objetos sem alterar os comportamentos existentes. Também torna difícil adicionar novos comportamentos a objetos existentes. As estruturas de dados expõem dados e não têm comportamento significativo. Isso facilita a adição de novos comportamentos às estruturas de dados existentes, mas dificulta a adição de novas estruturas de dados às funções existentes.

Em qualquer sistema, às vezes, desejaremos flexibilidade para adicionar novos tipos de dados e, portanto, preferimos objetos para essa parte do sistema. Outras vezes, desejaremos flexibilidade para adicionar novos comportamentos e, portanto, nessa parte do sistema, preferimos tipos de dados e procedimentos. Bons desenvolvedores de software entendem essas questões sem preconceito e escolhem a abordagem que é melhor para o trabalho em questão.

Bibliografia

[Refactoring]: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

Esta página foi intencionalmente deixada em branco

7

Manipulação de erros

por Michael Feathers



Pode parecer estranho ter uma seção sobre tratamento de erros em um livro sobre código limpo. O tratamento de erros é apenas uma aquela coisas que todos nós temos que fazer quando programamos. A entrada pode ser anormal e os dispositivos podem falhar. Resumindo, as coisas podem dar errado e, quando isso acontecer, nós, como programadores, somos responsáveis por garantir que nosso código faça o que precisa fazer.

A conexão com o código limpo, no entanto, deve ser clara. Muitas bases de código são completamente dominadas pelo tratamento de erros. Quando digo dominado, não quero dizer que o tratamento de erros é tudo o que eles fazem. Quero dizer que é quase impossível ver o que o código faz por causa de todo o tratamento de erros dispersos. O tratamento de erros é importante, *mas se obscurecer a lógica, está errado*.

Neste capítulo, descreverei uma série de técnicas e considerações que você pode usar para escrever um código que seja limpo e robusto — um código que lide com erros com graça e estilo.

Use exceções em vez de códigos de retorno

No passado distante, havia muitos idiomas que não tinham exceções. Nessas línguas, as técnicas para lidar e relatar erros eram limitadas. Você define um sinalizador de erro ou retorna um código de erro que o chamador pode verificar. O código na Listagem 7-1 ilustra essas abordagens.

Listagem

7-1 DeviceController.java public

```
class DeviceController {
    ...
    public void sendShutDown()
    { DeviceHandle handle = getHandle(DEV1); // Verifique o estado do dispositivo if
        (handle != DeviceHandle.INVALID) { // Salve o status do dispositivo no campo de registro
            retrieveDeviceRecord(handle); // Se
            não for suspenso, desliga if
            (record.getStatus() != DEVICE_SUSPENDED)
                { pauseDevice(handle);
                    clearDeviceWorkQueue(manipulador);
                    closeDevice(manipulador); }
            else
                { logger.log("Dispositivo suspenso. Não foi possível desligar");
                } } else
                { logger.log("Manipulação inválida para: " + DEV1.toString());
                }
    }
    ...
}
```

O problema com essas abordagens é que elas sobrecarregam o chamador. O chamador deve verificar se há erros imediatamente após a chamada. Infelizmente, é fácil esquecer. Por esta razão, é melhor lançar uma exceção quando você encontrar um erro. O código de chamada é mais limpo. Sua lógica não é obscurecida pelo tratamento de erros.

A Listagem 7-2 mostra o código depois que escolhemos lançar exceções em métodos que podem detectar erros.

Listagem

7-2 DeviceController.java (com exceções) public class

```
DeviceController {
    ...
    public void sendShutDown() { try
        { tryToShutdown(); }
        catch (DeviceShutDownError e)
            { logger.log(e);
            }
    }
}
```

Listagem 7-2 (continuação)**DeviceController.java (com exceções)**

```

private void tryToShutdown() lança DeviceShutdownError {
    Identificador DeviceHandle = getHandle(DEV1);
    registro DeviceRecord = retrieveDeviceRecord(handle);

    pausaDispositivo(manipulador);
    clearDeviceWorkQueue(manipulador);
    closeDevice(manipulador);
}

private DeviceHandle getHandle(ID do dispositivo) {
    ...
    throw new DeviceShutdownError("Identificador inválido para: " + id.toString());
    ...
}

...
}

```

Observe o quanto é mais limpo. Isso não é apenas uma questão de estética. O código está melhor porque duas preocupações que estavam emaranhadas, o algoritmo para desligamento do dispositivo e tratamento de erros, agora estão separadas. Você pode olhar para cada uma dessas preocupações e entendê-las de forma independente.

Escreva sua declaração Try-Catch-Finally primeiro

Uma das coisas mais interessantes sobre as exceções é que elas *definem um escopo* dentro do seu programa. Ao executar o código na parte try de uma instrução try-catch-finally , você está declarando que a execução pode ser interrompida a qualquer momento e, em seguida, retomada na captura.

De certa forma, os blocos try são como transações. Sua captura deve deixar seu programa em um estado consistente, não importa o que aconteça na tentativa. Por esse motivo, é uma boa prática começar com uma instrução try-catch-finally quando você estiver escrevendo um código que pode gerar exceções. Isso ajuda a definir o que o usuário desse código deve esperar, independentemente do que der errado com o código executado no try.

Vejamos um exemplo. Precisamos escrever algum código que acesse um arquivo e leia alguns objetos serializados.

Começamos com um teste de unidade que mostra que obteremos uma exceção quando o arquivo não existir:

```

@Test(esperado = StorageException.class) public
void retrieveSectionShouldThrowOnInvalidFileName()
{
    sectionStore.retrieveSection("invalid - file");
}

```

O teste nos leva a criar este stub:

```

public List<RecordedGrip> retrieveSection(String sectionName) { // retorno fictício
    até que tenhamos uma implementação real return new
    ArrayList<RecordedGrip>();
}

```

Nosso teste falha porque não lança uma exceção. Em seguida, mudamos nosso implementação para que ele tente acessar um arquivo inválido. Esta operação lança uma exceção:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    tentar {
        FileInputStream stream = new FileInputStream(sectionName) } catch (Exception
    e) {
        lançar novo StorageException("erro de recuperação", e);

    } return new ArrayList<RecordedGrip>();
}
```

Nosso teste passa agora porque capturamos a exceção. Neste ponto, podemos refatorar. Podemos restringir o tipo de exceção que capturamos para corresponder ao tipo que realmente é lançado do construtor FileInputStream : FileNotFoundException:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    tente
    { FileInputStream stream = new FileInputStream(sectionName); stream.close(); }
    catch
    (FileNotFoundException e) { throw new
        StorageException("erro de recuperação", e);

    } return new ArrayList<RecordedGrip>();
}
```

Agora que definimos o escopo com uma estrutura try-catch , podemos usar o TDD para construir o restante da lógica de que precisamos. Essa lógica será adicionada entre a criação do FileInputStream e o fechamento, podendo fingir que nada deu errado.

Tente escrever testes que forcem exceções e, em seguida, adicione comportamento ao seu manipulador para satisfazer seus testes. Isso fará com que você crie primeiro o escopo da transação do bloco try e o ajudará a manter a natureza da transação desse escopo.

Usar exceções não verificadas

O debate acabou. Durante anos, os programadores Java debateram os benefícios e as responsabilidades das exceções verificadas. Quando as exceções verificadas foram introduzidas na primeira versão do Java, elas pareceram uma ótima ideia. A assinatura de cada método listaria todas as exceções que ele poderia passar para seu chamador. Além disso, essas exceções faziam parte do tipo do método. Seu código literalmente não compilaria se a assinatura não correspondesse ao que seu código poderia fazer.

Na época, pensamos que as exceções verificadas eram uma ótima ideia; e sim, eles podem trazer *algum* benefício. No entanto, está claro agora que eles não são necessários para a produção de software robusto. O C# não tem exceções verificadas e, apesar das tentativas valiosas, o C++ também não. Nem Python nem Ruby. No entanto, é possível escrever um software robusto em todas essas linguagens. Como esse é o caso, temos que decidir - realmente - se as exceções verificadas valem seu preço.

Que preço? O preço das exceções verificadas é uma violação do Princípio1 Aberto/Fechado. Se você lançar uma exceção verificada de um método em seu código e o catch estiver três níveis acima, você deve declarar essa exceção na assinatura de cada método entre você e o catch. Isso significa que uma alteração em um nível baixo do software pode forçar alterações de assinatura em muitos níveis superiores. Os módulos alterados devem ser reconstruídos e reimplementados, mesmo que nada com o que eles se preocupem tenha mudado.

Considere a hierarquia de chamada de um sistema grande. Funções no topo chamam funções abaixo delas, que chamam mais funções abaixo delas, ad infinitum. Agora, digamos que uma das funções de nível mais baixo seja modificada de forma que deva gerar uma exceção. Se essa exceção for verificada, a assinatura da função deverá adicionar uma cláusula throws . Mas isso significa que toda função que chama nossa função modificada também deve ser modificada para capturar a nova exceção ou para anexar a cláusula throws apropriada à sua assinatura. Ao infinito. O resultado líquido é uma cascata de mudanças que vão desde os níveis mais baixos do software até os mais altos! O encapsulamento é interrompido porque todas as funções no caminho de um lance devem conhecer os detalhes dessa exceção de baixo nível. Dado que o propósito das exceções é permitir que você lide com os erros à distância, é uma pena que as exceções verificadas quebrem o encapsulamento dessa maneira.

Às vezes, as exceções verificadas podem ser úteis se você estiver escrevendo uma biblioteca crítica: você deve capturá-las. Mas, no desenvolvimento geral de aplicativos, os custos de dependência superam os benefícios.

Forneça contexto com exceções

Cada exceção lançada deve fornecer contexto suficiente para determinar a origem e o local de um erro. Em Java, você pode obter um rastreamento de pilha de qualquer exceção; no entanto, um rastreamento de pilha não pode informar a intenção da operação que falhou.

Crie mensagens de erro informativas e passe-as junto com suas exceções. Mencione a operação que falhou e o tipo de falha. Se você estiver logando em seu aplicativo, passe informações suficientes para poder registrar o erro em seu catch.

Definir classes de exceção em termos das necessidades de um chamador

Existem muitas maneiras de classificar os erros. Podemos classificá-los por sua origem: vieram de um componente ou de outro? Ou seu tipo: são falhas de dispositivo, falhas de rede ou erros de programação? No entanto, quando definimos classes de exceção em um aplicativo, nossa preocupação mais importante deve ser *como elas são capturadas*.

1. [Martinho].

Vejamos um exemplo de classificação de exceção ruim. Aqui está uma instrução try-catch-finally para uma chamada de biblioteca de terceiros. Abrange todas as exceções que as chamadas podem gerar:

```

porta ACMEPort = new ACMEPort(12);

tente
{ port.open(); }
catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Exceção de resposta do dispositivo", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Exceção de desbloqueio", e); }
catch (GMXError e)
{ reportPortError(e);
logger.log("Exceção de resposta do dispositivo"); }
finalmente {
...
}

```

Essa declaração contém muita duplicação e não devemos nos surpreender. Na maioria das situações de tratamento de exceções, o trabalho que fazemos é relativamente padrão, independentemente da causa real. Temos que registrar um erro e garantir que possamos prosseguir.

Neste caso, como sabemos que o trabalho que estamos fazendo é praticamente o mesmo independentemente da exceção, podemos simplificar consideravelmente nosso código envolvendo a API que estamos chamando e garantindo que ela retorne um tipo de exceção comum:

```

porta LocalPort = new LocalPort(12); tente

{ port.open(); }
catch (PortDeviceFailure e) { reportError(e);

    logger.log(e.getMessage(), e); } finalmente
{
...
}

```

Nossa classe LocalPort é apenas um wrapper simples que captura e traduz exceções lançado pela classe ACMEPort :

```

public class LocalPort {
    private ACMEPort innerPort;

    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }

    public void open() { try

        { innerPort.open(); } catch
        (DeviceResponseException e) { lançar novo
            PortDeviceFailure(e); } catch
        (ATM1212UnlockedException e) { lançar novo
            PortDeviceFailure(e); } catch (GMXError
        e) {

```

```

        lançar novo PortDeviceFailure(e);
    }
}
...
}
}

```

Wrappers como o que definimos para ACMEPort podem ser muito úteis. Na verdade, agrupar APIs de terceiros é uma prática recomendada. Ao agrupar uma API de terceiros, você minimiza suas dependências dela: você pode optar por mudar para uma biblioteca diferente no futuro sem muita penalidade. O empacotamento também facilita a simulação de chamadas de terceiros quando você está testando seu próprio código.

Uma vantagem final do empacotamento é que você não está preso às opções de design de API de um determinado fornecedor. Você pode definir uma API com a qual se sinta confortável. No exemplo anterior, definimos um único tipo de exceção para falha de dispositivo de porta e descobrimos que poderíamos escrever um código muito mais limpo.

Freqüentemente, uma única classe de exceção é adequada para uma área específica do código. As informações enviadas com a exceção podem distinguir os erros. Use classes diferentes apenas se houver momentos em que você deseja capturar uma exceção e permitir que a outra passe.

Definir o Fluxo Normal

Se você seguir os conselhos das seções anteriores, acabará com uma boa separação entre sua lógica de negócios e seu tratamento de erros. A maior parte do seu código começará a parecer um algoritmo limpo e sem adornos. No entanto, o processo de fazer isso empurra a detecção de erros para as bordas do seu programa. Você envolve APIs externas para poder lançar suas próprias exceções e define um manipulador acima do seu código para poder lidar com qualquer computação abortada. Na maioria das vezes, essa é uma ótima abordagem, mas há momentos em que você pode não querer abortar.



Vamos dar uma olhada em um exemplo. Aqui está um código estranho que soma despesas em um aplicativo de cobrança:

```

tente
{
    DespesasRefeições = despesasReportDAO.getMeals(employee.getID()); m_total +=
        despesas.getTotal();
}
catch(MealExpensesNotFound e) { m_total
    += getMealPerDiem();
}

```

Nesse negócio, se as refeições são despesas, elas passam a fazer parte do total. Caso contrário, o funcionário recebe uma diária de refeição naquele dia. A exceção atrapalha a lógica.

Não seria melhor se não tivéssemos que lidar com o caso especial? Se não o fizéssemos, nosso código pareceria muito mais simples. Ficaria assim:

```

Gastos com Refeições = GasReportDAO.getMeals(employee.getID()); m_total +=
    despesas.getTotal();

```

Podemos tornar o código tão simples? Acontece que podemos. Podemos alterar o ExpenseReportDAO para que ele sempre retorne um objeto MealExpense . Se não houver despesas com refeições, ele retorna um objeto MealExpense que retorna a *diária* como seu total:

```
public class PerDiemMealExpenses implements MealExpenses { public int
    getTotal() { // retorna o padrão
        da diária
    }
}
```

Isso é chamado de PADRÃO DE CASO ESPECIAL [Fowler]. Você cria uma classe ou configura um objeto para que ele lide com um caso especial para você. Ao fazer isso, o código do cliente não precisa lidar com um comportamento excepcional. Esse comportamento é encapsulado no objeto de caso especial.

Não retorne nulo

Acho que qualquer discussão sobre tratamento de erros deve incluir a menção das coisas que fazemos que convidam a erros. O primeiro da lista está retornando nulo. Não consigo contar o número de aplicativos que vi em que quase todas as outras linhas eram uma verificação de nulo. Aqui está algum código de exemplo:

```
public void registerItem(item item) {
    if (item != null) {
        Registro ItemRegistry = persistentStore.getItemRegistry(); if (registro != null) {

            Item existente = Registry.getItem(item.getItemId()); if
            (existing.getBillingPeriod().hasRetailOwner()) { existing.register(item);

            }
        }
    }
}
```

Se você trabalha em uma base de código com código como este, pode não parecer tão ruim para você, mas é ruim! Quando retornamos nulo, estamos essencialmente criando trabalho para nós mesmos e impingindo problemas aos nossos chamadores. Basta uma verificação nula ausente para enviar um aplicativo girando fora de controle.

Você notou o fato de que não havia uma verificação nula na segunda linha dessa instrução if aninhada ? O que teria acontecido em tempo de execução se persistStore fosse nulo? Teríamos um NullPointerException em tempo de execução e alguém está capturando NullPointerException no nível superior ou não. De qualquer forma é *ruim*. O que exatamente você deve fazer em resposta a uma NullPointerException lançada das profundezas de seu aplicativo?

É fácil dizer que o problema com o código acima é que falta uma verificação nula , mas, na verdade, o problema é que ele tem *muitos*. Se você for tentado a retornar nulo de um método, considere lançar uma exceção ou retornar um objeto SPECIAL CASE . Se você estiver chamando um método de retorno nulo de uma API de terceiros, considere agrupar esse método com um método que lance uma exceção ou retorne um objeto de caso especial.

Em muitos casos, objetos de casos especiais são uma solução fácil. Imagine que você tenha um código assim:

```
List<Empregado> funcionários = getEmpregados(); if
(funcionários != null)
{ for(Funcionário e : funcionários)
    { totalPagamento += e.getPay();
    }
}
```

No momento, `getEmployees` pode retornar nulo, mas precisa? Se alterarmos `getEmployee` para que retorne uma lista vazia, podemos limpar o código:

```
List<Empregado> funcionários = getEmpregados();
for(Empregado e : empregados)
{ totalPay += e.getPay();
}
```

Felizmente, Java tem `Collections.emptyList()` e retorna uma lista imutável predefinida que podemos usar para esse propósito:

```
public List<Empregado> getEmpregados() { if( .. não
há funcionários .. ) return Coleções.emptyList();
}
```

Se você codificar dessa maneira, minimizará a chance de `NullPointerExceptions` e seu código ficará mais limpo.

Não Passe Nulo

Retornar nulo de métodos é ruim, mas passar nulo para métodos é pior. A menos que você esteja trabalhando com uma API que espera que você passe nulo, evite passar nulo em seu código sempre que possível.

Vejamos um exemplo para ver o porquê. Aqui está um método simples que calcula uma métrica para dois pontos:

```
public class MetricsCalculator {

    public double xProjection(Ponto p1, Ponto p2) {
        retornar (p2.x - p1.x) * 1,5;
    }
    ...
}
```

O que acontece quando alguém passa null como argumento?

```
calculadora.xProjection(null, new Point(12, 13));
```

Obteremos um `NullPointerException`, é claro.

Como podemos arranjá-lo? Poderíamos criar um novo tipo de exceção e lançá-lo:

```
public class MetricsCalculator {
```

```

public double xProjection(Ponto p1, Ponto p2) {
    if (p1 == nulo || p2 == nulo) {
        throw InvalidArgumentException( "Argumento
            inválido para MetricsCalculator.xProjection");
    }
    }
}
} 
```

Isso é melhor? Pode ser um pouco melhor do que uma exceção de ponteiro nulo , mas lembre-se, temos que definir um manipulador para InvalidArgumentException. O que o manipulador deve fazer? Existe algum bom curso de ação?

Existe outra alternativa. Poderíamos usar um conjunto de afirmações:

```

public class MetricsCalculator {

    public double xProjection(Point p1, Point p2) { assert p1 != null :
        "p1 não deve ser nulo"; assert p2 != null : "p2 não deve ser
        nulo"; retornar (p2.x – p1.x) * 1,5;
    }
} 
```

É uma boa documentação, mas não resolve o problema. Se alguém passar nulo, ainda teremos um erro de tempo de execução.

Na maioria das linguagens de programação, não há uma boa maneira de lidar com um nulo que é passado por um chamador accidentalmente. Como esse é o caso, a abordagem racional é proibir a passagem de null por padrão. Ao fazer isso, você pode codificar com o conhecimento de que um nulo em uma lista de argumentos é uma indicação de um problema e acabar com muito menos erros por descuido.

Conclusão

O código limpo é legível, mas também deve ser robusto. Estes não são objetivos conflitantes. Podemos escrever um código limpo e robusto se considerarmos o tratamento de erros como uma preocupação separada, algo que pode ser visualizado independentemente de nossa lógica principal. Na medida em que somos capazes de fazer isso, podemos raciocinar sobre isso de forma independente e podemos fazer grandes avanços na manutenção de nosso código.

Bibliografia

[Martin]: Desenvolvimento Ágil de Software: Princípios, Padrões e Práticas, Robert C. Martin, PrenticeHall, 2002.

8

Limites

por James Grenning



Raramente controlamos todo o software em nossos sistemas. Às vezes, compramos pacotes de terceiros ou usamos código aberto. Outras vezes, dependemos de equipes de nossa própria empresa para produzir componentes ou subsistemas para nós. De alguma forma, devemos integrar de forma limpa este código estrangeiro

com os nossos. Neste capítulo, examinamos práticas e técnicas para manter os limites de nosso software limpos.

Usando Código de Terceiros

Existe uma tensão natural entre o provedor de uma interface e o usuário de uma interface.

Os provedores de pacotes e estruturas de terceiros buscam uma ampla aplicabilidade para que possam trabalhar em muitos ambientes e atrair um público amplo. Os usuários, por outro lado, desejam uma interface focada em suas necessidades específicas. Essa tensão pode causar problemas nos limites de nossos sistemas.

Vejamos `java.util.Map` como exemplo. Como você pode ver examinando a Figura 8-1, o `Maps` tem uma interface muito ampla com muitos recursos. Certamente esse poder e flexibilidade são úteis, mas também podem ser um risco. Por exemplo, nosso aplicativo pode criar um mapa e passá-lo adiante. Nossa intenção pode ser que nenhum dos destinatários de nosso Mapa exclua nada do mapa. Mas bem no topo da lista está o método `clear()`. Qualquer usuário do Mapa tem o poder de limpá-lo. Ou talvez nossa convenção de design seja que apenas determinados tipos de objetos podem ser armazenados no Mapa, mas os Mapas não restringem de forma confiável os tipos de objetos colocados dentro deles. Qualquer usuário determinado pode adicionar itens de qualquer tipo a qualquer Mapa.

- `clear() void – Map •`
- `containsKey(Object key) boolean – Map •`
- `containsValue(Object value) boolean – Map • entrySet()`
- `Set – Map • equals(Object o)`
- `boolean – Map • get(Object key) Object –`
- `Map • getClass() Classe<? extends`
- `Object> – Object • hashCode() int – Map • isEmpty() boolean`
- `– Map • keySet() Set – Map •`
- `notify() void – Object • notifyAll()`
- `void – Object • put(Object`
- `key, Object value) Object – Map •`
- `putAll(Map t) void – Map •`
- `remove(Object key) Object – Map • size() int – Map •`
- `toString() String – Object • values()`
- `Collection – Map • wait() void – Object • wait`
- `(long timeout) void –`
- `Objeto • wait(long timeout, int nanos)`
- `void – Objeto`

Figura 8-1
Os métodos do mapa

Se nosso aplicativo precisar de um mapa de sensores, você poderá encontrar os sensores configurados assim:

```
Sensores de mapa = new HashMap();
```

Então, quando alguma outra parte do código precisar acessar o sensor, você verá este código:

```
Sensor s = (Sensor)sensors.get(sensorId );
```

Não o vemos apenas uma vez, mas várias vezes ao longo do código. O cliente deste código carrega a responsabilidade de obter um objeto do mapa e convertê-lo para o tipo certo. Isso funciona, mas não é um código limpo. Além disso, esse código não conta sua história tão bem quanto poderia. A legibilidade deste código pode ser muito melhorada usando genéricos, conforme mostrado abaixo:

```
Map<Sensor> sensores = new HashMap<Sensor>();  
...  
Sensor s = sensores.get(sensorId );
```

No entanto, isso não resolve o problema de que Map<Sensor> fornece mais recursos do que precisamos ou desejamos.

Passar uma instância de Map<Sensor> liberalmente pelo sistema significa que haverá muitos lugares para corrigir se a interface para Map mudar. Você pode achar que essa mudança é improvável, mas lembre-se de que ela mudou quando o suporte a genéricos foi adicionado ao Java 5.

De fato, vimos sistemas que são inibidos de usar genéricos por causa da magnitude das mudanças necessárias para compensar o uso liberal do Maps.

Uma maneira mais limpa de usar o Map pode ser semelhante à seguinte. Nenhum usuário de sensores se importaria nem um pouco se os genéricos fossem usados ou não. Essa escolha se tornou (e sempre deve ser) um detalhe de implementação.

```
sensores de classe pública  
{ sensores de mapa privados = new HashMap();  
  
public Sensor getByld(String id) { return  
    (Sensor)sensors.get(id);  
}  
  
//recorte  
}
```

A interface no limite (Mapa) está oculta. É capaz de evoluir com muito pouco impacto no restante do aplicativo. O uso de genéricos não é mais um grande problema porque o casting e o gerenciamento de tipos são feitos dentro da classe Sensors .

Essa interface também é personalizada e restrita para atender às necessidades do aplicativo. Isso resulta em um código mais fácil de entender e mais difícil de usar indevidamente. A classe Sensors pode impor regras de design e negócios.

Não estamos sugerindo que todos os usos de Map sejam encapsulados dessa forma. Em vez disso, estamos aconselhando você a não passar mapas (ou qualquer outra interface em um limite) em seu sistema. Se você usar uma interface de limite como Map, mantenha-a dentro da classe ou feche a família de classes onde ela é usada. Evite devolvê-lo ou aceitá-lo como um argumento para APIs públicas.

Explorando e aprendendo limites

O código de terceiros nos ajuda a obter mais funcionalidades em menos tempo. Por onde começamos quando queremos utilizar algum pacote de terceiros? Não é nosso trabalho testar o código de terceiros, mas pode ser do nosso interesse escrever testes para o código de terceiros que usamos.

Suponha que não esteja claro como usar nossa biblioteca de terceiros. Podemos passar um ou dois dias (ou mais) lendo a documentação e decidindo como vamos usá-la. Em seguida, podemos escrever nosso código para usar o código de terceiros e ver se ele faz o que pensamos. Não ficaríamos surpresos se nos encontrássemos atolados em longas sessões de depuração tentando descobrir se os bugs que estamos enfrentando estão em nosso código ou no deles.

Aprender o código de terceiros é difícil. Integrar o código de terceiros também é difícil. Fazer as duas coisas ao mesmo tempo é duplamente difícil. E se tivéssemos uma abordagem diferente? Em vez de experimentar e testar as novidades em nosso código de produção, poderíamos escrever alguns testes para explorar nossa compreensão do código de terceiros. Jim Newkirk chama esses testes *de testes de aprendizagem*.¹

Nos testes de aprendizado, chamamos a API de terceiros, pois esperamos usá-la em nosso aplicativo. Estamos essencialmente fazendo experimentos controlados que verificam nossa compreensão dessa API. Os testes se concentram no que queremos da API.

Aprendendo log4j

Digamos que queremos usar o pacote apache log4j em vez de nosso próprio log ger personalizado. Nós o baixamos e abrimos a página de documentação introdutória. Sem muita leitura, escrevemos nosso primeiro caso de teste, esperando que ele escreva “hello” no console.

```
@Test
public void testLogCreate() { Logger
    logger = Logger.getLogger("MyLogger"); logger.info("olá");
}
```

Quando o executamos, o logger produz um erro que nos diz que precisamos de algo chamado Appender. Depois de ler um pouco mais, descobrimos que existe um ConsoleAppender. Portanto, criamos um ConsoleAppender e verificamos se descobrimos os segredos do registro no console.

```
@Test
public void testLogAddAppender() { Logger
    logger = Logger.getLogger("MyLogger"); ConsoleAppender
    appender = new ConsoleAppender(); logger.addAppender(appender);
    logger.info("olá");
}
```

1. [BeckTDD], pp. 136–137.

Desta vez, descobrimos que o Appender não tem fluxo de saída. Estranho - parece lógico que teria um. Depois de uma pequena ajuda do Google, tentamos o seguinte:

```
@Test
public void testLogAddAppender() { Logger
    logger = Logger.getLogger("MyLogger");
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender( new
        PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("olá");
}
```

Isso funcionou; uma mensagem de log que inclui “olá” saiu no console! Parece estranho termos que dizer ao ConsoleAppender que ele grava no console.

Curiosamente, quando removemos o argumento ConsoleAppender.SystemOut , vemos que “hello” ainda é impresso. Mas quando retiramos o PatternLayout, ele mais uma vez reclama da falta de um fluxo de saída. Este é um comportamento muito estranho.

Observando a documentação com um pouco mais de cuidado, vemos que o construtor ConsoleAppender padrão está “desconfigurado”, o que não parece muito óbvio ou útil. Isso parece um bug, ou pelo menos uma inconsistência, no log4j.

Um pouco mais de pesquisa no Google, leitura e teste, e finalmente chegamos à Listagem 8-1. Descobrimos muito sobre o funcionamento do log4j e codificamos esse conhecimento em um conjunto de testes de unidade simples.

Listagem

8-1 LogTest.java

```
public class LogTest
{
    private Logger logger;

    @Before
    public void initialize() { logger =
        Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger()
    {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }

    @Test
    public void addAppenderWithStream()
    {
        logger.addAppender(new ConsoleAppender( new
            PatternLayout("%p %t %m%n"),
            ConsoleAppender.SYSTEM_OUT));
        logger.info("addAppenderWithStream");
    }
}
```

Listagem 8-1 (continuação)**LogTest.java**

```

    @Test
    public void addAppenderWithoutStream()
    {
        logger.addAppender(new ConsoleAppender( new
            PatternLayout("%p %t %m%n")));
        logger.info("addAppenderWithoutStream");
    }
}

```

Agora sabemos como inicializar um registrador de console simples e podemos encapsular esse conhecimento em nossa própria classe de registrador para que o restante de nosso aplicativo seja isolado da interface de limite log4j .

Testes de aprendizado são melhores que gratuitos

Os testes de aprendizado acabam não custando nada. Tínhamos que aprender a API de qualquer maneira, e escrever esses testes foi uma maneira fácil e isolada de obter esse conhecimento. Os testes de aprendizado eram experimentos precisos que ajudavam a aumentar nossa compreensão.

Os testes de aprendizado não são apenas gratuitos, mas também têm um retorno positivo do investimento. Quando há novos lançamentos do pacote de terceiros, fazemos os testes de aprendizado para ver se há diferenças de comportamento.

Os testes de aprendizado verificam se os pacotes de terceiros que estamos usando funcionam da maneira que esperamos. Uma vez integrado, não há garantias de que o código de terceiros permanecerá compatível com nossas necessidades. Os autores originais sofrerão pressões para alterar seu código para atender às suas próprias necessidades. Eles corrigirão bugs e adicionarão novos recursos. Com cada lançamento vem um novo risco. Se o pacote de terceiros mudar de alguma forma incompatível com nossos testes, descobriremos imediatamente.

Quer você precise ou não do aprendizado fornecido pelos testes de aprendizado, um limite limpo deve ser suportado por um conjunto de testes de saída que exercitam a interface da mesma forma que o código de produção. Sem esses *testes de limite* para facilitar a migração, podemos ficar tentados a ficar com a versão antiga por mais tempo do que deveríamos.

Usando código que ainda não existe

Existe outro tipo de limite, aquele que separa o conhecido do desconhecido. Muitas vezes há lugares no código onde nosso conhecimento parece cair fora da borda. Às vezes, o que está do outro lado da fronteira é incognoscível (pelo menos agora). Às vezes, optamos por não olhar além do limite.

Alguns anos atrás, fiz parte de uma equipe de desenvolvimento de software para um sistema de comunicação por rádio. Existia um subsistema, o “Transmissor”, do qual pouco sabíamos, e os responsáveis pelo subsistema não haviam chegado ao ponto de definir sua interface. Não queríamos ser bloqueados, então começamos nosso trabalho longe da parte desconhecida do código.

Tínhamos uma boa ideia de onde nosso mundo terminava e o novo mundo começava. Enquanto trabalhávamos, às vezes esbarrávamos nesse limite. Embora névoas e nuvens de ignorância obscurecessem nossa visão além da fronteira, nosso trabalho nos tornou conscientes do que queríamos que fosse a interface da fronteira. Queríamos dizer ao transmissor algo assim:

Chaveie o transmissor na frequência fornecida e emita uma representação analógica dos dados provenientes desse fluxo.

Não tínhamos ideia de como isso seria feito porque a API ainda havia sido projetada. Então decidimos acertar os detalhes mais tarde.

Para não ser bloqueado, definimos nossa própria interface. Nós o chamamos de algo cativante, como Transmitter. Demos a ele um método chamado transmit que pega uma frequência e um fluxo de dados. Essa era a interface que *gostaríamos* de ter.

Uma coisa boa sobre escrever a interface que gostaríamos de ter é que ela está sob nosso controle. Isso ajuda a manter o código do cliente mais legível e focado no que ele está tentando realizar.

Na Figura 8-2, você pode ver que isolamos as classes CommunicationsController da API do transmissor (que estava fora de nosso controle e indefinida). Ao usar nossa própria interface específica de aplicativo, mantivemos nosso código CommunicationsController limpo e expressivo. Depois que a API do transmissor foi definida, escrevemos o TransmitterAdapter para preencher a lacuna. O ADAPTER2 encapsula a interação com a API e fornece um único local para alterar quando a API evolui.

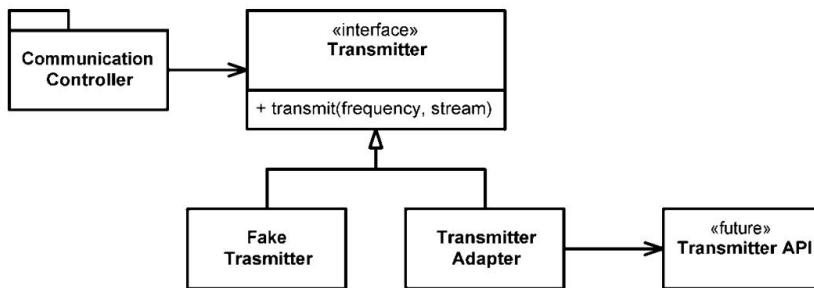


Figura 8-2
Prevendo o transmissor

Esse design também nos fornece um costura³ muito conveniente no código para teste. Usando um FakeTransmitter adequado, podemos testar as classes CommunicationsController . Também podemos criar testes de limite assim que tivermos o TransmitterAPI que garante que estamos usando a API corretamente.

2. Veja o padrão Adaptador em [GOF].

3. Veja mais sobre costuras em [WELC].