

# TDD DESENVOLVIMENTO GUIADO POR TESTES

---

KENT BECK





# Sobre o autor

Um dos mais criativos e aclamados líderes da indústria de software, Kent Beck passionalmente emprega padrões, XP e TDD. É filiado ao Three Rivers Institute, ele é o autor dos livros *Smalltalk Best Practice Patterns* (Prentice Hall, 1997), *Programação Extrema Explicada* (Bookman, 2004) e *Planning Extreme Programming* (com Martin Fowler, Addison-Wesley, 2001), e colaborador em *Refatoração* (de Martin Fowler, Bookman, 2004).



B393t Beck, Kent.

TDD desenvolvimento guiado por testes [recurso eletrônico]  
/ Kent Beck ; tradução: Jean Felipe Patkowski Cheiran ; revisão  
técnica: Marcelo Soares Pimenta. – Dados eletrônicos. – Porto  
Alegre : Bookman, 2010.

Editado também como livro impresso em 2010.  
ISBN 978-85-7780-747-5

1. Ciência da computação. 2. TDD – Técnica de  
desenvolvimento de software. I. Título.

CDU 004.4'2

Catalogação na publicação: Ana Paula M. Magnus – CRB-10/Prov-009/10

**KENT BECK**

**TDD  
DESENVOLVIMENTO  
GUIADO POR TESTES**

**Tradução:**

Jean Felipe Patikowski Cheiran

**Consultoria, supervisão e revisão técnica desta edição:**

Marcelo Soares Pimenta

Doutor em Informática pela Université Toulouse1/França  
Professor do Departamento de Informática/UFRGS

Versão impressa  
desta obra: 2010



2010

Obra originalmente publicada sob o título *Test Driven Development: By Example*, 1st Edition  
ISBN 978-0-321-14653-3

Authorized translation from the English language edition, entitled TEST DRIVEN DEVELOPMENT: BY EXAMPLE, 1st Edition by KENT BECK, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2003. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Portuguese language edition published by Bookman Companhia Editora Ltda, a Division of Artmed Editora SA, Copyright © 2010.

Tradução autorizada a partir do original em língua inglesa da obra intitulada TEST DRIVEN DEVELOPMENT: BY EXAMPLE, 1<sup>a</sup> Edição de autoria de KENT BECK, publicado por Pearson Education, Inc., sob o selo Addison-Wesley Professional, Copyright © 2003. Todos os direitos reservados. Este livro não poderá ser reproduzido nem em parte nem na íntegra, nem ter partes ou sua íntegra armazenado em qualquer meio, seja mecânico ou eletrônico, inclusive fotoreprografia, sem permissão da Pearson Education, Inc.

A edição em língua portuguesa desta obra é publicada por Bookman Companhia Editora Ltda, uma divisão da Artmed Editora SA, Copyright © 2010.

Capa: Rogério Grilho (arte sobre capa original)

Preparação de original: Leonardo Zilio

Editora Sênior: Denise Weber Nowaczyk

Projeto e editoração: Techbooks

Reservados todos os direitos de publicação, em língua portuguesa, à  
ARTMED® EDITORA S.A.

(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S.A.)

Av. Jerônimo de Ornelas, 670 - Santana

90040-340 Porto Alegre RS

Fone (51) 3027-7000 Fax (51) 3027-7070

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

#### SÃO PAULO

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

SAC 0800 703-3444

IMPRESSO NO BRASIL  
PRINTED IN BRAZIL

*Para Cindee: suas próprias asas*

# Agradecimentos

Obrigado a todos os meus muitos revisores poderosos. Eu assumo total responsabilidade pelo conteúdo, mas este livro teria sido muito menos legível e muito menos útil sem a ajuda deles. Na ordem que os digitei, eles são: Steve Freeman, Frank Westphal, Ron Jeffries, Dierk König, Edward Hieatt, Tammo Freese, Jim Newkirk, Johannes Link, Manfred Lange, Steve Hayes, Alan Francis, Jonathan Rasmussen, Shane Clauson, Simon Crase, Kay Pentecost, Murray Bishop, Ryan King, Bill Wake, Edmund Schweppe, Kevin Lawrence, John Carter, Phlip, Peter Hansen, Ben Schroeder, Alex Chaffee, Peter van Rooijen, Rick Kawala, Mark van Hamersveld, Doug Swartz, Laurent Bossavit, Ilja Preuß, Daniel Le Berre, Frank Carver, Justin Sampson, Mike Clark, Christian Pekeler, Karl Scotland, Carl Manaster, J. B. Rainsberger, Peter Lindberg, Darach Ennis, Kyle Cordes, Justin Sampson, Patrick Logan, Darren Hobbs, Aaron Sansone, Syver Enstad, Shinobu Kawai, Erik Meade, Patrick Logan, Dan Rawsthorne, Bill Rutiser, Eric Herman, Paul Chisholm, Asim Jalis, Ivan Moore, Levi Purvis, Rick Mugridge, Anthony Adachi, Nigel Thorne, John Bley, Kari Hoijarvi, Manuel Amago, Kaoru Hosokawa, Pat Eyler, Ross Shaw, Sam Gentle, Jean Rajotte, Phillip Antras, Jaime Nino e Guy Tremblay.

A todos os programadores com quem tenho codificado guiado por testes, eu certamente aprecio sua paciência compactuando com o que era uma ideia que soava maluca, especialmente nos primeiros anos. Eu aprendi muito mais com todos vocês do que eu poderia pensar. Não querendo ofender os demais, mas Massimo Arnoldi, Ralph Beattie, Ron Jeffries, Martin Fowler e, por último, mas certamente não menos importante, Erich Gamma se destacam em minha memória como condutores de testes com os quais aprendi muito.

Eu gostaria de agradecer a Martin Fowler pela ajuda oportuna do Frame-Maker. Ele deve ser o consultor tipográfico mais bem pago do planeta, mas felizmente me deixou (até agora) pendurar a conta.

Minha vida como um programador de verdade começou como discípulo e colaborador de Ward Cunningham. Às vezes, eu vejo Desenvolvimento Guiado por Testes (TDD) como uma tentativa de dar a qualquer engenheiro de software,

trabalhando em qualquer ambiente, a sensação de conforto e intimidade que tivemos com nosso ambiente Smalltalk e nossos programas Smalltalk. Não há formas de separar a fonte de ideias, uma vez que duas pessoas dividam um cérebro. Se você assume que todas as boas ideias aqui são do Ward, então não estará muito errado.

É um pouco clichê reconhecer o sacrifício de uma família quando um dos seus membros assume a peculiar aflição mental que resulta em um livro. Isso porque os sacrifícios da família são tão necessários para a escrita de um livro como é o papel. Às minhas crianças, que esperaram o café da manhã até que eu terminasse um capítulo e, acima de tudo, à minha esposa que passou dois meses repetindo tudo três vezes, meus mais profundos e adequados agradecimentos.

Obrigado a Mike Henderson pelo encorajamento gentil e a Marcy Barnes por correr para me socorrer.

Finalmente, ao autor desconhecido de um livro que li quando era um garoto esquisito de 12 anos, que sugeriu digitar na fita de saída esperada de uma fita de entrada real, então codificar até que os resultados reais correspondessem ao resultado esperado, obrigado, obrigado, obrigado.

# Prefácio

*Código limpo que funciona*, em uma frase concisa de Ron Jeffries, é o objetivo do Desenvolvimento Guiado por Testes (TDD). Código limpo que funciona é uma meta valiosa por um bocado de razões.

- É uma forma previsível de desenvolver. Você sabe quando acabou sem ter que se preocupar com uma longa trilha de erros.
- Dá a você uma chance de aprender todas as lições que o código tem para ensinar. Se você apenas fizer às pressas a primeira coisa que pensar, então nunca terá tempo para pensar em uma segunda coisa melhor.
- Melhora as vidas dos usuários de seu software.
- Permite que seus colegas de equipe contem com você, e você com eles.
- É bom escrevê-lo.

Mas como obtemos código limpo que funciona? Muitas forças nos desviam de código limpo, ou mesmo de código que funciona. Sem pedir conselhos aos nossos medos, aqui está o que fazemos: conduzimos o desenvolvimento com testes automatizados, um estilo de desenvolvimento chamado Desenvolvimento Guiado por Testes (TDD). No Desenvolvimento Guiado por Testes,

- Escrevemos código novo apenas se um teste automatizado falhou
- Eliminamos duplicação

Essas são duas regras simples, mas geram um complexo comportamento individual e de grupo com implicações técnicas, tais como:

- Devemos projetar organicamente com código, executando e fornecendo feedback entre as decisões.

- Devemos escrever nossos próprios testes, pois não podemos esperar 20 vezes por dia para outra pessoa escrever um teste.
- Nosso ambiente de desenvolvimento deve fornecer resposta rápida a pequenas mudanças.
- Nosso projeto deve consistir em muitos componentes altamente coesos e fracamente acoplados para tornar os testes fáceis.

As duas regras implicam em uma ordem para as tarefas de programação.

1. Vermelho – Escrever um pequeno teste que não funcione e que talvez nem mesmo compile inicialmente.
2. Verde – Fazer rapidamente o teste funcionar, mesmo cometendo algum pecado necessário no processo.
3. Refatorar – Eliminar todas as duplicatas criadas apenas para que o teste funcione.

Vermelho/verde/refatorar – o mantra do TDD.

Supondo por um momento que tal estilo de programação é possível, ainda pode-se reduzir significativamente a densidade de defeitos de código e fazer o tema de trabalho claro como cristal para todos os envolvidos. Se assim for, então escrever apenas aquele código que é requerido por testes que falham também tem implicações sociais.

- Se a densidade de defeitos pode ser suficientemente reduzida, então a garantia da qualidade (Quality Assurance – QA) pode mudar de trabalho reativo para trabalho pró-ativo.
- Se o número de surpresas desagradáveis pode ser suficientemente reduzido, então os gerentes de projeto podem estimar de forma precisa o bastante para envolver clientes reais no desenvolvimento diário.
- Se os tópicos das conversas técnicas passam a ser suficientemente claros, então os engenheiros de software podem trabalhar em uma colaboração que acontece minuto-a-minuto em vez de diária ou semanalmente.
- De novo, se a densidade de defeitos pode ser suficientemente reduzida, então podemos ter software pronto com novas funcionalidades a cada dia, levando a novos relacionamentos de negócios com clientes.

Então o conceito é simples, mas qual é minha motivação? Por que um engenheiro de software teria o trabalho adicional de escrever testes automatizados? Por que um engenheiro de software trabalharia em passos tão pequenos quando sua cabeça é capaz de grandes feitos? Coragem.

## Coragem

Desenvolvimento Guiado por Testes é uma forma de administrar o medo durante a programação. Não quero falar de medo de uma forma ruim – *poor widdle prwo-gwammew needs a pacifier*\* –, mas medo no sentido legítimo de esse-é-um-problema-difícil-e-eu-não-consigo-ver-o-fim-a-partir-do-começo. Se dor é o jeito da natureza dizer “Pare！”, então medo é o jeito da natureza dizer “tome cuidado”. Ser cuidadoso é bom, mas o medo tem uma série de outros efeitos.

- O medo o faz hesitante.
- O medo o faz querer comunicar-se menos.
- O medo o faz afastar-se do feedback.
- O medo o faz mal-humorado.

Nenhum desses efeitos é útil quando programamos, especialmente quando programamos algo difícil. Então a questão se torna como nós encaramos uma situação difícil e,

- Em vez de sermos hesitantes, começarmos a aprender concretamente tão rápido quanto possível.
- Em vez de nos calarmos, comunicar mais claramente.
- Em vez de evitar o feedback, procurar feedback útil e concreto.
- (Você terá de trabalhar seu mal-humor sozinho.)

Imagine que programar é como girar uma manivela para puxar um balde com água de um poço. Quando o balde é pequeno, uma manivela sem catraca é ótima. Quando o balde é grande e está cheio de água, você ficará cansado antes do balde ser completamente puxado. Você precisa de um mecanismo de catracas para descansar entre as séries de maniveladas. Quanto mais pesado é o balde, mais próximos os dentes da catraca precisam estar.

Os testes no desenvolvimento guiado por testes são os dentes da catraca. Uma vez que tenhamos um teste funcionando, sabemos que está funcionando agora e para sempre. Estamos um passo mais próximo de termos tudo funcionando do que estávamos quando o teste não funcionava. Agora fazemos o próximo funcionar, e o próximo, e o próximo. Por analogia, quanto mais árduo é o problema de programação, menor deve ser o terreno que cada teste deve cobrir.

Leitores do meu livro *Programação Extrema Explicada* notarão a diferença de tom entre Programação Extrema (XP) e TDD. TDD não é absoluta do jeito que XP é. XP diz: “Aqui estão coisas que você deve ser capaz de fazer para estar

\* N. de R. T.: Uma descontração do autor, a frase “*poor little programmer needs a pacifier*” está escrita no texto conforme o sotaque e modo de falar do Hortelino Troca-Letras (Elmer Fudd, em inglês), personagem arqui-inimigo do Pernalonga no desenho animado da Looney Tunes. Na versão em inglês do desenho, Hortelino fala “Wabbit” em vez de o correto “Rabbit”.

preparado para evoluir”. TDD é um pouco nebuloso. TDD é uma consciência da lacuna entre decisão e feedback durante a programação, e técnicas para controlar essa lacuna. “E se eu fizer um projeto em papel por uma semana, e então testar o código? Isso é TDD?” Claro, é TDD. Você estava ciente da lacuna entre decisão e feedback e controlou a lacuna deliberadamente.

Dito isso, a maioria das pessoas que aprendem TDD acham que suas práticas de programação mudaram para melhor. *Test Infected*\* é a frase que Erich Gamma cunhou para descrever essa mudança. Você pode se encontrar escrevendo testes mais cedo e trabalhando em passos menores como jamais sonhou ser sensato. Por outro lado, alguns engenheiros de software aprendem TDD e então voltam a suas práticas anteriores, reservando TDD para ocasiões especiais, naquelas em que a programação comum não está fazendo progressos.

Certamente, há tarefas de programação que não podem ser guiadas somente por testes (ou, ao menos, não ainda). Segurança de software e concorrência, por exemplo, são dois tópicos em que TDD é insuficiente para demonstrar mecanicamente que as metas do software foram alcançadas. Embora seja verdade que segurança depende essencialmente de código livre de defeitos, ela também depende do julgamento humano sobre os métodos usados para tornar o software mais seguro. Problemas sutis de concorrência não podem ser reproduzidos de forma confiável através da execução do código.

Depois de terminar de ler este livro, você estará pronto para

- Começar de forma simples
- Escrever testes automatizados
- Refatorar para adicionar uma decisão de projeto por vez

Este livro é organizado em três partes:

- Parte I, O Exemplo Financeiro – Um exemplo de código modelo tipicamente escrito usando TDD. O exemplo é um dos que eu obtive tempos atrás com Ward Cunningham e que tenho usado muitas vezes desde então: aritmética multi-moeda. Esse exemplo permitirá que você aprenda a escrever testes antes de código e desenvolver um projeto organicamente.
- Parte II, O Exemplo xUnit – Um exemplo de teste mais complicado do ponto de vista lógico, incluindo reflexão e exceções, no desenvolvimento de um framework para testes automatizados. Esse exemplo também o introduzirá à arquitetura xUnit, que está no coração de muitas ferramentas de teste orientadas ao programador. No segundo exemplo, você aprenderá a trabalhar em passos ainda menores do que no primeiro exemplo, incluindo o tipo de perturbação\*\* amado pelos cientistas da computação.

---

\* N. de R. T.: No Brasil, a expressão adequada poderia ser “Mordido pelo Bicho do Teste”.

\*\* N. de R. T.: No original é usada a expressão informal “hoo-ha” que significa tumulto, desordem, distúrbio e que preferimos traduzir por perturbação.

- Parte III, Padrões para Desenvolvimento Guiado por Testes – Estão incluídos padrões para decidir que testes escrever, como escrever testes usando xUnit, e uma seleção de grandes sucessos dos padrões de projeto e refatoração usados nos exemplos.

Eu escrevi os exemplos imaginando uma sessão de programação em pares. Se você gosta de olhar o mapa antes de sair andando, então pode querer ir direto aos padrões na Parte III e usar os exemplos como ilustrações. Se você prefere apenas sair andando e então olhar o mapa para ver aonde foi, então tente ler através dos exemplos, consultando os padrões quando quiser mais detalhes sobre uma técnica e usando os padrões como uma referência. Muitos revisores deste livro comentaram que obtiveram maior proveito dos exemplos quando abriram um ambiente de programação, entraram com o código e rodaram os testes como leram.

Uma observação sobre os exemplos. Os exemplos cálculo multi-moeda e um framework de teste parecem simples. Existem (e eu já vi) formas complicadas, feias e confusas de resolver os mesmos problemas. Eu poderia ter escolhido uma dessas soluções complicadas, feias e confusas para dar ao livro um ar de “realidade”. Entretanto, minha meta, e eu espero que a sua também, é escrever código limpo que funcione. Antes de rotular os exemplos como sendo simples demais, gaste 15 segundos imaginando um mundo da programação no qual todos os códigos fossem limpos e diretos daquele jeito, onde não houvessem soluções complicadas, apenas problemas aparentemente complicados implorando por uma consideração cuidadosa. TDD pode levá-lo exatamente até essa consideração cuidadosa.

# Sumário

INTRODUÇÃO 17

<b>PARTE I O Exemplo Financeiro .....</b>	<b>21</b>
CAPÍTULO 1 Dinheiro Multi-Moeda .....	23
CAPÍTULO 2 Degenerar Objetos.....	31
CAPÍTULO 3 Igualdade para Todos .....	35
CAPÍTULO 4 Privacidade .....	39
CAPÍTULO 5 Falando Franca-mente.....	43
CAPÍTULO 6 Igualdade para Todos, Restaurada.....	47
CAPÍTULO 7 Maçãs e Laranjas.....	53
CAPÍTULO 8 Fazendo Objetos .....	55
CAPÍTULO 9 Tempos em que Estamos Vivendo .....	59
CAPÍTULO 10 Tempos Interessantes.....	65
CAPÍTULO 11 A Raiz de Todo o Mal .....	71
CAPÍTULO 12 Adição, Finalmente .....	75
CAPÍTULO 13 Faça-o .....	81
CAPÍTULO 14 Mudança.....	87

CAPÍTULO 15	Moedas Misturadas.....	93
CAPÍTULO 16	Abstração, Finalmente.....	97
CAPÍTULO 17	Retrospectiva Financeira .....	101
<b>PARTE II</b>	<b>O Exemplo xUnit .....</b>	<b>109</b>
CAPÍTULO 18	Primeiros Passos para xUnit .....	111
CAPÍTULO 19	Iniciando.....	117
CAPÍTULO 20	Limpando em Seguida .....	121
CAPÍTULO 21	Contagem .....	125
CAPÍTULO 22	Lidando com Falha .....	129
CAPÍTULO 23	Como é a Suíte?.....	133
CAPÍTULO 24	Retrospectiva xUnit.....	139
<b>PARTE III</b>	<b>Padrões para Desenvolvimento Guiado por Testes .....</b>	<b>141</b>
CAPÍTULO 25	Padrões de Desenvolvimento Guiado por Testes .....	143
CAPÍTULO 26	Padrões de Barra Vermelha.....	153
CAPÍTULO 27	Padrões de Teste .....	163
CAPÍTULO 28	Padrões de Barra Verde.....	171
CAPÍTULO 29	Padrões xUnit.....	177
CAPÍTULO 30	Padrões de Projeto .....	185
CAPÍTULO 31	Refatoração .....	201
CAPÍTULO 32	Dominando TDD .....	213
APÊNDICE I	Diagramas de Influência .....	227
APÊNDICE II	Fibonacci .....	231
POSFÁCIO	235	
ÍNDICE	237	

# Introdução

Começo de uma sexta-feira, o chefe de Ward Cunningham procurou-o para apresentar Peter, um cliente potencial para WyCash, o sistema de gerenciamento de investimentos de títulos que a companhia estava vendendo. Peter disse: “Estou muito impressionado com a funcionalidade que eu vejo. Entretanto, percebi que vocês lidam apenas com títulos em dólar americano. Estou começando um novo fundo de títulos e minha estratégia requer que eu lide com títulos em moedas diferentes”. O chefe virou para Ward: “Bem, nós podemos fazer isso?”

Aqui está o cenário de um pesadelo para qualquer projetista de software. Você estava seguindo feliz e bem-sucedido com um conjunto de suposições. De repente, tudo mudou. E o pesadelo não é apenas do Ward. O chefe, uma pessoa experiente em desenvolvimento de software, não tinha certeza de qual resposta viria.

Um pequeno time havia desenvolvido WyCash ao longo de dois anos. O sistema estava apto a lidar com a maioria dos tipos de títulos de renda fixa comumente encontradas no mercado norte-americano e também com alguns dos novos instrumentos, como Contratos de Investimento Garantido, com o qual os competidores não lidavam.

WyCash tinha sido desenvolvido usando objetos e um banco de dados de objetos. No começo, a abstração fundamental da computação, `Dollar`, fora terceirizada a um grupo qualificado de engenheiros de software. O objeto resultante combinava responsabilidades de formatação e cálculo.

Nos últimos seis meses, Ward e o resto do time tinham lentamente descaracterizado `Dollar` de suas responsabilidades. As classes numéricas de Smalltalk acabaram por ser apenas um peso no cálculo. Todo o código complicado para arredondamento para três dígitos decimais começou como uma forma de produzir respostas precisas. Quando as respostas se tornaram mais precisas, os mecanismos complicados no framework de teste para comparação dentro de uma certa tolerância foram substituídos por uma correspondência exata de resultados esperados e reais.

A responsabilidade por formatação pertencia de fato às classes de interface com usuário. Como os testes foram escritos no nível das classes de interface com

usuário, em particular no framework de relatório,<sup>1</sup> esses testes não tinham de ser alterados para contemplar tal refinamento. Depois de seis meses de poda cuidadosa, não tinham sido deixadas muitas responsabilidades para Dollar.

Um dos algoritmos mais complicados do sistema, média ponderada, também tinha sofrido uma lenta transformação e havia muitas variações de código de média ponderada espalhadas por todo o sistema. Como o framework de relatório formou-se de um conjunto inicial de objetos, era óbvio que haveria um lugar para o algoritmo em `AveragedColumn`.

Então, foi para `AveragedColumn` que Ward se voltou. Se médias ponderadas pudessem ser multi-moeda, então o resto do sistema deveria ser possível. No coração do algoritmo era mantida uma contagem de dinheiro na coluna. De fato, o algoritmo era abstrato o suficiente para calcular a média de qualquer objeto que pudesse trabalhar aritmeticamente. Tornou-se possível ter médias ponderadas de datas, por exemplo.

O final de semana passou com as atividades habituais de finais de semana. Na segunda-feira pela manhã, o chefe estava de volta. “Nós podemos fazer isso?”

“Dê-me mais um dia, e eu falarei com certeza.”

Dollar agiu como um contador na média ponderada; portanto, a fim de calcular em múltiplas moedas, eles precisavam de um objeto com um contador por moeda, como uma espécie de polinômio. Em vez de  $3x^2$  e  $4y^3$ , entretanto, os termos seriam 15 USD e 200 CHF.

Um experimento rápido mostrou que era possível computar com um objeto `Currency` (moeda) genérico em vez de um `Dollar`, e retornar um `PolyCurrency` quando duas moedas diferentes eram adicionadas juntas. O truque, agora, era criar espaço para a nova funcionalidade sem estragar nada que já estivesse funcionando. O que aconteceria se Ward apenas rodasse os testes?

Depois da adição de algumas operações não implementadas para `Currency`, a maior parte dos testes passou. No final do dia, todos os testes haviam passado. Ward verificou o código existente e foi até o chefe. “Nós podemos fazer,” ele disse confiante.

Vamos pensar um pouco sobre essa história. Em dois dias, o mercado potencial foi multiplicado muitas vezes, multiplicando o valor de WyCash muitas vezes. Contudo, a habilidade para criar tanto valor comercial tão rápido não foi acidente. Muitos fatores entraram em jogo.

- Método – Ward e o time WyCash precisaram ter, pouco a pouco, constante crescimento de experiência no projeto do sistema, de modo que a mecânica da transformação foi bem feita.
- Motivo – Ward e o time precisaram entender claramente a importância do negócio de fazer WyCash multi-moeda e ter a coragem para começar essa tarefa aparentemente impossível.
- Oportunidade – A combinação de uma geração de testes amplos e confiáveis, um programa bem fatorado e uma linguagem de programação

<sup>1</sup> Para mais detalhes sobre o framework de relatório, consulte [c2.com/doc/oopsla91.html](http://c2.com/doc/oopsla91.html).

que torna possível isolar decisões de projeto significou que houve poucas fontes de erro e esses erros foram fáceis de identificar.

Você não pode controlar o motivo de ter de multiplicar o valor de seu projeto tecendo técnicas mágicas. Método e oportunidade, por outro lado, estão inteiramente sob seu controle. Ward e seu time criaram método e oportunidade por meio da combinação de talento, experiência e disciplina. Isso significa que se você não é um dos dez melhores engenheiros de software do planeta e não tem um maço de dinheiro no banco de modo a poder dizer para seu chefe dar uma caminhada para ter tempo suficiente para fazer as coisas direito, tais momentos estão além do seu alcance para sempre?

Não. É absolutamente possível colocar seus projetos em uma posição para que você trabalhe magicamente, mesmo sendo um engenheiro de software com habilidades comuns e, às vezes, sendo relutante e pegando atalhos quando a pressão aumenta. Desenvolvimento guiado por testes é um conjunto de técnicas que qualquer engenheiro de software pode seguir, que encoraja projetos simples e conjuntos de testes que inspiram confiança. Se você é um gênio, não precisa dessas regras. Se você é um idiota, as regras não vão ajudar. Para a grande maioria de nós entre esses dois extremos, seguir essas duas regras simples pode nos levar a trabalhar muito mais perto de nosso potencial.

- Escreva um teste automático que falhe antes de escrever qualquer código.
- Remova duplicação.

Como exatamente fazer isso, os estágios sutis na aplicação dessas regras e o quanto você pode ampliar os limites da aplicação dessas duas regras simples são o tópico deste livro. Nós começaremos com o objeto que Ward criou nesse momento de inspiração – dinheiro multi-moeda (multi-currency money).

# PARTE I

---

## O Exemplo Financeiro

Na Parte I, desenvolveremos um modelo de código típico e completamente guiado por testes (exceto quando cometemos alguns deslizes, puramente para fins educacionais). Minha meta é que você veja o ritmo do Desenvolvimento Guiado por Testes (TDD), o qual pode ser resumido como segue.

1. Adicionar um teste rapidamente.
2. Rodar todos os testes e ver o mais novo falhando.
3. Fazer uma pequena mudança.
4. Rodar todos os testes e ver todos funcionando.
5. Refatorar para remover duplicações.

As prováveis surpresas incluem:

- Como cada teste pode cobrir um pequeno aumento de funcionalidade
- Quão pequenas e feias as mudanças podem ser para fazer os novos testes rodarem
- Com frequência os testes são executados
- De quantos pequeninos passos as refatorações são compostas

# CAPÍTULO 1

---

## Dinheiro Multi-Moeda

Começaremos com o objeto que Ward criou no WyCash, dinheiro multi-moeda (mencionado na Introdução). Suponha que tenhamos um relatório como esse:

Instrumento	Ações	Preço	Total
IBM	1.000	25	25.000
GE	400	100	40.000
Total			65.000

Para fazer um relatório multi-moeda, precisamos adicionar moedas:

Instrumento	Ações	Preço	Total
IBM	1.000	25 USD	25.000 USD
Novartis	400	150 CHF	60.000 CHF
Total			65.000 USD

Precisamos também especificar taxas de câmbio:

De	Para	Taxa
CHF	USD	1,5

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1  
 $\$5 * 2 = \$10$

De que comportamento precisaremos para produzir o relatório revisado? Dito de outra forma, qual conjunto de testes, quando passarem, demonstrará a presença de código que estamos confiantes que irá calcular o relatório corretamente?

- Precisamos ser capazes de somar valores em duas moedas diferentes e de converter o resultado, dado um conjunto de taxas de câmbio.
- Precisamos ser capazes de multiplicar um valor (preço por ação) por um número (número de ações) e de receber uma quantia.

Faremos uma lista de tarefas para lembrarmos o que precisamos fazer para mantermos o foco e para dizer quando acabarmos. Quando começarmos a trabalhar em um item, nós o marcaremos em negrito, assim. Quando terminarmos um item, o riscaremos, assim. Quando pensarmos em outro teste para escrever, o adicionaremos à lista.

Como você pode ver, a partir da lista de tarefas na página anterior, trabalharemos, primeiro, na multiplicação. Então, de qual objeto precisamos primeiro? Uma questão traiçoeira. Nós não começamos com objetos, começamos com testes. (Continuo tendo de me lembrar disso, então vou fingir que você é tão devagar quanto eu.)

Tente novamente. De qual teste precisamos primeiro? Observando a lista, aquele primeiro teste parece complicado. Comece devagar ou não comece. Multiplicação, quão difícil isso pode ser? Trabalharemos nisso primeiro.

Quando escrevemos um teste, imaginamos uma interface perfeita para nossa operação. Estamos contando para nós mesmos como a operação se parecerá do lado de fora. Nossa história nem sempre se torna verdadeira, mas é melhor começar da melhor Interface de Programação de Aplicativos (API) possível e trabalhar ao contrário para fazer coisas complicadas, feias e “realistas” desde o começo.

Aqui está um exemplo simples de multiplicação:

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);  
    five.times(2);  
    assertEquals(10, five.amount);  
}
```

(Eu sei, eu sei, campos públicos, efeitos colaterais, inteiros para quantias monetárias e tudo isso. Pequenos passos. Tomaremos nota desse mau-cheiro e vamos em frente. Temos um teste falhando e queremos a barra no verde tão logo quanto possível.)

\$5 + 10 CHF = \$10 se a taxa é 2:1

\$5 \* 2 = \$10

Tornar “quantidade” privada

Efeitos colaterais em Dollar?

Arredondamento de dinheiro?

O teste que acabamos de digitar nem mesmo compila. (Explicarei onde e como digitá-lo mais tarde, quando falarmos sobre o framework de testes, JUnit.) Isso é fácil de corrigir. Qual é o mínimo que podemos fazer para tê-lo compilando, mesmo se ele não rodar? Temos quatro erros de compilação:

- Sem classe `Dollar`
- Sem construtor
- Sem método `times(int)`
- Sem campo `amount`

Vamos cuidar deles um de cada vez. (Sempre procuro por alguma medida numérica de progresso.) Podemos eliminar um erro definindo a classe `Dollar`:

### **Dollar**

```
class Dollar
```

Um erro a menos, três erros sobrando. Agora precisamos de um construtor, mas ele não tem que fazer coisa alguma; apenas fazer o teste compilar:

### **Dollar**

```
Dollar(int amount) {  
}
```

Faltam dois erros. Precisamos de uma implementação de um stub\* para `times()`. De novo, faremos o mínimo de trabalho possível apenas para ter o teste compilando:

### **Dollar**

```
void times(int multiplier) {  
}
```

Falta um erro. Finalmente, precisamos de um campo `amount`:

### **Dollar**

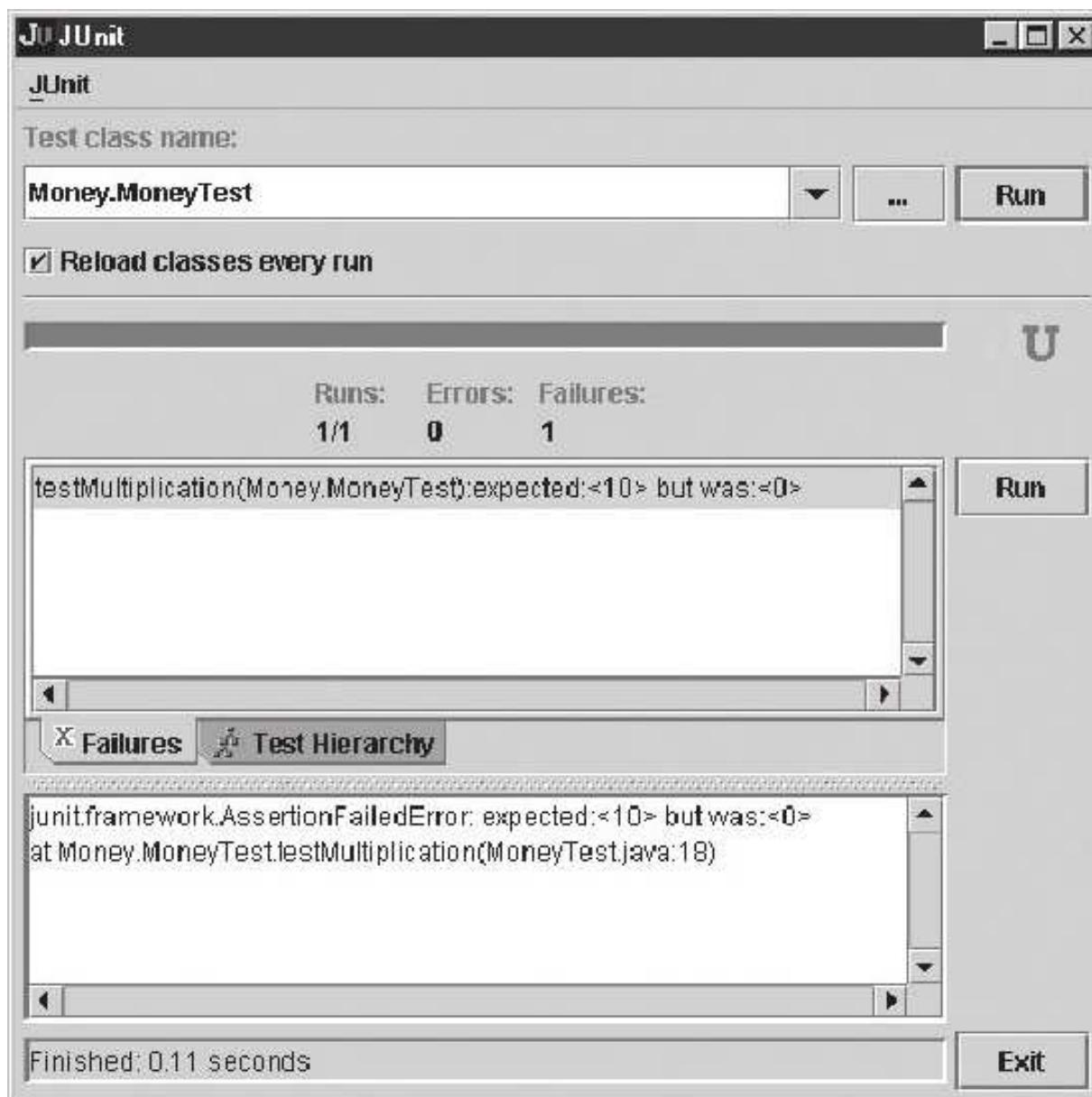
```
int amount;
```

Bingo! Agora podemos rodar o teste e vê-lo falhando, como mostrado na Figura 1.1.

Você está vendo a temida barra vermelha\*. Nossa framework de testes (JUnit, nesse caso) rodou o pequeno trecho de código com o qual começamos e informou que, embora esperássemos “10” como resultado, nós vimos “0”.

---

\* N. de R. T.: Usa-se o termo original, pois não há tradução consensual no Brasil. Um stub tem por objetivo substituir outra unidade de software envolvida em um caso de teste, eliminando a dependência entre as unidades. No caso, um “stub para `times()`” significa criar um stub que substitua o método `“times()”`. Para mais detalhes sobre teste, ver o livro PEZZÈ, Mauro; YOUNG, Michal. *Teste e Análise de software: processos, princípios e técnicas*. Porto Alegre: Bookman, 2008.



**Figura 1.1** Progresso! O teste falha.

Tristeza? Não, não. Falha é progresso. Agora temos uma medida concreta da falha. Isso é melhor que apenas saber vagamente que estamos falhando. Nossa problema de programação foi transformado de “me dê multi-moeda” para “faça esse teste funcionar e, então, faça o resto dos testes funcionarem”. Muito mais simples. Muito menos escopo para temer. Nós podemos fazer esse teste funcionar.

Você provavelmente não vai gostar da solução, mas a meta agora não é obter a resposta perfeita, mas passar no teste. Nós faremos nosso sacrifício no altar da verdade e da beleza mais tarde.

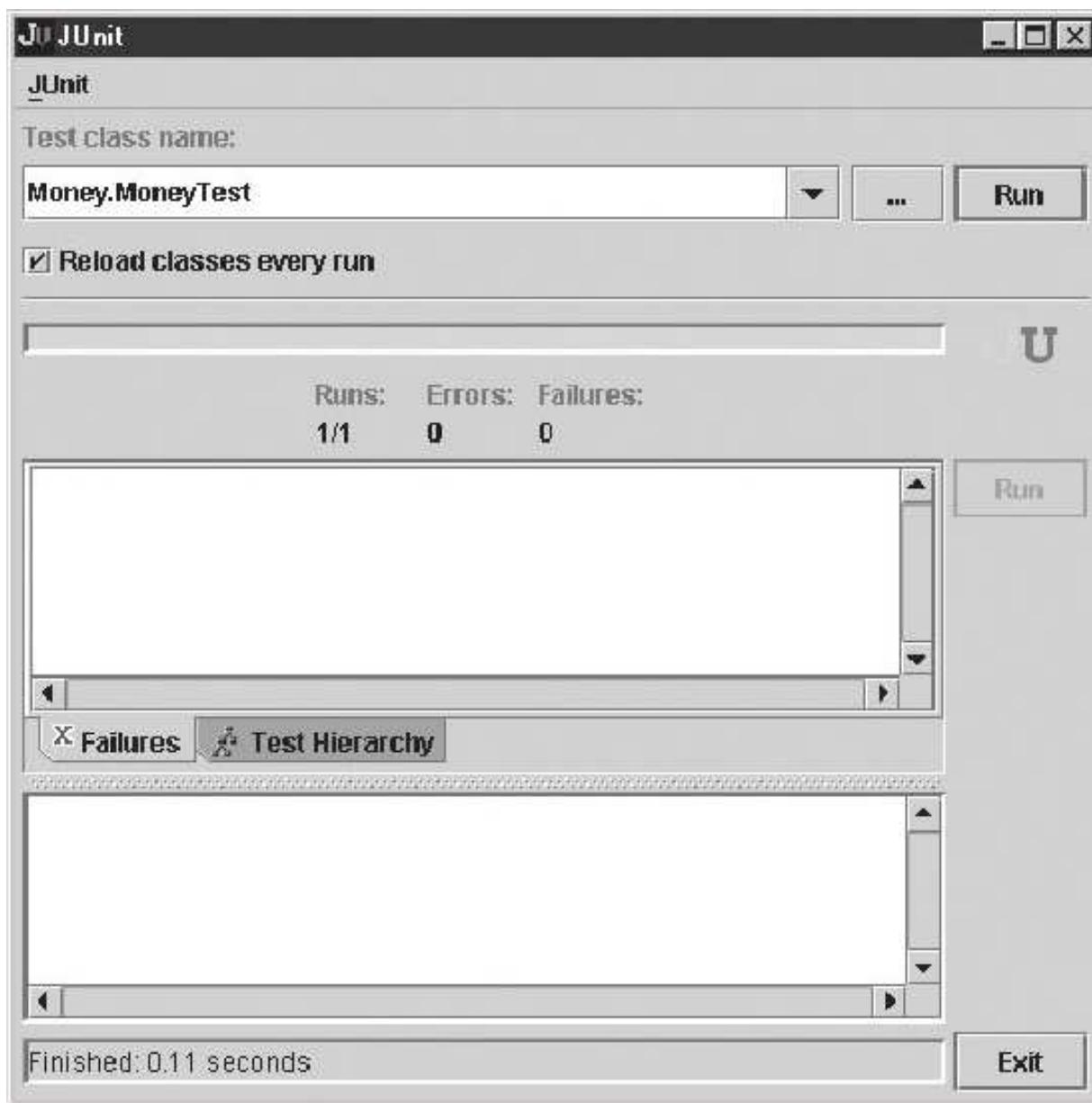
Aqui está a menor mudança que eu pude imaginar que faria nosso teste passar:

### Dollar

```
int amount= 10;
```

---

\* N. de R. T.: Um resultado da execução do framework JUnit é a barra vermelha significando “teste falhou” (fail) em contraposição à barra verde “teste passou” (pass). Aos leitores desta edição: no livro original em inglês também não há uma Figura 1.1 colorida (se isso serve de consolo). Vejam mais detalhes sobre frameworks de teste na Parte II deste livro.



**Figura 1.2** O teste roda.

A Figura 1.2 mostra o resultado quando o teste roda de novo. Agora nós temos a barra verde, imaginada em canção e história

Ó, alegria, ó, encanto! Não tão rápido, meu caro hacker. O ciclo não está completo. Há pouquíssimas entradas no mundo que fariam tão limitada, tão mal cheirosa, tão ingênuas implementações passar. Precisamos generalizar antes de prosseguirmos. Lembre-se, o ciclo é como segue.

1. Adicione um pequeno teste
2. Rode todos os testes e falhe
3. Faça uma pequena mudança
4. Rode os testes e seja bem-sucedido
5. Refatore para remover duplicação

Steve Freeman salientou que o problema com o testar e codificar tal como está colocado não é duplicação (que ainda não apontei para você, mas eu prometo fazê-lo tão logo essa digressão estiver terminada). O problema é a dependência entre o código e o teste – você não pode mudar um sem mudar o outro. Nossa meta é sermos capazes de escrever outro teste que “faça sentido” para nós, sem ter que mudar o código (algo que não é possível com a implementação atual).

Dependência é o problema chave no desenvolvimento de software em todas as escalas. Se você tem detalhes da implementação de SQL de um fornecedor espalhados por todo o código e você decide mudar para outro fornecedor, então descobrirá que seu código é dependente do banco de dados do fornecedor. Você não pode mudar o banco de dados sem mudar o código.

Se dependência é o problema, duplicação é o sintoma. Duplicação na maioria das vezes toma a forma de lógica duplicada – a mesma expressão aparecendo em múltiplos lugares no código. Objetos são excelentes para abstrair a duplicação de lógica. Diferente da maioria dos problemas na vida, onde eliminar os sintomas apenas faz o problema brotar em algum outro lugar de uma forma ainda pior, eliminar duplicação em programas elimina dependência. É por isso que a segunda regra aparece em TDD. Eliminando a duplicação antes de continuarmos com o próximo teste, maximizamos nossas chances de obtermos o próximo teste com uma, e apenas uma, mudança.

---

Nós rodamos os itens de 1 até 4. Agora estamos preparados para remover duplicação. Mas, onde está a duplicação? Geralmente, você vê duplicação entre dois pedaços de código, mas aqui a duplicação está entre o dado no teste e o dado no código. Não vê? E se escrevêssemos o seguinte:

### Dollar

```
int amount= 5 * 2;
```

Aquele 10 tinha que vir de algum lugar. Fizemos a multiplicação em nossas cabeças tão rápido que nem sequer notamos. O 5 e o 2 estão agora em dois lugares, e devemos eliminar implacavelmente a duplicação antes de prosseguir. As regras dizem isso. Não há um passo único que elimine o 5 e o 2. Mas, e se nós movermos a atribuição de quantidade da inicialização do objeto para o método `times()`?

### Dollar

```
int amount;
```

```
void times(int multiplier) {  
    amount= 5 * 2;  
}
```

O teste ainda passa; a barra permanece verde. A felicidade ainda é nossa.

Esses passos não parecem muito pequenos para você? Lembre-se, TDD não é sobre darmos passos pequeninos; é sobre sermos capazes de dar passos pequeninos. Eu codificaria no dia a dia com passos desse tamanho? Não. Mas, quando as coisas ficam no mínimo um pouco estranhas, eu fico feliz de poder fazer isso. Tente passos pequeninos com um exemplo de sua própria escolha. Se você puder dar passos desse tamanho, pode certamente dar passos do tamanho certo. Se você apenas dá passos maiores, nunca saberá se passos menores são apropriados. Deixando a defensiva de lado, onde estávamos? Ah, sim, estávamos eliminando a duplicação entre o teste e o código funcionando. Onde podemos obter um 5? Aquele foi o valor passado para o construtor, então, se nós o salvamos em uma variável quantidade,

### Dollar

```
Dollar(int amount) {
    this.amount= amount;
}
```

Podemos usá-lo em times():

### Dollar

```
void times(int multiplier) {
    amount= amount * 2;
}
```

O valor do parâmetro “multiplicador” é 2, então podemos substituir o parâmetro pela constante:

### Dollar

```
void times(int multiplier) {
    amount= amount * multiplier;
}
```

Para demonstrar nosso profundo conhecimento da sintaxe de Java, queremos usar o operador \*= (o qual faz, isso deve ser dito, reduzir a duplicação):

### Dollar

```
void times(int multiplier) {
    amount *= multiplier;
}
```

$\$5 + 10 \text{ CHF} = \$10 \text{ se a taxa é } 2:1$

$\$5 * 2 = \$10$

Tornar “quantidade” privada

Efeitos colaterais em Dollar?

Arredondamento de dinheiro?

Podemos marcar agora o primeiro teste como feito. Em seguida, cuidaremos daqueles efeitos colaterais estranhos. Mas, primeiro, vamos revisar. Fizemos o seguinte:

- Fizemos uma lista dos testes que sabíamos que precisávamos ter funcionando.
- Contamos uma história com um trecho de código sobre como queríamos ver uma operação.
- Ignoramos os detalhes do JUnit por enquanto.
- Fizemos o teste compilar com stubs.
- Fizemos o teste rodar, cometendo pecados horríveis.
- Gradualmente, generalizamos o código que está funcionando, substituindo constantes por variáveis.
- Adicionamos itens à nossa lista de tarefas em vez de resolvê-los todos de uma vez.

# CAPÍTULO 2

---

## Degenerar Objetos



1. Escreva um teste. Pense em como você gostaria que a operação em sua mente aparecesse em seu código. Você está escrevendo uma história. Invença a interface que deseja ter. Inclua na história todos os elementos que você imagina que serão necessários para calcular as respostas certas.
2. Faça-o rodar. Fazer rapidamente aquela barra ir para verde domina todo o resto. Se a solução limpa e simples é óbvia, então codifique-a. Se a solução limpa e simples é óbvia, mas vai levar um minuto, então tome nota dela e volte para o problema principal que é deixar a barra verde em segundos. Essa mudança na estética é difícil para alguns engenheiros de software experientes. Eles apenas sabem como seguir as regras da boa engenharia. Um verde rápido perdoa todos os pecados. Mas, apenas por um momento.
3. Faça direito. Agora que o sistema está se comportando, ponha os caminhos pecaminosos do passado recente atrás de você. Dê um passo para trás na correta e minuciosa trilha da integridade de software. Remova a duplicação que você introduziu e chegue ao verde rapidamente.

O objetivo é código limpo que funciona (obrigado a Ron Jeffries por esse resumo conciso). Código limpo que funciona está fora do alcance mesmo dos melhores programadores em algum momento, e fora do alcance da maior parte dos programadores (como eu) a maior parte do tempo. Dividir e conquistar. Primeiro resolveremos a parte do “que funciona” do problema. Então resolveremos a parte do “código limpo”. Isso é o oposto do desenvolvimento guiado por arquitetura, em que, primeiro, você resolve o “código limpo”, então luta tentando integrar no projeto as coisas que aprende conforme resolve o problema do “que funciona”.

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1  
 ~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada  
**Efeitos colaterais em Dollar?**  
 Arredondamento de dinheiro?

Temos um teste para trabalhar, mas percebemos algo estranho no processo: quando fazemos uma operação em Dollar, o Dollar muda. Eu quero ser capaz de escrever:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

Eu não consigo imaginar uma maneira limpa de fazer esse teste funcionar. Depois da primeira chamada de `times()`, cinco não é mais cinco – é, na realidade, dez. Se, contudo, retornarmos um novo objeto de `times()`, então podemos multiplicar nossos cinco contos originais para sempre e nunca teremos de mudá-lo.

Estamos mudando a interface de `Dollar` quando fazemos essa mudança, então temos que mudar o teste. Tudo bem. Nossos palpites sobre a interface certa provavelmente não são mais perfeitos que nossos palpites sobre a implementação certa.

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(10, product.amount);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

O novo teste não compila até que mudemos a declaração de `Dollar.times()`:

### Dollar

```
Dollar times(int multiplier) {
    amount *= multiplier;
    return null;
}
```

Agora o teste compila, mas não roda. Progresso! Fazê-lo rodar requer que retornemos um novo `Dollar` com a quantia correta:

### Dollar

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

Tornar “quantidade” privada

~~Efeitos colaterais em Dollar?~~

Arredondamento de dinheiro?

No Capítulo 1, quando fizemos um teste funcionar, começamos com uma implementação fajuta e gradativamente a fizemos real. Aqui, codificamos o que pensávamos estar correto e rezamos enquanto os testes rodavam (preces pequenas, para ser exato, pois rodar os testes levava apenas alguns milissegundos). Porque fomos sortudos e o teste rodou, conseguimos riscar outro item.

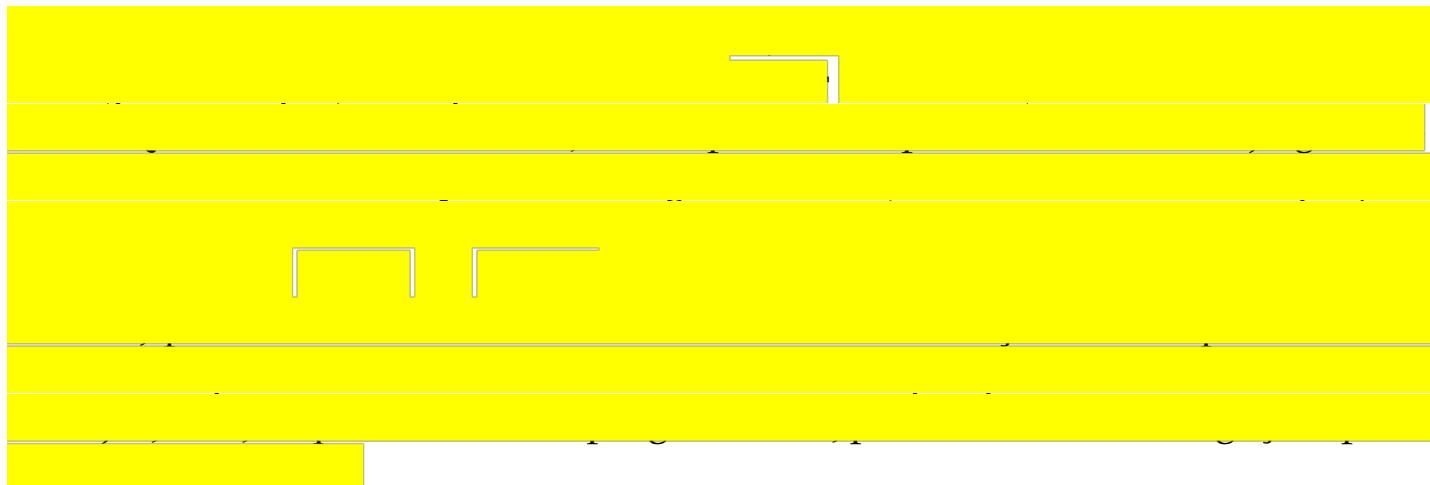
A seguir, estão duas das três estratégias que conheço para chegar rapidamente ao verde:

- Engane-o – Retorne uma constante e gradualmente substitua constantes por variáveis até ter o código real.
- Use Implementação Óvia – Codifique a implementação real.

Quando uso TDD na prática, comumente troco entre esses dois modos de implementação. Quando tudo está indo bem e eu sei o que digitar, ponho Implementação Óvia atrás de Implementação Óvia (rodando os testes a cada vez para garantir que o óbvio para mim também é óbvio para o computador). Assim que eu recebo uma barra vermelha inesperada, dou marcha ré, mudo para implementações falsas e refatoro para o código certo. Quando minha confiança retorna, eu volto para Implementações Óbias.

Há um terceiro estilo de TDD, Triangulação, que demonstraremos no Capítulo 3. Contudo, para revisar:

- Traduzimos um obstáculo de projeto (efeitos colaterais) em um caso de testes que falhou por causa desse obstáculo
- Obtivemos o código compilando rapidamente com uma implementação de stub
- Fizemos o teste funcionar codificando o que parecia ser o código certo



# CAPÍTULO 3

---

## Igualdade para Todos

Se eu tenho um inteiro e somo 1 a ele, não espero o inteiro original mudar; espero usar o novo valor. Objetos geralmente não se comportam desse jeito. Se eu tenho um contrato e adiciono um a sua cobertura, então a cobertura do contrato deveria mudar (sim, sim, sujeita a todo tipo de regras de negócio interessantes que não nos interessam aqui).

Podemos usar objetos como valores, como estamos usando nosso `Dollar` agora. O padrão para isso é `Value Object`\*. Uma das restrições em usar `Value Object` é que os valores das variáveis de instância do objeto nunca mudem uma vez que foram criados no construtor.

Há uma grande vantagem em usar `Value Object`: você não tem que se preocupar com problemas de sinônimos. Digamos que eu tenho um cheque e coloque sua quantia em \$5, e então eu crio outra quantia de cheque com os mesmos \$5. Alguns dos piores bugs da minha carreira ocorreram quando mudar o valor do primeiro cheque mudou, inadvertidamente, o valor do segundo cheque. Isso são sinônimos.

Quando você tem `Value Object`, não precisa se preocupar com sinônimos. Se eu tenho \$5, então tenho certeza que ele vai ser \$5 para todo o sempre. Se alguém quer \$7, então terá que criar um objeto totalmente novo.

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

~~Efeitos colaterais em `Dollar`?~~

Arredondamento de dinheiro?

`equals()`

---

\* N. de R. T.: `Value Object`: padrão de projeto (*design pattern*) que permite a criação de objeto simples com cópias dos atributos de um objeto sem permitir sua alteração. Não é um padrão GoF. Ver mais detalhes em <http://java.sun.com/j2ee/patterns/ValueObject.html>.

Uma implicação do Value Object é que todas as operações devem retornar um novo objeto, como vimos no Capítulo 2. Outra implicação é que Value Object deve implementar `equals()`, pois um \$5 é tão bom quanto qualquer outro.

\$5 + 10 CHF = \$10 se a taxa é 2:1  
~~\$5 \* 2 = \$10~~  
Tornar “quantidade” privada  
~~Efeitos colaterais em Dollar?~~  
Arredondamento de dinheiro?  
`equals()`  
`hashCode()`

Se você usar Dollars como a chave para uma tabela hash, então tem que implementar `hashCode()` se você implementar `equals()`. Colocaremos isso na lista de tarefas também e chegaremos a ela quando for um problema.

Você não está pensando sobre a implementação de `equals()`, está? Bom. Nem eu. Depois de bater atrás da minha mão com uma régua, estou pensando em como testar igualdade. Primeiro, \$5 deveria ser igual a \$5:

```
public void testEquality() {  
    assertTrue(new Dollar(5).equals(new Dollar(5)));  
}
```

A barra fica adoravelmente vermelha. A implementação falsa é para retornar apenas `true`:

#### Dollar

```
public boolean equals(Object object) {  
    return true;  
}
```

Você e eu sabemos que `true` é realmente “`5 == 5`”, que é, na realidade, “`quantidade == 5`”, que é, na realidade, “`quantidade == dollar.amount`”. Sem passar por essas etapas, eu não seria capaz de demonstrar a terceira e mais conservadora estratégia de implementação: Triangulação.

Se duas estações receptoras a uma distância conhecida uma da outra podem medir a direção de um sinal de rádio, então há informação suficiente para calcular a distância e a direção do sinal (se você lembra mais de trigonometria que eu, pelo menos). Esse cálculo é chamado Triangulação.

Por analogia, quando triangulamos, apenas generalizamos código quando temos dois exemplos ou mais. Vamos ignorar brevemente a duplicação entre teste e código do modelo. Quando o segundo exemplo exige uma solução mais geral, então, e só então, generalizamos.

Assim, para triangula-mos precisamos de um segundo exemplo. Que tal \$5 != \$6?

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
    assertFalse(new Dollar(5).equals(new Dollar(6)));
}
```

Agora podemos generalizar a igualdade:

### Dollar

```
public boolean equals(Object object) {
    Dollar dollar= (Dollar) object;
    return amount == dollar.amount;
}
```

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

~~Efeitos colaterais em Dollar?~~

Arredondamento de dinheiro?

~~equals()~~

~~hashCode()~~

Poderíamos ter usado Triangulação para guiar a generalização de `times()` também. Se tivéssemos  $\$5 \times 2 = \$10$  e  $\$5 \times 3 = \$15$ , então não estariámos mais aptos a retornar uma constante.

Triangulação parece engraçada para mim. Eu a uso apenas quando estou completamente inseguro sobre como refatorar. Se consigo ver como eliminar duplicação entre código e testes e criar a solução geral, então faço apenas isso. Por que eu precisaria escrever outro teste para me permitir escrever o que, provavelmente, teria escrito em primeiro lugar?

Contudo, quando as ideias de projeto simplesmente não estão surgindo, Triangulação dá a chance para pensarmos sobre o problema de uma direção ligeiramente diferente. Quais eixos de variabilidade você está tentando prover em seu projeto? Faça alguns deles variarem, e a resposta pode se tornar mais clara.

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

~~Efeitos colaterais em Dollar?~~

Arredondamento de dinheiro?

~~equals()~~

~~hashCode()~~

Igualdade de null

Igualdade de objeto

Então, igualdade está feita por enquanto. Mas, e a comparação com null e a comparação com outros objetos? Essas são operações comumente usadas, mas não necessárias no momento, então vamos adicioná-las à lista de tarefas.

Agora que temos igualdade, podemos comparar diretamente Dollars com Dollars. O que nos permite tornar “quantidade” privada (private), como todas as boas variáveis de instância deveriam ser. Para revisar o que vimos:

- Percebemos que nosso padrão de projeto (Value Object) sugeria uma operação
- Testamos para aquela operação
- Implementamos de uma forma simples
- Não refatoramos imediatamente, mas, em vez disso, testamos mais
- Refatoramos para capturar os dois casos de uma só vez

# CAPÍTULO 4

## Privacidade

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

~~Efeitos colaterais em Dollar?~~

Arredondamento de dinheiro?

`equals()`

`hashCode()`

Igualdade de null

Igualdade de objeto

Agora que definimos igualdade, podemos usá-la para fazer nossos testes mais “expressivos”. Conceitualmente, a operação `Dollar.times()` deveria retornar um `Dollar` cujo valor é o valor recebido vezes o multiplicador. Nosso teste não diz exatamente isso:

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);  
    Dollar product= five.times(2);  
    assertEquals(10, product.amount);  
    product= five.times(3);  
    assertEquals(15, product.amount);  
}
```

Nós podemos reescrever a primeira asserção para comparar `Dollars` com `Dollars`:

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);  
    Dollar product= five.times(2);  
    assertEquals(new Dollar(10), product);
```

```

product= five.times(3);
assertEquals(15, product.amount);
}

```

Isso parece melhor, então reescrevemos a segunda asserção também:

```

public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(new Dollar(10), product);
    product= five.times(3);
    assertEquals(new Dollar(15), product);
}

```

Agora a variável temporária `product` não está ajudando muito, então podemos otimizá-la:

```

public void testMultiplication() {
    Dollar five= new Dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}

```

Esse teste nos fala mais claramente, como se fosse uma afirmação de verdade e não uma sequência de operações.

Com essas mudanças para o teste, `Dollar` é agora a única classe usando sua variável de instância `amount`, então podemos fazê-la privada:

```

Dollar
private int amount;

```

\$5 + 10 CHF = \$10 se a taxa é 2:1  
~~\$5 \* 2 = \$10~~  
Tornar “quantidade” privada  
Efeitos colaterais em Dollar?  
Arredondamento de dinheiro?  
equals()  
hashCode()  
Igualdade de null  
Igualdade de objeto

E podemos riscar outro item da lista de tarefas. Perceba que nos abrimos ao risco. Se o teste de igualdade falha em checar cuidadosamente se a igualdade está funcionando, então o teste para multiplicação poderia falhar também em checar cuidadosamente se a multiplicação está funcionando. Esse é um risco que gerenciamos ativamente em TDD. Não estamos lutando pela perfeição. Ao dizer tudo de duas formas – como código e como teste – esperamos reduzir nossos defeitos o suficiente para seguir em frente com confiança. De tempos em tempos nosso raciocínio

falhará e um defeito vai passar. Quando isso acontecer, aprenderemos nossa lição de como o teste deveria ter sido escrito e seguiremos em frente. No resto do tempo, continuaremos em frente audaciosamente sob nossa barra verde bravamente tremulante (minha barra realmente não tremula, mas a gente pode sonhar).

Para revisar:

- Usamos uma funcionalidade recém-desenvolvida para melhorar um teste
- Percebemos que, se dois testes falham de uma vez, estamos arruinados
- Prosseguimos a despeito do risco
- Usamos uma nova funcionalidade em um objeto sob teste para reduzir o acoplamento entre os testes e o código

# CAPÍTULO 5

## Falando Franca-mente\*

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

$\$5 * 2 = \$10$

Tornar “quantidade” privada

Efeitos colaterais em Dollar?

Arredondamento de dinheiro?

equals()

hashCode()

Igualdade de null

Igualdade de objeto

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Como vamos abordar o primeiro teste, o teste mais interessante da lista? Ainda parece ser um grande salto. Não estou certo de poder escrever um teste que eu possa implementar em uma pequena etapa. Um pré-requisito parece ser termos um objeto como `Dollar`, mas para representar francos. Se conseguirmos que o objeto `Franc` funcione da forma que o objeto `Dollar` funciona agora, estaremos mais próximos de escrever e rodar um teste misto de adição.

Podemos copiar e editar o teste de `Dollar`:

```
public void testFrancMultiplication() {  
    Franc five= new Franc(5);  
    assertEquals(new Franc(10), five.times(2));  
    assertEquals(new Franc(15), five.times(3));  
}
```

---

\* N. de R. T.: No título original deste capítulo, *Franc-ly speaking*, o autor faz um jogo de palavras com o termo *franc* que aparece na palavra *francly* (francamente, sinceramente) mas que refere-se também à moeda usada na Suíça (francos suíços). O jogo é irônico pois no exemplo monetário nesta Parte I do livro, moedas como dólar e franco são comumente usadas.

(Você não está feliz por simplificarmos o teste do Capítulo 4? Foi o que fez nosso teste aqui ser mais fácil. Não é maravilhoso como muitas vezes as coisas funcionam assim como nos livros? Eu realmente não planejei desse jeito, mas não farei promessas para o futuro.)

Qual pequeno passo nos levará à barra verde? Copiar o código de `Dollar` e substituir `Dollar` por `Franc`.

Pare. Espere. Eu posso ouvir a inclinação estética em você, zombando e cuspido. Reuso do tipo copiar-e-colar? A morte da abstração? O assassino da clareza de projeto?

Se você está chateado, respire fundo. Inspire pelo nariz... segure, 1, 2, 3... expire pela boca. Pronto. Lembre-se, nosso ciclo tem fases diferentes (elas passam rapidamente, em segundos, mas são fases):

1. Escreva um teste
2. Faça-o compilar
3. Rode-o e veja-o falhar
4. Faça-o rodar
5. Remova duplicação

As diferentes fases têm diferentes propósitos. Elas invocam estilos diferentes de solução, diferentes perspectivas estéticas. As primeiras três fases precisam passar rapidamente, assim chegamos a um estado conhecido com a nova funcionalidade. Podemos cometer qualquer número de pecados para chegar lá, pois a velocidade agora é melhor que o projeto, ao menos por um breve momento.

Agora estou preocupado. Dei a você permissão para abandonar todos os princípios de bom projeto. Lá vai você para sua equipe – “Kent disse que toda aquela coisa de projeto não importa”. Pare. O ciclo não está completo. Uma cadeira Aeron de quatro pés cai. As quatro primeiras etapas do ciclo não funcionarão sem a quinta. Bom projeto no bom momento. Faça-o rodar, faça-o direito.

Pronto, me sinto melhor. Agora estou certo de que você não mostrará a ninguém, exceto para seu colega, seu código, até que tenha removido a duplicação. Onde estávamos? Ah, sim. Violando todos os dogmas do bom projeto em prol da velocidade (a penitência por nossos pecados ocupará os próximos capítulos).

### Franc

```
class Franc {  
    private int amount;  
  
    Franc(int amount) {  
        this.amount= amount;  
    }  
  
    Franc times(int multiplier) {  
        return new Franc(amount * multiplier);  
    }  
}
```

```

public boolean equals(Object object) {
    Franc franc= (Franc) object;
    return amount == franc.amount;
}
}

```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

~~Tornar “quantidade” privada~~

~~Efeitos colaterais em Dollar?~~

~~Arredondamento de dinheiro?~~

~~equals()~~

~~hashCode()~~

~~Igualdade de null~~

~~Igualdade de objeto~~

~~5 CHF \* 2 = 10 CHF~~

Duplicação de Dólar/Franco

~~Igualdade comum~~

Multiplicação comum

Devido ao passo para rodar o código ser tão curto, fomos capazes de pular a etapa de “faça-o compilar”.

Agora temos duplicação abundante e temos que eliminá-la antes de escrever nosso próximo teste. Começaremos generalizando `equals()`. Entretanto, podemos riscar um item de nossa lista de tarefas, embora tenhamos acrescentado mais dois. Revisando:

- Não pudemos armar um grande teste, então inventamos um pequeno teste que representa progresso
- Escrevemos o teste por meio de uma duplicação e uma edição descaradas
- Ainda pior, fizemos o teste funcionar pela cópia e edição indiscriminada do código modelo
- Prometemos a nós mesmos que não iríamos para casa até a duplicação ter sumido

# CAPÍTULO 6

---

## Igualdade para Todos, Restaurada

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

Efeitos colaterais em Dollar?

Arredondamento de dinheiro?

`equals()`

`hashCode()`

Igualdade de null

Igualdade de objeto

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Duplicação de Dólar/Franco

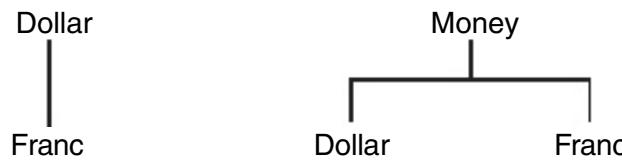
Igualdade comum

Multiplicação comum

Há uma fabulosa sequência em *Crossing to Safety* na qual o autor, Wallace Stegner, descreve a oficina de um personagem. Cada item está perfeitamente no lugar, o chão é impecável, tudo está em ordem e limpo. O personagem, contudo, nunca fez nada. “Preparar tem sido o trabalho de sua vida. Ele prepara, então limpa.” (Esse é também o livro cujo final me fez debulhar-me em lágrimas na classe executiva em um transatlântico 747. Leia com cuidado.)

Nós evitamos essa armadilha no Capítulo 5. Realmente conseguimos um novo caso de teste funcionando. Mas, pecamos enormemente ao copiar e colar toneladas de código de forma a fazê-lo rapidamente. Agora é hora da limpeza.

Uma possibilidade é fazer uma de nossas classes estender a outra. Eu tentei isso, e dificilmente salva qualquer código. Em vez disso, encontraremos uma superclasse comum para as duas classes, como mostra a Figura 6.1. (Eu já tentei isso também, e funciona muito bem, embora leve um pouco mais de tempo.)



**Figura 6.1** Uma superclasse comum para as duas classes.

E se tivéssemos uma classe `Money` para capturar o código de igualdade comum? Podemos começar pequeno:

```
Money
class Money
```

Todos os testes ainda rodam – não que pudéssemos ter estragado algo, mas é uma boa hora para rodar testes de qualquer forma. Se `Dollar` estende `Money`, isso não poderia estragar nada.

```
Dollar
class Dollar extends Money {
    private int amount;
}
```

Poderia? Não, todos os testes ainda rodam. Agora podemos mover a variável de instância `amount` para `Money`:

```
Money
class Money {
    protected int amount;
}
```

```
Dollar
class Dollar extends Money {
}
```

A visibilidade tem que mudar de particular para protegida para que a subclasse possa também vê-la. (Quiséssemos ir mais lentamente, poderíamos ter declarado o campo `Money` em uma etapa e então removido ele de `Dollar` em uma segunda etapa. Estou me sentindo arrojado.)

Agora podemos começar a trabalhar no código de `equals()` prontos para movê-lo para cima. Primeiro mudamos a declaração da variável temporária:

```
Dollar
public boolean equals(Object object) {
    Money dollar= (Dollar) object;
    return amount == dollar.amount;
}
```

Todos os testes ainda rodam. Agora mudamos a conversão:

### Dollar

```
public boolean equals(Object object) {
    Money dollar= (Money) object;
    return amount == dollar.amount;
}
```

Para comunicar melhor, deveríamos também mudar o nome da variável temporária:

### Dollar

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

Agora podemos movê-lo de `Dollar` para `Money`:

### Money

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

Agora, precisamos eliminar `Franc.equals()`. Primeiro, percebemos que os testes de igualdade não cobrem a comparação de Francs com Francs. Nossos pecados na cópia de código estão nos alcançando. Antes de mudarmos o código, escreveremos os testes que deveriam em primeiro lugar estar lá.

Você frequentemente estará implementando TDD em código que não tem testes adequados (ao menos para a próxima década ou coisa assim). Quando você não tem testes suficientes, é obrigado a encarar refatorações que não são suportadas por testes. Você poderia cometer um erro de refatoração, e os testes ainda assim rodariam. O que você faz?

Felizmente, aqui os testes são fáceis de escrever. Nós apenas copiamos os testes de `Dollar`:

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
    assertFalse(new Dollar(5).equals(new Dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
}
```

Mais duplicação, duas linhas a mais! Vamos expiar os pecados, também.  
Testes no lugar, podemos ter Franc estendendo Money:

### Franc

```
class Franc extends Money {
    private int amount;
}
```

Podemos deletar o campo amount de Franc em favor daquele em Money:

### Franc

```
class Franc extends Money {
}
```

Franc.equals() é quase o mesmo que Money.equals(). Se nós os fizermos precisamente iguais, então podemos deletar a implementação em Franc sem mudar o sentido do programa. Primeiro, mudamos a declaração da variável temporária:

### Franc

```
public boolean equals(Object object) {
    Money franc= (Franc) object;
    return amount == franc.amount;
}
```

Então mudamos a conversão:

### Franc

```
public boolean equals(Object object) {
    Money franc= (Money) object;
    return amount == franc.amount;
}
```

Realmente temos que mudar o nome da variável temporária para combinar com a superclasse? Eu deixarei isso para sua consciência.... certo, faremos isso:

### Franc

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount;
}
```

\$5 + 10 CHF = \$10 se a taxa é 2:1  
~~\$5 \* 2 = \$10~~  
 Tornar “quantidade” privada  
~~Efeitos colaterais em Dollar?~~  
 Arredondamento de dinheiro?  
~~equals()~~

hashCode()  
Igualdade de null  
Igualdade de objeto  
~~5 CHF \* 2 = 10 CHF~~  
Duplicação de Dólar/Franco  
~~Igualdade comum~~  
Multiplicação comum  
Comparar Francos com Dólares

Agora, não há diferença entre `Franc.equals()` e `Money.equals()`, então deletaremos a implementação redundante em `Franc`. E rodamos os testes. Eles rodam. O que acontece quando comparamos `Francs` com `Dollars`? Chegaremos a isso no Capítulo 7. Revisando o que fizemos aqui:

- Movemos passo a passo código comum de uma classe (`Dollar`) para uma superclasse (`Money`)
- Fizemos de uma segunda classe (`Franc`) uma subclasse
- Reconciliamos duas implementações – `equals()` – antes de eliminar aquela redundante

# CAPÍTULO 7

## Maçãs e Laranjas

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

Efeitos colaterais em Dollar?

Arredondamento de dinheiro?

equals()

hashCode()

Igualdade de null

Igualdade de objeto

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Duplicação de Dólar/Franco

Igualdade comum

Multiplicação comum

Comparar Francos com Dólares

Um pensamento nos surpreendeu no final do Capítulo 6: o que acontece quando comparamos Francs com Dollars? Nós respeitosamente transformamos nosso pavoroso pensamento em um item na lista de tarefas. Mas não conseguimos tirá-lo da cabeça. O que acontece?

```
public void testEquality() {  
    assertTrue(new Dollar(5).equals(new Dollar(5)));  
    assertFalse(new Dollar(5).equals(new Dollar(6)));  
    assertTrue(new Franc(5).equals(new Franc(5)));  
    assertFalse(new Franc(5).equals(new Franc(6)));  
    assertFalse(new Franc(5).equals(new Dollar(5)));  
}
```

Ele falha. Dollars são Francs. Antes que os compradores suíços fiquem entusiasmados, vamos tentar corrigir o código. O código de igualdade precisa verificar

que não estamos comparando Dollars com Francs. Podemos fazer isso agora comparando a classes dos dois objetos – dois Moneys são iguais apenas se suas quantidades e classes são iguais.

### Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

Usar as classes dessa forma no código modelo é um pouco mau cheiroso\*. Gostaríamos de usar um critério que fizesse sentido no domínio das finanças, não no domínio de objetos Java. Mas, não temos atualmente nada como uma moeda, e isso não parece ser razão suficiente para introduzir uma, então isso terá que ficar assim por enquanto.

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~  
~~\$5 \* 2 = \$10~~  
~~Tornar “quantidade” privada~~  
~~Efeitos colaterais em Dollar?~~  
~~Arredondamento de dinheiro?~~  
~~equals()~~  
~~hashCode()~~  
~~Igualdade de null~~  
~~Igualdade de objeto~~  
~~5 CHF \* 2 = 10 CHF~~  
~~Duplicação de Dólar/Franco~~  
~~Igualdade comum~~  
~~Multiplicação comum~~  
~~Comparar Francos com Dólares~~  
~~Moeda?~~

Agora, realmente precisamos nos livrar do código comum de `times()` para que possamos obter uma aritmética de moeda mista. Antes de fazê-lo, contudo, podemos revisar nossas grandes realizações desse capítulo:

- Pegamos uma dúvida que nos incomodava e a transformamos em um teste.
- Fizemos o teste rodar de uma forma razoável, mas não perfeita – `getClass()`.
- Decidimos não introduzir mais projeto até termos uma motivação melhor.

---

\* N. de R. T.: Referência a *bad smells* (mau cheiro), termo que a comunidade OO adotou para expressar que um trecho de código é potencialmente fonte de problemas, suspeito e forte candidato a refatoração.

# CAPÍTULO 8

## Fazendo Objetos

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

Efeitos colaterais em Dollar?

Arredondamento de dinheiro?

equals()

hashCode()

Igualdade de null

Igualdade de objeto

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Duplicação de Dólar/Franco

Igualdade comum

Multiplicação comum

~~Comparar Francos com Dólares~~

Moeda?

As duas implementações de times() são extraordinariamente similares:

### Franc

```
Franc times(int multiplier) {  
    return new Franc(amount * multiplier);  
}
```

### Dollar

```
Dollar times(int multiplier) {  
    return new Dollar(amount * multiplier);  
}
```

Podemos dar um passo na direção de conciliá-los, fazendo ambos retornarem um `Money`:

#### **Franc**

```
Money times(int multiplier) {
    return new Franc(amount * multiplier);
}
```

#### **Dollar**

```
Money times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

O próximo passo não é tão óbvio. As duas subclasses de `Money` não estão fazendo o suficiente para justificarem sua existência, logo, gostaríamos de eliminá-las. Mas, não podemos fazer isso em um grande passo, pois não seria uma demonstração muito eficaz de TDD.

Certo, estaremos um passo mais próximos de eliminarmos as subclasses se houver menos referências diretas às subclasses. Podemos introduzir um método fábrica em `Money` que retorne um `Dollar`. Usaremos algo assim:

```
public void testMultiplication() {
    Dollar five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

A implementação cria e retorna um `Dollar`:

#### **Money**

```
static Dollar dollar(int amount) {
    return new Dollar(amount);
}
```

Mas, queremos que as referências a `Dollars` desapareçam, logo precisamos mudar a declaração no teste:

```
public void testMultiplication() {
    Money five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Nossa compilador educadamente nos informa que `times()` não está definido para `Money`. Ainda não estamos prontos para implementá-lo, então fazemos `Money` abstrato (suponho que nós deveríamos ter começado fazendo isso, não?) e declaramos `Money.times()`:

#### **Money**

```
abstract class Money
abstract Money times(int multiplier);
```

Agora podemos mudar a declaração do método fábrica:

### Money

```
static Money dollar(int amount) {
    return new Dollar(amount);
}
```

Todos os testes rodam, então, pelo menos, não estragamos nada. Nós podemos, agora, usar nosso método fábrica em qualquer lugar dos testes:

```
public void testMultiplication() {
    Money five = Money.dollar(5);
    assertEquals(Money.dollar(10), five.times(2));
    assertEquals(Money.dollar(15), five.times(3));
}
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
    assertFalse(new Franc(5).equals(Money.dollar(5)));
}
```

Agora estamos em uma posição ligeiramente melhor que antes. Nenhum código cliente sabe que existe uma subclasse chamada `Dollar`. Desacoplando os testes da existência das subclasses, nos demos liberdade para mudar herança sem afetar qualquer código modelo.

Antes de irmos mudar cegamente o `testFrancMultiplication`, percebemos que ele não está testando qualquer lógica que não é testada pela multiplicação de `Dollar`. Se deletarmos o teste, perderemos qualquer confiança no código? Um pouco, sim, por isso o deixamos lá. Mas é suspeito.

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
public void testFrancMultiplication() {
    Money five = Money.franc(5);
    assertEquals(Money.franc(10), five.times(2));
    assertEquals(Money.franc(15), five.times(3));
}
```

A implementação é como `Money.dollar()`:

### Money

```
static Money franc(int amount) {
    return new Franc(amount);
}
```

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

~~$\$5 * 2 = \$10$~~

Tornar “quantidade” privada

Efeitos colaterais em Dólar?

Arredondamento de dinheiro?

~~equals()~~

hashCode()

Igualdade de null

Igualdade de objeto

~~5 CHF \* 2 = 10 CHF~~

Duplicação de Dólar/Franco

Igualdade comum

Multiplicação comum

~~Comparar Francos com Dólares~~

Moeda?

Deletar testFrancMultiplication?

Em seguida, vamos nos livrar da duplicação de `times()`. Por agora, para revisar:

- Avançamos um passo eliminando a duplicação pela conciliação das assinaturas de duas variantes do mesmo método – `times()`.
- Movemos ao menos uma declaração dos métodos para uma superclasse comum.
- Desacoplamos código de teste da existência de subclasses concretas pela introdução de métodos fábrica.
- Percebemos que, quando as subclasses desaparecerem, alguns testes serão redundantes, mas não tomamos providências.

# CAPÍTULO 9

## Tempos em que Estamos Vivendo\*

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

$\$5 * 2 = \$10$

Tornar “quantidade” privada  
~~Efeitos colaterais em Dollar?~~  
Arredondamento de dinheiro?

equals()

hashCode()

Igualdade de null

Igualdade de objeto

~~5 CHF \* 2 = 10 CHF~~

Duplicação de Dólar/Franco

Igualdade comum

Multiplicação comum

~~Comparar Francos com Dólares~~

Moeda?

Deletar testFrancMultiplication?

O que há em nossa lista de tarefas que pode nos ajudar a eliminar essas malditas subclasses inúteis? O que aconteceria se introduzíssemos a noção de moeda?

Como queremos implementar moedas no momento? Estraguei tudo, de novo. Antes do “fazedor-de-regras” sair, vou reformular: Como queremos testar moedas no momento? Pronto. Dedos salvos, por enquanto.

---

\* N. de R. T.: O título deste capítulo no original *Times We're Living In* é outro jogo de palavras do autor: a palavra *times* pode se referir à operação de multiplicação (traduzida como “vezes”) – muito usada no capítulo – mas também poderia ser traduzida como “tempos”. O título assim seria “Tempos em que Estamos Vivendo” ou “Vezes em que estamos Vivendo”, mas optamos pelo primeiro...

Podemos querer ter objetos complicados representando moedas com fábricas flyweight\* para garantir que não criamos mais objetos do que realmente precisamos. Mas, por enquanto, strings servirão:

```
public void testCurrency() {
    assertEquals("USD", Money.dollar(1).currency());
    assertEquals("CHF", Money.franc(1).currency());
}
```

Primeiro, declaramos currency() em Money:

#### **Money**

```
abstract String currency();
```

Então o implementamos em ambas as subclasses:

#### **Franc**

```
String currency() {
    return "CHF";
}
```

#### **Dollar**

```
String currency() {
    return "USD";
}
```

Queremos que a mesma implementação seja suficiente para ambas as classes. Poderíamos armazenar a moeda em uma variável de instância e apenas retornar a variável. (Vou começar a ir um pouco mais rápido com as refatorações devido ao tempo. Se eu for muito rápido, por favor, me avise para ir mais devagar. Ei, espere, isso é um livro – talvez eu apenas não deva acelerar demais.)

#### **Franc**

```
private String currency;
Franc(int amount) {
    this.amount = amount;
    currency = "CHF";
}
String currency() {
    return currency;
}
```

Podemos fazer o mesmo com Dollar:

#### **Dollar**

```
private String currency;
```

---

\* N. de R. T.: Referência ao padrão GoF Flyweight. Flyweight é usado para criar compartilhamento entre dados de objetos e limitar a proliferação de muitas classes similares pequenas. Ver livro GAMMA, Erich et al. *Padrões de projeto*: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

```
Dollar(int amount) {
    this.amount = amount;
    currency = "USD";
}
String currency() {
    return currency;
}
```

Agora, podemos subir a declaração da variável e a implementação de `currency()`, pois elas são idênticas:

### Money

```
protected String currency;
String currency() {
    return currency;
}
```

Se movermos as strings constantes “USD” e “CHF” para os métodos fábrica (factory) estáticos, então os dois construtores serão idênticos e poderemos criar uma implementação comum.

Primeiro, adicionaremos um parâmetro ao construtor:

### Franc

```
Franc(int amount, String currency) {
    this.amount = amount;
    this.currency = "CHF";
}
```

Isso estraga os dois chamadores do construtor:

### Money

```
static Money franc(int amount) {
    return new Franc(amount, null);
}
```

### Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier, null);
}
```

Espere um minuto! Por que `Franc.times()` está chamando o construtor em vez do método fábrica? Queremos fazer essa mudança agora ou vamos esperar? A resposta dogmática é que esperaremos; não interromper o que estamos fazendo. A resposta, na minha prática, é que vou fazer uma breve interrupção, mas bem breve, e que nunca interromperei uma interrupção (Jim Coplien me ensinou essa regra). Para ser realista, vamos limpar `times()` antes de prosseguir:

### Franc

```
Money times(int multiplier) {
    return Money.franc(amount * multiplier);
}
```

Agora, o método fábrica pode passar “CHF”:

### Money

```
static Money franc(int amount) {
    return new Franc(amount, "CHF");
}
```

E, finalmente, podemos atribuir o parâmetro para a variável de instância:

### Franc

```
Franc(int amount, String currency) {
    this.amount = amount;
    this.currency = currency ;
}
```

Estou me sentindo na defensiva novamente sobre dar passos muito pequenos. Estou recomendando que você realmente trabalhe dessa maneira? Não. Estou recomendando que você seja capaz de trabalhar dessa forma. O que eu acabei de fazer foi trabalhar em passos maiores e cometer um erro meio estúpido. Eu tirei um valioso minuto de mudanças, troquei para uma marcha menor, e fiz isso com passos pequenos. Estou me sentindo melhor agora, então veremos se conseguimos fazer a mudança análoga de `Dollar` de uma vez só:

### Money

```
static Money dollar(int amount) {
    return new Dollar(amount, "USD");
}
```

### Dollar

```
Dollar(int amount, String currency) {
    this.amount = amount;
    this.currency = currency;
}
Money times(int multiplier) {
    return Money.dollar(amount * multiplier);
}
```

E funcionou de primeira! Iúpi!

Esse é o tipo de sintonia que você fará constantemente com TDD. Os passos pequenos estão parecendo restritivos? Dê passos maiores. Está se sentindo um pouco inseguro? Dê passos menores. TDD é um processo conduzido um pouco desse jeito, um pouco de outro. Não há tamanho certo de passo, agora e eternamente.

Os dois construtores agora são idênticos, então conseguimos subir a implementação:

### Money

```
Money(int amount, String currency) {
    this.amount = amount;
```

```
this.currency = currency;
}
```

**Franc**

```
Franc(int amount, String currency) {
    super(amount, currency);
}
```

**Dollar**

```
Dollar(int amount, String currency) {
    super(amount, currency);
}
```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

~~Tornar “quantidade” privada~~

~~Efeitos colaterais em Dollar?~~

~~Arredondamento de dinheiro?~~

~~equals()~~

~~Igualdade de null~~

~~Igualdade de objeto~~

~~5 CHF \* 2 = 10 CHF~~

~~Duplicação de Dólar/Franco~~

~~Igualdade comum~~

~~Multiplicação comum~~

~~Comparar Francos com Dólares~~

~~Moeda?~~

~~Deletar testFrancMultiplication?~~

Estamos quase prontos para subir a implementação de `times()` e eliminar as subclasses, mas, primeiro, para revisar:

- Fomos um pouco travados em grandes ideias de projeto, assim, trabalhamos em algo menor que percebemos antes.
- Conciliamos os dois construtores, movendo a variação para o chamador (o método fábrica).
- Interrompemos a refatoração para uma pequena volta, usando o método fábrica em `times()`.
- Repetimos uma refatoração análoga (fazendo em `Dollar` o que já fizemos em `Franc`) em um grande passo.
- Subimos dois construtores idênticos.

# CAPÍTULO 10

## Tempos Interessantes\*

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

~~Tornar “quantidade” privada~~

~~Efeitos colaterais em Dollar?~~

~~Arredondamento de dinheiro?~~

~~equals()~~

~~hashCode()~~

~~Igualdade de null~~

~~Igualdade de objeto~~

~~5 CHF \* 2 = 10 CHF~~

~~Duplicação de Dólar/Franco~~

~~Igualdade comum~~

~~Multiplicação comum~~

~~Comparar Francos com Dólares~~

~~Moeda?~~

~~Deletar testFrancMultiplication?~~

Quando terminarmos este capítulo, teremos uma única classe para representar Money. As duas implementações de `times()` são parecidas, mas não idênticas:

### Franc

```
Money times(int multiplier) {  
    return Money.franc(amount * multiplier);  
}
```

\* N. de R. T.: O título deste capítulo no original *Interesting Times* é outro jogo de palavras do autor: a palavra *times* pode se referir à operação de multiplicação (traduzida como “vezes”) – muito usada no capítulo – mas também poderia ser traduzida como “tempos”. O título assim seria “Tempos Interessantes” ou “Vezes Interessantes”, ambos interessantes, mas optamos pelo primeiro...

**Dollar**

```
Money times(int multiplier) {  
    return Money.dollar(amount * multiplier);  
}
```

Não há um jeito óbvio de fazê-las idênticas. Às vezes você tem que voltar atrás para seguir em frente, como quando resolvemos um Cubo de Rubik. O que acontece se nós otimizarmos os métodos fábrica? (Eu sei, eu sei, nós chamamos o método fábrica pela primeira vez apenas um capítulo atrás. Frustrante, não é?)

**Franc**

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier, "CHF");  
}
```

**Dollar**

```
Money times(int multiplier) {  
    return new Dollar(amount * multiplier, "USD");  
}
```

Em Franc, entretanto, sabemos que a variável de instância de moeda é sempre "CHF", então podemos escrever:

**Franc**

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier, currency);  
}
```

Isso funciona. O mesmo truque funciona em Dollar:

**Dollar**

```
Money times(int multiplier) {  
    return new Dollar(amount * multiplier, currency);  
}
```

Estamos quase lá. Realmente importa se temos um Franc ou um Money? Poderíamos pensar sobre isso cuidadosamente, dado nosso conhecimento sobre o sistema, mas temos código limpo e temos testes que nos dão confiança de que o código limpo funciona. Em vez de usar minutos de raciocínio suspeito, podemos apenas perguntar para o computador fazendo as mudanças e rodando os testes. No ensino de TDD, eu vejo essa situação o tempo todo – excelentes engenheiros de software pensando em uma questão durante 5 ou 10 minutos que o computador poderia responder em 15 segundos. Sem os testes, você não tem escolha, você tem de pensar. Com os testes, você pode decidir se um experimento responderia a questão mais rápido. Às vezes, você deveria apenas perguntar para o computador.

Para rodar nosso experimento, mudamos `Franc.times()` para retornar um `Money`:

### Franc

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

O compilador nos diz que `Money` deve ser uma classe concreta:

### Money

```
class Money
Money times(int amount) {
    return null;
}
```

E nós temos uma barra vermelha. A mensagem de erro diz, “expected:<Money.Franc@31aebf> but was:<Money.Money@478a43>”. Não tão útil como talvez gostaríamos. Podemos definir `toString()` para nos dar uma mensagem de erro melhor:

### Money

```
public String toString() {
    return amount + " " + currency;
}
```

Uau! Código sem um teste? Você pode fazer isso? Poderíamos certamente ter escrito um teste para `toString()` antes de codificá-lo. Contudo:

- Estamos prestes a ver os resultados na tela.
- Devido a `toString()` ser usado apenas para saída de debug, o risco de falha é baixo.
- Já temos uma barra vermelha e preferimos não escrever um teste quando temos uma barra vermelha.

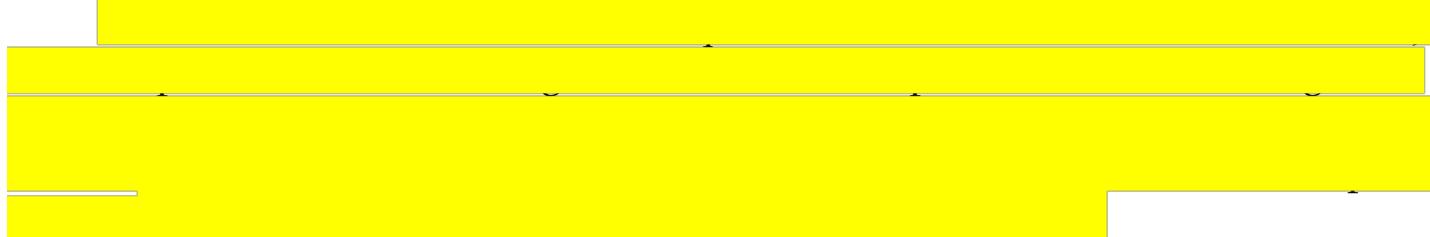
Exceção anotada.

Agora, a mensagem de erro diz: “expected:<10 CHF> but was:<10 CHF>”. Está um pouco melhor, mas ainda confuso. Nós temos o dado certo na resposta, mas a classe está errada – `Money` em vez de `Franc`. O problema está na nossa implementação de `equals()`:

### Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

Realmente deveríamos verificar se as moedas são as mesmas, não se as classes são as mesmas.



Dessa vez, seremos conservadores. (Às vezes, sigo adiante e escrevo um teste no vermelho, mas não enquanto as crianças estão acordadas.)

### Franc

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier, currency);  
}
```

Isso nos leva de volta ao verde. A situação que tínhamos era um `Franc(10, "CHF")` e um `Money(10, "CHF")` que foram relatados não serem iguais. Podemos usar exatamente isso para nosso teste:

```
public void testDifferentClassEquality() {  
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));  
}
```

Falha, como esperado. O código de `equals()` deveria comparar moedas, não classes:

### Money

```
public boolean equals(Object object) {  
    Money money = (Money) object;  
    return amount == money.amount  
        && currency().equals(money.currency());  
}
```

Agora podemos retornar um `Money` de um `Franc.times()` e ainda passar nos testes:

### Franc

```
Money times(int multiplier) {  
    return new Money(amount * multiplier, currency);  
}
```

O mesmo funcionará para `Dollar.times()`?

### Dollar

```
Money times(int multiplier) {  
    return new Money(amount * multiplier, currency);  
}
```



Sim! Agora que as duas implementações são idênticas, podemos movê-las para cima.

### Money

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

~~Tornar “quantidade” privada~~

~~Efeitos colaterais em Dollar?~~

~~Arredondamento de dinheiro?~~

~~equals()~~

hashCode()

Igualdade de null

Igualdade de objeto

~~5 CHF \* 2 = 10 CHF~~

Duplicação de Dólar/Franco

Igualdade comum

~~Multiplicação comum~~

~~Comparar Francos com Dólares~~

~~Moeda?~~

Deletar testFrancMultiplication?

Com a multiplicação no lugar, estamos preparados para eliminar as subclases estúpidas. Para revisar:

- Conciliamos dois métodos – `times()` – pela otimização dos métodos que eles chamavam e pela substituição de constantes por variáveis.
- Escrevemos um `toString()` sem um teste apenas para nos ajudar a depurar.
- Tentamos uma mudança (retornar `Money` em vez de `Franc`) e deixamos os testes nos dizerem se funcionou.
- Retrocedemos um experimento e escrevemos outro teste. Fazer o teste funcionar fez o experimento funcionar.

# CAPÍTULO 11

## A Raiz de Todo o Mal

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

~~Tornar “quantidade” privada~~

~~Efeitos colaterais em Dollar?~~

~~Arredondamento de dinheiro?~~

~~equals()~~

~~hashCode()~~

~~Igualdade de null~~

~~Igualdade de objeto~~

~~5 CHF \* 2 = 10 CHF~~

~~Duplicação de Dólar/Franco~~

~~Igualdade comum~~

~~Multiplicação comum~~

~~Comparar Francos com Dólares~~

~~Moeda?~~

~~Deletar testFrancMultiplication?~~

As duas subclasses, `Dollar` e `Franc`, têm somente seus construtores. Mas, como um construtor não é motivo suficiente para termos uma subclass, queremos deletar as subclasses.

Podemos substituir referências para as subclasses por referências pela superclasse sem mudar o sentido do código. Primeiro, `Franc`:

### Franc

```
static Money franc(int amount) {  
    return new Money(amount, "CHF");  
}
```

Então, Dollar:

### Dollar

```
static Money dollar(int amount) {
    return new Money(amount, "USD");
}
```

Agora, não há referências para Dollar, então podemos deletá-lo. Franc, por outro lado, ainda tem uma referência no teste que acabamos de escrever.

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

A igualdade está suficientemente coberta em outros lugares para podermos deletar esse teste? Olhando para o outro teste de igualdade,

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

parece que temos os casos para igualdade bem cobertos – muito bem cobertos, realmente. Podemos deletar as terceira e quarta declarações, pois elas duplicam o exercício das primeira e segunda declarações:

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~

~~\$5 \* 2 = \$10~~

~~Tornar “quantidade” privada~~

~~Efeitos colaterais em Dollar?~~

~~Arredondamento de dinheiro?~~

~~equals()~~

~~hashCode()~~

~~Igualdade de null~~

~~Igualdade de objeto~~

~~5 CHF \* 2 = 10 CHF~~

~~Duplicação de Dólar/Franco~~

~~Igualdade comum~~

~~Multiplicação comum~~

~~Comparar Francos com Dólares~~

~~Moeda?~~

~~Deletar testFrancMultiplication?~~

O teste que escrevemos, que nos força a comparar moedas em vez de classes, faz sentido apenas se há múltiplas classes. Devido a estarmos tentando eliminar a classe Franc, um teste para garantir que o sistema funciona se há uma classe Franc é um fardo e não ajuda. `testDifferentClassEquality()` vai embora, e Franc vai com ele. Similarmente, há testes separados para multiplicação de dólares e francos. Olhando o código, podemos ver que não há diferença na lógica, no momento, baseados na moeda (havia uma diferença quando havia duas classes). Podemos deletar `testFrancMultiplication()` sem perder qualquer confiança no comportamento do sistema.

Com a classe única no lugar, estamos prontos para enfrentar adição. Primeiro, para revisar:

- Terminamos eviscerando subclasses e deletando-as.
- Eliminamos testes que faziam sentido com a estrutura do código antigo, mas que eram redundantes com a nova estrutura de código.



# CAPÍTULO 12

---

## Adição, Finalmente

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

É um novo dia, e nossa lista de tarefas tornou-se um pouco bagunçada, então copiaremos os itens pendentes para uma nova lista. (Eu gosto de copiar fisicamente itens por fazer para uma nova lista. Se há muitos itenzinhos, eu tendo a fazê-los em vez de copiá-los. Coisinhas que de outra forma poderiam se acumular são resolvidas só porque sou preguiçoso. Jogue com seus pontos fortes.)

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1  
 $\$5 + \$5 = \$10$

Não estou certo de como escrever a história de toda a adição, então começaremos com um exemplo mais simples:  $\$5 + \$5 = \$10$ .

```
public void testSimpleAddition() {  
    Money sum= Money.dollar(5).plus(Money.dollar(5));  
    assertEquals(Money.dollar(10), sum);  
}
```

Poderíamos fazer de conta a implementação apenas retornando “`Money.dollar(10)`”, mas a implementação parece óbvia. Tentaremos:

```
Money  
Money plus(Money addend) {  
    return new Money(amount + addend.amount, currency);  
}
```

(Em geral, começarei acelerando as implementações para salvar árvores e manter seu interesse. Onde o projeto não é óbvio, vou utilizar fazer de conta a implementação e refatorarei. Com isso espero que você veja como TDD dá a você controle sobre o tamanho dos passos.)

Tendo dito que estaria indo mais rápido, eu imediatamente irei mais devagar – não em obter os testes funcionando, mas em escrever o teste em si. Há vezes e testes que clamam por uma reflexão cuidadosa. Como vamos representar aritmética multi-moeda? Essa é uma dessas vezes que requerem reflexão cuidadosa.

A mais difícil restrição de projeto é querermos que a maior parte do código no sistema esteja inconsciente de que está, potencialmente, lidando com múltiplas moedas. Uma estratégia possível é converter imediatamente todos os valores de dinheiro para uma moeda de referência. (Deixarei você imaginar qual moeda americana imperialista de referência os programadores geralmente escolhem.) Entretanto, isso não permite à taxa de câmbio variar facilmente.

Em vez disso, gostaríamos de uma solução que nos deixasse representar convenientemente múltiplas taxas de câmbio e ainda permitir que a maioria das expressões similares a expressões aritméticas parecesse com, bem, expressões aritméticas.

Objetos para o resgate. Quando o objeto que temos não se comporta da forma que queremos, fazemos outro objeto com o mesmo protocolo externo (um impostor), mas com uma implementação diferente.

Isso, provavelmente, soa um pouco como mágica. Como sabemos que temos de pensar em criar um impostor aqui? Não vou enganar você – não há fórmula para isso. Ward Cunningham veio com o “truque” há uma década, e eu não o vi ainda independentemente duplicado, logo, deve ser um belo e complicado truque. TDD não pode garantir que teremos flashes de ideias no momento certo. Contudo, testes que dão confiança e código fatorado cuidadosamente nos dão preparação para ideias e preparação para aplicá-las quando surgirem.

A solução é criar um objeto que age como um `Money`, mas representa a soma de dois `Money`s. Tentei várias metáforas diferentes para explicar essa ideia. Uma é tratar a soma como uma *carteira*: você pode ter várias notas diferentes de diferentes denominações e moedas na mesma carteira.

Outra metáfora é *expressão*, como em “ $(2 + 3) * 5$ ”, ou, no nosso caso, “ $(\$2 + 3\text{CHF}) * 5$ ”. Um `Money` é a forma atômica de uma expressão. Operações resultam em `Expressions`, uma das quais será uma `Sum`. Uma vez que a operação (como a soma do valor de uma carteira) está completa, a `Expression` resultante pode ser reduzida de volta a uma moeda simples dado um conjunto de taxas de câmbio.

Aplicando essa metáfora em nosso teste, sabemos que terminamos com:

```
public void testSimpleAddition() {  
    ...  
    assertEquals(Money.dollar(10), reduced);  
}
```

A `reduced Expression` é criada pela aplicação de taxas de câmbio a uma `Expression`. O que, no mundo real, aplica taxas cambiais? Um *banco*. Gostaríamos de estar aptos a escrever:

```
public void testSimpleAddition() {
    ...
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

(É um pouco estranho misturar as metáforas *banco* e *expressão*. Teremos, primeiro, toda a história contada e então veremos o que podemos fazer sobre seu valor literário.)

Fizemos uma importante decisão de projeto aqui. Poderíamos facilmente ter escrito ...reduce= sum.reduce("USD", bank). Por que fazer o banco responsável? Uma resposta é: “Essa é a primeira coisa que me veio à cabeça”, mas isso não é muito informativo. Por que me veio à cabeça que a redução deveria ser responsabilidade do banco em vez da expressão? No momento, estou ciente disto:



Essas não parecem ser razões suficientes para pender a balança permanentemente, mas são suficientes para eu começar nessa direção. Também estou perfeitamente disposto a mover responsabilidades para a redução de Expression, se isso fizer com que os Banks não precisem estar envolvidos.

O Bank, em nosso exemplo simples, não precisa fazer realmente nada. Enquanto temos um objeto, estamos bem:

```
public void testSimpleAddition() {
    ...
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

A soma de dois Moneys deveria ser uma Expression:

```
public void testSimpleAddition() {
    ...
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Ao menos sabemos com certeza como obter cinco dólares:

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

Como fazemos para isso compilar? Precisamos de uma interface `Expression` (poderíamos ter uma classe, mas uma interface é ainda mais leve):

### **Expression**

```
interface Expression
```

`Money.plus()` precisa retornar uma `Expression`,

### **Money**

```
Expression plus(Money addend) {  
    return new Money(amount + addend.amount, currency);  
}
```

o que significa que `Money` tem que implementar `Expression` (o que é fácil, pois não há operações ainda):

### **Money**

```
class Money implements Expression
```

Precisamos de uma classe `Bank` vazia,

### **Bank**

```
class Bank
```

a qual precisa de um stub `reduce()`:

### **Bank**

```
Money reduce(Expression source, String to) {  
    return null;  
}
```

Agora compila, e falha miseravelmente. Iúpi! Progresso! Podemos facilmente fazer de conta a implementação:

### **Bank**

```
Money reduce(Expression source, String to) {  
    return Money.dollar(10);  
}
```

Estamos de volta à barra verde e preparados para refatorar. Primeiro, para revisar:

- Reduzimos um grande teste para um menor que representava progresso ( $\$5 + 10 \text{ CHF}$  para  $\$5 + \$5$ ).
- Refletimos cuidadosamente sobre as possíveis metáforas para nossa computação.
- Reescrevemos nossos testes anteriores baseados em nossa nova metáfora.
- Conseguimos fazer o teste compilar rapidamente.
- Fizemos ele rodar.
- Esperamos com um pouco de agitação a refatoração necessária para fazer a implementação real.

# CAPÍTULO 13

---

## Faça-o

```
$5 + 10 CHF = $10 se a taxa é 2:1  
$5 + $5 = $10
```

Nós não podemos marcar nosso teste  $\$5 + \$5$  como feito até que tenhamos removido toda a duplicação. Não temos código duplicado, mas temos, sim, duplicação de dados – o  $\$10$  na implementação faz de conta:

### Bank

```
Money reduce(Expression source, String to) {  
    return Money.dollar(10);  
}
```

é realmente o mesmo  $\$5 + \$5$  no teste:

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

Antes, quando tínhamos uma implementação faz de conta, era óbvio como voltar atrás e fazer a implementação real. Simplesmente, era uma questão de substituir constantes por variáveis. Dessa vez, entretanto, não é óbvio para mim como voltar atrás. Então, mesmo que pareça meio especulativo, seguiremos em frente.

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1  
 $\$5 + \$5 = \$10$   
 Retornar Money de  $\$5 + \$5$

Primeiro, `Money.plus()` precisa retornar uma `Expression` real – uma `Sum`, não apenas um `Money`. (Talvez depois otimizaremos o caso especial de somar duas moedas idênticas, mas isso é mais tarde.)

A soma de dois `Money`s deveria ser uma `Sum`:

```
public void testPlusReturnsSum() {
    Money five= Money.dollar(5);
    Expression result= five.plus(five);
    Sum sum= (Sum) result;
    assertEquals(five, sum.augend);
    assertEquals(five, sum.addend);
}
```

(Você sabia que o primeiro argumento de adições, em inglês, é chamado *augend*? Eu não sabia até escrever isso. Alegria nerd.)

O teste acima não é o que eu esperaria que vivesse muito. Está profundamente preocupado com a implementação para nossa operação em vez de seu comportamento externo visível. Entretanto, se fizermos ele funcionar, esperamos ter avançado um passo em direção à nossa meta. Para tê-lo compilando, tudo o que precisamos é uma classe `Sum` com dois campos, `augend` and `addend`\*:

### Sum

```
class Sum {
    Money augend;
    Money addend;
}
```

Isso nos dá uma `ClassCastException`, pois `Money.plus()` está retornando um `Money`, não uma `Sum`:

### Money

```
Expression plus(Money addend) {
    return new Sum(this, addend);
}
```

`Sum` precisa de um construtor:

### Sum

```
Sum(Money augend, Money addend) {
```

---

\* N. de R. T.: *Augend* é o nome em inglês da primeira parcela da adição; *Addend* é o nome em inglês da segunda parcela da adição. Ambos referem-se a um número ou quantidade à qual outro número ou quantidade é somado para produzir o resultado (soma).

E `Sum` precisa ser um tipo de `Expression`:

### Sum

```
class Sum implements Expression
```

Agora, o sistema compila de novo, mas o teste ainda falha: dessa vez porque o construtor `Sum` não está iniciando os campos. (Poderíamos fazer de conta a implementação pela inicialização dos campos, mas eu disse que andaria mais rápido.)

### Sum

```
Sum(Money augend, Money addend) {
    this.augend= augend;
    this.addend= addend;
}
```

Agora, `Bank.reduce()` está recebendo uma `Sum`. Se as moedas na `Sum` são as mesmas e a moeda alvo é também a mesma, então o resultado deveria ser um `Money` cuja quantidade é a soma das quantidades:

```
public void testReduceSum() {
    Expression sum= new Sum(Money.dollar(3), Money.dollar(4));
    Bank bank= new Bank();
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(7), result);
}
```

Eu escolhi cuidadosamente parâmetros que estragariam o teste existente. Quando reduzimos uma `Sum`, o resultado (sob essas circunstâncias simplificadas) deveria ser um `Money` cuja quantidade é a soma das quantidades dos dois `Money`s e cuja moeda é a moeda que estamos reduzindo.

### Bank

```
Money reduce(Expression source, String to) {
    Sum sum= (Sum) source;
    int amount= sum.augend.amount + sum.addend.amount;
    return new Money(amount, to);
}
```

Isso é imediatamente feio sob dois aspectos:

- A conversão. Esse código não deveria trabalhar com qualquer `Expression`.
- Os campos públicos e os dois níveis de referências para isso.

Bastante fácil de corrigir. Primeiro, podemos mover o corpo do método para `Sum` e nos livrarmos de alguns campos visíveis. Estamos “certos” de que precisaremos do `Bank` como parâmetro no futuro, mas isso é refatoração pura e simples, então vamos deixá-lo de fora. (Na verdade, só agora o coloquei, porque eu “sabia” que precisaria dele – vergonha, devia me envergonhar.)

### Bank

```
Money reduce(Expression source, String to) {
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

### Sum

```
public Money reduce(String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1

$\$5 + \$5 = \$10$

Retornar Money de  $\$5 + \$5$

Bank.reduce( Money )

(O que levanta a questão de como iremos implementar, hum... testar Bank.reduce() quando o argumento é um Money.)

Já que a barra está verde e nada há mais óbvio para fazer com o código acima, vamos escrever um teste:

```
public void testReduceMoney() {
    Bank bank= new Bank();
    Money result= bank.reduce(Money.dollar(1), "USD");
    assertEquals(Money.dollar(1), result);
}
```

### Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money) return (Money) source;
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Feio, feio, feio. Entretanto, agora temos uma barra verde e a possibilidade de refatoração. Deveríamos estar usando polimorfismo em vez de verificarmos classes explicitamente a qualquer hora. Devido a Sum implementar reduce(String), se Money a implementou também, poderíamos então adicioná-la à interface de Expression.

### Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money)
        return (Money) source.reduce(to);
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

**Money**

```
public Money reduce(String to) {
    return this;
}
```

Se adicionarmos `reduce(String)` à interface de `Expression`,

**Expression**

```
Money reduce(String to);
```

então podemos eliminar todas aquelas conversões feias e verificações de classe:

**Bank**

```
Money reduce(Expression source, String to) {
    return source.reduce(to);
}
```

Não estou inteiramente satisfeito pelo nome do método ser o mesmo em `Expression` e em `Bank`, mas tendo diferentes parâmetros em cada um. Eu nunca encontrei uma solução geral satisfatória para esse problema em Java. Em linguagens com parâmetros com palavras-chave, comunicar a diferença entre `Bank.reduce(Expression, String)` e `Expression.reduce(String)` é bem suportado pela sintaxe da linguagem. Com parâmetros posicionais, não é tão fácil fazer o código falar por nós sobre como são diferentes.

$\$5 + 10 \text{ CHF} = \$10$  se a taxa é 2:1  
 $\$5 + \$5 = \$10$   
 Retornar Money de  $\$5 + \$5$   
~~Bank.reduce(Money)~~  
 Reduzir Money com conversões  
 Reduce(Bank, String)

Em seguida, trocaremos realmente de uma moeda para outra. Primeiro, para revisar:

- Não marcamos um teste como feito, pois a duplicação não tinha sido eliminada.
- Seguimos em frente em vez de voltarmos atrás para tornarmos real a implementação.
- Escrevemos um teste para forçar a criação de um objeto que nós esperávamos precisar depois (`Sum`).
- Começamos a implementar mais rápido (o construtor de `Sum`).
- Implementamos código com conversões em um lugar, e então movemos o código para onde pertencia uma vez que os testes estavam rodando.
- Introduzimos polimorfismo para eliminar verificação explícita de classes.

# CAPÍTULO 14

---

## Mudança

```
$5 + 10 CHF = $10 se a taxa é 2:1  
$5 + $5 = $10  
Retornar Money de $5 + $5  
Bank.reduce(Money)  
Reducir Money com conversões  
Reduce(Bank, String)
```

Vale a pena acolher a mudança (especialmente se você tem um livro com “acolha as mudanças” no título\*). Aqui, entretanto, estamos pensando em uma forma muito mais simples de mudança – temos dois francos e queremos um dólar. Isso já soa como um caso de teste:

```
public void testReduceMoneyDifferentCurrency() {  
    Bank bank= new Bank();  
    bank.addRate("CHF", "USD", 2);  
    Money result= bank.reduce(Money.franc(2), "USD");  
    assertEquals(Money.dollar(1), result);  
}
```

Quando converto francos para dólares, eu divido por dois. (Ainda estamos ignorando todos aqueles desagradáveis problemas numéricos.) Podemos fazer a barra ficar verde fazendo algo feio:

### Money

```
public Money reduce(String to) {  
    int rate = (currency.equals("CHF") && to.equals("USD"))
```

---

\* N. de R. T.: Alusão ao (sub)título do livro de Kent Beck sobre XP: BECK, Kent. *Programação Extrema explicada: acolha as mudanças*. Porto Alegre: Bookman, 2004.

```
? 2
: 1;
return new Money(amount / rate, to);
}
```

Agora, de repente, Money conhece taxas de câmbio. Eca! O Bank deveria ser o único lugar em que nos preocupamos com taxas de câmbio. Teremos que passar o Bank como um parâmetro para Expression.reduce(). (Vê? Nós *sabíamos* que precisaríamos disso e estávamos certos. Nas palavras do avô em *A Princesa Prometida*, “Você é muito esperto...”) Primeiro, o chamador:

### **Bank**

```
Money reduce(Expression source, String to) {
    return source.reduce(this, to);
}
```

Então os implementadores:

### **Expression**

```
Money reduce(Bank bank, String to);
```

### **Sum**

```
public Money reduce(Bank bank, String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

### **Money**

```
public Money reduce(Bank bank, String to) {
    int rate = (currency.equals("CHF") && to.equals("USD"))
    ? 2
    : 1;
    return new Money(amount / rate, to);
}
```

Os métodos têm que ser públicos, pois métodos na interface têm que ser públicos (por alguma razão excelente, estou certo).

Agora conseguimos calcular a taxa no Bank:

### **Bank**

```
int rate(String from, String to) {
    return (from.equals("CHF") && to.equals("USD"))
    ? 2
    : 1;
}
```

E pedir ao bank a taxa correta:

### **Money**

```
public Money reduce(Bank bank, String to) {
    int rate = bank.rate(currency, to);
```

```

        return new Money(amount / rate, to);
    }
}

```

Aquele maldito 2 ainda aparece no teste e no código. Para nos livrarmos dele, precisamos manter uma tabela de taxas no Bank e procurar uma taxa quando precisarmos dela. Poderíamos usar uma tabela hash que mapeia pares de moedas em taxas. Podemos usar um array de elementos duplos contendo as duas moedas como chaves? Uma verificação com `Array.equals()` consegue ver se os elementos são iguais?

```

public void testArrayEquals() {
    assertEquals(new Object[] {"abc"}, new Object[] {"abc"});
}

```

Não. O teste falha, e, assim, temos que criar um objeto real para a chave:

#### **Pair**

```

private class Pair {
    private String from;
    private String to;

    Pair(String from, String to) {
        this.from= from;
        this.to= to;
    }
}

```

Devido a estarmos usando Pairs como chaves, teremos que implementar `equals()` e `hashCode()`. Não vou escrever testes para eles, pois estamos escrevendo esse código no contexto de uma refatoração. Se tivermos o êxito da refatoração e todos os testes rodarem, então esperamos que o código tenha sido exercitado. Se estivesse programando com alguém que não visse exatamente aonde estávamos indo com isso, ou se a lógica se tornasse um pouco menos complexa, eu começaria a escrever testes separados.

#### **Pair**

```

public boolean equals(Object object) {
    Pair pair= (Pair) object;
    return from.equals(pair.from) && to.equals(pair.to);
}

public int hashCode() {
    return 0;
}

```

0 é um valor de hash terrível, mas tem a vantagem de ser fácil de implementar e de nos deixar fazer funcionar rapidamente. A procura por moeda parecerá uma busca linear. Depois, quando tivermos balaios de moedas, podemos fazer um trabalho mais profundo com dados de uso real.

Precisamos de algum lugar para armazenar as taxas:

#### **Bank**

```
private Hashtable rates= new Hashtable();
```

Precisamos definir a taxa quando dito:

**Bank**

```
void addRate(String from, String to, int rate) {  
    rates.put(new Pair(from, to), new Integer(rate));  
}
```

E então podemos procurar a taxa quando pedido:

**Bank**

```
int rate(String from, String to) {  
    Integer rate= (Integer) rates.get(new Pair(from, to));  
    return rate.intValue();  
}
```

Espere um minuto! Temos uma barra vermelha. O que aconteceu? Uma breve bisbilhotada nos diz que se pedirmos a taxa de USD para USD, esperamos que o valor seja 1. Como isso foi uma surpresa, vamos escrever um teste para comunicar o que descobrimos:

```
public void testIdentityRate() {  
    assertEquals(1, new Bank().rate("USD", "USD"));  
}
```

Agora temos três erros, mas esperamos que todos eles sejam corrigidos com uma mudança:

**Bank**

```
int rate(String from, String to) {  
    if (from.equals(to)) return 1;  
    Integer rate= (Integer) rates.get(new Pair(from, to));  
    return rate.intValue();  
}
```

Barra verde!

---

```
$5 + 10 CHF = $10 se a taxa é 2:1  
$5 + $5 = $10  
Retornar Money de $5 + $5  
Bank.reduce( Money )  
Reducir Money com conversões  
Reduce(Bank, String)
```

---

Em seguida, implementaremos nosso último grande teste,  $\$5 + 10 \text{ CHF}$ . Várias técnicas significativas escorregaram nesse capítulo. Por agora, para revisar:

- Adicionamos um parâmetro, em segundos, que esperávamos ser necessário.
- Refatoramos a duplicação de dados entre código e testes.
- Escrevemos um teste (`testArrayEquals`) para verificar uma suposição sobre a operação de Java.
- Introduzimos uma classe privada auxiliar sem distinguir testes dela mesma.
- Cometemos um erro em uma refatoração e escolhemos seguir em frente, escrevendo outro teste para isolar o problema.

# CAPÍTULO 15

## Moedas Misturadas

```
$5 + 10 CHF = $10 se a taxa é 2:1  
$5 + $5 = $10  
Retornar Money de $5 + $5  
Bank.reduce(Money)  
Reducir Money com conversões  
Reduce(Bank, String)
```

Agora nós finalmente estamos preparados para adicionar o teste que começou isso tudo,  $\$5 + 10 \text{ CHF}$ :

```
public void testMixedAddition() {  
    Expression fiveBucks= Money.dollar(5);  
    Expression tenFrancs= Money.franc(10);  
    Bank bank= new Bank();  
    bank.addRate("CHF", "USD", 2);  
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");  
    assertEquals(Money.dollar(10), result);  
}
```

Isso é o que gostaríamos de escrever. Infelizmente, há uma série de erros de compilação. Quando estávamos generalizando de `Money` para `Expression`, deixamos um monte de pontas soltas largadas em volta. Eu estava preocupado com elas, mas não quis perturbar você. Chegou a hora.

Não seremos capazes de fazer o teste anterior compilar rapidamente. Nós faremos a primeira mudança, que provocará a próxima e a próxima. Temos dois caminhos a seguir. Podemos fazê-lo funcionar rapidamente escrevendo um teste mais específico e, então, generalizando, ou podemos confiar que nosso compilador não nos deixará cometer erros. Estou com você – vamos devagar (na prática, eu provavelmente apenas arrumaria as mudanças em cascata uma de cada vez).

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Money tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

O teste não funciona. Temos 15 USD em vez de 10 USD. É como se `Sum.reduce()` não estivesse reduzindo os argumentos. Não é:

### Sum

```
public Money reduce(Bank bank, String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

Se reduzirmos ambos os argumentos, o teste deverá passar:

### Sum

```
public Money reduce(Bank bank, String to) {
    int amount= augend. reduce(bank, to).amount
        + addend. reduce(bank, to).amount;
    return new Money(amount, to);
}
```

E ele passa. Agora podemos começar a catar em Moneys o que deveria ser Expressions. Para evitar o efeito cascata, começaremos pelas bordas e trabalharemos nosso caminho de volta ao caso de teste. Por exemplo, o *augend* e o *addend* podem agora ser Expressions:

### Sum

```
Expression augend;
Expression addend;
```

Os argumentos para o construtor de `Sum` também podem ser Expressions:

### Sum

```
Sum(Expression augend, Expression addend) {
    this.augend= augend;
    this.addend= addend;
}
```

(`Sum` está começando a me lembrar de Composite\*, mas não tanto que eu queira generalizar. No momento, queremos uma `Sum` com dois parâmetros diferentes, em-

---

\* N. de R. T.: Padrão GoF Composite – para representação e tratamento uniforme de estruturas hierárquicas recursivas; ver mais detalhes em GAMMA, Erich et al. *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Porto Alegre: Bookman, 2000.

bora eu esteja pronto para transformá-la.) Por ora, isso é tudo para `Sum` – e o que temos para o `Money`?

O argumento para `plus()` pode ser uma `Expression`:

### **Money**

```
Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

`Times()` pode retornar uma `Expression`:

### **Money**

```
Expression times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

Isso sugere que `Expression` deveria incluir as operações `plus()` e `times()`. Por ora, isso é o suficiente para `Money`. Agora podemos mudar o argumento para `plus()` em nosso caso de teste:

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

Quando mudamos `fiveBucks` para uma `Expression`, temos que fazer muitas mudanças. Felizmente, temos a lista de tarefas do compilador para manter-nos focados. Primeiro, fazemos a mudança:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

Estamos dizendo, educadamente, que `plus()` não está definido para `Expressions`. Nós o definimos:

### **Expression**

```
Expression plus(Expression addend);
```

E, então, temos que adicioná-lo a `Money` e a `Sum`. `Money`? Sim, ele tem que ser público em `Money`:

### **Money**

```
public Expression plus(Expression addend) {
```

```
    return new Sum(this, addend);
}
```

Vamos apenas fazer um stub da implementação em `Sum` e adicioná-la em nossa lista de tarefas:

### Sum

```
public Expression plus(Expression addend) {
    return null;
}
```

---

```
$5 + 10 CHF = $10 se a taxa é 2:1
$5 + $5 = $10
Retornar Money de $5 + $5
Bank.reduce( Money )
Reducir Money com conversões
Reduce(Bank, String)
Sum.plus
Expression.times
```

---

Agora que o programa compila, todos os testes rodam.

Estamos preparados para terminar a generalização de `Money` para `Expression`. Mas, primeiro, para revisar:

- Escrevemos o teste que queríamos, então recuamos para fazê-lo possível em um passo.
- Generalizamos (usando uma declaração mais abstrata) das folhas de volta à raiz (o caso de teste).
- Seguimos o compilador quando fizemos uma mudança (`Expression fiveBucks`), o que causou mudanças em cascata (`added plus()` em `Expression`, e assim por diante).

# CAPÍTULO 16

---

## Abstração, Finalmente

```
$5 + 10 CHF = $10 se a taxa é 2:1  
$5 + $5 = $10  
Retornar Money de $5 + $5  
Bank.reduce(Money)  
Reducir Money com conversões  
Reduce(Bank, String)  
Sum.plus  
Expression.times
```

Precisamos implementar `Sum.plus()` para terminar `Expression.plus`, e, então, precisamos de `Expression.times()`, e, então, teremos terminado todo o exemplo. Aqui está o teste para `Sum.plus()`:

```
public void testSumPlusMoney() {  
    Expression fiveBucks= Money.dollar(5);  
    Expression tenFrancs= Money.franc(10);  
    Bank bank= new Bank();  
    bank.addRate("CHF", "USD", 2);  
    Expression sum= new Sum(fiveBucks, tenFrancs).plus(fiveBucks);  
    Money result= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(15), result);  
}
```

Poderíamos ter criado uma `Sum` adicionando `fiveBucks` e `tenFrancs`, mas a forma acima, na qual explicitamente criamos a `Sum`, comunica mais diretamente. Estamos escrevendo esses testes não apenas para tornar nossa experiência de programação mais divertida e gratificante, mas também como uma Pedra de

Roseta\* para gerações futuras apreciarem nossa genialidade. Pense, ó pense, em nossos leitores.

O teste, nesse caso, é mais longo que o código. O código é o mesmo código em Money. (Será que eu ouvi uma classe abstrata ao longe?)

### Sum

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~  
~~\$5 + \$5 = \$10~~  
 Retornar Money de \$5 + \$5  
~~Bank.reduce( Money )~~  
~~Reducir Money com conversões~~  
~~Reduce(Bank, String)~~  
~~Sum.plus~~  
~~Expression.times~~

Você terminará, aproximadamente, com o mesmo número de linhas no código do teste e no código modelo quando implementar TDD. Para TDD fazer sentido em termos de economia, você precisará ser capaz de escrever o dobro de linhas por dia do que escrevia antes, ou escrever metade daquela quantidade de linhas para a mesma funcionalidade. Terá que medir e ver que efeitos TDD tem por sua própria prática. Certifique-se, no entanto, de considerar tempos de depuração, integração e explicação em suas métricas.

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~  
~~\$5 + \$5 = \$10~~  
 Retornar Money de \$5 + \$5  
~~Bank.reduce( Money )~~  
~~Reducir Money com conversões~~  
~~Reduce(Bank, String)~~  
~~Sum.plus~~  
~~Expression.times~~

Se conseguirmos fazer `Sum.times()` funcionar, então declarar `Expression.times()` será um passo só. O teste é:

```
public void testSumTimes() {
    Expression fiveBucks= Money.dollar(5);
```

---

\* N. de R. T: Pedra de Roseta é o nome da pedra encontrada em Roseta, no Egito, contendo o mesmo texto escrito em 3 línguas (grego clássico, demótico egípcio e hieróglifos) e que foi fundamental para a decifração da escrita hieroglífica por J-F. Champollion em 1822.

```

Expression tenFrancs= Money.franc(10);
Bank bank= new Bank();
bank.addRate("CHF", "USD", 2);
Expression sum= new Sum(fiveBucks, tenFrancs).times(2);
Money result= bank.reduce(sum, "USD");
assertEquals(Money.dollar(20), result);
}

```

Novamente, o teste é mais longo que o código. (Vocês, nerds do JUnit, saberão como corrigir isso – o resto de vocês terá que ler Fixture\*.)

### Sum

```

Expression times(int multiplier) {
    return new Sum(augend.times(multiplier), addend.times(multiplier));
}

```

Devido a abstrairmos *augend* e *addend* para Expressions no último capítulo, temos agora que declarar *times()* em Expression para o código compilar:

### Expression

```
Expression times(int multiplier);
```

Isso nos força a aumentar a visibilidade de *Money.times()* e *Sum.times()*:

### Sum

```

public Expression times(int multiplier) {
    return new Sum(augend.times(multiplier), addend.times(multiplier));
}

```

### Money

```

public Expression times(int multiplier) {
    return new Money(amount * multiplier, currency);
}

```

~~\$5 + 10 CHF = \$10 se a taxa é 2:1~~  
~~\$5 + \$5 = \$10~~  
~~Retornar Money de \$5 + \$5~~  
~~Bank.reduce(Money)~~  
~~Reducir Money com conversões~~  
~~Reduce(Bank, String)~~  
~~Sum.plus~~  
~~Expression.times~~

E isso funciona.

---

\* N. de R. T.: Fixture é o conjunto de recursos ou dados comuns necessários para a execução dos testes. Em geral, as classes são instanciadas ANTES da execução de cada método.

A única ponta solta para amarrar é experimentar o retorno de um Money quando somamos \$5 + \$5. O teste seria:

```
public void testPlusSameCurrencyReturnsMoney() {  
    Expression sum= Money.dollar(1).plus(Money.dollar(1));  
    assertTrue(sum instanceof Money);  
}
```

Esse teste é um pouco feio, pois está testando as entranhas da implementação e não o comportamento externamente visível dos objetos. Contudo, nos guiará a fazer as mudanças que precisamos fazer, e isso, afinal de contas, é somente um experimento. Aqui está o código que teríamos que modificar para fazê-lo funcionar:

### Money

```
public Expression plus(Expression addend) {  
    return new Sum(this, addend);  
}
```

```
$5 + 10 CHF = $10 se a taxa é 2.1  
$5 + $5 = $10  
Retornar Money de $5 + $5  
Bank.reduce( Money )  
Reducir Money com conversões  
Reduce(Bank, String)  
Sum.plus  
Expression.times
```

Não há maneira óbvia e limpa (não para mim; em todo o caso, estou certo de que você poderia pensar em alguma coisa) de checar a moeda do argumento se, e somente se, for um Money. O experimento falha, nós deletamos o teste (o qual não gostamos muito de qualquer jeito), e partimos.

Para revisar:

- Escrevemos um teste com os futuros leitores em mente.
- Sugerimos um experimento que compare TDD com seu estilo de programação atual.
- Mais uma vez tivemos mudanças de declarações ondulando pelo sistema, e mais uma vez seguimos os conselhos do compilador para corrigi-las.
- Tentamos um breve experimento, então o descartamos quando não deu certo.

# CAPÍTULO 17

---

## Retrospectiva Financeira

Vamos dar uma olhada para trás tanto no processo que usamos quanto nos resultados do exemplo financeiro. Vamos olhar:

- Qual é o próximo passo?
- Metáfora – O efeito dramático que a metáfora tem sobre a estrutura do projeto.
- Uso de JUnit – Quando rodamos os testes e como usamos o JUnit.
- Métricas de Código – Um resumo numérico do código resultante.
- Processo – Dizemos vermelho/verde/refatore, mas quanto trabalho há em cada passo?
- Qualidade de Teste – Como testes TDD empilham-se em comparação a métricas de teste convencionais?

---

### Qual é o próximo passo?

O código está terminado? Não. Há aquela duplicação desagradável entre `Sum.plus()` e `Money.plus()`. Se fizemos de `Expression` uma classe em vez de uma interface (não sendo a direção usual, visto que as classes, mais frequentemente, tornam-se interfaces), teríamos um habitat natural para o código comum.

Eu não acredito em “terminado”. TDD pode ser usado como uma maneira de lutar pela perfeição, mas esse não é seu uso mais eficaz. Se você tem um sistema grande, então as partes que você mexe o tempo todo deveriam ser rocha absolutamente sólida, assim você pode fazer mudanças diárias com confiança. Conforme você navega para a periferia do sistema, para partes que não mudam com frequê-

cia, os testes podem ser mais irregulares e o projeto mais feio sem interferir em sua confiabilidade.

Quando faço todas as tarefas óbvias, gosto de rodar um analisador de código\* como Small-Lint para Smalltalk. Muitas das sugestões que surgem eu já conheço ou não concordo. Todavia, analisadores automáticos não esquecem, logo, se não detecto uma implementação obsoleta, não tenho estresse. O analisador vai indicá-la.

Outra questão “qual é o próximo passo?” é: “De que testes adicionais eu preciso?” Às vezes, você pensa em um teste que “não deveria” funcionar, e ele funciona. Então você precisa descobrir o porquê. Às vezes, um teste que não deveria funcionar realmente não funciona, e você pode gravá-lo como uma limitação conhecida ou como trabalho a ser feito depois.

Finalmente, quando a lista está vazia é uma boa hora para rever o projeto. As palavras e os conceitos jogam juntos? Há duplicação que é difícil de eliminar dado o atual projeto? (Duplicação duradoura é um sintoma de projeto latente.)

---

## Métafora

A maior surpresa para mim na codificação do exemplo financeiro foi como saiu diferente dessa vez. Programei dinheiro em produção ao menos três vezes, que consigo pensar. Eu o usei como exemplo ilustrativo outra meia dúzia de vezes. Programei-o ao vivo no palco (relaxe, não é tão excitante como soa) outras quinze vezes. Codifiquei outras três ou quatro vezes me preparando para escrever (rascgando a Parte I e reescrevendo-a baseado em revisões anteriores). Então, enquanto estava escrevendo isso, pensei em usar *expressão* como metáfora, e o projeto foi em uma direção completamente diferente de antes.

Eu realmente não esperava que a metáfora fosse tão poderosa. Uma metáfora deveria ser apenas uma fonte de nomes, não deveria? Aparentemente não.

A metáfora que Ward Cunningham usou para “diversos dinheiros juntos com moedas potencialmente diferentes” foi um vetor, como um vetor matemático em que os coeficientes eram moedas em vez de  $x^2$ . Usei MoneySum por um tempo, então MoneyBag (que é melhor e físico), e, finalmente, Wallet (que é mais comum na experiência da maioria das pessoas). Todas essas metáforas implicam que a coleção de Moneys é plana. Por exemplo, “2 USD + 5 CHF + 3 USD” resultaria em “5 USD + 5 CHF”. Dois valores com a mesma moeda seriam mesclados.

A metáfora *expressão* me libertou de um monte de problemas desagradáveis sobre mesclar moedas duplicadas. O código surgiu mais claro e mais limpo do que

---

\* N. de R. T.: Analisador de código (ou, mais completamente, analisador estático de código) é uma ferramenta para análise automática de código, que detecta comandos/condições potencialmente suspeitos (por exemplo, retorno de função que não é usado, métodos definidos mas não usados) e gerando alertas (*warnings*) para a equipe. Ver mais informações e uma lista de analisadores (entre os quais o célebre Lint para a linguagem C e o Small-Lint citado no texto) na wikipedia (na inglesa, procure por *static code analysis*).

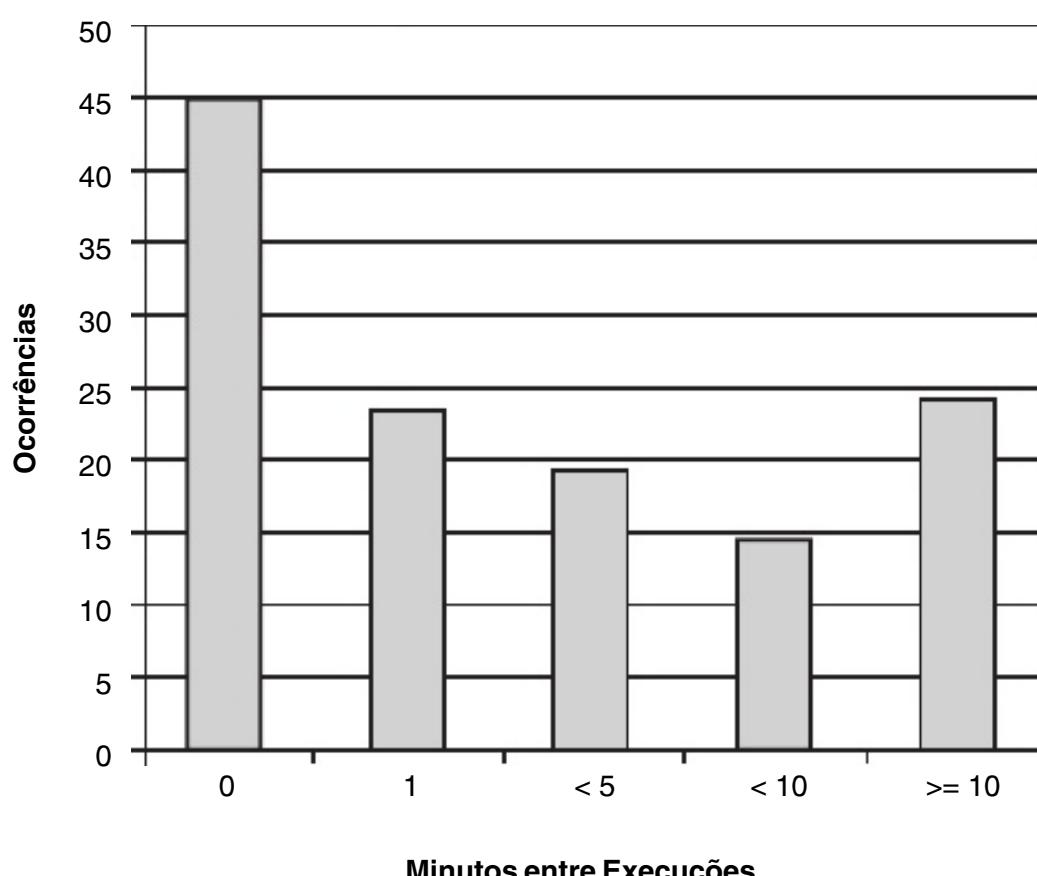
jamais havia visto. Estou preocupado com a performance de expressões, mas estou feliz em esperar até ver algumas estatísticas de uso antes de começar a otimizar.

E se eu tiver que escrever 20 vezes tudo o que já escrevi? Continuaria a encontrar ideias e surpresas a cada vez? Há algum jeito de ser mais atento conforme programo, de forma a poder ter todas as ideias nas primeiras três vezes? Na primeira vez?

## Uso de JUnit

Eu tinha JUnit para manter um log enquanto eu estava codificando o exemplo financeiro. Eu apertei o botão Run exatamente 125 vezes. Devido a eu estar escrevendo ao mesmo tempo em que estava programando, o intervalo entre as execuções não foi representativo, mas durante as vezes em que estava apenas programando, rodei os testes cerca de uma vez por minuto. Somente uma vez, nesse tempo todo, fui surpreendido pelo sucesso ou falha, e isso foi em uma refatoração feita às pressas.

A Figura 17.1 é um histograma do intervalo de tempo entre os testes executados. O grande número de intervalos longos provavelmente é por causa do tempo que eu gastava escrevendo.



**Figura 17.1** Histograma do intervalo de tempo entre execução de testes.

## Métricas de código

A Tabela 17.1 dá algumas estatísticas sobre o código.

**Tabela 17.1** *Métricas de código*

	Funcional	Teste
Classes	5	1
Funções (1)	22	15
Linhas (2)	91	89
Complexidade ciclomática (3)	1,04	1
Linhas/função	4,1 (4)	5,9 (5)

- (1) Devido a não termos implementado a API completa, não podemos avaliar o número absoluto de funções, ou o número de funções por classe, ou o número de linhas por classe. Contudo, os valores são instrutivos. Há aproximadamente a mesma quantidade de linhas e funções no teste e no código funcional.
- (2) O número de linhas de código teste pode ser reduzido extraíndo padrões comuns. Todavia, a correspondência aproximada entre linhas do código modelo e linhas do código teste permanecerá.
- (3) Complexidade ciclomática é uma medida convencional de complexidade de fluxo. Complexidade de teste é 1, pois não há decisões ou laços no código teste. A complexidade do código funcional é baixa por causa do uso pesado de polimorfismo como um substituto para controle de fluxo explícito.
- (4) Isso inclui o cabeçalho da função e as chaves.
- (5) Linhas por função nos testes são infladas porque não fatoramos código comum, como explicado na seção anterior.

## Processo

O ciclo do TDD é como segue:

- Adicione um pequeno teste
- Rode todos os testes e falhe
- Faça uma mudança
- Rode todos os testes e seja bem-sucedido
- Refatore para remover duplicação

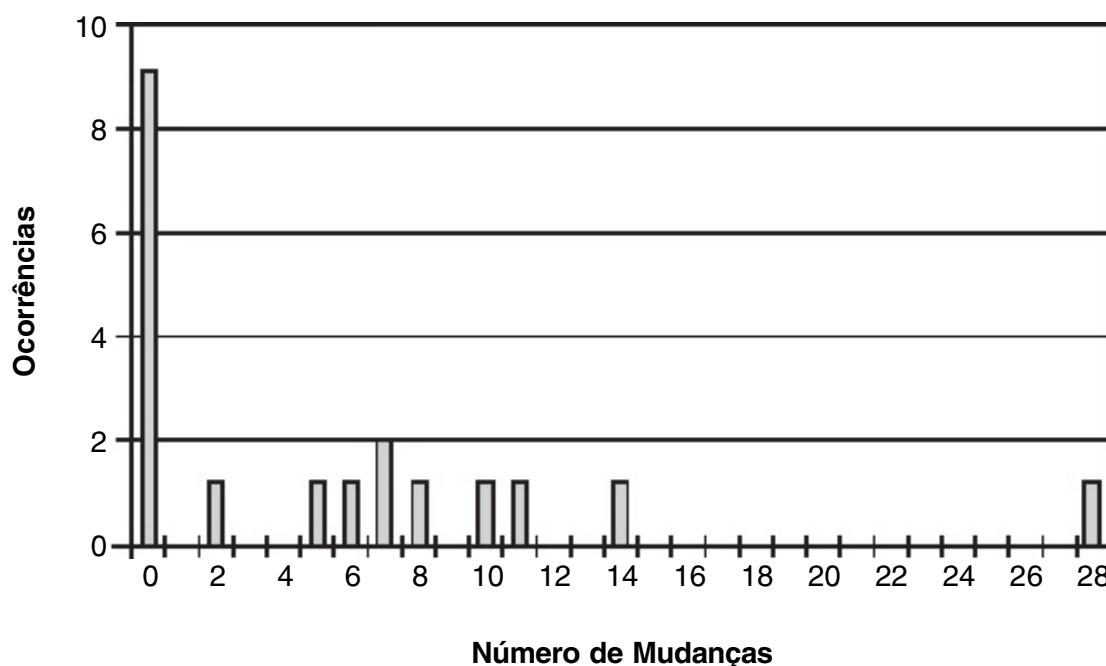
Partindo do princípio de que escrever um teste é um único passo, quantas mudanças são necessárias para compilar, executar e refatorar? (Por mudança, quero dizer mudar uma definição de método ou classe.) A Figura 17.2 mostra um histograma do número de mudanças para cada um dos testes financeiros que você viu.

Eu espero que, se reunimos dados para um grande projeto, o número de mudanças para compilar e executar permaneça relativamente pequeno (eles poderiam ser inclusive menores se o ambiente de programação entender o que os testes estão tentando dizer – criando stubs automaticamente, por exemplo). Contudo, (e aqui está ao menos uma dissertação de mestrado) o número de mudanças por refatoração deveria seguir uma “cauda pesada” ou perfil leptocúrtico, o qual é como a curva de um sino, mas com mais mudanças radicais que o previsto por uma curva de sino padrão. Muitas medidas na natureza seguem esse perfil, tal como variações de preço no mercado de ações.<sup>1</sup>

## Qualidade de teste

Os testes que são um subproduto natural de TDD são certamente bastante úteis para serem mantidos enquanto o sistema estiver rodando. Não espere que eles substituam os outros tipos de teste:

- Performance
- Estresse
- Usabilidade



**Figura 17.2** Número de mudanças por refatoração.

<sup>1</sup> Mandelbrot, Benoit, ed. 1997. *Fractals and Scaling in Finance*. New York: Springer-Verlag. ISBN: 0387983635.

Contudo, se a densidade de defeitos do código guiado por testes é suficientemente baixa, então o papel do teste profissional inevitavelmente mudará de “supervisão de um adulto” para algo mais parecido com um amplificador de comunicação entre aqueles que geralmente têm um *feeling* para o que o sistema deveria fazer e aqueles que o farão. Como um entendimento para uma conversa longa e interessante sobre o futuro do teste profissional, aqui estão algumas medidas amplamente compartilhadas pelos testes descritos.

- Cobertura de comando certamente não é uma medida suficiente de qualidade de teste, mas é um ponto de partida. TDD seguido religiosamente deveria resultar em 100% de cobertura de comandos. JProbe ([www.sitraka.com/software/jprobe](http://www.sitraka.com/software/jprobe)) informa apenas uma linha em um método não coberta pelos casos de teste – `Money.toString()`, o qual adicionamos explicitamente como uma ajuda de depuração e não como um código modelo real.
- Inserção de defeitos é outra forma de avaliar qualidade de teste. A ideia é simples: mude o significado de uma linha de código e um teste deverá parar de funcionar. Você pode fazer isso manualmente, ou usando uma ferramenta como Jester ([jester.sourceforge.net](http://jester.sourceforge.net)). Jester informa apenas uma linha capaz de mudar sem parar de funcionar, `Pair.hashCode()`. Nós podemos fazer de conta a implementação para apenas retornar 0 (zero). Retornar uma constante diferente não muda realmente o significado do programa (um número faz de conta é tão bom como qualquer outro), então isso não é realmente um defeito que foi inserido.

Phlip, um de meus revisores, fez uma observação sobre a cobertura de teste que vale a pena ser repetida aqui. Uma medida grosseira de cobertura é o número de testes que testam diferentes aspectos de um programa dividido pelo número de aspectos que precisam de teste (a complexidade da lógica). Uma forma de melhorar a cobertura é escrever mais testes, e daí vem a diferença dramática entre o número de testes que um desenvolvedor dirigido por testes escreveria para o código e o número de testes que um testador profissional escreveria. (O Capítulo 32 dá detalhes sobre um exemplo em que escrevi 6 testes e um testador escreveu 65 testes para o mesmo problema.) Contudo, outra forma de melhorar a cobertura é pegar um conjunto fixo de testes e simplificar a lógica do programa. O passo de refatoração frequentemente tem esse efeito – condicionais substituídos por mensagens, ou por nada. Nas palavras de Phlip, “em vez de aumentar a cobertura de testes para caminhar por todas as permutações das entradas (mais propriamente, uma amostra de todas as permutações possíveis), apenas deixamos os mesmos testes cobrirem várias permutações do código conforme ele diminui”.

---

## Uma última revisão

Os três itens que surgem de novo como surpresas ao ensinar TDD são:

- As três abordagens para fazer um teste funcionar de forma limpa – fazer de conta, triangulação e implementação óbvia.
- Remover duplicação entre teste e código como uma forma de guiar o projeto.
- A habilidade de controlar a lacuna entre testes para aumentar a tração quando a estrada fica escorregadia e para viajar mais rápido quando as condições estão boas.

## PARTE II

---

# O Exemplo xUnit

Como, ó como, falar sobre a implementação de uma ferramenta para desenvolvimento guiado por testes? Guiado por testes, naturalmente.

A arquitetura xUnit apareceu suavemente em Python, por isso vou trocar para Python na Parte II. Não se preocupe, darei uma comentadinha sobre Python para aqueles de vocês que nunca a viram antes. Quando estiver pronto, você terá uma introdução à Python, será capaz de escrever seu próprio framework de testes e terá visto um exemplo mais complicado de TDD – três pelo preço de um.

# CAPÍTULO 18

---

## Primeiros Passos para xUnit

Dirigir uma ferramenta de testes usando a própria ferramenta de testes para rodar os testes pode parecer um pouco com realizar uma cirurgia cerebral em si próprio. (“Não mexa nesses centros motores – opa, muito ruim, fim de jogo.”) Ficará estranho de vez em quando. Entretanto, a lógica do framework de testes é mais complicada que o exemplo financeiro fraquinho da Parte I. Você pode ler a Parte II como um passo em direção ao desenvolvimento guiado por testes de softwares “reais”. Você pode lê-lo como um exercício de ciência da computação em programação com autorreferência.

Primeiro, precisamos ser capazes de criar um caso de teste e rodar um método de teste. Por exemplo: `TestCase("testMethod").run()`. Temos um problema de inicialização (bootstrap): estamos escrevendo casos de teste para testar um framework que será usado para escrever casos de teste. Ainda não temos um framework, então teremos que verificar a operação do primeiro pequeno passo à mão. Felizmente, estamos bem descansados e relaxados e longe de cometer erros, e é por isso que iremos a passos pequeninos, verificando tudo de diversas maneiras. Aqui está a lista de tarefas que vêm à mente para um framework de testes.

<b>Invoque o método teste</b>
Invoque <code>setUp</code> primeiro
Invoque <code>tearDown</code> depois
Invoque <code>tearDown</code> mesmo se o método teste falhar
Rode múltiplos testes
Informe resultados coletados

Ainda estamos trabalhando no primeiro teste, certamente. Para nosso primeiro prototeste, precisamos de um pequeno programa que mostrará verdadeiro se um método de teste for chamado, e falso caso contrário. Se tivermos um caso de

teste que inicializa uma flag dentro do método de teste, então podemos mostrar a flag depois que tivermos feito e tivermos certeza de que está correto. Uma vez que tenhamos verificado isso manualmente, podemos automatizar o processo.

Aqui está a estratégia para nosso teste de inicialização (bootstrap): criaremos um caso de teste que contenha uma flag. Antes de o método de teste ser executado, a flag deveria ser falsa. O método de teste inicializará a flag. Depois que esse método de teste rodar, a flag deverá ser verdadeira. Nós chamaremos a classe `TestCase` de `WasRun`, pois é um caso de teste que informa se um método rodou. A flag também será chamada `wasRun` (talvez confuso, mas é um bom nome), então podemos finalmente escrever `assert test.wasRun` (`assert` é um recurso já existente em Python).

Python executa comandos como lê um arquivo, então podemos começar invocando o método de teste manualmente:

```
test= WasRun("testMethod")
print test.wasRun
test.testMethod()
print test.wasRun
```

Esperamos que isso exiba (print) “None” antes do método ser executado, e “1” depois disso. (None em Python é como null ou nil, e significa falso, junto com 0 e alguns outros objetos.) Mas ele não faz o que esperávamos, pois não definimos a classe `WasRun` ainda (teste primeiro, teste primeiro).

### WasRun

```
class WasRun:
    pass
```

(A palavra-chave `pass` é usada quando não há implementação de uma classe ou método.) Agora somos informados de que precisamos de um atributo `wasRun`. Precisamos criar o atributo quando criamos a instância (o construtor é chamado `__init__` por conveniência). Nele, inicializamos a flag `wasRun` como falso.

### WasRun

```
class WasRun:
    def __init__(self, name):
        self.wasRun= None
```

Executando, o arquivo exibe fielmente “None”, então nos diz que precisamos definir o método `testMethod`. (Não seria maravilhoso se sua IDE avisasse isso, fornecesse um stub e abrisse um editor para ele? Nã, útil demais.)

### WasRun

```
def testMethod(self):
    pass
```

Agora quando executamos o arquivo, vemos “None” e “None”. Queremos ver “None” e “1”. Podemos ter isso inicializando a flag em `testMethod()`:

**WasRun**

```
def testMethod(self):
    self.wasRun= 1
```

Agora temos a resposta certa – a barra verde, iúpi! Temos agora um monte de refatoração para fazer, mas enquanto mantivermos a barra verde, sabemos que fizemos progresso.

Em seguida, precisamos usar nossa interface real, `run()`, em vez de chamar o método de teste diretamente. O teste muda para o seguinte:

```
test= WasRun("testMethod")
print test.wasRun
test.run()
print test.wasRun
```

A implementação pode ser construída no momento como:

**WasRun**

```
def run(self):
    self.testMethod()
```

E nosso teste volta a exibir os valores certos de novo. Montes de refatorações têm essa ideia – separar em duas partes para que você possa trabalhar nelas separadamente. Se elas voltarem a se juntar quando você terminar, legal; se não, você pode deixá-las separadas. Nesse caso, esperamos criar uma superclasse `TestCase`, em algum momento, mas, primeiro, temos que diferenciar as partes para nosso único exemplo. Há provavelmente alguma analogia inteligente com mitose aqui, mas eu não conheço o suficiente de biologia celular para explicá-la.

O próximo passo é invocar dinamicamente o `testMethod`. Uma das características mais legais de Python é que itens como os nomes de classes e métodos podem ser tratados como funções (veja a invocação de `WasRun` acima). Quando temos o atributo correspondente ao nome do caso de teste, retornamos um objeto que, quando invocado como uma função, invoca o método.<sup>1</sup>

**WasRun**

```
class WasRun:
    def __init__(self, name):
        self.wasRun= None
        self.name= name
    def run(self):
        method = getattr(self, self.name)
        method()
```

Aqui está outro padrão geral de refatoração: pegar código que funciona em uma instância e o generalizar para funcionar em várias, substituindo constantes por variáveis. Aqui a constante foi forçada no código, não é um valor de dados, mas o princípio é o mesmo. TDD faz bem o trabalho, dando a você exemplos concretos para generalizar, em vez de ter que generalizar puramente pelo raciocínio.

<sup>1</sup> Obrigado a Duncan Booth por corrigir meus erros em Python e por sugerir a solução em Python.

Agora nossa pequena classe `WasRun` está fazendo dois trabalhos distintos: um é manter o rastro se um método foi invocado ou não, e outro é invocar dinamicamente o método. Hora de um pouco de ação de mitose. Primeiro, criamos uma superclasse `TestCase` vazia e fazemos de `WasRun` uma subclasse:

**TestCase**

```
class TestCase:  
    pass
```

**WasRun**

```
class WasRun(TestCase): . . .
```

Agora podemos mover o atributo `name` para a superclasse:

**TestCase**

```
def __init__(self, name):  
    self.name= name
```

**WasRun**

```
def __init__(self, name):  
    self.wasRun= None  
    TestCase.__init__(self, name)
```

Finalmente, o método `run()` usa apenas atributos da superclasse, então ele provavelmente pertence à superclasse. (Estou sempre procurando pôr as operações próximas aos dados.)

**TestCase**

```
def __init__(self, name):  
    self.name= name  
def run(self):  
    method = getattr(self, self.name)  
    method()
```

Entre cada uma dessas etapas, eu rodei os testes para ter certeza que estou tendo a mesma resposta.

Estamos ficando cansados de ver aquele “None” e “1” mostrados a cada vez. Usando o mecanismo que recém construímos, podemos escrever agora:

**TestCaseTest**

```
class TestCaseTest(TestCase):  
    def testRunning(self):  
        test= WasRun("testMethod")  
        assert(not test.wasRun)  
        test.run()  
        assert(test.wasRun)  
    TestCaseTest("testRunning").run()
```

**Invoca o método teste**

Invoca setUp primeiro

Invoca tearDown depois

Invoca tearDown mesmo se o método teste falhar

Rode múltiplos testes

Informe resultados coletados

O corpo do teste está apenas exibindo declarações transformadas em atribuições, então você poderia ver o que fizemos como uma forma complicada de Extract Method\*.

Vou contar-lhe um segredinho. Eu olho para o tamanho dos passos no desenvolvimento que recém mostrei para você, e parece ridículo. Por outro lado, tentei isso com passos maiores, gastando provavelmente seis horas em tudo (gastei um tempão à procura de coisas de Python) e começando do zero duas vezes, e, em ambas as vezes, pensei ter código funcionando quando não tinha. Esse é o pior caso possível para TDD, pois estamos tentando superar a etapa de inicialização.

Não é necessário trabalhar em passos tão pequenos como esses. Uma vez que você tenha dominado TDD, será capaz de trabalhar funcionalidades em saltos muito maiores entre os casos de teste. Entretanto, para dominar TDD, você precisa ser capaz de trabalhar em passos muito pequenos quando eles são necessários.

A seguir, vamos chamar `setUp()` antes de rodar o teste. Para revisar, primeiro:

- Depois de um monte de falsos começos orgulhosos, descobrimos como começar com um passo pequeno.
- Implementamos funcionalidade, primeiro forçando sua implementação, e então tornando-a mais geral, substituindo constantes por variáveis.
- Usamos Pluggable Selector\*\*, o qual prometemos não usar de novo por quatro meses no mínimo, pois faz o código difícil de se analisar estatisticamente.
- Inicializamos nosso framework de testes; tudo em passos pequenos.

---

\* N. de R. T.: Extract Method é uma refatoração que extrai um trecho de código de um método e o torna um método cujo nome explica adequadamente seu propósito. Mais detalhes em <http://www.refactoring.com/catalog/extractMethod.html> ou no livro FOWLER, Martin. *Refatoração: aperfeiçoando o projeto de código existente*. Porto Alegre: Bookman, 2004.

\*\* N. de R. T.: Pluggable Selector é o padrão de implementação usado quando o nome do método a ser invocado é parametrizável (neste caso em um atributo) e assim uma classe pode executar lógica diferente sem subclasses.

# CAPÍTULO 19

---

## Iniciando

Quando você começa a escrever testes, descobrirá um padrão comum (Bill Wake cunhou o termo 3A para isso).

1. Arranje – Crie alguns objetos.
2. Aja – Estimule-os.
3. (Faça) Asserções – Verifique os resultados.

~~Invoca o método teste~~

~~Invoca setUp primeiro~~

~~Invoca tearDown depois~~

~~Invoca tearDown mesmo se o método teste falhar~~

~~Rode múltiplos testes~~

~~Informe resultados coletados~~

O primeiro passo, arranje (organize), é frequentemente o mesmo de teste em teste, enquanto a segunda e terceira etapas, agir e (fazer) asserções, são únicas. Eu tenho um 7 e um 9. Se somá-los, eu espero 16; se subtraí-los, eu espero -2; e, se multiplicá-los, eu espero 63. O estímulo e os resultados esperados são únicos, mas o 7 e o 9 não mudam.

Se esse padrão se repete em diferentes escalas (e ele se repete, sim), então somos confrontados com a questão de quão frequentemente queremos criar novos objetos para testar. Duas restrições entram em conflito.

- Performance – Gostaríamos que nossos testes rodassem o mais rápido possível, isto é, se usamos objetos similares em muitos testes, gostaríamos de criá-los uma vez para todos os testes.

- Isolamento – Gostaríamos que o sucesso ou a falha de um teste fosse irrelevante para outros testes. Se testes dividem objetos e um teste muda os objetos, os testes seguintes provavelmente mudarão seus resultados.

O acoplamento de testes possui um efeito desagradável e óbvio pelo qual fazer um teste falhar faz os próximos dez falharem, mesmo que o código esteja correto. O acoplamento de testes pode ter um efeito útil, mas muito desagradável, em situações nas quais a ordem dos testes importa: se eu executo um teste A antes do teste B, os dois funcionam, mas se eu executo o teste B antes do teste A, então o teste A falha. Ou ainda pior, o código exercitado por B está errado, mas devido ao teste A rodar primeiro, os testes passam.

Acoplamento de testes – não vá por aí. Vamos assumir por um momento que podemos criar objetos de forma rápida o suficiente. Nesse caso, gostaríamos de criar os objetos para um teste cada vez que ele executa. Já vimos uma forma disfarçada disso em `WasRun`, onde queríamos ter uma flag mudada para falso antes de rodarmos o teste. Seguindo os passos nessa direção, primeiro, precisamos de um teste:

#### TestCaseTest

```
def testSetUp(self):  
    test= WasRun("testMethod")  
    test.run()  
    assert(test.wassetUp)
```

Ao executar isso (adicionando a última linha `TestCaseTest("testSetUp").run()` ao nosso arquivo), Python nos informa educadamente que não há um atributo `wassetUp`. É certo que não. Nós não o inicializamos. Esse método deveria fazer:

#### WasRun

```
def setUp(self):  
    self.wassetUp= 1
```

Ele existiria se o estivéssemos chamando. Chamar `setUp` é o trabalho de `TestCase`, assim, voltamos lá:

#### TestCase

```
def setUp(self):  
    pass  
def run(self):  
    self.setUp()  
    method = getattr(self, self.name)  
    method()
```

Ou seja, dois passos para ter um caso de teste rodando, o que é demais em tais circunstâncias delicadas. Veremos se funcionará. Sim, ele passa. Contudo, se

quiser aprender algo, tente descobrir como poderíamos ter obtido o teste funcionando mudando não mais do que um método de cada vez.

Podemos usar imediatamente nossa nova fábrica para encurtar nossos testes. Primeiro, podemos simplificar `WasRun` iniciando a flag `wasRun` em `setUp`:

### WasRun

```
def setUp(self):
    self.wasRun= None
    self.wasSetUp= 1
```

Temos que simplificar `testRunning` para não buscar a flag antes de rodar o teste. Estamos dispostos a desistir desse tanto de confiança do nosso código? Apenas se `testSetUp` estiver no lugar. Esse é um padrão comum – um teste pode ser simplificado se, e somente se, outro teste está no lugar e executando corretamente:

### TestCaseTest

```
def testRunning(self):
    test= WasRun("testMethod")
    test.run()
    assert(test.wasRun)
```

Podemos também simplificar os próprios testes. Em ambos os casos, criamos uma instância de `WasRun`, exatamente a fixture que estávamos falando anteriormente. Podemos criar `WasRun` em `setUp` e usá-lo nos métodos de teste. Cada método de teste é executado em uma instância limpa de `TestCaseTest`, assim não há como os dois testes estarem acoplados. (Estamos assumindo que os objetos não interagem de uma forma incrivelmente feia como inicializar variáveis globais, mas não faríamos isso, não com todos esses outros leitores olhando.)

### TestCaseTest

```
def setUp(self):
    self.test= WasRun("testMethod")
def testRunning(self):
    self.test.run()
    assert(self.test.wasRun)
def testSetUp(self):
    self.test.run()
    assert(self.test.wasSetUp)
```

Invoque o método teste

~~Invoque setUp primeiro~~

Invoque tearDown depois

Invoque tearDown mesmo se o método teste falhar

Rode múltiplos testes

Informe resultados coletados

Em seguida, rodaremos `tearDown()` depois do método de teste. Para revisar esse capítulo:

- Decidimos que a simplicidade da escrita do teste era mais importante que a performance para o momento.
- Testamos e implementamos `setup()`.
- Usamos `setup()` para simplificar o caso de teste exemplo.
- Usamos `setUp()` para simplificar casos de teste verificando o caso de teste exemplo (eu falei para você que isso seria como fazer uma cirurgia no próprio cérebro).

# CAPÍTULO 20

---

## Limpando em Seguida

**Invoque o método teste**  
**Invoque setUp primeiro**  
**Invoque tearDown depois**  
Invoque tearDown mesmo se o método teste falhar  
Rode múltiplos testes  
Informe resultados coletados

Às vezes, testes precisam alocar recursos externos em `setUp()`. Se quisermos que os testes mantenham-se independentes, então um teste que aloca recursos externos precisa liberá-los antes de estar acabado, talvez em um método `tearDown()`.

O jeito simplista de escrever o teste para desalocação é introduzir outra flag. Todas essas flags estão começando a me incomodar e estão deixando escapar um aspecto importante dos métodos: `setUp()` é chamado antes que o método de teste seja executado, e `tearDown()` é chamado depois disso. Vou mudar a estratégia de testes para manter um pequeno registro (log) de métodos que são chamados. Ao acrescentar estas chamadas ao registro, preservaremos a ordem em que os métodos são chamados.

**Invoque o método teste**  
**~~Invoque setUp primeiro~~**  
Invoque tearDown depois  
Invoque tearDown mesmo se o método teste falhar  
Rode múltiplos testes  
Informe resultados coletados  
**String de registro em WasRun**

**WasRun**

```
def setUp(self):
    self.wasRun= None
    self.wasSetUp= 1
    self.log= "setUp "
```

Agora podemos mudar `testSetUp()` para olhar para o registro (`log`) e não para a flag:

**TestCaseTest**

```
def testSetUp(self):
    self.test.run()
    assert("setUp " == self.test.log)
```

Em seguida, podemos deletar a flag `wasSetUp`. Podemos também gravar a execução do método de teste:

**WasRun**

```
def testMethod(self):
    self.wasRun= 1
    self.log= self.log + "testMethod "
```

Isso estraga `testSetUp`, pois o registro atual contém “`setUp testMethod`”. Mudamos o valor esperado:

**TestCaseTest**

```
def testSetUp(self):
    self.test.run()
    assert("setUp testMethod " == self.test.log)
```

Agora esse teste está fazendo o trabalho de ambos os testes, então podemos deletar `testRunning` e renomear `testSetUp`:

**TestCaseTest**

```
def setUp(self):
    self.test= WasRun("testMethod")
def testTemplateMethod(self):
    self.test.run()
    assert("setUp testMethod " == self.test.log)
```

Infelizmente, estamos usando a instância `WasRun` em apenas um lugar, então temos que desfazer nosso inteligente hack `setUp`:

**TestCaseTest**

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run()
    assert("setUp testMethod " == test.log)
```

Fazer uma refatoração baseada em um monte de usos anteriores e então ter que desfazê-la logo depois é bastante comum. Algumas pessoas esperam até terem três

ou quatro usos antes de refatorar, pois não gostam de desfazer trabalho. Eu prefiro gastar meus ciclos de pensamento no projeto, assim apenas faço as refatorações reflexivamente, sem me preocupar se terei que desfazê-las imediatamente depois.

---

Invoque o método teste  
 Invoque setUp primeiro  
**Invoque tearDown depois**  
 Invoque tearDown mesmo se o método teste falhar  
 Rode múltiplos testes  
 Informe resultados coletados  
~~String de registro em WasRun~~

---

Agora estamos preparados para implementar tearDown(). Peguei você! Agora estamos preparados para testar tearDown:

#### TestCaseTest

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run()
    assert("setUp testMethod tearDown " == test.log)
```

Ele falha. Fazê-lo funcionar é simples:

#### TestCase

```
def run(self, result):
    result.testStarted()
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
```

#### WasRun

```
def setUp(self):
    self.log= "setUp "
def testMethod(self):
    self.log= self.log + "testMethod "
def tearDown(self):
    self.log= self.log + "tearDown "
```

Surpreendentemente, temos um erro; não em WasRun, mas em TestCaseTest. Não temos uma implementação nop (“não-faça-nada”) de tearDown() em TestCase:

#### TestCase

```
def tearDown(self):
    pass
```

Dessa vez temos que valorizar o uso do mesmo framework de testes que estamos desenvolvendo. Iúpi! Nenhuma refatoração é necessária. A Implementação Óbvia, depois daquela única falha, funcionou e estava limpa.

~~Invoque o método teste~~

~~Invoque setUp primeiro~~

Invoque tearDown depois

Invoque tearDown mesmo se o método teste falhar

Rode múltiplos testes

Informe resultados coletados

~~String de registro em WasRun~~

Em seguida, continuaremos a informar os resultados de rodar um teste explicitamente em vez de deixar o sistema de tratamento de erros nativo do Python tratar e nos informar quando há um problema com uma asserção. Para revisar, neste capítulo:

- Reestruturamos a estratégia de testes de flags para um registro (log).
- Testamos e implementamos tearDown() usando o novo registro.
- Encontramos um problema e, ousadamente, o corrigimos em vez de voltarmos atrás (Foi uma boa ideia?).

# CAPÍTULO 21

---

## Contagem

Invoque o método teste  
Invoque setUp primeiro  
Invoque tearDown depois  
Invoque tearDown mesmo se o método teste falhar  
Rode múltiplos testes  
**Informe resultados coletados**  
~~String de registro em WasRun~~

Eu ia incluir uma implementação para assegurar que tearDown() fosse chamada a despeito das exceções durante o método de teste. Contudo, precisamos capturar exceções de forma a fazer o teste funcionar. (Eu sei, eu tentei e recuei.) Se cometermos um engano implementando isso, não seremos capazes de ver o engano, pois as exceções não serão informadas.

Em geral, a ordem da implementação dos testes é importante. Quando eu pego o próximo teste para implementar, eu acho que um teste me ensinará algo e eu tenho confiança que posso fazer funcionar. Se eu deixar aquele teste funcionando, mas ficar emperrado no próximo, então considero o backup de dois passos. Seria ótimo se o ambiente de programação me ajudasse com isso, funcionando como um checkpoint para o código cada vez que todos os testes rodarem.

Gostaríamos de ver os resultados da execução de qualquer número de testes – “5 rodaram, 2 falharam, TestCaseTest.testFooBar – ExceçãoDeDivisãoPorZero, MoneyTest.testNegation – ErroDeAserção”. Então, se os testes pararem de ser chamados ou se os resultados pararem de ser informados, ao menos temos uma chance de capturar o erro. Ter o framework informando automaticamente todos os casos de teste dos quais ele não sabe nada parece um pouco forçado, ao menos para o primeiro caso de teste.

Teremos `TestCase.run()` retornando um objeto `TestResult` que grava os resultados da execução do teste (singular, por enquanto, mas vamos chegar lá).

### TestCaseTest

```
def testResult(self):
    test= WasRun("testMethod")
    result= test.run()
    assert("1 run, 0 failed" == result.summary())
```

Começaremos com uma implementação faz de conta:

### TestResult

```
class TestResult:
    def summary(self):
        return "1 run, 0 failed"
```

e retornando `TestResult` como o resultado de `TestCase.run()`

### TestCase

```
def run(self):
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
    return TestResult()
```

Agora que o teste roda, podemos tornar concreta a implementação de `summary()` um pouco de cada vez. Primeiro, podemos fazer do número de testes que executam uma constante simbólica:

### TestResult

```
def __init__(self):
    self.runCount= 1
def summary(self):
    return "%d run, 0 failed" % self.runCount
```

(O operador `%` é o `sprintf` do Python.) Entretanto, `runCount` não devia ser uma constante; devia ser computada pela contagem do número de testes que executam. Podemos inicializá-la com 0 e então incrementá-la cada vez que um teste é executado.

### TestResult

```
def __init__(self):
    self.runCount= 0
def testStarted(self):
    self.runCount= self.runCount + 1
def summary(self):
    return "%d run, 0 failed" % self.runCount
```

Temos realmente que chamar esse novo método bacana:

**TestCase**

```
def run(self):
    result= TestResult()
    result.testStarted()
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
    return result
```

Poderíamos transformar a string constante “0” do número de testes que falharam em uma variável da mesma forma que concretizamos `runCount`, mas os testes não exigem isso. Então, em vez disso, escrevemos outro teste:

**TestCaseTest**

```
def testFailedResult(self):
    test= WasRun("testBrokenMethod")
    result= test.run()
    assert("1 run, 1 failed", result.summary)
```

onde:

**WasRun**

```
def testBrokenMethod(self):
    raise Exception
```

Invoque o método teste
Invoque setUp primeiro
Invoque tearDown depois
Invoque tearDown mesmo se o método teste falhar
Rode múltiplos testes
Informe resultados coletados
String de registro em WasRun
Informar testes que falharam

A primeira coisa que notamos é que não estamos capturando a exceção lançada por `WasRun.testBrokenMethod`. Gostaríamos de capturar a exceção e fazer uma anotação no resultado que o teste falhou. Colocaremos esse teste na prateleira por enquanto.

Para revisar:

- Escrevemos uma implementação faz de conta e, gradualmente, começamos a torná-la concreta, substituindo constantes por variáveis.
- Escrevemos outro teste.
- Quando o teste falhou, escrevemos outro teste, em menor escala, para dar suporte e fazer o teste que falhou funcionar.

# CAPÍTULO 22

---

## Lidando com Falha

Invoque o método teste  
Invoque setUp primeiro  
Invoque tearDown depois  
Invoque tearDown mesmo se o método teste falhar  
Rode múltiplos testes  
Informe resultados coletados  
~~String de registro em WasRun~~  
Informar testes que falharam

Escreveremos um teste mais granular para garantir que, se notarmos que um teste falhou, mostraremos os resultados certos:

### TestCaseTest

```
def testFailedResultFormatting(self):
    result= TestResult()
    result.testStarted()
    result.testFailed()
    assert("1 run, 1 failed" == result.summary())
```

“testStarted()” e “testFailed()” são as mensagens que esperamos enviar para o resultado quando um teste começa e quanto um teste falha, respectivamente. Se pudermos fazer o sumário mostrar corretamente quando essas mensagens forem enviadas nessa ordem, então nosso problema de programação está reduzido a como enviar essas mensagens. Uma vez que elas sejam enviadas, esperamos que a coisa toda funcione.

A implementação é para manter uma contagem de falhas:

### TestResult

```
def __init__(self):
```

```
self.runCount= 0
self.errorCount= 0
def testFailed(self):
    self.errorCount= self.errorCount + 1
```

Com a contagem correta (a qual suponho que poderíamos ter testado, se estivéssemos dando passos muito, muito pequeninos – mas eu não vou me incomodar, pois o café fez efeito agora), podemos mostrar corretamente:

### TestResult

```
def summary(self):
    return "%d run, %d failed" % (self.runCount, self.failureCount)
```

Agora esperamos que se chamarmos `testFailed()` corretamente, teremos a resposta esperada. Quando a chamamos? Quando capturarmos uma exceção no método de teste:

### TestCase

```
def run(self):
    result= TestResult()
    result.testStarted()
    self.setUp()
    try:
        method = getattr(self, self.name)
        method()
    except:
        result.testFailed()
    self.tearDown()
    return result
```

Há uma sutileza escondida nesse método. Pela forma como foi escrito, se um desastre acontecer durante `setUp()`, então a exceção não será pega. Isso pode não ser o que queremos – queremos que nossos testes executem independentemente uns dos outros. Contudo, precisamos de outro teste antes que possamos mudar o código. (Eu ensinei para Bethany, minha filha mais velha, TDD como seu primeiro estilo de programação quando ela tinha uns 12 anos. Ela acha que você não pode digitar código a menos que exista um teste que não funciona. O resto de nós tem que se atrapalhar todo lembrando de escrever os testes.) Deixarei aquele próximo teste e sua implementação como um exercício para você (dedos doloridos, de novo).

~~Invoque o método teste~~  
~~Invoque setUp primeiro~~  
~~Invoque tearDown depois~~  
~~Invoque tearDown mesmo se o método teste falhar~~  
~~Rode múltiplos testes~~  
~~Informe resultados coletados~~  
~~String de registro em WasRun~~  
~~Informar testes que falharam~~  
~~Capturar e informar erros em setUp~~

A seguir, trabalharemos para obter vários testes rodando juntos. Para revisar este capítulo:

- Fizemos nosso trabalho de teste em pequena escala.
- Reintroduzimos o teste em larga escala.
- Fizemos o trabalho do teste maior rapidamente, usando o mecanismo demonstrado pelo teste menor.
- Percebemos um problema potencial e o anotamos na lista de tarefas em vez de atacá-lo imediatamente.

# CAPÍTULO 23

---

## Como é a Suíte?

~~Invoque o método teste~~  
~~Invoque setUp primeiro~~  
~~Invoque tearDown depois~~  
Invoque tearDown mesmo se o método teste falhar  
Rode múltiplos testes  
Informe resultados coletados  
~~String de registro em WasRun~~  
~~Informar testes que falharam~~  
Capturar e informar erros em setUp

Não podemos deixar xUnit sem visitar TestSuite (suíte\* de testes). O final de nosso arquivo, onde invocamos todos os testes, está parecendo um ninho de ratos:

```
print TestCaseTest("testTemplateMethod").run().summary()
print TestCaseTest("testResult").run().summary()
print TestCaseTest("testFailedResultFormatting").run().summary()
print TestCaseTest("testFailedResult").run().summary()
```

Duplicação é sempre uma coisa ruim, a menos que você olhe para ela como uma motivação para encontrar o elemento de projeto que falta. Aqui gostaríamos da habilidade para compor testes e os rodar juntos. (Trabalhar duro para fazer os testes executarem isoladamente não nos faz tão bem se apenas rodamos um teste por vez.) Outra boa razão para implementar TestSuite é que isso nos dá um exemplo

---

\* N. de R. T.: O termo “suíte” é usado como sinônimo de “conjunto” pela comunidade de teste de software.

puro de Composite\* – queremos ser capazes de tratar um único teste e um grupo de testes exatamente da mesma forma.

Gostaríamos de ser capazes de criar um TestSuite, adicionar alguns testes a ele e, então, obter resultados coletivos ao executá-lo:

### TestCaseTest

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    result= suite.run()
    assert("2 run, 1 failed" == result.summary())
```

Implementar o método add() meramente adiciona testes a uma lista:

### TestSuite

```
class TestSuite:
    def __init__(self):
        self.tests= []
    def add(self, test):
        self.tests.append(test)
```

(nota de Python: [ ] cria uma coleção vazia.)

O método run é um pequeno problema. Queremos que apenas um TestResult seja usado por todos os testes que executam. Portanto, devemos escrever:

### TestSuite

```
def run(self):
    result= TestResult()
    for test in tests:
        test.run(result)
    return result
```

(nota de Python: for test in self.tests itera através dos elementos de tests, associando cada um a tests e avaliando o código a seguir.) Contudo, uma das principais restrições em Composite é que a coleção (objeto composto) deve responder às mesmas mensagens que os objetos primitivos. Se adicionarmos um parâmetro a TestCase.run(), então temos que adicionar o mesmo parâmetro a TestSuite.run(). Eu posso pensar em três alternativas:

- Usar o mecanismo de parâmetros padrão de Python. Infelizmente, o valor padrão é avaliado em tempo de compilação, não em tempo de execução, e não queremos reusar o mesmo TestResult.
- Dividir o método em duas partes – uma que aloca o TestResult e outra que executa o teste dado um TestResult. Eu não consigo pensar em bons nomes

---

\* N. de R. T.: Composite é um padrão de projeto GoF que permite tratar uniformemente objetos primitivos e compostos.

para as duas partes desse método, o que sugere que essa não é uma boa estratégia.

- Alocar o TestResult no chamador.

Alocaremos os TestResults nos chamadores. Esse padrão é chamado Collecting Parameter\*.

#### **TestCaseTest**

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    result= TestResult()
    suite.run(result)
    assert("2 run, 1 failed" == result.summary())
```

Essa solução tem a vantagem de run() agora não ter retorno explícito:

#### **TestSuite**

```
def run(self, result):
    for test in tests:
        test.run(result)
```

#### **TestCase**

```
def run(self, result):
    result.testStarted()
    self.setUp()
    try:
        method = getattr(self, self.name)
        method()
    except:
        result.testFailed()
    self.tearDown()
```

Agora podemos limpar a invocação dos testes no final do arquivo:

```
suite= TestSuite()
suite.add(TestCaseTest("testTemplateMethod"))
suite.add(TestCaseTest("testResult"))
suite.add(TestCaseTest("testFailedResultFormatting"))
suite.add(TestCaseTest("testFailedResult"))
suite.add(TestCaseTest("testSuite"))
result= TestResult()
suite.run(result)
print result.summary()
```

---

\* N. de R. T.: Padrão de implementação Collecting Parameter, em que um objeto – na verdade uma coleção (lista, vetor, etc.) – é passado repetidamente como um parâmetro para um método de forma a coletar informação desse método, que adiciona itens à coleção. Kent Beck definiu esse padrão. Mais detalhes em KERIEVSKY, Joshua. *Refatoração para padrões*. Porto Alegre: Bookman, 2008.

~~Invoque o método teste~~  
~~Invoque setUp primeiro~~  
~~Invoque tearDown depois~~  
Invoque tearDown mesmo se o método teste falhar  
**Rode múltiplos testes**  
~~Informe resultados coletados~~  
~~String de registro em WasRun~~  
~~Informar testes que falharam~~  
Capturar e informar erros em setUp  
Crie TestSuite de uma classe TestCase

Há duplicação substancial aqui que poderíamos eliminar se tivéssemos uma forma de construir uma suíte automaticamente, dada uma classe de testes.

Contudo, primeiro temos que corrigir os quatro testes que falham (eles usam a antiga interface de run sem argumentos):

#### TestCaseTest

```
def testTemplateMethod(self):  
    test= WasRun("testMethod")  
    result= TestResult()  
    test.run(result)  
    assert("setUp testMethod tearDown " == test.log)  
def testResult(self):  
    test= WasRun("testMethod")  
    result= TestResult()  
    test.run(result)  
    assert("1 run, 0 failed" == result.summary())  
def testFailedResult(self):  
    test= WasRun("testBrokenMethod")  
    result= TestResult()  
    test.run(result)  
    assert("1 run, 1 failed" == result.summary())  
def testFailedResultFormatting(self):  
    result= TestResult()  
    result.testStarted()  
    result.testFailed()  
    assert("1 run, 1 failed" == result.summary())
```

Perceba que cada teste aloca um `TestResult`, exatamente o problema resolvido por `setUp()`. Podemos simplificar os testes (ao custo de fazê-los um pouco mais difíceis de ler) criando o `TestResult` em `setUp()`:

#### TestCaseTest

```
def setUp(self):  
    self.result= TestResult()  
def testTemplateMethod(self):  
    test= WasRun("testMethod")  
    test.run(self.result)  
    assert("setUp testMethod tearDown " == test.log)  
def testResult(self):  
    test= WasRun("testMethod")  
    test.run(self.result)
```

```

assert("1 run, 0 failed" == self.result.summary())
def testFailedResult(self):
    test= WasRun("testBrokenMethod")
    test.run(self.result)
    assert("1 run, 1 failed" == self.result.summary())
def testFailedResultFormatting(self):
    self.result.testStarted()
    self.result.testFailed()
    assert("1 run, 1 failed" == self.result.summary())
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    suite.run(self.result)
    assert("2 run, 1 failed" == self.result.summary())

```

~~Invoque o método teste~~

~~Invoque setUp primeiro~~

~~Invoque tearDown depois~~

Invoque tearDown mesmo se o método teste falhar

~~Rode múltiplos testes~~

~~Informe resultados coletados~~

~~String de registro em WasRun~~

~~Informar testes que falharam~~

Capturar e informar erros em setUp

Crie TestSuite de uma classe TestCase

Todos aqueles `selfs` extras são um pouco feios, mas isso é Python. Se fosse uma linguagem objeto, então o `self` seria assumido e referências para variáveis globais requereriam qualificação. Em vez disso, é uma linguagem de script com adição de suporte a objetos (excelente suporte a objetos para ser exato), então a referência global é implícita e se referir a `self` é explícito.

Eu deixarei o resto desses itens para você e suas recém-descobertas habilidades em TDD.

Para revisar, nesse capítulo:

- Escrevemos um teste para um `TestSuite` (suíte de testes).
- Escrevemos parte da implementação, mas sem fazer o teste funcionar. Isso era uma violação da Regra\*. Se você observou isso agora, pegue dois casos de testes do fundo da caixa. Estou certo de que há uma implementação faz de conta simples que teria feito o caso de teste funcionar, então poderíamos refatorar sob uma barra verde, mas não posso achar que é o momento.
- Mudamos a interface do método `run` para que o item e o Composite de itens pudessem funcionar de forma idêntica, e, então, finalmente obtivemos o teste funcionando.
- Fatoramos o código comum de inicialização.

\* N. de R. T.: Alusão do autor à regra básica do TDD de escrever testes que funcionam antes de escrever código.

# CAPÍTULO 24

---

## Retrospectiva xUnit

Se chegar a hora de implementar seu próprio framework de testes, então a sequência apresentada na Parte II desse livro pode servir como seu guia. Os detalhes da implementação não são tão importantes como os casos de teste. Se você puder dar suporte a um conjunto de casos de teste como aqueles mostrados daqui, então pode escrever testes que são isolados e podem ser compostos, e você estará no caminho de ser capaz de desenvolver de modo test-first (teste primeiro).

xUnit foi portado para mais de 30 linguagens de programação no momento da escrita deste livro. É provável que a linguagem que você usa já tenha uma implementação. Mas há um monte de razões para implementar sua própria xUnit, mesmo se houver uma versão já disponível:

- Domínio – O espírito da xUnit é simplicidade. Martin Fowler disse, “nunca antes nos anais da engenharia de software tantos deveram tanto a algo que foi desenvolvido em tão poucas linhas de código”. Algumas das implementações são um pouco complicadas para o meu gosto. Desenvolver a sua própria dará a você uma ferramenta sobre a qual tem um sentimento de domínio.
- Exploração – Quando estou diante de uma nova linguagem de programação, eu implemento xUnit. Quanto eu tiver os primeiros oito testes dos dez rodando terei explorado muitas das habilidades que usarei na programação do dia a dia.

Quando você começa a usar xUnit, descobrirá uma grande diferença entre asserções que falham e os outros tipos de erros que ocorrem na execução dos testes: falhas de asserção sistematicamente levam mais tempo para depurar. Por causa disso, a maioria das implementações de xUnit distingue entre falhas – isto é, falhas de asserção – e erros. As GUIs as apresentam de forma diferente, frequentemente com os erros no topo.

JUnit declara uma única interface Test que TestCase e TestSuite implementam. Se você quiser que ferramentas JUnit sejam capazes de rodar seus testes, então pode implementar a interface Test também.

```
public interface Test {  
    public abstract int countTestCases();  
    public abstract void run(TestResult result);  
}
```

Linguagens com tipagem otimista (dinâmica) nem mesmo precisam declarar sua fidelidade a uma interface – elas podem simplesmente implementar as operações. Se você escrever uma linguagem de script de testes, então Script pode implementar countTestCases() para retornar 1 e executar para notificar o TestResult no caso de falha, e você pode executar seus scripts com os TestCases comuns.

# PARTE III

---

## Padrões para Desenvolvimento Guiado por Testes

O que segue são os padrões de “maior sucesso” para TDD. Alguns dos padrões são truques de TDD, alguns são padrões de projeto e alguns são refatorações. Se você está familiarizado com esses truques, então os padrões daqui mostrará a você como os tópicos interagem em TDD. Caso contrário, há material suficiente para levá-lo pelos exemplos do livro e para aguçar seu apetite pelos tratamentos abrangentes encontrados em outros lugares.

# CAPÍTULO 25

---

## Padrões de Desenvolvimento Guiado por Testes

Precisamos responder algumas questões estratégicas básicas antes de falarmos sobre os detalhes de como testar:

- O que queremos dizer com testar?
- Quando testamos?
- Como escolhemos que lógica testar?
- Como escolhemos quais dados testar?

---

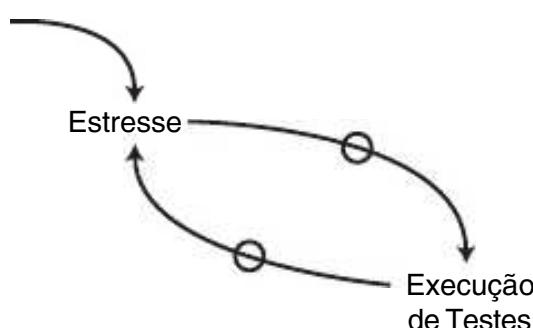
### Teste (substantivo)

Como você testa seu software? Escreva um teste automático.

Testar é um verbo que significa “avaliar”. Nenhum engenheiro de software libera nem mesmo a menor mudança sem testá-la, exceto os muito confiantes e os muito desleixados. Eu vou assumir que se você chegou tão longe, não é nenhum dos dois. Embora você possa testar suas mudanças, testar mudanças não é o mesmo que ter testes. Teste é também um substantivo, “um procedimento que conduz a aceitação ou rejeição”. Por que o substantivo teste, um procedimento que se executa automaticamente, parece diferente do verbo testar, como apertar alguns botões e olhar respostas na tela?

O que segue é um diagrama de influência, à la Gerência de Qualidade de Software de Gerry Weinberg. Uma seta entre nodos significa que um aumento no primeiro nodo implica em um aumento no segundo nodo. Uma seta com um círculo significa que um aumento no primeiro nodo implica um decréscimo no segundo nodo.

O que acontece quando o nível de estresse aumenta?



**Figura 25.1** A espiral da morte do “sem tempo para testar”.

Isso é um laço de feedback positivo. Quanto mais estresse você sentir, menos vai testar. Quanto menos você testar, mais erros vai cometer. Quanto mais erros você cometer, mais estresse vai sentir. Enxague e repita.

Como você escapa desse laço? Ou introduz um novo elemento, substituindo um dos elementos, ou muda as setas. Nesse caso, substituiremos Testes por Testes Automáticos.

“Acabei de estragar alguma outra coisa com aquela mudança?” A Figura 25.1 mostra a dinâmica no trabalho. Com testes automáticos, quando eu começo a sentir estresse, eu rodo os testes. Testes são a Pedra Filosofal do programador, transmutando medo em tédio. “Não, eu não estraguei nada. Os testes ainda estão verdes.” Quanto mais estresse eu sentir, mais testes vou rodar. Rodar os testes imediatamente me traz uma sensação agradável e reduz o número de erros que eu cometo, o que reduz ainda mais o estresse que sinto.

“Não temos tempo para fazer testes. Entregue assim mesmo!!” O segundo quadro não é garantido. Se o nível de estresse sobe alto o suficiente, ele desaba. Contudo, com os testes automáticos, você tem uma chance de escolher seu nível de temor.

Você deveria rodar o teste depois de tê-lo escrito, mesmo sabendo que vai falhar? Não, não se incomode. Por exemplo, estava trabalhando com um monte de jovens programadores na implementação de transações em memória (uma técnica muito legal que toda linguagem de programação deveria ter). A questão era: como implementaríamos o rollback (retrocesso) se começássemos uma transação, mudássemos algumas variáveis e deixássemos a transação ser recolhida pelo coletor de lixo? Muito fácil de testar, rapazes. Para trás, e observem o mestre trabalhar. Aqui está o teste. Agora, como vamos implementar isso?

Duas horas depois – horas marcadas por frustração, pois um erro ao implementar características de tão baixo nível geralmente fecham o ambiente de desenvolvimento – voltamos para onde começamos. Escrevemos o teste. Executamos ele no capricho. Ele passou. Dã.... A questão toda do mecanismo de transação era que as variáveis não eram realmente mudadas até a transação ser enviada. Certo, eu suponho que você poderia ir em frente e rodar aquele novo teste se quiser.

## Teste isolado (Isolated Test)\*

Como a execução de testes deveria afetar outra execução? Em nada.

Quando eu era um jovem programador – há muito, muito tempo quando tínhamos que desenterrar nossos próprios bits da neve e carregá-los descalços em baldes pesados de volta a nossos cubículos, deixando pequenas pegadas sangrentas para que os lobos seguissem.... Desculpe, apenas reminiscências. Minha primeira experiência com testes automáticos foi um conjunto de testes baseados em GUI de longa duração, durante a noite (você sabe, gravar as teclas pressionadas e os eventos de teclado e repeti-los), para um depurador em que eu estava trabalhando. (Oi, Jothy, oi John!) Toda manhã, quando eu chegava, havia uma bela pilha de papel em minha cadeira descrevendo a execução dos testes da última noite. (Oi, Al!) Em dias bons, haveria uma única folha resumindo que nada estava estragado. Em dias ruins, haveria muitas, muitas folhas, uma para cada teste que não funcionou. Eu comecei a temer os dias em que via uma pilha de papel na minha cadeira.

Eu tirei duas lições dessa experiência. Primeiro, faça testes tão rápidos de executar que possa rodá-los sozinho e rodá-los frequentemente. Desse jeito, eu posso capturar erros antes que outros possam vê-los e não ter que temer chegar de manhã. Segundo, percebi depois de um tempo que uma enorme pilha de papel geralmente não significava uma grande lista de problemas. Mais frequentemente, significava que um teste inicial não tinha funcionado, deixando o sistema em um estado imprevisível para o próximo teste.

Tentamos contornar esse problema começando e parando o sistema entre cada teste, mas isso levava muito tempo, o que me ensinou outra lição sobre buscar testes em pequena escala em vez de buscar na aplicação inteira. Mas a lição principal que eu aprendi foi que testes deveriam ser capazes de ignorar um ao outro completamente. Se tenho um teste que não funciona, eu quero um problema. Se tenho dois testes que não funcionam, eu quero dois problemas.

Uma implicação conveniente de testes isolados é que os testes são independentes de ordem. Se quero pegar um subconjunto de testes e rodá-los, então eu posso fazê-lo sem me preocupar com que o teste vá falhar agora porque um teste que é pré-requisito sumiu.

Performance é a razão usual citada para ter testes compartilhando dados. Uma segunda implicação de testes isolados é que você tem que trabalhar, às vezes trabalhar duro, para dividir seu problema em pequenas dimensões ortogonais, então configurar o ambiente para cada teste é fácil e rápido. Isolamento de testes o encoraja a compor soluções de muitos objetos altamente coesos e fracamente acoplados. Eu sempre ouço que isso é uma boa ideia e ficava feliz quando a alcançava, mas nunca sabia exatamente como atingir alta coesão e baixo acoplamento regularmente até começar a escrever testes isolados.

\* N. de R. T.: Decidimos manter o nome original do padrão em inglês para facilitar identificação, em caso de consulta a bibliografia em inglês.

## Listas de testes (Test List)

O que você deveria testar? Antes de começar, escreva uma lista de todos os testes que sabe que terá que escrever. A primeira parte de nossa abordagem para lidar com estresse de programação é nunca dar um passo à frente a menos que saibamos que nosso pé está indo para terra firme. Quando sentamos para uma sessão de programação, o que é que pretendemos realizar?

Uma estratégia para manter o rastro do que estamos tentando fazer é manter isso tudo em nossas cabeças. Eu tentei isso por muitos anos antes e acho que entrei em um laço de feedback positivo. Quanto mais experiência eu acumulei, eu sabia que mais coisas precisariam ser feitas. Quanto mais coisas eu sabia que precisariam ser feitas, menos atenção eu dava para aquilo que estava fazendo. Quanto menos atenção, menos eu fazia. Quanto menos eu fazia, mais coisas eu sabia que precisariam ser feitas.

Apenas ignorar itens aleatórios na lista e programá-los ao bel prazer não parece quebrar esse ciclo.

Eu tinha o hábito de escrever tudo o que queria fazer nas próximas horas em um pedaço de papel perto do meu computador. Tinha uma lista similar, mas com um escopo semanal ou mensal, fixada na parede. Tão logo tivesse tudo escrito, eu sabia que não esqueceria alguma coisa. Quando um novo item chegava, eu decidia rápida e conscientemente se ele pertencia à lista “agora” ou à lista “mais tarde”, ou se ele não precisava realmente ser feito.

Aplicado ao desenvolvimento dirigido por testes, o que pusemos na lista são os testes que queremos implementar. Primeiro, coloque na lista exemplos de cada operação que você sabe que precisa implementar. Depois, para aquelas operações que já não existem, coloque a versão nula daquela operação na lista. Finalmente, liste todas as refatorações que você acha que terá que fazer de forma a limpar o código no final daquela sessão.

Em vez de definir os testes, poderíamos apenas ir em frente e implementá-los todos. Há um monte de razões para a escrita em massa de testes não ter funcionado para mim. Primeiro, cada teste que você implementa é um pouco de inércia quando tem que refatorar. Com ferramentas de refatoração automática (por exemplo, você tem um item no menu que renomeia a declaração e todos os usos de uma variável), isso é o menor problema. Mas, quando você implementa dez testes e então descobre que os argumentos precisam estar em ordem contrária, é muito menos provável que você vá arrumá-los. Segundo, se você tem dez testes que não funcionam, tem um longo caminho até a barra verde. Se você quer chegar ao verde rapidamente, tem que jogar fora os dez testes. Se você quer todos os testes funcionando, então vai ficar olhando para barra vermelha por um longo tempo. Se você está tão viciado na barra verde que não pode ir ao banheiro se ela está vermelha, então isso pode ser uma eternidade.

Alpinistas conservadores têm uma regra segundo a qual, de seus dois pés e duas mãos, três deles devem estar sempre fixados. Movimentos em que você vai soltar dois deles ao mesmo tempo são muito mais perigosos. A forma pura de TDD, na qual você nunca está a mais que uma mudança da barra verde, é como essa regra de três dos quatro.

Conforme você faz os testes rodarem, a implementação implicará em novos testes. Escreva os testes no final da lista. Da mesma forma com refatorações.

“Isso está ficando feio.”

“(suspiro) Ponha isso na lista. Nós chegaremos lá antes da verificação.”

Itens que são deixados na lista quando a sessão está feita precisam de cuidados. Se você está realmente no meio de um pedaço de funcionalidade, então use a mesma lista depois. Se você descobriu refatorações maiores que estão fora do escopo do seu momento, então as coloque na lista de “mais tarde”. Não consigo lembrar de alguma vez ter movido um caso de teste para a lista de “mais tarde”. Se eu conseguir pensar em um teste que pode não funcionar, fazê-lo funcionar é mais importante que liberar meu código.

---

## Teste primeiro (Test First)

Quando você deveria escrever seus testes? Antes de escrever o código que vai ser testado.

Você não testará depois. Sua meta como programador é executar a funcionalidade. Contudo, você precisa de uma maneira de pensar sobre o projeto; você precisa de um método para o controle de escopo.

Vamos considerar o diagrama de influência usual que relaciona estresse e teste (mas não teste de estresse, isso é diferente): Estresse acima, negativamente conectado a Teste abaixo, negativamente conectado a Estresse (veja Teste (substantivo) anteriormente neste capítulo). Quanto mais estresse sentir, é menos provável que testará o suficiente. Quando você sabe que não testou o suficiente, aumenta seu estresse. Laço de feedback positivo. Mais uma vez, é necessária uma forma de quebrar o ciclo.

E se adotarmos a regra de sempre testarmos primeiro? Então poderíamos inverter o diagrama e obter um ciclo virtuoso: Teste Primeiro acima está negativamente conectado a Estresse abaixo, negativamente conectado a Teste Primeiro.

Quando testamos primeiro, reduzimos o estresse, o que nos faz mais aptos a testar. Há um monte de outros elementos que alimentam o estresse, contudo, assim os testes devem existir em outros ciclos virtuosos, ou eles serão abandonados quando o estresse aumentar o bastante. Mas, a compensação imediata para teste – uma ferramenta para controle de projeto e escopo – sugere que seremos capazes de começar a fazê-lo e nos mantermos fazendo-o mesmo sob estresse moderado.

## Defina uma asserção primeiro (Assert First)

Quando eu deveria escrever as asserções? Tente escrevê-las primeiro. Você simplesmente não ama autossimilaridade?

- Por onde você deve começar a construir um sistema? Com histórias de que quer ser capaz de contar sobre o sistema terminado.
- Por onde você deve começar a escrever um pedaço de funcionalidade? Com testes que quer que passem com o código terminado.
- Por onde você deve começar a escrever um teste? Com as asserções que passarão (serão bem-sucedidas) quando estiver feito.

Jim Newkirk me introduziu a essa técnica. Quando eu testo definindo uma asserção primeiro, acho que isso tem um poderoso efeito simplificador. Quando você está escrevendo um teste, está resolvendo muitos problemas de uma vez, mesmo se não tiver mais que pensar sobre a implementação.

- A que lugar a funcionalidade pertence? Ela é uma modificação de um método existente, um novo método em uma classe existente, um nome de método existente implementado em um novo lugar ou uma nova classe?
- Quais nomes deveriam ser chamados?
- Como você vai verificar a resposta certa?
- Qual é a resposta certa?
- Quais outros testes esse teste sugere?

Cérebros do tamanho de ervilhas como o meu não podem, possivelmente, fazer um bom trabalho ao resolver todos esse problemas de uma vez. Os dois problemas da lista que podem ser facilmente separados do resto são “qual é a resposta certa?” e “como eu vou verificar?”

Aqui está um exemplo. Suponha que quero me comunicar com outro sistema através de um socket. Quando estiver pronto, o socket deveria estar fechado e deveríamos ter lido a string abc.

```
testCompleteTransaction() {  
    ...  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

De onde a resposta vem? Do socket, certamente:

```
testCompleteTransaction() {  
    ...  
    Buffer reply= reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

E o socket? Nós o criamos conectando ao servidor:

```
testCompleteTransaction() {  
    ...  
    Socket reader= Socket("localhost", defaultPort());  
    Buffer reply= reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

Mas, antes disso, precisamos abrir um servidor:

```
testCompleteTransaction() {  
    Server writer= Server(defaultPort(), "abc");  
    Socket reader= Socket("localhost", defaultPort());  
    Buffer reply= reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

Agora podemos ter que ajustar alguns nomes baseados no uso real, mas criamos o esboço do teste em passos pequeninos, informando cada decisão com feedback em poucos segundos.

---

## Dados de teste (Test Data)

Que dados você usa para testes da forma “testar primeiro”? Use dados que façam os testes fáceis de ler e seguir. Você está escrevendo testes para uma audiência. Não espalhe valores de dados apenas por espalhar. Se há uma diferença nos dados, então ela deveria ser significativa. Se não há diferença conceitual entre 1 e 2, use 1.

Dados de Teste não são uma licença para uma parada repentina na confiança plena. Se seu sistema tem que manipular múltiplas entradas, então seus testes de-

veriam refletir múltiplas entradas. Contudo, não tenha uma lista de dez itens como entradas se uma lista de três itens guiarão você às mesmas decisões de projeto e implementação.

Um truque nos Dados de Teste é tentar nunca usar a mesma constante para significar mais de uma coisa. Se estou testando um método `plus()`, é tentador testar  $2 + 2$ , pois é o exemplo clássico de adição, ou  $1 + 1$ , pois é muito simples. E se estivermos com os argumentos invertidos na implementação? (Certo, certo, isso não importa no caso de `plus()`, mas você pegou a ideia.) Se usarmos  $2$  para o primeiro argumento, por exemplo, então deveríamos usar  $3$  para o segundo argumento. ( $3 + 4$  foi um caso de teste divisor de águas quando trazia de volta uma nova máquina virtual para Smalltalk nos velhos dias.)

A alternativa para Dados de Teste é o uso de Dados Realistas, em que você usa dados do mundo real. Dados Realistas são úteis quando:

- Você está testando sistemas de tempo real usando rastros de eventos externos recolhidos da execução real.
- Você está equiparando a saída do sistema atual com a saída do sistema anterior (teste paralelo).
- Você está refatorando uma simulação e esperando, precisamente, as mesmas respostas quando tiver terminado, particularmente se a precisão de ponto flutuante pode ser um problema.

---

## Dados evidentes (Evident Data)

Como você representa o objetivo do dado? Inclua resultados esperados e reais no próprio teste e tente fazer seu relacionamento evidente. Você está escrevendo testes para um leitor, não apenas para o computador. Alguém daqui a décadas perguntará a si mesmo ou mesma, “em que diabos esse palhaço estava pensando?” Você gostaria de deixar tantas pistas quanto possível, especialmente se aquele leitor frustrado será você.

Aqui está um exemplo. Se convertermos uma moeda em outra, levamos uma porcentagem de 1,5 como comissão pela transação. Se a taxa de câmbio de USD para GBP está 2:1, então, se trocamos \$100, deveríamos ter  $50 \text{ GBP} - 1,5\% = 49,25 \text{ GBP}$ .

Poderíamos escrever esse teste assim:

```
Bank bank= new Bank();
bank.addRate("USD", "GBP", STANDARD_RATE);
bank.commission(STANDARD_COMMISSION);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(49.25, "GBP"), result);
```

ou poderíamos tentar fazer o cálculo óbvio:

```
Bank bank= new Bank();
bank.addRate("USD", "GBP", 2);
bank.commission(0.015);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(100 / 2 * (1 - 0.015), "GBP"), result);
```

Posso ler esse teste e ver a conexão entre os números usados na entrada e os números usados para calcular o resultado esperado.

Um efeito colateral benéfico de Dados Evidentes é que ele torna a programação mais fácil. Uma vez que tenhamos escrito a expressão na asserção, sabemos o que precisamos programar. De alguma forma, temos que começar o programa para avaliar uma divisão e uma multiplicação. Podemos até usar Faça de conta (*Fake it*<sup>\*</sup>) para descobrir aos poucos a que pertencem as operações.

Dados Evidentes parece ser uma exceção para a regra de não quer números mágicos em seu código. Dentro do escopo de um só método, o relacionamento entre os 5's é óbvio. Se tenho constantes simbólicas que já foram definidas, contudo, eu usaria a forma simbólica.

---

\* N. de R. T.: *Fake it* é um padrão de teste (*testing pattern*) – “retorne uma constante e gradualmente substitua constantes por variáveis”, ver Capítulo 28.

# CAPÍTULO 26

---

## Padrões de Barra Vermelha

Esses padrões são sobre quando você escreve testes, onde você escreve testes e quando você para de escrever testes.

---

### Teste de um só passo (One Step Test)\*

Qual é o próximo teste que você deveria pegar da lista? Pegue um teste que ensinará a você algo e que você confia que pode implementar.

Cada teste deveria representar um passo em direção à sua meta global. Olhando para a seguinte Lista de Testes, qual é o próximo teste que deveríamos pegar?

- Soma
- Subtração
- Multiplicação
- Divisão
- Soma de similares
- Igualdade
- Igualdade entre nulos
- Troca nula
- Trocá uma moeda
- Trocá duas moedas
- Taxa cruzada

Não há resposta certa. O que é um passo para mim, nunca tendo implementado esses objetos antes, será um décimo de um passo para você com sua vasta experiência. Se você não achar que qualquer teste na lista representa um só passo,

---

\* N. de R. T.: Decidimos manter também o nome original do padrão em inglês para facilitar identificação, em caso de consulta a bibliografia em inglês.

então adicione alguns testes novos que representariam progresso em direção aos itens de lá.

Quando olho uma Lista de Testes, eu penso, “aquilo é óbvio, aquilo é óbvio, eu não tenho ideia, óbvio, em que eu estava pensando com esse aqui, ah, esse aqui eu posso fazer”. O último teste é o próximo teste que eu implemento. Ele não me bate como óbvio, mas eu estou confiante de que posso fazer o trabalho.

Um programa produzido a partir de testes como esse pode parecer ser escrito de forma top-down, pois você pode começar com um teste que representa um caso simples da computação inteira. Um programa produzido a partir de teste pode também parecer ser escrito de forma bottom-up, pois você começa com pedaços pequenos e os agrava mais e mais.

Nem a visão top-down, nem a visão bottom-up descrevem realmente o processo de forma útil. Primeiro, uma metáfora vertical é uma visualização simplista de como programas mudam com o tempo. Produção implica em um tipo de laço de feedback recíproco no qual o ambiente afeta o programa e o programa afeta o ambiente. Segundo, se precisamos ter uma direção em nossa metáfora, então de-conhecido-a-desconhecido é uma descrição útil. De-conhecido-a-desconhecido implica que temos algum conhecimento e alguma experiência no que desenhar, e que esperamos aprender no curso do desenvolvimento. Ponha esses dois juntos e temos programas produzidos do conhecido ao desconhecido.

---

## Teste inicial (Starter Test)

Com qual teste você deveria começar? Comece testando uma variante de uma operação que não faz nada.

A primeira questão que você tem que perguntar com uma nova operação é “a que lugar ela pertence?”. Até ter respondido essa questão, você não saberá o que digitar para o teste. No espírito de resolver um problema por vez, como você pode responder apenas essa questão e não outra?

Se você escrever um teste realista primeiro, então se encontrará resolvendo um monte de problemas de uma vez:

- A que lugar essa operação pertence?
- Quais são as entradas corretas?
- Qual é a saída correta para dadas entradas?

Começar com um teste realista deixará você sem feedback por muito tempo. Vermelho/verde/refatore, vermelho/verde/refatore. Você quer que esse laço leve apenas alguns minutos.

Você pode encurtar o laço escolhendo entradas e saídas que são trivialmente fáceis de descobrir. Por exemplo, um cartaz no grupo de discussão de Programação Extrema perguntava como escrever um redutor de polígonos com “teste primeiro”. As entradas são uma malha de polígonos e a saída é uma malha de polígonos que

descreve precisamente a mesma superfície, mas com o menor número de polígonos possível. “Como eu posso me guiar por testes nesse problema quando ter um teste funcionando requer a leitura de teses de doutorado?”

Teste Inicial fornece uma resposta:

- A saída deveria ser a mesma da entrada. Algumas configurações de polígonos já estão normalizadas, incapazes de sofrerem redução.
- A entrada deveria ser a menor possível, como um polígono simples, ou mesmo uma lista vazia de polígonos.

Meu Teste Inicial pareceu com isso:

```
Reducer r= new Reducer(new Polygon());
assertEquals(0, reducer.result().npoints);
```

Pronto! O primeiro teste está rodando. Agora para todo o resto dos testes na lista

...

Testes de Um Só Passo se aplicam a seu Teste Inicial. Pegue um Teste Inicial que o ensinará alguma coisa, mas que você está certo que pode fazer funcionar rapidamente. Se está implementando uma aplicação pela enésima vez, então pegue um teste que requer uma operação ou duas. Você estará, justificadamente, confiante de que pode fazê-lo funcionar. Se você está implementando algo cabeludo e complicado pela primeira vez, então precisa de uma pequena dose de coragem imediatamente.

Acho que meu Teste Inicial está muitas vezes em um nível mais alto: mais como um teste de aplicação que como os testes a seguir. Um exemplo que eu frequentemente uso para guiar testes é um servidor simples baseado em socket. O primeiro teste parece com isso:

```
StartServer
Socket= new Socket
Message= "hello"
Socket.write(message)
assertEquals(message, socket.read)
```

O resto dos testes são escritos no servidor, “Assumindo que recebamos uma string como essa....”

## Teste de explicação (Explanation Test)

Como você difunde o uso de teste automatizado? Peça e dê explicações em termos de testes.

Pode ser frustrante ser o único TDD em um time. Em breve, você perceberá menos problemas de integração e notificações de defeitos no código testado, e os projetos serão mais simples e mais fáceis de explicar. Pode até acontecer de pessoas ficarem completamente entusiasmadas com testes e com testar primeiro.

Cuidado com o entusiasmo dos recém-convertidos. Nada vai parar mais rápido a disseminação de TDD que enfiá-lo na cara das pessoas. Se você não é um gerente ou um líder, não pode forçar ninguém a mudar o jeito que trabalha.

O que pode fazer? Um início simples é começar pedindo explicações em termos de casos de teste: “Deixa eu ver se entendi o que está dizendo. Por exemplo, se eu tenho um Foo como esse e um Bar como aquele, então a resposta devia ser 76?” Uma técnica acompanhante é começar a dar explicações em termos de teste: “Aqui é como funciona agora. Quando eu tenho um Foo como esse e um Bar como aquele, então a resposta é 76. Se eu tenho um Foo como aquele e um Bar como esse, entretanto, eu gostaria que a resposta fosse 67.”

Você pode fazer isso em mais altos níveis de abstração. Se alguém está explicando um diagrama de sequência para você, então pode pedir permissão para convertê-lo em uma notação mais familiar. Então você digita um caso de teste que contenha todos os objetos e variáveis externamente visíveis no diagrama.

---

## Teste de aprendizado (Learning Test)<sup>1</sup>

Quando você escreve testes para software produzido externamente? Antes da primeira vez que você vai usar uma nova fábrica no pacote.

Digamos que vamos desenvolver algo em cima da biblioteca Perfil de Dispositivo de Informação Móvel para Java. Queremos armazenar alguns dados no Record-Store e recuperá-los. Nós apenas escrevemos o código e esperamos que funcione? Isso é um jeito de desenvolver.

Uma alternativa é perceber que estamos prestes a usar um novo método de uma classe. Em vez de apenas usá-lo, escrevemos um pequeno teste que verifica que a API funciona como esperado. Então, poderíamos escrever:

```
RecordStore store;

public void setUp() {
    store= RecordStore.openRecordStore("testing", true);
}

public void tearDown() {
    RecordStore.deleteRecordStore("testing");
}

public void testStore() {
    int id= store.addRecord(new byte[] {5, 6}, 0, 2);
    assertEquals(2, store.getRecordSize(id));
    byte[] buffer= new byte[2];
```

---

<sup>1</sup> Obrigado a Jim Newkirk e Laurent Bossavit por sugerirem (independentemente um do outro) esse padrão.

```
assertEquals(2, store.getRecord(id, buffer, 0));
assertEquals(5, buffer[0]);
assertEquals(6, buffer[1]);
}
```

Se nosso entendimento da API estiver correto, então o teste passará de primeira.

Jim Newkirk apresentou um projeto em que Testes de Aprendizagem eram rotineiramente escritos. Quando novas liberações do pacote chegavam, primeiro os testes eram executados (e corrigidos, se necessário). Se os testes não rodavam, então não havia sentido executar a aplicação, pois certamente não rodaria. Uma vez que os testes executavam, a aplicação executava sempre.

---

## Outro teste (Another Test)

Como você evita que uma discussão técnica desvie do assunto? Quando uma ideia tangencial emerge, adicione um teste à lista e volte ao assunto.

Eu amo discussões errantes (você leu a maior parte do livro agora, então, provavelmente, chegou a essa conclusão sozinho). Manter uma conversa estritamente no caminho é uma grande forma de asfixiar ideias brilhantes. Você pula daqui para lá para acolá, e como chegamos aqui? Quem se importa, isso é legal!

Às vezes, programação apoia-se em avanços. A maioria da programação, contudo, progride em pequenos avanços. Eu tenho dez coisas para implementar. Eu me torno um perfeito procrastinador do item número quatro. Recuar a conversas fiadas é uma das minhas formas de evitar trabalho (e talvez o medo que vem com ele).

Dias improdutivos inteiros me ensinaram que, às vezes, é melhor ficar na trilha. Quando eu me sinto dessa forma, saúdo novas ideias com respeito, mas não permito que desviem minha atenção. Eu as escrevo no final da lista e, então, volto ao que estava trabalhando.

---

## Teste de regressão (Regression Test)

Qual é a primeira coisa que você faz quando um defeito é informado? Escreva o menor teste possível que falhe e que, uma vez rodado, será reparado.

Testes de regressão são testes que, com perfeito conhecimento prévio, você teria escrito quando estava codificando originalmente. Cada vez que você tem que escrever um teste de regressão, pense em como poderia saber, em primeiro lugar, escrever o teste.

Você também ganhará valor testando em nível de aplicação inteira. Testes de regressão para a aplicação dão a seus usuários a chance de falar concretamente a você sobre o que está errado e o que esperavam. Testes de regressão em menor escala são um jeito de melhorar seu teste. O relatório de defeitos será um grande e bizarro número negativo em um relatório. A lição para você é que precisa usar todos os testes que fez quando estiver escrevendo sua Lista de Testes.

Você pode ter que refatorar o sistema antes de poder isolar facilmente o defeito. O defeito nesse caso era o jeito do seu sistema dizer: “Você ainda não terminou de me projetar completamente.”

---

## Pausa (Break)

O que você faz quando se sente cansado ou travado? Faça uma pausa.

Tome um drink, dê uma caminhada, tire uma soneca. Lave suas mãos do compromisso emocional das decisões que acabou de tomar e dos caracteres que digitou.

Frequentemente, essa distância é o que precisa para libertar a ideia que estava faltando. Você estará de pé quando perceber “Eu não tentei com os parâmetros invertidos!” Faça uma pausa de qualquer forma. Dê a si alguns minutos. A ideia não irá embora.

Se você não tiver “a ideia”, então reveja seus objetivos para a sessão. Eles ainda são realistas, ou você deveria escolher novos objetivos? O que está tentando realizar é impossível? Se for, quais são as implicações para o time?

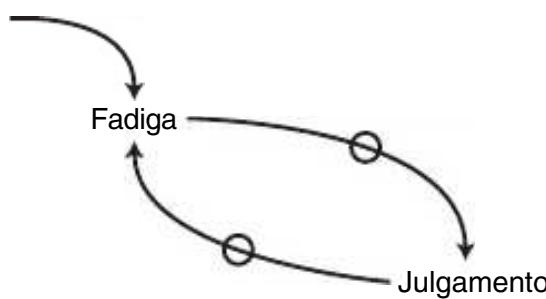
Dave Ungar chama isso de sua Metodologia do Chuveiro. Se você sabe o que digitar, então digite. Se você não sabe o que digitar, então tome uma ducha e fique no chuveiro até saber o que digitar. Muitos times serão mais felizes, mais produtivos e vão cheirar muito melhor se seguirem seu conselho. TDD é um refinamento da Metodologia do Chuveiro de Ungar. Se você sabe o que digitar, digite a Implementação Óbvia. Se você não sabe o que digitar, então Faça de conta\*. Se o projeto certo ainda não está claro, então Triangule\*\*. Se você ainda não sabe o que digitar, então pode tomar aquela ducha.

A Figura 26.1 mostra a dinâmica no trabalho em fazer um Intervalo. Você está ficando cansado, então é menos capaz de perceber que está cansado, então vai ficando mais e mais cansado.

---

\* N. de R. T.: Use o padrão de teste Faça de conta (Fake It), ver Capítulo 28.

\*\* N. de R. T.: Use o padrão de teste Triangule (Triangulate), ver Capítulo 28.



**Figura 26.1** *Fadiga afeta negativamente o Julgamento, o que afeta negativamente a Fadiga.*

O caminho para fora desse laço é introduzir um elemento adicional de fora.

- Na escala de horas, manter uma garrafa d'água ao lado do teclado para que a biologia forneça a motivação para intervalos regulares.
- Na escala de um dia, compromissos depois do horário regular de trabalho poderão ajudá-lo a parar quando precisar dormir antes de continuar.
- Na escala de uma semana, compromissos de finais de semana ajudam a tirar sua consciência, pensamentos sugadores de energia, do trabalho. (Minha esposa jura que eu tenho minhas melhores ideias na sexta-feira à noite.)
- Na escala de um ano, políticas de férias obrigatórias ajudam você a renovar-se completamente. Os franceses fazem isso direito – duas semanas contíguas de férias não são o suficiente. Você gasta a primeira semana descomprimindo e a segunda semana se preparando para voltar ao trabalho. Assim, três semanas, ou melhor, quatro semanas são necessárias para você ser o mais eficiente possível pelo resto do ano.

Há um outro lado em fazer intervalos. Às vezes, quando confrontados com um problema difícil, o que você precisa fazer é continuar, avançar sobre ele. Contudo, a cultura de programação é tão infectada pelo espírito do macho – “Eu vou arruinar minha saúde, me alienar da minha família e me matar se necessário” – que não me sinto compelido a dar qualquer conselho nesse sentido. Se você está viciado em cafeína e não está fazendo qualquer progresso, então talvez não devesse fazer tantos intervalos. Entretanto, dê uma caminhada.

## Faça de novo (Do Over)

O que você faz quando se sente perdido? Jogue o código fora e comece de novo.

Você está perdido. Fez um intervalo, lavou as mãos no riacho, escutou o sino do templo Tibetano e ainda está perdido. O código que estava indo tão bem uma hora atrás está uma confusão agora; você não consegue pensar em como ter o próximo caso de teste funcionando e pensou em mais 20 testes que realmente deveria implementar.

Isso me aconteceu muitas vezes enquanto estava escrevendo este livro. Eu tinha o código um pouco retorcido. “Mas, eu tenho que terminar o livro. As crianças estão famintas, e os coletores de impostos estão batendo à porta.” Minha reação seria desenrolá-lo o suficiente para seguir em frente. Depois de uma pausa para reflexão, começar de novo sempre fazia mais sentido. A única vez que avancei de qualquer maneira, tive que jogar fora 25 páginas de manuscrito, pois estava baseado em uma óbvia decisão de programação estúpida.

Meu exemplo preferido de fazer de novo é uma história que Tim Mackinnon me contou. Ele estava entrevistando uma pessoa através do recurso simples de pedir a ela para programar em par com ele por uma hora. No final da sessão, eles implementaram muitos novos casos de teste e fizeram algumas boas refatorações. Contudo, era final do dia, e eles se sentiam cansados quando fizeram, então descartaram seu trabalho.

Se você programa em pares, mudar de parceiros é uma boa forma de motivar fazer de novo produtivos. Você tentará explicar a bagunça complicada que fez por alguns minutos quando seu novo parceiro, completamente não envolvido com os enganos que você fez, pegará gentilmente o teclado e dirá: “Eu sinto muitíssimo por ser tão direto, mas e se começássemos assim...”

---

## Mesa barata, cadeira legal (Cheap Desk, Nice Chair)

Qual configuração física você deveria usar para TDD? Tenha uma cadeira realmente legal, poupando no resto dos móveis se necessário.

Você não consegue programar bem se suas costas doem. Contudo, organizações que gastarão \$100.000 por mês em um time não gastarão \$10.000 em cadeiras decentes.

Minha solução é usar mesas dobráveis baratas e feias para meus computadores, mas comprar as melhores cadeiras que eu puder encontrar. Eu tenho muito espaço na mesa e posso facilmente conseguir mais, e estou fresco e preparado para programar de tarde e pela manhã. Fique confortável quando estiver programando em pares. Limpe a superfície da mesa o suficiente para poder deslizar o teclado para trás e para frente. Cada parceiro deveria ser capaz de sentar-se confortável e diretamente em frente ao teclado quando estiver pilotando\*. Um dos meus truques de treino favorito é chegar atrás de um par que está programando sem parar e gentilmente deslizar o teclado para que fique confortavelmente colocado para a pessoa digitar.

---

\* N. de R. T.: Na programação por pares, usa-se o termo “piloto” para designar quem está usando o teclado e “copiloto” para o outro parceiro.

Manfred Lange assinala que alocação cuidadosa de recursos também se aplica ao hardware de computador. Obtenha máquinas baratas/lentas/velhas para ver e-mails e navegação pessoal e as máquinas mais potentes possíveis para desenvolvimento compartilhado.

# CAPÍTULO 27

---

## Padrões de Teste

Esses padrões são técnicas mais detalhas para escrever testes.

---

### Teste filho (Child Test)

Como você executa um caso de teste que se mostrou muito grande? Escreva um caso de teste menor que represente a parte que não funciona do caso de teste maior. Consiga rodar o caso de teste menor. Reintroduza o caso de teste maior.

O ritmo vermelho/verde/refatore é tão importante para sucesso contínuo que, quando você está arriscando perdê-lo, vale a pena um esforço extra para mantê-lo. Isso comumente me acontece quando escrevo um teste que, accidentalmente, requer muitas mudanças de forma a funcionar. Mesmo dez minutos com uma barra vermelha me dá arrepios.

Quando escrevo um teste que é muito grande, primeiro tento aprender a lição. Por que ele era muito grande? O que eu poderia ter feito diferente para ser menor? Como estou me sentindo agora mesmo?

Com a contemplação metafísica central terminada, eu apago o teste desagradável e começo de novo. “Bem, ter essas três coisas funcionando de uma vez era demais. Contudo, se eu tiver A, B e C funcionando, conseguir fazer a coisa inteira funcionar seria moleza.” Às vezes, eu realmente excluo o teste e, às vezes, apenas mudo o nome para começar com um x para que não execute. (Posso contar um segredo? Às vezes, nem mesmo me incomodo em apagar o teste desagradável. Pssss... eu permaneço com os dois, conte-os, dois testes não funcionando por não mais que alguns minutos enquanto eu faço o teste filho funcionar. Eu poderia estar cometendo um erro ao fazer isso. Dois testes não funcionando poderiam facilmente ser um resquício dos meus maus velhos tempos de teste-depois-se-der.)

Tente ambas as formas. Veja se você se sente diferente; programe diferente quando tiver dois testes que não funcionam. Responda conforme o caso.

---

## Objeto simulado (Mock Object)

Como você testa um objeto que se baseia em um recurso caro ou complicado? Crie uma versão faz de conta do recurso que responde com constantes.

Há ao menos um material com valor equivalente a um livro sobre Objeto Simulado<sup>1</sup>, mas isso servirá como uma introdução. O exemplo clássico é um banco de dados. Bancos de dados levam um longo tempo para iniciarem; são difíceis de manter limpos; e, se estão localizados em um servidor remoto, amarram seus testes a um local físico em uma rede. O banco de dados também é uma fonte fértil de erros no desenvolvimento.

A solução é não usar um banco de dados real na maior parte do tempo. A maioria dos testes é escrita em termos de um objeto que age como um banco de dados, mas está apenas assentado na memória.

```
public void testOrderLookup() {  
    Database db= new MockDatabase();  
    db.expectQuery("select order_no from Order where cust_no is 123");  
    db.returnResult(new String[] {"Order 2" , "Order 3"});  
    . . .  
}
```

Se o MockDatabase não tiver a consulta que espera, então ele lança uma exceção. Se a consulta está correta, então ele retorna algo que parece com um conjunto resultando construído por strings constantes.

Outro valor de simulações, além de desempenho e confiabilidade, é legibilidade. Você pode ler o teste precedente de um extremo ao outro. Se tivesse um banco de dados de testes cheio de dados reais, quando vê que uma consulta deveria ter resultado em 14 respostas, não tem ideia do porquê de 14 ser a resposta certa.

Se você quiser usar Objetos Simulados, não pode armazenar facilmente recursos caros em variáveis globais (mesmo que mascarados como Singletons). Se você fizer isso, então terá que configurar a global para um Objeto Simulado, rodar o teste e ter certeza de reinicializar a global quando tiver terminado.

Houve vezes em que eu estava furioso com essa restrição. Massimo Arnoldi e eu estávamos trabalhando em algum código que se baseava em um conjunto de taxas de câmbio armazenadas em uma variável global. Cada teste precisava de subconjuntos diferentes de dados, e, às vezes, precisavam de diferentes taxas de câmbio. Depois de um tempo tentando fazer a variável global funcionar, decidimos, em uma manhã (decisões de projeto corajosas vêm mais frequentemente de manhã para mim), apenas passar o Exchange sempre que precisássemos dele. Pensávamos que teríamos que modificar centenas de métodos. No final, adicionamos

---

<sup>1</sup> Por exemplo, ver [www.mockobjects.com](http://www.mockobjects.com)

um parâmetro a dez ou quinze métodos, e limpamos outros aspectos do projeto pelo caminho.

Objetos Simulados o encorajam no caminho de considerar cuidadosamente a visibilidade de cada objeto, reduzindo o acoplamento no seu projeto. Eles adicionam um risco ao projeto – e se o Objeto Simulado não se comportar como o objeto real? Você pode reduzir essa estratégia tendo um conjunto de testes para o Objeto Simulado que pode também ser aplicado ao objeto real quando tornar-se disponível.

## Autodesvio (Self Shunt)

Como você testa se um objeto se comunica corretamente com outro? Tenha o objeto que está sob teste se comunicando com o caso de teste, em vez do objeto que ele espera.

Suponha que quiséssemos atualizar dinamicamente a barra verde na interface de teste do usuário. Se pudéssemos conectar um objeto ao `TestResult`, então poderíamos ser notificados quando um teste executou, quando ele falhou, quando uma suíte inteira começou e terminou, e assim por diante. Se formos notificados de que um teste executou, atualizaremos a interface.

Aqui está um teste para isso:

### **ResultListenerTest**

```
def testNotification(self):
    result= TestResult()
    listener= ResultListener()
    result.addListener(listener)
    WasRun("testMethod").run(result)
    assert 1 == listener.count
```

O teste precisa de um objeto para contar o número de notificações:

### **ResultListener**

```
class ResultListener:
    def __init__(self):
        self.count= 0
    def startTest(self):
        self.count= self.count + 1
```

Mas, espere. Por que precisamos de um objeto separado para o ouvinte? Podemos apenas usar o próprio caso de teste. O próprio `TestCase` se torna uma espécie de Objeto Simulado.

### **ResultListenerTest**

```
def testNotification(self):
    self.count= 0
    result= TestResult()
    result.addListener(self)
```

```
WasRun("testMethod").run(result)
assert 1 == self.count
def startTest(self):
    self.count= self.count + 1
```

Testes escritos com Self Shunt tendem a serem lidos melhor que testes escritos sem ele. O teste anterior é um bom exemplo. A contagem era 0, e, então, ela era 1. Você pode ler a sequência certa no teste. Como ele chegou a 1? Alguém deve ter chamado `startTest()`. Como `startTest()` foi chamado? Deve ter acontecido quando o teste foi executado. Esse é outro exemplo de simetria – a segunda versão do método teste tem dois valores para `count` em um só lugar, enquanto a primeira versão tem `count` configurado para 0 em uma classe e esperava ser 1 em outra.

Self Shunt pode necessitar que você use Extrair Interface (Extract Interface)\* para ter uma interface para implementar. Você terá que decidir se extrair a interface é mais fácil, ou se testar a classe existente como uma caixa preta é mais fácil. No entanto, eu tenho notado que interfaces extraídas para shunts tendem a ter suas terceira e subsequentes implementações logo em seguida.

Como resultado de usar AutoDesvio, você verá testes em Java que implementam todo o tipo de interfaces bizarras. Em linguagens de tipagem dinâmica, a classe do caso de teste precisa apenas implementar aquelas operações que são realmente usadas na execução do teste. Em Java, todavia, você tem que implementar todas as operações da interface, mesmo que a maioria das implementações esteja vazia; portanto, você gostaria de interfaces tão restritas quanto possível. As implementações deveriam retornar um valor razoável ou lançar uma exceção, dependendo se você quer ser notificado se uma operação inesperada for invocada.

---

## String de registro (Log String)

Como você testa se a sequência em que as mensagens são chamadas está correta? Mantenha um registro (log) em uma string e, quando uma mensagem for chamada, acrescente-a à string.

O exemplo de xUnit serve. Temos um Método Template, no qual esperamos chamar `setUp()`, um método de teste, e `tearDown()`, nessa ordem. Implementando os métodos para gravarem em uma string quando são chamados, lê-se bem o teste:

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    result= TestResult()
    test.run(result)
    assert("setUp testMethod tearDown " == test.log)
```

---

\* N. de R. T.: Extrair Interface é uma refatoração que cria uma nova interface a partir da interface de uma classe, geralmente um subconjunto da original. Ver mais informações no Capítulo 31 deste livro, FOWLER, Martin. *Refatoração: aperfeiçoando o projeto de código existente*. Porto Alegre: Bookman, 2004.

E a implementação é simples também:

### WasRun

```
def setUp(self):
    self.log= "setUp "
def testMethod(self):
    self.log= self.log + "testMethod "
def tearDown(self):
    self.log= self.log + "tearDown "
```

Strings de Registro são particularmente úteis quando você está implementando Observer e espera notificações chegarem em uma certa ordem. Se esperava determinadas notificações, mas não se importa com a ordem, então poderia manter um conjunto de strings e usar comparação de conjuntos na asserção.

String de registro funciona bem com AutoDesvio. O caso de teste implementa os métodos na interface desviada adicionando ao registro e então retornando valores razoáveis.

---

## Modelo de teste de acidentes (Crash Test Dummy)\*

Como você testa código de erro que provavelmente será pouco invocado? Invoque-o de qualquer forma com um objeto especial que lança uma exceção em vez de fazer trabalho real.

Código que não é testado não funciona. Essa parece ser a suposição segura. O que você faz com todas aquelas condições estranhas de erro então? Você tem que testá-las também? Apenas se quiser que elas funcionem.

Vamos dizer que queiramos testar o que acontece com a nossa aplicação quando o sistema de arquivos está cheio. Poderíamos ter um trabalhão criando muitos arquivos grandes e enchendo o sistema de arquivos, ou poderíamos Fazer de conta\*\*. Fazer de conta não soa digno, não é? Nós vamos *simulá-lo*.

Aqui está nosso Modelo de Teste de Acidentes para um arquivo:

```
private class FullFile extends File {
    public FullFile(String path) {
        super(path);
    }
    public boolean createNewFile() throws IOException {
        throw new IOException();
    }
}
```

---

\* N. de R. T.: Crash test dummy é o nome dado aos bonecos usados para testes e simulações de acidentes de carro. O nome traduzido visa manter essa metáfora.

\*\* N. de R. T.: Usar o padrão Faz de conta (Fake It), ver Capítulo 28.

Agora podemos escrever nosso teste de Exceção Esperada:

```
public void testFileSystemError() {  
    File f= new FullFile("foo");  
    try {  
        saveAs(f);  
        fail();  
    } catch (IOException e) {  
    }  
}
```

Um Modelo de Teste de Acidente é como um Objeto Simulado, exceto por não precisar simular todo o objeto. Classes internas anônimas de Java funcionam bem para sabotar apenas o método certo para simular o erro que queremos exercitar. Você pode sobreescriver apenas aquele método que quiser, bem ali no seu caso de teste, fazendo o caso de teste mais fácil de ler:

```
public void testFileSystemError() {  
    File f= new File("foo") {  
        public boolean createNewFile() throws IOException {  
            throw new IOException();  
        }  
    };  
    try {  
        saveAs(f);  
        fail();  
    } catch (IOException e) {  
    }  
}
```

---

## Teste quebrado (Broken Test)

Como você deixa uma sessão de programação quando está programando sozinho? Deixe o último teste quebrado.

Richard Gabriel me ensinou esse truque de terminar uma sessão de escrita no meio da frase. Quando você se senta de novo, olha a meia frase e tem que descobrir o que estava pensando quando a escreveu. Uma vez que tenha a linha de pensamento de volta, termina a frase e continua. Sem o estímulo de terminar a frase, você pode gastar muitos minutos farejando o que trabalhar em seguida, tentando lembrar seu estado mental e então, finalmente, voltando a digitar.

Eu tentei a técnica análoga para meus projetos solo, e realmente gosto do efeito. Terminar uma sessão solo escrevendo um caso de teste e o rodando

para ter certeza de que não passe. Quando você volta ao código, tem então um lugar óbvio para começar. Você tem uma marca óbvia e concreta para ajudar a lembrá-lo do que estava pensando; e fazer com que o teste funcione deveria ser trabalho rápido, assim você colocará rapidamente seus pés de volta no caminho da vitória.

Eu pensei que me incomodaria ter um teste quebrado durante a noite. Isso não me incomoda, talvez por que saiba que o programa não está terminado. Um teste que não funciona não faz do programa menos terminado, ele apenas faz o estado do programa se manifestar. A habilidade de pegar uma linha de desenvolvimento rapidamente depois de semanas de intervalo é digna daquele pequeno remorso de se afastar de uma barra vermelha.

---

## Check-in limpo (Clean Check-in)

Como você deixa uma sessão de programação quando está programando em um time? Deixe todos os testes rodando.

Eu me contradigo? Difícil.

—Bubba Whitman, irmão estivador de Walt (Whitman)

Quando você é responsável por sua equipe, o cenário muda completamente. Quando você começa a programar em um projeto em equipe, não sabe em detalhes o que aconteceu no código desde a última vez que o viu. Você precisa começar de um lugar de confiança e certeza. Portanto, certifique-se sempre de que todos os testes estão rodando antes de dar entrada no seu código. (É um pouco como cada caso de teste deixar o mundo em um bom estado conhecido, se você é inclinado a metáforas de computador para comportamentos humanos, o que não sou, geralmente.)

A suíte de testes que você roda quando faz check-in pode ser mais extensa que aquela que roda a cada minuto durante o desenvolvimento. (Não desista de rodar a suíte inteira sempre até ser lento o suficiente para ser irritante.) Você ocasionalmente encontrará um teste que não funciona no conjunto de integração quando tenta fazer check-in. O que fazer?

A regra mais simples é apenas jogar fora seu trabalho e começar de novo. O teste que não funciona é uma evidência muito forte de que você não sabia o suficiente para programar o que acabou de programar. Se o time adotasse essa regra, então haveria uma tendência de as pessoas fazerem check-in mais frequentemente, pois a primeira pessoa a fazer check-in não se arriscaria a perder trabalho. Fazer check-in mais frequentemente é provavelmente uma coisa boa.

Uma abordagem ligeiramente mais libertina dá a você a chance de corrigir o defeito e tentar novamente. Para evitar tomar conta dos recursos de integração, você provavelmente deveria desistir depois de alguns minutos e começar de novo. Nem preciso dizer, mas vou dizer mesmo assim, que comentar testes para fazer a suíte passar é proibido e justifica a multa de ter que pagar algumas cervejas no final da tarde de sexta-feira depois da reunião de planejamento.

# CAPÍTULO 28

---

## Padrões de Barra Verde

Depois de ter um teste não funcionando, você precisa arrumá-lo. Se você trata uma barra vermelha como uma condição a ser corrigida o mais rápido possível, então descobrirá que pode ter o verde rapidamente. Use esses padrões para fazer o código funcionar (mesmo se o resultado não é algo com o qual queira conviver por mais de uma hora).

---

### Fazer de conta (até fazê-lo) – Fake It (Til You Make It)

Qual é sua primeira implementação uma vez que tem um teste que não funciona? Retorne uma constante. Depois de ter o teste rodando, gradualmente transforme a constante em uma expressão usando variáveis.

Um exemplo simples aconteceu em nossa implementação de xUnit:

```
return "1 run, 0 failed"
```

tornou-se:

```
return "%d run, 0 failed" % self.runCount
```

tornou-se:

```
return "%d run, %d failed" % (self.runCount, self.failureCount)
```

Fazer de conta é um pouco como cravar uma estaca\* acima de sua cabeça quando está escalando uma rocha. Você realmente não chegou lá ainda (o teste

\* N. de R. T.: O termo correto em montanhismo é píton, mas estaca parece ser um conceito mais fácil de entender.

está lá, mas a estrutura do código está errada). Mas, quando você chegar lá, você sabe que estará seguro (o teste ainda vai rodar).

Fazer de conta realmente arrasta algumas pessoas para o caminho errado. Por que você faria algo que sabe que terá que arrancar? Porque ter algo rodando é melhor que não ter nada rodando, especialmente se você tem o teste para provar isto. Peter Hansen enviou essa história:

Algo aconteceu ontem quando, como dois novatos em TDD, meu parceiro e eu agressivamente seguimos a lei à risca e cometemos pecados para termos um teste funcionando rapidamente. No processo, percebemos que não tínhamos implementado adequadamente o teste, voltamos atrás e o corrigimos, e então fizemos o código funcionar de novo. O primeiro código que funcionou acabou não sendo visto durante o tempo em que trabalhamos para que funcionasse e nós escolhemos olhar um para o outro e dissemos “ããã . . . você olhou aquilo!”, pois aquela abordagem nos ensinou algo que não sabíamos.

Como uma implementação faz de conta os teria ensinado que seu teste estava escrito errado? Eu não sei, mas aposto que ficariam felizes em não investir na solução real para descobrir.

Há um monte de efeitos que tornam o faz de conta poderoso.

- Psicológico – Ter uma barra verde nos faz sentir completamente diferentes do que ter uma barra vermelha. Quando a barra está verde, você sabe onde está. Você pode refatorar dali com confiança.
- Controle de escopo – Programadores são bons em imaginar todo o tipo de problemas futuros. Começando com um exemplo concreto e generalizando daí, você se previne de se confundir prematuramente com preocupações alheias. Você pode fazer um trabalho melhor resolvendo o problema imediato, pois está focado. Quando vai implementar o próximo caso de teste, você pode se focar também naquele, sabendo que o teste anterior está garantidamente funcionando.

Faz de conta viola a regra que diz para você não escrever qualquer código que não precisa? Eu não acho, porque no passo de refatoração você está eliminando duplicação de dados entre o caso de teste e o código. Quando eu escrevo:

#### MyDate

```
public MyDate yesterday() {  
    return new MyDate("28.2.02");  
}
```

há duplicação entre o teste e o código. Posso contorná-la escrevendo:

#### MyDate

```
public MyDate yesterday() {  
    return new MyDate(new MyDate("31.3.02").days()-1);  
}
```

Mas ainda há duplicação. Contudo, posso eliminar a duplicação de dados (porque `this` é igual a `MyDate("31.1.02")` para os propósitos do meu teste) escrevendo:

#### **MyDate**

```
public MyDate yesterday() {
    return new MyDate(this.days()-1);
}
```

Nem todo mundo é convencido por esse monte de sofismas, motivo pelo qual você pode usar Triangulação ao menos até se cansar e começar a usar Fazer de conta ou mesmo Implementação Óbvia.

Quando uso Fazer de conta, lembro de longas viagens de carro com os filhos no banco de trás. Escrevo o teste, o faço funcionar de algum jeito feio, e então: “Não me faça parar esse carro e escrever outro teste. Se eu tiver que encostar, você vai se arrepender.”

“Certo, certo, pai. Eu vou limpar o código. Não tem que ficar tão zangado.”

## Triangular (Triangulate)

Como você conduz abstração com testes de forma mais conservadora? Abstraia apenas quando tiver dois ou mais exemplos.

Aqui está uma situação. Suponha que queiramos escrever uma função que retornará a soma de dois inteiros. Escrevemos:

```
public void testSum() {
    assertEquals(4, plus(3, 1));
}

private int plus(int augend, int addend) {
    return 4;
}
```

Se estamos triangulando para o projeto certo, temos de escrever:

```
public void testSum() {
    assertEquals(4, plus(3, 1));
    assertEquals(7, plus(3, 4));
}
```

Quando temos o segundo exemplo, podemos abstrair a implementação de `plus()`:

```
private int plus(int augend, int addend) {
    return augend + addend;
}
```

Triangulação é atrativa, pois as regras para ela parecem tão claras. As regras para Fazer de conta, o qual se baseia em nosso senso de duplicação entre os casos de teste e a implementação faz de conta para guiar a abstração, parecem um pouco vagas e sujeitas à interpretação. Embora pareçam simples, as regras para Triangulação criam um laço infinito. Uma vez que tenhamos as duas asserções e tenhamos abstraído a implementação correta para `plus`, podemos apagar uma das asserções em razão de ser completamente redundante com a outra. Se fizermos isso, todavia, podemos simplificar a implementação de `plus()` para retornar apenas uma constante que requer que adicionemos uma asserção.

Eu uso Triangulação apenas quando estou realmente muito inseguro sobre a abstração correta para o cálculo. Caso contrário, confio na Implementação Óbvia ou Fazer de conta.

---

## Implementação óbvia (Obvious Implementation)

Como você implementa operações simples? Apenas implemente-as.

Fazer de conta e Triangulação são passinhos pequeninos. Às vezes, você tem certeza que sabe como implementar uma operação. Vá em frente. Por exemplo, eu realmente usaria Faz de conta para implementar algo simples como `plus()`? Geralmente não. Eu apenas digitaria a Implementação Óbvia. Se percebesse que fui surpreendido por barras vermelhas, então iria em passos menores.

Não há virtude especial na natureza intermediária de Faz de conta ou Triangulação. Se você sabe o que digitar e pode fazê-lo rapidamente, então faça. Contudo, usando apenas Implementação Óbvia, você está exigindo perfeição de si. Psicologicamente, esse pode ser um movimento devastador. E se o que você escreve não é realmente a mais simples mudança que faria o teste funcionar? E se seu parceiro mostrar a você um mais simples? Você é um fracassado. Seu mundo desaba a sua volta! Você morre. Você congela.

Resolver “código limpo” ao mesmo tempo que resolve “que funciona” pode ser demais para fazer de uma só vez. Logo que possível, volte a resolver “que funciona”, e então resolva “código limpo” no tempo livre.

Mantenha o rastro de quão frequente você é surpreendido por barras vermelhas usando Implementação Óbvia. Eu ficarei preso nesses ciclos onde digitarei uma Implementação Óbvia, mas não vai funcionar. Mas, agora tenho certeza que sei o que devia digitar, então eu digito aquilo. Não funciona. Então agora. . . . Isso acontece especialmente com erros falta-ou-sobra-um e erros de positivo/negativo.

Você quer manter o ritmo vermelho/verde/refatore. Implementação Óbvia é uma segunda marcha. Esteja preparado para reduzir a marcha se seu cérebro começar a preencher cheques que seus dedos não podem descontar.

## Um para muitos (One to Many)

Como você implementa uma operação que funciona com coleções de objetos? Implemente-a sem as coleções primeiro, então a faça funcionar com coleções.

Por exemplo, suponha que estejamos escrevendo uma função para somar um vetor de números. Podemos começar com um:

```
public void testSum() {
    assertEquals(5, sum(5));
}

private int sum(int value) {
    return value;
}
```

(Estou implementando `sum()` na classe `TestCase` para evitar escrever uma nova classe apenas para um método.)

Queremos testar `sum(new int[] {5, 7})` em seguida. Primeiro, adicionamos um parâmetro para `sum()`, pegando um vetor de valores:

```
public void testSum() {
    assertEquals(5, sum(5, new int[] {5}));
}

private int sum(int value, int[] values) {
    return value;
}
```

Podemos olhar para esse passo como um exemplo de Isolar a Mudança\*. Uma vez que adicionemos o parâmetro no caso de teste, estamos livres para mudar a implementação sem afetar o caso de teste.

Agora podemos usar a coleção em vez de um só valor:

```
private int sum(int value, int[] values) {
    int sum= 0;
    for (int i= 0; i<values.length; i++)
        sum += values[i];
    return sum;
}
```

Em seguida, podemos apagar o parâmetro simples não usado:

```
public void testSum() {
    assertEquals(5, sum(new int[] {5}));
}
```

---

\* N. de R. T.: Isolar a Mudança (Isolate Change) é uma refatoração descrita no Capítulo 31.

```
private int sum(int[] values) {  
    int sum= 0;  
    for (int i= 0; i<values.length; i++)  
        sum += values[i];  
    return sum;  
}
```

O passo anterior também é um exemplo de Isolar a Mudança, onde mudamos o código para que possamos mudar os casos de teste sem afetar o código. Agora podemos enriquecer o caso de teste como planejado:

```
public void testSum() {  
    assertEquals(12, sum(new int[] {5, 7}));  
}
```

# CAPÍTULO 29

---

## Padrões xUnit

Esses são padrões para utilização de um membro da família xUnit de frameworks de teste.

---

### Asserção (Assertion)

Como você verifica que os testes funcionaram corretamente? Escreva expressões booleanas que automatizem seu julgamento sobre se o código funcionou ou não.

Se vamos fazer os testes totalmente automatizados, então cada pedaço de julgamento humano tem de ser extraído da avaliação de resultados. Precisamos pressionar um botão e fazer todas as decisões necessárias para verificar o correto funcionamento do código que o computador roda. Isso sugere o seguinte.

- As decisões têm de ser booleanas – Verdadeiro geralmente significa que tudo está certo, e falso significa que algo inesperado aconteceu.
- O estado dos booleanos tem de ser verificado pelo computador chamando alguma variante de um método assert().

Eu vi asserções como `assertTrue(rectangle.area() != 0)`. Você poderia retornar qualquer coisa não nula e satisfazer esse teste, logo isso não é muito útil. Seja específico. Se a área deveria ser 50, então diga que ela deve ser 50: `assertTrue(rectangle.area() == 50)`. Muitas implementações de xUnit têm uma asserção especial para teste de igualdade. Testar igualdade é comum, e, se você sabe que está testando igualdade, pode escrever uma mensagem de erro instrutiva. O valor esperado geralmente vem primeiro, assim, em JUnit, escreveríamos isso como `assertEquals(50, rectangle.area())`.

Pensar em objetos como caixas pretas é difícil. Se eu tenho um `Contract` com um `Status` que pode ser uma instância de `Offered` ou `Running`, poderia me sentir como se estivesse escrevendo um teste baseado na minha implementação esperada:

```
Contract contract= new Contract(); // Estado Offered por padrão  
contract.begin(); // Muda o estado para Running  
assertEquals(Running.class, contract.status.class);
```

Esse teste é muito dependente da implementação atual de `status`. O teste deveria passar mesmo se a representação de `status` mudasse para um booleano. Talvez uma vez que `status` mude para `Running`, seja possível pedirmos a data de início real.

```
assertEquals(. . ., contract.startDate()); // Lança uma exceção se o estado é Offered
```

Estou ciente de que estou nadando contra a maré ao insistir que todos os testes devam ser escritos usando apenas o protocolo público. Há até um pacote que estende JUnit, chamado JXUnit, que permite testar o valor de variáveis, mesmo daquelas declaradas privadas.

Desejar teste caixa branca não é um problema de teste, é um problema de projeto. Sempre que eu quiser usar uma variável como uma forma de verificar se o código roda corretamente ou não, eu tenho uma oportunidade de melhorar o projeto. Se eu me entregar a meu medo e apenas verificar a variável, então eu perco a oportunidade. Dito isso, se a ideia de projeto não vier, ela não vem. Eu verificarei a variável, derramarei uma lágrima, farei uma nota para voltar aqui em um dos meus dias mais espertos, e seguirei em frente.

O SUnit original (a primeira versão do framework de teste em Smalltalk) tinha asserções simples. Se uma delas falha, então um depurador surgiu, você arrumava o código e ia embora. Devido às IDEs de Java não serem tão sofisticadas, e devido à construção de software baseado em Java frequentemente acontecer em um ambiente batch, faz sentido adicionar informação da asserção que será exibida se ela falhar.

Em JUnit, isso toma a forma de um primeiro parâmetro opcional<sup>1</sup>. Se você escrever `assertTrue("Deveria ser verdadeiro", false)` quando o teste é executado, verá uma mensagem de erro como “Asserção falhou: Deveria ser verdadeiro”. Isso é frequentemente informação suficiente para enviá-lo diretamente à fonte do erro no código. Alguns times adotam a convenção de todas as asserções deverem ser acompanhadas de uma mensagem de erro instrutiva. Tente isso de ambas as formas e veja se o investimento nas mensagens de erro compensa.

---

## Fixture

Como você cria objetos comuns necessários para vários testes? Converta as variáveis locais nos testes em variáveis de instância. Sobrescreva `setUp()` e inicialize aquelas variáveis.

---

<sup>1</sup> Parâmetros opcionais viriam no fim, mas, por legibilidade dos testes, ajuda ter a string explicativa no início.

Se quisermos remover duplicação de nosso código modelo, queremos removê-la do nosso código de teste também? Talvez.

Aqui está o problema: frequentemente você escreve mais código para configurar objetos em um estado interessante do que escreve para manipulá-los e verificar resultados. O código para configurar os objetos é o mesmo para muitos testes (aqueles objetos que são as fixtures do teste, também conhecidos como scaffolding). Essa duplicação é ruim pelas seguintes razões.

- Levam muito tempo para escrever, mesmo para copiar e colar, e gostaríamos de escrever testes rápido.
- Se precisamos mudar uma interface na mão, então temos que mudá-la em muitos testes (exatamente o que esperaríamos de duplicação).

Contudo, a mesma duplicação também é boa. Testes escritos com o código de configuração bem ali, junto com as asserções, são legíveis de cima a baixo. Se fatorarmos o código de configuração em um método separado, então teríamos de lembrar qual método foi chamado e lembrar quais objetos eram usados antes de podermos escrever o resto do teste.

xUnit dá suporte a ambos os estilos de escrita de teste. Você pode escrever o código de criação-de-base-de-teste com o teste, se espera que os leitores não sejam capazes de lembrar os objetos base facilmente. Todavia, pode também mover código comum de criação-de-base-de-teste em um método chamado `setUp()`. Nele, configure variáveis de instância para os objetos que serão usados no teste.

Aqui está um exemplo muito simples para motivar o valor de refatorar código comum de configuração, mas curto o bastante para caber neste livro. Poderíamos escrever:

```
EmptyRectangleTest
public void testEmpty() {
    Rectangle empty= new Rectangle(0,0,0,0);
    assertTrue(empty.isEmpty());
}

public void testWidth() {
    Rectangle empty= new Rectangle(0,0,0,0);
    assertEquals(0.0, empty.getWidth(), 0.0);
}
```

(Isso também demonstra a versão em ponto flutuante de `assertEquals()`, a qual requer uma certa tolerância.) Poderíamos nos livrar da duplicação escrevendo:

```
EmptyRectangleTest
private Rectangle empty;

public void setUp() {
    empty= new Rectangle(0,0,0,0);
}
```

```
public void testEmpty() {  
    assertTrue(empty.isEmpty());  
}  
  
public void testWidth() {  
    assertEquals(0.0, empty.getWidth(), 0.0);  
}
```

Extraímos o código comum como um método, um que o framework garante chamar antes de nosso método de teste ser chamado. Os métodos de teste são mais simples, mas temos que lembrar o que está em `setUp()` antes de podermos entendê-los.

Qual estilo você deveria usar? Tente ambos. Eu quase sempre fatoro código de configuração comum, mas tenho uma memória forte para detalhes. Os leitores dos meus testes às vezes reclamam que há demais para lembrar, dessa forma, talvez eu devesse fatorar menos.

O relacionamento de subclasses de `TestCase` e de instâncias daquelas subclasses é uma das mais confusas partes de xUnit. Cada novo tipo de fixture deveria ser uma nova subclass de `TestCase`. Cada nova fixture é criada em uma instância daquela subclass, usada uma vez, e então descartada.

No exemplo anterior, se quiséssemos escrever testes para um `Rectangle` não vazio, então criariamos uma nova classe, talvez `NormalRectangleTest`, e inicializariamos uma variável diferente para um retângulo diferente em `setUp()`. Em geral, se eu me vejo querendo uma fixture ligeiramente diferente, então começo uma nova subclass de `TestCase`.

Isso implica que não há relacionamento simples entre classes de teste e classes de modelo. Às vezes, uma fixture serve para testar várias classes (embora isso seja raro). Às vezes, duas ou três fixtures são necessárias para uma só classe do modelo. Na prática, você geralmente vai terminar com aproximadamente o mesmo número de classes de teste e de classes de modelo, mas não porque para cada classe de modelo você escreva uma, e apenas uma, classe de teste.

---

## Fixture externa

Como você libera recursos externos na fixture? Sobrescreva `tearDown()` e libere os recursos.

Lembre que a meta de cada teste é deixar o mundo exatamente no mesmo estado de antes de rodar. Por exemplo, se você abriu um arquivo durante o teste, você precisa certificar-se de fechá-lo antes do teste terminar. Você poderia escrever:

```
testMethod(self):  
    file= File("foobar").open()
```

```

try:
    ...run the test...
finally:
    file.close()

```

Se o arquivo foi usado em muitos testes, então poderia fazê-lo parte da fixture comum:

```

setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    try:
        ...run the test...
    finally:
        self.file.close()

```

Primeiro, há aquela maldita duplicação da cláusula `finally` nos dizendo que estamos esquecendo algo no projeto. Segundo, esse método está propenso a erros, pois é fácil esquecer a cláusula `finally` ou esquecer completamente de fechar o arquivo. Finalmente, há três linhas de ruído no teste – `try`, `finally` e o próprio `close`, que não são centrais para a execução do teste.

xUnit garante que um método chamado `tearDown()` será executado depois do método de teste. `TearDown()` será chamado a despeito do que aconteça no método de teste (apesar de se `setUp()` falhar, `tearDown()` não será chamado). Podemos transformar o teste anterior em:

```

setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    ...run the test...
tearDown(self):
    self.file.close()

```

## Método de teste (Test Method)

Como você representa um único caso de teste? Como um método cujo nome começa com “teste”.

Você vai ter centenas, até milhares, de testes em seu sistema. Como vai manter o rastro de todos eles?

Linguagens de programação orientada a objetos têm três níveis de hierarquia para organização:

- Módulo (“pacote” em Java)
- Classe
- Método

Se estamos escrevendo testes como fonte comum de código, então precisamos achar um jeito de ajustar isso nessa estrutura. Se estamos usando classes para representar fixtures, então o habitat para testes são os métodos. Todos os testes que dividem uma única fixture serão métodos na mesma classe. Testes que requerem uma fixture diferente estarão em uma classe diferente.

Por convenção, o nome do método começa com “test”. Ferramentas podem procurar esse padrão para automaticamente criar conjuntos de testes dada uma classe. O restante do nome do método deve sugerir a um futuro leitor desinformado por que esse teste foi escrito. JUnit, por exemplo, tem um teste chamado “`testAssertPosInfinityNotEqualsNegInfinity`”. Eu não lembro de escrever esse teste, mas, pelo nome, assumo que em algum ponto o código de asserção do JUnit para números em ponto flutuante não distingua entre infinito positivo e negativo. (É meio feio – há um condicional especial para manipular infinito.)

Métodos de teste deveriam ser fáceis de ler e ser código muito direto. Se um método de teste está ficando longo e complicado, então você precisa jogar “Passos de Bebê”. O objetivo desse jogo é escrever o menor método de teste que representa progresso real em direção a seu objetivo final. Três linhas parecem ser o mínimo sem discutir ofuscação (e lembre, você está escrevendo esses testes para pessoas, não apenas para o computador ou para si).

Patrick Logan contribuiu com uma ideia que vou experimentar, também descrita por McConnell<sup>2</sup> e Caine e Gordon<sup>3</sup>,

Por alguma razão, eu estive trabalhando com “esboços” em praticamente tudo que faço ultimamente. Teste não é diferente. Quando escrevo testes, eu primeiro crio um esboço do teste que eu quero escrever, por exemplo...

```
/* Adicionar a espaços da tupla. */  
/* Pegar dos espaços da tupla. */  
/* Ler dos espaços da tupla. */
```

Esses são lacunas indicadas até eu adicionar testes específicos sob cada categoria. Quando adiciono testes, adiciono outro nível de comentários ao esboço... .

```
/* Adicionar a espaços da tupla. */  
/* Pegar dos espaços da tupla. */  
/** Pegar uma tupla não existente. **/  
/** Pegar uma tupla existente. **/  
/** pegar múltiplas tuplas. **/  
/* Ler dos espaços da tupla. */
```

Eu geralmente tenho apenas dois ou três níveis no esboço. Não consigo pensar quando tenho mais. Mas, o esboço essencialmente se torna documentação do contrato para a classe sendo testada. Os exemplos aqui estão abreviados, mas eles

---

<sup>2</sup> McConnell, Steve. 1993. *Code Complete*, chapter 4. Seattle, Washington: Microsoft Press. ISBN 1556154844.

<sup>3</sup> Caine, S.H. & Gordon, E. K. 1975. “PDL: A Toll for Software Design”, *AFIPS Proceedings of the 1975 National Computer Conference*.

seriam mais específicos em linguagem contratual. (Eu não uso qualquer tipo de suplemento para Java para automação parecida com Eiffel.)

Imediatamente abaixo do mais baixo nível do esboço está o código do caso de teste.

---

## Teste de exceção (Exception Test)

Como você testa exceções inesperadas? Pegue exceções esperadas e ignore-as, falhando apenas se a exceção não é lançada.

Vamos dizer que estamos escrevendo algum código para buscar um valor. Se o valor não é encontrado, então queremos lançar uma exceção. Testar a busca é muito fácil:

```
public void testRate() {  
    exchange.addRate("USD", "GBP", 2);  
    int rate= exchange.findRate("USD", "GBP");  
    assertEquals(2, rate);  
}
```

Testar a exceção pode não ser tão óbvio. Aqui está como fazemos:

```
public void testMissingRate() {  
    try {  
        exchange.findRate("USD", "GBP");  
        fail();  
    } catch (IllegalArgumentException expected) {  
    }  
}
```

Se `findRate()` não lança uma exceção, chamaremos `fail()`, um método xUnit que notifica que o teste falhou. Perceba que somos cuidadosos apenas para pegar a exceção particular que esperamos, então seremos também notificados se o tipo errado de exceção é lançado (incluindo falhas de asserção).

---

## Todos os testes (All Tests)

Como rodar todos os testes juntos? Faça uma suíte de todas as suítes – uma para cada pacote, e uma agregando os pacotes de teste da aplicação inteira.

Suponha que você adicione uma subclasse `TestCase` a um pacote e você adiciona um método de teste a esta classe. A próxima vez que todos os testes rodarem, esse método de teste deveria rodar também. (Há aquela coisa guiada por testes – o anterior é o esboço para um teste que eu provavelmente implementaria se não estivesse ocupado escrevendo um livro.)

Devido a isso não ser suportado na maioria das implementações ou IDEs xUnit, cada pacote deve declarar uma classe AllTests que implementa um método estático suite() que retorna um TestSuite. Aqui está AllTests para o exemplo financeiro:

```
public class AllTests {  
    public static void main(String[] args) {  
        junit.swingui.TestRunner.run(AllTests.class);  
    }  
  
    public static Test suite() {  
        TestSuite result= new TestSuite("TFD tests");  
        result.addTestSuite(MoneyTest.class);  
        result.addTestSuite(ExchangeTest.class);  
        result.addTestSuite(IdentityRateTest.class);  
        return result;  
    }  
}
```

Você pode também fornecer a AllTests um método main() para que a classe possa ser diretamente rodada da IDE ou por uma linha de comando.

# CAPÍTULO 30

---

## Padrões de Projeto

Uma das principais ideias de padrões é que, embora possa parecer como se estivéssemos todo o tempo resolvendo problemas completamente diferentes, a maioria dos problemas que resolvemos é gerada pelas ferramentas que usamos e não por problemas externos próximos<sup>1</sup>. Por causa disso, podemos esperar encontrar (e realmente vamos encontrar) problemas comuns com soluções comuns, mesmo em meio a uma incrível diversidade de contextos externos de solução de problemas.

Aplicar objetos para organizar a computação é um dos melhores exemplos de subproblemas comuns internamente gerados, sendo resolvidos de formas previsíveis e comuns. O enorme sucesso de padrões de projeto é um atestado das coisas em comum vistas por programadores na orientação a objetos. O sucesso do livro *Padrões de Projeto*<sup>2</sup>, contudo, asfixiou qualquer diversidade na expressão desses padrões. O livro parece ter um viés util em direção ao projeto como uma fase. Isso certamente não faz menção à refatoração como uma atividade de projeto. Projeto em TDD requer um olhar ligeiramente diferente sobre os padrões de projeto.

Os padrões de projeto tratados aqui não pretendem ser completos. Eles são descritos o suficiente apenas para nos guiarmos através dos exemplos. Aqui estão eles em resumo.

- Command – Representa a invocação de uma computação como um objeto e não apenas como uma mensagem.
- Value Object – Evita problemas de sinônimos construindo objetos cujos valores nunca mudam depois de criados.

---

<sup>1</sup> Alexander, Christofer 1970. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press. ISBN: 0674627512.

<sup>2</sup> Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley. ISBN 0201633612.

- Null Object – Representa o caso básico de uma computação por um objeto.
- Template Method – Representa sequências invariáveis de computação com um método abstrato que precisa ser especializado através de herança.
- Pluggable Object – Representa variação invocando outro objeto com duas ou mais implementações.
- Pluggable Selector – Evita subclasses gratuitas invocando dinamicamente métodos diferentes para instâncias diferentes.
- Factory Method – Cria um objeto chamando um método em vez de um construtor.
- Imposter – Introduz variação introduzindo uma nova implementação de um protocolo existente.
- Composite – Representa a composição do comportamento de uma lista de objetos como um só objeto.
- Collecting Parameter – Passa um parâmetro para ser usado para agregar os resultados de uma computação em muitos objetos diferentes.

Os padrões de projeto são agrupados com base em como são usados em TDD, conforme mostrado na Tabela 30.1.

**Tabela 30.1** *Uso de padrões de projeto no desenvolvimento dirigido por testes*

Padrão	Escrita de teste	Refatoração
Command	X	
Value Object	X	
Null Object		X
Template Method		X
Pluggable Object		X
Pluggable Selector		X
Factory Method	X	X
Imposter	X	X
Composite	X	X
Collecting Parameter	X	X

---

## Command

O que você faz quando precisa que a invocação de uma computação seja mais complicada do que uma simples chamada de método? Faça um objeto para a computação e invoque-o.

Enviar mensagens é maravilhoso. Linguagens de programação tornam o envio de mensagens sintaticamente fácil; e ambientes de programação tornam a manipulação de mensagens fácil (por exemplo, refatorações para renomear uma mensagem automaticamente). Contudo, às vezes, apenas enviar uma mensagem não é o suficiente.

Por exemplo, suponha que queiramos registrar o fato de uma mensagem ser enviada. Poderíamos adicionar características à linguagem (métodos envelope – *wrappers*) para fazermos isso, mas o registro é raro o suficiente e o valor de linguagens simples é alto demais, de forma a preferirmos não fazer isso. Ou, suponha que quiséssemos invocar uma computação, mas mais tarde. Poderíamos começar o processo, imediatamente suspendê-lo e recomeçá-lo depois, mas, então, teríamos todas as alegrias de concorrência para lidar.

Invocações complicadas de computações requerem mecanismos caros. Mas, na maioria do tempo, não precisamos toda a complexidade e preferimos não pagar o custo. Quando precisamos que a invocação seja só um pouquinho mais concreta e manipulável que uma mensagem, objetos nos trazem a resposta. Faça um objeto representar a invocação. Semeio-o com todos os parâmetros que a computação irá precisar. Quando estivermos preparados para invocá-lo, use um protocolo genérico como `run()`.

A interface `Runnable` de Java é um excelente exemplo disso:

### Runnable

```
interface Runnable  
    public abstract void run();
```

Na implementação de `run()`, você pode fazer o que quiser. Infelizmente, Java não tem uma forma sintaticamente leve de criar e invocar `Runnables`, então eles não são usados tanto quanto o equivalente em outras linguagens – blocos ou lambda em Smalltalk/Ruby ou LISP.

---

## Value object

Como você projeta objetos que serão amplamente compartilhados, mas para os quais a identidade não é importante? Configure seu estado quando são criados e nunca mude-o. Operações no objeto sempre retornam um novo objeto.

Objetos são maravilhosos. Posso dizer isso aqui, não posso? Objetos são uma grande forma de organizar lógica para posterior entendimento e extensão. Contudo, há um pequeno problema (certo, mais que um, mas esse aqui vai bastar por agora).

Suponha que eu (um objeto) tenha um Retângulo. Eu computo alguns valores baseado no Retângulo, como sua área. Depois, alguém educadamente pede o meu Retângulo, e eu, não querendo parecer não cooperativo, dou a ele. Momentos depois, vejam só, o Retângulo foi alterado pelas minhas costas. A área que computei antes está desatualizada e não há jeito de saber.

Esse é um clássico problema de sinônimos. Se dois objetos dividem uma referência com um terceiro, e se um objeto muda o objeto compartilhado, então é melhor que os outros objetos não contem com o estado do objeto compartilhado.

Há muitas maneiras de escapar do problema de sinônimos. Uma solução é nunca fornecer os objetos dos quais você depende, mas, em vez disso, sempre fazer cópias. Isso pode parecer caro em tempo e espaço e ignora aquelas vezes em que você quer dividir mudanças de um objeto compartilhado. Outra solução é Observer, onde você explicitamente registra objetos dos quais depende e espera ser notificado quando eles mudarem. Observer pode fazer o controle de fluxo difícil de seguir e a lógica de configuração e remoção de dependências ficar feia.

Outra solução é tratar o objeto como menos que um objeto. Objetos tem um estado que muda no decorrer do tempo. Podemos, se escolhermos, eliminar o “que muda no decorrer do tempo”. Se eu tiver um objeto e souber que ele não vai mudar, então posso passar suas referências por todo o lugar que quiser sabendo que sinônimos não serão um problema. Não podem haver mudanças escondidas em um objeto compartilhado se não há mudanças.

Eu lembro da confusão sobre inteiros quando estava aprendendo Smalltalk pela primeira vez. Se eu mudo o bit 2 para 1, por que todos os 2 não se tornam 6?

```
a := 2.  
b := a.  
a := a bitAt: 2 put: 1.  
a => 6  
b => 2
```

Inteiros são realmente valores mascarados como objetos. Em Smalltalk isso é literalmente verdadeiro para inteiros pequenos, e simulado no caso de inteiros que não cabem em uma única *word* da máquina. Quando eu marco aquele bit, o que é trazido de volta é um novo objeto com o bit marcado e não o antigo com o bit trocado.

Quando implementar um Value Object, cada operação tem que retornar um objeto fresco, deixando o original inalterado. Usuários têm que estar cientes de que estão usando um Value Object e têm que armazenar o resultado (como no exemplo precedente com a, b). Todas essas alocações de objetos podem criar problemas de desempenho, os quais devem ser tratados como todos os problemas de desempenho, quando você usa conjuntos de dados reais, padrões de uso real, dados de perfis e queixas sobre desempenho.

Eu tenho a tendência a usar Value Object sempre que tenho uma situação que parece como álgebra – com intersecção e união de figuras geométricas, valores de unidades em que unidades são transportadas com um número, aritmética simbólica. Sempre que Value Object faz o mínimo sentido, eu tento, pois isso torna a leitura e a depuração muito mais fáceis.

Todos os Value Objects têm de implementar igualdade (e em muitas linguagens, por implicação, têm de implementar hashing). Se eu tenho esse contrato e aquele contrato e eles não são o mesmo objeto, então são diferentes, e não iguais. Contudo, se tenho esses cinco francos e aqueles cinco francos, não importa se são os mesmos cinco francos; cinco francos são cinco francos, e eles deveriam ser iguais.

## Null object

Como você representa casos especiais usando objetos? Crie um objeto que representa o caso especial. Dê a ele o mesmo protocolo dos objetos regulares.

**inspirado por java.io.File**

```
public boolean setReadOnly() {
    SecurityManager guard = System.getSecurityManager();
    if (guard != null) {
        guard.canWrite(path);
    }
    return fileSystem.setReadOnly(this);
}
```

Há 18 verificações para `guard != null` em `java.io.File`. Embora eu aprecie o empenho deles em fazer arquivos seguros para o mundo, também fico um pouco nervoso. Eles são cuidadosos a ponto de sempre verificarem um nulo como resultado de `getSecurityManager()`?

A alternativa é criar uma nova classe, `LaxSecurity`, que nunca lança exceções:

**LaxSecurity**

```
public void canWrite(String path) { }
```

Se alguém pede um `SecurityManager` e não há nenhum disponível, então, em vez disso, enviamos de volta um `LaxSecurity`:

**SecurityManager**

```
public static SecurityManager getSecurityManager() {
    return security == null ? new LaxSecurity() : security;
}
```

Agora não temos que nos preocupar se alguém esquecer de verificar o nulo. O código original fica consideravelmente mais limpo:

#### Arquivo

```
public boolean setReadOnly() {  
    SecurityManager security = System.getSecurityManager();  
    security.canWrite(path);  
    return fileSystem.setReadOnly(this);  
}
```

Erich Gamma e eu uma vez tivemos uma discussão em um tutorial OOPSLA sobre se um Null Object era adequado em algum lugar do JHotDraw. Eu estava à frente por pontos quando Erich calculou o custo de introduzir o Null Object como dez linhas de código para as quais nós eliminariámos um condicional. Eu odeio esses nocautas técnicos no último round. (Nós também tivemos indicações ruins da audiência por não sermos organizados. Aparentemente eles não estavam cientes de que ter discussões produtivas de projeto é uma difícil, porém frutífera, habilidade.)

---

## Template method

Como você representa a sequência invariável de uma computação enquanto prepara refinamento futuro? Escreva um método que é implementado inteiramente em termos de outros métodos.

A programação está cheia de sequências clássicas:

- Entrada/processo/saída
- Enviar mensagem/receber mensagem
- Ler comando/retornar resultado

Gostaríamos de ser capazes de comunicar claramente a universalidade dessas sequências e, ao mesmo tempo, preparar variação na implementação dos passos.

Em herança, linguagens orientadas a objetos fornecem um simples, mesmo que limitado, mecanismo para comunicar sequências universais. Uma superclasse pode conter um método escrito inteiramente em termos de outros métodos, e subclasses podem implementar esses métodos de diferentes formas. Por exemplo, JUnit implementa a sequência básica de executar um teste como:

#### TestCase

```
public void runBare() throws Throwable {  
    setUp();  
    try {  
        runTest();  
    }
```

```

    }
    finally {
        tearDown();
    }
}

```

As subclasses podem implementar `setUp()`, `runTest()`, e `tearDown()` de qualquer forma que queiram.

Uma questão ao escrever um Template Method é saber escrever uma implementação padrão dos submétodos. Em `TestCase.runBare()`, todos os três submétodos têm implementações padrão.

- `setUp()` e `tearDown()` são métodos *nop*\*
- `runTest()` dinamicamente encontra e invoca um método de teste baseado no nome do caso de teste.

Se a computação não faz sentido sem um subpasso sendo preenchido, então anote isso da forma que sua linguagem de programação oferecer.

- Em Java, declare o submétodo abstrato.
- Em Smalltalk, implemente o método lançando um erro `SubclassResponsibility`.

Template method é melhor encontrado por meio de experiência, em vez de projetado de um jeito desde o começo. Sempre que eu digo a mim mesmo, “ah, essa é a sequência e aqui estão os detalhes”, me vejo otimizando os métodos depois e reextraindo as partes verdadeiramente variantes.

Quando você encontra duas variantes de uma sequência em duas subclasses, precisa movê-las gradualmente para mais perto. Uma vez que tenha extraído as partes diferentes de outros métodos, o que nos resta é o Template Method. Então você pode mover o Template Method para a superclasse e eliminar a duplicação.

## Pluggable object

Como você expressa variação? O jeito mais simples é com condicionais explícitos:

```

if (circle) then {
    . . . código para círculo. . .
} else {
    . . . código para o que não é círculo
}

```

\* N. de R. T.: Acrônimo de métodos *no-operation*, isto é, métodos que não executam nada, não alteram nada.

Você rapidamente vai achar que tal decisão explícita começa a se espalhar. Se você representa a distinção entre círculos e não círculos como um condicional explícito em um lugar, então é provável que o condicional se espalhe.

Devido ao segundo imperativo de TDD ser a eliminação de duplicação, você deve conter a praga de condicionais explícitos no embrião. Na segunda vez que ver um condicional, é hora de sacar o mais básico dos movimentos de projeto de orientação a objetos, o Pluggable Object.

Os Pluggable Objects revelam-se por simplesmente eliminar a duplicação que é algumas vezes contra intuitiva. Erich Gamma e eu encontramos isso, um dos meus exemplos favoritos de um Pluggable Object imprevisível. Ao escrever um editor gráfico, a seleção geralmente é um pouco complicada. Se você está sobre uma figura quando pressiona o botão do mouse, movimentos subsequentes do mouse movem aquela figura, e liberar o botão do mouse libera a figura selecionada. Se você não está sobre uma figura, então está selecionando um grupo de figuras, e movimentos subsequentes do mouse tipicamente redimensionam um retângulo usado para selecionar várias figuras. Liberar o botão do mouse seleciona as figuras dentro do retângulo. O código inicial parece algo como isso:

#### FerramentaDeSelecao

```
Figure selected;
public void mouseDown() {
    selected= findFigure();
    if (selected != null)
        select(selected);
}
public void mouseMove() {
    if (selected != null)
        move(selected);
    else
        moveSelectionRectangle();
}
public void mouseUp() {
    if (selected == null)
        selectAll();
}
```

Existe aquele condicional feio duplicado (eu disse para você que eles se espalham como uma doença). A resposta nesses casos é criar um Pluggable Object, um `SelectionMode`, com duas implementações: `SingleSelection` e `MultipleSelection`.

#### FerramentaDeSelecao

```
SelectionMode mode;
public void mouseDown() {
    selected= findFigure();
    if (selected != null)
        mode= SingleSelection(selected);
    else
        mode= MultipleSelection();
```

```

    }
    public void mouseMove() {
        mode.mouseMove();
    }
    public void mouseUp() {
        mode.mouseUp();
    }
}

```

Em linguagens com interfaces explícitas, você terá que implementar uma interface junto com os dois (ou mais) Pluggable Objects.

## Pluggable selector<sup>3</sup>

Como você invoca comportamentos diferentes de instâncias diferentes? Armazene o nome de um método e invoque o método dinamicamente.

O que você faz quando tem dez subclasses de uma classe, cada uma implementando apenas um método? Fazer subclasses é um mecanismo pesado para capturar tão pouca quantidade de variação.

```

abstract class Report {
    abstract void print();
}

class HTMLReport extends Report {
    void print() { ... }
}
class XMLReport extends Report {
    void print() { ... }
}

```

Uma alternativa é ter uma só classe com um comando switch. Dependendo do valor de um campo, você invoca métodos diferentes. Contudo, o nome do método aparece em três lugares:

- A criação da instância
- O estado do interruptor
- O método em si

<sup>3</sup> Para mais detalhes, ver Beck, K. *The Smalltalk Best Practices Patterns*, pp. 70-73, Englewood-Ciffs, NJ: Prentice-Hall. ISBN 013476904X. É ruim referenciar seu próprio trabalho, mas, como disse o filósofo Phyllis Diller uma vez: “Claro que eu rio de minhas próprias piadas. Você não pode confiar em estranhos.”

```
abstract class Report {  
    String printMessage;  
  
    Report(String printMessage) {  
        this.printMessage= printMessage;  
    }  
  
    void print() {  
        switch (printMessage) {  
            case "printHTML" :  
                printHTML();  
                break;  
            case "printXML" :  
                printXML();  
                break;  
        }  
    };  
  
    void printHTML() {  
    }  
  
    void printXML() {  
    }  
}
```

Cada vez que você adiciona um novo tipo de impressão, tem que se certificar de adicionar o método de impressão e mudar o comando switch.

A solução Pluggable Selector é para invocar dinamicamente o método usando reflexão:

```
void print() {  
    Method runMethod= getClass().getMethod(printMessage, null);  
    runMethod.invoke(this, new Class[0]);  
}
```

Agora ainda há uma dependência feia entre criadores de relatórios e os nomes dos métodos de impressão, mas, ao menos, você não tem a cláusula case lá também.

O Pluggable Selector pode, definitivamente, ser usado demais. O maior problema com ele é rastrear código para ver se o método é invocado. Use Pluggable Selector apenas quando está limpando uma situação razoavelmente direta na qual cada um dos ramos de subclasses tem apenas um método.

---

## Factory method

Como você cria um objeto quando quer flexibilidade na criação de novos objetos? Crie um objeto em um método em vez de usar um construtor.

Construtores são expressivos. Você pode ver que definitivamente está criando um objeto quando usa um. Contudo, construtores, particularmente em Java, têm uma falta de expressividade e flexibilidade.

Um eixo de flexibilidade que queríamos em nosso exemplo financeiro era sermos capazes de retornar um objeto de uma classe diferente quando criássemos um objeto. Temos testes como:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Queremos introduzir a classe `Money`, mas não podemos enquanto estivermos presos criando uma instância de `Dollar`. Introduzindo um nível de indireção através de um método, ganhamos a flexibilidade de retornar uma instância de uma classe diferente sem mudar o teste:

```
public void testMultiplication() {
    Dollar five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

### `Money`

```
static Dollar dollar(int amount) {
    return new Dollar(amount);
}
```

Esse método é chamado um *Factory Method*, pois fabrica objetos.

A desvantagem do uso do *Factory Method* é precisamente sua indireção. Você tem que lembrar que o método está realmente criando um objeto mesmo que não pareça um construtor. Use o *Factory Method* apenas quando você precisar da flexibilidade que ele cria. Caso contrário, construtores funcionam bem para a criação de objetos.

## Imposter

Como introduzir uma nova variação em uma computação? Introduza um novo objeto com o mesmo protocolo de um objeto existente, mas com uma implementação diferente.

Introduzir variação em um programa procedural envolve adicionar lógica condicional. Como vimos com *Pluggable Object*, tal lógica tende a se proliferar e uma dose saudável de mensagens polimórficas são necessárias para curar a duplicação.

Suponha que tenha uma estrutura já no lugar. Já existe um objeto. Agora você precisa que o sistema faça algo diferente. Se há um lugar óbvio para inserir

uma cláusula if e você não está duplicando lógica de nenhum outro lugar, então vá em frente. Frequentemente, entretanto, a variação obviamente vai precisar de mudanças em muitos métodos.

Esse momento de decisão surge de duas formas em TDD. Às vezes, você está escrevendo um caso de teste e precisa representar um novo cenário. Nenhum dos objetos existentes expressa o que você quer expressar. Suponha que estamos testando um editor gráfico e que já tenhamos retângulos desenhados corretamente:

```
testRectangle() {  
    Drawing d= new Drawing();  
    d.addFigure(new RectangleFigure(0, 10, 50, 100));  
    RecordingMedium brush= new RecordingMedium();  
    d.display(brush);  
    assertEquals("rectangle 0 10 50 100\n", brush.log());  
}
```

Agora queremos mostrar ovais. Nesse caso, o Imposter é fácil de perceber – substitua um `RectangleFigure` por um `OvalFigure`.

```
testOval() {  
    Drawing d= new Drawing();  
    d.addFigure(new OvalFigure(0, 10, 50, 100));  
    RecordingMedium brush= new RecordingMedium();  
    d.display(brush);  
    assertEquals("oval 0 10 50 100\n", brush.log());  
}
```

Geralmente, perceber a possibilidade de um Imposter pela primeira vez requer um *insight*. A ideia de Ward Cunningham de que um vetor de Moneys poderia agir como um Money é um momento único. Você pensava que eram diferentes, e agora consegue vê-los como a mesma coisa.

A seguir estão dois exemplos de Impostors que surgem durante a refatoração:

- Null Object – Você pode tratar a ausência de dados da mesma forma que a presença de dados.
- Composite – Você pode tratar a coleção de objetos da mesma forma que um só objeto.

O encontro de Impostors durante a refatoração é guiado pela eliminação de duplicação, assim como toda refatoração é guiada pela eliminação de duplicação.

---

## Composite

Como você implementa um objeto cujo comportamento é a composição do comportamento de uma lista de outros objetos? Torne-o um Imposter para objetos componentes.

Meu exemplo favorito é também um exemplo da contradição de Composites: Account e Transaction. Transactions armazenam um incremento de valor (elas são realmente muito mais complexas e interessantes, mas, por enquanto...):

### Transaction

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

Accounts calculam saldo somando os valores de suas Transactions:

### Account

```
Transaction transactions[];
Money balance() {
    Money sum= Money.zero();
    for (int i= 0; i < transactions.length; i++)
        sum= sum.plus(transactions[i].value);
    return sum;
}
```

Parece bastante simples.

- Transactions têm um valor.
- Accounts têm um saldo.

Então vem a parte interessante. Um cliente tem um monte de contas e gostaria de ver um saldo geral. O jeito óbvio de implementar isso é com uma nova classe, OverallAccount, a qual soma os saldos de uma lista de Accounts. Duplicação! Duplicação!

E se Account e Balance implementarem a mesma interface? Vamos chamá-la Holding, porque não consigo pensar em nada melhor no momento.

### Holding

```
interface Holding
    Money balance();
```

Transactions podem implementar balance() retornando seu valor:

### Transaction

```
Money balance() {
    return value;
}
```

Agora, Accounts pode ser composta de Holdings, não de Transactions:

### Account

```
Holding holdings[];
Money balance() {
```

```
Money sum= Money.zero();
for (int i= 0; i < holdings.length; i++)
    sum= sum.plus(holdings[i].balance());
return sum;
}
```

Agora, seu problema com OverallAccounts desaparece. Uma OverallAccount é apenas uma Account contendo Accounts.

O cheiro de Composite é ilustrado acima. Transações não têm saldos, não no mundo real. Aplicar Composite é um truque de programador geralmente não apreciado pelo resto do mundo. Entretanto, os benefícios ao projeto do programa são enormes, assim a desconexão contextual frequentemente vale a pena. Diretórios contêm Diretórios, SuítesDeTestes contêm SuítesDeTestes, Desenhos contêm Desenhos – nenhuma dessas traduz bem o mundo, mas todas elas fazem o código muito mais simples.

Eu tive que brincar com Composite por um longo tempo antes que encontrasse onde usá-lo e onde não usá-lo. Como é óbvio a partir dessa discussão, ainda não sou capaz de articular como adivinhar se uma coleção de objetos é apenas uma coleção de objetos ou se você realmente tem um Composite. A boa notícia é que, desde que você esteja ficando bom em refatorar, no momento em que a duplicação aparecer, você pode introduzir Composite e observar a complexidade do programa desaparecer.

---

## Collecting parameter

Como você coleta os resultados de uma operação que está espalhada em vários objetos? Adicione um parâmetro à operação em que os resultados serão coleados.

Um exemplo simples é a interface `java.io.Externalizable`. O método `writeExternal` escreve em um objeto e em todos os objetos que ele referencia. Como todos os objetos têm que cooperar livremente para a escrita, ao método é passado um parâmetro, um `ObjectOutput`, como parâmetro coletor (Collecting Parameter):

### `java.io.Externalizable`

```
public interface Externalizable extends java.io.Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
}
```

Adicionar um Collecting Parameter é uma consequência comum de Composite. No desenvolvimento de JUnit, não precisávamos que o `TestResult` colocasse os resultados de vários testes até que tivéssemos escrito vários testes.

À medida que a sofisticação dos resultados esperados cresce, você pode encontrar a necessidade de introduzir um Collecting Parameter. Por exemplo, suponha que estamos mostrando Expressions.

Se tudo que quisermos é uma string plana, então concatenação é suficiente:

```
testSumPrinting() {
    Sum sum= new Sum(Money.dollar(5), Money.franc(7));
    assertEquals("5 USD + 7 CHF", sum.toString());
}

String toString() {
    return augend + " + " + addend;
}
```

Contudo, se quisermos a forma de árvore indentada na expressão, o código se pareceria com isso:

```
testSumPrinting() {
    Sum sum= new Sum(Money.dollar(5), Money.franc(7));
    assertEquals("+\n\t5 USD\n\t7 CHF", sum.toString());
}
```

Teremos que introduzir um Collecting Parameter; algo como isso:

```
String toString() {
    IndentingStream writer= new IndentingStream();
    toString(writer);
    return writer.contents();
}

void toString(IndentingWriter writer) {
    writer.println("+");
    writer.indent();
    augend.toString(writer);
    writer.println();
    addend.toString(writer);
    writer.exdent();
}
```

## Singleton

Como você fornece variáveis globais em linguagens sem variáveis globais? Não faça isso. Seus programas agradecerão por aproveitar o tempo para pensar sobre o projeto em vez disso.

# CAPÍTULO 31

---

## Refatoração

Esses padrões descrevem como mudar o projeto do sistema, mesmo radicalmente, em passos pequenos.

Em TDD, usamos refatoração<sup>1</sup> de uma forma interessante. Geralmente, uma refatoração não pode mudar a semântica do programa sob qualquer circunstância. Em TDD, as circunstâncias com as quais nos importamos são os testes que já passaram. Então, por exemplo, podemos mudar constantes por variáveis em TDD e, com consciência, chamar essa operação de refatoração, pois não muda o conjunto de testes bem-sucedidos. A única circunstância sob a qual a semântica é preservada pode, realmente, ser aquele caso de teste em particular. Qualquer outro caso de teste que estivesse funcionando falharia. Contudo, não temos esses testes ainda, então não precisamos nos preocupar com eles.

Essa “equivalência observacional” põe uma obrigação sobre você para que tenha testes suficientes tal que – até onde você sabe e ao menos quando tiver acabado – uma refatoração com respeito aos testes é o mesmo que uma refatoração com respeito a todos os testes possíveis. Não é desculpa para dizer: “Eu sei que há um problema, mas todos os testes passaram, então eu verifiquei o código”. Escreva mais testes.

---

### Reconciliar diferenças (Reconcile Differences)

Como você unifica dois trechos similares de código? Gradualmente aproxime-os. Unifique-os apenas quando eles forem absolutamente idênticos.

Refatoração pode ser uma experiência enervante. As mais fáceis são óbvias. Se eu extraio um método e faço isso de forma mecanicamente correta, há pouquíss-

---

<sup>1</sup> Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley. ISBN 0201485672.

sima chance de mudar o comportamento do sistema. Mas, algumas refatorações o impulsionam a examinar os fluxos de controle e os valores de dados cuidadosamente. Uma longa cadeia de raciocínio conduz você a acreditar que aquela mudança que está prestes a fazer não mudará qualquer resposta. Essas são as refatorações que aumentam sua calvície.

Tal refatoração com esse salto de fé é exatamente o que estamos tentando evitar com nossa estratégia de passos pequenos e feedback concreto. Embora você nem sempre possa evitar refatorações saltadas, pode reduzir sua incidência.

Essa refatoração ocorre em todos os níveis de escala.

- Duas estruturas de laço são similares. Fazendo-as idênticas, você pode mesclá-las.
- Dois desvios de um condicional são similares. Fazendo-os idênticos, você pode eliminar o condicional.
- Dois métodos são similares. Fazendo-os idênticos, você pode eliminar um.
- Duas classes são similares. Fazendo-as idênticas, você pode eliminar uma.

Às vezes, você precisa abordar a reconciliação de diferenças de forma invertida – isto é, pensar em como o último passo da mudança poderia ser trivial, então trabalhar no sentido contrário. Por exemplo, se você quer remover muitas subclasses, o último passo trivial é se uma classe não contém nada. Então a superclasse pode substituir a classe sem mudar o comportamento do sistema. Para esvaziar essa classe, esse método precisa ser idêntico àquele na superclasse. Uma a uma, esvazie as subclasses e, quando estiverem vazias, substitua referências a elas por referências à superclasse.

---

## Isolar mudança (Isolate Change)

Como você muda uma parte de um método ou objeto multi-part? Primeiro, isole a parte que tem que mudar.

O cenário que vem à minha cabeça é cirurgia: O paciente inteiro, exceto a parte a ser operada, está coberto. A cortina deixa o cirurgião com apenas um conjunto fixo de variáveis. Agora, poderíamos ter longas discussões sobre se essa abstração de uma pessoa a um quadrante inferior esquerdo do abdômen leva a um bom cuidado de saúde, mas, no momento da cirurgia, eu fico feliz que o cirurgião possa se concentrar.

Você pode achar que uma vez isolada a mudança e então feita a mudança, o resultado é tão trivial que pode desfazer o isolamento. Se nós acharmos que tudo o que realmente precisávamos era retornar a variável de instância em `findRate()`, então devemos considerar otimizar `findRate()` em todos os lugares que é usado e deletá-lo. Entretanto, não faça essas mudanças automaticamente. Pese o custo de um método adicional em relação ao valor de ter um conceito adicional explícito no código.

Algumas formas possíveis de isolar mudança são Extrair Método (Extract Method), o mais comum, Extrair Objeto (Extract Object) e Objeto Método (Method Object).

## Migrar dados (Migrate Data)

Como você troca uma representação? Duplique os dados temporariamente.

### Como

Aqui está a versão interna-para-externa na qual você muda a representação internamente e então muda a interface visível externamente.

- Adicione uma variável de instância no novo formato.
- Configure a variável no novo formato em todo lugar que configurou o antigo formato.
- Use a variável no novo formato em todo o lugar que usou o antigo formato.
- Apague o antigo formato.
- Mude a interface externa para refletir o novo formato.

Contudo, às vezes você vai querer mudar a API primeiro. Então deve:

- Adicionar um parâmetro no novo formato.
- Traduzir o parâmetro de novo formato para a representação interna do formato antigo.
- Apagar o parâmetro no formato antigo.
- Substituir usos do formato antigo pelo formato novo.
- Apagar o formato antigo.

### Por quê?

Um para Muitos sempre cria um problema de migração de dados. Suponha que queiramos implementar TestSuite usando Um para Muitos. Começaríamos com:

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.run(self.result)
    assert("1 run, 0 failed" == self.result.summary())
```

que é implementado (na parte “Um” do Um para Muitos) por:

```
class TestSuite:  
    def add(self, test):  
        self.test= test  
    def run(self, result):  
        self.test.run(result)
```

Agora começamos a duplicar dados. Primeiro inicializamos a coleção de testes:

**TestSuite**

```
def __init__(self):  
    self.tests= []
```

Em todo o lugar em que `test` é alterado (set), adicionamos à coleção também:

**TestSuite**

```
def add(self, test):  
    self.test= test  
    self.tests.append(test)
```

Agora usamos a lista de testes em vez de um único teste. Para o propósito dos atuais casos de teste, essa é uma refatoração (ela preserva a semântica), porque há apenas um elemento na coleção.

**TestSuite**

```
def run(self, result):  
    for test in self.tests:  
        test.run(result)
```

Apagamos a variável de instância `test` não mais usada:

**TestSuite**

```
def add(self, test):  
    self.tests.append(test)
```

Você pode também usar migração de dados passo a passo quando trocar entre formatos equivalentes com protocolos diferentes, como na troca de Vetor/Enumeraçor (Vector/Enumerator) de Java para Coleção/Iterador (Collection/Iterator).

---

## Extrair método (Extract Method)

Como você faz um método longo e complicado ser mais fácil de ler? Transforme uma parte pequena dele em um método separado e chame o novo método.

## Como

Extrair Método é realmente uma das mais complicadas refatorações atômicas. Eu descreverei o caso típico aqui. Felizmente, é também a mais comumente implementada refatoração automática, dessa forma é improvável que tenha que fazer isso na mão.

1. Encontre uma região do método que faz sentido ter o próprio método. Corpos de laços, laços inteiros e ramos de condicionais são candidatos comuns para extração.
2. Assegure-se de que não há atribuições para variáveis temporárias declaradas fora do escopo da região a ser extraída.
3. Copie o código do método antigo para o método novo. Compile-o.
4. Para cada variável ou parâmetro do método original usado no novo método, adicione um parâmetro ao novo método.
5. Chame o novo método a partir do método original.

## Por quê?

Eu uso Extrair Método quando estou tentando entender código complicado. “Aqui, esse pedaço aqui está fazendo alguma coisa. O que deveremos chamar aqui?” Depois de meia hora, o código parece melhor, seu parceiro percebe que realmente você está ali para ajudar, e você entende muito melhor o que está acontecendo.

Eu uso Extrair Método para eliminar duplicação quando vejo que dois métodos tem algumas partes iguais e outras partes diferentes. Extraio os pedaços similares como métodos. (A ferramenta *Smalltalk Refactoring Browser* até avança e verifica para ver se você está extraíndo um método que é equivalente a um que já tem e oferece o uso do método existente em vez de criar um novo.)

Quebrar métodos em partes pequenas pode, às vezes, ir longe demais. Quando eu não posso mais ver um jeito de avançar, frequentemente uso Método em uma Linha – *Inline Method* (convenientemente, a próxima refatoração) para ter todo o código em um lugar para que eu possa ver o que deve ser extraído de novo.

---

## Método em uma linha (*Inline Method*)

Como você simplifica fluxos e dados que se tornaram muito retorcidos ou espalhados? Substitua uma invocação de método pelo próprio método.

## Como

1. Copie o método.
2. Cole o método sobre a invocação do método.
3. Substitua todos os parâmetros formais pelos parâmetros reais. Se, por exemplo, você passa `reader.getNext()` (uma expressão que causa efeitos colaterais), então seja cuidadoso ao atribuí-la a uma variável local.

## Por quê?

Um dos revisores deste livro reclamou sobre a sequência na Parte I onde é pedido a um `Bank` para reduzir uma `Expression` para um `Money` simples.

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

“Isso é muito complicado. Por que apenas não pede para `Money` se reduzir?” Como nós experimentamos? Coloque em uma linha a implementação de `Bank.reduce()` e veja o que ela parece:

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= sum.reduce(bank, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

Você poderia gostar mais da segunda versão, ou não. O ponto a observarmos aqui é que você pode usar Método em uma Linha para jogar com o fluxo de controle. Quando estou refatorando, tenho o cenário mental do sistema com pedaços de lógica e fluxos de controle circulando entre os objetos. Quando penso que vejo algo promissor, uso refatorações para testá-lo e ver o resultado.

No calor da batalha, eu ocasionalmente fui apanhado pela minha própria inteligência (Eu não vou dizer quantas vezes isso aconteceu.) Quando isso acontece, Método em uma Linha é uma forma de me rebobinar para: “Eu tenho esse enviando para aquele, enviando para aquele... Opa, Nelly. O que está acontecendo aqui?” Eu coloco em uma linha algumas camadas de abstração, vejo o que está realmente se passando e, então, posso reabstrair o código de acordo com suas reais necessidades, não com meus preconceitos.

---

## Extrair interface (Extract Interface)

Como você introduz uma segunda implementação de operações em Java? Crie uma interface contendo as operações compartilhadas.

### Como

1. Declare uma interface. Às vezes, o nome da classe existente deve ser o nome da interface, e, nesse caso, você deve, primeiro, renomear a classe.
2. Faça a classe existente implementar a interface.
3. Adicione os métodos necessários à interface, expandindo a visibilidade dos métodos na classe se necessário.
4. Mude declarações de tipo da classe para a interface onde for possível.

### Por quê?

Às vezes, quando você precisa extrair uma interface, está genuinamente trocando da primeira implementação para a segunda. Você tem um `Rectangle` e quer adicionar `Oval`, então você cria uma interface `Shape`. Encontrar nomes para as interfaces, nesse caso, é geralmente fácil, embora, às vezes, tenha que se esforçar para encontrar a metáfora certa.

Às vezes, você está introduzindo um Modelo de Teste de Acidentes (Crash Test Dummy) ou outro Objeto Simulado (Mock Object) quando precisa extrair uma interface. Nomear é geralmente mais difícil nesse caso, pois você ainda só tem um exemplo real. Esses são os momentos em que eu fico mais tentado a dar um jeitinho e nomear a interface como `IFile` e deixar a classe como `File`. Eu me eduquei a parar um momento e ver se não estou entendendo alguma coisa mais profunda sobre o que está acontecendo. Talvez a interface devesse ser chamada `File` e a classe `DiskFile`, pois a classe assume que os bits estão em um disco.

---

## Mover método (Move Method)

Como você move um método para o lugar ao qual ele pertence? Adicione-o à classe à qual ele pertence, então invoque-o.

### Como

1. Copie o método.
2. Cole o método, adequadamente nomeado, na classe alvo. Compile-o.

3. Se o objeto original é referenciado no método, então adicione um parâmetro para passar o objeto original. Se variáveis do objeto original são referenciadas, então passe-as como parâmetros. Se variáveis do original são alteradas (set), então você deveria desistir.
4. Substitua o corpo do método original por uma invocação do novo método.

## Como

Essa é uma das minhas consultorias favoritas em refatoração, pois é tão bom descobrir preconceitos não comprovados. Calcular áreas é a responsabilidade de Shape:

### Shape

```
...
int width= bounds.right() - bounds.left();
int height= bounds.bottom() - bounds.top();
int area= width * height;
...
```

A qualquer momento em que eu veja mais que uma mensagem ser enviada a outro objeto em um método, fico desconfiado. Nesse caso, vejo que bounds (um Rectangle) está enviando quatro mensagens. Hora de mover essa parte do método:

### Rectangle

```
public int area() {
    int width= this.right() - this.left();
    int height= this.bottom() - this.top();
    return width * height;
}
```

### Shape

```
...
int area= bounds.area();
...
```

As três grandes propriedades de Mover Método são as seguintes.

- É fácil ver a necessidade dele sem conhecer profundamente o significado do código. Você vê duas ou mais mensagens para um objeto diferente e aí vai você.

- As mecânicas são rápidas e seguras.
- Os resultados são frequentemente iluminados. “Mas Rectangles não fazem qualquer cálculo... ó, eu entendo. Isso é melhor.”

Às vezes, você vai querer mover apenas parte de um método. Você pode primeiro extrair um método, mover o método inteiro, então colocar em uma linha o método (agora cabendo em uma linha) na classe original. Ou você pode descobrir os mecanismos para fazê-lo de uma vez.

---

## Objeto método (Method Object)

Como você representa um método complicado que requer muitos parâmetros e variáveis locais? Faça um objeto a partir do método.

### Como?

- Crie um objeto com os mesmos parâmetros do método.
- Faça das variáveis locais as variáveis da instância do objeto também.
- Crie um método chamado `run()` cujo corpo é o mesmo do método original.
- No método original, crie um novo objeto e invoque `run()`.

### Por quê?

Objeto Método é útil na preparação para adicionar um tipo de lógica completamente nova no sistema. Por exemplo, você poderia ter muitos métodos envolvidos em calcular o fluxo de caixa do componente fluxos de caixa. Quando quiser começar a calcular o valor atual do fluxo de caixa líquido, você pode, primeiro, criar o Objeto Método fora do primeiro estilo de computação. Então, você pode escrever o novo estilo de computação com seus próprios testes em menor escala. Então ligar (“plugar”) o novo estilo será um único passo.

Objeto Método é também bom para simplificar código que não se submete a Extrair Método. Às vezes, você encontrará um bloco de código que tem um monte de variáveis temporárias e parâmetros, e, toda vez que tenta extraír um pedaço dele, tem que carregar cinco ou seis temporárias e parâmetros. O método extraído

resultante não parece melhor que o código original, porque a assinatura do método é muito longa. Criar um Objeto Método dá a você um novo namespace no qual você pode extrair métodos sem ter que passar nada.

---

## Adicionar parâmetro (Add Parameter)

Como você adiciona um parâmetro a um método?

### Como

1. Se o método está em uma interface, adicione o parâmetro à interface primeiro.
2. Adicione o parâmetro.
3. Use os erros do compilador para ver qual código chamado você precisa mudar.

### Por quê?

Adicionar um parâmetro é, frequentemente, um passo de extensão. Você teve o primeiro caso de teste executando sem precisar do parâmetro, mas, nessa nova circunstância, você tem que levar em consideração mais informação, de forma a computar corretamente.

Adicionar um parâmetro pode ser também parte da migração de uma representação de dados para outra. Primeiro, você adiciona o parâmetro, então elimina todos os usos do parâmetro antigo, e então remove o parâmetro antigo.

---

## Parâmetro de método para parâmetro de construtor (Method Parameter to Constructor Parameter)

Como você troca um parâmetro de um método ou de métodos para o construtor?

### Como

1. Adicione um parâmetro ao construtor.
2. Adicione uma variável de instância com o mesmo nome do parâmetro.
3. Atribua (set) a variável no construtor.
4. Uma a uma, converta as referências de “parameter” para “this.parameter”.

5. Quando não existirem mais referências ao parâmetro, elimine o parâmetro do método e de todos aqueles que o chamam.
6. Remova os agora supérfluos “this.” das referências.
7. Renomeie a variável corretamente.

### Por quê?

Se você passa o mesmo parâmetro a muitos métodos diferentes no mesmo objeto, então pode simplificar a API passando o parâmetro uma vez (eliminando duplicação). Você pode executar essa refatoração ao reverso se achar que uma variável de instância é usada em apenas um método.

# CAPÍTULO 32

---

## Dominando TDD

Eu espero levantar aqui questões para você ponderar sobre como integrar TDD em sua própria prática. Algumas das questões são pequenas, e algumas são grandes. Às vezes, as respostas estão aqui, ou, ao menos, sugeridas, e, às vezes, as questões são deixadas para você explorar.

### Quão grandes deveriam ser seus passos?

Há realmente duas questões espreitando aqui:

- Quanto terreno cada teste deveria cobrir?
- Por quantos estágios intermediários você deveria passar enquanto refatora?

Você poderia escrever os testes de forma que cada um deles incentive a adição de uma só linha de lógica e um punhado de refatorações. Você poderia escrever os testes de forma que cada um deles incentive a adição de centenas de linhas de lógica e horas de refatoração. Qual você deve escolher?

Parte da resposta é que você deveria ser capaz de fazer qualquer uma. A tendência de Desenvolvedores Guiados por Testes ao longo do tempo, porém, é clara – passos menores. Contudo, pessoas estão experimentando dirigir desenvolvimento por testes em nível de aplicação sozinhos ou em conjunto com os testes em nível de programador que viemos escrevendo.

A princípio, quando você refatora, deveria estar preparado para dar muitos passinhos pequenos. Refatoração manual está propensa a erros, e, quanto mais erros você cometer e descobrir apenas depois, menos provável é a refatoração. Uma vez que tenha feito uma refatoração 20 vezes à mão em passinhos pequenos, experimente deixar de fora alguns dos passos.

Refatoração automática acelera a refatoração enormemente. O que teria exigido de você 20 passos manuais agora se torna um único item do menu. Uma ordem de magnitude mudada na quantidade geralmente constitui uma mudança na qualidade, e isso é verdadeiro para refatoração automática. Quando você sabe que tem o suporte de uma ferramenta excelente, você se torna muito mais agressivo em suas refatorações, tentando muito mais experimentos para ver como o código quer ser estruturado.

O Refactoring Browser for Smalltalk é ainda, enquanto escrevo isso, a melhor ferramenta de refatoração disponível. Suporte à refatoração em Java está aparecendo em muitas IDEs de Java, e o suporte à refatoração certamente vai se espalhar rapidamente a outras linguagens e ambientes.

## O que você não tem que testar?

A resposta simples, fornecida por Phlip, é: “Escreva testes até que o medo se transforme em tédio”. Contudo, esse é um laço de feedback e requer que você encontre a resposta sozinho. Como você veio a esse livro para ter respostas, não questões (nesse caso você já está lendo a seção errada, mas chega de trechos literários autorreferenciais), tente essa lista. Você deveria testar:

- Condicionais
- Laços
- Operações
- Polimorfismo

Mas, apenas aqueles que você escreve. A menos que tenha razão para desconfiar, não teste código de outros. Às vezes, a especificação exata de código externo requer que você escreva mais sua própria lógica. Veja acima se você tem que testar isso. Às vezes, apenas para ser mais cuidadoso, documentarei a presença de, hum, comportamento incomum em código externo com um teste que falhará se o erro nunca for corrigido, digo, se o comportamento nunca for refinado.

## Como você sabe se tem bons testes?

Os testes são um canário em uma mina de carvão revelando por sua aflição a presença de vapores malignos no projeto. Aqui estão alguns atributos de teste que sugerem um projeto com problemas.

- Longo código de configuração – Se você tem que gastar uma centena de linhas criando objetos para uma asserção simples, então algo está errado. Seus objetos são muito grandes e precisam ser divididos.
- Duplicação de configuração – Se você não puder encontrar facilmente um lugar comum para código de configuração comum, então há muitos objetos fortemente entrelaçados.

- Testes de longa execução – Testes de TDD que rodam por um longo tempo não serão rodados frequentemente, e frequentemente não foram executados por algum tempo, e, provavelmente, não funcionam. Pior que isso, eles sugerem que testar os trechos e partes da aplicação é difícil. A dificuldade de testar trechos e partes é um problema de projeto e precisa ser tratado com projeto. (O equivalente a  $9,8\text{m/s}^2$ \* é a suíte de teste de dez minutos. Suítes que levam mais que dez minutos inevitavelmente são aparadas, ou a aplicação é ajustada, de forma que a suíte leve dez minutos de novo.)
- Testes frágeis – Testes que quebram inesperadamente sugerem que uma parte da aplicação está surpreendentemente afetando outra parte. Você precisa projetar até que o efeito seja eliminado, ou quebrando a conexão ou juntando as duas partes.

## Como TDD leva a frameworks?

Paradoxo: não considerando o futuro do seu código, você torna seu código muito mais provável de ser adaptável no futuro.

Eu aprendi exatamente o oposto dos livros: “Codifique para hoje, projete para amanhã”. TDD parece estar com esse conselho na cabeça: “Codifique para amanhã, projete para hoje”. Aqui está o que acontece na prática.

- A primeira feature entra. Ela é implementada simples e diretamente, logo, está pronta rapidamente e com alguns defeitos.
- A segunda feature, uma variação da primeira, entra. A duplicação entre as duas features é posta em um lugar, ao passo que as diferenças tendem a entrar em lugares diferentes (métodos diferentes ou mesmo classes diferentes).
- A terceira feature, uma variação da segunda, entra. A lógica comum é plausível de ser reutilizada, talvez com alguns ajustes. A lógica única tende a ter um habitat óbvio em um método diferente ou em uma classe diferente.

O Princípio Aberto/Fechado (objetos deveriam ser abertos para uso e fechados para outras modificações) é gradualmente satisfeito e, precisamente, para esses tipos de variação que ocorrem na prática. Desenvolvimento dirigido por testes deixa você com frameworks que são bons em expressar exatamente o tipo de variação que ocorre, mesmo que os frameworks possam não ser bons em expressar o tipo de variação que não ocorre (ou não ocorreu ainda).

Então, o que acontece quando uma variação incomum surge três anos depois? O projeto passa por uma evolução rápida exatamente nos pontos necessários

---

\* N. de R. T.: Este é o valor constante aproximado da aceleração da gravidade na Terra. O autor provavelmente quer relacionar esse valor constante à sua sugestão de tempo (constante) máximo de testes – 10 minutos.

para acomodar a variação. O Princípio Aberto/Fechado é violado, apenas por um momento, mas a violação não é tão custosa, pois você tem todos aqueles testes para dar a confiança de que não está estragando nada.

No limite, onde você introduz as variações muito rápido, TDD é indistinguível de projetar para frente. Eu ampliei um framework de notificação certa vez no decorrer de umas poucas horas, e observadores estavam absolutamente certos de que era um truque. Devo ter começado com o framework resultante na cabeça. Não, desculpe. Apenas tive desenvolvimento dirigido por teste o suficiente para que pudesse recuperar a maioria dos meus erros mais rápido que você possa reconhecê-los. Eu mesmo os fiz.

## Quanto feedback você precisa?

Quantos testes você deve escrever? Aqui há um problema simples: dados três inteiros representando o comprimento dos lados de um triângulo, retorne:

- 1 se o triângulo é equilátero
- 2 se o triângulo é isósceles
- 3 se o triângulo é escaleno

e lance uma exceção se o triângulo não é bem formado.

Vá em frente e tente resolver o problema (minha solução em Smalltalk está listada no final dessa questão).

Eu escrevi seis testes (um tipo de “Qual é a Música?”: “Eu posso codificar aquele problema em quatro testes.” “Codifique aquele problema.”). Bob Binder, no seu abrangente livro *Testing Object-Oriented Systems*<sup>1</sup>, escreveu 65 para o mesmo problema. Você terá que decidir, da experiência e da reflexão, quantos testes quer escrever.

Eu penso em Tempo Médio Entre Falhas (TMEF) quando penso em quantos testes escrever. Por exemplo, inteiros em Smalltalk agem como inteiros, não como um contador de 32 bits, então não faz sentido testar MAXINT. Bem, há um tamanho máximo para um inteiro, mas tem a ver com a quantidade de memória que você tem. Preciso escrever um teste que encha a memória com inteiros extremamente grandes? Como isso afetará o TMEF do meu programa? Se nunca vou chegar tão perto desse tamanho de triângulo, meu programa não é mensuravelmente mais robusto com esse teste que sem esse teste.

Se um teste tem sentido em ser escrito depende de quanto cuidadosamente você mede o TMEF. Se você está tentando mudar um TMEF de 10 anos para um TMEF de 100 anos em seu marca-passo, então testes para condições extremamente improváveis e combinações de condições fazem sentido, a menos que possa demonstrar de outro jeito que as condições não surgem.

---

<sup>1</sup> Binder, Bob. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, MA: Addison-Wesley. ISBN 0201809389. Esta é a referência abrangente sobre teste.

A visão de teste de TDD é pragmática. Em TDD, os testes são um meio para atingir um fim – o ser código final no qual temos grande confiança. Se nosso conhecimento da implementação nos dá confiança mesmo sem um teste, então não escreveremos aquele teste. Teste caixa preta, em que deliberadamente escolhemos ignorar a implementação, tem algumas vantagens. Ao ignorar o código, ele demonstra um sistema de valor diferente – os testes são valiosos sozinhos. É uma atitude apropriada para ser tomada em algumas circunstâncias, mas que é diferente de TDD.

### TriangleTest

```
testEquilateral
    self assert: (self evaluate: 2 side: 2 side: 2) = 1

testIsosceles
    self assert: (self evaluate: 1 side: 2 side: 2) = 2

testScalene
    self assert: (self evaluate: 2 side: 3 side: 4) = 3

testIrrational
    [self evaluate: 1 side: 2 side: 3]
        on: Exception
        do: [:ex | ^self].
    self fail

testNegative
    [self evaluate: -1 side: 2 side: 2]
        on: Exception
        do: [:ex | ^self].
    self fail

testStrings
    [self evaluate: 'a' side: 'b' side: 'c']
        on: Exception
        do: [:ex | ^self].
    self fail

evaluate: aNumber1 side: aNumber2 side: aNumber3
    | sides |
    sides := SortedCollection
        with: aNumber1
        with: aNumber2
        with: aNumber3.
    sides first <= 0 ifTrue: [self fail].
    (sides at: 1) + (sides at: 2) <= (sides at: 3) ifTrue: [self fail].
    ^sides asSet size
```

## Quando você deve apagar testes?

Quanto mais testes, melhor, mas se dois testes são redundantes um com respeito ao outro, deveria manter ambos? Isso depende de dois critérios.

- O primeiro critério para seus testes é confiança. Nunca apague um teste se isso reduz sua confiança no comportamento do sistema.
- O segundo critério é comunicação. Se você tem dois testes que exercitam o mesmo caminho de código, mas falam de cenários diferentes para um leitor, deixe-os em paz.

Dito isso, se você tem dois testes que são redundantes com respeito a confiança e comunicação, apague o menos útil dos dois.

## Como a linguagem e o ambiente de programação influenciam TDD?

Tente TDD em Smalltalk com o Refactoring Browser. Tente-o em C++ com vi. Como sua experiência difere?

Em linguagens e ambientes de programação em que os ciclos de TDD (teste/compile/rode/refatore) são mais difíceis de fazer, você provavelmente será tentado a dar passos maiores:

- Cubra mais terreno com cada teste.
- Refatore com menos passos intermediários.

Isso faz você ir mais rápido ou devagar?

Em linguagens e ambientes de programação onde os ciclos de TDD são abundantes, você provavelmente será instigado a tentar muito mais experimentos. Isso ajuda você a ir mais rápido ou a alcançar soluções melhores, ou você estaria melhor livre da institucionalização de algum tipo de tempo para pura reflexão (revisões ou uso de *literate programming*<sup>\*</sup>)?

## Você pode guiar-se por testes em sistemas enormes?

TDD é escalável para sistemas extremamente grandes? Que novos testes você teria que escrever? Que novos tipos de refatoração você precisaria?

O maior sistema totalmente dirigido por testes em que estive envolvido é no *LifeWare* ([www.lifeware.ch](http://www.lifeware.ch)). Depois de 4 anos e 40 pessoas/ano, o sistema continha aproximadamente 250.000 linhas de código funcional e 250.000 linhas de código de teste em Smalltalk. Existem 4.000 testes executando em cerca de 20

---

\* N. de R. T.: *Literate programming* é um enfoque de programação introduzido por Donald Knuth e tem sido tipicamente usado para ensino de programação. O uso de *literate programming* leva a programas escritos como em um texto em linguagem natural, explicando a lógica de resolução de um problema, mas que contém também macros e código fonte tradicional, que pode ser compilado e executado.

minutos. A suíte completa é rodada muitas vezes por dia. A quantidade de funcionalidade no sistema parece não ter influência sobre a efetividade de TDD. Eliminando duplicação, você tende a criar mais objetos menores, e esses objetos podem ser testados em isolamento independente do tamanho da aplicação.

### Você pode guiar o desenvolvimento com testes em nível de aplicação?

O problema com desenvolvimento dirigido por testes em pequena escala (eu os chamo “testes de unidade”, mas eles não condizem muito bem com a definição aceita de testes de unidade) é que você corre o risco de implementar o que acha que os usuários querem, mas ter isso pode ser o caso de não ser absolutamente o que eles queriam. E se escrevêssemos os testes no nível da aplicação? Então os usuários (com ajuda) poderiam escrever os próprios testes para exatamente o que querem que o sistema faça em seguida.

Há um problema técnico – *fixturing*. Como você pode escrever e rodar um teste para uma feature que ainda não existe? Sempre parece haver alguma saída para esse problema, tipicamente introduzindo um interpretador que, educadamente, sinaliza um erro quando ele se depara com um teste que ainda não sabia como interpretar.

Há também um problema social com desenvolvimento dirigido por testes de aplicação (*Application Test-Driven Development* – ATDD). Escrever testes é uma nova responsabilidade para usuários (aqui eu realmente quero dizer um time que inclui usuários), e essa responsabilidade vem em um novo lugar no ciclo de desenvolvimento – nominalmente, antes da implementação começar. Organizações resistem a esse tipo de mudança de responsabilidade. Ela vai exigir um esforço concentrado (que é o esforço de muitas pessoas no time trabalhando juntas) para obter testes de aplicação escritos primeiro.

TDD como descrita neste livro é uma técnica que está inteiramente sob seu controle. Você pode pegá-la e começar a usá-la hoje, se assim escolher. A mistura do ritmo de vermelho/verde/refatore, das questões técnicas das fixtures da aplicação e das questões de mudança organizacional envolvendo testes escritos por usuários é improvável que seja bem-sucedida. A regra do Teste de Um Só Passo se aplica. Tenha vermelho/verde/refatore em sua própria prática, então espalhe a mensagem.

Outro aspecto de ATDD é o comprimento do ciclo entre teste e retorno. Se um cliente escreveu um teste e, dez dias depois, ele finalmente funcionou, você estaria vislumbrando a barra vermelha a maior parte do tempo. Eu acho que ainda queria fazer TDD em nível de programador, de modo que:

- Eu tenha barras verdes imediatamente;
- Eu simplifique o projeto interno.

### Como você muda para TDD no meio do caminho?

Você tem um monte de código que funciona mais ou menos. Você quer dirigir por testes o seu novo código. O que você faz como próximo passo?

Há um livro (ou livros) inteiro a ser escrito sobre mudança para TDD quando você tem montes de código. O que segue é necessariamente apenas uma provocação.

O maior problema é que o código que não é escrito com testes em mente tipicamente não é muito testável. As interfaces não são projetadas, então é fácil para você isolar um pequeno pedaço de código, rodá-lo e verificar o resultado.

“Corrija-o”, você diz. Sim, bem, mas é possível que a refatoração (sem ferramentas automatizadas) resulte em erros, erros que você não pegará, pois não tem os testes. Galinhas e ovos. Artil 22. Destrução mutuamente garantida. O que você faz?

O que você não faz é escrever testes para a coisa toda e refatorar a coisa toda. Isso levaria meses, meses em que nenhuma nova funcionalidade apareceria. Gastar dinheiro sem fazê-lo é geralmente falar de um processo não sustentável.

Então, primeiro, temos que decidir limitar o escopo de nossas mudanças. Se vemos partes do sistema que poderiam ser dramaticamente simplificadas, mas não exigem mudança no momento, então as deixamos em paz. Derrame uma lágrima, talvez, pelos pecados do passado, mas deixe-as em paz.

Segundo, temos que quebrar o impasse entre testes e refatoração. Podemos ter feedback de outras formas além de testes, como trabalhar muito cuidadosamente e com um parceiro. Podemos ter feedback a grosso nível, como testes em nível de sistema que sabem não ser adequados, mas que nos dão alguma confiança. Com esse feedback, podemos fazer as áreas que temos que mudar mais receptivas à mudança.

No decorrer do tempo, as partes do sistema que mudam todo o tempo vão parecer dirigidas por testes. Ocasionalmente, vamos passar por um beco sem iluminação e ser atacados por nossos problemas, lembrando-nos de quão lentas as coisas costumavam ser. Então vamos desacelerar, quebrar o impasse e ir andando de novo.

## Para quem TDD é destinado?

Cada prática de programação codifica um sistema de valores, explícita ou implicitamente. TDD não é diferente. Se você está feliz jogando algum código que funciona mais ou menos e está feliz em nunca olhar para o resultado de novo, TDD não é para você. TDD repousa em uma encantadora e ingênua suposição “nérdica”\*: se você escreve código melhor, será mais bem-sucedido. TDD o ajuda a prestar atenção nas questões certas no tempo certo, assim você pode fazer seus projetos mais limpos; você pode refiná-los conforme aprende.

Eu digo “ingênua”, mas talvez seja exagero. O que é ingênuo é assumir que código limpo é tudo o que existe para o sucesso. Boa engenharia é talvez 20 por cento do sucesso de um projeto. Má engenharia certamente afunda projetos, mas engenharia modesta pode permitir o sucesso de um projeto enquanto os outros 80 por cento funcionam direito.

---

\* N. de R. T.: Relativa a nerds (no original era *geekoid*, relativa a *geeks*), em que *nerds* e *geeks* são termos usados para pessoas aficionadas por tecnologia e consideradas não muito sociáveis.

Nessa perspectiva, TDD é fantástica. Ele deixa você escrever código com muito menos defeitos e com um design muito mais limpo do que é comum no mercado. Contudo, aqueles cujas almas estão curadas pelo bálsamo da elegância podem achar em TDD uma forma de ir bem por fazer o que é bom.

TDD é também boa para nerds que formam uma ligação sentimental com o código. Uma das grandes frustrações da minha vida de jovem engenheiro era começar um projeto com grande excitação, então observar o código se degenerar com o tempo. Um ano depois, eu não queria mais nada além de jogar no lixo o código, agora fedorento, e começar o próximo projeto. TDD possibilita a você ganhar confiança no código no decorrer do tempo. Conforme os testes se acumulam (e seu teste melhora), você ganha confiança no comportamento do sistema. Conforme você refina o projeto, mais e mais mudanças se tornam possíveis. Meu objetivo é me sentir melhor com o código depois de um ano do começo deslumbrado, e TDD me ajuda a conseguir isso.

## TDD é sensível a condições iniciais?

Há uma certa ordem em que os testes parecem funcionar muito bem. Vermelho/verde/refatore/vermelho/verde/refatore. Você pode pegar os mesmos testes e implementá-los em uma ordem diferente, e parece que não existe jeito de avançar em pequenos passos. É realmente verdade que uma sequência de testes é uma ordem de grandeza mais rápida/fácil de implementar que outra? É só porque minha técnica de implementação não está à altura do desafio? Há alguma coisa sobre os testes que deveria me dizer para enfrentá-los em certa ordem? Se TDD é sensível a condições iniciais num projeto pequeno, então é previsível num projeto grande? (Da mesma forma que pequenos redemoinhos no Mississippi são imprevisíveis, mas você pode contar com 56.000 metros cúbicos por segundo, mais ou menos, na foz do rio.)

## Como TDD se relaciona com padrões?

Todos os meus textos técnicos têm sido sobre tentar encontrar regras fundamentais que gerem comportamento similar ao dos especialistas. Em parte, é porque é como eu aprendo – eu acho um especialista para agir como ele, e, então, gradualmente descrevo o que está realmente acontecendo. Certamente não estou procurando por regras a serem seguidas mecanicamente, embora seja assim que os que têm raciocínio mecânico as tenham interpretado.

Minha filha mais velha (Olá, Bethany! Eu disse que teria você aqui – fique feliz por isso não ser mais constrangedor) passou muitos anos aprendendo a fazer rapidamente uma multiplicação. Minha esposa e eu nos orgulhamos de fazer multiplicação rápida, e aprendemos muito rápido. O que estava acontecendo? Acontece que toda a vez que Bethany se defrontava com  $6 \times 9$ , ela somaria 6 nove vezes (ou 9 seis vezes, eu suponho). Longe de ser uma multiplicadora lenta, ela era uma somadora realmente rápida.

O efeito que eu notei, e que espero que outros encontrem, é que, reduzindo comportamento repetitivo a regras, aplicar as regras se torna mecânico e um hábito.

Isso é mais rápido que rediscutir tudo sempre dos princípios. Quando aparece uma exceção ou um problema que não se encaixa em nenhuma das regras, você tem mais tempo e energia para criar e aplicar criatividade.

Isso me acontecia quando escrevia os Padrões das Melhores Práticas Small-talk. Em algum ponto decidi apenas seguir as regras que eu estava escrevendo. Era muito mais lento no começo, ficar procurando as regras ou parando para escrever uma nova regra. Depois de uma semana, contudo, descobri que o código estava arrancando as pontas dos meus dedos e que exigiria, antes, uma pausa para reflexão. Isso me deu mais tempo e atenção para reflexões maiores sobre projeto e análise.

Outro relacionamento entre TDD e padrões é TDD como um método de implementação para projeto dirigido por padrões. Digamos que decidimos que queremos um *Strategy*\* para algo. Nós escrevemos um teste para a primeira variante e a implementamos como um método. Então, conscientemente, escrevemos um teste para a segunda variante, esperando que a fase de refatoração nos guie a um *Strategy*. Robert Martin e eu fizemos algumas pesquisas sobre esse estilo de TDD. O problema é que o projeto continua surpreendendo você. Ideias de projeto perfeitamente sensatas revelam-se erradas. O melhor é apenas pensar no que você quer que o sistema faça e deixar o projeto se resolver mais tarde.

## Por que TDD funciona?

Prepare-se para deixar a galáxia. Vamos assumir por um momento que TDD ajuda as equipes a construir produtivamente sistemas fracamente acoplados e altamente coesos com baixas taxas de defeito e baixo custo de perfil de manutenção. (Eu não estou afirmando isso no geral, mas confio em você para imaginar coisas impossíveis.) Como tal coisa poderia acontecer?

Parte do efeito certamente vem de reduzir defeitos. Quanto mais cedo encontrar e corrigir um defeito, mais barato é, e, frequentemente, mais comovente (basta perguntar a Mars Lander). Existem abundantes efeitos psicológicos e sociais na redução de defeitos. Minha própria prática de programação se tornou muito menos estressante quando comecei com TDD. Nunca mais precisei me preocupar com tudo de uma vez. Eu poderia fazer esse teste rodar e depois todo o resto. O relacionamento com minha equipe ficou mais positivo. Parei de estragar builds, e as pessoas podiam confiar que meu software funciona. Clientes dos meus sistemas ficaram mais positivos também. Uma nova liberação de sistema apenas significava mais funcionalidade, não uma tropa de novos defeitos para identificar dentre todos os seus antigos erros favoritos.

Redução de defeitos. Onde posso sair afirmando isso? Eu tenho uma prova científica?

Não. Nenhum estudo demonstrou categoricamente a diferença entre TDD e qualquer uma das muitas alternativas de qualidade, produtividade ou diver-

---

\* N. de R. T.: Padrão Gof Strategy. *Strategy* é usado para encapsular um algoritmo dentro de uma classe, de modo que o algoritmo possa variar independentemente dos clientes que o utilizam.

são. Contudo, a evidência dos casos que todos contam é esmagadora e os efeitos secundários são inconfundíveis. Programadores realmente relaxam, times realmente desenvolvem confiança e clientes realmente aprendem a olhar adiante para as novas entregas (*releases*). “De modo geral”, eu direi, embora não tenha visto o efeito oposto. Sua quilometragem pode variar, mas terá que tentar para descobrir.

Outra vantagem de TDD que pode explicar esse seu efeito é o modo como ele encurta o ciclo de feedback nas decisões de projeto. O ciclo de feedback para decisões de implementação é obviamente curto – segundos ou minutos, seguido por reexecutar os testes dezenas ou milhares de vezes por dia. O ciclo para decisões de projeto vai entre a reflexão do projeto – talvez a API devesse preferir isso, ou talvez a metáfora devesse ser essa – e o primeiro exemplo, um teste que incorpora aquela reflexão. Em vez de projetar e então esperar semanas ou meses para alguma outra pessoa sentir a dor ou a glória, feedback entra em segundos ou minutos conforme você tenta traduzir suas ideias em uma interface plausível.

Uma resposta mais estranha para “Por que TDD funciona?” vem da imaginação febril de sistemas complexos. O inimitável Phlip diz:

Adote práticas de programação que “atraiam” código correto como uma função limite, não como um valor absoluto. Se você escreve testes de unidade para cada feature, e se você refatora para simplificar o código entre cada etapa, e se você adiciona features uma por vez e apenas depois de todos os testes de unidade passarem, você criará o que matemáticos chamam de um “atrator”. Isso é um ponto em um espaço de estado para o qual todos os fluxos convergem. Codificar é mais como mudar para o melhor ao longo do tempo do que mudar para o pior; o atrator aproxima a corretude como uma função limite.

Essa é a “corretude” com que quase todos os programadores lidam com dificuldade (exceto, claro, para software nas áreas médica ou aeroespacial). Mas, é melhor entender explicitamente o conceito de atrator que negá-lo ou desconsiderar sua importância.

## E qual é a do o nome?

- Desenvolvimento – A velha maneira de pensar em desenvolvimento de software como “fases” está enfraquecida devido ao feedback entre as decisões ser difícil se elas estão separadas no tempo. Desenvolvimento nesse sentido significa uma dança complexa de análise, projeto lógico, projeto físico, implementação, teste, revisão, integração e implantação.
- Guiado – Eu costumo chamar TDD de “programação com teste primeiro”. Contudo, o oposto de “primeiro” é “no final”, e muitas pessoas testam depois que elas programam. Há uma regra de nomeação em que o oposto de um nome deve ser, no mínimo, vagamente insatisfatório. (Parte do apelo da programação estruturada é que ninguém quer que seja não estruturada.) Se você não desenvolve guiado por testes, pelo que você se guia? Especulação? Especificações? (Nunca notou que essas duas palavras vêm da mesma raiz comum?)

- Teste – Testes automatizados, materializados, concretos. Aperte um botão e eles executam. Uma das ironias de TDD é que não é uma técnica de teste. É uma técnica de análise, uma técnica de projeto, realmente uma técnica para estruturar todas as atividades de desenvolvimento.

## Como TDD se relaciona com as práticas da Programação Extrema?

Alguns revisores deste livro estavam preocupados que, por eu escrever um livro exclusivamente sobre TDD, as pessoas tomariam isso como uma desculpa para ignorar o resto dos conselhos em Programação Extrema (XP). Por exemplo, se você se guia por testes, você ainda precisa programar em pares? Aqui está um breve resumo de como o resto de XP melhora TDD, e como TDD melhorar o resto de XP.

- Programação em pares – Os testes que você escreve em TDD são excelentes pedaços de conversa quando você está programando em par. O problema que você evita é de os parceiros não concordarem em qual dos problemas estão resolvendo, mesmo que estejam tentando trabalhar no mesmo código. Isso soa meio maluco, mas acontece o tempo todo, especialmente quando está aprendendo a programar em par com alguém. Pares melhoraram TDD dando a você uma mente descansada para assumir quando você fica cansado. O ritmo de TDD pode drená-lo e fazê-lo continuar a programar mesmo quando está cansado. Seu parceiro, contudo, está preparado para pegar o teclado quando você tombar.
- Trabalhe descansado – Em uma nota relacionada, XP aconselha você a trabalhar quando estiver descansado e parar quando estiver cansado. Quando não puder fazer o próximo teste funcionar, ou aqueles dois testes funcionarem juntos, é hora de um intervalo. O tio Bob Martin\* e eu estávamos trabalhando em um algoritmo de quebra de linha uma vez, e não conseguíamos fazê-lo funcionar. Lutamos contra a frustração por alguns minutos, mas era óbvio que não estávamos fazendo progresso, então paramos.
- Integração contínua – Os testes são um recurso excelente, fazendo você capaz de integrar mais frequentemente. Você obtém outro teste funcionando e remove a duplicação, e você faz o *check-in*. O ciclo pode ser de 15 a 30 minutos em vez das 1 ou 2 horas que eu geralmente levava. Isso pode ser parte da chave para ter times maiores de programadores no mesmo código base. Como Bill Wake diz “um problema  $n^2$  não é um problema se  $n$  é sempre 1.”
- Projeto simples – Codificando apenas o que você precisa para os testes e removendo toda a duplicação, você automaticamente tem um projeto que é perfeitamente adaptado aos requisitos atuais e igualmente preparado

---

\* N. de R. T.: Robert Martin, vulgo “Tio Bob” (*Uncle Bob*), conhecido desenvolvedor de software, consultor e autor de livros e, como Kent Beck, signatário do Manifesto Ágil.

para histórias futuras. A concepção que você está procurando para apenas projetar o suficiente para atingir a arquitetura perfeita para o sistema atual também torna mais fácil a escrita de testes.

- Refatoração – A regra de Remoção de Duplicação é outra forma de dizer “refatoração”. Mas, os testes dão a você confiança de que suas refatorações maiores não mudaram o comportamento do sistema. Quanto maior é sua confiança, mais agressivo você será ao tentar refatorações em larga escala que estendem a vida do seu sistema. Refatorando, você torna a escrita da próxima rodada de testes muito mais fácil.
- Entrega contínua – Se testes de TDD realmente aumentam o TMEF do seu sistema (uma prova você terá que verificar sozinho), então você pode colocar código em produção muito mais frequentemente sem perturbar clientes. Gareth Reeves faz a analogia à negociação diurna (*day trading*). Em uma negociação diurna, você fecha suas posições toda a noite, pois não quer um risco que não esteja gerenciando. Em programação, você prefere todas as suas mudanças na produção, pois não quer código que não esteja recebendo feedback concreto.

## Desafio de Darach

Darach Ennis fez um desafio para estender o alcance de TDD. Ele disse:

Há um monte de falácias soprando em torno de várias organizações de engenharia e entre vários engenheiros que este livro poderia ajudar a desmitificar, e algumas delas são:

- Você não pode testar GUIs automaticamente (por exemplo, Swing, CGI, JSP/Servlets/Struts).
- Você não pode fazer testes de unidade em objetos distribuídos automaticamente (por exemplo, RPC e estilo Mensagem, ou CORBA/EJB e JMS).
- Você não pode desenvolver testando primeiro seu esquema de banco de dados (por exemplo, JDBC).
- Não há necessidade de testar código de terceiros ou gerado por ferramentas externas.
- Você não pode desenvolver testando primeiro um compilador/interpretador de linguagem de BNF para implementação em qualidade de produção.

Não tenho certeza de que ele esteja certo, mas eu também não tenho certeza de que ele esteja errado. Ele me deu algo para mascar conforme penso em quão longe empurrar TDD.

# APÊNDICE I

---

## Diagramas de Influência

Este livro contém muitos exemplos de diagramas de influência. A ideia dos diagramas de influência é obtida da excelente série *Quality Software Management* de Gerald Weinberg, particularmente do Livro 1: *Systems Thinking*<sup>1</sup>. O propósito de um diagrama de influência é ver como os elementos de um sistema afetam um ao outro.

Diagramas de influência têm três elementos:

- Atividades, anotadas como uma palavra ou frase curta.
- Conexões positivas, anotadas como uma flecha direcionada entre duas atividades, significando que mais da atividade origem tende a criar mais da atividade destino, ou menos da atividade de origem tende a criar menos da atividade destino.
- Conexões negativas, anotadas como flechas direcionadas entre duas atividades com um círculo sobre ela, significando que mais da atividade de origem tende a criar menos da atividade destino, ou menos da atividade de origem tende a criar mais da atividade destino.

Um monte de palavras para um conceito simples. As Figuras A.1 a A.3 fornecem alguns exemplos.

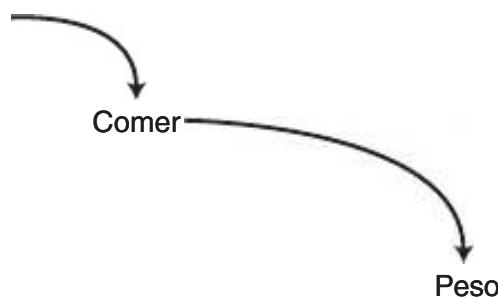
Quanto mais eu como, mais pesado eu fico. Quanto menos eu como, menos pesado eu fico. Peso pessoal é um sistema bem mais complicado que isso, certamente. Diagramas de influência são modelos para ajudar você a entender alguns aspectos do sistema, não entender e controlá-lo perfeitamente.

---

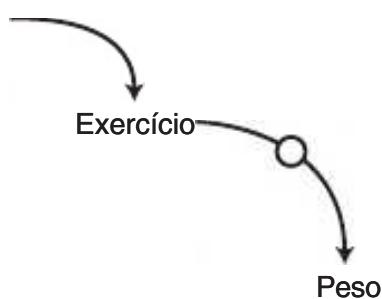
<sup>1</sup> Weinberg, Gerald. 1992. *Systems Thinking. Quality Software Management*. New York: Dorset House. ISBN: 0932633226.



**Figura A.1** *Duas atividades aparentemente não relacionadas.*



**Figura A.2** *Atividades positivamente conectadas.*



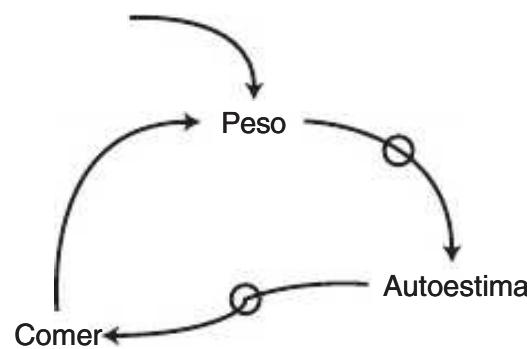
**Figura A.3** *Atividades negativamente conectadas.*

---

## Feedback

A influência não funciona em apenas uma direção. Frequentemente o efeito de uma atividade volta para mudar a própria atividade, positivamente ou negativamente, como mostrado na Figura A.4.

Se meu peso aumenta, então minha autoestima cai, o que me faz querer comer mais, o que faz meu peso aumentar, e assim por diante. Sempre que você tem um ciclo em um diagrama de influência, você tem feedback.



**Figura A.4 Feedback.**

Há dois tipos de feedback – positivo ou negativo. Feedback positivo causa sistemas que encorajam mais e mais de uma atividade. Você pode achar ciclos de feedback positivos contando o número de conexões negativas em um ciclo. Se há um número par de conexões negativas, então você tem um ciclo de feedback positivo. O ciclo de feedback na Figura A.4 é um ciclo de feedback positivo. Ele o obrigará a ficar ganhando peso até que a influência de alguma outra atividade tenha efeito.

Feedback negativo desencoraja ou reduz uma atividade. Ciclos com um número ímpar de conexões negativas são ciclos de feedback negativos.

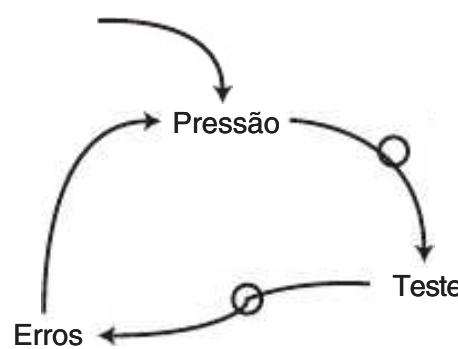
As chaves para o projeto de sistema são:

- Criar ciclos virtuosos cujos ciclos de feedback positivos encorajam o crescimento de boas atividades.
- Evitar espirais da morte cujos ciclos de feedback positivos encorajam o crescimento de atividades improdutivas ou destrutivas.
- Criar ciclos de feedback negativos para prevenir exploração de boas atividades.

## Controle de sistema

Quando escolher um sistema de práticas de desenvolvimento de software, você gostaria que as práticas oferecessem suporte uma à outra de modo que você tende a fazer a quantia certa de qualquer atividade, mesmo sob estresse. A Figura A.5 é um exemplo de um sistema de práticas que conduzem a teste insuficiente.

Sob a pressão do tempo, você reduz a quantidade de teste, o que aumenta o número de erros, o que aumenta a pressão do tempo. Eventualmente, alguma atividade de fora (como “Pânico no Fluxo de Caixa”) age para entregar o software de qualquer maneira.



**Figura A.5** A falta de tempo suficiente para teste reduz o tempo disponível.

Quando você tem um sistema que não está se comportando, você tem opções.

- Dircione um ciclo de feedback positivo para outra direção. Se você tem um ciclo entre testes e confiança, e testes têm falhado de modo a reduzir a confiança, então você pode fazer mais testes funcionarem para aumentar a confiança em sua habilidade de ter mais testes funcionando.
- Introduza um ciclo de feedback negativo para controlar uma atividade que cresceu demais.
- Crie ou quebre conexões para eliminar ciclos que não estão ajudando.

# APÊNDICE II

---

## Fibonacci

Em resposta a uma questão de um dos revisores deste livro, disponibilizo um Fibonacci dirigido por testes. Muitos revisores comentaram que esse exemplo ajudou a “cair a ficha” de como TDD funciona. Contudo, ele não é longo o suficiente, nem demonstra técnicas de TDD suficientes, para substituir os exemplos existentes. Se suas luzes ainda estão apagadas depois de ler os exemplos principais, dê uma olhada aqui e veja.

O primeiro teste mostra que  $\text{fib}(0) = 0$ . A implementação retorna uma constante.

```
public void testFibonacci() {  
    assertEquals(0, fib(0));  
}  
  
int fib(int n) {  
    return 0;  
}
```

(Eu apenas estou usando a classe `TestCase` como habitat para o código, pois essa é somente uma única função.)

O segundo teste mostra que  $\text{fib}(1) = 1$ .

```
public void testFibonacci() {  
    assertEquals(0, fib(0));  
    assertEquals(1, fib(1));  
}
```

Eu só pus a segunda asserção no mesmo método, porque lá não parecia estar qualquer valor de comunicação substancial para escrever `testFibonacci0f0neIsOne`.

Há muitos jeitos que eu poderia usar para fazer isso rodar. Escolherei tratar 0 como um caso especial:

```
int fib(int n) {
    if (n == 0) return 0;
    return 1;
}
```

A duplicação no caso de teste está começando a me incomodar, e ela só vai ficar pior conforme adicionarmos novos casos. Podemos fatorar a estrutura comum das assertões dirigindo o teste a partir de uma tabela de entrada e valores esperados.

```
public void testFibonacci() {
    int cases[][]= {{0,0},{1,1}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Agora adicionar o próximo caso requer seis teclas pressionadas e nenhuma linha adicional:

```
public void testFibonacci() {
    int cases[][]= {{0,0},{1,1},{2,1}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Desconcertantemente, o teste funciona. Acontece que nossa constante 1 está certa para esse caso também. No próximo teste:

```
public void testFibonacci() {
    int cases[][]= {{0,0},{1,1},{2,1},{3,2}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Iúpi, ele falha. Aplicando a mesma estratégia de antes (tratando entradas menores como casos especiais), escrevemos:

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 2;
}
```

Agora estamos prontos para generalizar. Escrevemos 2, mas não queremos dizer realmente 2, nós queremos dizer  $1 + 1$ .

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 1 + 1;
}
```

Aquele primeiro 1 é um exemplo de `fib(n-1)`:

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return fib(n-1) + 1;  
}
```

O segundo 1 é um exemplo de `fib(n-2)`:

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

Fazendo agora a limpeza, a mesma estrutura deve funcionar para `fib(2)`, logo podemos estreitar a segunda condição:

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

E aí temos Fibonacci, derivado totalmente de testes.

# Posfácio

*Martin Fowler*

Uma das coisas mais difíceis de comunicar sobre o desenvolvimento guiado por testes é o estado mental em que ele o coloca. Eu me lembro de uma sessão no projeto C3 original com Ralph Beattie em que tínhamos de implementar um conjunto complicado de condições de pagamento. Ralph os quebrou em um conjunto de casos de teste, e nós os configuramos um a um para fazê-los funcionar. O progresso foi uniforme e sem pressa; devido a não ter pressa, parecia lento, mas, olhando para trás para o quanto tínhamos pronto, era claro que, a despeito do sentimento de que não havia pressa, o progresso era realmente rápido.

Apesar de todas as ferramentas extravagantes que temos, programar ainda é difícil. Eu posso lembrar-me de muitas vezes em que estava programando e me sentia como se estivesse tentando manter muitas bolas no ar de uma só vez; qualquer lapso de concentração e tudo viria abaixo. Desenvolvimento guiado por testes reduz esse sentimento e, como resultado, você tem essa rápida redução de pressão.

Eu acho que a razão para isso é que trabalhar em um estilo de desenvolvimento guiado por testes dá a você essa sensação de manter apenas uma bola no ar por vez, então você pode se concentrar adequadamente naquela bola e fazer realmente um bom trabalho com ela. Quando estou tentando adicionar alguma nova funcionalidade, não estou preocupado com o que realmente faz um bom projeto para esse pedaço de função; estou apenas tentando fazer um teste passar tão facilmente quanto eu puder. Quando mudo para o modo de refatoração, não estou preocupado em adicionar alguma nova funcionalidade, apenas estou preocupado em obter o design certo. Com essas duas coisas, estou concentrado em uma coisa por vez e, como resultado, posso me concentrar melhor naquela única coisa.

Adicionar features testando primeiro e refatorando depois são dois desses sabores monológicos de programação. Em uma recente breve passagem pelo teclado, experimentei outro: o padrão cópia. Eu estava escrevendo um pequeno script em

Ruby que puxava alguns dados de um banco de dados. Conforme eu fazia isso, comecei em uma classe que empacotava a tabela do banco de dados e pensei comigo que, uma vez que recém tinha terminado um livro de padrões de banco de dados, eu deveria usar um padrão. Embora o código de exemplo fosse em Java, não era difícil adaptá-lo para Ruby. Enquanto programava isso, eu não pensei realmente sobre o problema; apenas pensei em fazer uma adaptação fiel do padrão para os dados específicos e linguagem que estava manipulando.

O padrão cópia em si não é boa programação – um fato que sempre friso quando falo sobre padrões. Padrões são sempre meio crus e precisam ser adaptados no forno do seu próprio projeto. Mas, uma boa maneira de fazer isso é copiar o padrão primeiro cegamente e então usar algum *mix* de refatoração ou teste-primeiro para fazer a adaptação. Desse jeito, quando você está fazendo uso do padrão cópia, você pode concentrar-se apenas no padrão – uma coisa por vez.

A comunidade XP batalhou para achar onde padrões se encaixam nesse cenário. Claramente, a comunidade XP é a favor dos padrões, afinal de contas há uma enorme intersecção entre os advogados de XP e os advogados de padrões – Ward e Kent foram líderes de ambos. Talvez o padrão cópia seja um terceiro modo monológico de seguir testando primeiro e refatorando, e, como esses dois, seja perigoso em si, mas poderoso quando combinado.

Uma grande parte de fazer atividades sistemáticas é identificar tarefas centrais e nos permitir concentrar em apenas uma de cada vez. Uma linha de montagem é um exemplo entorpecedor disso – entorpecedor, pois você apenas faz uma coisa. Talvez o que desenvolvimento guiado por testes sugira é uma forma de desmembrar o ato de programar em modos elementais, mas evitando a monotonia, trocando rapidamente entre esses modos. A combinação de modos monológicos e essas trocas dá a você os benefícios de concentração e baixo estresse no cérebro, sem a monotonia da linha de montagem.

Eu admito que essas reflexões estão de alguma forma meio cruas. Conforme escrevo isto, estou ainda incerto se acredito no que digo, e sei que estarei ruminando esses pensamentos por alguns ou muitos meses. Mas, achei que você poderia gostar deles de qualquer jeito, talvez para estimular suas reflexões no cenário maior em que desenvolvimento guiado por testes se encaixa. Não é um cenário que vemos claramente ainda, mas acho que lentamente ele está se revelando.

# Índice

## A

Acoplamento de teste, 118-119  
Adição, escrevendo/testando código para o processo de adicionando parâmetros, 87-91  
  declarações, fazendo outras mudanças se propagarem, 93-100  
  eliminando duplicação de dados, 87-91  
  implementando/movendo código, 81-85  
  metáforas, 76-79  
Asserção (Assertion), padrão de projeto, 177-179  
ATDD (desenvolvimento guiado por testes de aplicação), *versus* TDD, 219  
AutoDesvio (Self Shunt), Testes, 165-166

## B

Barra verde, padrões de projeto de faz de conta, 171-173  
Implementação Óbvia (Obvious Implementation), 174-175  
Migração de dados em Um para Muitos (One to Many), 203-204  
Triangulação, 173-174  
Um para Muitos (One to Many), 174-176  
Barra vermelha, padrões de projeto de  
  Teste de Aprendizado (Learning Test), 156-157  
  Teste de Explicação (Explanation Test), 155-156  
  Teste de Regressão (Regression Test), 157-158

Teste de Um Só Passo (One Step Test), 153-155  
Teste Inicial (Starter Test), 154-155

## C

Check-in Limpo (Clean Check-in), 169-170  
Classes concretas, 66-67  
Classes métricas de código, 104-105  
  concretas, 66-67  
  definindo, 25-26  
  eliminando verificação explícita de classes com polimorfismo, 81-85  
  subclasses, 47-51, 56-57  
  subclasses, eliminação de, 59-63  
  subclasses, substituindo referências por referências de superclasses, 71-73  
  superclasses, 47-51  
  teste de igualdade, 35-38, 47-51  
Código funcionando, *versus* código de teste, 28, 29  
Collecting Parameter, padrão de projeto, 134-135, 186, 198-199  
Command, padrão de projeto, 185-187  
Complexidade ciclomática, métricas de código, 104-105  
Composite, padrão de projeto, 133-137, 186, 196-198  
  e Impostors, 196  
Conjuntos para múltiplos testes  
  AllTests, 182-184  
  TestSuite, 133-137

## D

Dados de Teste (Test Data), 148-150  
Dados Evidentes (Evident Data), 149-151  
Dados Realistas, 149-150  
Defina uma Asserção Primeiro (Assert First), Teste, 147-149  
Dependência no código, e duplicação de código, 27, 28  
Desenvolvimento guiado por Testes.  
  *Veja TDD*  
Desenvolvimento guiado por testes de aplicação (ATDD), *versus* TDD, 219  
Diagramas de influência, 143-144, 146-147  
  elementos, 227-228  
  feedback, 228-229  
  sistemas de práticas de desenvolvimento, 229-230  
Dinheiro multi-moeda, criação do objeto, 23-30  
Duplicação de código e dependência de código, 27, 28  
  removendo, 44, 45

## E

Estresse em testes  
  diagramas de influência, 143-144  
  *versus* TDD, 105-107  
Exceções  
  manipulando exceções, 125-127  
  Teste de Exceção (Exception Test), 182-183  
Extrair Interface (Extract Interface) e Testes AutoDesvio (Self Shunt), 166

Extrair Método (Extract Method), 115  
*refatoração*, 203-205  
*versus* Objeto Método (Method Object), 209  
 Extrair Objetos (Extract Object), 203

**F**

Factory Methods, 56-57  
*padrões de projeto*, 186, 194-195  
 Falhas em testes  
   framework xUnit, 129-131  
   processo de multiplicação, 24-27  
   Tempo Médio Entre Falhas, TMEF (Mean Time Between Failures - MTBF), 216-217  
 Fazer de conta implementações de TDD, 33, 126, 158  
   Modelo de Teste de Acidentes (Crash Test Dummy), 167  
   *padrões de projeto de barra verde*, 171-173  
 Feedback  
   diagramas de influência, 228-229  
   quantidade necessária em TDD, 216-217  
 Feedback negativo, diagramas de influência de, 228-229  
 Feedback positivo, diagramas de influência de, 228-229  
 Fibonacci, 231-233  
 Flags de métodos chamados, 111-120  
   *versus* registros, 121  
 Funções, métricas de código, 104-105

**I**

Igualdade, teste de, 35-38, 47-51, 53-54  
 Implementação de triangulação de TDD, 33, 36-38, 158  
   *padrões de projeto de barra verde*, 173-174  
 Implementação Óbvia (Obvious Implementation) de TDD, 33, 123, 158  
   *padrões de barra verde*, 174-175  
 Implementações de TDD  
   Faz de conta (Fake It), e Modelo de Teste de Acidentes (Crash Test Dummy), 167  
   Faz de conta (Fake It), *padrões de projeto de barra verde*, 171-173  
   Faz de conta, 33, 123, 158  
   Implementação Óbvia (Obvious Implementation), *padrões de barra verde*, 174-175  
   Óbvio, 33, 123, 158  
   Triangulação, 33, 36-38, 158  
   Triangulação, *padrões de projeto de barra verde*, 173-174

Imposter, padrão de projeto, 186, 195-196  
 Isolar Mudança (Isolate Change) *refatoração*, 202-203  
 Um para Muitos (One to Many), *padrões de projeto*, 174-176

**J**

JProbe, 105-106  
 JUnit, usando, 103  
 JJUnit, 178-179

**L**

Lista de Testes (Test List), 146-147  
 Listas de Tarefas, 24

**M**

Mensagens de erro, 67-68  
 Métaforas, 76-79  
   vantagens, 102-103  
 Método em uma Linha (Inline Method), 205-206  
 Metodologia do Chuveiro, 158  
 Métodos  
   Extrair Método (Extract Method), *refatoração*, 203-205  
   Extrair Método (Extract Method), *versus* Objeto Método (Method Object), 209  
   Factory Methods, 56-57, 186, 194-195  
   Método em uma Linha (Inline Method), 205-206  
   Mover Método (Move Method), 207-209  
   parâmetros a métodos, adicionando, 210-211  
   parâmetros de métodos, mudando, 210-211  
   reconciliação, 65-66-69  
   setUp(), 117-120  
   tearDown(), 121-127  
   Template Method, 166, 186, 190-192  
   testMethod(), 111-115  
 Métricas de código, 104-105  
 Modelo de Teste de Acidentes (Crash Test Dummy), 167-168  
   extraindo interfaces, 207  
 Multiplicação, escrevendo/testando  
   código para o processo de, 27, 31  
   ciclo de teste, 21, 27, 31, 104-105  
   ciclo de teste, decisões nos passos iniciais, 24-26  
   ciclo de testes, escopo, 25-26, 43-45  
   ciclo de testes, falhas, 24-27  
   escrevendo testes copiando/colando código, 43-45  
   implementações, Triangulação, 33, 36-38  
   implementações Faz de conta, 33  
   implementações Óbviias, 33

implementações redundantes, 51  
 implementações redundantes, eliminando, 71-73  
 melhorando testes com funcionalidade, 39-41  
 métodos fábrica, 56-57  
 refatoração de código, 60-63  
 teste de igualdade, 35-38, 47-51, 53-54

**O**

Objeto Método (Method Object), *versus* Extrair Método (Extract Method), 209  
 Objetos  
   criando, 55-57  
   criando, dinheiro multi-moeda, 23-30  
   criando, pela força dos testes, 81-85  
   criando, restrições causando conflitos, 117-120  
 Null Object, *padrões de projeto*, 186, 189-190  
 Null Object, *padrões de projeto*, e Impostors, 196  
 Objeto Simulado (Mock Object), 164-165  
 Objeto Simulado (Mock Object), e extração de interfaces, 207  
 Objeto Simulado (Mock Object), e Modelo de Teste de Acidentes (Crash Test Dummy), 168  
 Princípio Aberto/Fechado, 215-216  
 Value Objects, 35-38

**P**

Padrão Arrange/Aja/Asserções (3A), 117-118  
 Padrões de projeto  
   AutoDesvio (Self Shunt), testes, 165-166  
   Check-in Limpo (Clean Check-in), 169-170  
   Collecting Parameters, 134-135, 186, 198-199  
   Command, 185-187  
   Composite, 133-137, 186, 196-198  
   Factory Methods, 186, 194-195  
   Impostors, 186, 195-196  
   Modelo de Teste de Acidentes (Crash Test Dummy), 167-168  
   Null Objects, 186, 189-190  
   Objeto Simulado (Mock Object), 207  
   panorama, 185-186  
   Pluggable Objects, 186, 191-193  
   Pluggable Selectors, 186, 193-194  
   Singletons, 199  
   String de Registro (Log String), 166-167

- Template Method, 186, 190-192  
 Teste Filho (Child Test), 163  
 Teste Quebrado (Broken Test), 168-169  
 Value Objects, 185-189  
 Padrões de projeto, framework xUnit  
   Asserções, 177-179  
   Fixtures, 178-180  
   Fixtures Externas, 180-181  
   Método de Teste (Test Method), 181-183  
   Teste de Exceção (Exception Test), 182-183  
   Todos os Testes (All Tests), 182-184  
 Padrões de projeto, padrões de barra verde  
   Faça de conta (Fake It), 171-173  
   Implementação Óbvia (Obvious Implementation), 174-175  
   Triangulação, 173-174  
   Um para Muitos (One to Many), 174-176, 203-204  
 Padrões de projeto, padrões de barra vermelha  
   Teste de Aprendizado (Learning Test), 156-157  
   Teste de Explicação (Explanation Test), 155-156  
   Teste de Regressão (Regression Test), 157-158  
   Teste de Um Só Passo (One Step Test), 153-155  
   Teste Inicial (Starter Test), 154-155  
 Pluggable Objects, padrões de projeto, 186, 191-193  
 Pluggable Selectors, 115  
   padrões de design, 186, 193-194  
 Polimorfismo, eliminando verificação explícita de classe por, 81-85  
 Princípio Aberto/Fechado (objetos), 215-216  
 Privadas, variáveis de instância, 40-41  
 Problemas de inicialização, método de teste, 111-115  
 Programação em pares, configuração física da, 160-161  
 Programação Extrema (XP), *versus* TDD, 224-225
- R**
- Refatoração de código, 60-63  
   extraíndo interfaces, 207  
 Extrair Método (Extract Method), 203-205  
 Extrair Objetos (Extract Object), 203  
 Isolar Mudança (Isolate Change), 202-203
- Método em uma Linha (Inline Method), 205-206  
 migração de dados, Migrar Dados (Migrate Data) 203-204  
 Mover Método (Move Method), 207-209  
 Objeto Método (Method Object), 203, 209-211  
 parâmetros a métodos, adicionando, 210-211  
 parâmetros de método, mudando, 210-211  
 Reconciliar Diferenças (Reconcile Differences), 201-202  
 Registro de métodos chamados, 121-127  
   *versus* flags, 121
- S**
- Scripts com o framework de testes xUnit, 140  
 SetUp(), método, 117-120  
 Singleton, padrões de projeto, 199  
 SmallLint para SmallTalk, 102-103  
 String de Registro (Log String), 166-167  
 Subclasses, 47-51, 56-57  
   eliminando, 59-63  
   substituindo referências por referências de superclasses, 71-73  
 Suíte de testes AllTests, 182-184  
 Superclasses, 47-51  
   substituindo referências para subclasses, 71-73
- T**
- TDD  
   análise do termo, 223-225  
   apagando testes, 218  
   atributos de testes eficazes, 214-216  
   aumentando o alcance de TDD, 225  
   código reusável, 215-216  
   começando práticas de TDD no meio de projetos, 219-220  
   decisões do conteúdo para testar, 214-215  
   escalando para grandes sistemas, 218-219  
   feedback necessário, quantidade de, 216-217  
   Fibonacci, 231-233  
   influência de linguagens e ambientes de programação, 218  
   JProbe, 105-106  
   Princípio Aberto/Fechado (objetos), 215-216  
   qualidade de teste, 105-107  
   razões para efetividade, 221-223  
   sequência de teste, 221  
   SmallLint para SmallTalk, 102-103
- Tempo Médio Entre Falhas, 216-217  
 teste, definição, 143  
 usuários potenciais, 220-221  
*versus* desenvolvimento dirigido por testes de aplicação, 219  
*versus* outros estilos de programação, 97-100  
*versus* outros tipos de teste, 105-107  
*versus* Programação Extrema, 224-225  
 TDD, ciclo de teste, 21, 27, 31, 104-105  
   passos, tamanho dos, 213-215  
   passos, passos iniciais, 24-26  
   escopo, 25-26, 43-45  
   falhas, 24-27  
 TDD, diretrizes de  
   começar de novo, Faça de Novo (Do Over), 159-160  
   configuração física, 160-161  
   faça Pausa (Break), 158-159  
 TDD, padrões de projeto para, 221-222  
   básico, 143-145  
   Check-in Limpo (Clean Check-in), 169-170  
   Collecting Parameter, 134-135  
   Collecting Parameters, 186, 198-199  
   Command, 185-187  
   Composite, 133-137, 186, 196-198  
   Dados de Teste (Test Data), 148-150  
   Dados Evidentes (Evident Data), 149-151  
   Dados Realistas, 149-150  
   Defina uma Asserção Primeiro (Assert First), 147-149  
   Factory Methods, 186, 194-195  
   Imposters, 186, 195-196  
   Lista de Testes (Test List), 146-147  
   Modelo de Teste de Acidentes (Crash Test Dummy), 167-168  
   Modelo de Teste de Acidentes (Crash Test Dummy), extraíndo interfaces, 207  
   Null Objects, 186, 189-190  
   Objeto Simulado (Mock Object), 164-165  
   panorama, 185-186  
   Pluggable Objects, 186, 191-193  
   Pluggable Selectors, 186, 193-194  
   Self Shunt, Testes, 165-166  
   Singletons, 199  
   String de Registro (Log String), 166-167  
   Template Method, 186, 190-192  
   Teste Filho (Child Test), 163