

Test-Driven Development

Teste e Design no Mundo Real com Ruby



 Casa do
Código

HUGO CORBUCCI
MAURICIO ANICHE

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

Agradecimentos

Aos que não leram outras versões desse livro, Mauricio escreveu o livro originalmente em Java e C# e Hugo ajudou na “tradução” para Ruby. Os agradecimentos estão divididos pois foram escritos em momentos diferentes.

Agradecimentos do Mauricio

Essa talvez seja a seção mais difícil de se escrever, pois a quantidade de pessoas que participaram direta ou indiretamente do livro é muito grande.

Vou começar agradecimento a meu pai, mãe e irmão, que a todo momento me apoiaram na decisão de fazer um mestrado, entender como ciência deve ser feita, e que sofreram junto comigo nos momentos de grande estresse (que todo mestrado proporciona!).

Agradeço também ao meu orientador de mestrado e doutorado, prof. Dr. Marco Aurelio Gerosa, que me ensinou como as coisas funcionam “do lado de lá”. Sem ele, acho que esse livro seria muito diferente; seria mais apaixonado, porém menos verdadeiro. Se meu texto olha TDD de maneira fria e imparcial, a culpa é dele.

Os Srs. Paulo Silveira e Adriano Almeida também merecem uma lembrança. Mesmo na época em que a Casa do Código não existia de fato, eles já haviam aceitado a ideia do livro de TDD. Obrigado pela confiança.

Todas as pessoas das últimas empresas em que atuei também me ajudaram muito com as incontáveis conversas de corredor sobre o assunto. Isso com certeza enriqueceu muito o texto.

Agradeço também aos amigos José Donizetti, Guilherme Moreira e Rafael Ferreira, que gastaram tempo lendo o livro e me dando sugestões de como melhorar.

Por fim, obrigado a você que está lendo esse livro. Espero que ele ajude.

Agradecimentos do Hugo

Seções de agradecimentos são sempre injustas já que é impossível listar todos que contribuíram de alguma forma para um trabalho ou o tamanho de suas contribuições. No entanto, vou tentar fazer a minha parte e já peço desculpas pelos que não pude mencionar.

Agradeço aos profs. Drs. Fabio Kon e Alfredo Goldman que, durante o curso de Bacharelado em Ciências da Computação (BCC) da USP, me apresentaram a técnica de TDD entre muitas outras no curso de Programação Extrema (XP). Ao Danilo Sato que foi nosso *coach* na disciplina e nos anos seguintes e todos os outros membros da AgilCoop que muito contribuíram no meu aprendizado de métodos ágeis.

Obrigado ao Paulo Silveira e Adriano Almeida pelas oportunidades com a Casa do Código e pela paciência com minha demora para escrever. Junto com eles, todos os amigos da Caelum que, apesar de não terem sido colegas de trabalho diretamente, sempre foram uma parte importante das conversas e aprendizados.

Muitos agradecimentos aos irmãos Alexandre Freire e Fernando Freire pelo nosso longo trabalho junto durante o qual comecei a trabalhar em Ruby. Nesse tempo juntos evoluímos nosso conhecimento e prática de TDD. Em especial, as diferenças entre TDD em linguagens mais dinâmicas como Ruby e linguagens mais estáticas como Java. À Mariana Bravo não poderei fazer justiça mas, além de tudo que os irmãos Freire fizeram, pelas revisões, discussões, ajudas, paciência, amor e carinho.

Agradeço também ao Mauricio obviamente pela oportunidade de “traduzir” o livro mas também pelas incontáveis e intermináveis discussões. Discussões sobre TDD, mocks, modelagem, linguagens, padrões e basicamente qualquer coisa que podemos encontrar para discordar nos ajudaram a enriquecer nossos argumentos e entendimentos sobre desenvolvimento.

Finalmente, a todos que têm paixão pelo seu trabalho e procuram melhorar o mundo com artigos, posts, livros, código aberto, palestras e voluntariado, continuem fazendo o que gostam e não deixem as pessoas ao seu redor se acomodarem.

Quem somos nós?

Como todo indivíduo, cada um de nós é único. No entanto, Mauricio e Hugo compartilham algumas coisas. Entre elas, ambos trabalham com desenvolvimento de software há perto de uma década, estão ligados com o Instituto de Matemática e Estatística (IME) da Universidade de São Paulo (USP) e têm opiniões fortes (e frequentemente divergentes) sobre desenvolvimento de software.

O GRANDE HUGO CORBUCCI

Java e .NET são linguagens/plataformas com que eu tive um grande contato. Com Ruby, ao contrário, tive muito pouca experiência. E dado o sucesso que o livro fez em ambas as linguagens, muitos desenvolvedores da comunidade Ruby me pediram uma versão do livro.

Não tive dúvidas em convidar o Hugo para enfrentar esse desafio. Deixando a nossa amizade de lado, o Hugo é um desenvolvedor de quem eu sou fã. É um desenvolvedor com muita experiência e opiniões fortes. Perdi a conta de quantas vezes tivemos infinitas discussões sobre TDD, design etc. E, com certeza, aprendi muito com todas elas.

Por isso, a minha total confiança nele.

Mauricio Aniche

Em boa parte dos 10 anos de experiência que acumulei até a escrita desse livro, atuei como consultor para diferentes empresas do mercado brasileiro e internacional. Com certeza, as linguagens mais utilizadas por mim ao longo

da minha carreira foram Java, C# e C.

Como sempre pulei de projeto em projeto (e, por consequência, de tecnologia em tecnologia), nunca fui a fundo em nenhuma delas. Pelo contrário, sempre foquei em entender princípios que pudessem ser levados de uma para outra, para que no fim, o código saísse com qualidade, independente da tecnologia.

Em meu último ano da graduação, 2007, comecei a ler mais sobre testes automatizados e TDD. Achei muito interessante e útil a ideia de se escrever um programa para testar seu programa, e decidi praticar TDD, por conta própria, para entender melhor como ela funcionava.

Gostei muito do que vi. De 2007 em diante, resolvi praticar, pesquisar e divulgar melhor minhas ideias sobre o assunto. Comecei devagar, apenas blogando o que estava na minha cabeça e sobre o que gostaria de *feedback* de outros desenvolvedores. Mas para fazer isso de maneira mais decente, resolvi ingressar no programa de Mestrado da Universidade de São Paulo. Lá, pesquisei sobre os efeitos da prática de TDD no design de classes.

Ao longo desse tempo participei da grande maioria dos eventos relacionados ao assunto. Palestrei nos principais eventos de métodos ágeis do país (como Agile Brazil, Encontro Ágil), de desenvolvimento de software (QCON SP e DNAD), entre outros menores. Cheguei a participar de eventos internacionais também; fui o único palestrante brasileiro no Primeiro Workshop Internacional sobre TDD, em 2010, na cidade de Paris. Isso mostra também que tenho participado dos eventos acadêmicos. Em 2011, apresentei um estudo sobre TDD no WBMA (Workshop Brasileiro de Métodos Ágeis), e em 2012, no maior simpósio brasileiro sobre engenharia de software, o SBES.

Atualmente trabalho pela Caelum, como consultor e instrutor. Também sou aluno de doutorado pela Universidade de São Paulo, onde continuo a pesquisar sobre a relação dos testes de unidade e qualidade do código.

Portanto, esse é meu relacionamento com TDD. Nos últimos anos tenho olhado ele de todos os pontos de vista possíveis: de praticante, de acadêmico, de pesquisador, de apaixonado, de frio. Esse livro é o relato de tudo que aprendi nesses últimos anos.

Hugo Corbucci

Lá na década de 1990, eu escrevi meus primeiros trechos de código. Começando com Visual Basic, PHP, Asp (não .Net, o anterior), shell (do DOS e bash) e Javascript, eu não aprendi muito sobre qualidade de código. O que importava era que funcionasse exatamente naquela hora. Era super divertido mas eu sempre me senti limitado.

Em 2005, ao participar do 7º Fórum Internacional de Software Livre (FISL), resolvi responder, com vários amigos, ao chamado de John “Maddog” Hall para desenvolver um sistema livre de Desenho Auxiliado por Computador (*Computer Aided Design* ou CAD). O trabalho teve início na matéria de Laboratório de Programação Extrema do curso de computação do IME-USP.

Foi meu primeiro projeto com mais de 5 desenvolvedores envolvidos, com mais de 10000 linhas de código Java e a necessidade de funcionar em computadores de pessoas completamente desconhecidas. Esse projeto (Archimedes-CAD) não teria conseguido sobreviver seus 6 anos de desenvolvimento ativo com mais de 50 pessoas diferentes envolvidas sem ter sido iniciado com TDD.

Pouco depois, Danilo Sato trouxe da conferência XP 2005 uma ideia de *Coding Dojo* [8]. A ideia de praticar programação com TDD trouxe muitas pessoas juntas e resultou no *Coding Dojo São Paulo* no qual participei de em torno de 100 sessões de dojo, sempre com TDD.

Junto com o dojo, comecei a participar de conferências nacionais e internacionais e passei a organizar algumas como Encontro Ágil e Agile Brazil. Graças a elas, conheci muitos praticantes de TDDs e variei minhas técnicas e decisões de design.

Atualmente, trabalho como consultor na ThoughtWorks em Chicago. Desde de 2011, participo de projetos Ruby e Rails nos quais pratico TDD diariamente.

Minha experiência com TDD se resume então em práticas pontuais (dojos), projetos novos e curtos em Java e Ruby (3 meses), projetos novos e longos em Java (mais de 4 anos) e projetos velhos e longos em Ruby (mais de 7 anos). Ainda não achei um projeto no qual TDD não me ajude a desenvolver código mais sustentável de forma eficiente.

Prefácio

TDD é uma das práticas de desenvolvimento de software sugeridas por diversas metodologias ágeis, como XP. A ideia é fazer com que o desenvolvedor escreva testes automatizados de maneira constante ao longo do desenvolvimento. Mas, diferentemente do que estamos acostumados, TDD sugere que o desenvolvedor escreva o teste antes mesmo da implementação.

Essa simples inversão no ciclo traz diversos benefícios para o projeto. Baterias de testes tendem a ser maiores, cobrindo mais casos, e garantindo uma maior qualidade externa. Além disso, escrever testes de unidade forçará o desenvolvedor a escrever um código de maior qualidade pois, como veremos ao longo do livro, para escrever bons testes de unidade, o desenvolvedor é obrigado a encapsular bem o código escrito e definir bem as interações possíveis com aquele código. Em linguagens orientadas a objeto, como Ruby, esses benefícios se apresentam como melhorias no design dos objetos.

A prática nos ajuda a escrever um software melhor, com mais qualidade, e um código melhor, mais fácil de ser mantido e evoluído. Esses dois pontos são importantíssimos em qualquer software, e TDD nos ajuda a alcançá-los. Toda prática que ajuda a aumentar a qualidade do software produzido deve ser estudada.

Neste livro, tentei colocar toda a experiência e tudo que aprendi ao longo desses últimos anos praticando e pesquisando sobre o assunto. Mostrei também o outro lado da prática, seus efeitos no design de classes, que é muito falada mas pouco discutida e explicada. A prática de TDD, quando bem usada, pode ser bastante produtiva. Mas, como se verá ao longo do livro, os praticantes devem estar sempre alertas às dicas que o teste dará sobre nosso código. Aqui, passaremos por eles e o leitor ao final do livro terá em mãos uma nova

e excelente ferramenta de desenvolvimento.

A quem se destina esse livro?

Esse livro é destinado a desenvolvedores que querem aprender a escrever testes de maneira eficiente, e que desejam aprender a como melhorar ainda mais o código que produzem. Neste livro, utilizamos Ruby para demonstrar os conceitos discutidos, mas você pode facilmente levar as discussões feitas aqui para a sua linguagem de programação favorita. Mesmo que você já pratique TDD, tenho certeza que aqui encontrará discussões interessantes sobre como a prática dá *feedback* sobre problemas de acoplamento e coesão, bem como técnicas para escrever testes melhores e mais fáceis de serem mantidos.

Testadores também podem se beneficiar deste livro, entendendo como escrever códigos de teste de qualidade, quando ou não usar TDD, e como reportar problemas de código para os desenvolvedores.

Como devo estudar?

Ao longo do livro, trabalhamos em diversos exemplos, muito similares ao mundo real. Todo capítulo possui sua parte prática e parte teórica. Na parte prática, muito código de teste é escrito. Na parte teórica, refletimos sobre o código que produzimos até aquele momento, o que foi feito de bom, o que foi feito de ruim, e melhoramos de acordo.

O leitor pode refazer todos os códigos produzidos nos capítulos. Praticar TDD é essencial para que as ideias fiquem naturais. Além disso, a Caelum também disponibiliza um curso online sobre testes automatizados [10], que pode ser usado como complemento desse livro.

Boa leitura!

Por que uma versão em Ruby?

Essa é a terceira versão deste livro (Java e .NET já foram lançadas). Por que então decidimos escrever uma versão para Ruby?

Ruby tem algumas diferenças importantes com relação a Java e C#. O traço comum é a orientação a objetos que todas essas linguagens seguem. Isso significa que, nas três linguagens, podemos criar objetos, isto é, elementos que acoplam dados (atributos) e comportamentos (métodos). Além disso, Ruby, Java e C# são linguagens fortemente tipadas, isto é, todo objeto tem um tipo e esse tipo não pode mudar.

No entanto, Ruby é uma linguagem dinamicamente tipada, ao contrário de Java e C#, que são estaticamente tipadas. A diferença é que, em Java e C#, uma variável é declarada de um determinado tipo, que não pode mudar. Em Ruby, variáveis podem ser atribuídas dinamicamente, ou seja, sem a necessidade de explicitar o tipo no momento de sua criação. A vantagem é que reduz a verbosidade e fica mais fácil definir variáveis e comportamentos de certos objetos. A desvantagem é que não ficam explicitamente registrados no código os tipos das variáveis utilizadas, apesar de o interpretador fazer a verificação, pois, assim como Java e C#, Ruby é fortemente tipada. Veremos ao longo do livro que isso faz com que, em Ruby, ocorra algumas particularidades no momento de escrever testes automatizados, comparado a Java ou C#, pois podem ser atribuídas e reatribuídas dinamicamente.

Outra característica importante de Ruby é a facilidade com a qual a linguagem permite o programador utilizar o que é chamado de meta-programação. Ruby permite ao programador escrever código que gera código de forma muito fácil. Por exemplo:

```
class Exemplo
```

```
def um_getter
  @um_getter
end
def um_setter=(valor)
  @um_setter = valor
end
def ambos
  @ambos
end
def ambos=(valor)
  @ambos = valor
end
end
```

Esse código que tem um *getter* para o atributo `@um_getter`, um *setter* para o atributo `@um_setter` e um *getter* e um *setter* para o atributo `@ambos`, pode ser escrito em Ruby da seguinte forma:

```
class Exemplo
  attr_reader :um_getter
  attr_writer :um_setter
  attr_accessor :ambos
end
```

`attr_reader`, `attr_writer` e `attr_accessor` são métodos que geram *getters*, *setters* e *ambos* respectivamente. Ou seja, com essas 3 linhas, geramos o código que tinha sido escrito antes com 12 linhas. Existem muitas outras formas de tirar proveito dessa metaprogramação simples em Ruby conforme veremos ao longo do livro.

Essas várias diferenças fazem com que o desenvolvimento em Ruby costume ser mais denso e fácil de se cometer erros bobos que só são descobertos ao executar aquele trecho exato de código. De uma certa forma, isso faz com que seja ainda mais importante fazer bom uso de testes automatizados e, provavelmente, de TDD para garantir o sucesso do seu código Ruby.

Por esses vários motivos e para ajudar todos aqueles que têm curiosidade com relação a Ruby, o Hugo e eu decidimos reformular os exemplos do livro para uma versão Ruby. Você verá que os códigos são um pouco diferentes dos

encontrados nas outras versões deste livro e assim como alguns conselhos e preocupações.

Esperamos que esse material seja bastante útil para toda a comunidade Ruby.

Sumário

1	Introdução	1
1.1	Era uma vez um projeto sem testes...	1
1.2	Por que devemos testar?	2
1.3	Por que não testamos?	3
1.4	Testes automatizados e TDD	3
1.5	Conclusão	4
1.6	Como tirar dúvidas?	5
2	Testes de Unidade	7
2.1	O que é um teste de unidade?	7
2.2	Preciso mesmo escrevê-los?	8
2.3	O primeiro teste de unidade	10
2.4	Continuando a testar	17
2.5	Conclusão	18
3	Introdução ao <i>Test-Driven Development</i>	21
3.1	O problema dos números romanos	22
3.2	O primeiro teste	23
3.3	Refletindo sobre o assunto	31
3.4	Quais as vantagens?	32
3.5	Um pouco da história de TDD	34
3.6	Conclusão	35

4	Simplicidade e <i>Baby Steps</i>	37
4.1	O problema do cálculo de salário	38
4.2	Implementando da maneira mais simples possível	38
4.3	Passos de Bebê (ou <i>Baby Steps</i>)	41
4.4	Usando <i>baby steps</i> de maneira consciente	47
4.5	Conclusão	51
5	TDD e design de classes	53
5.1	O problema do carrinho de compras	53
5.2	Testes que influenciam no design de classes	59
5.3	Diferenças entre TDD e testes da maneira tradicional	60
5.4	Testes como rascunho	62
5.5	Conclusão	63
6	Qualidade no código do teste	65
6.1	Repetição de código entre testes	66
6.2	Nomenclatura dos testes	69
6.3	<i>Test Data Builders</i>	70
6.4	Testes repetidos	72
6.5	Escrevendo boas asserções	76
6.6	Testando listas	79
6.7	Separando as classes de teste	81
6.8	Conclusão	82
7	TDD e a coesão	83
7.1	Novamente o problema do cálculo de salário	84
7.2	Ouvindo o feedback dos testes	88
7.3	Testes em métodos privados?	90
7.4	Resolvendo o problema da calculadora de salário	91
7.5	O que olhar no teste em relação a coesão?	95
7.6	Conclusão	96

8	TDD e o acoplamento	97
8.1	O problema da nota fiscal	98
8.2	<i>Mock Objects</i>	100
8.3	Dependências explícitas	104
8.4	Ouvindo o feedback dos testes	105
8.5	Classes estáveis	106
8.6	Resolvendo o problema da nota fiscal	110
8.7	Testando métodos estáticos	113
8.8	TDD e a constante criação de interfaces	115
8.9	O que olhar no teste em relação ao acoplamento?	118
8.10	Conclusão	119
9	TDD e o encapsulamento	121
9.1	O problema do processador de boleto	122
9.2	Ouvindo o feedback dos testes	126
9.3	<i>Tell, Don't Ask</i> e Lei de Demeter	127
9.4	Resolvendo o problema do processador de boletos	129
9.5	O que olhar no teste em relação ao encapsulamento?	130
9.6	Conclusão	131
10	Testes de integração e TDD	133
10.1	Testes de unidade, integração e sistema	133
10.2	Quando não usar <i>mocks</i> ?	135
10.3	Testes em subclasses de ActiveRecord::Base	137
10.4	Devo usar TDD em testes de integração?	140
10.5	Testes em aplicações Web	141
10.6	Conclusão	143
11	Testando código dinâmico	145
11.1	É possível utilizar TDD com <i>Duck Typing</i> ?	146
11.2	A mágica do <i>send</i> e seus testes	148
11.3	O uso e teste de <i>method_missing</i>	154

12 Quando não usar TDD?	157
12.1 Quando não praticar TDD?	157
12.2 100% de cobertura de código?	159
12.3 Devo testar códigos simples?	159
12.4 Erros comuns durante a prática de TDD	160
12.5 Como convencer seu chefe sobre TDD?	162
12.6 TDD em sistemas legados	164
12.7 Conclusão	166
13 E agora?	167
13.1 Como aprender mais agora?	167
13.2 Dificuldade no aprendizado	168
13.3 Como interagir com outros praticantes?	169
13.4 Conclusão final	169
14 Apêndice: princípios SOLID	171
14.1 Sintomas de projetos de classes em degradação	171
14.2 Princípios de projeto de classes	175
14.3 Conclusão	178
15 Apêndice: RSpec como biblioteca de testes de unidade	181
15.1 Asserções na primeira versão do RSpec	182
15.2 Biblioteca de <i>mocks</i> embutida no RSpec	183
15.3 Comparando a saída das duas bibliotecas	183
15.4 Novas asserções do RSpec	187
15.5 Conclusão	187
Índice Remissivo	188
Bibliografia	191

CAPÍTULO 1

Introdução

Será que testar software é realmente importante? Neste capítulo, discutimos um pouco sobre as consequências de software não testado e uma possível solução para isso, que hoje é um problema para a sociedade como um todo, já que softwares estão em todos os lugares.

1.1 ERA UMA VEZ UM PROJETO SEM TESTES...

Durante os anos de 2005 e 2006, trabalhei em um projeto cujo objetivo era automatizar todo o processo de postos de gasolina. O sistema deveria tomar conta de todo o fluxo: desde a comunicação com as bombas de gasolina, liberando ou não o abastecimento, até relatórios de fluxo de caixa e quantidade de combustível vendido por dia ou por bomba.

A aplicação era desenvolvida inteira em C e deveria rodar em um microdispositivo de 200MHz de processamento e 2MB de RAM. Nós éramos em

4 desenvolvedores. Todos programavam e todos testavam ao mesmo tempo. Os testes não eram tão simples de serem feitos, afinal precisávamos simular bombas de gasolinas, abastecimentos simultâneos etc. Mas, mesmo assim, nós nos revezávamos e testávamos da forma que conseguíamos.

Após alguns meses de desenvolvimento, encontramos o primeiro posto disposto a participar do piloto da aplicação. Esse posto ficava em Santo Domingo, capital da República Dominicana.

Eu, na época líder técnico dessa equipe, viajei para o lugar com o objetivo de acompanhar nosso produto rodando pela primeira vez. Fizemos a instalação do produto pela manhã e acompanhamos nosso “bebê” rodando por todo o dia. Funcionou perfeitamente. Saímos de lá e fomos jantar em um dos melhores lugares da cidade para comemorar.

Missão cumprida.

1.2 POR QUE DEVEMOS TESTAR?

Voltando do jantar, fui direto pra cama, afinal no dia seguinte entenderíamos quais seriam as próximas etapas do produto. Mas, às 7h da manhã, o telefone tocou. Era o responsável pelo posto de gasolina piloto. Ele me ligou justamente para contar que o posto de gasolina estava **completamente parado** desde as 0h: o software parou de funcionar e bloqueou completamente o posto de gasolina.

Nosso software nunca havia sido testado com uma quantidade grande de abastecimentos. Os postos em Santo Domingo fazem muitos pequenos abastecimentos ao longo do dia (diferente daqui, onde enchemos o tanque de uma vez). O sistema não entendeu isso muito bem, e optou por bloquear as bombas de gasolina, para evitar fraude, já que não conseguia registrar as futuras compras.

O software fez com que o estabelecimento ficasse 12h parado, sem vender nada. Quanto será que isso custou ao dono do estabelecimento? Como será que foi a reação dele ao descobrir que o novo produto, no primeiro dia, causou tamanho estrago?

Os Estados Unidos estimam que erros de software lhes custam aproximadamente 60 bilhões de dólares por ano [38]. O dinheiro que poderia estar

sendo usado para erradicar a fome do planeta está sendo gasto em correções de software que não funcionam.

É incrível a quantidade de software que não funciona. Pergunte ao seu amigo não-técnico se ele já ficou irritado porque algum programa do seu dia a dia simplesmente parou de funcionar. Alguns erros de software são inclusive famosos por todos: o foguete Ariane 5 explodiu por um erro de software; um hospital panamenho matou pacientes pois seu software para dosagem de remédios errou.

1.3 POR QUE NÃO TESTAMOS?

Não há um desenvolvedor que não saiba que a solução para o problema é testar seus códigos. A pergunta é: **por que não testamos?**

Não testamos porque testar sai caro. Imagine o sistema em que você trabalha hoje. Se uma pessoa precisasse testá-lo do começo ao fim, quanto tempo ela levaria? Semanas? Meses? Pagar um mês de uma pessoa a cada mudança feita no código (sim, os desenvolvedores também sabem que uma mudança em um trecho pode gerar problemas em outro) é simplesmente impossível.

Testar sai caro, no fim, porque estamos pagando “a pessoa” errada para fazer o trabalho. Acho muito interessante a quantidade de tempo que gastamos criando soluções tecnológicas para resolver problemas “dos outros”. Por que não escrevemos programas que resolvam também os nossos problemas?

1.4 TESTES AUTOMATIZADOS E TDD

Uma maneira de conseguir testar o sistema todo de maneira constante e contínua a um preço justo é automatizando os testes. Ou seja, escrevendo um programa que testa o seu programa. Esse programa invocaria os comportamentos do seu sistema e garantiria que a saída é sempre a esperada.

Se isso fosse realmente viável, teríamos diversas vantagens. O teste executaria muito rápido (afinal, é uma máquina!). Se ele executa rápido, logo o rodaríamos constantemente. Se os rodarmos o tempo todo, descobriríamos os problemas mais cedo, diminuindo o custo que o erro geraria.

Um ponto que é sempre levantado em qualquer discussão sobre testes

manuais versus testes automatizados é produtividade. O argumento mais comum é o de que agora a equipe de desenvolvimento gastará tempo escrevendo código de teste; antes ela só gastava tempo escrevendo código de produção. Portanto, essa equipe será menos produtiva.

O contra-argumento nesse caso é: o que é produtividade? Se produtividade for medida através do número de linhas de código de produção escritos por dia, talvez o desenvolvedor seja sim menos produtivo. Agora, se produtividade for a quantidade de linhas de código de produção sem defeitos escritos por dia, provavelmente o desenvolvedor será mais produtivo ao usar testes automatizados.

Além disso, se analisarmos o dia a dia de um desenvolvedor que faz testes manuais, podemos perceber a quantidade de tempo que ele gasta com teste. Geralmente o desenvolvedor executa testes enquanto desenvolve o algoritmo completo. Ele escreve um pouco, roda o programa, e o programa falha. Nesse momento, o desenvolvedor entende o problema, corrige-o, e em seguida executa novamente o mesmo teste. Quantas vezes por dia o desenvolvedor executa o mesmo teste manual? O desenvolvedor que automatiza seus testes perde tempo apenas 1 vez com ele; nas próximas, ele simplesmente aperta um botão e vê a máquina executando o teste pra ele, de forma correta e rápida.

1.5 CONCLUSÃO

Minha família inteira é da área médica. Um jantar de fim de semana em casa parece mais um daqueles episódios de seriados médicos da televisão: pessoas discutindo casos e como resolvê-los. Apesar de entender praticamente nada sobre medicina, uma coisa me chama muito a atenção: o fanatismo deles por qualidade.

Um médico, ao longo de uma cirurgia, nunca abre mão de qualidade. Se o paciente falar para ele: “Doutor, o senhor poderia não lavar a mão e terminar a cirurgia 15 minutos mais cedo?”, tenho certeza de que o médico negaria na hora. Ele saberia que chegaria ao resultado final mais rápido, mas a chance de um problema é tão grande, que simplesmente não valeria a pena.

Em nossa área, vejo justamente o contrário. Qual desenvolvedor nunca

escreveu um código de má qualidade de maneira consciente? Quem nunca escreveu uma “gambiarra”? Quem nunca colocou software em produção sem executar o mínimo suficiente de testes para tal?

Não há desculpas para não testar software. E a solução para que seus testes sejam sustentáveis é automatizando. Testar é divertido, aumenta a qualidade do seu produto, e pode ainda ajudá-lo a identificar trechos de código que foram mal escritos ou projetados (aqui entra a prática de TDD). É muita vantagem.

Ao longo do livro, espero convencê-lo de que escrever testes automatizados é importante, e que, na verdade, é mais fácil do que parece.

1.6 COMO TIRAR DÚVIDAS?

Para facilitar a comunicação entre os leitores deste livro e interessados no assunto, criei um grupo de discussão no Google Groups. Você pode se cadastrar em <https://groups.google.com/forum/#!forum/tdd-no-mundo-real>.

Toda e qualquer discussão, dúvida ou sugestão será bem-vinda.

Não se esqueça também de participar do GUJ, o maior portal de desenvolvedores do Brasil: <http://www.guj.com.br/>

Lá você pode perguntar, responder, editar e dar pontos às dúvidas e respostas de outros desenvolvedores.

CAPÍTULO 2

Testes de Unidade

2.1 O QUE É UM TESTE DE UNIDADE?

Imagine-se passeando em uma loja virtual qualquer na web. Ao selecionar um produto, o sistema coloca-o no seu carrinho de compras. Ao finalizar a compra, o sistema fala com a operadora de cartão de crédito, retira o produto do estoque, dispara um evento para que a equipe de logística separe os produtos comprados e lhe envia um e-mail confirmando a compra.

O software que toma conta de tudo isso é complexo. Ele contém regras de negócio relacionadas ao carrinho de compras, ao pagamento, ao fechamento da compra. Mas, muito provavelmente, todo esse código não está implementado em apenas um único arquivo; esse sistema é composto por diversas pequenas classes, cada uma com sua tarefa específica.

Desenvolvedores, quando pensam em teste de software, geralmente imaginam um teste que cobre o sistema como um todo. Um teste de unidade não

se preocupa com todo o sistema; ele está interessado apenas em saber se uma pequena parte do sistema funciona.

Um teste de unidade testa uma única unidade do nosso sistema. Geralmente, em sistemas orientados a objetos, essa unidade é a classe. Em nosso sistema de exemplo, muito provavelmente existem classes como “CarrinhoDeCompras”, “Pedido”, e assim por diante. A ideia é termos baterias de testes de unidade separadas para cada uma dessas classes; cada bateria preocupada apenas com a sua classe.

2.2 PRECISO MESMO ESCRREVÊ-LOS?

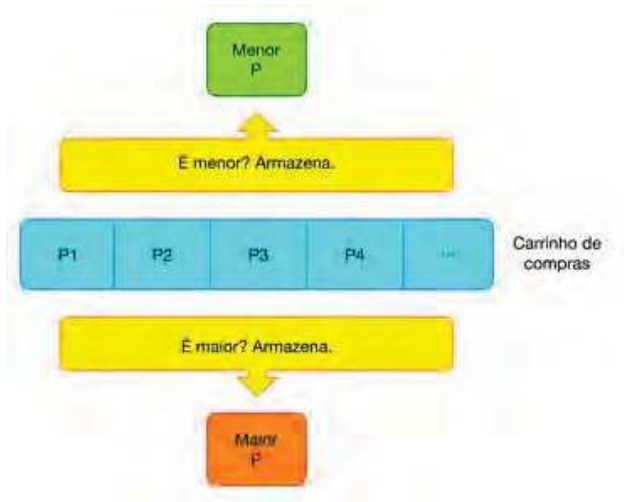
Essa mesma loja virtual precisa encontrar, dentro do seu carrinho de compras, os produtos de maior e menor valor. Um possível algoritmo para esse problema seria percorrer a lista de produtos no carrinho, comparar um a um, e guardar sempre a referência para o menor e o maior produto encontrado até então.

Em código, uma possível implementação seria:

```
class MaiorEMenor
  attr_reader :menor, :maior

  def encontra(carrinho)
    carrinho.produtos.each do |produto|
      if (@menor.nil? || produto.valor < @menor.valor)
        @menor = produto
      elsif (@maior.nil? || produto.valor > @maior.valor)
        @maior = produto
      end
    end
  end
end
```

Veja o método `encontra`. Ele recebe um `CarrinhoDeCompras` e percorre a lista de produtos, comparando sempre o produto corrente com o “menor e maior de todos”. Ao final, temos no atributo `maior` e `menor` os produtos desejados. A figura a seguir mostra como o algoritmo funciona:



Para exemplificar o uso dessa classe, veja o seguinte código:

```
require File.expand_path('./produto.rb',
  File.dirname(__FILE__))
require File.expand_path('./carrinho_de_compras.rb',
  File.dirname(__FILE__))
require File.expand_path('./maior_e_menor.rb',
  File.dirname(__FILE__))

carrinho = CarrinhoDeCompras.new
carrinho << Produto.new('Liquidificador', 250.0)
carrinho << Produto.new('Geladeira', 450.0)
carrinho << Produto.new('Jogo de pratos', 70.0)

algoritmo = MaiorEMenor.new
algoritmo.encontra carrinho

puts "O menor produto: #{algoritmo.menor.nome}"
puts "O maior produto: #{algoritmo.maior.nome}"
```

O carrinho contém três produtos: liquidificador, geladeira e jogo de pra-

tos. É fácil perceber que o jogo de pratos é o produto mais barato (R\$ 70,00), enquanto que a geladeira é o mais caro (R\$ 450,00). A saída do programa é exatamente igual à esperada:

```
0 menor produto: Jogo de pratos
0 maior produto: Geladeira
```

Apesar de aparentemente funcionar, se esse código for para produção, a loja virtual terá problemas.

2.3 O PRIMEIRO TESTE DE UNIDADE

A classe `MaiorEMenor` respondeu corretamente ao teste feito, mas ainda não é possível dizer se ela realmente funciona para outros cenários. Observe o código a seguir:

```
require File.expand_path('./produto.rb',
  File.dirname(__FILE__))
require File.expand_path('./carrinho_de_compras.rb',
  File.dirname(__FILE__))
require File.expand_path('./maior_e_menor.rb',
  File.dirname(__FILE__))

carrinho = CarrinhoDeCompras.new
carrinho << Produto.new('Geladeira', 450.0)
carrinho << Produto.new('Liquidificador', 250.0)
carrinho << Produto.new('Jogo de pratos', 70.0)

algoritmo = MaiorEMenor.new
algoritmo.encontra carrinho

puts "0 menor produto: #{algoritmo.menor.nome}"
puts "0 maior produto: #{algoritmo.maior.nome}"
```

Esse código não é tão diferente do anterior. Os produtos, bem como os valores, são os mesmos; apenas a ordem em que eles são inseridos no carrinho foi trocada. Espera-se então que o programa produza a mesma saída. Mas, ao executá-lo, a seguinte saída é gerada:

```
O menor produto: Jogo de pratos
testa_maior_e_menor.rb:14:
  undefined method `nome' for nil:NilClass (NoMethodError)
```

Problema! Essa não era a saída esperada! Agora está claro que a classe `MaiorEMenor` não funciona bem para todos os cenários. Se os produtos, por algum motivo, forem adicionados no carrinho em ordem decrescente, a classe não consegue calcular corretamente.

Uma pergunta importante para o momento é: será que o desenvolvedor, ao escrever a classe, perceberia esse erro? Será que ele faria todos os testes necessários para garantir que a classe realmente funciona?

Como discutido no capítulo anterior, equipes de software tendem a não testar software, pois o teste leva tempo e, por consequência, dinheiro. Na prática, equipes acabam por executar poucos testes ralos, que garantem apenas o cenário feliz e mais comum. Todos os problemas da falta de testes, que já foram discutidos anteriormente, agora fazem sentido. Imagine se a loja virtual colocasse esse código em produção. Quantas compras seriam perdidas por causa desse problema?

Para diminuir a quantidade de erros levados para o ambiente de produção, é necessário testar o código constantemente. Idealmente, a cada alteração feita, todo o sistema deve ser testado por inteiro novamente. Mas isso deve ser feito de maneira sustentável: é impraticável pedir para que seres humanos testem o sistema inteiro a cada alteração feita por um desenvolvedor.

A solução para o problema é fazer com que a máquina teste o software. A máquina executará o teste rapidamente e sem custo, e o desenvolvedor não gastaria mais tempo executando testes manuais, mas sim apenas em evoluir o sistema.

Escrever um teste automatizado não é uma tarefa tão árdua. Ele, na verdade, se parece muito com um teste manual. Imagine um desenvolvedor que deva testar o comportamento do carrinho de compras da loja virtual quando existem dois produtos lá cadastrados: primeiramente, ele “clikaria em comprar em dois produtos”, em seguida “iria para o carrinho de compras”, e por fim, verificaria “a quantidade de itens no carrinho (deve ser 2)” e o “o valor total do carrinho (que deve ser a soma dos dois produtos adicionados anteriormente)”.

Ou seja, de forma generalizada, o desenvolvedor primeiro pensa em um cenário (dois produtos comprados), depois executa uma ação (vai ao carrinho de compras), e por fim, valida a saída (vê a quantidade de itens e o valor total do carrinho). A figura a seguir mostra os passos que um testador geralmente faz quando deseja testar uma funcionalidade.



Um teste automatizado é similar. Ele descreve um cenário, executa uma ação e valida uma saída. A diferença é que quem fará tudo isso será a máquina, sem qualquer intervenção humana.

De certa forma, um teste automatizado já foi escrito neste capítulo. Veja o próximo código, escrito para testar superficialmente a classe `MaiorEMenor`:

```
require File.expand_path('./produto.rb',
  File.dirname(__FILE__))
require File.expand_path('./carrinho_de_compras.rb',
  File.dirname(__FILE__))
require File.expand_path('./maior_e_menor.rb',
  File.dirname(__FILE__))

carrinho = CarrinhoDeCompras.new
carrinho << Produto.new('Geladeira', 450.0)
carrinho << Produto.new('Liquidificador', 250.0)
carrinho << Produto.new('Jogo de pratos', 70.0)

algoritmo = MaiorEMenor.new
algoritmo.encontra carrinho

puts "O menor produto: #{algoritmo.menor.nome}"
puts "O maior produto: #{algoritmo.maior.nome}"
```

Veja que esse código contém, de forma automatizada, boa parte do que foi descrito em relação ao teste manual: ele monta um cenário (um carrinho de

compras com 3 produtos), executa uma ação (invoca o método `encontra`), e valida a saída (imprime o maior e o menor produto).

E o melhor: uma máquina faz (quase) tudo isso. Ao rodar o código, a máquina monta o cenário e executa a ação sem qualquer intervenção humana. Mas uma ação humana ainda é necessária para descobrir se o comportamento executou de acordo. Um humano precisa ver a saída e conferir com o resultado esperado.

É preciso que o próprio teste faça a validação e informe o desenvolvedor caso o resultado não seja o esperado. Para melhorar esse código, agora só introduzindo um framework de teste automatizado. Esse framework nos daria um relatório mais detalhado dos testes que foram executados com sucesso e, mais importante, dos testes que falharam, trazendo o nome do teste e a linha que apresentaram problemas. Neste livro, faremos uso do TestUnit [34], o *framework* de testes padrão em Ruby 1.8. Em Ruby 1.9 e superior, o *framework* padrão é o MiniTest [7] que é muito semelhante ao TestUnit. O *framework* RSpec [6] é outro *framework* (disponível para todas as versões de Ruby) que tem um sabor um pouco mais pro lado de BDD [31]. Veja o último apêndice do livro para entender melhor as diferenças entre TestUnit e RSpec e exemplos de testes RSpec equivalentes a testes TestUnit.

Também é possível acompanhar os exemplos de código do livro seguindo o histórico dos seguintes repositórios:

- Para Ruby 1.8 e Test Unit: <https://github.com/hugocorbucci/tdd-em-ruby-codigo-testunit>
- Para Ruby 1.9+ e MiniTest: <https://github.com/hugocorbucci/tdd-em-ruby-codigo-minitest>
- Para Ruby 1.9+ e RSpec: <https://github.com/hugocorbucci/tdd-em-ruby-codigo-rspec>

Para converter o código anterior em um teste que o TestUnit entenda, não é necessário muito trabalho. A única parte que ainda não é feita 100% pela máquina é a terceira parte do teste: a validação. É justamente ali que invocaremos os métodos do TestUnit. Eles que farão a comparação do resultado esperado

com o calculado. No TestUnit, o método para isso é o `assert_equal` que está definido na classe `Test::Unit::TestCase`:

```
require 'test/unit'
require File.expand_path('../lib/produto.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/carrinho_de_compras.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/maior_e_menor.rb',
  File.dirname(__FILE__))

class TesteMaiorEMenor < Test::Unit::TestCase
  def teste_ordem_decrescente
    carrinho = CarrinhoDeCompras.new
    carrinho << Produto.new('Geladeira', 450.0)
    carrinho << Produto.new('Liquidificador', 250.0)
    carrinho << Produto.new('Jogo de pratos', 70.0)

    algoritmo = MaiorEMenor.new
    algoritmo.encontra carrinho

    assert_equal 'Jogo de pratos', algoritmo.menor.nome
    assert_equal 'Geladeira', algoritmo.maior.nome
  end
end
```

Pronto. Esse código é executado pelo TestUnit. O teste, como bem sabemos, falhará:

```
$ ruby test/teste_maior_e_menor.rb
Loaded suite test/teste_maior_e_menor
Started
E
Finished in 0.00021 seconds.
```

1) Error:

```
teste_ordem_decrescente(TesteMaiorEMenor):
NoMethodError: undefined method `nome' for nil:NilClass
    test/teste_maior_e_menor.rb:17:in `teste_ordem_decrescente'
```



```
1 tests, 1 assertions, 0 failures, 1 errors
```

Para fazer o teste passar, é necessário corrigir o erro na classe de produção. O que faz o código falhar é a presença do `elsif` que fazia com que o código nunca passasse pelo segundo trecho de código, que verificava justamente o maior elemento. Ao trocá-lo por um `if`, o teste passa:

```
class MaiorEMenor
  attr_reader :menor, :maior

  def encontra(carrinho)
    carrinho.produtos.each do |produto|
      if(@menor.nil? || produto.valor < @menor.valor)
        @menor = produto
      end
      if (@maior.nil? || produto.valor > @maior.valor)
        @maior = produto
      end
    end
  end
end

$ ruby test/teste_maior_e_menor.rb
Loaded suite test/teste_maior_e_menor
Started
.
Finished in 0.000192 seconds.
```

```
1 tests, 2 assertions, 0 failures, 0 errors
```

Repare também no tempo que a máquina levou para executar o teste: 0.000192 segundos. Mais rápido do que qualquer ser humano faria. Isso possibilita ao desenvolvedor executar esse teste diversas vezes ao longo do seu dia de trabalho.

E o melhor: se algum dia um outro desenvolvedor alterar esse código, ele poderá executar a bateria de testes automatizados existente e descobrir se a sua alteração fez alguma funcionalidade que já funcionava anteriormente

parar de funcionar. Isso é conhecido como **testes de regressão**. Eles garantem que o sistema não regrediu.

MEU PRIMEIRO TESTE AUTOMATIZADO

Há muitos anos, eu trabalhava em uma pequena consultoria de desenvolvimento de software em São Paulo. E, como em toda consultoria, precisávamos anotar nossas horas de trabalho em cada um dos projetos da empresa, a fim de cobrar corretamente cada um dos clientes.

Resolvi, por conta própria, criar um simples projeto para controle de horas de trabalho (eu precisava de um motivo para experimentar todas as coisas que estava aprendendo naquele momento e, dentre elas, testes automatizados). O sistema era bem simples: cadastro de funcionários, cadastro de projetos, ligação entre um funcionário e um projeto e cadastro de horas em um projeto.

Ao final da implementação, apresentei o sistema ao diretor da empresa. Ele adorou, e me sugeriu a implementação de uma simples regra de negócio: se o número total de horas colocadas no projeto ultrapasse um determinado limite, nenhum funcionário poderia adicionar mais horas nele.

A implementação era bem trivial: bastava fazer um `if`. Lembro-me que implementei a regra de negócio e então escrevi o teste de unidade para garantir que havia implementado corretamente. O teste que acabei de escrever ficou verde, mas três ou quatro testes que já existiam ficaram vermelhos. A única linha de código que escrevi fez com que funcionalidades que já funcionavam parassem de trabalhar corretamente.

Naquele momento me vieram à cabeça todas as vezes que escrevi uma ou duas linhas de código, aparentemente simples, mas que poderiam ter quebrado o sistema. Percebi então a grande vantagem da bateria de testes automatizados: segurança. Percebi como é fácil quebrar código, e como é difícil perceber isso sem testes automatizados. Daquele dia em diante, penso duas vezes antes de escrever código sem teste.

2.4 CONTINUANDO A TESTAR

Ainda são necessários muitos testes para garantir que a classe `MaiorEMenor` funcione corretamente. Nesse momento, o único cenário testado são produtos em ordem decrescente. Muitos outros cenários precisam ser testados. Dentre eles:

- Produtos com valores em ordem crescente;
- Produtos com valores em ordem variada;
- Um único produto no carrinho.

Conhecendo o código da classe `MaiorEMenor`, é possível prever que esses cenários funcionarão corretamente. Escrever testes automatizados para esses cenários pode parecer inútil nesse momento.

Entretanto, é importante lembrar que, em códigos reais, muitos desenvolvedores fazem alterações nele. Para o desenvolvedor que implementa a classe, o código dela é bem natural e simples de ser entendido. Mas, para os futuros desenvolvedores que o alterarão, esse código pode não parecer tão simples. É necessário então prover segurança para eles nesse momento. Portanto, apesar de alguns testes parecerem desnecessários nesse momento, eles garantirão que a evolução dessa classe será feita com qualidade.

Apenas para exemplificar mais ainda, vamos escrever mais um teste, dessa vez para um carrinho com apenas 1 produto. Repare que, nesse cenário, é esperado que o maior e menor produtos seja o mesmo:

```
require 'test/unit'
require File.expand_path('../lib/produto.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/carrinho_de_compras.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/maior_e_menor.rb',
  File.dirname(__FILE__))

class TesteMaiorEMenor < Test::Unit::TestCase
  def teste_apenas_um_produto
    carrinho = CarrinhoDeCompras.new
```

```
carrinho << Produto.new('Geladeira', 450.0)

algoritmo = MaiorEMenor.new
algoritmo.encontra carrinho

assert_equal 'Geladeira', algoritmo.menor.nome
assert_equal 'Geladeira', algoritmo.maior.nome
end
end
```

COMPARANDO SÓ O NOME?

Ao invés de comparar apenas o nome do produto, você poderia comparar o objeto inteiro: `assert_equal produto, algoritmo.menor`, por exemplo. Mas, para isso, o método `==` deve estar implementado na classe `Produto`.

Em certos casos, essa solução pode ser mais interessante, afinal você comparará o objeto inteiro e não só apenas um atributo. No entanto, se o `==` só for útil para o teste, em geral, é melhor manter a definição de igualdade de produto dentro da classe de teste.

Nesse caso, o melhor é escrever um método auxiliar no teste que faça as múltiplas comparações que ficariam dentro do `Produto`. Algo como `verifica_produtos_iguais produto, algoritmo.menor`.

2.5 CONCLUSÃO

Desenvolvedores gastam toda sua vida automatizando processos de outras áreas de negócio, criando sistemas para RHs, controle financeiro, entre outros, com o intuito de facilitar a vida daqueles profissionais. Por que não criar software que automatize o seu próprio ciclo de trabalho?

Testes automatizados são fundamentais para um desenvolvimento de qualidade e é obrigação de todo desenvolvedor escrevê-los. Sua existência traz diversos benefícios para o software, como o aumento da qualidade e a diminuição de bugs em produção. Nos próximos capítulos, discutiremos sobre como aumentar ainda mais o *feedback* que os testes nos dão sobre a qualidade

do software.

CAPÍTULO 3

Introdução ao *Test-Driven Development*

Desenvolvedores (ou qualquer outro papel que é executado dentro de uma equipe de software) estão muito acostumados com o “processo tradicional” de desenvolvimento: primeiro a implementação e depois o teste. Entretanto, uma pergunta interessante é: **Será que é possível inverter, ou seja, testar primeiro e depois implementar? E mais importante, faz algum sentido?**

Para responder essa pergunta, ao longo deste capítulo desenvolveremos um simples algoritmo matemático, mas desta vez invertendo o ciclo de desenvolvimento: o teste será escrito antes da implementação. Ao final, discutiremos as vantagens e desvantagens dessa abordagem.

3.1 O PROBLEMA DOS NÚMEROS ROMANOS

Numerais romanos foram criados na Roma Antiga e eles foram utilizados em todo o seu império. Os números eram representados por sete diferentes símbolos, listados na tabela a seguir.

- I, unus, 1, (um)
- V, quinque, 5 (cinco)
- X, decem, 10 (dez)
- L, quinquaginta, 50 (cinquenta)
- C, centum, 100 (cem)
- D, quingenti, 500 (quinhentos)
- M, mille, 1.000 (mil)

Para representar outros números, os romanos combinavam estes símbolos, começando do algarismo de maior valor e seguindo a regra:

- Algarismos de menor ou igual valor à direita são somados ao algarismo de maior valor;
- Algarismos de menor valor à esquerda são subtraídos do algarismo de maior valor.

Por exemplo, XV representa 15 ($10 + 5$) e o número XXVIII representa 28 ($10 + 10 + 5 + 1 + 1 + 1$). Há ainda uma outra regra: nenhum símbolo pode ser repetido lado a lado por mais de 3 vezes. Por exemplo, o número 4 é representado pelo número IV ($5 - 1$) e não pelo número IIII.

Existem outras regras (especialmente para números maiores, que podem ser lidas na wikipedia [37]), mas em linhas gerais, este é o problema a ser resolvido. Dado um numeral romano, o programa deve convertê-lo para o número inteiro correspondente.

3.2 O PRIMEIRO TESTE

Conhecendo o problema dos numerais romanos, é possível levantar os diferentes cenários que precisam ser aceitos pelo algoritmo: um símbolo, dois símbolos iguais, três símbolos iguais, dois símbolos diferentes do maior para o menor, quatro símbolos dois a dois, e assim por diante. Dados todos estes cenários, uns mais simples que os outros, começaremos pelo mais simples: um único símbolo.

Começando pelo teste “deve entender o símbolo I”. A classe responsável pela conversão pode ser chamada, por exemplo, de `ConversorDeNumeroRomano`, e o método `converte`, recebendo uma string com o numeral romano e devolvendo o valor inteiro representado por aquele número:

```
require 'test/unit'
require File.expand_path(
  '../lib/conversor_de_numero_romano.rb',
  File.dirname(__FILE__))

class ConversorDeNumeroRomanoTest < Test::Unit::TestCase
  def teste_deve_entender_o_simbolo_I
    romano = ConversorDeNumeroRomano.new
    numero = romano.converte 'I'
    assert_equal 1, numero
  end
end
```

Veja que, nesse momento, esse código lança um erro de interpretação; a classe `ConversorDeNumeroRomano`, bem como o método `converte` não existem. Para resolver esse problema, é necessário criar classe no caminho especificado no teste (`lib/conversor_de_numero_romano.rb`), mesmo que sem uma implementação real:

```
class ConversorDeNumeroRomano
  def converte numero_em_romano
    0
  end
end
```

De volta ao teste, o interpretador Ruby consegue executar o teste mas ele falha. Porém não há problema; isso já era esperado. Para fazê-lo passar, introduziremos ainda uma segunda regra: **o código escrito deve ser sempre o mais simples possível**.

Com essa regra em mente, o código mais simples que fará o teste passar é fazer simplesmente o método `converte` retornar o número 1:

```
def converte numero_em_romano
  1
end
```

Desenvolvedores, muito provavelmente, não ficarão felizes com essa implementação, afinal ela funciona apenas para um caso. Mas isso não é problema, afinal a implementação não está pronta; ainda estamos trabalhando nela.

Um próximo cenário seria o símbolo V. Nesse caso, o algoritmo deve retornar 5. Novamente começando pelo teste:

```
def teste_deve_entender_o_simbolo_V
  romano = ConversorDeNumeroRomano.new
  numero = romano.converte 'V'
  assert_equal 5, numero
end
```

Esse teste também falha. Novamente faremos a implementação mais simples que resolverá o problema. Podemos, por exemplo, fazer com que o método `converte` verifique o conteúdo do número a ser convertido: se o valor for “I”, o método retorna 1; se o valor for “V”, o método retorna 5:

```
def converte numero_em_romano
  if numero_em_romano == 'I'
    return 1
  elsif numero_em_romano == 'V'
    return 5
  end
  return 0
end
```

Os testes passam (os dois que temos). Poderíamos repetir o mesmo teste e a mesma implementação para os símbolos que faltam (X, L, C, M, ...). Mas nesse momento, já temos uma primeira definição sobre nosso algoritmo: quando o numeral romano possui apenas um símbolo, basta devolvermos o inteiro associado a ele.

Ao invés de escrever um monte de `ifs` para cada símbolo, é possível usar um *case*, que corresponde melhor ao nosso cenário. Melhorando ainda mais, ao invés do *case*, é possível guardar os símbolos em uma tabela, ou em Ruby, em um mapa entre o algarismo e o inteiro correspondente à ele. Além disso, em Ruby, se não houver um retorno explícito, a última coisa avaliada no método será devolvida. Sabendo disso, vamos nesse momento alterar o código para refletir a solução:

```
class ConversorDeNumeroRomano
  TABELA = {
    'I' => 1,
    'V' => 5,
    'X' => 10,
    'L' => 50,
    'C' => 100,
    'D' => 500,
    'M' => 1000,
  }
  def converte numero_em_romano
    TABELA[numero_em_romano]
  end
end
```

Ambos os testes continuam passando. Passaremos agora para um segundo cenário: dois símbolos em sequência, como, por exemplo, “II” ou “XX”. Começando novamente pelo teste, temos:

```
def teste_deve_entender_dois_simbolos_como_II
  romano = ConversorDeNumeroRomano.new
  numero = romano.converte 'II'
  assert_equal 2, numero
end
```

Para fazer o teste passar de maneira simples, é possível simplesmente adicionar o símbolo “II” na tabela:

```
TABELA = {  
    'I' => 1,  
  
    'II' => 2,  
  
    'V' => 5,  
    'X' => 10,  
    'L' => 50,  
    'C' => 100,  
    'D' => 500,  
    'M' => 1000,  
}
```

O teste passa. Mas, apesar de simples, essa não parece uma boa ideia de implementação: seria necessário incluir todos os possíveis símbolos nessa tabela, o que não faz sentido.

É hora de refatorar esse código novamente. Uma possível solução seria iterar em cada um dos símbolos no numeral romano e acumular seu valor; ao final, retorna o valor acumulado. Para isso, é necessário mudar a `TABELA` para guardar somente os símbolos principais da numeração romana. Uma possível implementação deste algoritmo seria:

```
class ConversorDeNumeroRomano  
    TABELA = {  
        'I' => 1,  
        'V' => 5,  
        'X' => 10,  
        'L' => 50,  
        'C' => 100,  
        'D' => 500,  
        'M' => 1000,  
    }  
  
    def converte numero_em_romano  
        acumulador = 0  
        numero_em_romano.split('').each do |digito_em_romano|  
            acumulador += TABELA[digito_em_romano]        end  
    end
```

```
end
  acumulador
end
end
```

UMA ABORDAGEM MAIS FUNCIONAL

Desenvolvedores experientes com Ruby não ficarão muito satisfeitos com essa implementação por ela ser muito extensa e por ela não tirar proveito dos métodos existentes no módulo `Enumerable`. Esses métodos permitem que o código tenha um lado um pouco mais puxado para o lado de linguagens funcionais como Haskell, Erlang, Clojure e Scala.

Uma técnica comum nessas linguagens é conhecida como *Map/Reduce* que consiste em transformar uma lista de elementos em uma nova lista com outros elementos que sejam mais interessantes (*Map*) e reduzir essa nova lista em um único item que seja o resultado desejado (*Reduce*).

Usando essa técnica e o método `inject` disponível no módulo `Enumerable`, o método `converte` poderia ser escrito da seguinte forma:

```
def converte numero_em_romano
  numero_em_romano.split('').
    map{|digito| TABELA[digito]}.
    reduce(:+)
end
```

O primeiro passo (`numero_em_romano.split("")`) converte a `String` em uma lista de caracteres. O segundo passo (`map{|digito| TABELA[digito]}`) pega o valor de cada caractere na tabela. O último passo (`reduce(:+)`) soma todos os elementos.

Os três testes continuam passando. E, dessa forma, resolvemos o problema de dois símbolos iguais em seguida. O próximo cenário são quatro símbolos, dois a dois, como por exemplo, “XXII”, que deve resultar em 22. O teste:

```
def teste_deve_entender_quatro_simbolos_dois_a_dois_como_XXII
  romano = ConversorDeNumeroRomano.new
  numero = romano.converte 'XXII'
  assert_equal 22, numero
end
```

Esse teste já passa sem que precisemos fazer qualquer alteração. O algoritmo existente até então já resolve o problema. Vamos então para o próximo cenário: números como “IV” ou “IX”, nos quais não basta apenas somar os símbolos existentes.

Novamente, começando pelo teste:

```
def teste_deve_entender_numeros_como_IX
  romano = ConversorDeNumeroRomano.new
  numero = romano.converte 'IX'
  assert_equal 9, numero
end
```

Para fazer esse teste passar, é necessário pensar um pouco melhor sobre o problema. Repare que os símbolos em um numeral romano, da direita para a esquerda, sempre crescem. Quando um número à esquerda é menor do que seu vizinho à direita, esse número deve então ser subtraído ao invés de somado no acumulador. Esse algoritmo, em código:

```
def converte numero_em_romano
  acumulador = 0
  ultimo_vizinho_da_direita = 0
  numero_em_romano.split('').reverse.each do |digito_em_romano|
    # pega o inteiro referente ao simbolo atual
    atual = TABELA[digito_em_romano]

    # se o da direita for menor, o multiplicaremos
    # por -1 para torná-lo negativo
    multiplicador = 1
    multiplicador = -1 if(atual < ultimo_vizinho_da_direita)

    acumulador += TABELA[digito_em_romano] * multiplicador

    # atualiza o vizinho da direita
  end
end
```

```
ultimo_vizinho_da_direita = atual
end
acumulador
end
```

Os testes agora passam. Mas veja que existe código repetido ali. Na linha em que o acumulador é somado com o valor atual, `TABELA[digito_em_romano]` pode ser substituído pela variável `atual`. Melhorando o código, temos:

```
acumulador += atual * multiplicador;
```

A refatoração foi feita com sucesso, já que os testes continuam passando. Ao próximo cenário: numeral romano que contenha tanto números “invertidos”, como “IV”, e dois símbolos lado a lado, como “XX”. Por exemplo, o número “XXIV”, que representa o número 24.

Começando pelo teste:

```
def teste_deve_entender_numeros_complexos_como_XXIV
  romano = ConversorDeNumeroRomano.new
  numero = romano.converte 'XXIV'
  assert_equal 24, numero
end
```

Esse teste já passa; o algoritmo criado até então já atende o cenário do teste. Ainda há outros cenários que poderiam ser testados, mas já fizemos o suficiente para discutir sobre o assunto.

INCREMENTANDO A ABORDAGEM FUNCIONAL

Como ficaria a versão funcional apresentada antes? Veja o código que já existia:

```
def converte numero_em_romano
  numero_em_romano.split('').
    map{|digito| TABELA[digito]}.
    reduce(:+)
end
```

Agora ficou mais complicado. O que mudou? Foi necessário inverter a lista e lembrar qual era o último vizinho da direita. Em uma linguagem puramente funcional, ao iterar um elemento de uma lista, teríamos o resto da lista à disposição. Isso ajudaria a decidir se, no mapeamento, é necessário um sinal positivo ou negativo. Infelizmente, o módulo `Enumerable` não dá essa visibilidade. No entanto, Ruby não é uma linguagem puramente funcional então é possível usar um aspecto funcional (*Map/Reduce*) com um aspecto procedural (variáveis locais) como na solução a seguir:

```
def converte numero_em_romano
  ultimo_vizinho_da_direita = 0
  digitos = numero_em_romano.split('').
    map{|digito| TABELA[digito]}
  digitos.reverse.reduce(0) do |acumulador, digito|
    multiplicador = 1
    multiplicador = -1 if digito < ultimo_vizinho_da_direita

    # atualiza o vizinho da direita
    ultimo_vizinho_da_direita = digito

    acumulador + digito * multiplicador
  end
end
```

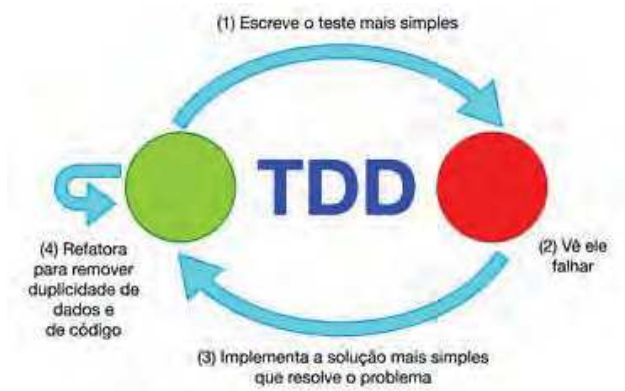
Nesse caso, o método `map` ajudou a converter os caracteres em números e o método `reduce` cuidou do acumulador. A diferença com o código final é pequena mas deixa mais clara a ideia do *Map/Reduce*.

3.3 REFLETINDO SOBRE O ASSUNTO

De maneira mais abstrata, o ciclo que foi repetido ao longo do processo de desenvolvimento da classe acima foi:

- Escrevemos um teste de unidade para uma nova funcionalidade;
- Vimos o teste falhar;
- Implementamos o código mais simples para resolver o problema;
- Vimos o teste passar;
- Melhoramos (refatoramos) nosso código quando necessário.

Esse ciclo de desenvolvimento é conhecido por **Test-Driven Development (TDD)**, ou, **Desenvolvimento Guiado pelos Testes**. A ideia é simples: o desenvolvedor deve começar a implementação pelo teste e deve, o tempo todo, fazer de tudo para que seu código fique simples e com qualidade.



Esse ciclo é também conhecido como ciclo **vermelho-verde-refatora** (ou *red-green-refactor*). O primeiro passo é escrever um teste que falha. A cor vermelha representa esse teste falhando. Em seguida o fazemos passar (a cor verde representa ele passando). Por fim, refatoramos para melhorar o código que escrevemos garantindo que os testes continuam passando.

3.4 QUAIS AS VANTAGENS?

Muitos praticantes de TDD afirmam que executar esse ciclo pode ser muito vantajoso para o processo de desenvolvimento. Alguns deles:

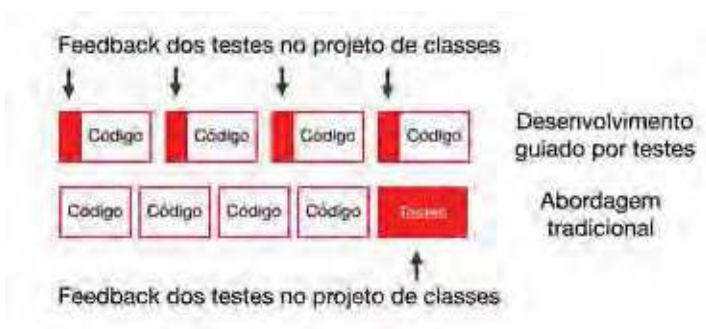
- **Foco no teste e não na implementação.** Ao começar pelo teste, o programador consegue pensar somente no que a classe **deve** fazer, e esquece por um momento da implementação. Isso o ajuda a pensar em melhores cenários de teste para a classe sob desenvolvimento.
- **Código nasce testado.** Se o programador pratica o ciclo corretamente, isso implica que todo o código de produção escrito possui ao menos um teste de unidade verificando que ele funciona corretamente.
- **Simplicidade.** Ao buscar o código mais simples constantemente, o desenvolvedor acaba fugindo de soluções complexas, comuns em todos os sistemas. O praticante de TDD escreve código que apenas resolve os problemas que estão representados por um teste de unidade. Quantas vezes o desenvolvedor não escreve código desnecessariamente complexo?
- **Melhor reflexão sobre o design da classe.** No cenário tradicional, muitas vezes a falta de coesão ou o excesso de acoplamento é causado pelo desenvolvedor que só pensa na implementação e acaba esquecendo como a classe vai funcionar perante o todo. Ao começar pelo teste, o desenvolvedor pensa sobre como sua classe deverá se comportar perante as outras classes do sistema. O teste atua como o primeiro cliente da classe que está sendo escrita. Nele, o desenvolvedor toma decisões como o nome da classe, os seus métodos, parâmetros, tipos de retorno etc. No fim, todas elas são decisões de design e, quando o desenvolvedor consegue observar com atenção o código do teste, seu design de classes pode crescer muito em qualidade.

Todos estes pontos serão melhor descritos e aprofundados no capítulo a seguir. Nesse momento, foque nestas vantagens. Uma pergunta que pode vir a cabeça é: “No modelo tradicional, onde os testes são escritos depois, o desenvolvedor não tem os mesmos benefícios?”

A resposta é sim. Mesmo desenvolvedores que escrevem testes depois podem obter as mesmas vantagens. Um desenvolvedor pode perceber que está com dificuldades de escrever um teste e descobrir que o design da classe que implementou tem problemas; ele pode também conseguir abstrair a implementação e escrever bons cenários de teste.

Mas há uma disparidade que faz toda a diferença: a quantidade de vezes que um programador praticante de TDD recebe feedback sobre esses pontos e a quantidade que um programador que não pratica TDD recebe.

Veja a figura a seguir. O praticante de TDD escreve um pouco de testes, um pouco de implementação e recebe feedback. Isso acontece ao longo do desenvolvimento de maneira frequente. Já um programador que não pratica TDD espera um tempo (às vezes longo demais) para obter o mesmo feedback.



Ao receber feedback desde cedo, o programador pode melhorar o código e corrigir problemas a um custo menor do que o programador que recebeu a mesma mensagem muito tempo depois. Todo programador sabe que alterar o código que ele acabara de escrever é muito mais fácil e rápido do que alterar o código escrito 3 dias atrás.

No fim, TDD apenas maximiza a quantidade de feedback sobre o código que está sendo produzido, fazendo o programador perceber os problemas antecipadamente e, por consequência, diminuindo os custos de manutenção e melhorando o código.

DESENVOLVEDORES E A BUSCA PELA COMPLEXIDADE

Acho impressionante como nós, desenvolvedores, somos atraídos por complexidade. Gostamos de problemas difíceis, que nos desafiem. Lembro-me que no começo de carreira, quando pegava alguma funcionalidade simples de ser implementada, eu mesmo “aumentava a funcionalidade” somente para torná-la interessante do ponto de vista técnico. Ou seja, um simples relatório se tornava um relatório complexo, com diversos filtros e que podia ser exportado para muitos formatos diferentes.

Aprender a ser simples foi difícil. O TDD, de certa forma, me ajudou nisso. Ao pensar de pouco em pouco, e implementar somente o necessário para resolver o problema naquele momento, comecei a perceber que poderia gastar meu tempo implementando funcionalidades que realmente agregariam valor para o usuário final.

Portanto, não ache que “ser simples” é fácil. Lembre-se que somos atraídos por complexidade.

3.5 UM POUCO DA HISTÓRIA DE TDD

TDD ficou bastante popular após a publicação do livro *TDD: By Example*, do Kent Beck, em 2002. O próprio Kent afirma que TDD não foi uma ideia totalmente original. Ele conta que, em algum momento de sua vida, que leu em algum dos livros de seu pai (que também era programador), sobre uma técnica de testes mais antiga, onde o programador colocava na fita o valor que ele esperava daquele programa, e então o programador desenvolvia até chegar naquele valor.

Ele próprio conta que achou a ideia estúpida. Qual o sentido de escrever um teste que falha? Mas resolveu experimentar. Após a experiência, ele disse que “as pessoas sempre falavam pra ele conseguir separar o que o programa deve fazer, da sua implementação final, mas que ele não sabia como fazer, até aquele momento em que resolveu escrever o teste antes.”

Daquela momento em diante, Kent Beck continuou a trabalhar na ideia. Em 1994, ele escreveu o seu primeiro framework de testes de unidade, o SUnit

(para Smalltalk). Em 1995, ele apresentou TDD pela primeira vez na OOPSLA (conferência muito famosa da área de computação, já que muitas novidades tendem a aparecer lá).

Já em 2000, o JUnit surgiu e Kent Beck, junto com Erich Gamma, publicou o artigo chamado de “Test Infected” [1], que mostrava as vantagens de se ter testes automatizados e como isso pode ser viciante.

Finalmente em 2002, Kent Beck lançou seu livro sobre isso, e desde então a prática tem se tornado cada vez mais popular entre os desenvolvedores.

A história mais completa pode ser vista na Wiki da C2 [2] ou na palestra do Steve Freeman e Michael Feathers na QCON de 2009 [17].

FERREIRA FALA

Duas histórias muito conhecidas cercam Kent Beck e o nascimento dos primeiros frameworks de testes de unidade. O SUnit, para Smalltalk, não era um código distribuído em arquivos da maneira usual. Beck na época atuava como consultor e tinha o hábito de recriar o framework junto com os programadores de cada cliente. A criação do JUnit também é legendaria: a primeira versão foi desenvolvida numa sessão de programação pareada entre o Kent Beck e o Erich Gamma em um voo Zurique-Atlanta.

3.6 CONCLUSÃO

Neste capítulo, apresentamos a prática de TDD e mostramos algumas das vantagens que o programador obtém ao utilizá-la no seu dia a dia. Entretanto, é possível melhorar ainda mais a maneira na qual o programador faz uso da prática.

Somente praticando TDD com esse simples exemplo, é possível levantar diversas questões, como:

- Muitos cenários foram deixados para trás, como por exemplo, a conversão de números como “CC”, “MM” etc. Eles devem ser escritos?

- O inverso da pergunta anterior: testes para “I”, “V”, “X” devem ser considerados diferentes ou repetidos?
- E quanto à repetição de código dentro de cada teste? É possível melhorar a qualidade de código do próprio teste?
- A ideia de sempre fazer o teste passar da maneira mais simples possível faz sentido? Deve ser aplicado 100% do tempo?
- Existe alguma regra para dar nomes aos testes?
- Qual o momento ideal para refatorar o código?
- Se durante a implementação, um teste antigo, que passava, quebra? O que devo fazer?

Para que um desenvolvedor faça uso de TDD de maneira profissional e produtiva, estes e outros pontos devem ser discutidos e clarificados. Nos próximos capítulos, responderemos a cada uma dessas perguntas, usando como exemplo um sistema razoavelmente complexo muito similar aos encontrados no mundo real.

CAPÍTULO 4

Simplicidade e *Baby Steps*

No capítulo anterior, apresentamos TDD em um exemplo complexo do ponto de vista algorítmico, mas bem controlado. Mesmo assim, o capítulo anterior foi muito útil e serviu para mostrar os passos de um desenvolvedor que pratica TDD.

O mundo real é mais complexo do que isso; algoritmos matemáticos são combinados e interagem com entidades que vem do banco de dados e são apresentados para o usuário através de alguma interface. De agora em diante, os exemplos a serem trabalhados serão outros e, com certeza, mais parecido com a realidade. No mundo real, as coisas evoluem e mudam rapidamente. Portanto, nos próximos exemplos, sempre imagine que eles ficarão mais complexos e evoluirão.

4.1 O PROBLEMA DO CÁLCULO DE SALÁRIO

Suponha que uma empresa precise calcular o salário do funcionário e seus descontos. Para calcular esse desconto, a empresa leva em consideração o salário atual e o cargo do funcionário. Vamos representar funcionários e cargos da seguinte maneira:

```
module Cargo
  DESENVOLVEDOR = 0
  DBA = 1
  TESTADOR = 2
end
class Funcionario
  attr_reader :nome, :salario, :cargo
  def initialize(nome, salario, cargo)
    @nome = nome
    @salario = salario
    @cargo = cargo
  end
end
```

Repare que um funcionário possui nome, salário e cargo. O cargo é representado por constantes em um módulo. Nesse momento, o módulo contém apenas constantes para desenvolvedor, testador e DBA. Em uma situação real, esse módulo seria maior ainda; mas com esses 3 já gera uma boa discussão.

As regras de negócio são as seguintes:

- Desenvolvedores possuem 20% de desconto caso seu salário seja maior do que R\$ 3000,0. Caso contrário, o desconto é de 10%.
- DBAs e testadores possuem desconto de 25% se seus salários forem maiores do que R\$ 2500,0. 15%, em caso contrário.

4.2 IMPLEMENTANDO DA MANEIRA MAIS SIMPLES POSSÍVEL

É hora de implementar essas regras. Uma primeira ideia é criar uma classe `CalculadoraDeSalario`, que recebe um `Funcionario` e devolve o salá-

rio do funcionário com o desconto já subtraído.

O primeiro cenário a ser testado será o de desenvolvedores com salários menor do que R\$3000,00. Sabemos que o desconto é de 10%. Portanto, se o desenvolvedor ganhar R\$1500,00, seu salário menos desconto deve ser de R\$ 1350,00 ($1500 * 90\%$):

```
require 'test/unit'
require File.expand_path(
  '../lib/calculadora_de_salario.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/funcionario.rb',
  File.dirname(__FILE__))

class CalculadoraDeSalarioTest < Test::Unit::TestCase
  def
    teste_deve_calcular_salario_para_desenvolvedores_
com_salario_abaixo_do_limite
    calculadora = CalculadoraDeSalario.new
    desenvolvedor = Funcionario.new('Mauricio', 1500.0,
      Cargo::DESENVOLVEDOR)

    salario = calculadora.calcula_salario desenvolvedor

    assert_equal 1500 * 0.9, salario
  end
end
```

Hora de fazer o teste passar. Mas lembre-se que a ideia é fazer o teste passar da maneira **mais simples possível**. Veja o código a seguir:

```
class CalculadoraDeSalario
  def calcula_salario funcionario
    return 1350.0
  end
end
```

É possível ser mais simples do que isso? O código retorna diretamente o valor esperado pelo teste! Isso é aceitável, pois o código ainda não está finalizado.

Vamos agora ao próximo cenário: desenvolvedores que ganham mais do que R\$ 3000.0. O teste é bem similar ao anterior:

```
def
teste_deve_calcular_salario_para_desenvolvedores
_com_salario_acima_do_limite
  calculadora = CalculadoraDeSalario.new
  desenvolvedor = Funcionario.new('Mauricio', 4000.0,
    Cargo::DESENVOLVEDOR)

  salario = calculadora.calcula_salario desenvolvedor

  assert_equal 4000 * 0.8, salario
end
```

Novamente, fazemos o teste passar da maneira mais simples possível. Veja o código:

```
def calcula_salario funcionario
  return 3200.0 if funcionario.salario > 3000
  return 1350.0
end
```

Um simples *if* que verifica o salário do funcionário e retorna os valores esperados.

O próximo teste agora garante que DBAs com salários inferior a R\$1500,00 recebem 15% de desconto:

```
def
teste_deve_calcular_salario_para_DBAs_com_salario_abaixo
do_limite
  calculadora = CalculadoraDeSalario.new
  desenvolvedor = Funcionario.new('Mauricio', 500.0, Cargo::DBA)

  salario = calculadora.calcula_salario desenvolvedor

  assert_equal 500 * 0.85, salario
end
```

Para fazer esse teste passar da maneira mais simples, basta novamente colocar um outro *if*:

```
def calcula_salario funcionario
  if funcionario.cargo == Cargo::DESENVOLVEDOR
    return 3200.0 if funcionario.salario > 3000
    return 1350.0
  end
  return 425.0
end
```

Repare que, se for desenvolvedor, continuamos usando aquele código simples que escrevemos. Caso contrário, retornamos 425.0, que o valor esperado para o salário do DBA.

Mas agora, antes de continuarmos, precisamos discutir o processo utilizado até agora.

4.3 PASSOS DE BEBÊ (OU *BABY STEPS*)

A ideia de sempre tomar o passo mais simples que resolva o problema naquele momento (e faça o teste passar) é conhecido pelos praticantes de TDD como *baby steps*. A vantagem desses passos de bebê é tentar levar o desenvolvedor sempre ao código mais simples e, por consequência, mais fácil de ser compreendido e mantido posteriormente.

O problema é que a ideia dos passos de bebê é muito mal interpretada por muitos praticantes. Por exemplo, veja que, ao final, tínhamos 3 testes, 2 para as regras do desenvolvedor e 1 para a regra do DBA, e a implementação final gerada até então era:

```
def calcula_salario funcionario
  if funcionario.cargo == Cargo::DESENVOLVEDOR
    return 3200.0 if funcionario.salario > 3000
    return 1350.0
  end
  return 425.0
end
```

Repare que, a cada teste, nós implementamos a **modificação mais simples** (TDD prega por simplicidade), que era justamente adicionar mais um *if* e colocar o valor fixo que fazia o teste passar. Não há modificação mais simples do que essa.

O problema é que em algum momento fizemos tanto isso que nossa implementação passou a ficar muito longe da solução ideal. Veja que, dada a implementação atual, levá-la para a solução flexível que resolverá o problema de forma correta não será fácil. O código ficou complexo sem percebermos.

Generalizando o ponto levantado: o desenvolvedor, ao invés de partir para uma solução mais abstrata, que resolveria o problema de forma genérica, prefere ficar postergando a implementação final, com a desculpa de estar praticando passos de bebê.

Imagine isso em um problema do mundo real, complexo e cheio de casos excepcionais. Na prática, o que acontece é que o programador gasta tanto tempo “contornando” a solução que resolveria o problema de uma vez que, quando chega a hora de generalizar o problema, ele se perde. Em sessões de Coding Dojo, onde os programadores se juntam para praticar TDD, é comum ver sessões que duram muito tempo e, ao final, os desenvolvedores pouco avançaram no problema.

Isso é simplesmente uma má interpretação do que são passos de bebê. O desenvolvedor deve buscar a **solução mais simples, e não a modificação mais simples**. Veja que a modificação mais simples não é necessariamente a solução mais simples.

No começo, modificações mais simples podem ajudar o desenvolvedor a pensar melhor no problema que está resolvendo; esse foi o caso quando implementamos o *return 1350.0*. Mas, caso o desenvolvedor continue somente implementando as modificações mais simples, isso pode afastá-lo da solução mais simples, que é o que o desenvolvedor deveria estar buscando ao final. Isso pode ser exemplificado pela sequência de *if* que fizemos onde, ao invés de partirmos para a solução correta (igual fizemos no capítulo anterior), optamos pela modificação mais simples.

CODING DOJO

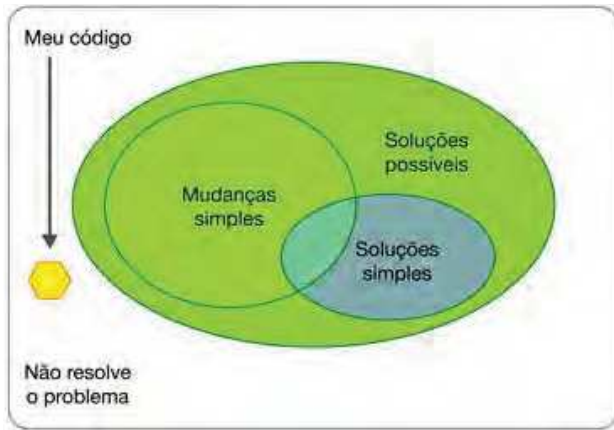
Um *coding dojo* é um encontro de desenvolvedores que estão interessados em aprender alguma coisa nova, como uma linguagem de programação diferente, ou técnicas de desenvolvimento diferentes. O objetivo é criar um ambiente tranquilo e favorável a novos aprendizados.

A prática mais comum é juntar alguns desenvolvedores e projetar uma máquina na parede. Dois desenvolvedores sentam em frente ao computador e começam a resolver algum problema predeterminado, usando a linguagem e/ou a prática que desejam aprender melhor.

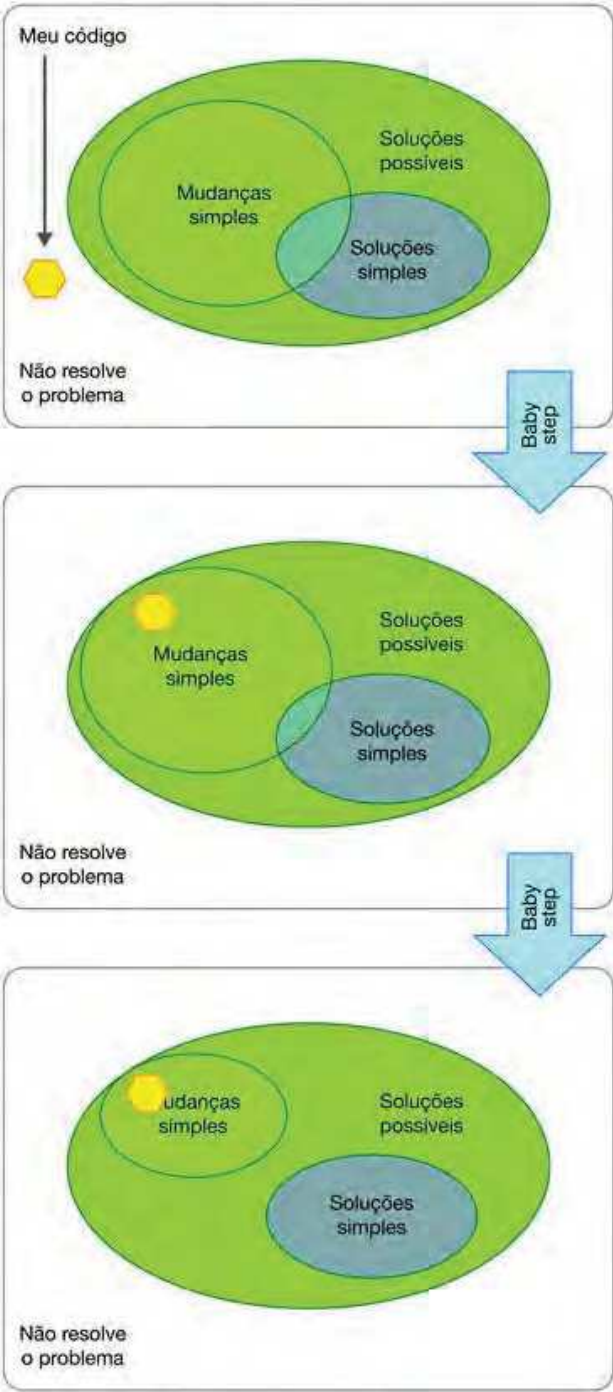
Geralmente esses dois desenvolvedores têm apenas alguns minutos para trabalhar. Enquanto eles trabalham, a plateia assiste e eventualmente sugere alterações para o “piloto e copiloto”. Ao final do tempo (geralmente 7 minutos), o copiloto vira piloto, o piloto volta para a plateia, e alguém da plateia se torna “copiloto”.

Você pode ler mais sobre coding dojos, diferentes estilos, problemas sugeridos nas mais diversas referências na internet [8].

Veja as imagens a seguir. Nela temos o código e as possíveis soluções para o problema. Dentro desse conjunto, existem as que são resolvidas pelas mudanças mais simples, que podem ou não, ser a solução mais simples também para o problema:



Se o desenvolvedor continuamente opta pela modificação mais simples que não é a solução mais simples, isso só vai afastá-lo da melhor solução:



Em nosso exemplo, veja que a solução mais simples acabou sendo a utilização de *ifs*. Sabemos que todo condicional faz o código ser mais difícil de ser mantido e de ser testado. Sabemos que essa não é a melhor solução para o problema; o uso de polimorfismo seria mais adequado. Nesse problema específico, discutiremos como resolvê-lo da melhor forma nos capítulos posteriores.

Novamente, resumindo a discussão: **A mudança mais simples para resolver um problema não é necessariamente a solução mais simples para resolvê-lo.**

Os passos de bebê servem para ajudar o desenvolvedor a entender melhor o problema e diminuir o passo caso ele não se sinta confortável com o problema que está resolvendo. Se o desenvolvedor está implementando um trecho de código complexo e tem dúvidas sobre como fazê-lo, ele pode desacelerar e fazer implementações mais simples; mas caso o desenvolvedor esteja seguro sobre o problema, ele pode tomar passos maiores e ir direto para uma implementação mais abstrata.

O próprio Kent Beck comenta sobre isso em seu livro. Em seu exemplo, ele implementa uma classe `Dinheiro`, responsável por representar uma unidade monetária e realizar operações sobre ela. No começo, ele toma passos simplórios, como os feitos anteriormente, mas finaliza o capítulo dizendo que **ele não toma passos pequenos o tempo todo; mas que fica feliz por poder tomá-los quando necessário.**

CÓDIGO DUPLICADO ENTRE O CÓDIGO DE TESTE E O CÓDIGO DE PRODUÇÃO

Todo desenvolvedor sabe que deve refatorar qualquer duplicidade de código no momento que a encontrar. No próprio problema da soma, a duplicidade já aparece logo no primeiro teste: repare que o “1”, “2” existentes no código de testes está duplicado com o “I” e “II” no código de produção. É fácil ver que cada “I”, adiciona 1 no número final. Sim, você deve ficar atento com duplicidade de código entre a classe de teste e a classe de produção. Nesse caso, após o teste ficar verde, o desenvolvedor poderia já refatorar o código de produção para algo mais genérico, e resolver a duplicidade.

O mesmo acontece em nosso código. No momento em que colocamos o valor “1350.0” fixo em nosso código de produção, geramos uma duplicação. O mesmo valor estava no código de teste: “1500.0 * 0.9”. Nesse momento, o desenvolvedor já poderia ter refatorado e generalizado a solução para algo como `funcionario.salario * 0.9`.

4.4 USANDO *BABY STEPS* DE MANEIRA CONSCIENTE

De volta ao problema da calculadora de salário, vamos repetir a implementação, mas dessa vez usando passos de bebê de forma consciente e produtiva.

Novamente, o primeiro teste:

```
def
teste_deve_calcular_salario_para_desenvolvedores_\
com_salario_abaixo_do_limite
  calculadora = CalculadoraDeSalario.new
  desenvolvedor = Funcionario.new('Mauricio', 1500.0,
    Cargo::DESENVOLVEDOR)

  salario = calculadora.calcula_salario desenvolvedor

  assert_equal 1500 * 0.9, salario
end
```

Vamos fazê-lo passar da maneira mais simples possível:

```
def calcula_salario funcionario
  return 1500.0 * 0.9
end
```

Mas, dessa vez, ao contrário de partir para o próximo teste, vamos remover a duplicação de dado que existe entre o código de teste e o código de produção:

```
def calcula_salario funcionario
  return funcionario.salario * 0.9
end
```

Já está melhor. O código já está genérico e mais perto da solução final. O próximo teste garante o desconto correto para desenvolvedores que ganham acima do limite:

```
def
teste_deve_calcular_salario_para_desenvolvedores_\
com_salario_acima_do_limite
  calculadora = CalculadoraDeSalario.new
  desenvolvedor = Funcionario.new('Mauricio', 4000.0,
    Cargo::DESENVOLVEDOR)

  salario = calculadora.calcula_salario desenvolvedor

  assert_equal 4000 * 0.8, salario
end
```

Nesse momento, não há dúvidas de como será a implementação. Precisamos verificar o salário do desenvolvedor. Se ele for maior que R\$3000,00, devemos descontar 20%. Caso contrário, descontamos os mesmos 10%. Como a implementação está bem clara em nossa cabeça, podemos ir direto para a solução final:

```
def calcula_salario funcionario
  return funcionario.salario * 0.8 if funcionario.salario > 3000
  return funcionario.salario * 0.9
end
```

Veja a diferença dos passos de bebê nos dois testes que escrevemos. No primeiro, tomamos a decisão mais simples possível na hora de fazer o teste passar. Mas logo em seguida, seguimos o ciclo de TDD à risca: removemos a duplicação de código que já existia. No segundo teste, estávamos seguros de como fazer a implementação, e a buscamos diretamente.

Será que praticamos TDD errado? Lembre-se que o objetivo do desenvolvedor não é praticar TDD, mas sim entregar código de qualidade. TDD e passos de bebê estão lá para ajudar, mas não são regra de ouro. Um bom programador sabe a hora de ir mais devagar (geralmente quando está passando por um problema complicado do ponto de vista lógico ou do ponto de vista de design de classes) e realmente usar passos de bebê, e também sabe a hora de acelerar um pouco mais (geralmente quando não há muitas dúvidas sobre como resolver o problema ou sobre como projetar a classe), ganhando em produtividade.

Continuaremos a implementação desta calculadora nos próximos capítulos. Vamos agora finalizar nossa discussão sobre passos de bebê.

UNCLE BOB E OS PASSOS DE REFATORAÇÃO DO TDD

Há alguns anos, Robert Martin fez um post em seu blog sobre uma suposta sequência de passos de refatoração que levariam o desenvolvedor ao melhor código [26]. Seguindo a ideia dos baby steps, ele comenta que um código de produção começa bem específico e caminha até em direção a generalização, para que atenda o problema por completo.

Segundo ele, na hora de generalizar o código, se você usasse baby steps e seguisse as refatorações na sequência que ele descreveu, você sempre chegaria ao melhor algoritmo possível para aquele código. Era como se ele tivesse uma maneira mecânica para se produzir código de qualidade.

Alguns dias depois, eu e o Guilherme Silveira respondemos ao post, mostrando nosso ponto contrário. Em nossa opinião, não é possível criar uma simples sequência de passos para se chegar ao melhor algoritmo possível. Novamente, a experiência e conhecimento do desenvolvedor são necessários.

Por fim, o ponto é que é realmente difícil saber o tamanho do passo a ser dado, e como refatorar o código para que ele fique cada vez melhor. Sua experiência deve ser levada em conta nessa hora.

VANTAGEM DE FAZER O TESTE PASSAR RÁPIDO

Uma das vantagens de fazer o teste passar de maneira simples e rápida é “testar o teste”. Seu teste é código; e ele pode ter erros também.

Como não faz sentido escrever um teste para o teste, uma maneira de testá-lo é garantir que ele falhe quando precisa falhar, e passe quando precisa passar. Ou seja, antes de começar a implementação, veja o teste falhar. Com ele falhando, tente fazê-lo passar da maneira mais simples possível, com o objetivo apenas de vê-lo ficando verde. Dessa forma, você “testou o teste”.

4.5 CONCLUSÃO

Neste capítulo, discutimos sobre os famosos *baby steps*. A ideia é possibilitar ao desenvolvedor andar na velocidade que achar necessário. Se o código que ele está implementando naquele momento é complicado e/ou complexo, tomar passos de bebê pode ajudá-lo a entender melhor o problema e a buscar por soluções mais simples.

Mas, se o código em que ele está trabalhando é algo que já está bem claro e não há muitas dúvidas sobre a implementação, o desenvolvedor pode então dar um passo um pouco maior, já que passos pequenos farão com que ele apenas diminua sua produtividade.

No fim, a grande vantagem dos passos de bebê é poder aprender algo sobre o código para que o mesmo possa ser melhorado. Se o desenvolvedor não está aprendendo nada, então talvez não haja razão para dar passos de bebê naquele instante.

Discutimos também que **a modificação mais simples não é necessariamente a solução mais simples que resolve o problema**. Desenvolvedores devem buscar simplicidade não apenas no nível de código, mas também no nível de design de classes. Às vezes um simples *if* pode ser o código mais simples a ser escrito, mas talvez não seja a melhor solução do ponto de vista do design.

Use *baby steps* com parcimônia. Um bom praticante de TDD sabe a hora de aumentar ou diminuir o passo. Use os passos de bebê para o bem do seu projeto, e não simplesmente porque é uma regra. Parafraseando Jason Gorman, “*se fazer a coisa mais simples significa fazer uso de muitos ifs ou switches, muito provavelmente você não entendeu TDD*”.

CAPÍTULO 5

TDD e design de classes

Como dito anteriormente, TDD é bastante popular pelos seus efeitos positivos no design das classes do nosso sistema. Neste capítulo, começaremos a discutir como os testes podem efetivamente ajudar desenvolvedores a pensar melhor em relação às classes que estão criando.

5.1 O PROBLEMA DO CARRINHO DE COMPRAS

Suponha agora que o projeto atual seja uma loja virtual. Essa loja virtual possui um carrinho de compras, que guarda uma lista de itens comprados. Um item possui a descrição de um produto, a quantidade, o valor unitário e o valor total desse item. Veja o código que representa esse carrinho:

```
class CarrinhoDeCompras
    attr_reader :itens
```

```
def initialize
  @itens = []
end
def <<(item)
  itens << item
end
end
```

Um item também é uma simples classe:

```
class Item
  attr_reader :descricao, :quantidade, :valor_unitario
  def initialize(descricao, quantidade, valor_unitario)
    @descricao = descricao
    @quantidade = quantidade
    @valor_unitario = valor_unitario
  end

  def valor_total
    valor_unitario * quantidade
  end
end
```

Agora imagine que o programador deva implementar uma funcionalidade que devolva o valor do item de maior valor dentro desse carrinho de compras. Pensando já nos testes, temos os seguintes cenários:

- Se o carrinho só tiver um item, ele mesmo será o item de maior valor.
- Se o carrinho tiver muitos itens, o item de maior valor é o que deve ser retornado.
- Um carrinho sem nenhum item deve retornar zero.

Seguindo TDD a risca, vamos começar pelo cenário mais simples, que nesse caso é o carrinho vazio. Vamos criar um teste para a classe `MaiorPreco`, responsável por essa tarefa:

```
require 'test/unit'
require File.expand_path('../lib/item.rb',
```



```
File.dirname(__FILE__))
require File.expand_path(
  '../lib/carrinho_de_compras.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/maior_preco.rb',
  File.dirname(__FILE__))

class TesteMaiorPreco < Test::Unit::TestCase
  def teste_deve_retornar_zero_se_carrinho_vazio
    carrinho = CarrinhoDeCompras.new

    algoritmo = MaiorPreco.new
    valor = algoritmo.encontra carrinho

    assert_equal 0, valor
  end
end
```

Fazer esse teste passar é fácil; basta retornar zero.

```
class MaiorPreco
  def encontra carrinho
    0
  end
end
```

Ainda não há uma repetição de código grande a ser eliminada, então podemos ir para o próximo teste, que é o caso de o carrinho conter apenas um produto. O teste:

```
def teste_deve_retornar_valor_do_item_se_carrinho_com_1_elemento
  carrinho = CarrinhoDeCompras.new
  carrinho << Item.new('Geladeira', 1, 900.0)

  algoritmo = MaiorPreco.new
  valor = algoritmo.encontra carrinho

  assert_equal 900.0, valor
end
```

Para fazer esse teste passar, precisaremos escrever um pouco mais de código. Mas não há muitas dúvidas sobre o algoritmo nesse momento: se o carrinho estiver vazio, retorna 0. Caso contrário, retorna o valor do primeiro elemento desse carrinho:

```
def encontra carrinho
  return 0 if carrinho.itens.empty?
  return carrinho.itens[0].valor_total
end
```

Por fim, o cenário que resta é encontrar o item de maior valor caso o carrinho contenha muitos itens:

```
def teste_deve_retornar_maior_valor_se_carrinho_\
contem_muitos_elementos
  carrinho = CarrinhoDeCompras.new
  carrinho << Item.new('Geladeira', 1, 900.0)
  carrinho << Item.new('Fogão', 1, 1500.0)
  carrinho << Item.new('Máquina de Lavar', 1, 750.0)

  algoritmo = MaiorPreco.new
  valor = algoritmo.encontra carrinho

  assert_equal 1500.0, valor
end
```

Vamos à implementação. É necessário navegar pelos itens da coleção, procurando pelo item de maior valor total. Para armazenar esse maior valor é necessário uma variável chamada “maior”:

```
def encontra carrinho
  return 0 if carrinho.itens.empty?

  maior = carrinho.itens[0].valor_total
  carrinho.itens.each do |item|
    if maior < item.valor_total
      maior = item.valor_total
    end
  end
end
```

```
    maior
end
```

Com todos os testes passando, é hora de avaliá-los. Observe, por exemplo, o último teste que escrevemos. Veja que o teste instancia a classe `MaiorPreco`, passa para ela um “carrinho” e verifica o retorno desse método. Repare que todo o cenário montado foi em cima da classe `CarrinhoDeCompras`; não fizemos nada na classe `MaiorPreco`.

Isso pode ser um mau sinal. Sabemos que uma das vantagens da orientação a objetos em relação a códigos mais procedurais é justamente a capacidade de unir dados e comportamentos. A classe `MaiorPreco` parece “estranha” já que não houve necessidade de setar atributos ou qualquer outro dado nela.

Sabendo desse possível problema na classe, vamos ver seu código. Repare que o método `encontra` faz uso do `carrinho` praticamente o tempo todo, e não faz uso de nada específico da sua própria classe.

A pergunta é: por que criamos essa classe? Por que o método `encontra` não está dentro da própria classe `CarrinhoDeCompras`?

Vamos fazer essa mudança. Se levarmos toda a lógica do método `encontra` para um método chamado, por exemplo, `maior_valor` dentro do `CarrinhoDeCompras`, a classe ficará assim:

```
def maior_valor
  return 0 if itens.empty?

  maior = itens[0].valor_total
  itens.each do |item|
    maior = item.valor_total if maior < item.valor_total
  end
  maior
end
```

Já os testes ficarão assim:

```
require 'test/unit'
require File.expand_path('../lib/item.rb',
  File.dirname(__FILE__))
require File.expand_path(
```

```
    './lib/carrinho_de_compras.rb',
    File.dirname(__FILE__))

class TesteCarrinhoDeCompras < Test::Unit::TestCase
  def teste_deve_retornar_zero_se_carrinho_vazio
    carrinho = CarrinhoDeCompras.new

    assert_equal 0, carrinho.maior_valor
  end
  def
    teste_deve_retornar_valor_do_item_se_carrinho_com_1_elemento
    carrinho = CarrinhoDeCompras.new
    carrinho << Item.new('Geladeira', 1, 900.0)

    assert_equal 900.0, carrinho.maior_valor
  end
  def teste_deve_retornar_maior_valor_\
se_carrinho_contem_muitos_elementos
    carrinho = CarrinhoDeCompras.new
    carrinho << Item.new('Geladeira', 1, 900.0)
    carrinho << Item.new('Fogão', 1, 1500.0)
    carrinho << Item.new('Máquina de Lavar', 1, 750.0)

    assert_equal 1500.0, carrinho.maior_valor
  end
end
```

Veja agora que nosso teste está muito mais claro e elegante. Os testes do `CarrinhoDeCompras` instanciam um carrinho, montam o cenário desejado e invocam um método do próprio carrinho. Agora o código dentro do teste parece melhor e mais orientado a objetos. O comportamento está no lugar esperado. É justamente isso que se espera de um código de qualidade.

USANDO *MAP/REDUCE* PARA UM CÓDIGO MAIS IDIOMÁTICO EM RUBY

Lembrando da abordagem mais funcional, o cálculo do maior valor seria um mapeamento da lista de itens para uma lista com o valor de cada item e uma redução para o maior valor. O código fica assim:

```
def maior_valor
  itens.map(&:valor_total).max || 0
end
```

As bibliotecas padrão do Ruby nos ajudam com duas funcionalidades para isso. A primeira é a de invocação de métodos num mapeamento como em `map(&:valor_total)` que faz com que o método `valor_total` seja chamado em cada elemento da lista que está sendo mapeada. A segunda é o método `max` do módulo `Enumerable` que acha o maior elemento de uma lista comparando os elementos com o método `<=>`. Por fim, como em Ruby `nil` e `false` são as únicas coisas que avaliam como falso em uma condição, o `||` diz que, se não houver resultado no *Map/Reduce*, o método deve devolver o.

Antes de continuarmos, vamos refletir um pouco sobre como descartamos a primeira solução e chegamos à solução que melhor resolveu o problema.

5.2 TESTES QUE INFLUENCIAM NO DESIGN DE CLASSES

O que nos fez optar pelo segundo caminho ao invés do primeiro? Em que momento mudamos o rumo da implementação? Releia os parágrafos anteriores. Repare que o que nos fez mudar de ideia foi justamente o fato de termos encontrado um teste com uma característica estranha (a não-necessidade de um cenário para a classe sob teste).

Esse é o principal ponto deste capítulo. Muitos praticantes de TDD afirmam que a prática lhes guia no projeto de classes. A grande pergunta é: como isso acontece? Isso é outro ponto bastante curioso e mal entendido sobre

TDD. Apesar de muitos praticantes de TDD acreditarem que a prática guia a criação do design, poucos sabem explicar como isso realmente acontece. Este é, aliás, o grande ponto que este livro tenta atacar.

A prática de TDD pode influenciar no processo de criação do projeto de classes. No entanto, ao contrário do que afirmações mais superficiais fazem parecer, **a prática de TDD não guia o desenvolvedor para um bom projeto de classes de forma automática; a experiência e conhecimento do desenvolvedor são fundamentais ao criar software orientado a objetos.**

A prática, por meio dos seus possíveis feedbacks em relação ao projeto de classes, que serão discutidos em detalhes nos capítulos a seguir, pode servir de guia para o desenvolvedor. Esses feedbacks, quando observados, fazem com que o desenvolvedor perceba problemas de projeto de classes de forma antecipada, facilitando a refatoração do código. Um exemplo desses feedbacks foi justamente o que observamos no primeiro teste que escrevemos. O teste não montava qualquer cenário na classe que estava sob teste. Isso simplesmente não parecia correto. São pequenas dicas como essas que ajudam o desenvolvedor a optar por um melhor projeto de classes.

Portanto, esta é a forma na qual a prática guia o desenvolvedor para um melhor projeto de classes: dando retorno constante sobre os possíveis problemas existentes no atual projeto de classes. É tarefa do desenvolvedor perceber esses problemas e melhorar o projeto de acordo.

5.3 DIFERENÇAS ENTRE TDD E TESTES DA MANEIRA TRADICIONAL

O desenvolvedor que pratica TDD escreve os testes antes do código. Isso faz com que o teste de unidade que está sendo escrito sirva de rascunho para o desenvolvedor. Foi novamente o que fizemos. Começamos pelo teste e, ao perceber que o teste “estava estranho”, o jogamos fora e começamos outro. O custo de jogá-lo fora naquele momento era quase zero. Não havíamos escrito uma linha na classe de produção. Tínhamos a chance de começar de novo, sem qualquer perda.

Mas o mesmo feedback poderia ser obtido caso o desenvolvedor deixasse para escrever o teste ao final da implementação. Mas então qual seria a van-

tagem da prática de TDD?

Ao deixar a escrita do teste somente para o final, no momento em que o desenvolvedor perceber um problema de design graças ao teste, o custo de mudança será alto. Leve isso para o mundo real, onde o programador escreve dezenas de classes que interagem entre si e só depois ele escreve o teste.

Todas as decisões de design (boas ou ruins) já foram tomadas. Mudar algo nesse momento pode ser caro, afinal uma mudança em uma classe pode impactar em muitas mudanças nas outras classes que dependem da classe alterada.

O praticante de TDD não tem esse problema. Afinal, graças à escrita do teste antes e à simplicidade inerente do processo, ele escreve um teste para uma pequena funcionalidade, recebe feedback dela, faz as mudanças que achar necessárias e implementa. Depois, ele repete o ciclo. Ou seja, o praticante de TDD recebe feedback constante ao longo do seu dia a dia de trabalho, fazendo-o encontrar possíveis problemas de design mais cedo e, por consequência, diminuindo o custo de mudança.

FISHBOWLING NO ENCONTRO ÁGIL

Ao longo do meu mestrado, participei e conversei com diversas pessoas praticantes ou interessadas em TDD. Em um desses eventos, o Encontro Ágil de 2010, organizei um pequeno debate sobre o assunto [28]. Éramos em torno de 12 pessoas discutindo sobre TDD (eu não participei da discussão, apenas a moderei), e ficamos lá por 1h30.

Durante toda a conversa, praticamente todos os participantes afirmavam firmemente que TDD os ajudava a produzir classes melhores. Mas, quando perguntei “como isso acontece exatamente”, todos eles passaram a falar menos. Eles não sabiam dizer como a prática os guiava, apenas “sentiam” que isso acontecia.

Essa foi uma das grandes motivações da minha pesquisa de mestrado e deste livro sobre o assunto. Realmente não é claro para todos como TDD guia o desenvolvedor. Espero com este livro mostrar, de maneira explícita, como isso acontece.

5.4 TESTES COMO RASCUNHO

Observe o teste:

```
def teste_deve_retornar_valor_do_item_se_carrinho_com_1_elemento
  carrinho = CarrinhoDeCompras.new
  carrinho << Item.new('Geladeira', 1, 900.0)

  assert_equal 900.0, carrinho.maior_valor
end
```

Repare que, apesar de simples, ele deixa bem claro uma grande quantidade de decisões tomadas na classe `CarrinhoDeCompras`:

- O nome do método que calculará o maior valor (`maior_valor`);
- O nome do método que recebe um novo item (`<<`);
- Os parâmetros que esses métodos recebem (um `Item`);
- O retorno do método (um *Float*).

Todas elas são decisões de design! Se alguma dessas decisões não agradar o programador, ele pode simplesmente mudar de ideia. O teste serve como um rascunho para o desenvolvedor, onde ele pode experimentar as diferentes maneiras de se projetar a classe.

E o teste, na verdade, é um excelente rascunho. O teste é o primeiro “cliente” da classe que está sendo criada. Ele é a primeira classe que está interagindo com ela. E olhar para o teste com esse ponto de vista pode fazer sentido. Se está difícil testar um comportamento, é porque, muito provavelmente, sua classe está mal projetada.

Um teste no fim das contas é apenas uma classe que instancia a classe sob teste e invoca um comportamento (método), passando um cenário determinado. Como isso poderia ser difícil? Se está difícil, novamente, talvez sua classe esteja muito acoplada ou seja pouco coesa (possua responsabilidades demais). Um programador atento percebe que está difícil escrever o teste e muda de ideia em relação ao design da classe que está criando. Testar deve ser uma atividade fácil e prazerosa.

O próprio Michael Feathers [19] já escreveu um post sobre isso, e ele resume a discussão dizendo que **existe uma relação muito forte entre uma classe fácil de testar e uma classe que está bem projetada**.

Dada essa grande relação entre um código fácil de ser testado e um código de qualidade, será difícil separar TDD de design de classes daqui para a frente. Por esse motivo, ao final do livro, encontra-se um apêndice sobre design de classes. Nele, discutimos os **princípios SOLID**, um conjunto de 5 boas ideias de orientação a objetos. Esses princípios discutem basicamente as vantagens de se ter classes coesas, pouco acopladas e simples. Caso o leitor não esteja familiarizado com estes princípios, é sugerido que ele o leia; conhecer estes princípios facilitará a leitura dos próximos capítulos.

5.5 CONCLUSÃO

Já vimos que os testes podem influenciar o design das classes que estamos desenvolvendo. Ao observar o código do teste de unidade com atenção, o desenvolvedor pode perceber problemas no projeto de classes que está criando. Problemas como classes que possuem diversas responsabilidades ou que possuem muitas dependências.

Ao escrever um teste de unidade para uma determinada classe, o desenvolvedor é obrigado a passar sempre pelos mesmos passos: a escrita do cenário, a execução da ação sob teste e, por fim, a garantia de que o comportamento foi executado de acordo com o esperado. Uma dificuldade na escrita de qualquer uma dessas partes pode implicar em problemas no projeto de classes. O desenvolvedor, atento, percebe e melhora seu projeto de classes de acordo.

Ao longo do livro, descreveremos padrões que podem ajudar o praticante a perceber problemas de design específicos, como baixa coesão, alto acoplamento e complexidade desnecessária.

CAPÍTULO 6

Qualidade no código do teste

No capítulo anterior, discutimos que os testes podem eventualmente indicar problemas no código do teste. Com essa informação em mãos, desenvolvedores podem refatorar o código e melhorá-lo. Mas, para que isso aconteça, o código do teste deve ser o mais claro e limpo possível.

Escrever testes deve ser algo fácil e produtivo. Todas as boas práticas que o desenvolvedor aplica no código de produção podem ser utilizadas no código do teste também, para que ele fique mais claro e mais reutilizável.

Nas seções a seguir, discutiremos práticas que tornam os testes mais fáceis de ler e manter.

A EQUIPE QUE ODIAVA OS TESTES

Uma vez conversando com um aluno, ele me comentou que a equipe escrevia testes e que o projeto em que atuavam possuía uma bateria de testes imensa, mas pasmem: eles odiavam os testes!

Curioso pelo assunto, comecei a fazer perguntas para entender melhor qual era o real problema deles. Após alguns minutos de conversa, descobri que o que os incomodava era o fato de que qualquer alteração simples no código de produção impactava em profundas alterações no código dos testes. Ou seja, eles gastavam 30 segundos implementando uma nova funcionalidade, e 30 minutos alterando testes.

Por que isso aconteceu? A resposta é simples: falta de cuidado com o código dos testes. Agora nossos testes são automatizados, eles são código. E código ruim impacta profundamente na produtividade da equipe.

Todo o carinho que os desenvolvedores têm com seus códigos de produção deve acontecer também com o código de testes. Evitar repetição de código, isolar responsabilidades em classes específicas etc. são fundamentais para que sua bateria de testes viva para sempre.

6.1 REPETIÇÃO DE CÓDIGO ENTRE TESTES

Veja a classe `TesteCarrinhoDeCompras`. Ela contém apenas 3 testes, mas repare na quantidade de código repetido que já existe. Veja que todos os testes possuem a linha que instancia o carrinho de compras. E, muito provavelmente, os próximos testes para essa classe conterão o mesmo conjunto de linhas.

```
require 'test/unit'
require File.expand_path('../lib/item.rb',
  File.dirname(__FILE__))
require File.expand_path(
  '../lib/carrinho_de_compras.rb',
  File.dirname(__FILE__))

class TesteCarrinhoDeCompras < Test::Unit::TestCase
```

```
def teste_deve_retornar_zero_se_carrinho_vazio
  carrinho = CarrinhoDeCompras.new

  # continuação do teste aqui...
end
def teste_deve_retornar_valor_do_item\
se_carrinho_com_1_elemento
  carrinho = CarrinhoDeCompras.new

  # continuação do teste aqui...
end
def teste_deve_retornar_maior_valor\
se_carrinho_contem_muitos_elementos
  carrinho = CarrinhoDeCompras.new

  # continuação do teste aqui...
end
end
```

Imagine se, por algum motivo, o construtor da classe `CarrinhoDeCompras` mudar. A mudança deverá acontecer em quantos testes? Imagine se essa classe contivesse 100 testes; seria necessário alterar em 100 pontos diferentes.

É bem claro para todo desenvolvedor que qualquer código repetido é problemático. Se surge a necessidade de mudar esse código, ele deve alterar em todos os pontos cujo código se repete. Isso torna a manutenção cara. O mesmo ponto vale para os testes.

A solução, portanto, é escrever esse código em apenas um único lugar. Algo como um método de inicialização da classe de teste. O `TestUnit` possui uma alternativa simples e elegante para isso. Basta definir o método `setup` na sua classe de teste e o `TestUnit` se encarregará de invocar esse método antes de cada teste. Veja como o código fica muito mais claro, com o método `setup` centralizando a criação do cenário inicial. Cada método de teste agora é muito mais simples de ser lido:

```
require 'test/unit'
require File.expand_path('../lib/item.rb',
```

```
File.dirname(__FILE__))
require File.expand_path(
  '../lib/carrinho_de_compras.rb',
  File.dirname(__FILE__))

class TesteCarrinhoDeCompras < Test::Unit::TestCase
  def setup
    @carrinho = CarrinhoDeCompras.new
  end
  def teste_deve_retornar_zero_se_carrinho_vazio
    assert_equal 0, @carrinho.maior_valor
  end
  def teste_deve_retornar_valor_do_item_se_carrinho_\
com_1_elemento
    @carrinho << Item.new('Geladeira', 1, 900.0)

    assert_equal 900.0, @carrinho.maior_valor
  end
  def teste_deve_retornar_maior_valor_se_carrinho_\
contem_muitos_elementos
    @carrinho << Item.new('Geladeira', 1, 900.0)
    @carrinho << Item.new('Fogão', 1, 1500.0)
    @carrinho << Item.new('Máquina de Lavar', 1, 750.0)

    assert_equal 1500.0, @carrinho.maior_valor
  end
end
```

Fazer uso de métodos de inicialização é sempre uma boa ideia. Todo e qualquer código repetido nos testes pode ser levado para métodos como esse. A vantagem? Novamente, código claro.

QUANDO USAR *TEARDOWN*?

Análogo ao `setup`, o método `teardown` diz ao `TestUnit` para executar esse método logo após a execução do teste.

Geralmente fazemos uso dele quando queremos “limpar” alguma sujeira que o teste criou. Em testes puramente de unidade, raramente os utilizamos. Mas imagine um teste que garante que um registro foi salvo no banco de dados, ou que um arquivo XML foi escrito no disco. Após executarmos o método de teste, precisaremos limpar a tabela ou deletar o arquivo gerado.

Portanto, apesar do `teardown` não ser útil em testes de unidade, ele é extremamente importante em testes de integração.

6.2 NOMENCLATURA DOS TESTES

Outro ponto igualmente importante em um teste de unidade é o seu nome. Escolher um bom nome pode fazer uma grande diferença na produtividade do desenvolvedor.

Veja por exemplo, os métodos de teste apresentados na seção anterior. Eles validam o funcionamento do método `maior_valor`. Imagine que a classe de teste fosse parecido com a seguir:

```
class TesteCarrinhoDeCompras < Test::Unit::TestCase
  def maior_valor_1
    # ...
  end
  def maior_valor_2
    # ...
  end
  def maior_valor_3
    # ...
  end
end
```

Se um desses testes falhar, o programador precisará ler o código do teste para entender qual o cenário que faz o comportamento quebrar.

Lembre-se que projetos contêm inúmeros testes de unidade, e se o desenvolvedor precisar ler todo o corpo de um método de teste (mais seu método de inicialização) para entender o que ele faz, perderá muito tempo. É um fato que ler código é muito mais difícil do que escrever código [32]. A escolha de um bom nome do teste pode ajudar o desenvolvedor a ler menos.

Uma sugestão é deixar bem claro no nome do teste qual o comportamento esperado da classe perante determinado cenário. Por exemplo, veja os nomes de testes que demos anteriormente:

```
class TesteCarrinhoDeCompras < Test::Unit::TestCase
  def teste_deve_retornar_zero_se_carrinho_vazio
    # ...
  end
  def teste_deve_retornar_valor_do_item_se_carrinho_com_1
                                                                    _elemento
    # ...
  end
  def teste_deve_retornar_maior_valor_se_carrinho_contem_muitos
                                                                    _elementos
    # ...
  end
end
```

Um método cujo nome tenha um tamanho como esse não seria tão prático em um método de produção. Mas veja que basta o desenvolvedor ler o nome do teste para entender quais são os comportamentos esperados desta classe.

Não há uma regra especial para nomear um método de teste (assim como não há nenhuma regra clara para nomear classes, métodos e variáveis) e, portanto, sua equipe pode definir o padrão que achar melhor. A única ressalva, novamente, é que seu nome deixe claro qual o comportamento esperado e validado naquele método de teste.

6.3 TEST DATA BUILDERS

Uma outra maneira de reduzir a quantidade de linhas gastas com a criação do cenário é criar classes que auxiliam nesse trabalho. Por exemplo, veja a classe

`Carrinho`. Para criar um, é necessário passar uma lista de itens. Ou seja, em todos os futuros testes que serão escritos, gastaremos ao menos 2 linhas para se criar um carrinho. Isso não parece uma boa ideia.

Veja o pseudocódigo adiante. Com apenas 1 linha de código, criamos um `Carrinho` com 2 itens no valor de 200,00 e 300,00 reais, respectivamente, pronto para ser utilizado pelos testes.

```
class CarrinhoDeComprasBuilder
  def cria
    # ...
  end
  def com_itens(*valores)
    # ...
  end
end
carrinho = CarrinhoDeComprasBuilder.new
  .com_itens(200.0, 300.0)
  .cria
```

Fazendo uso do padrão de projeto *Builder* [20], podemos facilmente criar uma classe como essa. Veja o código abaixo:

```
require File.expand_path('./item.rb', File.dirname(__FILE__))
require File.expand_path('./carrinho_de_compras.rb',
                        File.dirname(__FILE__))

class CarrinhoDeComprasBuilder
  def initialize
    @carrinho = CarrinhoDeCompras.new
  end
  def com_itens(*valores)
    valores.each do |valor|
      @carrinho << Item.new('item', 1, valor)
    end
    self
  end
  def cria
    @carrinho
```

```
end  
end
```

A classe `CarrinhoDeComprasBuilder` possibilita a criação de um carrinho com itens de maneira fácil e rápida. Esse *builder* pode ainda ser melhorado, dando opções para se criar itens com diferentes quantidades etc. Lembre-se que você tem todo o poder da orientação a objetos aqui para criar classes que facilitem sua vida.

No fim, você pode implementar *Builders* da maneira que achar mais conveniente. Assim como em todo padrão de projeto, os detalhes de implementação não são a parte mais importante. Lembre-se que a ideia é facilitar o processo de criação de objetos que serão utilizados em cenários de teste. Classes como essas recebem o nome de ***Test Data Builders*** [3].

Test Data Builders são especialmente úteis quando o teste envolve um cenário complexo, cheio de entidades com os mais diferentes valores. Imagine no mundo real, onde cenários são grandes e complicados. Não fazer uso de *Builders* implica em repetição de muitas linhas de código para se criar cenários semelhantes.

Uma outra vantagem da utilização de *Test Data Builders* ao longo do código de teste é a facilidade na evolução desse mesmo código. Se a classe `CarrinhoDeCompras` mudar, todas as classes de teste que fazem uso dela precisarão mudar também. Isso pode ser caro. Ao contrário, se o desenvolvedor sempre cria seus cenários a partir de *Test Data Builders*, qualquer mudança na classe será refletida apenas no *Builder*; bastará ao desenvolvedor mexer nele e todos os testes voltarão a funcionar.

6.4 TESTES REPETIDOS

Uma frase comum entre desenvolvedores de software é que “*melhor do que escrever código, é apagar código!*”. A pergunta é: faz sentido apagar um teste?

A primeira e mais óbvia resposta é: apague o teste quando ele deixar de fazer sentido. Se a funcionalidade foi removida, o desenvolvedor deve atualizar a bateria de testes e apagar todos os testes relacionados a ela. Se a funcionalidade evoluir, você deve evoluir seus testes juntos. Uma bateria de testes desatualizada não serve de nada; só atrapalha o andamento da equipe.

A segunda resposta é: quando a bateria de testes contiver testes repetidos. Em algumas situações, desenvolvedores ficam em dúvida sobre como implementar determinada funcionalidade, e optam por escrever testes que recebem entradas semelhantes. Isso serve para que consigam tomar passos menores (baby steps) e cheguem na solução mais simples para o problema.

No exemplo da Calculadora, o desenvolvedor criaria testes para a soma de 1+1, 1+2 e 2+2, por exemplo:

```
require 'test/unit'
require File.expand_path('../lib/calculadora.rb',
  File.dirname(__FILE__))

class CalculadoraTest < Test::Unit::TestCase
  def teste_deve_somar_um_mais_um
    assert_equal 2, Calculadora.new.soma(1,1)
  end
  def teste_deve_somar_um_mais_dois
    assert_equal 3, Calculadora.new.soma(1,2)
  end
  def teste_deve_somar_dois_mais_dois
    assert_equal 4, Calculadora.new.soma(2,2)
  end
end
```

Esses testes, muito úteis durante o tempo de desenvolvimento do algoritmo, agora se tornaram repetidos. O desenvolvedor deve, portanto, apagá-los. Eles, além de serem inúteis, ainda dificultam o trabalho do desenvolvedor. Se um dia o método testado mudar seu comportamento ou sua interface pública, o desenvolvedor terá que mudar em 10, 20 testes diferentes (mas que no fim testam a mesma coisa). Lembre-se do acoplamento entre seu código de teste e seu código de produção.

Mas será que apenas um teste por funcionalidade é suficiente? Sim. Uma bateria de testes deve ter apenas um teste de unidade para cada conjunto de estados válidos e inválidos para uma condição de entrada. Espera-se que todos os elementos de uma classe se comportem de maneira similar dado duas entradas semelhantes.

Esses conjuntos são conhecidos por **classes de equivalência** [24]. Escrever apenas um teste por classe de equivalência é uma prática muito comum em testes de caixa preta e é conhecida como particionamento em classes de equivalência. Essa prática também faz muito sentido em testes de caixa branca, como os testes de unidade.

No exemplo da calculadora, poderíamos ter testes para as seguintes classes de equivalência:

- soma de dois números positivos;
- soma de um número positivo com outro negativo;
- soma de um número negativo com outro positivo;
- soma de dois números negativos;
- soma com um dos elementos sendo zero;

Por exemplo, repare que os cenários que somam os valores $1+1$ e $5+7$, por exemplo, pertencem à mesma classe e portanto não precisam de testes para cada um destes cenários. Basta o desenvolvedor escolher um deles e escrever o teste.

Uma ótima maneira para garantir que não há testes repetidos é de novo apelar para um bom nome de teste. Por exemplo, se o nome do teste refere-se diretamente aos valores concretos do cenário, como `teste_deve_somar_um_mais_um`, onde o cenário é claramente a soma de $(1 + 1)$, isso sugere ao desenvolvedor que crie um novo teste para a mesma classe de equivalência (a existência do método de teste `teste_deve_somar_dois_mais_dois` não “fica estranho”).

Ao contrário, caso o desenvolvedor opte por dar nomes que representam a classe de equivalência como, por exemplo, `teste_deve_somar_dois_numeros_positivos`, dificilmente o desenvolvedor criará o teste `teste_deve_somar_dois_numeros_positivos_2`.

```
require 'test/unit'
require File.expand_path('../lib/calculadora',
```

```
File.dirname(__FILE__))

class CalculadoraTest < Test::Unit::TestCase
  def teste_deve_somar_dois_numeros_positivos
    assert_equal 4, Calculadora.new.soma(2,2)
  end
  def teste_deve_somar_positivo_com_negativo
    assert_equal 4, Calculadora.new.soma(6,-2)
  end
  def teste_deve_somar_negativo_com_positivo
    assert_equal -4, Calculadora.new.soma(-6,2)
  end
  def teste_deve_somar_dois_numeros_negativos
    assert_equal -4, Calculadora.new.soma(-2,-2)
  end
  def teste_deve_somar_com_zero
    assert_equal 4, Calculadora.new.soma(0,4)
    assert_equal 4, Calculadora.new.soma(4,0)
  end
end
```

Obviamente, encontrar todas as classes de equivalência não é um trabalho fácil, e por isso existe a gigante área de testes de software. Mas fato é que não é repetindo teste que o desenvolvedor garante a qualidade do software produzido.

HUGO FALA

Em Java e C# muitos casos de mudança em códigos de produção podem ser realizados com a ajuda de uma IDE (*Integrated Development Environment*). Em Ruby, dada a natureza dinâmica da linguagem, algumas dessas mudanças são mais difíceis de ser automatizadas. Por conta disso, limpar a bateria de testes frequentemente é ainda mais importante em Ruby que nas outras linguagens.

6.5 ESCRREVENDO BOAS ASSERÇÕES

Boas asserções garantirão que seus testes realmente falharão na hora certa. Mas escrevê-los com qualidade envolve uma série de pequenos detalhes e o programador deve estar atento para eles.

Lembre-se que um bom teste é aquele que falhará quando preciso. E mais, ele mostrará o erro da forma mais clara possível. Um primeiro detalhe simples, mas importante, é a utilização do método `assert_equal`, ou qualquer outro da família. Eles usualmente recebem 2 parâmetros. Como ele comparará se os dois objetos são iguais, então aparentemente a ordem em que os passamos não faz diferença.

```
assert_equal a,b
assert_equal b,a
```

No entanto, a ordem é importante para que o `TestUnit` apresente a mensagem de erro corretamente. No momento em que um teste falha, o `TestUnit` mostra uma mensagem parecida com essa: `nome_do_teste(ArquivoTest) [test/arquivo_test.rb:5]:` `<1> expected but was <0>`, ou seja, esperava `X`, mas o valor calculado foi `Y`. Imagine se essa mensagem viesse ao contrário: `nome_do_teste(ArquivoTest) [test/arquivo_test.rb:5]:` `<0> expected but was <1>`. Isso só dificultaria a vida do programador na hora de entender a causa do problema.

Para que ele exiba a mensagem de erro correta, o desenvolvedor deve passar os parâmetros na ordem: o primeiro parâmetro é o valor esperado, o segundo parâmetro é o valor calculado. Geralmente o primeiro valor é “fixo” ou vem de uma variável declarada dentro do próprio método do teste; já o segundo, vem geralmente de um objeto que foi devolvido pelo método sob teste.

A SOLUÇÃO DO RSpec

No último apêndice do livro discutimos a biblioteca de testes RSpec, uma das mais comuns na comunidade Ruby junto com o TestUnit. RSpec apresenta uma sintaxe que resolve esse problema da ordem dos parâmetros para comparações. Nas versões antigas do RSpec a comparação ficaria assim:

```
a.should == b
```

Nesse caso, fica bem claro que *b* é o valor esperado e que *a* é o valor calculado. O código de um teste normal seria:

```
item.valor_unitario.should == 100.0
```

Dessa forma, fica bem natural colocar os argumentos na ordem certa. A versão mais nova do RSpec mudou a forma como fazer comparações para que a implementação seja mais limpa. O código novo fica assim:

```
expect(item.valor_unitario).to eq(100.0)
```

Note que a sintaxe ainda deixa claro qual o elemento calculado e qual o elemento esperado. No entanto, a nova sintaxe não exige que todo objeto implemente o método `should`.

Ao verificar que o objeto retornado pelo método de teste está correto, o desenvolvedor pode fazer uso de mais de um *assert* no mesmo método de teste. Aliás, em muitos casos, isso é obrigatório como, por exemplo, se o método sob teste devolveu um novo objeto, é obrigação do teste garantir que todo seu conteúdo está correto.

Para isso, o desenvolvedor pode fazer uso de diversos *asserts* no mesmo objeto, um para cada atributo modificado, ou mesmo escrever apenas um, mas verificando o objeto inteiro através do método `==`, que é invocado pelo TestUnit. O desenvolvedor deve escolher a alternativa que mais lhe agrada. Particularmente, prefiro a segunda opção.

```
# alternativa 1
```

```
item = item_que_veio_do_metodo_sob_teste
assert_equal 'Geladeira', item.descricao
assert_equal 900.0, item.valor_unitario
assert_equal 1, item.quantidade
assert_equal 900.0, item.valor_total

# alternativa 2
item = item_que_veio_do_metodo_sob_teste
item_esperado = Item.new('Geladeira', 1, 900.0)
assert_equal item_esperado, item
```

Alguns desenvolvedores também gostam de deixar claro as pré-condições do cenário, e o fazem por meio de *asserts*. Por exemplo, se fossemos testar o método `adiciona` do carrinho de compras, garantiríamos que, após a invocação do comportamento, o item foi guardado na lista.

Para isso, verificaríamos o tamanho da lista, que seria 1, por exemplo. Mas, para que isso seja verdade, precisamos garantir que o carrinho esteja vazio nesse momento. Para tal, uma asserção pode ser escrita antes do comportamento ser invocado.

```
def teste_deve_adicionar_itens
  # garante que o carrinho está vazio
  assert_equal 0, @carrinho.itens.size

  item = Item.new('Geladeira', 1, 900.0)
  @carrinho << item

  assert_equal 1, @carrinho.itens.size
  assert_equal item, @carrinho.itens[0]
end
```

Essa estratégia deve ser usada com parcimônia. O teste que contém *asserts* como esses pode ser mais difícil de ser lido.

Com exceção da necessidade de se escrever mais de 1 *assert* para garantir o conteúdo de objetos retornados pelo método sob teste, desenvolvedores devem evitar ao máximo fazer asserções em mais atributos do que deveriam.

Fuja ao máximo de testes que validam mais de um comportamento. Testes devem ser curtos e testar apenas uma única responsabilidade da classe.

Por exemplo, nos testes da classe `CarrinhoDeCompras`, o teste que valida o método `maior_valor` não deve validar o comportamento do método `adiciona` e vice-versa.

Testes com duas responsabilidades tendem a ser mais complexos (afinal, precisam de mais linhas para montar um cenário que atenda aos dois comportamentos) e dão menos feedback para o desenvolvedor; se o teste falhar, qual das duas responsabilidades não está funcionando de acordo? O desenvolvedor precisará ler o código para entender. Além disso, o nome do método de teste pode ficar confuso, já que o desenvolvedor explicará o comportamento de duas funcionalidades em uma única frase.

6.6 TESTANDO LISTAS

Métodos que retornam listas também merecem atenção especial. Garantir só a quantidade de elementos da lista não é suficiente. Isso não garantirá que os objetos lá dentro estão como esperado. Portanto, o teste deve também validar o conteúdo de cada um dos objetos pertencentes à lista.

```
lista = @carrinho.itens

assert_equal 2, lista.size
assert_equal 100.0, lista[0].valor_unitario
assert_equal 200.0, lista[1].valor_unitario

# asserts no outros atributos, como quantidade,
# etc. nos objetos dessa lista
```

Repare que às vezes seu método pode devolver uma lista com uma quantidade grande de elementos, tornando difícil a escrita dos *asserts*. Tente ao máximo montar cenários que facilitem seu teste, ou encontre um subconjunto de elementos nessa lista que garantam que o resto do conjunto estará correto e teste apenas esses elementos.

MAIS DE UM ASSERT NO TESTE. O QUE ISSO SIGNIFICA?

Em meu doutorado, pesquisei muito sobre a relação do código de testes e a qualidade do código de produção. Em um de meus estudos, ainda em andamento, tenho verificado a relação entre a quantidade de *asserts* em um teste e qualidade do código que está sendo testado.

Apesar de um pouco imaturo, o estudo mostrou que, quando um método faz *asserts* em mais de uma instância de objetos diferentes, o código de produção que está sendo testado tende a ser pior (em termos de complexidade ciclomática e quantidade de linhas de código) do que métodos cujos testes fazem *asserts* em apenas uma única instância de objeto.

Portanto, essa pode ser mais um *feedback* que o teste lhe dá: se você está fazendo *asserts* em objetos diferentes no mesmo teste, isso pode ser um indício de código problemático.

Outra opção, comum na comunidade Ruby mas um pouco mais delicada, é a de asserções parciais na lista. A ideia, nesse caso, é que seus testes sejam menos abrangentes (não olhem para toda a lista) mas sejam específicos entre um e outro com relação ao que está presente. Ao invés de fazer um teste que olhe para a lista toda, fazemos vários testes que olham para itens específicos da lista. O que torna esses testes mais delicados é que é necessário entender a fundo quais são as classes de equivalência e quais elementos são suficientes para cobrir bem o código.

Pensando no caso do carrinho de compras, provavelmente gostaríamos de 3 testes. Um que valida que o carrinho começa vazio, um que verifica que adicionar um único item funciona e outro que garante que também funciona para uma multiplicidade de itens. Nossos testes poderiam ficar assim:

```
def teste_deve_comecar_vazio
  assert_equal 0, @carrinho.itens.size
end
def teste_deve_ter_um_item_apos_adicao
  geladeira = Item.new('Geladeira', 1, 900.0)
  @carrinho << geladeira
```

```
assert_equal 1, @carrinho.itens.size
assert @carrinho.itens.include?(geladeira)
end
def teste_deve_ter_dois_itens_apos_duas_adicoes
  geladeira = Item.new('Geladeira', 1, 900.0)
  @carrinho << geladeira

  fogao = Item.new('Fogão', 1, 600.0)
  @carrinho << fogao

  assert_equal 2, @carrinho.itens.size
  assert @carrinho.itens.include?(fogao)
end
```

Note que os *asserts* não olham para a lista toda. Só verificam que o elemento que é diferente entre o cenário atual e os outros estão presentes. Obviamente, esses testes são dependentes uns dos outros e se o primeiro falhar, todos falharão. Por outro lado, o primeiro que falhar também indica em qual estágio está o problema.

Não há resposta certa com relação a qual estilo é o melhor. Em alguns casos, verificar a lista inteira ajudará a identificar a ação que não está funcionando. Em outros casos, ter os múltiplos testes ajudará a perceber a partir de qual elemento os problemas ocorrem.

6.7 SEPARANDO AS CLASSES DE TESTE

Uma pergunta importante também é onde guardar as classes de teste. A sugestão principal é sempre separar suas classes de produção das classes de teste. O padrão em Ruby é que os arquivos de produção fiquem na pasta 'lib' enquanto os arquivos de teste ficam na pasta 'test' para TestUnit ou 'spec' para RSpec.

Uma outra questão importante é em qual pacote colocar as classes de teste. Em Ruby, o padrão é que as classes de teste fiquem em caminhos idênticos às classes de produção apenas em pastas raízes diferentes ('lib' para produção e 'test' ou 'spec' para TestUnit ou RSpec respectivamente). Dessa forma, fica fácil de encontrar onde está a classe de teste de determinada classe de produção.

Na verdade, esse padrão também é a forma com que a maioria das IDEs

de Ruby conseguem encontrar a classe de teste correspondente à classe de produção e vice-versa para facilitar a navegação.

6.8 CONCLUSÃO

Cuidar dos testes é primordial. Testes de baixa qualidade em algum momento atrapalharão a equipe de desenvolvimento. Por esse motivo, todas as boas práticas de código que desenvolvedores já usam em seus códigos de produção devem ser aplicadas ao código de teste. *Test data builders*, métodos de inicialização, reúso de código para evitar repetição, boas asserções etc., garantirão facilidade de leitura e evolução desses códigos.

Além disso, conforme discutido no capítulo anterior, buscaremos nos testes pequenas dicas sobre a qualidade do código de produção. Isso será difícil se o código do teste estiver bagunçado. Ou seja, se o desenvolvedor tenta manter o código de testes limpo, mas mesmo assim não consegue, há uma grande chance de o código de produção ter problemas.

CAPÍTULO 7

TDD e a coesão

Classes que fazem muita coisa são difíceis de serem mantidas. Ao contrário, classes com poucas responsabilidades são mais simples e mais fáceis de serem evoluídas.

Uma classe coesa é justamente aquela que possui apenas uma única responsabilidade. Em sistemas orientados a objetos, a ideia é sempre buscar escrever classes coesas. Neste capítulo, discutiremos como TDD nos ajuda a encontrar classes com problemas de coesão e, a partir dessa informação, como refatorar o código.

7.1 NOVAMENTE O PROBLEMA DO CÁLCULO DE SALÁRIO

Relembrando o problema, é necessário calcular o salário dos funcionários da empresa a partir do seu cargo. Para isso, é necessário seguir as seguintes regras:

- Desenvolvedores possuem 20% de desconto caso seu salário seja maior do que R\$ 3000,0. Caso contrário, o desconto é de 10%.
- DBAs e testadores possuem desconto de 25% se seus salários forem maiores do que R\$ 2500,0. 15%, em caso contrário.

Já temos alguns testes escritos. Por exemplo, o teste a seguir garante que desenvolvedores com salário inferior a R\$3000,00 recebam apenas 10% de desconto:

```
def
teste_deve_calcular_salario_para_desenvolvedores_\
com_salario_abaixo_do_limite
  calculadora = CalculadoraDeSalario.new
  desenvolvedor = Funcionario.new('Mauricio', 1500.0,
    Cargo::DESENVOLVEDOR)

  salario = calculadora.calcula_salario desenvolvedor

  assert_equal 1500 * 0.9, salario
end
```

A implementação atual, que resolvia o problema da regra de negócios para os desenvolvedores, parava aqui:

```
def calcula_salario funcionario
  return funcionario.salario * 0.8 if funcionario.salario > 3000
  return funcionario.salario * 0.9
end
```

O problema começa a aparecer quando testamos um cargo diferente de desenvolvedor. Por exemplo, os testes a seguir validam o comportamento da classe para DBAs:

```
def teste_deve_calcular_salario_para_DBAs_com_\nsalario_abaixo_do_limite\n  calculadora = CalculadoraDeSalario.new\n  desenvolvedor = Funcionario.new('Mauricio', 1500.0,\n                                   Cargo::DBA)\n\n  salario = calculadora.calcula_salario desenvolvedor\n\n  assert_equal 1500 * 0.85, salario\nend\n\ndef teste_deve_calcular_salario_para_DBAs_\ncom_salario_acima_do_limite\n  calculadora = CalculadoraDeSalario.new\n  desenvolvedor = Funcionario.new('Mauricio', 4500.0,\n                                   Cargo::DBA)\n\n  salario = calculadora.calcula_salario desenvolvedor\n\n  assert_equal 4500 * 0.75, salario\nend
```

Para fazê-los passar, é necessário algum condicional no código de produção para diferenciar o cálculo de desenvolvedores e DBAs, afinal as regras são diferentes. A implementação adiante, por exemplo, resolve o problema e faz todos os testes passarem:

```
def calcula_salario funcionario\n  if funcionario.cargo == Cargo::DESENVOLVEDOR\n    return funcionario.salario * 0.8 if funcionario.salario > 3000\n\n    return funcionario.salario * 0.9\n  elsif funcionario.cargo == Cargo::DBA\n    return funcionario.salario * 0.85 if funcionario.salario < 2500\n\n    return funcionario.salario * 0.75\n  end\n  raise Exception.new('Funcionario invalido')\nend
```

A implementação dos testes para o cargo de testador é bem similar. Basta garantir o comportamento do sistema para funcionários que recebem acima e abaixo do limite. O código de produção também terá uma solução parecida. Basta adicionar um novo *if* para o cargo específico e a implementação será finalizada.

A implementação final seria algo do tipo:

```
def calcula_salario funcionario
  if funcionario.cargo == Cargo::DESENVOLVEDOR
    return funcionario.salario * 0.8 if funcionario.salario >
                                          3000

    return funcionario.salario * 0.9
  elsif funcionario.cargo == Cargo::DBA ||
        funcionario.cargo == Cargo::TESTADOR
    return funcionario.salario * 0.85 if funcionario.salario <
                                          2500

    return funcionario.salario * 0.75
  end
  raise Exception.new('Funcionario invalido')
end
```

Como o código está um pouco extenso, é possível refatorá-lo, extraindo algumas dessas linhas para métodos privados. Por exemplo:

```
class CalculadoraDeSalario
  def calcula_salario funcionario
    if funcionario.cargo == Cargo::DESENVOLVEDOR
      return dez_ou_vinte_por_cento_de_desconto funcionario
    elsif funcionario.cargo == Cargo::DBA ||
          funcionario.cargo == Cargo::TESTADOR
      return
      quinze_ou_vinte_cinco_por_cento_de_desconto funcionario
    end
    raise Exception.new('Funcionario invalido')
  end

  private

  def dez_ou_vinte_por_cento_de_desconto funcionario
    return funcionario.salario * 0.8 if funcionario.salario >
```


3000

```
    return funcionario.salario * 0.9
end
def quinze_ou_vinte_cinco_por_cento_de_desconto funcionario
    return funcionario.salario * 0.85 if funcionario.salario <
    2500

    return funcionario.salario * 0.75
end
end
```

Excelente. É hora de discutir melhor sobre nosso código de testes e código de produção.

HUGO FALA

Os métodos `dez_ou_vinte_por_cento_de_desconto` e `quinze_ou_vinte_cinco_por_cento_de_desconto` ambos têm nomes bem descritivos de seus funcionamentos. No entanto, a descrição é bem próxima do detalhe da implementação. Imagine que os valores dos descontos mudam. A implementação do método precisa mudar mas também os nomes dos métodos. Se o nome do método não mudar, os desenvolvedores que lerão o código ficarão confusos com o nome.

Por este motivo, eu prefiro dar nomes que representem as decisões de negócio ao invés dos detalhes de implementação. Nesse caso, nomes como `desconto_para_desenvolvedor` e `desconto_para_dba_e_testadores` provavelmente seriam minhas escolhas. Costuma ser menos frequente para regras de negócio mudarem drasticamente do que seus parâmetros mudarem.

A outra vantagem de manter um foco no negócio é identificar problemas de modelagem ou de oportunidades de refatoração. Nesse caso, os novos nomes dão uma boa dica de que esses métodos não realmente deveriam estar na `CalculadoraDeSalario` mas nas próprias classes que representam o cargo (Desenvolvedor, DBA e Testador).

```
item = item_que_veio_do_metodo_sob_teste
assert_equal 'Geladeira', item.descricao
assert_equal 900.0, item.valor_unitario
assert_equal 1, item.quantidade
assert_equal 900.0, item.valor_total

# alternativa 2
item = item_que_veio_do_metodo_sob_teste
item_esperado = Item.new('Geladeira', 1, 900.0)
assert_equal item_esperado, item
```

Alguns desenvolvedores também gostam de deixar claro as pré-condições do cenário, e o fazem por meio de *asserts*. Por exemplo, se fossemos testar o método `adiciona` do carrinho de compras, garantiríamos que, após a invocação do comportamento, o item foi guardado na lista.

Para isso, verificaríamos o tamanho da lista, que seria 1, por exemplo. Mas, para que isso seja verdade, precisamos garantir que o carrinho esteja vazio nesse momento. Para tal, uma asserção pode ser escrita antes do comportamento ser invocado.

```
def teste_deve_adicionar_itens
  # garante que o carrinho está vazio
  assert_equal 0, @carrinho.itens.size

  item = Item.new('Geladeira', 1, 900.0)
  @carrinho << item

  assert_equal 1, @carrinho.itens.size
  assert_equal item, @carrinho.itens[0]
end
```

Essa estratégia deve ser usada com parcimônia. O teste que contém *asserts* como esses pode ser mais difícil de ser lido.

Com exceção da necessidade de se escrever mais de 1 *assert* para garantir o conteúdo de objetos retornados pelo método sob teste, desenvolvedores devem evitar ao máximo fazer asserções em mais atributos do que deveriam.

Fuja ao máximo de testes que validam mais de um comportamento. Testes devem ser curtos e testar apenas uma única responsabilidade da classe.

Por exemplo, nos testes da classe `CarrinhoDeCompras`, o teste que valida o método `maior_valor` não deve validar o comportamento do método `adiciona` e vice-versa.

Testes com duas responsabilidades tendem a ser mais complexos (afinal, precisam de mais linhas para montar um cenário que atenda aos dois comportamentos) e dão menos feedback para o desenvolvedor; se o teste falhar, qual das duas responsabilidades não está funcionando de acordo? O desenvolvedor precisará ler o código para entender. Além disso, o nome do método de teste pode ficar confuso, já que o desenvolvedor explicará o comportamento de duas funcionalidades em uma única frase.

6.6 TESTANDO LISTAS

Métodos que retornam listas também merecem atenção especial. Garantir só a quantidade de elementos da lista não é suficiente. Isso não garantirá que os objetos lá dentro estão como esperado. Portanto, o teste deve também validar o conteúdo de cada um dos objetos pertencentes à lista.

```
lista = @carrinho.itens

assert_equal 2, lista.size
assert_equal 100.0, lista[0].valor_unitario
assert_equal 200.0, lista[1].valor_unitario

# asserts no outros atributos, como quantidade,
# etc. nos objetos dessa lista
```

Repare que às vezes seu método pode devolver uma lista com uma quantidade grande de elementos, tornando difícil a escrita dos *asserts*. Tente ao máximo montar cenários que facilitem seu teste, ou encontre um subconjunto de elementos nessa lista que garantam que o resto do conjunto estará correto e teste apenas esses elementos.

MAIS DE UM ASSERT NO TESTE. O QUE ISSO SIGNIFICA?

Em meu doutorado, pesquisei muito sobre a relação do código de testes e a qualidade do código de produção. Em um de meus estudos, ainda em andamento, tenho verificado a relação entre a quantidade de *asserts* em um teste e qualidade do código que está sendo testado.

Apesar de um pouco imaturo, o estudo mostrou que, quando um método faz *asserts* em mais de uma instância de objetos diferentes, o código de produção que está sendo testado tende a ser pior (em termos de complexidade ciclomática e quantidade de linhas de código) do que métodos cujos testes fazem *asserts* em apenas uma única instância de objeto.

Portanto, essa pode ser mais um *feedback* que o teste lhe dá: se você está fazendo *asserts* em objetos diferentes no mesmo teste, isso pode ser um indício de código problemático.

Outra opção, comum na comunidade Ruby mas um pouco mais delicada, é a de asserções parciais na lista. A ideia, nesse caso, é que seus testes sejam menos abrangentes (não olhem para toda a lista) mas sejam específicos entre um e outro com relação ao que está presente. Ao invés de fazer um teste que olhe para a lista toda, fazemos vários testes que olham para itens específicos da lista. O que torna esses testes mais delicados é que é necessário entender a fundo quais são as classes de equivalência e quais elementos são suficientes para cobrir bem o código.

Pensando no caso do carrinho de compras, provavelmente gostaríamos de 3 testes. Um que valida que o carrinho começa vazio, um que verifica que adicionar um único item funciona e outro que garante que também funciona para uma multiplicidade de itens. Nossos testes poderiam ficar assim:

```
def teste_deve_comecar_vazio
  assert_equal 0, @carrinho.itens.size
end
def teste_deve_ter_um_item_apos_adicao
  geladeira = Item.new('Geladeira', 1, 900.0)
  @carrinho << geladeira
```

```
assert_equal 1, @carrinho.itens.size
assert @carrinho.itens.include?(geladeira)
end
def teste_deve_ter_dois_itens_apos_duas_adicoes
  geladeira = Item.new('Geladeira', 1, 900.0)
  @carrinho << geladeira

  fogao = Item.new('Fogão', 1, 600.0)
  @carrinho << fogao

  assert_equal 2, @carrinho.itens.size
  assert @carrinho.itens.include?(fogao)
end
```

Note que os *asserts* não olham para a lista toda. Só verificam que o elemento que é diferente entre o cenário atual e os outros estão presentes. Obviamente, esses testes são dependentes uns dos outros e se o primeiro falhar, todos falharão. Por outro lado, o primeiro que falhar também indica em qual estágio está o problema.

Não há resposta certa com relação a qual estilo é o melhor. Em alguns casos, verificar a lista inteira ajudará a identificar a ação que não está funcionando. Em outros casos, ter os múltiplos testes ajudará a perceber a partir de qual elemento os problemas ocorrem.

6.7 SEPARANDO AS CLASSES DE TESTE

Uma pergunta importante também é onde guardar as classes de teste. A sugestão principal é sempre separar suas classes de produção das classes de teste. O padrão em Ruby é que os arquivos de produção fiquem na pasta 'lib' enquanto os arquivos de teste ficam na pasta 'test' para TestUnit ou 'spec' para RSpec.

Uma outra questão importante é em qual pacote colocar as classes de teste. Em Ruby, o padrão é que as classes de teste fiquem em caminhos idênticos às classes de produção apenas em pastas raízes diferentes ('lib' para produção e 'test' ou 'spec' para TestUnit ou RSpec respectivamente). Dessa forma, fica fácil de encontrar onde está a classe de teste de determinada classe de produção.

Na verdade, esse padrão também é a forma com que a maioria das IDEs

de Ruby conseguem encontrar a classe de teste correspondente à classe de produção e vice-versa para facilitar a navegação.

6.8 CONCLUSÃO

Cuidar dos testes é primordial. Testes de baixa qualidade em algum momento atrapalharão a equipe de desenvolvimento. Por esse motivo, todas as boas práticas de código que desenvolvedores já usam em seus códigos de produção devem ser aplicadas ao código de teste. *Test data builders*, métodos de inicialização, reúso de código para evitar repetição, boas asserções etc., garantirão facilidade de leitura e evolução desses códigos.

Além disso, conforme discutido no capítulo anterior, buscaremos nos testes pequenas dicas sobre a qualidade do código de produção. Isso será difícil se o código do teste estiver bagunçado. Ou seja, se o desenvolvedor tenta manter o código de testes limpo, mas mesmo assim não consegue, há uma grande chance de o código de produção ter problemas.

CAPÍTULO 7

TDD e a coesão

Classes que fazem muita coisa são difíceis de serem mantidas. Ao contrário, classes com poucas responsabilidades são mais simples e mais fáceis de serem evoluídas.

Uma classe coesa é justamente aquela que possui apenas uma única responsabilidade. Em sistemas orientados a objetos, a ideia é sempre buscar escrever classes coesas. Neste capítulo, discutiremos como TDD nos ajuda a encontrar classes com problemas de coesão e, a partir dessa informação, como refatorar o código.

7.1 NOVAMENTE O PROBLEMA DO CÁLCULO DE SALÁRIO

Relembrando o problema, é necessário calcular o salário dos funcionários da empresa a partir do seu cargo. Para isso, é necessário seguir as seguintes regras:

- Desenvolvedores possuem 20% de desconto caso seu salário seja maior do que R\$ 3000,0. Caso contrário, o desconto é de 10%.
- DBAs e testadores possuem desconto de 25% se seus salários forem maiores do que R\$ 2500,0. 15%, em caso contrário.

Já temos alguns testes escritos. Por exemplo, o teste a seguir garante que desenvolvedores com salário inferior a R\$3000,00 recebam apenas 10% de desconto:

```
def
teste_deve_calcular_salario_para_desenvolvedores_\
com_salario_abaixo_do_limite
  calculadora = CalculadoraDeSalario.new
  desenvolvedor = Funcionario.new('Mauricio', 1500.0,
    Cargo::DESENVOLVEDOR)

  salario = calculadora.calcula_salario desenvolvedor

  assert_equal 1500 * 0.9, salario
end
```

A implementação atual, que resolvia o problema da regra de negócios para os desenvolvedores, parava aqui:

```
def calcula_salario funcionario
  return funcionario.salario * 0.8 if funcionario.salario > 3000
  return funcionario.salario * 0.9
end
```

O problema começa a aparecer quando testamos um cargo diferente de desenvolvedor. Por exemplo, os testes a seguir validam o comportamento da classe para DBAs:


```
def teste_deve_calcular_salario_para_DBAs_com_\nsalario_abaixo_do_limite\n  calculadora = CalculadoraDeSalario.new\n  desenvolvedor = Funcionario.new('Mauricio', 1500.0,\n                                   Cargo::DBA)\n\n  salario = calculadora.calcula_salario desenvolvedor\n\n  assert_equal 1500 * 0.85, salario\nend\n\ndef teste_deve_calcular_salario_para_DBAs_\ncom_salario_acima_do_limite\n  calculadora = CalculadoraDeSalario.new\n  desenvolvedor = Funcionario.new('Mauricio', 4500.0,\n                                   Cargo::DBA)\n\n  salario = calculadora.calcula_salario desenvolvedor\n\n  assert_equal 4500 * 0.75, salario\nend
```

Para fazê-los passar, é necessário algum condicional no código de produção para diferenciar o cálculo de desenvolvedores e DBAs, afinal as regras são diferentes. A implementação adiante, por exemplo, resolve o problema e faz todos os testes passarem:

```
def calcula_salario funcionario\n  if funcionario.cargo == Cargo::DESENVOLVEDOR\n    return funcionario.salario * 0.8 if funcionario.salario > 3000\n\n    return funcionario.salario * 0.9\n  elsif funcionario.cargo == Cargo::DBA\n    return funcionario.salario * 0.85 if funcionario.salario < 2500\n\n    return funcionario.salario * 0.75\n  end\n  raise Exception.new('Funcionario invalido')\nend
```

A implementação dos testes para o cargo de testador é bem similar. Basta garantir o comportamento do sistema para funcionários que recebem acima e abaixo do limite. O código de produção também terá uma solução parecida. Basta adicionar um novo *if* para o cargo específico e a implementação será finalizada.

A implementação final seria algo do tipo:

```
def calcula_salario funcionario
  if funcionario.cargo == Cargo::DESENVOLVEDOR
    return funcionario.salario * 0.8 if funcionario.salario >
                                                    3000

    return funcionario.salario * 0.9
  elsif funcionario.cargo == Cargo::DBA ||
    funcionario.cargo == Cargo::TESTADOR
    return funcionario.salario * 0.85 if funcionario.salario <
                                                    2500

    return funcionario.salario * 0.75
  end
  raise Exception.new('Funcionario invalido')
end
```

Como o código está um pouco extenso, é possível refatorá-lo, extraindo algumas dessas linhas para métodos privados. Por exemplo:

```
class CalculadoraDeSalario
  def calcula_salario funcionario
    if funcionario.cargo == Cargo::DESENVOLVEDOR
      return dez_ou_vinte_por_cento_de_desconto funcionario
    elsif funcionario.cargo == Cargo::DBA ||
      funcionario.cargo == Cargo::TESTADOR
      return
      quinze_ou_vinte_cinco_por_cento_de_desconto funcionario
    end
    raise Exception.new('Funcionario invalido')
  end

  private

  def dez_ou_vinte_por_cento_de_desconto funcionario
    return funcionario.salario * 0.8 if funcionario.salario >
```

3000

```
    return funcionario.salario * 0.9
end
def quinze_ou_vinte_cinco_por_cento_de_desconto funcionario
    return funcionario.salario * 0.85 if funcionario.salario <
2500

    return funcionario.salario * 0.75
end
end
```

Excelente. É hora de discutir melhor sobre nosso código de testes e código de produção.

HUGO FALA

Os métodos `dez_ou_vinte_por_cento_de_desconto` e `quinze_ou_vinte_cinco_por_cento_de_desconto` ambos têm nomes bem descritivos de seus funcionamentos. No entanto, a descrição é bem próxima do detalhe da implementação. Imagine que os valores dos descontos mudam. A implementação do método precisa mudar mas também os nomes dos métodos. Se o nome do método não mudar, os desenvolvedores que lerão o código ficarão confusos com o nome.

Por este motivo, eu prefiro dar nomes que representem as decisões de negócio ao invés dos detalhes de implementação. Nesse caso, nomes como `desconto_para_desenvolvedor` e `desconto_para_dba_e_testadores` provavelmente seriam minhas escolhas. Costuma ser menos frequente para regras de negócio mudarem drasticamente do que seus parâmetros mudarem.

A outra vantagem de manter um foco no negócio é identificar problemas de modelagem ou de oportunidades de refatoração. Nesse caso, os novos nomes dão uma boa dica de que esses métodos não realmente deveriam estar na `CalculadoraDeSalario` mas nas próprias classes que representam o cargo (Desenvolvedor, DBA e Testador).

ESCREVER MAIS DE UM TESTE DE UMA SÓ VEZ?

No código anterior, mostrei diretamente dois métodos de teste de uma só vez. A pergunta é: você deve fazer isso? Afinal, isso não é TDD.

Nesse momento, minha sugestão é para que você escreva teste a teste, veja cada um falhar, e faça cada um passar na hora certa. Apenas por questões de didática e facilidade de leitura, coleí ambos de uma só vez.

Mais para frente, discutiremos sobre quando ou não usar TDD.

7.2 OUVINDO O FEEDBACK DOS TESTES

Todos os testes passam e a implementação resolve o problema atual. Mas será que o código está simples e fácil de ser mantido? O código de produção nesse momento possui alguns problemas graves em termos de evolução.

O primeiro deles é a complexidade. Veja que, com apenas 2 cargos implementados, escrevemos 2 *ifs*, com outros *ifs* dentro deles. Quanto maior a quantidade de condicionais, mais complexo o código fica. Essa complexidade, aliás, tem um nome: complexidade ciclomática [36]. Complexidade ciclomática, de maneira simplificada, é o número de diferentes caminhos que seu método pode executar. Por exemplo, quando adicionamos um *if* no código, automaticamente geramos dois diferentes caminhos: um caminho onde a condição do *if* é verdadeiro, e outro caminho onde a condição é falsa. Quanto maior a combinação de *ifs*, *fors*, e coisas do gênero, maior será esse número. Portanto, do ponto de vista de implementação, esse código é bastante complexo.

Agora, do ponto de vista de design, o código atual apresenta um problema ainda pior: sempre que criarmos um novo cargo no sistema (e isso é razoavelmente simples, basta adicionar um novo item no módulo), é necessário fazer essa alteração também na calculadora de salário. Em um sistema real, essas “dependências implícitas” são geralmente uma das causas de sistemas apresentarem constantes problemas, pois o desenvolvedor nunca sabe em quais classes ele precisa mexer para propagar uma nova regra de negócio. Imagine que a cada novo cargo criado, o desenvolvedor precisasse atualizar outras 20

classes? Em algum momento ele esqueceria, afinal nada o “força” a fazer essa alteração. Códigos frágeis como esse são comuns em implementações geralmente procedurais.

Mas o teste, de certa forma, já estava avisando sobre esses problemas. Veja a bateria atual de testes:

```
teste_deve_calcular_salario_para_desenvolvedores_\  
com_salario_abaixo_do_limite
```

```
teste_deve_calcular_salario_para_desenvolvedores_\  
com_salario_acima_do_limite
```

```
teste_deve_calcular_salario_para_DBAs_\  
com_salario_abaixo_do_limite
```

```
teste_deve_calcular_salario_para_DBAs_\  
com_salario_acima_do_limite
```

```
teste_deve_calcular_salario_para_testadores_\  
com_salario_abaixo_do_limite
```

```
teste_deve_calcular_salario_para_testadores_\  
com_salario_acima_do_limite
```

Repare que essa bateria de testes tende a ser infinita. Quanto mais cargos aparecerem, mais testes serão criados nessa classe. Se a classe de testes possui muitos testes, isso quer dizer que a classe de produção possui muitas responsabilidades. E, em sistemas orientados a objetos, sabemos que classes devem ser coesas, conter apenas uma única responsabilidade. Essa classe tende a ter milhares de responsabilidades. Quando uma classe de testes tende a crescer indefinidamente, isso pode ser um sinal de má coesão na classe de produção.

Além disso, repare no nome dos testes: deve calcular salário **para desenvolvedores** com salário abaixo do limite. Repare que esse “para desenvolvedores” indica que o comportamento da classe muda de acordo com uma característica do objeto que o método recebe como parâmetro (no caso, o cargo do funcionário). Isso nos leva a crer que todo novo cargo precisará de um novo teste **para o cargo novo**. Qualquer variação dessa no nome do teste,

(“para X”, “se X”, “como X” etc.) pode indicar um problema na abstração dessa classe. Em um bom sistema orientado a objetos, os comportamentos evoluem naturalmente, geralmente por meio da criação de novas classes, e não pela alteração das classes já existentes.

Ou seja, de bater o olho no código de testes, encontramos duas maneiras baratas de encontrar problemas na classe de produção: classes de teste que não param de crescer e nomes de testes que evidenciam a falta de uma abstração melhor para o problema. O trabalho do desenvolvedor nesse momento é entender por que isso está acontecendo, e eventualmente refatorar a classe de produção para resolver esse problema.

COMPLEXIDADE CICLOMÁTICA E QUANTIDADE DE TESTES

Já que a complexidade ciclomática nos diz a quantidade de diferentes caminhos que um método tem, e já que sabemos que devemos testar todos os diferentes comportamentos do nosso método, é possível inferir que a complexidade ciclomática tem uma relação direta com a quantidade de testes de um método.

Ou seja, quanto maior a complexidade ciclomática, maior a quantidade de testes necessários para garantir seu comportamento. Portanto, um código com alta complexidade ciclomática, além de ser confuso, ainda exige um alto esforço para ser testado.

7.3 TESTES EM MÉTODOS PRIVADOS?

Veja agora o método `quinze_ou_vinte_cinco_por_cento_de_desconto`. Ele é responsável por calcular quinze ou vinte por cento de desconto de acordo com o salário do funcionário. Um método não complicado, mas que possui, sim, alguma regra de negócio envolvida. É necessário testá-lo. A pergunta é como fazer para testá-lo, afinal ele é um método privado, o que o impossibilita de ser invocado diretamente por um teste.

Uma discussão grande em torno da comunidade de software é justamente sobre a necessidade de se testar métodos privados [5]. Alguns desenvolvedores acreditam que métodos como esse devem ser testados, e optam por fa-

zer uso de frameworks que possibilitam a escrita de testes para métodos privados (através de uso de *reflection*, o framework consegue invocar esse método). Outros acreditam que você não deve testá-lo diretamente, mas sim indiretamente, através de um método público que faz uso. No caso, igual feito nos testes anteriores (o método `calcula_salario` invoca o método `quinze_ou_vinte_cinco_por_cento_de_desconto`).

Novamente a discussão sobre o feedback que o teste dá ao desenvolvedor. **Se o desenvolvedor sente a necessidade de testar um método privado de maneira isolada, direta, ou seja, sem passar por um método público que faz uso dele, muito provavelmente é porque esse método privado faz muita coisa.**

Métodos privados são geralmente subdivisões de um método público maior. Optamos por eles para facilitar a leitura desse método público. Mas muitas vezes criamos métodos privados que trabalham demais e possuem uma responsabilidade tão bem definida que poderia constituir uma nova classe.

Ao perceber isso, o desenvolvedor deve se perguntar se o método está realmente no lugar certo. Talvez movê-lo para alguma outra classe, ou até mesmo criar uma nova justamente para acomodar esse comportamento faça sentido.

Portanto, evite testar métodos privados. Leve isso como um feedback sobre a qualidade da sua classe. Extraia esse comportamento para uma nova classe ou mova-o para uma classe já existente. Transforme-o em um método público que faça sentido e aí teste-o decentemente.

7.4 RESOLVENDO O PROBLEMA DA CALCULADORA DE SALÁRIO

Sempre que houver uma separação dos comportamentos em várias pequenas classes, com o objetivo de torná-las mais coesas, é necessário uni-las novamente para se obter o comportamento maior, esperado.

Para fazer isso de maneira elegante, é necessário algum conhecimento em orientação a objetos. Muitos padrões de projeto [20], por exemplo, têm como objetivo unir, de maneira clara, classes que precisam trabalhar juntas.

Para esse novo problema, por exemplo, podemos começar por resolver o problema dos métodos privados. Vamos extraí-los para classes específicas, cada uma responsável por uma regra. Como elas são semelhantes, ambas classes responderão ao mesmo método `calcula`. Essa implementação se parece com o padrão de projeto Strategy:

```
class DezOuVintePorCento
  def calcula funcionario
    return funcionario.salario * 0.8 if funcionario.salario >
                                                    3000

    return funcionario.salario * 0.9
  end
end

class QuinzeOuVinteCincoPorCento
  def calcula funcionario
    return funcionario.salario * 0.85 if funcionario.salario <
                                                    2500

    return funcionario.salario * 0.75
  end
end
```

Repare que cada regra de cálculo agora está em sua classe específica, bem definida. Testar essa classe agora é fácil. Teremos 2 ou 3 testes e pronto. Diferentemente da antiga classe `CalculadoraDeSalario`, a bateria de ambas as regras de cálculo não tendem a crescer infinitamente. Isso nos mostra que essas classes são razoavelmente coesas.

Agora, para forçar o desenvolvedor a sempre definir uma regra de cálculo para todo e qualquer novo cargo, podemos forçá-lo a decidir isso no Cargo. Porém, para isso, precisaremos transformar Cargo em uma classe com um construtor que receberá a regra:

```
class Cargo
  def initialize(regra)
    @regra = regra
  end

  DESENVOLVEDOR = Cargo.new(DezOuVintePorCento.new)
```



```
DBA = Cargo.new(QuinzeOuVinteCincoPorCento.new)
TESTADOR = Cargo.new(QuinzeOuVinteCincoPorCento.new)
attr_reader :regra
private :initialize
end
```

Por fim, com todas as responsabilidades bem definidas, a classe `CalculadoraDeSalario` agora ficará bem mais simples. Veja:

```
class CalculadoraDeSalario
  def calcula_salario funcionario
    funcionario.cargo.regra.calcula funcionario
  end
end
```

Ela somente repassa a chamada para a regra de cálculo. Nosso código agora está muito mais orientado a objetos. Se um novo cargo aparecer, precisamos apenas adicionar na classe `Cargo`. Se uma nova regra de cálculo aparecer, basta criarmos uma nova classe que implementa a interface certa. Todas as classes são pequenas e fáceis de serem testadas. Nosso sistema evolui facilmente e todo código escrito é simples.

PRECISO TESTAR A CLASSE CALCULADORADESALARIO?

A calculadora de salário agora contém apenas uma linha, que repassa para a regra fazer o cálculo.

Talvez essa classe não precise nem existir mais no sistema. Mas, caso exista, talvez não seja necessário testá-la. O teste só garantiria a delegação para a regra específica. Se um dia a calculadora ficar mais complexa, aí sim testes farão mais sentido para ela.

Hugo fala!

A refatoração do código de exemplo da calculadora de salário demonstrou muito bem como a atenção à coesão nos leva a um design melhor das nossas classes. Será que podemos ir além? Outro mau cheiro no código é a

repetição, no caso, da lógica de decisão a respeito do desconto a ser aplicado. Observamos que os métodos `calcula` das duas implementações de regra de cálculo são muito similares. Podemos abstrair essa lógica e fazer com que a `CalculadoraDeSalario` utilize dados definidos dentro das constantes do módulo `Cargo`:

```
class Funcionario
  attr_reader :nome, :salario, :cargo
  def initialize(nome, salario, cargo)
    @nome = nome
    @salario = salario
    @cargo = cargo
  end
end

module Cargo
  DESENVOLVEDOR = {:porcentagem_base => 0.9,
    :porcentagem_acima_do_limite => 0.8,
    :limite => 3000}
  DBA = {:porcentagem_base => 0.85,
    :porcentagem_acima_do_limite => 0.75,
    :limite => 2500}
  TESTADOR = {:porcentagem_base => 0.85,
    :porcentagem_acima_do_limite => 0.75,
    :limite => 2500}
end

require File.expand_path('./funcionario.rb',
  File.dirname(__FILE__))

class CalculadoraDeSalario
  def calcula_salario funcionario
    funcionario.salario * (salario_acima_do_limite(funcionario) ?
      funcionario.cargo[:porcentagem_acima_do_limite] :
      funcionario.cargo[:porcentagem_base])
  end
  def salario_acima_do_limite funcionario
    funcionario.salario > funcionario.cargo[:limite]
  end
end
```

end

Assim eliminaríamos a duplicação; esse tipo de estrutura é chamada de **Template Method**. Mas será mesmo que o design ficou melhor? Não existe uma resposta fácil a essa pergunta, é um *trade-off* entre flexibilidade e abstração, mas na minha opinião eu diria que sim. Em Ruby, não é muito comum ter objetos complexos como constantes. Dá-se preferência para objetos simples (como números, Strings e mapas) que servem de dados de entrada para algum algoritmo como nesse caso.

7.5 O QUE OLHAR NO TESTE EM RELAÇÃO A COESÃO?

Como visto aqui, os testes podem nos avisar sobre problemas de coesão em nossas classes. Lembre-se que classes não coesas fazem muita coisa; testar “muita coisa” não é fácil. Escrever testes deve ser uma tarefa fácil.

Quando um único método necessita de diversos testes para garantir seu comportamento, o método em questão provavelmente é complexo e/ou possui diversas responsabilidades. Códigos assim possuem geralmente diversos caminhos diferentes e tendem a alterar muitos atributos internos do objeto, obrigando o desenvolvedor a criar muitos testes, caso queira ter uma alta cobertura de testes. A esse padrão, dei o nome de **Muitos Testes Para Um Método**.

Também pode ser entendido quando o desenvolvedor escreve muitos testes para a classe como um todo. Classes que expõem muitos métodos para o mundo de fora também tendem a possuir muitas responsabilidades. Chamo esse padrão de **Muitos Testes Para Uma Classe**.

Outro problema de coesão pode ser encontrado quando o programador sente a necessidade de escrever cenários de teste muito grandes para uma única classe ou método. É possível inferir que essa necessidade surge em códigos que lidam com muitos objetos e fazem muita coisa. Nomeei esse padrão de **Cenário Muito Grande**.

A vontade de testar um método privado também pode ser considerado um indício de problemas de coesão. Métodos privados geralmente servem para transformar o método público em algo mais fácil de ler. Ao desejar testá-lo de maneira isolada, o programador pode ter encontrado um método que

possua uma responsabilidade suficiente para ser alocada em uma outra classe. A esse padrão, chamo de **Testes em Método Que Não É Público**.

Lembre-se que esses padrões não dão qualquer certeza sobre o problema. Eles são apenas indícios. O programador, ao encontrar um deles, deve visitar o código de produção e realmente comprovar se existe um problema de coesão. De maneira mais geral, lembre-se que qualquer dificuldade na escrita de um cenário ou o trabalho excessivo para se criar cenários para testar uma única classe ou método pode indicar problemas de coesão.

Classes devem ser simples. E classes simples são testadas de forma simples. A relação entre código de produção e código de teste é realmente forte. A busca pela simplicidade no teste nos leva ao encontro de um código de produção mais simples. E assim que deve ser.

7.6 CONCLUSÃO

Os testes podem nos dar dicas boas e baratas sobre o nível de coesão das nossas classes. Obviamente, você, desenvolvedor, deve estar atento a isso e melhorar o código de acordo.

Mas lembre-se que essas dicas são apenas heurísticas para problemas de coesão. Não é possível garantir que toda a vez que você se deparar com uma classe com muitos testes, ela apresenta problemas de coesão. Em engenharia de software, tudo depende de um contexto. Ou seja, mesmo com esses pequenos padrões, não conseguimos tirar o lado criativo e racional do desenvolvedor. Você deve avaliar caso a caso.

CAPÍTULO 8

TDD e o acoplamento

No capítulo anterior, discutimos a relação entre a prática de TDD e classes coesas. Agora é necessário olhar o outro lado da balança: o acoplamento. Dizemos que uma classe está acoplada a outra quando existe alguma relação de dependência entre elas. Por exemplo, em nosso código anterior, a classe `Funcionario` é acoplada com a classe `Cargo`. Ou seja, ela depende da classe. Isso quer dizer que mudanças na classe podem impactar a classe de forma negativa.

Classes altamente coesas e pouco acopladas são difíceis de serem projetadas. Neste capítulo, discutiremos como TDD ajuda o desenvolvedor a encontrar problemas de acoplamento no seu projeto de classes.

8.1 O PROBLEMA DA NOTA FISCAL

O processo de geração de nota fiscal é geralmente complicado. Envolve uma série de cálculos de acordo com as diversas características do produto vendido. Porém, mais complicado ainda pode ser o fluxo de negócio após a geração da nota: enviá-la para um sistema maior, como um SAP, enviar um e-mail para o cliente com a nota fiscal gerada, persistir as informações na base de dados, e assim por diante.

De maneira simplificada, uma possível representação da Nota Fiscal e Pedido pode ser semelhante ao que segue:

```
class Pedido
  attr_reader :cliente, :valor_total, :quantidade_itens
  def initialize(cliente, valor_total, quantidade_itens)
    @cliente = cliente
    @valor_total = valor_total
    @quantidade_itens = quantidade_itens
  end
end

class NotaFiscal
  attr_reader :cliente, :valor, :data
  def initialize(cliente, valor, data)
    @cliente = cliente
    @valor = valor
    @data = data
  end
end
```

Imagine que hoje esse processo consiste em gerar a nota, persistir no banco de dados e enviá-la por e-mail. No capítulo passado, discutimos sobre coesão e classes com responsabilidades específicas. Para o problema proposto, precisamos de 3 diferentes classes: uma classe responsável por enviar o e-mail, outra responsável por persistir as informações na base de dados, e por fim, uma classe responsável pelo cálculo do valor da nota fiscal.

Para facilitar o entendimento do exemplo, a implementação das classes que se comunicam com o banco de dados e com o SAP serão simplificadas. Veja o código a seguir:

```
class SAP
  def envia nf
    # envia NF para o SAP
  end
end

class NFDao
  def persiste nf
    # persiste NF
  end
end
```

Agora é possível iniciar a escrita da classe `GeradorDeNotaFiscal`, responsável por calcular o valor final da nota e, em seguida, disparar a sequência do processo de negócio.

Imagine que a regra para cálculo da nota fiscal seja simples: o valor da nota deve ser o valor do produto subtraído de 6%. Ou seja, 94% do valor total. Esse é um problema parecido com os anteriores. Começando pelo teste:

```
def teste_deve_gerar_NF_com_valor_de_imposto_descontado
  gerador = GeradorDeNotaFiscal.new
  pedido = Pedido.new('Mauricio', 1000, 1)

  nf = gerador.gera pedido

  assert_equal 1000 * 0.94, nf.valor
end
```

Fazer o teste passar é trivial. Basta instanciarmos uma nota fiscal com 6% do valor a menos do valor do pedido. Além disso, a data da nota fiscal será a data de hoje:

```
require File.expand_path('./nota_fiscal.rb',
                          File.dirname(__FILE__))

class GeradorDeNotaFiscal
  def gera(pedido)
    NotaFiscal.new pedido.cliente,
      pedido.valor_total * 0.94, Time.now
  end
end
```

```
end  
end
```

O teste passa. O próximo passo é persistir essa nota fiscal. A classe `NFDao` já existe. Basta fazermos uso dela. Dessa vez, sem escrever o teste antes, vamos ver como ficaria a implementação final:

```
require File.expand_path('./nota_fiscal.rb',  
                          File.dirname(__FILE__))  
require File.expand_path('./nf_dao.rb', File.dirname(__FILE__))  
  
class GeradorDeNotaFiscal  
  def gera pedido  
    nf = NotaFiscal.new pedido.cliente,  
                        pedido.valor_total * 0.94, Time.now  
  
    NFDao.new.persiste nf  
  
    nf  
  end  
end
```

A grande pergunta agora é: como testar esse comportamento? Acessar o banco de dados e garantir que o dado foi persistido? Não parece uma boa ideia.

8.2 MOCK OBJECTS

A ideia de um teste de unidade é realmente testar a classe de maneira isolada, sem qualquer interferência das classes que a rodeiam. A vantagem desse isolamento é conseguir um maior feedback do teste em relação à classe sob teste; em um teste onde as classes estão integradas, se o teste falha, qual classe gerou o problema?

Além disso, testar classes integradas pode ser difícil para o desenvolvedor. Em nosso exemplo, como garantir que o elemento foi realmente persistido? Seria necessário fazer uma consulta posterior ao banco de dados, garantir que a linha está lá, limpar a tabela para que na próxima vez que o teste for

executado, os dados já existentes na tabela não atrapalhem o teste, e assim por diante.

É muito trabalho. E na verdade, razoavelmente desnecessário. Persistir o dado no banco de dados é tarefa da classe `NFDao`. A tarefa da classe `GeradorDeNotaFiscal` é somente invocar o DAO. Não há por que os testes da `GeradorDeNotaFiscalTest` garantirem que o dado foi persistido com sucesso; isso seria tarefa da classe `NFDaoTest`.

Nesse caso, uma alternativa seria simular o comportamento do `NFDao` no momento do teste do gerador de nota fiscal. Ou seja, queremos criar um clone de `NFDao` que “finja” o acesso ao banco de dados. Classes que simulam o comportamento de outras são chamadas de *mock objects*, ou objetos dublês.

Um *framework* de *mock* facilita a vida do desenvolvedor que deseja criar classes falsas para seus testes. Um *mock* pode ser bem inteligente. É possível ensinar o *mock* a reagir da maneira que queremos quando um método for invocado, descobrir a quantidade de vezes que o método foi invocado pela classe de produção, e assim por diante.

No caso de nosso exemplo, precisamos garantir que o método `dao.persiste` foi invocado pela classe `GeradorDeNotaFiscal`. Nosso teste deve então criar o *mock* e garantir que o método esperado foi invocado.

Aqui utilizaremos o framework conhecido como Mocha [29]. Com ele, criar *mocks* e validar o esperado é fácil. Sua API é bem simples. Veja o teste a seguir, onde criamos o *mock* e depois validamos que o método foi invocado:

```
require 'test/unit'
require 'mocha/setup'
require File.expand_path('../lib/pedido.rb',
  File.dirname(__FILE__))
require File.expand_path(
  '../lib/gerador_de_nota_fiscal.rb',
  File.dirname(__FILE__))

class GeradorDeNotaFiscalTest < Test::Unit::TestCase
  def teste_deve_persistir_NF_gerada
    # criando o mock
    dao = mock
    # marcando o que esperamos que aconteça
```

```

dao.expects(:persiste)

gerador = GeradorDeNotaFiscal.new
pedido = Pedido.new('Mauricio', 1000, 1)

nf = gerador.gera pedido
end
end

```

O problema é que se rodarmos o teste, ele falha. Isso acontece porque, apesar de termos criado o *mock*, o `GeradorDeNotaFiscal` não faz uso dele. Repare que, na implementação, ele instancia um `NFDao`. É necessário que o gerador seja mais esperto: ele deve usar o *mock* no momento do teste, e a classe `NFDao` quando o código estiver em produção.

Uma solução para isso é receber a classe pelo construtor. Dessa forma, é possível passar o *mock* ou a classe concreta. E em seguida, fazer uso do DAO recebido:

```

require File.expand_path('./nota_fiscal.rb',
                          File.dirname(__FILE__))

class GeradorDeNotaFiscal
  def initialize(dao)
    @dao = dao
  end

  def gera pedido
    nf = NotaFiscal.new pedido.cliente,
                        pedido.valor_total * 0.94, Time.now

    @dao.persiste nf

    nf
  end
end

```

Agora, basta passar o *mock* para a classe sob teste e o comportamento do DAO será simulado:

```

def teste_deve_persistir_NF_gerada
  # criando o mock

```

```
dao = mock
# marcando o que esperamos que aconteça
dao.expects(:persiste)

gerador = GeradorDeNotaFiscal.new dao
pedido = Pedido.new('Mauricio', 1000, 1)

nf = generator.gera pedido
end
```

Podemos fazer a mesma coisa com os próximos passos da geração da nota fiscal, como enviar para o SAP, enviar por e-mail etc. Basta passar mais uma dependência pelo construtor, criar outro *mock* e garantir que o método foi enviado. Com o SAP, por exemplo:

```
def teste_deve_enviar_NF_gerada_para_SAP
  dao = stub_everything
  sap = mock.tap{|sap| sap.expects(:envia)}

  generator = GeradorDeNotaFiscal.new dao, sap
  pedido = Pedido.new('Mauricio', 1000, 1)

  nf = generator.gera pedido
end

class GeradorDeNotaFiscal
  def initialize dao, sap
    @dao = dao
    @sap = sap
  end

  def gera pedido
    nf = NotaFiscal.new pedido.cliente,
      pedido.valor_total * 0.94, Time.now

    @dao.persiste nf
    @sap.envia nf

    return nf
  end
end
```

```
end  
end
```

O QUE É `STUB_EVERYTHING`?

`stub_everything` é um método que o *mocha* (*framework* de *mocks*) disponibiliza para criar um objeto que responde com `nil` para qualquer coisa.

No caso desse teste, o `dao` é um objeto que, independente do método que for chamado nele, devolve `nil`.

TDD NO ESTILO LONDRINO

Muitas pessoas conhecem a prática de TDD e *mockar* as dependências como “TDD ao estilo londrino”. Várias das discussões importantes na área de *mock objects* surgiram por lá. Famosos autores como Steve Freeman e Nat Pryce (ambos britânicos) são fãs dessa abordagem.

Independente do nome, agrada-me muito a utilização de *mock objects* durante a escrita dos testes. Quando estou criando uma classe, *mocks* me ajudam a pensar somente no que ela vai fazer e como ela vai interagir com as outras classes do sistema. Nesse momento, preocupo-me muito pouco com as outras classes e foco somente no que espero da classe atual. Sem a utilização de *mocks*, isso seria um pouco mais trabalhoso.

8.3 DEPENDÊNCIAS EXPLÍCITAS

Repare que, por uma necessidade do teste, optamos por receber as dependências da classe pelo construtor. Isso é, na verdade, uma boa prática quando se pensa em orientação a objetos.

Em primeiro lugar o desenvolvedor deixa as dependências da classe explícitas. Basta olhar seu construtor, e ver quais classes são necessárias para que ela faça seu trabalho por completo. Em segundo lugar, ao receber a dependência pelo construtor, a classe facilita sua extensão. Por meio de polimorfismo, o

desenvolvedor pode, a qualquer momento, optar por passar alguma classe que responda aos mesmos métodos, mudando/evoluindo seu comportamento.

Novamente retomo a frase: um código fácil de ser testado possui características interessantes do ponto de vista de design. Explicitar as dependências é uma necessidade quando se pensa em testes de unidade, afinal essa é a única maneira de se passar um *mock* para a classe.

8.4 OUVINDO O FEEDBACK DOS TESTES

A classe `GeradorDeNotaFiscal` tem um problema grave de evolução. Imagine se após a geração da nota houvesse mais 10 atividades para serem executadas, similares a persistir no banco de dados e enviar para o SAP. A classe passaria a receber uma dezena de dependências no construtor, tornando-se altamente acoplada. Classes como essas são um dos múltiplos tipos de **God Classes**, pois elas geralmente contêm pouca regra de negócio, e apenas coordenam o processo de várias classes juntas. Elas costumam acumular muitas responsabilidades e rapidamente tornam-se classes enormes que fazem muito mais do que esparava-se delas.

Repare que, na prática, essa é a balança tradicional em sistemas orientados a objetos. No capítulo anterior, tínhamos uma única classe, pouco acoplada, mas que continha todas as regras de negócio dentro dela, tornando-o difícil de ser lida e testada. Por fim, optamos por separar os comportamentos em pequenas classes. Já neste capítulo, os comportamentos já estão separados, já que a classe `NFDao` contém a lógica de acesso a dados, a classe `SAP` contém a comunicação com o SAP, e assim por diante. Mas é necessário juntar essas pequenas classes para realizar o comportamento total esperado e, ao fazermos, caímos no problema do acoplamento.

Os testes escritos podem nos dar dicas importantes sobre problemas de acoplamento. Um primeiro *feedback* importante é justamente a quantidade de *mock objects* em um teste. Imagine se nossa classe realmente fizesse uso de outras 10 classes; o teste teria 10 *mock objects*. Apesar da não necessidade nesse cenário específico, em muitos testes existe a necessidade de ensinar o *mock object* a reagir de acordo com o esperado. Logo, o teste gastaria muitas linhas simplesmente para montar todos os objetos dublês necessários para o

teste.

Além disso, o “mau uso” de *mock objects* pode ser indício de problemas com a abstração e o excesso de acoplamento. Veja o teste `teste_deve_enviar_NF_gerada_para_SAP`, por exemplo. Repare que, para esse teste, a interação com o dublê do NFDao pouco importa. Ou seja, a classe contém uma dependência que interessa apenas a um subconjunto dos testes, mas não para todos eles. Por que isso aconteceria? Provavelmente existe uma outra maneira de desenhar isso de forma a melhorar o gerenciamento de dependências dessa classe e diminuir seu acoplamento.

8.5 CLASSES ESTÁVEIS

Uma das conhecidas vantagens do TDD é que ele “diminui o acoplamento”, ou “ajuda a diminuir o acoplamento”. A grande pergunta é como ele faz. Observe o código a seguir, que lê um XML e escreve o conteúdo por uma porta serial:

```
class Copiadora
  def copiar
    leitor = LeitorDeXML.new
    escritor = EscriitorPelaSerial.new
    while leitor.tem_caracteres
      escritor.escreve leitor.le_caracteres
    end
  end
end
```

Veja que essa classe é fortemente acoplada com duas classes: `LeitorDeXML` e `EscriitorPelaSerial`. Se quantificarmos isso, podemos dizer que o acoplamento dessa classe é igual a 2 (já que ela está acoplada a 2 classes).

Ao praticar TDD para a criação da classe `Copiadora`, o desenvolvedor provavelmente focará primeiramente na classe `Copiadora` e no que ela deve fazer, esquecendo os detalhes da implementação do leitor e do escritor. Para conseguir esquecer desses detalhes, o desenvolvedor provavelmente definiria interações que representariam as ações esperadas de leitura e escrita:

```
def teste_deve_ler_e_enviar_o_conteudo_lido
  escritor = mock
```

```
leitor = mock

leitor.expects(:tem_caracteres).twice.returns(true, false)
leitor.expects(:le_caracteres).returns('mauricio')

escritor.expects(:escreve).with('mauricio')

copiadora = Copiadora.new leitor, escritor

copiadora.copiar
end
```

Repare que o teste cria dois *mocks*, escritor e leitor. Em seguida, define o comportamento esperado pelo dublê do leitor: o método `tem_caracteres` deve devolver “verdadeiro” e “falso”, nessa ordem, e `le_caracteres` devolvendo o texto “mauricio”. A seguir, cria a `Copiadora`, passando os dublês, invoca o comportamento sob teste. Ao terminar o teste, o *framework* de *mocks* verifica que o escritor recebeu a instrução para escrever a palavra esperada.

Repare que, para esse teste, pouco importa como funcionará a classe que lê e a classe que escreve. Ao praticar TDD, isso é comum. Nós nos preocupamos menos com as classes com que a classe atual interagirá, e mais apenas com os métodos que a classe deverá implementar e sua assinatura.

***DUCK TYPING* OU INTERFACES EM RUBY**

Ruby não possui um elemento na linguagem para definir um tipo abstrato, isto é, um conjunto de métodos que podem ser chamados em objetos daquele tipo. Java e C# definem tipos de várias formas. A mais simples é obviamente uma classe. No entanto, também se pode definir um tipo abstratos em uma classe abstrata, uma enumeração ou uma interface. Ruby só possui classes para definir tipos e não existem tipos abstratos.

Contudo, a comunidade desencoraja o desenvolvimento baseado em tipos em favor de uma técnica conhecida como *Duck Typing*. O termo foi descrito em 2000 da seguinte forma (em tradução livre):

“Em outras palavras, não verifique se É um pato: verifique se *grasne-como-um pato, anda-como-um pato* etc., dependendo exatamente do subconjunto de comportamentos de um pato que você precisa para conseguir fazer o que deseja.”

Ou seja, independente da classe que define determinado objeto em Ruby, só importa se o objeto responde aos métodos que a classe que usa o objeto deseja. O contrato de uso é implícito e não é garantido pelo compilador (já que nem existe um) ou qualquer outra coisa na linguagem. Toda a responsabilidade de garantir que determinado objeto segue o contrato necessário fica por conta dos testes que escrevemos como desenvolvedores.

No restante do livro, sempre que falarmos de interface, estaremos nos referindo ao conjunto de métodos que o objeto deve implementar.

Mas qual a diferença entre ambos os códigos do ponto de vista do acoplamento? Afinal, nas duas implementações, a classe `Copiadora` continua dependendo de 2 outras classes (ou interfaces).

A diferença é em relação à **estabilidade** da segunda classe [25]. Repare que a segunda implementação depende apenas de interfaces e não de classes concretas. O real problema do alto acoplamento é que as dependências de uma classe podem sofrer mudanças, propagando-as para a classe principal.

Logo, quanto maior a dependência, mais instável é a classe, ou seja, maior a chance de ela sofrer uma mudança.



Interfaces, por sua vez, tendem a não mudar muito por várias razões. Primeiro porque elas não contêm detalhes de implementação, atributos ou qualquer outra coisa que tenda a mudar. Segundo porque geralmente interfaces contêm diversas implementações embaixo delas e o desenvolvedor tende a não alterá-la, pois sabe que, se o fizer, precisará alterar em todas as implementações. Logo, se interfaces não mudam (ou mudam muito pouco), acoplar-se com elas pode não ser tão problemático.

Veja no exemplo anterior. As classes `LeitorDeXML` e `EscritorPelaSerial` provavelmente dependem de outras classes (a leitora de XML deve fazer uso de alguma biblioteca para tal, bem como a que escreve pela serial), possuem atributos e muito código para fazer seu serviço. A chance de elas mudarem é alta, e portanto tendem a ser instáveis. Ao contrário, as interfaces `Leitor` e `Escritor` não dependem de nada, não possuem código e classes as implementam. Logo, são estáveis, o que significa que depender delas não é uma ideia tão má.

Podemos dizer que uma boa dependência é uma dependência com um módulo (ou classe) estável, e uma má dependência é uma dependência com um módulo (ou classe) instável. Portanto, se houver a necessidade de se acoplar com alguma outra classe, que seja com uma classe razoavelmente estável.

TDD faz com que o programador acople suas classes com módulos geralmente mais estáveis! TDD força o desenvolvedor a pensar apenas no que você espera das outras classes, sem pensar ainda em uma implementação concreta. Esses comportamentos esperados acabam geralmente se transformando em interfaces, que frequentemente se tornam estáveis.

8.6 RESOLVENDO O PROBLEMA DA NOTA FISCAL

Para gerenciar melhor a dependência da classe `GeradorDeNotaFiscal`, é necessário fazer com que ela dependa de módulos estáveis; uma interface é a melhor candidata. Repare que toda ação executada após a geração da nota, independente de qual for, precisa receber apenas a nota fiscal gerada para fazer seu trabalho.

Logo, é possível definir uma interface para representar todas elas e fazer com que `SAP`, `NFDao`, ou qualquer outra ação a ser executada após a geração da nota, implemente-a:

```
def executa nota_fiscal
end
```

Veja que essa definição de método tende a ser estável: é uma assinatura de método e possui algumas classes concretas que a implementam.

O próximo passo agora é fazer com que o gerador de nota fiscal dependa de uma lista dessas ações, e não mais de classes concretas. Uma sugestão é receber essa lista pelo construtor, já deixando a dependência explícita:

```
class GeradorDeNotaFiscal
  def initialize acoes
    @acoes = acoes
  end

  # ...
end
```

Basta agora fazer com que o gerador, após criar a nota, invoque todas as ações que estão na lista:

```
def gera_pedido
  nf = NotaFiscal.new(pedido.cliente,
    pedido.valor_total * 0.94, Time.now)

  @acoes.each { |acao| acao.executa nf }

  nf
end
```

Repare que a classe `GeradorDeNotaFiscal` não está mais fortemente acoplada a uma ação concreta, mas sim a uma lista qualquer de ações. A chance de ela sofrer uma alteração propagada por uma dependência é bem menor. Além do mais, a evolução dessa classe passa a ser natural: basta passar mais itens na lista de ações, que ela as executará. Não há mais a necessidade de alterar a classe para adicionar um novo comportamento. Em orientação a objetos, a ideia de estendermos o comportamento de uma classe sem alterar seu código é conhecido como **Princípio do Aberto-Fechado**, ou OCP (*Open-Closed Principle*).



Para testarmos, precisamos apenas garantir que as ações da lista, quais-

quer que sejam, serão executadas pelo gerador:

```
def teste_deve_invocar_acoes_posteriores
  acao1 = mock.tap{|dao| dao.expects(:executa)}
  acao2 = mock.tap{|sap| sap.expects(:executa)}

  gerador = GeradorDeNotaFiscal.new [acao1, acao2]
  pedido = Pedido.new('Mauricio', 1000, 1)

  nf = gerador.gera pedido
end
```

O MÉTODO `TAP`

Todo objeto Ruby responde ao método `tap`, que é um método auxiliar para facilitar mudanças simples no próprio objeto. No nosso exemplo, as linhas

```
acao1 = mock.tap{|dao| dao.expects(:executa)}
acao2 = mock.tap{|sap| sap.expects(:executa)}
```

são equivalentes a:

```
acao1 = mock
acao1.expects(:executa)
acao2 = mock
acao2.expects(:executa)
```

`tap` é chamada em um objeto, passa aquele objeto para o bloco que recebeu como parâmetro e devolve o próprio objeto, independente do que aconteceu na execução do bloco. É um método simples para reduzir o número de linhas necessárias para criar um objeto como desejado.

Toda essa refatoração, no fim, foi disparada por um *feedback* do nosso teste: o mau uso ou o uso excessivo de *mock objects*. O teste, avisando o problema de design aliado ao conhecimento de orientação a objetos, proporcionou uma melhoria significativa na classe.

O PERIGO DA DINAMICIDADE COM MOCKS

Como mencionamos no começo do livro, Ruby é uma linguagem muito dinâmica e que permite muita metaprogramação. A consequência dessas características para *mocks* é que basicamente é possível *mockar* **TUDO**.

A linguagem permite basicamente qualquer coisa que você queira fazer e as bibliotecas de *mock* se aproveitam disso. É possível *mockar* atributos, métodos (públicos e privados, de instância e de classe), constantes, classes, módulos e o que mais você bem entender.

O perigo acaba sendo que é muito fácil *mockar* muito mais do que o necessário, desejado ou certo. Escrever testes que só passam por código *mockado* em Ruby é muito fácil e o código do teste não fica tão feio ou suspeito assim. É necessária atenção redobrada para conseguir identificar o *feedback* dos testes e evitar ser enganado pelos seus próprios testes.

Em regra geral, em Ruby, assim como em outras linguagens, se o código do teste não puder ser reescrito sem usar *mocks* (mesmo que com muito mais linhas), provavelmente há um problema de design que não foi identificado. Injeção de dependências e *duck typing* devem permitir que seus testes sejam escritos sem *mocks*. O *mock* deve virar apenas uma ferramenta para facilitar a leitura do teste.

8.7 TESTANDO MÉTODOS ESTÁTICOS

Outro tipo de acoplamento, difícil de ser testado, é o acoplamento com métodos estáticos. Acabamos de ver que, para simular o comportamento das classes, é necessário fazer uso de objetos dublês e, de alguma forma, fazer com que a classe receba e utilize esses dublês. No caso de métodos estáticos, não há como recebê-los pelo construtor.

Em nosso exemplo, imagine que a data da nota fiscal nunca possa ser no fim de semana. Se a nota for gerada no sistema, o sistema deve empurrar sua data para segunda-feira. Em Ruby, para pegarmos a data atual do sistema, fazemos uso do método estático `Time.now`. Como simular seu comporta-

mento e escrever um teste para a geração de uma nota fiscal em um sábado?

Nesses casos, a sugestão é sempre criar uma abstração para facilitar o teste. Ou seja, ao invés de fazer uso direto do método estático, deve-se criar uma classe responsável por devolver a hora atual, e que seja possível de ser *mockada*. Podemos, por exemplo, criar a classe `RelogioDoSistema`:

```
class RelogioDoSistema
  def hoje
    Time.now
  end
end
```

O gerador, por sua vez, passa a fazer uso de um relógio para pegar a data atual:

```
require File.expand_path('./nota_fiscal.rb',
  File.dirname(__FILE__))
require File.expand_path('./relogio_do_sistema.rb',
  File.dirname(__FILE__))

class GeradorDeNotaFiscal
  # construtor com implementação padrão para
  # não quebrar o resto do sistema
  def initialize(acoes, relógio = RelogioDoSistema.new)
    @acoes = acoes
    @relogio = relógio
  end
  def gera pedido
    nf = NotaFiscal.new pedido.cliente,
                        pedido.valor_total * 0.94,
                        @relogio.hoje

    @acoes.each { |acao| acao.executa nf }

    nf
  end
end
```

Recebendo esse `RelogioDoSistema` pelo construtor, o teste é natural.

Basta montar o cenário esperado e verificar a saída do método de acordo com o esperado.

Métodos estáticos são problemáticos. Eles dificultam a evolução da classe já que o desenvolvedor não consegue fazer uso de polimorfismo nem fazer uso de uma implementação diferente desse método. Além disso, eles também dificultam a escrita de testes automatizados.

Evite ao máximo criar métodos estáticos. Só o faça quando o método for realmente simples. Tente sempre optar por classes como parâmetros, fáceis de serem dubladas. Ao lidar com APIs de terceiros, crie abstrações em cima delas única e exclusivamente para facilitar o teste.

Lembre-se: **você pode tomar decisões de design pensando exclusivamente na testabilidade**. Seu código, acima de tudo, deve ser confiável, e para isso deve ser testado.

8.8 TDD E A CONSTANTE CRIAÇÃO DE INTERFACES

Imagine que o problema do gerador de notas fiscais torne-se um pouco mais complicado. Para calcular o valor do imposto, devemos olhar o valor do pedido. Essa tabela contém uma série de faixas de valor, cada uma com uma porcentagem associada. Por exemplo, valores entre zero e R\$999 possuem 2% de imposto, valores entre R\$1000,00 e R\$2999,00 possuem 6% de imposto, e assim por diante. Imagine que essa tabela é razoavelmente grande e está armazenada em um arquivo XML. O gerenciador, com essa porcentagem em mãos, deve apenas multiplicar pelo valor total do pedido.

Para o gerenciador de nota fiscal, não importa como a tabela faz seu trabalho, mas sim o valor que ela retorna. A experiência nos diz que a lógica responsável por essa tabela deve estar em uma classe separada, e o gerenciador deve fazer uso dela.

Podemos deixar isso explícito no teste, passando uma dependência `Tabela` (que ainda nem existe) para o construtor do gerenciador. Além disso, podemos exigir que essa tabela tenha sido consultada pelo gerenciador. O Mocha nos ajudará nisso:

```
def teste_deve_consultar_a_tabela_para_calcular_valor
  relógio = mock.tap{|relógio| relógio.expects(:hoje) }
```

```

tabela = mock
tabela.expects(:[]).with(1000.0).returns(0.2)

gerador = GeradorDeNotaFiscal.new [], relógio, tabela
pedido = Pedido.new('Maurício', 1000, 1)

nf = gerador.gera pedido

assert_equal 1000 * 0.2, nf.valor
end

```

Ao escrever esse teste, definimos uma interface que representa o comportamento esperado do objeto passado como tabela. Essa interface é:

```

def [] valor
end

```

Agora, podemos fazer o teste passar recebendo a tabela no construtor do `GeradorDeNotaFiscal` e definindo o valor padrão desse argumento. Para isso, precisamos criar uma classe `Tabela` vazia. Por fim, fazemos uso desse novo atributo no método `gera`:

```

require File.expand_path('./nota_fiscal.rb',
  File.dirname(__FILE__))
require File.expand_path('./relogio_do_sistema.rb',
  File.dirname(__FILE__))
require File.expand_path('./tabela.rb',
  File.dirname(__FILE__))

class GeradorDeNotaFiscal
  def initialize(acoes, relógio = RelógioDoSistema.new,
    tabela = Tabela.new)
    @acoes = acoes
    @relogio = relógio
    @tabela = tabela
  end
  def gera pedido
    nf = NotaFiscal.new pedido.cliente,
      pedido.valor_total * @tabela[pedido.valor_total],

```



```
@relogio.hoje

@acoes.each { |acao| acao.executa(nf) }

  nf
end
end
```

Com essa interface podemos criar mais testes que fazem uso da tabela, sem, na verdade, nos preocuparmos com a implementação, pois já conhecemos seu contrato. Esse tipo de procedimento é muito comum durante a prática de TDD. Ao criar classes, percebemos que é necessário dividir o comportamento em classes diferentes. Nesse momento, é comum criarmos implicitamente uma interface para representar o comportamento esperado e continuar a criação e testes daquela classe sem nos preocuparmos com a implementação de cada dependência.

Essa nova maneira de programar ajuda, inclusive, a criar contratos mais simples de serem implementados, afinal as interfaces conterão apenas os comportamentos simples, diretos e necessários para que a classe principal faça seu trabalho. Em nosso exemplo, após acabar a implementação do gerador, o próximo passo agora seria criar uma classe como `TabelaDoGoverno`, que contém a implementação real que adere à interface definida pelo teste e descobre a porcentagem de acordo com a faixa de valores. Talvez essa classe precise fazer uso de alguma outra classe. Não há problema: criaríamos novamente implicitamente uma outra interface para representar o comportamento esperado e continuaríamos a trabalhar na tabela.

Portanto, ao praticar TDD, o programador se foca apenas no que ele precisa das outras classes. Isso faz com ele crie contratos estáveis e simples de serem implementados. Veja novamente como o teste e o foco nele fez com que pensássemos no projeto de classes.

ESTILO LONDRINO NA PRÁTICA

É assim que funciona o estilo londrino de TDD. As interfaces vão emergindo a medida que são necessárias. Os testes acabam por deixar explícito como funciona a interação entre essas várias classes.

Nesse momento, o uso de *mock objects* torna-se fundamental para facilitar o teste.

8.9 O QUE OLHAR NO TESTE EM RELAÇÃO AO ACOPLAMENTO?

O uso abusivo de objetos duplês para testar uma única classe indica que a classe sob teste possui problemas de acoplamento. É possível deduzir que uma classe que faz uso de muitos objetos duplês depende de muitas classes, e portanto, tende a ser uma classe instável. A esse padrão, dei o nome de **Objetos Duplê em Excesso**.

A criação de objetos duplês que não são utilizados em alguns métodos de testes é outro feedback importante. Isso geralmente acontece quando a classe é altamente acoplada, e o resultado da ação de uma dependência não interfere na outra. Quando isso acontece, o programador acaba por escrever conjuntos de testes, sendo que alguns deles lidam com um subconjunto dos objetos duplês, enquanto outros lidam com o outro subconjunto de objetos duplês. Isso indica um alto acoplamento da classe, que precisa ser refatorada. A esse padrão dei o nome de **Objetos Duplê Não Utilizados**.

Quando o desenvolvedor começa o teste e percebe que tem dificuldade em expressar sua intenção com os métodos já definidos em outros testes, significa que a interface pública definida pelos testes anteriores não está amigável. Em geral, isso indica que a abstração corrente não é clara o suficiente e poderia ser melhorada. A esse padrão, chamei de **Interface Não Amigável**.

A falta de abstração geralmente também faz com que uma simples mudança precise ser feita em diferentes pontos do código. Quando uma mudança acontece e o programador é obrigado a fazer a mesma alteração em diferentes testes, isso indica a falta de uma abstração correta para evitar a

repetição desnecessária de código. A esse padrão dei o nome de **Mesma Alteração Em Diferentes Testes**. A única exceção em Ruby é no caso de uma mudança de interface por conta de uma **Interface Não Amigável**. Como as IDEs para Ruby não possuem ferramentas de refatoração muito avançadas, é comum que o desenvolvedor precise alterar os testes existentes quando decidir alterar a interface.

Analogamente à **Mesma Alteração em Diferentes Testes**, o programador pode perceber a mesma coisa quando ele começa a criar testes repetidos para entidades diferentes. Chamei esse padrão de **Testes Repetidos Para Entidades Diferentes**.

8.10 CONCLUSÃO

Neste capítulo, vimos padrões de *feedback* que os testes nos dão em relação ao acoplamento da classe. A grande maioria deles está totalmente relacionado ao mau uso de *mock objects*.

Discutimos também que, para um melhor gerenciamento de dependências, classes devem sempre tentar depender de módulos estáveis, diminuindo assim as chances da propagação de mudanças para a classe principal. Isso é alcançado através do bom uso de orientação a objetos e interfaces.

É difícil lidar com a balança de acoplamento e coesão. Classes coesas são fáceis de serem lidas e testadas, mas, para que o sistema se comporte como o esperado, é necessário juntar todas essas pequenas classes. Nesse momento é quando deixamos o lado do acoplamento se perder. Infelizmente é impossível criar um sistema cujas classes são todas altamente coesas a pouco acopladas. Mas, espero que, ao final desses dois capítulos, eu tenha mostrado como atacar ambos os problemas e criar classes que balanceiam entre esses dois pontos-chave da orientação a objetos.

CAPÍTULO 9

TDD e o encapsulamento

Já discutimos sobre coesão e acoplamento nos capítulos anteriores, e vimos como os testes podem nos dar informações valiosas sobre esses dois pontos. Um próximo ponto importante em sistemas orientados a objetos é o **encapsulamento**.

Encapsular é esconder os detalhes de **como** a classe realiza sua tarefa; as outras classes devem conhecer apenas **o quê** ela faz. Ao não encapsular corretamente regras de negócios em classes específicas, desenvolvedores acabam por criar código repetido, e espalhar as regras de negócio em diferentes partes do sistema.

Neste capítulo, discutiremos como os testes podem nos ajudar a encontrar regras de negócio que não estão bem encapsuladas.

9.1 O PROBLEMA DO PROCESSADOR DE BOLETO

É comum que uma fatura possa ser paga por meio de diferentes boletos. Para resolver esse problema, um processador de boletos passa por todos os boletos pagos para uma fatura, e simplesmente faz o vínculo de ambos.

Antes de começarmos, suponha a existência das classes `Fatura`, `Boleto` e `Pagamento`. Uma `Fatura` contém uma lista de `Pagamento` que, por sua vez, armazena um valor e uma forma da pagamento (boleto, cartão de crédito etc.). Um `Boleto` contém apenas o valor pago do boleto.

De antemão, já é possível imaginar alguns cenários para esse processador:

- Usuário pagou com apenas um boleto.
- Usuário utilizou mais de um boleto para pagar.

Começando pelo primeiro cenário, mais simples, temos o seguinte teste, que cria um único boleto, invoca o processador, e ao final garante que o pagamento foi criado na fatura:

```
require 'test/unit'
require File.expand_path('../lib/processador_de_boletos.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/fatura.rb',
  File.dirname(__FILE__))
require File.expand_path('../lib/boleto.rb',
  File.dirname(__FILE__))

class ProcessadorDeBoletosTest < Test::Unit::TestCase
  def teste_deve_processar_pagamento_via_boleto_unico
    processador = ProcessadorDeBoletos.new

    fatura = Fatura.new('Cliente', 150.0)
    b1 = Boleto.new(150.0)

    processador.processa([b1], fatura)

    assert_equal 1, fatura.pagamentos.size
    assert_equal 150.0, fatura.pagamentos[0].valor
  end
end
```

```
end  
end
```

A implementação para fazer o teste passar é simples:

```
require File.expand_path('./pagamento.rb',  
  File.dirname(__FILE__))  
  
class ProcessadorDeBoletos  
  def processa boletos, fatura  
    boleto = boletos[0]  
    pagamento =  
      Pagamento.new boleto.valor, MeioDePagamento::BOLETO  
    fatura.pagamentos << pagamento  
  end  
end  
  
module MeioDePagamento  
  BOLETO = 0  
end
```

Agora, o próximo cenário é garantir que o processador de boletos consiga processar um usuário que fez o pagamento por mais de um boleto:

```
def teste_deve_processar_pagamento_via_muitos_boletos  
  processador = ProcessadorDeBoletos.new  
  
  fatura = Fatura.new('Cliente', 300.0)  
  b1 = Boleto.new 100.0  
  b2 = Boleto.new 200.0  
  
  processador.processa [b1, b2], fatura  
  
  assert_equal 2, fatura.pagamentos.size  
  assert_equal 100.0, fatura.pagamentos[0].valor  
  assert_equal 200.0, fatura.pagamentos[1].valor  
end
```

Para fazer o teste passar, basta navegar pela lista de boletos e criar um pagamento para cada um deles:

```
require File.expand_path('./pagamento.rb',
                        File.dirname(__FILE__))

class ProcessadorDeBoletos
  def processa boletos, fatura
    boletos.each do |boleto|
      pagamento =
        Pagamento.new boleto.valor, MeioDePagamento::BOLETO
      fatura.pagamentos << pagamento
    end
  end
end

module MeioDePagamento
  BOLETO = 0
end
```

Com o processador de boletos já funcionando, o próximo passo agora é marcar a fatura como paga, caso o valor dos boletos pagos seja igual ou superior ao valor da fatura. Ou seja, o sistema deve se comportar da seguinte maneira para os cenários a seguir:

- Se o usuário pagar um único boleto com valor inferior ao da fatura, ela não deve ser marcada como paga.
- Se o usuário pagar um único boleto com valor superior ao da fatura, ela deve ser marcada como paga.
- Se o usuário pagar um único boleto com valor igual ao da fatura, ela deve ser marcada como paga.
- Se o usuário pagar vários boletos e a soma deles for inferior ao da fatura, ela não deve ser marcada como paga.
- Se o usuário pagar vários boletos e a soma deles for superior ao da fatura, ela deve ser marcada como paga.
- Se o usuário pagar vários boletos e a soma deles for igual ao da fatura, ela deve ser marcada como paga.

Apesar da quantidade de cenários ser grande, a implementação não é tão complicada assim. Precisamos guardar a soma de todos os boletos pagos e, ao final, verificar se ela é maior ou igual ao valor da fatura. Em caso positivo, basta marcar a fatura como paga.

Novamente começando pelo cenário mais simples:

```
def
  teste_deve_marcar_fatura_como_paga_caso_boleto_unico_pague_tudo
    processador = ProcessadorDeBoletos.new

    fatura = Fatura.new('Cliente', 150.0)
    b1 = Boleto.new 150.0

    processador.processa [b1], fatura

    assert fatura.paga?
  end
```

Fazendo o teste passar da maneira que discutimos anteriormente:

```
def processa boletos, fatura
  valor_total = 0
  boletos.each do |boleto|
    pagamento =
      Pagamento.new boleto.valor, MeioDePagamento::BOLETO
    fatura.pagamentos << pagamento

    valor_total += boleto.valor
  end

  fatura.paga = true if valor_total >= fatura.valor
end
```

Com todos os testes verdes, é hora de discutirmos sobre a implementação feita até então.

DEVO ESCREVER OS OUTROS TESTES?

Mesmo que a implementação pareça já resolver todos os cenários propostos, é importante que esses testes sejam automatizados.

Lembre-se que amanhã a implementação poderá mudar. Como garantir que a nova implementação funcionará para todos os casos em que a implementação atual funciona? Todo código que você acabou de escrever parece simples e funcional, mas não se esqueça que ele ficará lá para sempre e será mantido por muitas outras pessoas.

Portanto, não se deixe enganar. Escreva os testes e garanta que nenhuma futura evolução quebrará o que já funciona hoje.

9.2 OUVINDO O FEEDBACK DOS TESTES

Apesar da classe anterior atender a regra de negócio, o código produzido não é dos melhores. O que aconteceria com o sistema caso precisássemos criar agora um processador de cartão de crédito? Seria necessário repetir a mesma lógica de marcar uma fatura como paga lá. Ou seja, a cada nova forma de pagamento, trechos de código seriam replicados entre classes. E não há necessidade de se discutir os problemas de código repetido.

O princípio ferido aqui é justamente o encapsulamento. As classes devem ser responsáveis por manter o seu próprio estado; elas é quem devem conhecer as suas próprias regras de negócio. Quando a regra não é seguida, pedaços da regra de negócio são espalhadas pelo código. É exatamente isso o que aconteceu no exemplo: a regra de marcar uma fatura como paga está fora da classe `Fatura`.

O teste, por sua vez, nos dá dicas sobre esse problema. Veja, por exemplo, qualquer um dos testes escritos na classe `ProcessadorDeBoletoTest`. Todos eles fazem asserções na classe `Fatura`. A pergunta é: por que os testes de uma classe fazem asserções somente em outras classes?

Responder essa pergunta pode ser difícil. Essa valiosa dica pode estar nos avisando sobre possíveis problemas de encapsulamento na classe que recebe a asserção. Neste caso, a classe `Fatura` não está encapsulando bem as suas

regras de negócio.

9.3 *TELL, DON'T ASK* E LEI DE DEMETER

Classes como a `ProcessadorDeBoletos`, que “conhecem demais” sobre o comportamento de outras classes não são bem vistas em sistemas orientados a objetos. É comum ouvir o termo **intimidade inapropriada**, já que a classe conhece detalhes que não deveria de alguma outra classe.

Repare na implementação do método `processa` em relação ao processo de marcar a fatura como paga. Veja que ele faz uma pergunta para a fatura (pergunta o valor dela), e depois, com a resposta em mãos, toma uma ação (marca ou não a fatura como paga). Em códigos orientados a objetos, geralmente dizemos que as classes não devem fazer perguntas e tomar decisões baseadas nas respostas, mas sim **mandar** o objeto executar uma ação, e ele por conta própria tomar a decisão certa. Essa ideia é conhecida por *Tell, Don't Ask*.

Em nosso exemplo, a ideia seria mandar a classe fatura se marcar como paga, caso necessário, afinal ela deve ser a responsável por manter seu próprio estado. Essa ação poderia ser tomada, por exemplo, no momento em que adicionamos um novo pagamento.

Se a Fatura tivesse um método `adiciona_pagamento(pagamento)`, que, além de adicionar o pagamento na fatura, ainda somasse os valores pagos e marcasse a fatura como paga se necessário, o problema estaria resolvido. Não teríamos mais o `each` do lado de fora da classe. Essa regra estaria dentro da classe, encapsulada.

Uma outra maneira de descobrir por possíveis problemas de encapsulamento é contando a quantidade de métodos invocados em uma só linha dentro de um único objeto. Por exemplo, imagine o seguinte código:

```
a.b.c.d.faz_alguma_coisa
```

Para invocar o comportamento desejado, partimos de “a”, pegamos “b”, “c” e “d”. Relembre agora a discussão sobre acoplamento. Quando uma classe depende da outra, mudanças em uma classe podem se propagar para a classe principal. Perceba que a classe que contém não está acoplada somente à classe

do tipo do atributo “a”, mas indiretamente também aos tipos de “b”, “c” e “d”. Qualquer mudança na estrutura deles pode impactar na classe principal.

Isso pode ser considerado também um problema de encapsulamento. A classe do tipo “a” está expondo demais sua estrutura interna. O “mundo de fora” sabe que ela lá dentro tem “b”, que por sua vez, tem “c”, “d”, e assim por diante. Se o comportamento `faz_alguma_coisa` deve realmente ser invocado, pode-se fazer algo como um método dentro de “a” que esconde o processo de invocá-lo:

```
class A
  def faz_alguma_coisa
    b.c.faz_alguma_coisa
  end
end

A.new.faz_alguma_coisa
```

Diminuir a quantidade de iterações com os objetos, ou seja, navegar menos dentro deles, é o que conhecemos por **Lei de Demeter**. Ela nos diz justamente isso: tente nunca conversar com classes que estão dentro de classes; para isso, crie um método que esconda esse trabalho pra você. Dessa forma, você encapsula melhor o comportamento esperado e ainda reduz o acoplamento.

DEVO SEGUIR A LEI DE DEMETER À RISCA?

Como tudo em engenharia de software, não. A Lei de Demeter, apesar de muito interessante, às vezes pode mais atrapalhar do que ajudar.

Ruby facilita tomar essa decisão com a ajuda do módulo `Forwardable`. Esse módulo define os métodos `delegator` e `delegators`, que permitem que, com uma linha de código, seus métodos possam ser enviados para uma de suas dependências sem poluir o código da classe original.

No caso do exemplo anterior, para deixar o atributo `b` ser responsável por `faz_alguma_coisa`, teríamos:

```
class A
  require 'forwardable'

  def_delegator @b, :faz_alguma_coisa
end

A.new.faz_alguma_coisa
```

Geralmente não ligo para linhas como `pessoa.endereco.rua`, pois estamos apenas pegando dados de um objeto. Não faz sentido criar um método `pessoa.rua`, ou `pessoa.x` para todo dado que representa uma pessoa. Esses métodos não fazem muito sentido.

Essa é geralmente a minha regra: para exibição, aceito não seguir a regra. Mas para invocar um comportamento, geralmente penso duas vezes antes de não seguir.

9.4 RESOLVENDO O PROBLEMA DO PROCESSADOR DE BOLETOS

Conforme já discutido aqui, é necessário levar a regra de negócios da marcação da fatura como paga para dentro da própria classe `Fatura`. Seguindo a ideia do método `adiciona_pagamento`, faremos com que ele adicione o pagamento e marque a fatura como paga caso necessário:

```
def adiciona_pagamento pagamento
  @pagamentos << pagamento

  valor_total = @pagamentos.map(&:valor).reduce(:+)

  @paga = true if valor_total >= @valor
end
```

O processador de boletos volta a ficar com o código enxuto:

```
def processa boletos, fatura
  boletos.each do |boleto|
    pagamento =
      Pagamento.new boleto.valor, MeioDePagamento::BOLETO
    fatura.adiciona_pagamento pagamento
  end
end
```

Veja que agora ele não conhece detalhes da classe `Fatura`. Isso é responsabilidade da própria fatura. Não há mais intimidade inapropriada. O desenvolvedor deve agora mover os testes que garantem a marcação de uma fatura como paga para dentro da classe `FaturaTest`, já que isso não é mais um comportamento do processador de boletos.

9.5 O QUE OLHAR NO TESTE EM RELAÇÃO AO ENCAPSULAMENTO?

Testes que lidam demais com outros objetos ao invés de lidar com o objeto sob teste podem estar avisando o desenvolvedor em relação a problemas de encapsulamento. A própria não utilização da Lei de Demeter, tanto nos testes quanto no código de produção, também pode avisar sobre os mesmos problemas.

Isso é comum em bateria de testes de classes anêmicas [4]. Um modelo anêmico é aquele onde classes contêm apenas atributos ou apenas métodos. Classes que contêm atributos apenas representam as entidades, enquanto outras classes, que contêm apenas métodos, realizam ações sobre eles. Esse tipo

de projeto (que faz com que o código se pareça mais com um código procedural do que com um código orientado a objetos) deve ser evitado ao máximo.

9.6 CONCLUSÃO

Neste capítulo, discutimos o ponto que faltava para encerrarmos a discussão sobre feedback dos testes em relação a pontos importantes em projetos orientados a objetos: coesão, acoplamento e encapsulamento.

Muito provavelmente você, desenvolvedor, encontrará outros padrões de feedback que o teste pode dar. Siga seu instinto, experiência e conhecimento. O importante é ter código de qualidade ao final.

CAPÍTULO 10

Testes de integração e TDD

Até o momento, escrevemos apenas testes de unidade. Será que faz sentido escrever testes ou até mesmo praticar TDD para outros níveis de teste?

Neste capítulo, discutiremos sobre o mau uso de *mocks*, testes para acesso a dados, testes de integração, e quando não os fazer.

10.1 TESTES DE UNIDADE, INTEGRAÇÃO E SISTEMA

Podemos escrever um teste de diferentes maneiras, de acordo com o que esperamos obter dele. Todos os testes que escrevemos até agora são conhecidos por **testes de unidade**.

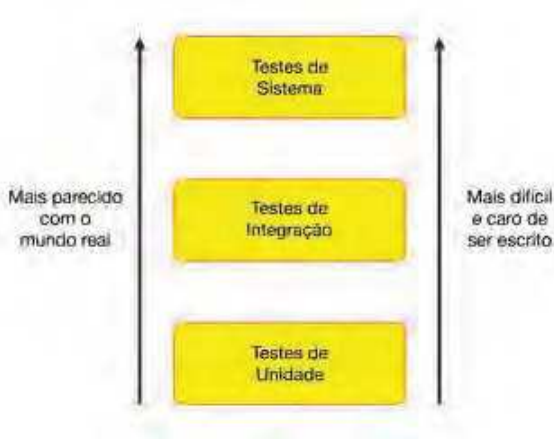
Um teste de unidade é aquele que garante que uma classe funciona, de maneira isolada ao resto do sistema. Ou seja, testamos o comportamento dela sem se preocupar com o comportamento das outras classes. Uma vantagem dos testes de unidade é que eles são fáceis de serem escritos e rodam

muito rápido. A desvantagem deles é que eles não simulam bem a aplicação no mundo real. No mundo real, temos as mais diversas classes trabalhando juntas para produzir o comportamento maior esperado.

Se quisermos um teste que se pareça com o mundo real, ou seja, que realmente teste a aplicação do ponto de vista do usuário, é necessário escrever o que chamamos de **teste de sistema**. Um teste de sistema é aquele que é idêntico ao executado pelo usuário da aplicação. Se sua aplicação é uma aplicação web, esse teste deve subir o browser, clicar em links, submeter formulários etc. A vantagem desse tipo de teste é que ele consegue encontrar problemas que só ocorrem no mundo real, como problemas de integração entre a aplicação e banco de dados. O problema é que eles geralmente são mais difíceis de serem escritos e levam muito mais tempo para serem executados.

No entanto, muitas vezes não queremos testar só uma classe, mas também não o sistema todo; queremos testar a integração entre uma classe e um sistema externo. Esse tipo de teste, que garante a integração entre 2 pontos da aplicação, é conhecido por **teste de integração**.

O desenvolvedor deve fazer uso dos diferentes níveis de teste para garantir qualidade do seu sistema. Mas deve sempre ter em mente que, quanto mais parecido com o mundo real, mais difícil, frágil e caro o teste será.



10.2 QUANDO NÃO USAR MOCKS?

O primeiro passo para que um teste deixe de ser exclusivamente de unidade e passe a ser de integração é não usar *mocks* e passar dependências concretas para a classe sob teste.

Muitos desenvolvedores, inclusive, defendem que um bom teste nunca faz uso de *mocks*. O argumento deles é que *mocks* acabam por “esconder” possíveis problemas que só seriam pegos na integração. O ponto não é descobrir se devemos ou não usar *mocks*, mas sim **quando ou não** os usar.

Veja, por exemplo, um dos testes implementados para a calculadora de salário:

```
def teste_deve_calcular_salario_para_desenvolvedores_
  com_salario_abaixo_do_limite
    calculadora = CalculadoraDeSalario.new
    desenvolvedor = Funcionario.new('Mauricio', 1500.0,
      Cargo::DESENVOLVEDOR)

    salario = calculadora.calcula_salario desenvolvedor

    assert_equal 1500 * 0.9, salario
  end
```

Esse teste garante o comportamento da classe `CalculadoraDeSalario`. Idealmente, gostaríamos de testá-la independente do comportamento das outras classes. Mas veja que no teste instanciamos um `Funcionario`. Usamos a classe concreta e não fizemos uso de *mock*. Por quê?

Geralmente classes que representam entidades, serviços, utilitários, ou qualquer outra coisa que não encosta em infraestrutura, não são *mockadas*. Elas são classes Ruby puras e simples, e *mocká-las* só irá dar mais trabalho ao desenvolvedor.

Ao tomar essa decisão, diminuímos a qualidade do retorno desse teste, afinal ele pode falhar não por culpa da calculadora de salário, mas sim por culpa do funcionário. Por outro lado, esse teste também pode revelar problemas de integração entre o funcionário e a calculadora de salários. Apesar de não ser um teste perfeito, é uma troca justa entre produtividade e feedback.

Opte por *mockar* classes que lidam com infraestrutura e que tornariam seu teste muito complicado. Por exemplo, lembre nosso gerador de nota fiscal. A nota era persistida em um banco de dados e depois enviada para o SAP. Preparar tanto o banco quanto o SAP para receber o teste não é fácil. Portanto, simular a interação de ambas as classes é uma boa ideia.

Use *mocks* também quando não importa a implementação que for utilizada do ponto de vista do teste. Em nossa versão final do gerador de nota fiscal, definimos (implicitamente) uma interface de ações que implementam o método `executa`. Nesses casos, existem duas opções. Uma de criar um *mock* que responde a esse método e a outra de criar uma implementação concreta “simples”, somente para o teste. Para escolher a melhor solução siga os princípios que já vimos para evitar duplicação e simplificar o entendimento. Algumas vezes será mais fácil usar o *mock*, outras será mais fácil implementar uma classe simples.

Um ponto negativo do uso de *mock objects* é o alto acoplamento criado entre o código de teste e o código de produção. Já discutimos o conceito de encapsulamento no capítulo anterior: uma classe deve esconder a maneira na qual ela implementa determinada regra de negócio. Quando um teste faz uso de um *mock*, esse teste passa a ter uma “intimidade inapropriada” com a implementação; ele passa a saber quais métodos serão invocados, e como a classe deve reagir de acordo com o resultado.

Veja o código de produção e teste a seguir, responsável por calcular o imposto de um pedido, de acordo com uma tabela de preços:

```
def
teste_deve_calcular_imposto_para_pedidos_superiores_a_2000_reais
    tabela = mock

    # ensinando o mock a devolver 0.1 caso o método
    # pega_para_valor seja invocado com o valor 2500.0
    tabela.expects(:pega_para_valor).with(2500.0).returns(0.1)

    pedido = Pedido.new 2500.0
    calculadora = CalculadoraDeImposto.new tabela

    valor = calculadora.calcula_imposto pedido
```

```
    assert_equal 2500 * 0.1, valor
end

def calcula_imposto pedido
  taxa = @tabela.pega_para_valor pedido.valor
  pedido.valor * taxa
end
```

Veja que esse teste sabe exatamente qual método será invocado no código de produção. Isso quer dizer que qualquer mudança na implementação do método pode fazer o teste quebrar. Ou seja, quanto maior o uso de *mocks*, mais delicado e frágil seu teste fica.

Muitos desenvolvedores, quando são apresentados à ideia de *mock objects*, passam a *mockar* todos os objetos nos seus testes. Muitos testes escritos com esse pensamento passam a ser inúteis, pois acabam “testando o *mock*”. No capítulo seguinte, veremos um exemplo disso. Mas um primeiro ponto de alerta é: se sua bateria de testes só faz asserções em objetos dublês, talvez essa bateria não lhe esteja dando o feedback necessário.

Portanto, faça uso de *mock objects* quando utilizar a instância concreta da classe for complexo ou trabalhoso. Quando usar a classe concreta não for diminuir o feedback dos seus testes e nem dificultar a escrita dele, então use a classe concreta.

10.3 TESTES EM SUBCLASSES DE `ActiveRecord::Base`

O padrão `ActiveRecord` oferece uma forma fácil de persistir e recuperar objetos de um banco de dados sem precisar se preocupar com o banco que está por trás. Para salvar um objeto no banco, basta chamar o método `save` no objeto que deseja salvar após o construir. Para adicionar ou remover atributos ao objeto, basta fazer a mudança no banco de dados que o `ActiveRecord` reflete a mudança nos métodos e atributos disponíveis no objeto.

Por fim, é muito fácil buscar objetos. Até Rails 3, bastava chamar um dos métodos dinâmico do tipo `find_by` como `find_by_name`,

`find_all_by_price` e similares. Por exemplo, o seguinte código compatível com Rails 2 salva alguns objetos e os recupera:

```
voo = Voo.new(:partida => 'ORD', :chegada => 'ATL',
: data => '2013-12-08')

voo.save

Voo.create(:partida => 'ATL', :chegada => 'SFO',
: data =>
)

Voo.find_all_by_partida_and_chegada('ATL', 'SFO')
```

Para poder testar esse código com um teste de unidade, precisamos *mockar* `save`, `create` e `find_all_by_partida_and_chegada`. No entanto, com Rails 3, o modelo para buscas mudou drasticamente para permitir composição de predicados e tirar benefício da quantidade de possibilidades que isso cria.

Os métodos `find_by` foram substituídos por usos de predicados `where`. A busca `Voo.find_all_by_partida_and_chegada('ATL', 'SFO')` virou:

```
Voo.where(:partida => 'ATL', :chegada => 'SFO').all
```

A consequência da mudança foi que ficou mais difícil escrever os *mocks* para os testes. Agora, o teste fica assim:

```
mock_do_where = mock.expects(:all).returns([])
Voo.expects(:where).returns(mock_do_where)
```

Lembrem-se, nossos testes nos indicam problemas de design. Nesse caso, o teste sabe muito sobre como o modelo `Voo` integra com o *framework* de persistência. A sugestão, nesse caso, é extrair essa busca em um escopo:

```
class Voo
  scope :com_partida_e_chegada, Proc { |partida, chegada|
    where(:partida => partida, :chegada => chegada).all
  }
end
```

Dessa forma, o *mock* volta a ficar mais simples:

```
Voo.expects(:com_partida_e_chegada).returns([])
```

Independente de ter sido escrito com TDD, o teste nos indicou um problema de design. Como os objetos que herdam de `ActiveRecord::Base` costumam ter muitas responsabilidades, os testes acabam ficando grandes e pouco compreensíveis. Essa costuma ser uma boa indicação de que é necessário separar algumas responsabilidades.

A MUDANÇA DO RAILS 4.0

Até a última versão de Rails 3, na maioria dos projetos, todas as classes na pasta 'app/models' eram subclasses de `ActiveRecord::Base`. No entanto, essa herança encoraja o crescimento dos modelos para armazenamento de dados. Conforme o modelo de dados cresce, o modelo de negócios também precisa evoluir. A consequência é que esses objetos acabavam crescendo por vários motivos e tinham mais de uma responsabilidade.

Para desincentivar essa modelagem, Rails 4 separou do `ActiveRecord::Base` a parte para validações e inicializações em `ActiveModel`. Dessa forma, a responsabilidade dos objetos `ActiveRecord` é puramente persistência de dados. Com isso, fica mais claro o que vale a pena ser desenvolvido com TDD por ser lógica de negócios (como `ActiveModel`) e o que não necessariamente precisa de TDD para o desenvolvimento.

Um dos pontos mais controversos sobre testes de objetos que herdam de `ActiveRecord::Base` é escrever testes que usam *mocks* ou deixá-los ir ao banco de dados. Como vimos nessa seção, escrever os *mocks* pode ser mais difícil do que se espera para esses objetos. Por outro lado, eles nos garantem velocidade e simplificam o que precisa ser feito para preparar o ambiente para o teste.

Um dos motivos pelo qual essa discussão é tão comum é que muitos objetos `ActiveRecord` misturam responsabilidades de persistência com lógica

de negócios. Toda a parte de lógica de negócios, conforme discutimos ao longo do livro, faz muito sentido de ser testada com *mocks*. Já a parte que é responsável pela persistência no banco de dados, não faz muito sentido de testar com *mocks* já que a integração é tudo que importa.

No fim das contas, o ideal é escrever testes que dêem o melhor *feedback* possível com ou sem `ActiveRecord`. Na prática, costuma ser uma mistura de testes com *mocks* para as partes que conseguimos ignorar o banco de dados e testes sem *mocks* para tudo que lidar com persistência.

Testes com efeitos colaterais

Outro problema comum para objetos `ActiveRecord` é limpar o banco de dados. Imagine, por exemplo, um teste que conte a quantidade de produtos cadastrados. Se já tivermos algum produto cadastrado antes da execução do teste, provavelmente o teste falhará, pois ele não contava com esse dado a mais. Portanto, cada teste precisa ter o banco de dados limpo para que dados antigos não influenciem no resultado.

É comum fazermos uso de métodos de finalização para isso. Veja um exemplo:

```
def teardown
  @produto.destroy
end
```

Novamente, testes de integração podem se tornar complicados. Por exemplo, caso um teste exija dados pré-salvos no banco de dados, é responsabilidade do teste de montar todo o cenário e persisti-lo.

10.4 DEVO USAR TDD EM TESTES DE INTEGRAÇÃO?

Em classes que lidam com infraestrutura, nem sempre é fácil usar TDD para evoluir o design. TDD faz muito sentido quando queremos testar algoritmos ou projetos de classe complexos. É uma boa prática manter as classes que lidam com infraestrutura simples com apenas código que integra com outros sistema. Por cima dessas classes é que se monta uma camada onde pode ficar a lógica complexa de negócio que, essa sim, pode ser desenvolvida com TDD.

Como também discutido no capítulo sobre acoplamento, ao praticar TDD, o desenvolvedor acaba por criar interfaces que representam a interação com sistemas externos. Essas interfaces tendem a ser bem claras e específicas. Ao terminar a implementação da classe principal, o desenvolvedor parte para as classes ao redor. Nessa hora, como não há muitas decisões a ser tomada, não há grandes problemas em implementar a classe e só depois escrever o teste.

É óbvio que, caso você esteja criando alguma integração mais complexa, TDD muito provavelmente o ajudará a entender melhor o problema e o guiará (através do já conhecido *feedback*) a uma solução melhor. Analise caso a caso.

10.5 TESTES EM APLICAÇÕES WEB

Novamente, a ideia de se escrever testes antes pode ser aplicada a qualquer contexto. Se o programador sentir que isso irá ajudar, então deve fazê-lo. Em sistemas web, geralmente a grande dificuldade de se escrever um teste é justamente separar as “camadas de integração” da camada de domínio.

Independente do *framework* escolhido para ajudar no desenvolvimento (Rails, Sinatra, Merb etc.), o desenvolvedor é obrigado a escrever uma camada que “conecta” o mundo web, cheio de requisições, respostas, HTTP e HTML, com o mundo de domínio, cheio de classes de negócio ricas. Chamamos essa camada geralmente de “controlador” (ou *Controller*). Controllers devem ser adaptadores. Não devem possuir regras de negócio, apenas regras de fluxo e exibição.

Veja a seguir um exemplo de um método de um controlador de uma aplicação escrita com Rails, responsável por decidir qual o próximo passo do usuário em um sistema de ensino online. Se o aluno já terminou o curso, o controlador redireciona para o certificado, se ele não terminou, redireciona para a próxima seção que ele deve fazer:

```
class CourseController
  def continue_course
    course = Course.where(:code => params[:course][:code]).first
    enrollment = Enrollment.where(:user => logged_user,
      :course => course).first
```

```
if enrollment.achieved_certification?
  redirect_to user_course_certificate_path(logged_user,
                                           course)

elsif course.pre_sale?
  redirect_to dashboard_path
else
  section = enrollment.continuing

  redirect_to course_section_page_path(section.course.code,
                                       section.number, 0)

end
end
end
```

Veja que não há regras de negócio nesse código. As regras de negócio estão isoladas em classes de domínio (e, como mostrado ao longo do livro, são facilmente testadas) e o acesso à infraestrutura, como banco de dados, também está isolado (e pode ser testado também, como discutido neste capítulo). Ao isolar todo código de negócio da camada de controle, o desenvolvedor consegue testar a “parte que importa”.

Agora, um ponto interessante a ser discutido é sobre a necessidade de se testar um controlador ou não. Muitos desenvolvedores gostam de testar controladores a fim de garantir que eles sempre tomarão a decisão certa sobre a próxima página a ser exibida.

Mas veja o código anterior. Para testá-lo, são necessários muitos *mocks*: `Course.where` e `Enrollment.where` são métodos fornecidos pelo `ActiveRecord` que buscam, respectivamente, cursos e inscrições em cursos, além dos objetos que esses *mocks* deverão devolver. Um simples teste fará uso de diversos *mocks*. Já discutimos isso anteriormente: isso mostra o alto acoplamento dessa classe.

Controladores são um bom exemplo de classes cujo acoplamento alto é aceitável. De novo, eles são adaptadores, e portanto se conectam a dois mundos diferentes, cada um com sua classe. Já discutimos também sobre como testes que usam *mocks* são altamente acoplados com a implementação. Eles sabem como a implementação funciona, e portanto qualquer mudança na implementação impacta no teste. São testes frágeis.

Geralmente, eu opto por não escrever testes de unidade para controladores. Acho esses testes muito frágeis, mudam o tempo inteiro e não dão o feedback esperado. Erros em controladores geralmente acontecem não pelo seu código em si, mas por problemas de integração com a camada web. Nomes de elementos no HTML que são submetidos pelo formulário e capturados pelo controlador são causadores de muitos bugs.

E sim, essa integração deve ser testada. Mas, na minha opinião, a melhor solução é escrever um teste de sistema. Ou seja, deve-se subir o *browser*, navegar na aplicação web, submeter formulários e validar a saída. Para testes como esses, ferramentas como Selenium (que sabem como abrir o *browser* e manipular os elementos) são de grande valia.

Fazer TDD em testes de sistema tem sido estudado pela comunidade acadêmica há algum tempo. Eles batizaram a técnica de ATDD (*Acceptance Test-Driven Development*). A ideia é escrever um teste de sistema que falha, em seguida fazer o ciclo de TDD para criar todas as classes necessárias para implementar a funcionalidade e, por consequência, fazer o teste de sistema passar. Na prática, não vejo como fazer isso funcionar de maneira produtiva. Um teste de sistema é um teste caro e frágil de ser escrito. Gosto deles, mas escritos após a implementação.

10.6 CONCLUSÃO

Não há uma receita mágica para escrever testes de integração. Mas lembre-se que, caso você tenha uma classe cuja responsabilidade é justamente se integrar com outro sistema, você deve fazer um teste de integração.

Consiga uma réplica do sistema ou alguma maneira de consumi-lo para testes, seja ele um banco de dados, um serviço web, ou qualquer outra coisa.

O grande desafio, no fim, é conseguir montar o cenário e validar a saída nesse serviço externo. Talvez você escreva um “teste feio”, com muitas linhas de código. Mas não se preocupe, seu sistema estará bem testado e você dormirá em paz.

CAPÍTULO 11

Testando código dinâmico

Ruby difere das outras linguagens para as quais este livro foi escrito originalmente (Java e C#) pela sua dinamicidade. Vários aspectos da linguagem favorecem isso. O primeiro é a tipagem dita dinâmica porque permite que a mesma variável seja usada para armazenar objetos de tipos diferentes. As consequências de tal tipagem são extensas na filosofia da linguagem, mas a mais importante é a ideia do *Duck Typing*. A primeira sessão deste capítulo mostrará algumas diferenças que essa importante técnica implica na prática de TDD.

Outro aspecto que impressiona em Ruby são as ferramentas para reflexão disponíveis na linguagem. A troca de mensagens entre objetos em vez de chamada de métodos permite ser mais flexível com relação ao comportamento do objeto. Ao invés de chamar um método diretamente, objetos enviam mensagens para outros objetos. Esse envio se dá na forma de uma mensagem básica chamada `send`. O poder dessa troca de mensagens fica claro quando o pró-

prio programador pode fazer uso disso para determinar de forma dinâmica qual mensagem enviar. A segunda sessão deste capítulo mostrará alguns padrões e antipadrões do uso de `send` e da escrita de seus testes.

Por fim, juntamente com a mensagem `send`, Ruby usa a mensagem `method_missing` para lidar com situações em que não há resposta para uma mensagem enviada. Isso permite que o programador determine um comportamento no objeto caso a mensagem não seja tratada pelo objeto. A última sessão desse capítulo apresentará alguns exemplos de uso de `method_missing` e como testá-los.

11.1 É POSSÍVEL UTILIZAR TDD COM *DUCK TYPING*?

A ideia toda de *Duck Typing* é que um tipo em Ruby não é definido por uma classe ou interface mas pelo uso que se faz dos objetos. Isso significa que, ao contrário de Java ou C#, o acoplamento entre um objeto e suas dependências é bem mais fraco, já que as dependências podem ser trocadas de implementação simplesmente passando um objeto de outro tipo na inicialização ou em algum *setter*.

Uma consequência desse fato é que, na escrita do teste, não temos a opção de criar uma interface para definir o comportamento das dependências do objeto que estamos testando. A contrapartida é que o teste é o artefato que define os comportamentos dessas dependências. Seja com um *mock* ou com um objeto *fake* (falso), o teste deixa claro que não importa o tipo do objeto passado desde que ele responda a determinadas mensagens.

```
require 'test/unit'
require File.expand_path('../lib/carrinho_de_compras.rb',
  File.dirname(__FILE__))

class TesteCarrinhoDeCompras < Test::Unit::TestCase
  class ItemFalso
    attr_reader :quantidade
    def initialize(quantidade)
      @quantidade = quantidade
    end
    def valor
```

```
1
end
end
def teste_calcula_total_considerando_quantidades
  carrinho = CarrinhoDeCompras.new

  carrinho << ItemFalso.new(1)
  carrinho << ItemFalso.new(5)

  assert_equal 6, carrinho.valor_total
end
end
```

Note que o teste não requer nada além do próprio carrinho de compras. Isso permite seguir uma técnica de TDD sugerida por Kent Beck que consiste em só alterar o código do teste para os passos de verde e vermelho do ciclo de TDD. A ideia é evitar adicionar qualquer coisa ao código de produção, a não ser no passo de refatoração. Dessa forma, o perigo de uma mudança para o teste atual causar um problema em outro teste é muito pequeno. A técnica também facilita enxergar potenciais melhorias de modelo que tornam os testes mais simples de serem escritos.

Outro ponto que vale observar sobre este teste é que o `ItemFalso` é uma implementação manual de um *fake* ou *stub*. É um objeto que responde à mesma interface do objeto de produção mas com uma implementação bem mais simples. Isso nos permite utilizar o conceito de *stub* mesmo sem precisar de uma biblioteca. Também é possível fazer o mesmo com *mock* usando a mesma ideia. No entanto fica como exercício para vocês pensarem em como o fariam.

Em suma, *Duck Typing* faz com que o estágio de extrair interfaces que permitem criar *mocks* ou *stubs* seja implícito, tornando os passos do TDD mais simples e curtos. No entanto, é mais difícil para um leitor descobrir quais são todos os métodos que precisam ser implementados para substituir uma dependência.

11.2 A MÁGICA DO *SEND* E SEUS TESTES

Ruby como linguagem funciona com base na troca de mensagens, como dito anteriormente. O método `send` é responsável por enviar mensagens. Por exemplo:

```
['send', 'é', 'muito', 'poderoso'].join(' ')
> send é muito poderoso
```

É equivalente a:

```
['send', 'é', 'muito', 'poderoso'].send(:join, ' ')
> send é muito poderoso
```

Praticamente qualquer método de qualquer objeto pode ser invocado usando `send`. Por si só, isso não traz nenhuma vantagem e torna o código mais difícil de se ler. No entanto, basta perceber que o nome do método (`join` no exemplo) pode ser uma variável para enxergar o poder que isso pode ter. Vejamos outro exemplo mais complexo. A ideia é um sistema que recebe cliques em determinados pontos e reage diferentemente de acordo com o ponto clicado. A infraestrutura simplificada para o sistema é a seguinte:

```
class Ponto
  attr_reader :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
  def to_s
    "(#{@x}, #{@y})"
  end
end
class Vetor
  def initialize(x, y)
    @origem = Ponto.new(0,0)
    @destino = Ponto.new(x, y)
  end
  def to_s
    "#{@origem} -> #{@destino}"
  end
end
```



```
end
end
```

Essa é a nossa base para criar um desenho que será alterado. Simplificamos as alterações para imprimir uma mensagem de texto apenas, já que a implementação do desenho não importa para o que queremos mostrar:

```
class Desenho
  def adiciona_linha
    puts 'adiciona_linha'
  end
  def remove_linha
    puts 'remove_linha'
  end
  def move_linha(vetor)
    puts "move linha de acordo com vetor #{vetor}"
  end
end
```

Agora vejamos uma interface bem simples que recebe itens visuais (no nosso caso botões) e apenas os guarda sequencialmente em uma matriz:

```
class Interface
  def initialize
    @itens_visuais = []
  end
  def adiciona(linha, *itens_visuais)
    @itens_visuais << [] while linha >= @itens_visuais.size
    @itens_visuais[linha] += itens_visuais
  end
  def clique(posicao)
    item = identifica_item_sob(posicao)
    item.executa_em(desenho_atual)
  end
  def identifica_item_sob(posicao)
    @itens_visuais[posicao.y][posicao.x]
  end
  def desenho_atual
    Desenho.new
  end
end
```

```
end
end
```

Aqui vem nosso super botão mágico que usa o `send`:

```
class Botao
  def initialize(acao, *argumentos)
    @acao = acao
    @argumentos = argumentos
  end
  def executa_em(alvo)
    alvo.send(@acao, *@argumentos)
  end
end
```

Agora que temos tudo isso, podemos montar uma interface e receber cliques nela. Vamos simular isso chamando métodos simples que já definimos:

```
botoes = []
botoes << Botao.new(:adiciona_linha)
botoes << Botao.new(:remove_linha)
botoes << Botao.new(:move_linha, Vetor.new(-1, -1))

interface = Interface.new
interface.adiciona(0, *botoes)

interface.clique(Ponto.new(0, 0))
> adiciona_linha
interface.clique(Ponto.new(2, 0))
> move linha de acordo com vetor (0, 0) -> (-1, -1)
interface.clique(Ponto.new(1, 0))
> remove_linha
```

Note que isso permite escrever uma única classe `Botao` que define o comportamento de um botão independente da ação que deve ser chamada. Para atingir o mesmo resultado em Java ou C#, precisaríamos criar uma interface que unificasse o nome das ações chamadas ou precisaríamos passar um bloco de código para execução posterior. Graças ao `send`, o parâmetro do botão fica sendo simplesmente o nome do método e os parâmetros para o mesmo.

Agora que entendemos um pouco melhor como `send` funciona, vem a pergunta sobre como podemos testar o uso de `send`. Vejamos como ficaríamos os testes da classe `Botao`:

```
require 'test/unit'
require 'mocha/setup'
require File.expand_path('../lib/botao.rb',
  File.dirname(__FILE__))

class TesteBotao < Test::Unit::TestCase
  def teste_chama_metodo_no_alvo_ao_executar
    botao = Botao.new(:teste)
    alvo = mock
    alvo.expects(:teste).with()

    botao.executa_em(alvo)
  end
  def teste_chama_metodo_no_alvo_com_parametros_ao_executar
    botao = Botao.new(:teste, :parametro1, [:parametro2])
    alvo = mock
    alvo.expects(:teste).with(:parametro1, [:parametro2])

    botao.executa_em(alvo)
  end
end
```

Note que esse teste não depende de nada fora a classe `Botao`. Note também que ambos os testes têm suas verificações embutidas nos *mocks* que usamos. Isso porque a classe `Botao` por si própria não tem nenhum trabalho computacional. A única lógica que ela contém é a associação entre os parâmetros de inicialização e a chamada do método no alvo. Portanto, é a única coisa testada.

Note também que nada faz com que não possamos escrever esses testes antes de escrever a classe `Botao`. Na verdade, eu os escrevi dessa forma para ter certeza de apresentar a implementação correta. A parte que não fica clara é qual foi o passo no qual TDD ajudou a identificar um bom uso de `send`. Vejamos uma versão dos testes anterior ao uso do `send` na implementação:

```
require 'test/unit'
require 'mocha/setup'
require File.expand_path('../lib/botao.rb',
  File.dirname(__FILE__))

class TesteBotao < Test::Unit::TestCase
  def teste_chama_metodo_no_alvo
    botao = Botao.new(lambda{|alvo| alvo.teste})
    alvo = mock
    alvo.expects(:teste).with()

    botao.executa_em(alvo)
  end
  def teste_chama_metodo_no_alvo_com_parametros_ao_executar
    botao = Botao.new(lambda{
      |alvo| alvo.teste(:parametro1, [:parametro2])
    })
    alvo = mock
    alvo.expects(:teste).with(:parametro1, [:parametro2])

    botao.executa_em(alvo)
  end
end
```

Existem duas dicas que sugerem possíveis refatorações nesse teste. A primeira é um pouco mais fácil de se enxergar mas não é muito preocupante: todo botão tem um conjunto de caracteres que precisam ser repetidos. É o caso de `lambda{|alvo| alvo..`. Como esse tipo de expressão é muito comum na linguagem, costumamos não nos preocupar muito com ela. No entanto, a dica mais preocupante é a de que nosso mock na verdade só testa o código que está no próprio teste dentro do bloco do lambda. Poderíamos mudar o teste para dizer que o lambda precisa ser chamado com o alvo como parâmetro, mas isso torna a intenção do teste difícil de se enxergar. O uso de *send* permite remover a duplicação de código e passa a verificar realmente o comportamento do botão.

OS PERIGOS DO USO DE `SEND`

Como toda forma de metaprogramação, `send` parece muito atraente para quem descobre como usá-lo. No entanto, é importante avisar de alguns problemas que podem surgir ao usar `send` sem a devida atenção.

Podemos começar com um dos “abusos” de `send` mais comuns: usar a entrada do usuário como parâmetro para o `send`. A tentação vem de oferecer várias opções para o usuário e agir diferentemente de acordo com a ação escolhida. Por exemplo, um formulário que envia a forma de pagamento e os detalhes da mesma. O *controller* que irá atender a requisição ficaria assim:

```
class PagamentoController < ActionController
  def pagar
    send(params[:forma_de_pagamento],
         params[:detalhes_de_pagamento])
  end
  private
  def cartao(detalhes); ...; end
  def deposito(detalhes); ...; end
end
```

A solução é tentadora pois separa as implementações de forma simples e limpa. O problema é, obviamente, que nada garante que o usuário não passe valores inesperados como, por exemplo, `forma_de_pagamento` sendo `eval` e `detalhes_de_pagamento` como `Http::Post("url_maliciosa", params)` ou coisas muito piores. Note que nada em TDD fará você se defender disso se não pensar nesse caso anteriormente. Uma “solução” comum nesse caso é não usar a entrada do usuário pura:

```
send("valida_detalhes_para_#{params[:forma_de_pagamento]}",
     params[:detalhes_de_pagamento])
```

Apesar de diminuir bastante os potenciais ataques de usuários, há um custo associado. Além de perder legibilidade, fica mais difícil para alguém novo no código saber quais métodos são válidos e quais já não são mais usados. Os testes ajudam bastante mas ainda é sua responsabilidade garantir que a consistência entre o controlador e o HTML gerado¹⁵³ está correta.

Agora que vimos os poderes do `send`, vejamos outro método “mágico” que é frequentemente usado com `send`: o `method_missing`.

11.3 O USO E TESTE DE `METHOD_MISSING`

O método `method_missing` é chamado em um objeto quando a mensagem enviada anteriormente para ele não foi respondida por nenhum método. `method_missing` recebe como parâmetro o nome da mensagem enviada e os argumentos que a acompanhavam. A implementação padrão do método lança uma exceção `NoMethodError`.

No entanto, como qualquer método público, subclasses de `Object` podem sobrescrever esse comportamento. Veja um exemplo:

```
require 'timeout'
class ComLimiteDeTempo
  def initialize(timeout, target)
    @timeout = timeout
    @target = target
  end
  def method_missing(method, *args)
    Timeout::timeout(@timeout) do
      @target.send(method, *args)
    end
  end
end
```

O interessante do código anterior é que criamos facilmente um objeto que finge ser qualquer objeto mas adiciona exigências de tempo de execução em seus métodos. Note que usamos todos os recursos de metaprogramação apresentados até agora para gerar essas 10 linhas de código. Usamos *duck typing* para poder passar esse objeto `ComLimiteDeTempo` no lugar de qualquer outro. Usamos `method_missing` para capturar chamadas de métodos que deveriam ser destinadas ao objeto original e usamos `send` para reencaminhar essas chamadas.

Apesar de todo o dinamismo e flexibilidade, não há nada que faça com que o método `method_missing` seja diferente de qualquer outro método na hora de escrever o teste para ele. Na verdade, o teste para o objeto anterior

não precisa, apesar de poder, saber nada sobre `method_missing`. Veja uma versão dos testes que não se preocupa com os detalhes de implementação:

```
require 'test/unit'
require 'mocha/setup'
require File.expand_path('../lib/com_limite_de_tempo.rb',
  File.dirname(__FILE__))

class ObjetoFalso
  def initialize(velocidade_de_execucao)
    @velocidade = velocidade_de_execucao
  end
  def metodo
    sleep(@velocidade)
    return true
  end
end

class TesteComLimiteDeTempo < Test::Unit::TestCase
  def
    teste_chama_metodo_e_termina_com_sucesso_para_execucao_rapida
      execucao_limitada =
        ComLimiteDeTempo.new(1, ObjetoFalso.new(0))

      assert_equal true, execucao_limitada.metodo
    end
    def teste_chama_metodo_e_lanca_erro_para_execucao_lenta
      execucao_limitada =
        ComLimiteDeTempo.new(1, ObjetoFalso.new(5))

      assert_raises(TimeoutError) do
        execucao_limitada.metodo
      end
    end
  end
end
```

No entanto, como todo método, conforme ele vai ficando complicado, precisamos de mais testes para definir e verificar seu comportamento. Tipicamente, toda implementação de `method_missing` deveria ser o mais

simples possível. Isto é, se sentir vontade de escrever um teste para o `method_missing`, ele provavelmente não deveria fazer tudo que faz. Idealmente, a única lógica de um `method_missing` é delegar para o método correto que contém a lógica de negócios.

Dessa forma, seus testes ficam como os anteriores: o teste descreve o comportamento de acordo com as regras definidas mas não tem conhecimento sobre o `method_missing` em si. Lembre-se sempre que metaprogramação é um grande poder, e como todo grande poder, ela traz grandes responsabilidades. Se for utilizar as técnicas descritas neste capítulo, tenha certeza de que seus testes capturam bem sua intenção e que os limites de uso estão bem delimitados.

CAPÍTULO 12

Quando não usar TDD?

Ao longo deste livro, discutimos as diversas vantagens da escrita de testes de unidade e TDD, tanto na qualidade externa quanto na qualidade interna do sistema.

Mas será que devemos fazer uso da prática 100% do tempo? Neste capítulo discutimos quando usar e, mais importante, quando não usar TDD.

12.1 QUANDO NÃO PRATICAR TDD?

Repetindo uma frase que apareceu diversas vezes ao longo do livro. Em engenharia de software, não se deve usar as palavras “nunca” e “sempre”. Nenhuma prática deve ser usada 100% do tempo ou descartada de uma vez.

Em momentos em que o desenvolvedor não sabe bem como resolver o problema, dada sua complexidade, ou não sabe bem como projetar uma

classe, dada sua necessidade de ser flexível, TDD pode ser de grande valia devido ao seu constante *feedback* sobre a qualidade do código.

Mas, em momentos em que o desenvolvedor já sabe como resolver o problema, tanto do ponto de implementação, quanto do ponto de vista de design, não utilizar TDD pode não ser um problema tão grande assim. Uma outra situação na qual o uso de TDD pode ser desnecessário é durante a escrita de códigos que lidam com infraestrutura ou com camada de visualização.

Novamente, a ideia não é usar TDD porque a prática está em alta no mercado, mas sim pelo benefício que ele agrega ao processo. Se não há a necessidade de *feedback* constante naquele momento, não há necessidade de se praticar TDD. O que talvez seja difícil é justamente julgar corretamente quais são esses momentos. A experiência do desenvolvedor em desenvolvimento de software e em TDD é crucial.

Outro ponto importante a ser levantado é que estamos discutindo a não utilização de TDD em alguns momentos, mas não a falta de testes. Caso o desenvolvedor opte por não usar TDD, espera-se que ele, após a implementação, escreva testes para aquela funcionalidade, sejam eles de unidade ou integração. É preciso garantir a qualidade externa de maneira sustentável, e como discutido na introdução, essa maneira é automatizando o teste.

Deixe sua experiência lhe guiar também na escolha de usar ou não TDD. Lembre-se que, no fim das contas, o que precisamos é de *feedback* sobre o que estamos fazendo.

VOCÊ NÃO FAZ TDD 100% DO TEMPO ENTÃO?

Não. Dou preferência sim para TDD, mas nos casos discutidos anteriormente, geralmente opto por não fazer. Mas, mesmo quando não faço, tento ao máximo aumentar a quantidade de *feedback* que recebo dos testes que escrevo após a implementação.

Ou seja, mesmo não praticando TDD, tento sempre escrever uma pequena quantidade de código e em seguida um teste. E repito.

12.2 100% DE COBERTURA DE CÓDIGO?

Cobertura de código é a métrica que nos diz a quantidade de código que está sendo testada e a quantidade de código em que nenhum teste o exercita. Uma cobertura de código de 100% indica que todo código de produção tem ao menos um teste passando por ele.

Muitas pessoas discutem a necessidade de ter 100% de cobertura em um código. Mas na verdade, não há problemas em códigos que não tenham 100% de cobertura de testes de unidade. Uma equipe deve ter como meta buscar sempre a maior cobertura possível; mas essa é uma meta que provavelmente não será alcançada. Alguns trechos de código simplesmente não valem a pena se testar de maneira isolada, ou são simples demais para isso.

Um outro exemplo de código que não vale a pena ser testado são *getters* e *setters*. Na grande maioria das vezes, usaremos um método auxiliar como `attr_reader`, `attr_writer` ou `attr_accessor` para criar esses métodos.

Um desenvolvedor deve cobrir seu código de testes, mas pode usar testes de diferentes níveis para isso (testes de unidade, de integração, de sistema). Escrever testes de unidade inúteis, apenas para chegar nos 100% de cobertura, é desperdício.

MAS COMO FAZER ESSA MÉTRICA SER ÚTIL?

Configure a métrica para ignorar essas chamadas a métodos auxiliares de Ruby (e de Rails, caso trabalhe com Rails) e todas as outras classes para as quais você já sabe que não escreverá testes. Desse jeito, a métrica pode passar a ser mais útil.

12.3 DEVO TESTAR CÓDIGOS SIMPLES?

Muitos desenvolvedores possuem essa heurística: “se o código é simples, então não preciso testá-lo”. A regra pode até fazer algum sentido, mas ela é incompleta.

Alguns trechos de código não merecem ser testados mesmo. Alguns construtores simples não precisam de testes próprios. Métodos que repassam a in-

vocação do seu método para um método de uma de suas dependências e, por isso, só possuem uma única linha ou usam um método auxiliar do módulo `Forwardable` talvez também não precisem ser testados.

Mas cuidado para não achar que todo método é simples. Muitos erros caros são gerados por apenas uma linha de código. Lembre-se que, no momento em que o desenvolvedor escreve o código, ele parece simples aos olhos dele. Mas é fato que esse código sofrerá alterações no futuro; e muito provavelmente por um outro desenvolvedor. Como garantir que as alterações feitas não façam com que as regras de negócio antigas parem de funcionar? Testes são necessários.

Portanto, cuidado na hora de julgar se um código merece ou não ser testado. Na dúvida, teste. Lembre-se que todo código é considerado culpado até que se prove inocente.

12.4 ERROS COMUNS DURANTE A PRÁTICA DE TDD

Ao praticar TDD, desenvolvedores às vezes cometem alguns deslizes que podem prejudicar o feedback da prática [15].

Escrever o teste e não o ver falhar pode não ser uma boa ideia. Se o desenvolvedor escreve um teste achando que ele falharia, mas ele passa, algo está errado. Ou ele não entende muito bem o código de produção que existe, ou o teste não está bem implementado.

A ideia de também ver o teste passar rapidamente é justamente para que você “teste o teste”. Teste automatizado é código; e esse código pode conter erros. Ao ver o teste passar, você garantiu que o teste fica vermelho ou verde na hora certa.

HUGO FALA

Uma técnica divertida e muito útil quando se está usando TDD é a de, antes de rodar o teste, narrar o resultado esperado. Algo como:

- Espero que este teste falhe com uma exceção que diz que o método `maior_valor` não existe.
- Agora espero que o teste falhe dizendo que esperava 250 mas devolveu `nil`.
- Agora o teste vai passar.

Apesar de o exercício parecer fútil, falar em voz alta o resultado que esperamos nos ajuda a tomar consciência do nosso erro quando o resultado não bater. Também torna a prática de programação em pares mais divertida e garante que ambas as pessoas no par estejam acompanhando uma a outra. Nesse caso, a pessoa que não escreveu o teste é a que precisa prever o que vai acontecer.

Refatorar também é um passo importante do ciclo de TDD. Como discutido ao longo do livro, TDD faz com que você escreva código simples frequentemente. Mas o simples pode não ser a melhor solução para o problema. Para chegar nela, o desenvolvedor deve refatorar seu código constantemente. Em um de meus estudos, observei que a grande maioria dos desenvolvedores diz esquecer da etapa de refatoração e partir logo para o próximo teste de maneira frequente. Isso pode não ser uma boa ideia.

Não começar pelo cenário mais simples e ir evoluindo aos poucos também pode gerar código complexo desnecessário. Ao começar pelo cenário simples, o desenvolvedor garante que irá escrever a menor quantidade de código para fazer o teste passar. Evoluir aos poucos ajuda o desenvolvedor a aprender mais sobre o problema e ir evoluindo o código de maneira mais sensata.

Os passos de TDD fazem sentido. Mas, novamente, use seu bom senso e experiência na hora de decidir usá-los ou não.

12.5 COMO CONVENCER SEU CHEFE SOBRE TDD?

Geralmente um programador que nunca praticou TDD tem essa dúvida: será que TDD realmente ajuda na qualidade do código? E na redução de defeitos? Ele aumenta ou diminui a produtividade, afinal? Mas como toda e qualquer prática em engenharia de software, é muito difícil avaliar e chegar a uma conclusão exata sobre seus ganhos e benefícios.

Ao longo do livro, citamos diversas vantagens da escrita de testes e TDD. Maior qualidade externa, interna (do ponto de vista de implementação e projeto de classes), produtividade (testes manuais no fim saem mais caros), entre outras.

Nos últimos anos, a comunidade acadêmica vem rodando diversos experimentos para tentar mostrar de maneira empírica que TDD realmente ajuda no processo de desenvolvimento de software. Alguns desses estudos são feitos por professores bastante conhecidos na comunidade, como a prof. Laurie Williams (North Carolina State University) e o prof. David Janzen (California Polytechnic State University).

Algumas dessas pesquisas investigam o fato de TDD reduzir o número de defeitos de um software; já outros investigam o fato de TDD produzir código de melhor qualidade. Alguns até pesquisam por indícios de aumento de produtividade. A seguir listo alguns desses estudos que podem ser usados para convencer seu chefe.

Estudos na indústria

Janzen [22] mostrou que programadores usando TDD na indústria produziram código que passaram em aproximadamente 50% mais testes caixa-preta do que o código produzido por grupos de controle que não usavam TDD. Além do mais, o grupo que usava TDD gastou menos tempo debugando. Janzen também mostrou que a complexidade dos algoritmos era muito menor e a quantidade e cobertura dos testes era maior nos códigos escritos com TDD.

Um estudo feito pelo Maximillien e Williams [14] mostrou uma redução de 40-50% na quantidade de defeitos e um impacto mínimo na produtividade quando programadores usaram TDD.

Outro estudo feito por Lui e Chan [11] comparando dois grupos, um utilizando TDD e o outro escrevendo testes apenas após a implementação, mostrou uma redução significativa no número de defeitos. Além do mais, os defeitos que foram encontrados eram corrigidos mais rapidamente pelo grupo que utilizou TDD. O estudo feito por Damm, Lundberg e Olson [12] também mostra uma redução significativa nos defeitos.

O estudo feito por George e Williams [13] mostrou que, apesar de TDD poder reduzir inicialmente a produtividade dos desenvolvedores mais inexperientes, o código produzido passou entre 18% a 50% mais em testes caixa-preta do que códigos produzidos por grupos que não utilizavam TDD. Esse código também apresentou uma cobertura entre 92% a 98%. Uma análise qualitativa mostrou que 87,5% dos programadores acreditam que TDD facilitou o entendimento dos requisitos e 95,8% acreditam que TDD reduziu o tempo gasto com debug. 78% também acreditam que TDD aumentou a produtividade da equipe. Entretanto, apenas 50% acreditam que TDD ajuda a diminuir o tempo de desenvolvimento. Sobre qualidade, 92% acreditam que TDD ajuda a manter um código de maior qualidade e 79% acreditam que ele promove um design mais simples.

Nagappan [30] mostrou um estudo de caso na Microsoft e na IBM e os resultados indicaram que o número de defeitos de quatro produtos diminuiu entre 40% a 90% em relação a projetos similares que não usaram TDD. Entretanto, o estudo mostrou também TDD aumentou o tempo inicial de desenvolvimento entre 15% a 35%.

Langr [23] mostrou que TDD aumenta a qualidade do código, provê uma facilidade maior de manutenção e ajuda a produzir 33% mais testes comparados a abordagens tradicionais.

Estudos na academia

Um estudo feito por Erdogmus et al [21] com 24 estudantes de graduação mostrou que TDD aumenta a produtividade. Entretanto nenhuma diferença de qualidade no código foi encontrada.

Outro estudo feito por Janzen [9] com três diferentes grupos de alunos (cada um deles usando uma abordagem diferente: TDD, test-last, sem testes), mostrou que o código produzido pelo time que fez TDD usou melhor con-

ceitos de orientação a objetos, e as responsabilidades foram separadas em 13 diferentes classes enquanto que os outros times produziram um código mais procedural. O time de TDD também produziu mais código e entregou mais funcionalidades. Os testes produzidos por esse time teve duas vezes mais asserções que os outros e cobriu 86% mais branches do que o time test-last. Além do mais, as classes testadas tinham valores de acoplamento 104% menor do que as classes não testadas e os métodos eram, na média, 43% menos complexos do que os não-testados.

O estudo de Müller e Hagner [16] mostrou que TDD não resulta em melhor qualidade ou produtividade. Entretanto, os estudantes perceberam um melhor reuso dos códigos produzidos com TDD. Steinberg [33] mostrou que código produzido com TDD é mais coeso e menos acoplado. Os estudantes também reportaram que os defeitos eram mais fáceis de serem corrigidos. O estudo do Edwards [18] com 59 estudantes mostrou que código produzido com TDD tem 45% menos defeitos e faz com que o programador se sinta mais à vontade com ele.

12.6 TDD EM SISTEMAS LEGADOS

A grande dificuldade de se praticar TDD (e testes de unidade) em sistemas legados é que geralmente eles apresentam uma grande quantidade de código procedural. Códigos procedurais costumam conter muitas linhas de código, lidar com diferentes responsabilidades, acessar banco de dados, modificar a interface do usuário, o que os torna difíceis de serem testados.

Nesses casos, a sugestão é para que o desenvolvedor, antes de começar a refatorar o legado, escreva testes automatizados da maneira que conseguir. Se conseguir escrever um teste de unidade, excelente. Mas caso não consiga, então ele deve começar a subir o nível até conseguir escrever um teste. Em aplicações legadas web, por exemplo, o desenvolvedor pode começar com testes de sistema, para garantir o comportamento do sistema.

Com esses testes escritos, o desenvolvedor conseguirá refatorar o código legado problemático que está por baixo, e ao mesmo tempo garantir a qualidade do que está fazendo. Nessa refatoração, o desenvolvedor deve criar classes menores e mais coesas e já escrever testes de unidade para elas.

SISTEMAS LEGADOS EM RUBY

Por ser uma linguagem bastante dinâmica, Ruby oferece algumas facilidades na hora de trabalhar em um sistema legado. As técnicas de metaprogramação apresentadas anteriormente e a habilidade de substituir um objeto por outro na ideia de *Duck Typing* faz com que seja muito mais fácil injetar código novo no sistema antigo. Dessa forma, a tática de extrair código que pode ser testado lentamente fica mais fácil.

Infelizmente, a mesma técnica que permite facilitar a injeção de código novo também pode ter sido usada no sistema legado tornando seu entendimento muito mais complexo e difícil de se testar. A recomendação continua sendo a mesma: use o código mais simples que puder para escrever sua solução e tenha testes simples e diretos. Se precisar utilizar algum recurso mais avançado da linguagem, limite sua influência o quanto puder e torne-o o mais óbvio possível.

Durante essa reescrita, pode ser que o desenvolvedor sinta a necessidade de escrever testes “feios”, como testes para verificar *stored procedures* ou coisas do tipo. Nesses casos, escreva o teste, implemente a nova solução, elimine a *stored procedure* bem como o seu teste. Em sistemas legados, não tenha medo de escrever código de teste que será jogado fora depois. Se fizermos uma analogia com a engenharia nesse momento, imagine que esses códigos são como os andaimes utilizados durante a obra. Eles não fazem parte do produto final, estão lá apenas para ajudar no desenvolvimento.

Por mim, essa é a sugestão. Escreva testes para o sistema legado da maneira que conseguir, refatore o código já criando uma bateria de testes amigável. Ao final, caso faça sentido, jogue fora o código legado bem como testes que não são mais úteis.

A dificuldade em se testar sistemas legados não se aplica para novas funcionalidades. Nelas, ele tem a chance de escrever um código melhor e mais fácil de ser testado. Alguns padrões de projeto, como o *Adapter*, ajudam o desenvolvedor a “conectar” código bem escrito com o legado que deve ser mantido.

12.7 CONCLUSÃO

TDD é uma excelente ferramenta, deve constar no cinto de utilidades de todo desenvolvedor, mas obviamente deve ser usada nos momentos certos. Está em um momento complicado do desenvolvimento daquela classe? Use TDD. Está em um momento em que não há muitas dúvidas sobre o que fazer? Não se sinta mal por escrever testes após a implementação.

Lembre-se, o objetivo final é escrever código de qualidade.

CAPÍTULO 13

E agora?

Agora que TDD já foi discutido sob todos os pontos de vista, qual o próximo passo? Neste capítulo, finalizo a discussão apontando para outras excelentes referências sobre o assunto, e quais os próximos passos para o praticante.

13.1 COMO APRENDER MAIS AGORA?

A referência mais famosa sobre TDD é o livro do Kent Beck, intitulado “Test-Driven Development: By Example”. Ele é um livro bem introdutório sobre TDD. Ele discute aos poucos como escrever os testes antes, como ser simples, o que são baby steps etc. Ao longo do livro ele cria uma classe `Dinheiro`, que realiza operações financeiras em unidades monetárias distintas. Talvez seja uma ótima referência para quem nunca leu nada sobre o assunto e quer começar.

Um excelente livro é o famoso “Growing Object Oriented Software, Gui-

ded by Tests”, escrito por Steve Freeman e Nat Pryce. Neste livro, os autores mostram como TDD pode ajudar no design de suas classes. Ambos os autores são famosos na comunidade por suas diversas contribuições para a comunidade ágil e código aberto. É um livro que vale a pena ser lido.

Falando mais sobre design de classes, um bom livro sobre o assunto é o “Agile Software Development, Principles, Patterns, and Practices” do Robert Martin. Nele são discutidos os vários princípios de orientação a objetos de maneira profunda. Este livro contém um apêndice sobre isso, mas com certeza o livro do Uncle Bob é uma excelente referência para quem deseja se aprofundar em design orientado a objetos.

Por fim, aponto também para o blog da Caelum (<http://blog.caelum.com.br>) e meu blog pessoal (<http://www.aniche.com.br>), onde tenho escrito bastante conteúdo sobre o assunto. Os posts, aliás, foram uma grande motivação para a escrita desse livro.

Caso você prefira uma abordagem com cursos presenciais, certamente os cursos da Caelum, os quais ajudei a criar, são uma excelente fonte de aprendizado:

<http://www.caelum.com.br/cursos/agile/>

13.2 DIFICULDADE NO APRENDIZADO

Nos últimos anos, tenho dado diversas aulas e workshops sobre o assunto, e hoje posso afirmar que as maiores dificuldades na hora de se praticar TDD são:

- **Falta de conhecimento em orientação a objetos.** Quando o desenvolvedor não conhece orientação a objetos a fundo, ele sente dificuldades no momento da escrita do teste. Entender bem os conceitos e princípios do paradigma o ajudará a aprender TDD mais facilmente.
- **Pensar em cenários para os testes.** Pensar em cenários não é algo tão natural. Lembre-se que um teste automatizado é muito parecido com um teste manual. Imagine quais os cenários que você testaria em sua aplicação como um todo, e vá descendo de nível até imaginar como aquela classe deve responder de acordo com as diferentes entradas.

Como toda nova prática introduzida em um ambiente de trabalho, é normal que a produtividade caia no começo. Para que TDD torne-se natural para a equipe, é necessário que seja praticado constantemente. Existem diversos exercícios e vídeos explicativos para que o desenvolvedor consiga praticar e dominar melhor a prática.

Junto com a Caelum, escrevi um curso online sobre o assunto. Ele pode ser encontrado no próprio site da Caelum (<http://www.caelum.com.br/online>). Lá, produzi um conjunto de vídeos bem didáticos sobre a utilização de TDD em um sistema de leilões. Talvez essa seja uma ótima maneira para você treinar sua equipe também.

13.3 COMO INTERAGIR COM OUTROS PRATICANTES?

Como falado no começo do livro, criei um grupo de discussão no Google Groups. Você pode se cadastrar em <http://forum.casadocodigo.com.br/>.

Também abuse do GUJ, uma comunidade de perguntas e respostas para tirar dúvidas técnicas: <http://www.guj.com.br/>

13.4 CONCLUSÃO FINAL

Testar é vital para a qualidade de qualquer sistema. Lembre-se que qualidade é algo de que não podemos abrir mão, dada a importância que sistemas de computador têm hoje perante a sociedade. Ao longo deste livro, tentei mostrar como pode ser fácil e divertido a escrita de testes automatizados. Espero ter mostrado também que a busca por testabilidade acaba guiando o programador a um design de classes melhor e mais sustentável.

Agora depende de você. É hora de escrever testes!

CAPÍTULO 14

Apêndice: princípios SOLID

Ao longo do curso, discutimos diversos princípios de design, de maneira às vezes informal. Este apêndice serve para consolidar os princípios apresentados ao decorrer do livro. Para maiores informações sobre eles, é sugerido que o leitor busque os trabalhos do Robert Martin [27].

14.1 SINTOMAS DE PROJETOS DE CLASSES EM DEGRADAÇÃO

Diz-se que um projeto de classes está “degradando” quando ele começa a ficar difícil de evoluir, o reúso de código se torna mais complicado do que repetir o trecho de código, ou o custo de se fazer qualquer alteração no projeto de classes se torna alto.

Robert Martin enumerou alguns sintomas de projeto de classes em de-

gradação, chamados também de “maus cheiros” de projeto de classes. Esses sintomas são parecidos com os maus cheiros de código (*code smells*), mas em um nível mais alto: eles estão presentes na estrutura geral do software em vez de estarem localizados em apenas um pequeno trecho de código.

Esses sintomas podem ser medidos de forma subjetiva e algumas vezes de forma até objetiva. Geralmente, esses sintomas são causados por violações de um ou mais princípios de projeto de classes. Muitos desses problemas são relacionados à gerência de dependências. Quando essa atividade não é feita corretamente, o código gerado torna-se difícil de manter e reusar. Entretanto, quando bem feita, o software tende a ser flexível, robusto e suas partes reusáveis.

Rigidez

Rigidez é a tendência do software em se tornar difícil de mudar, mesmo de maneiras simples. Toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes. Quanto mais módulos precisam ser modificados, maior é a rigidez do projeto de classes.

Quando um projeto de classes está muito rígido, não se sabe com segurança quando uma mudança terá fim. Mudanças simples passam a demorar muito tempo até serem aplicadas no código e frequentemente acabam superando em várias vezes a estimativa de esforço inicial. Frases como *“isto foi muito mais complicado do que eu imaginei*

tornam-se populares. Neste momento, gerentes de desenvolvimento começam a ficar receosos em permitir que os desenvolvedores consertem problemas não críticos.

Fragilidade

Fragilidade é a tendência do software em quebrar em muitos lugares diferentes toda vez que uma única mudança acontece. Frequentemente, os novos problemas ocorrem em áreas não relacionadas conceitualmente com a área que foi mudada, tornando o processo de manutenção demasiadamente custoso, complexo e tedioso.

Consertar os novos problemas geralmente passa a resultar em outros novos problemas e assim por diante. Infelizmente, módulos frágeis são comuns

em sistemas de software. São estes os módulos que sempre aparecem na lista de defeitos a serem corrigidos. Além disso, desenvolvedores começam a ficar receosos de alterar certos trechos de código, pois sabem que estes estão tão frágeis que qualquer mudança simples fatalmente acarretará na introdução de problemas inesperados e de naturezas diversas.

Imobilidade

Imobilidade é a impossibilidade de se reusar software de outros projetos ou de partes do mesmo projeto. Neste cenário, o módulo que se deseja reutilizar frequentemente tem uma bagagem muito grande de dependências e não possui código claro. Depois de muita investigação, os arquitetos descobrem que o trabalho e o risco de separar as partes desejáveis das indesejáveis são tão grandes, que o módulo acaba sendo reescrito ao invés de reutilizado.

Viscosidade

Quando uma mudança deve ser realizada, geralmente há várias opções para realizar tal mudança. Quando as opções que preservam o projeto de classes são mais difíceis de serem implementadas do que aquelas que não o preservam, há alta viscosidade de projeto de classes. Neste cenário, é fácil fazer a “coisa errada” e é difícil fazer a “coisa certa”, ou seja, é difícil preservar e aprimorar o projeto de classes.

Complexidade desnecessária

Detecta-se complexidade desnecessária no projeto de classes quando ele contém muitos elementos inúteis ou não utilizados (*dead code*). Geralmente ocorre quando há muito projeto inicial (*up-front design*) e não se segue uma abordagem de desenvolvimento iterativa e incremental, de modo que os projetistas tentam prever uma série de futuros requisitos para o sistema e concebem um projeto de classes demasiadamente flexível ou desnecessariamente sofisticado.

Frequentemente apenas algumas previsões acabam se concretizando ao longo do tempo e, neste meio período, o projeto de classes carrega o peso de elementos e construções não utilizados. O software então se torna complexo

e difícil de ser entendido. Projetos com complexidade muito alta comumente afetam a produtividade, porque quando os desenvolvedores herdaram tal projeto, eles gastam muito tempo aprendendo as nuances do projeto de classes antes que possam efetivamente estendê-lo ou mantê-lo de maneira confortável.

Repetição desnecessária

Quando há repetição de trechos de código, é sinal de que uma abstração apropriada não foi capturada durante o processo de projeto de classes (ou inclusive na análise). Esse problema é frequente e é comum encontrar softwares que contenham dezenas e até centenas de elementos com códigos repetidos.

Descobrir a melhor abstração para eliminar a repetição de código geralmente não está na lista de itens de alta prioridade dos desenvolvedores, de modo que a resolução do problema acaba sendo eternamente postergada. Também, o sistema se torna cada vez mais difícil de entender e principalmente de se manter, pois os problemas encontrados em uma unidade de repetição devem ser corrigidos potencialmente em toda repetição, com o agravante de que uma repetição pode ter forma ligeiramente diferente de outra.

Opacidade

Opacidade é a tendência de um módulo ser difícil de ser entendido. Códigos podem ser escritos de maneira clara e expressiva ou de maneira “opaca” e complicada. A tendência de um código é se tornar mais e mais opaco à medida que o tempo passa e, para que isso seja evitado, é necessário um esforço constante em manter esse código claro e expressivo.

Uma maneira para prevenir isso é fazer com que os desenvolvedores se ponham no papel de leitores do código e o refatorem de maneira que qualquer outro leitor possa entender. Além disso, revisões de código feita por outros desenvolvedores é também uma possível solução para manter o código menos opaco.

14.2 PRINCÍPIOS DE PROJETO DE CLASSES

Todos os problemas citados na seção anterior podem ser evitados pelo uso puro e simples de orientação a objetos. A máxima da programação orientada a objetos diz que classes devem possuir um baixo acoplamento e uma alta coesão.

Alcançando esse objetivo, mudanças no código seriam executadas mais facilmente; alterações seriam feitas em pontos únicos e a propagação de mudanças seria bem menor. Com as abstrações bem estabelecidas, novas funcionalidades seriam implementadas através de novo código, sem a necessidade de alterações no código já existente. Necessidades de evolução do projeto de classes seriam feitas com pouco esforço, já que módulos dependeriam apenas de abstrações.

Mas alcançar tal objetivo não é tarefa fácil. Criar classes pouco acopladas e altamente coesas demanda um grande esforço por parte do desenvolvedor e requer grande conhecimento e experiência no paradigma da orientação a objetos.

Os princípios comentados nesta seção são muito discutidos por Robert Martin em vários de seus livros e artigos publicados. Esses princípios são produto de décadas de experiência em engenharia de software. Segundo ele, eles não são produto de uma única pessoa, mas sim a integração de pensamentos e trabalhos de um grande número de desenvolvedores de software e pesquisadores, e visam combater todos os sintomas de degradação discutidos aqui.

Conhecidos pelo acrônimo **SOLID** (sólido, em português), são eles:

- Princípio da Responsabilidade Única (*Single-Responsibility Principle* (SRP))
- Princípio do Aberto-Fechado (*Open-Closed Principle* (OCP))
- Princípio de Substituição de Liskov (*Liskov Substitution Principle* (LSP))
- Princípio da Inversão de Dependência (*Dependency Inversion Principle* (DIP))

- Princípio da Segregação de Interfaces (*Interface Segregation Principle* (ISP))

Princípio da Responsabilidade Única

O termo coesão define a relação entre os elementos de um mesmo módulo. Isso significa que os todos elementos de uma classe que tem apenas uma responsabilidade tendem a se relacionar. Diz-se que uma classe como essa é uma classe que possui alta coesão (ou que é coesa). Já em uma classe com muitas responsabilidades diferentes, os elementos tendem a se relacionar apenas em “grupos”, ou seja, com os elementos que tratam de uma das responsabilidades da classe. A esse tipo de classe, diz-se que ela possui uma baixa coesão (ou que não é coesa). Robert Martin altera esse conceito de coesão e a relaciona com as forças que causam um módulo ou uma classe a mudar. No caso, o Princípio de Responsabilidade Única diz que uma classe deve ter apenas uma única razão para mudar.

Esse princípio é importante no momento em que há uma alteração em alguma funcionalidade do software. Quando isso ocorre, o programador precisa procurar pelas classes que possuem a responsabilidade a ser modificada. Supondo uma classe que possua mais de uma razão para mudar, isso significa que ela é acessada por duas partes do software que fazem coisas diferentes. Fazer uma alteração em uma das responsabilidades dessa classe pode, de maneira não intencional, quebrar a outra parte de maneira inesperada. Isso torna o projeto de classes frágil.

Princípio do Aberto-Fechado

O Princípio do Aberto-Fechado, cunhado por Bertrand Meyer, diz que as entidades do software (como classes, módulos, funções etc.) devem ser abertas para extensão, mas fechadas para alteração. Se uma simples alteração resulta em uma cascata de alterações em módulos dependentes, isso cheira à rigidez. O princípio pede então para que o programador sempre refatore as classes de modo que mudanças desse tipo não causem mais modificações.

Quando esse princípio é aplicado de maneira correta, novas alterações fazem com que o programador adicione novo código, e não modifique o anterior. Isso é alcançado através da criação de abstrações para o problema.

Algumas linguagens orientadas a objetos possuem mecanismos para criá-las como interfaces em Java e C# enquanto outras, como Ruby, deixam o desenvolvedor criar essas abstrações implicitamente. Através dessas abstrações, o programador consegue descrever a maneira como uma determinada classe deve se portar, mas sem se preocupar em como essa classe faz isso.

Princípio de Substituição de Liskov

Esse princípio, que discute sobre tipos e subtipos, criado por Barbara Liskov em 1988, é importante já que herança é uma das maneiras para se suportar abstrações e polimorfismo em linguagens orientadas a objetos. E, como visto anteriormente, o Princípio do Aberto-Fechado se baseia fortemente na utilização desses recursos.

O problema é que utilizar herança não é tarefa fácil, pois o acoplamento criado entre classe filha e classe pai é grande. Fazer as classes filhas respeitarem o contrato do pai, e ainda permitir que mudanças na classe pai não influenciem nas classes filhas requer trabalho.

O princípio de Liskov diz que, se um tipo S é subclasse de um tipo T, objetos do tipo T podem ser substituídos por objetos do tipo S, sem alterar nenhuma das propriedades desejadas daquele programa.

Um clássico exemplo sobre Princípio de Substituição de Liskov é o dos Quadrados e Retângulos. Imagine uma classe Retângulo. Um retângulo possui dois lados de tamanhos diferentes. Imagine agora uma classe Quadrado (figura geométrica que possui todos os lados com o mesmo tamanho) que herde de Retângulo. A única alteração é fazer com que os dois lados tenham o mesmo tamanho. Apesar de parecer lógico, afinal um Quadrado é um Retângulo com apenas uma condição diferente, a classe Quadrado quebra o Princípio de Liskov: a pré-condição dela é mais forte do que a do quadrado, afinal os dois lados devem ter o mesmo tamanho.

Quebras do princípio de Liskov geralmente levam o programador a quebrar o princípio do OCP também. Ele percebe que, para determinados subtipos, ele precisa fazer um tratamento especial, e acaba escrevendo condições nas classes clientes que fazem uso disso.

Princípio da Inversão de Dependências

Classes de baixo nível, que fazem uso de infraestrutura ou de outros detalhes de implementação podem facilmente sofrer modificações. E, se classes de mais alto nível dependerem dessas classes, essas modificações podem se propagar, tornando o código frágil.

O Princípio de Inversão de Dependências se baseia em duas afirmações:

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações;
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Em resumo, as classes devem, na medida do possível, acoplar-se sempre com módulos mais estáveis do que ela própria, já que, como as mudanças em módulos estáveis são menos prováveis, raramente essa classe precisará ser alterada por mudanças em suas dependências.

Princípio da Segregação de Interfaces

Acoplar-se com uma interface de baixa granularidade (ou gordas, do termo em inglês *fat interfaces*) pode ser perigoso, já que qualquer alteração que um outro cliente forçar nessa interface poderá ser propagada para essa classe.

O princípio da segregação de interfaces diz que classes cliente não devem ser forçadas a depender de métodos que elas não usam. Quando uma interface não é coesa, ela contém métodos que são usados por um grupo de clientes, e outros métodos que são usados por outro grupo de clientes. Apesar de uma classe poder implementar mais de uma interface, o princípio diz que o cliente da classe deve apenas depender de interfaces coesas.

14.3 CONCLUSÃO

Todos os princípios discutidos na seção anterior tentam diminuir os possíveis problemas de projeto de classes que possam eventualmente aparecer. Discutir o que é um bom projeto de classes é algo difícil; mas é possível enumerar

algumas das características desejáveis: isolar elementos reusáveis de elementos não reusáveis, diminuir a propagação de alterações em caso de uma nova funcionalidade.

Um bom programador OO deve conhecer e aplicar esses princípios ao longo de sua base de código.

CAPÍTULO 15

Apêndice: RSpec como biblioteca de testes de unidade

A comunidade Ruby, como de toda linguagem amplamente adotada, tem várias bibliotecas que cumprem a mesma função de formas diferentes. Este apêndice apresenta o RSpec, uma biblioteca que pode ser usada no lugar do TestUnit para escrever testes automatizados. A principal diferença entre o RSpec e o TestUnit são os nomes dos métodos que são utilizados para realizar as asserções. Veremos aqui alguns exemplos de testes escritos com RSpec em comparação com os testes TestUnit, suas sintaxes, suas saídas e, em suma, vantagens e desvantagens.

15.1 ASSESSÕES NA PRIMEIRA VERSÃO DO RSPEC

O TestUnit foi escrito inspirado no SUnit, JUnit e outras bibliotecas conhecidas como xUnit. Muito de seu design e detalhes de implementação vêm dessas bibliotecas mais antigas. Isso fica bastante evidente com o uso de uma super classe para as classes de testes (`Test::Unit::TestCase`) e métodos como `assert_equal`, `assert` e outros semelhantes.

Essa sintaxe e design acabou sendo conhecida e fortemente associada com a comunidade de praticantes de desenvolvimento dirigido por testes. Anos depois de TDD ter ganhado força, Dan North lançou um post [31] apontando algumas mudanças no estilo de TDD para encorajar desenvolvedores a pensarem mais no lado das regras de negócios na hora de escreverem seus testes. Esse movimento ganhou forças ao longo do tempo e ficou conhecido como desenvolvimento dirigido por comportamento, ou *behavior driven development*, BDD.

Pouco tempo depois, Dan lançou com alguns colegas a biblioteca JBehave para escrita de testes no estilo BDD em Java. O RSpec foi a implementação da ideia em Ruby mantida por David Chelimsky. A primeira versão seguia bastante a sintaxe apresentada no JBehave com alguns benefícios de implementação proporcionados pela linguagem Ruby.

Por exemplo, o `assert_equal` usado no primeiro exemplo com TestUnit:

```
assert_equal 'Jogo de pratos', algoritmo.menor.nome
assert_equal 'Geladeira', algoritmo.maior.nome
```

Em RSpec:

```
algoritmo.menor.nome.should == 'Jogo de pratos'
algoritmo.maior.nome.should == 'Geladeira'
```

ou

```
algoritmo.menor.nome.should eq('Jogo de pratos')
algoritmo.maior.nome.should eq('Geladeira')
```

Note que fica mais claro na versão RSpec qual é o argumento esperado e qual o argumento que está sendo calculado. Também fica mais natural de ler o código já que dizemos que a implementação deveria ser igual ao valor.

Mas existem outros tipos de asserções que também fazem bastante sentido nessa sintaxe. Veja os exemplos RSpec válidos a seguir:

```
@carrinho.itens.should include(fogao)
fatura.should be_paga
@carrinho.itens.should_not == 'Geladeira'
@carrinho.itens.should_not include(fogao)
fatura.should_not be_paga
```

Isso faz com que seja muito mais fácil escrever e ler as condições testadas do que apenas com `assert_equal` e `assert` do TestUnit.

15.2 BIBLIOTECA DE MOCKS EMBUTIDA NO RSpec

O RSpec inclui uma biblioteca de *mocks* embutida na biblioteca de testes. Em termos de funcionalidades, o Mocha e os *mocks* do RSpec são essencialmente os mesmos. Infelizmente a sintaxe dos dois é ligeiramente diferente. Veja o exemplo a seguir com Mocha:

```
# criando o mock
dao = mock
# marcando o que esperamos que aconteça
dao.expects(:executa).with(nota_fiscal).returns(true)
```

E o equivalente com o padrão do RSpec:

```
# criando o mock
dao = double
# marcando o que esperamos que aconteça
dao.should_receive(:executa).with(nota_fiscal).and_return(true)
```

Note que o RSpec não força o uso de sua sintaxe para *mocks* e é possível utilizar o Mocha com o RSpec mas não é muito comum já que requer duas bibliotecas em vez de uma.

15.3 COMPARANDO A SAÍDA DAS DUAS BIBLIOTECAS

Bibliotecas de testes automatizados têm duas macrofuncionalidades muito importantes. A primeira é a que já vimos de asserções. Como verificar que

o código de produção faz o que deveria fazer. *Mocks* são uma parte dessa macrofuncionalidade já que eles proveem uma forma de descrever um comportamento esperado do código de produção.

A segunda funcionalidade é a de gerar relatórios a partir da execução. Nesse livro falamos muito pouco dessa parte mas ela é crucial para que uma biblioteca de testes seja realmente útil. Em especial, a filosofia por trás de testes automatizados diz que, por padrão se o teste passa, eu não preciso ver nenhuma mensagem longa. Um simples sinal de que o teste foi executado e passou é suficiente. Na maioria das bibliotecas cuja saída é um texto, isso é representado por um ‘.’. No entanto, quando o teste falha é que a saída é mais importante para o desenvolvedor.

Vejamos a saída de alguns exemplos em TestUnit:

```
require 'test/unit'
require 'mocha/setup'
class Teste < Test::Unit::TestCase
  def teste_assert_equal
    assert_equal 0, (1+1)
  end
  def teste_assert
    assert (nil || false)
  end
  def teste_mock
    object = mock
    object.expects(:hi).with(2)

    object.hi(1)
  end
end

Started
FFF
Finished in 0.0039 seconds.

1) Failure:
teste_assert(Teste)
[test/teste_test.rb:8:in `teste_assert'
mocha-0.14.0/lib/mocha/integration/test_unit/
```

```

    ruby_version_186_and_above.rb:29:in `__send__'
    mocha-0.14.0/lib/mocha/integration/test_unit/
    ruby_version_186_and_above.rb:29:in `run']:
```

<false> is not true.

2) Failure:

```

teste_assert_equal(Teste)
  [test/teste_test.rb:5:in `teste_assert_equal'
  mocha-0.14.0/lib/mocha/integration/test_unit/
  ruby_version_186_and_above.rb:29:in `__send__'
  mocha-0.14.0/lib/mocha/integration/test_unit/
  ruby_version_186_and_above.rb:29:in `run']:
```

<0> expected but was
<2>.

3) Failure:

```

teste_mock(Teste) [test/teste_test.rb:14]:
unexpected invocation: #<Mock:0x1083a0990>.hi(1)
unsatisfied expectations:
- expected exactly once, not yet invoked:
    #<Mock:0x1083a0990>.hi(2)
```

3 tests, 2 assertions, 3 failures, 0 errors

E seus equivalentes em RSpec:

```

require 'rspec'
describe "Teste" do
  it "deve ser igual" do
    (1+1).should == 0
  end
  it "deve ser verdadeiro" do
    (nil || false).should be_true
  end
  it "deve ter mocks" do
    object = double
    object.should_receive(:hi).with(2)

    object.hi(1)
  end
end
```

```
end
```

```
FFF
```

```
Failures:
```

- ```
1) Teste deve ser igual
 Failure/Error: (1+1).should == 0
 expected: 0
 got: 2 (using ==)
 # ./spec/teste_spec.rb:4

2) Teste deve ser verdadeiro
 Failure/Error: (nil || false).should be_true
 expected: true value
 got: false
 # ./spec/teste_spec.rb:7

3) Teste deve ter mocks
 Failure/Error: object.hi(1)
 Double received :hi with unexpected arguments
 expected: (2)
 got: (1)
 # ./spec/teste_spec.rb:13
```

```
Finished in 0.00241 seconds
```

```
3 examples, 3 failures
```

```
Failed examples:
```

```
rspec ./spec/teste_spec.rb:3 # Teste deve ser igual
rspec ./spec/teste_spec.rb:6 # Teste deve ser verdadeiro
rspec ./spec/teste_spec.rb:9 # Teste deve ter mocks
```

A saída do TestUnit é simples. O nome do teste, a classe na qual o teste está definido, a forma com que se chegou nessa linha e a diferença entre o resultado esperado e o ocorrido. Apesar de ter todos os dados necessários nessa saída, ela é bastante densa e difícil de ser lida e entendida. Já a saída do RSpec tenta ser a mais sucinta possível com ênfase na linha de código que foi

responsável pelo erro, o que ocorreu e o que era esperado.

## 15.4 NOVAS ASSERÇÕES DO RSpec

Um dos grandes problemas de implementação do RSpec em suas primeiras versões é que, para atingir a sintaxe fluida que vimos até agora, todo objeto em Ruby tinha que responder ao método `should` com a implementação definida pela biblioteca. Não só isso, mas a questão do escopo dos métodos ficava toda um pouco confusa quando era necessário escrever testes para bibliotecas que lidavam com metaprogramação. Por conta desses problemas e de algumas melhorias no design interno do RSpec, as novas versões oferecem uma nova sintaxe para asserções que deve ser mais limpa internamente e deve resolver os problemas de escopo anteriores de forma que novas asserções possam ser escritas com mais facilidade.

Agora, o escopo que precisa conter os métodos de asserção e verificações é o de execução do teste e não mais o do objeto. Para isso, o `should` foi descontinuado em favor do método de teste `expect`. Veja um exemplo:

```
fatura.pagamentos.size.should == 1
virou
expect(fatura.pagamentos.size).to eq(1)
```

Essa implementação é muito parecida com a ideia de verificações por *matchers* que foi amplamente desenvolvida e adotada em Java com os *matchers* do Hamcrest [35].

A antiga sintaxe ainda deve ser válida por algum tempo mas, se puder escrever seus novos testes com a nova sintaxe, é altamente recomendado que o faça. Também ficou mais fácil escrever seus próprios *matchers* para que suas verificações possam dar mais informações ao *framework* de testes em casos de falha.

## 15.5 CONCLUSÃO

Obviamente, tudo isso é muito subjetivo e o importante é que ambas bibliotecas conseguem prover o *feedback* que um desenvolvedor precisa para escrever código correto e de qualidade. A diferença definitivamente não é importante

o suficiente para tornar o TestUnit obsoleto nem tão pequena que o RSpec não trás nenhum benefício. O mais importante é que a sua equipe concorde em uma ferramenta e aproveite tudo que a ferramenta pode trazer.

O RSpec exige um aprendizado maior já que a sintaxe é bem diferente das ferramentas mais comuns em outras linguagens. No entanto, uma vez aprendido, o RSpec oferece algumas funcionalidades interessantes.

No fim das contas, o melhor para Ruby é que as duas ferramentas existam e cresçam separadamente já que elas forçam uma a outra a melhorar e reciclam ideias e implementações.



# Referências Bibliográficas

- [1] Kent Beck. Tdd 10 years later. <http://junit.sourceforge.net/doc/testinfected/testing.htm>.
- [2] c2. Ten years of test driven development. <http://c2.com/cgi/wiki?TenYearsOfTestDrivenDevelopment>.
- [3] c2. Test data builder. <http://c2.com/cgi/wiki?TestDataBuilder>.
- [4] Philip Calçado. Evitando vos e bos. [http://fragmental.com.br/wiki/index.php/Evitando\\_VO\\_s\\_e\\_BO\\_s.html](http://fragmental.com.br/wiki/index.php/Evitando_VO_s_e_BO_s.html).
- [5] Paulo Caroli. Testing private methods, tdd, and test-driven refactoring. <http://agiletips.blogspot.com/2008/11/testing-private-methods-tdd-and-test.html>.
- [6] David Chelimsky. Rspec. <http://rspec.info/>.
- [7] Ryan Davis. Minitest framework. <https://github.com/seattlerb/minitest>.
- [8] Coding Dojo. Tdd 10 years later. <http://www.codingdojo.org>.
- [9] D. Janzen e H. Saiedian. On the influence of test-driven development on software design. 2006.
- [10] Caelum Ensino e Inovação. Formação online de testes automatizados. <http://www.caelum.com.br/curso/online/testes-automatizados/>.
- [11] K. M. Lui e K. C. C. Chan. Test-driven development and software process improvement in china. 2004.

- [12] L.-O. Damn e L. Lundberg. Introducing test automation and test-driven development: An experience report. 2005.
- [13] B. George e L. Williams. An initial investigation of test- driven development in industry. 2003.
- [14] E. M. Maximilien e L. Williams. Assessing test-driven development at ibm. 2003.
- [15] Mauricio Aniche e Marco Aurélio Gerosa. Most common mistakes in test-driven development practice: Results from an online survey with developers. 2010.
- [16] M. M. Müller e O. Hagner. Experiment about test-first programming. 2002.
- [17] Michael Feathers e Steve Freeman. Tdd 10 years later. <http://www.infoq.com/presentations/tdd-ten-years-later>.
- [18] S. H. Edwards. Using test-driven development in a classroom: Providing students with automatic, concrete feedback on performance. 2003.
- [19] Michael Feathers. The deep synergy between testability and good design. [http://michaelfeathers.typepad.com/michael\\_feathers\\_blog/2007/09/the-deep-synerg.html](http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html), 2008.
- [20] et al Freeman, Bates. *Head First Design Patterns*. O'Reilly Media, 2004.
- [21] et al. H. Erdogmus, M. Morisio. On the effectiveness of the test-first approach to programming. 2005.
- [22] D. Janzen. Software architecture improvement through test-driven development. 2005.
- [23] J. Langr. Evolution of test and code via test-first design. <http://www.objectmentor.com/resources/articles/tfd.pdf>, 2010.
- [24] Delamaro Maldonado, Jino. *Introdução ao Teste de Software*. Editora Campus, 2007.

- [25] Robert Martin. Stability. <http://www.objectmentor.com/resources/articles/stability.pdf>.
- [26] Robert Martin. The transformation priority premise. <http://cleancoder.posterous.com/the-transformation-priority-premise>.
- [27] Robert Martin. *Agile Practices, Patterns, and Practices in C*. Prentice Hall, 2006.
- [28] Marco Aurélio Gerosa Mauricio Aniche, Thiago Ferreira. What concerns beginner test-driven development practitioners: A qualitative analysis of opinions in an agile conference. 2011.
- [29] James Mead. Mocha. <http://gofreerange.com/mocha/docs/>.
- [30] e T. Bhat N. Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. 2006.
- [31] Dan North. Behaviour driven development. <http://dannorth.net/introducing-bdd/>.
- [32] Joel on Software. Things you should never do, part i. <http://www.joelonsoftware.com/articles/fogooooooooo69.html>.
- [33] D. H. Steinberg. The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. 2001.
- [34] Nathaniel Talbott. Test-unit framework. <http://test-unit.rubyforge.org/>.
- [35] Joe Walnes. Hamcrest. <http://hamcrest.org/>.
- [36] Wikipedia. Cyclomatic complexity. [http://en.wikipedia.org/wiki/Cyclomatic\\_complexity](http://en.wikipedia.org/wiki/Cyclomatic_complexity).
- [37] Wikipedia. Numeração romana. [http://pt.wikipedia.org/wiki/Numera%C3%A7%C3%A3o\\_romana](http://pt.wikipedia.org/wiki/Numera%C3%A7%C3%A3o_romana).
- [38] Computer World. Study: Buggy software costs users, vendors nearly 60b annually. [http://www.computerworld.com/s/article/72245/Study\\_Buggy\\_software\\_costs\\_users\\_vendors\\_nearly\\_60B\\_annually](http://www.computerworld.com/s/article/72245/Study_Buggy_software_costs_users_vendors_nearly_60B_annually).