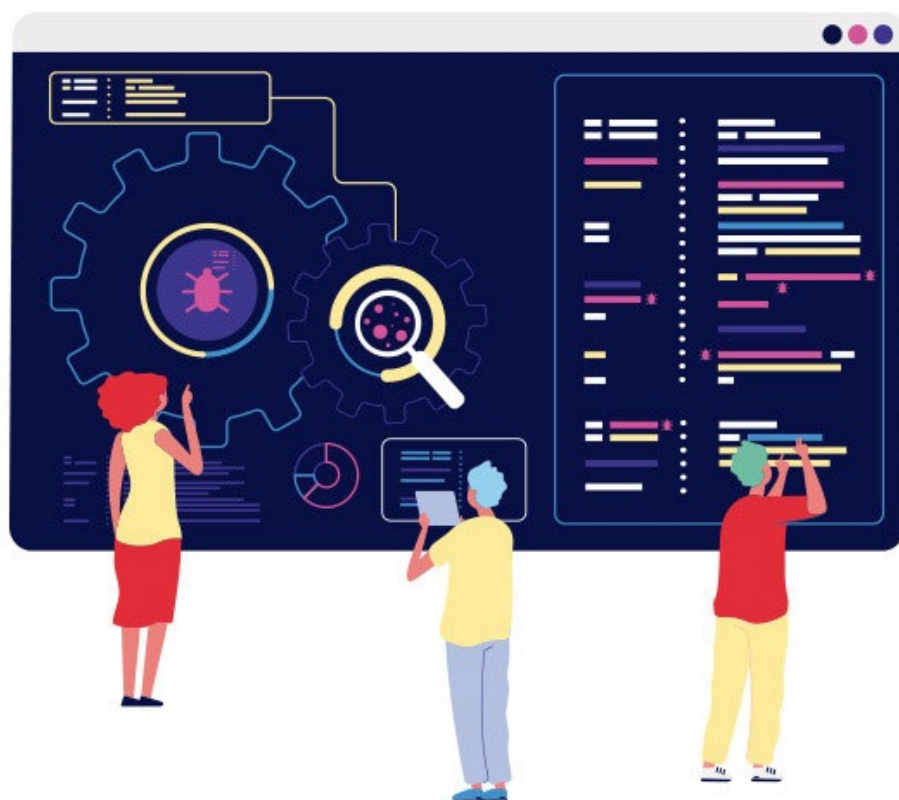


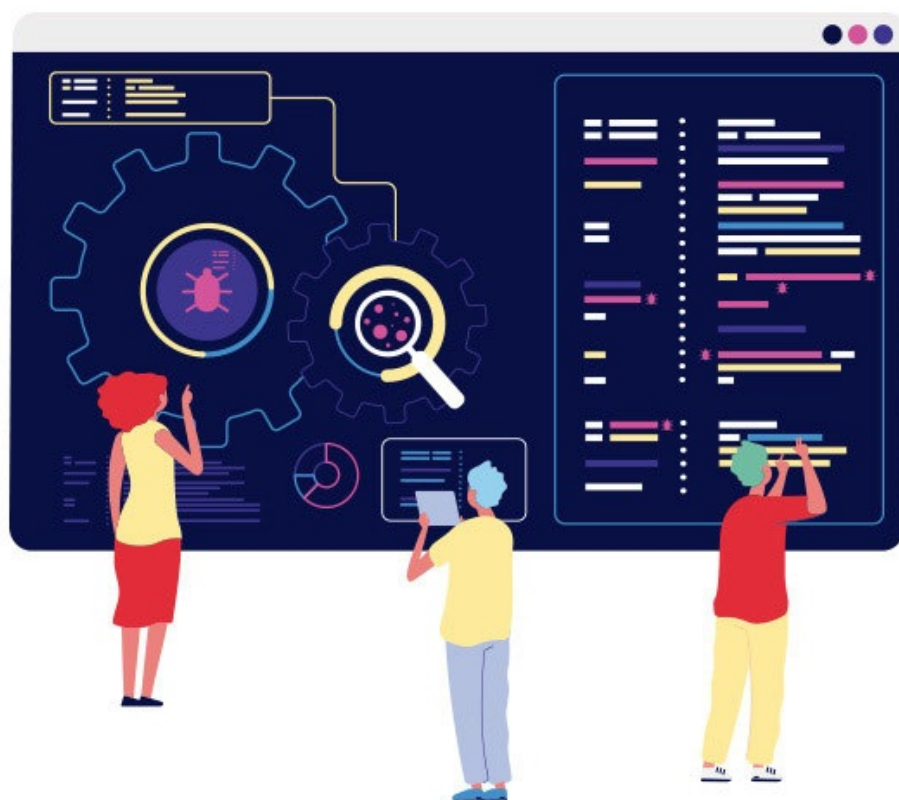
TDD e BDD na prática

Construa aplicações Ruby usando
RSpec e Cucumber



TDD e BDD na prática

Construa aplicações Ruby usando
RSpec e Cucumber



Sumário

[ISBN](#)

[Agradecimentos](#)

[Sobre os autores](#)

[Apresentação](#)

[1. Visão geral sobre TDD](#)

[2. Primeiros passos com RSpec e Cucumber](#)

[3. Conhecendo o RSpec](#)

[4. Organização, refatoração e reúso de testes com o RSpec](#)

[5. Mocks e stubs](#)

[6. Conhecendo o Cucumber](#)

[7. Especificando funcionalidades com Cucumber](#)

[8. Automatizando especificações com Cucumber](#)

[9. Boas práticas no uso de Cucumber](#)

[10. BDD na prática: começando um projeto com BDD](#)

[11. Começando o segundo cenário](#)

[12. Finalizando a primeira funcionalidade](#)

[13. Refatorando nosso código](#)

[14. Especificando a segunda funcionalidade](#)

[15. Finalizando a segunda funcionalidade](#)

[16. Finalizando nosso jogo](#)

[17. Referências bibliográficas](#)

ISBN

Impresso e PDF: 978-65-86110-29-6

EPUB: 978-65-86110-30-2

MOBI: 978-65-86110-31-9

■

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

■

Agradecimentos

Hugo Baraúna

Primeiro eu gostaria de agradecer ao Adriano Almeida da Casa do Código, pelo convite para escrever este livro. Sem esse primeiro empurrão é muito improvável que este livro existisse. Escrevê-lo foi não só um excelente desafio, mas também uma grande oportunidade de aprendizado.

Outra pessoa essencial neste trabalho é o Philippe Hardardt, meu co-autor. Sem ele esta segunda edição do livro não existiria.

Gostaria de agradecer também à Plataformatec e a todo mundo que trabalhou comigo lá. São os melhores profissionais com quem eu já trabalhei e me inspiram todo dia a ser um profissional melhor.

Não posso deixar também de agradecer à minha família e aos amigos como um todo. Ao fazer este livro, tive que abdicar de passar inúmeras horas com eles, decisão que não foi fácil de tomar e muito menos de manter. Um obrigado especial à minha mãe, Lilian Pessoa e à minha esposa, Ana Augusto.

Por fim, gostaria de agradecer às pessoas que investiram seu tempo revisando a primeira edição deste livro. Eles não tinham nenhuma obrigação de fazê-lo, fizeram pela boa vontade, amizade, por serem colaborativos e pela vontade de aprender algo novo. São eles: Bernardo Chaves, Erich Kist, Danilo Inacio e Anna Cruz.

Philippe Hardardt

Em primeiro lugar, quero agradecer ao Igor Florian que me deu um empurrão no começo e me deu confiança para participar desta segunda edição. Obrigado também, Hugo, pela oportunidade de trabalhar neste livro, que já era uma referência para mim.

Gostaria de agradecer a todos que trabalharam comigo na Plataformatec, onde aprendi muito e conheci pessoas incríveis.

Por fim, quero agradecer à minha família, à minha namorada e aos meus amigos pelo apoio durante todo o tempo que trabalhei na segunda edição. Um agradecimento especial ao Guilherme Garcia, que me ajudou a deslanchar na fase em que eu estava mais bloqueado.

Sobre os autores

Hugo Baraúna

Hugo Baraúna fundou a Plataformatec, empresa de consultoria de software especializada em Ruby, agile e Elixir. A Plataformatec foi a empresa que criou a linguagem Elixir. Também foi uma referência nacional e internacional no mundo Ruby, devido principalmente a seus projetos open source, como Devise e Simple Form. Hugo se formou em Engenharia de Computação pela Politécnica da USP e tem um MBA pelo Insper. Atualmente, ele mora em São Paulo com sua esposa e seus dois gatinhos.

Philippe Hardardt

Philippe trabalha com desenvolvimento de software desde 2008 e já experimentou muitas linguagens de programação diferentes, desde ASP até Clojure. Tem um interesse especial por tópicos que se aplicam a qualquer tecnologia, como testes automatizados. Teve a oportunidade de trabalhar na Plataformatec até sua aquisição pelo Nubank, onde trabalha atualmente.

Apresentação

O que você vai aprender aqui?

Saber fazer testes automatizados é uma habilidade essencial para todo mundo que desenvolve software atualmente. Toda vez que você cria um projeto novo com um framework moderno de software, essa ferramenta vai criar uma pasta para você colocar seus testes. Isso levanta a importância de duas questões:

Como devo escrever testes que ajudem a garantir o funcionamento do meu código?

Como devo escrever testes de modo bem feito?

Neste livro você vai desenvolver as habilidades necessárias para responder essas duas perguntas.

Você aprenderá desde os fundamentos, tal como sobre como escrever um teste que o ajude a garantir a qualidade do seu software, até tópicos mais avançados, tais como:

usar testes automatizados para ajudar no design do seu código, usando TDD e BDD;

a diferença entre mocks e stubs, quando usar um ou outro;

como criar uma documentação executável do seu software e como isso pode ser útil.

Abordagem do livro

Este livro não é um manual do Cucumber e de RSpec. Já existem diversos lugares listando as funcionalidades, classes e métodos do Cucumber e RSpec. Portanto, o objetivo não é repetir o que já existe pronto em vários outros lugares.

A abordagem deste livro é apresentar como testar, em vez de explorar os detalhes de cada ferramenta. Como organizar seus testes e como aplicar TDD e BDD é um conhecimento que você pode usar em outras linguagens de programação. Além disso, várias boas práticas não documentadas previamente também são apresentadas com exemplos ao longo do livro inteiro.

Estrutura do livro

Este livro está estruturado em quatro partes:

A primeira consiste em uma introdução ao conceito e histórico do TDD e BDD, assim como um primeiro contato com o RSpec e com o Cucumber. Ela é formada pelos capítulos Visão geral sobre TDD e Primeiros passos com RSpec e Cucumber;

A segunda é uma apresentação geral do RSpec. Passando pela estrutura básica de um teste de unidade feito com ele, pela organização de testes e pelo uso de test doubles como mocks e stubs. Ela é formada pelos capítulos Conhecendo o RSpec a Mocks e stubs;

A terceira parte consiste em uma apresentação do Cucumber e de como usá-lo para escrever especificações executáveis. Ela é formada pelos capítulos Conhecendo o Cucumber a Boas práticas no uso de Cucumber;

Por fim, na quarta e última parte, nós construiremos uma aplicação do zero seguindo o conceito de outside-in development do BDD, utilizando RSpec e Cucumber. Ela é formada pelos capítulos BDD na prática: começando um projeto com BDD a Finalizando nosso jogo.

Para quem é este livro?

Estou aprendendo ou já sei programar em Ruby mas nunca fiz TDD

Se você se enquadra nesse caso, este livro é perfeito para você. Você vai aprender como fazer testes automatizados e seguir o fluxo de TDD e BDD para fazer um código mais fácil de ser mantido e com mais qualidade.

Aprender uma habilidade nova não é simples, mas pode ser mais eficiente com a ajuda de uma apresentação estruturada e de um caminho definido. Este livro mostra passo a passo como usar o RSpec e o Cucumber para construir uma aplicação inteira seguindo o fluxo de TDD/BDD.

Já faço testes automatizados mas não sei se estou fazendo do jeito certo

Existem diversos fatores que influenciam o desenvolvimento de bons testes. O que testar? Como testar? Em qual camada testar? Por onde começo?

Fazer testes do jeito certo não é apenas ter cobertura de testes em 100%. Por exemplo, a maioria das pessoas não sabe que em um teste a clareza é muito mais importante do que o DRY (don't repeat yourself). A maioria das pessoas não sabe a diferença entre mocks e stubs e quando usar um ao invés do outro.

Este livro responde todas essas dúvidas e mais várias outras, que o ajudarão a estruturar seu conhecimento em testes automatizados e a escrever testes de qualidade.

Já fiz testes de unidade, mas não conheço Cucumber, nem o conceito de especificação executável

Testes de unidade são uma parte muito importante na prática de TDD, mas existem outras camadas de teste, tal como a camada de testes de aceitação. Este livro mostra o que são testes de aceitação e como fazê-los utilizando Cucumber.

Na explicação do uso de Cucumber, este livro vai além dos testes de aceitação. O Cucumber é, na verdade, uma ferramenta de documentação, que une especificação e testes automatizados, formando o que conhecemos por "especificação executável".

Este livro mostra como usar o Cucumber do modo certo, quando vale a pena usá-lo e como usá-lo em conjunto com o RSpec para fechar o ciclo de outside-in development.

O que preciso ter instalado?

Ao longo do livro veremos vários exemplos de código, e na última parte construiremos um projeto de 0 a 100. Para ir desenvolvendo o código junto com o livro, você precisará de algumas coisas instaladas.

Você precisará ter instalado o Ruby 2.6 ou 2.5. Qualquer uma das duas versões deve funcionar, mas dê preferência para a 2.6.

Além do Ruby, será necessário instalar o Bundler, RSpec e Cucumber. A versão do Bundler utilizada é a 2.0.1. A versão do Cucumber é a 3.1.2. As versões do RSpec e de seus componentes usadas são:

rspec: 3.8.0

rspec-core: 3.8.0

rspec-expectations: 3.8.2

rspec-mocks: 3.8.0

Sobre o sistema operacional, você pode usar o macOS, o Linux ou Windows. Dito isso, historicamente o Ruby funciona melhor no Linux e no macOS do que no Windows. Portanto, se você for usuário de Windows e encontrar problemas, uma boa opção é usar uma máquina virtual rodando Linux.

Um último detalhe sobre o sistema operacional, ao longo do livro alguns comandos de shell Unix-like são usados, tal como o `mkdir` para criar diretório, o `cd` para entrar em um diretório e o `touch` para criar um arquivo. Apesar de serem usados comandos de um shell Unix-like, a intenção de cada comando será explicada ao longo do livro, de modo que você possa saber o que deve ser feito, caso não esteja usando um shell Unix-like, como no Windows.

Se você tiver alguma dúvida sobre o código desenvolvido na quarta parte deste livro, a parte do projeto, você pode verificar um repositório no GitHub com o código final como referência: <https://github.com/hugobarauna/forca/>.

Capítulo 1

Visão geral sobre TDD

Uma das melhores qualidades da comunidade Ruby é o uso constante de TDD (Test-Driven Development). Todo projeto open source famoso na nossa comunidade tem uma suíte de testes automatizados. É o padrão e o dia a dia de todo desenvolvedor Ruby. Se você não faz isso, você ainda não é um desenvolvedor Ruby completo. Ao ler este livro, você está dando mais alguns passos em direção a ser um desenvolvedor melhor.

Durante a leitura deste livro, você vai aprender a fazer TDD, usando algumas das ferramentas mais famosas na comunidade Ruby: o RSpec e o Cucumber. Você verá como usar essas ferramentas e também como aplicar a técnica de TDD para construir um software de mais qualidade.

Se você ainda não conhece ou conhece pouco sobre TDD, prepare-se, porque essa prática vai mudar o modo como você escreve software... Para melhor!

1.1 TDD e sua história

A história do TDD começa principalmente no começo da década de 90, quando Kent Beck escreveu, em Smalltalk, sua primeira biblioteca de testes, o SUnit. A ideia inicial dele era facilitar a execução de testes de software, automatizando essa tarefa que muitas vezes era feita manualmente.

Alguns anos se passaram e, em uma viagem de avião para o OOPSLA (Object-Oriented Programming, Systems, Languages & Applications), tradicional evento sobre Orientação a Objetos, Kent Beck e Erich Gamma escreveram uma versão do SUnit em Java, o JUnit.

O JUnit foi ganhando mais e mais espaço no mundo de desenvolvimento de software, tendo vários ports feitos para outras linguagens como Ruby, C++, Perl, Python e PHP. Ao padrão da família formada por todas essas bibliotecas se deu o nome de xUnit.

Ao passo que o uso das bibliotecas xUnit foi amadurecendo, utilizá-las não era mais visto como apenas uma atividade de teste, mas sim como uma atividade de design (projeto) de código. Essa visão faz sentido ao se pensar que, antes de implementar um determinado método ou classe, os usuários de xUnit primeiro escrevem um teste especificando o comportamento esperado, e só então desenvolvem o código que vai fazer com que esse teste passe. A esse uso das bibliotecas xUnit se nomeou Test-Driven Development (TDD).

No final da década de 90, Kent Beck formalizou o Extreme Programming (XP),

que viria a ser uma das principais metodologias ágeis de desenvolvimento de software do mundo. O XP é formado por algumas práticas essenciais, entre elas o TDD. Com a evangelização de TDD dentro do XP, a prática ganhou ainda mais tração, sendo vista como uma das bases para o desenvolvimento de software com qualidade.

Entrando no mundo Ruby, a primeira biblioteca de testes automatizados na nossa linguagem foi escrita por Nathaniel Talbott, batizada com o nome Lapidary. Talbott apresentou o Lapidary na primeira RubyConf da história, em 2002. O Lapidary deu origem ao Test::Unit, tradicional biblioteca xUnit em Ruby, que usamos até hoje. Desde o começo da nossa comunidade, a prática de TDD e testes automatizados sempre foi difundida e amplamente utilizada. Sua ausência em projetos open source gera uma má impressão e é vista como um sinônimo de falta de qualidade.

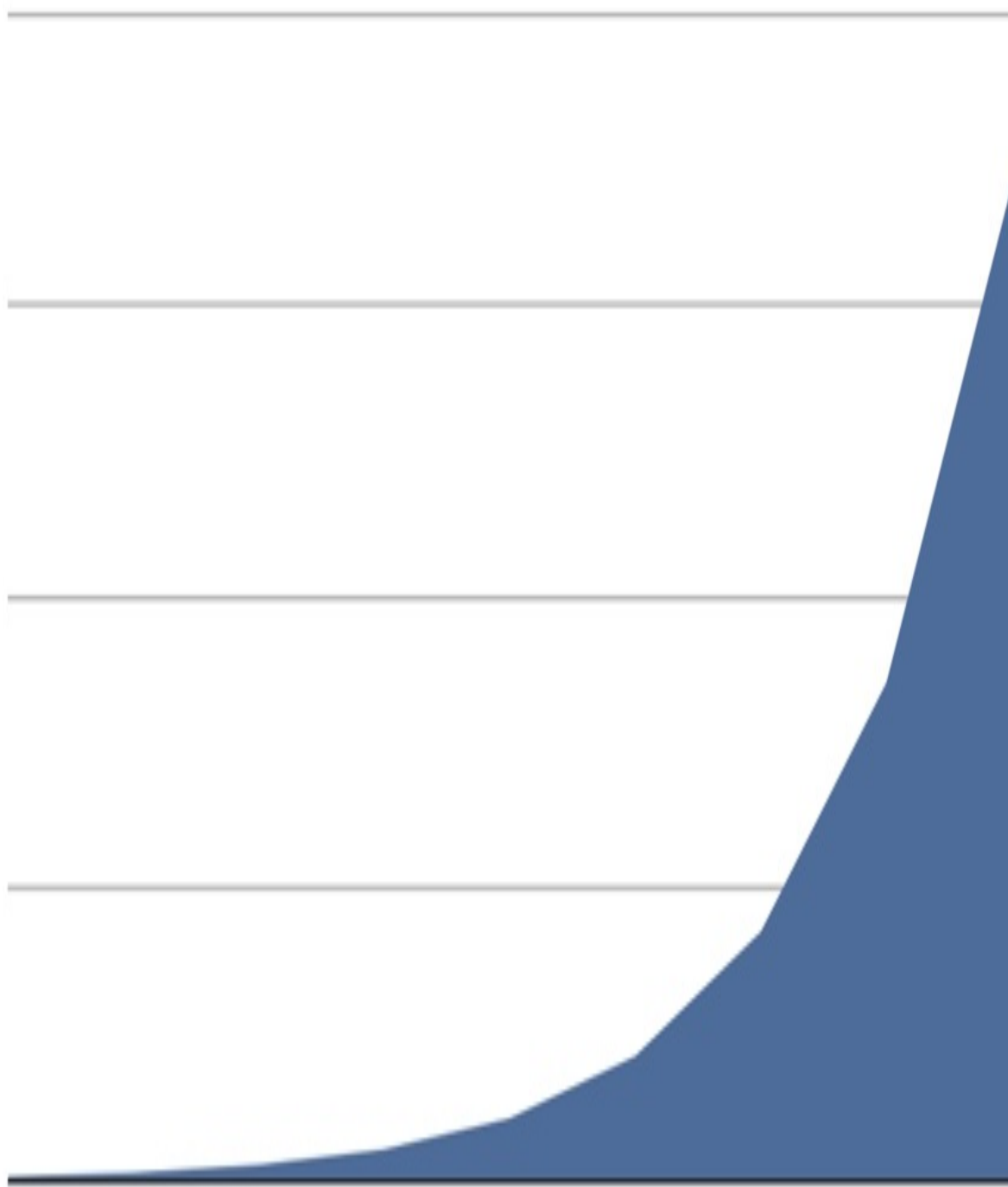
Em 2003, David Heinemeier Hansson (DHH) escreveu o Ruby on Rails (ou para os íntimos, Rails). Rails se tornou o killer app da linguagem Ruby. Com o tempo, Rails se tornou tão famoso que as pessoas começaram a aprender Ruby só porque queriam utilizar o Rails. Como um dos softwares mais famosos escritos em Ruby, o Rails também teve grande importância na evangelização de TDD na nossa comunidade. Isso porque, por padrão, todo projeto gerado pelo Rails já vem com um diretório específico para testes automatizados, ou seja, Rails o convida a fazer TDD.

Daí para a frente, o resto é história. Hoje, TDD é uma prática muito bem difundida no mundo de desenvolvimento de software, sendo vista como uma das bases para desenvolvimento de software com qualidade e de fácil manutenção.

1.2 E por qual motivo eu deveria usar TDD?

Em 1981, no livro *Software Engineering Economics* (BOEHM, 1981), Barry Boehm sugeriu que o custo de alteração em um projeto de software cresce exponencialmente à medida que se avança nas fases do desenvolvimento.

Custo de uma alteração



Requisitos

Análise

Design

Implementação

Teste

Produção

Figura 1.1: Custo de alteração em um projeto de software

Como o custo de alteração do software cresce muito ao longo das fases de desenvolvimento, seria melhor fazer a maioria das alterações necessárias logo no começo de um projeto, que seriam as fases de levantamento de requisitos e análise. Uma metodologia que segue essa ideia é a Cascata (Waterfall).

Outra abordagem que pode ser tomada em relação a esse aumento exponencial no custo de alteração é tentar reduzi-lo e mantê-lo mais constante ao longo do desenvolvimento do projeto e do ciclo de vida do software. Essa é uma das ideias de metodologias ágeis, como o XP, e uma das práticas que ajuda a diminuir esse custo é o próprio TDD.

Uma das consequências de se desenvolver utilizando TDD é que o seu sistema fica coberto por uma suíte de testes automatizados, de modo que toda vez que você for fazer uma mudança no código, é possível rodar a suíte e ela dirá se você quebrou algum comportamento previamente implementado.

Segundo o XP, o desenvolvedor deve, ao longo do projeto, refatorar constantemente o código para deixar seu design o melhor possível. Com a refatoração constante, o design do software tende a se manter bom, de forma que o custo de alteração do sistema não cresça exponencialmente conforme o tempo. Na prática, isso quer dizer que, se no começo do projeto leva-se uma semana para adicionar uma funcionalidade, um ano depois, deve-se continuar levando uma semana ou pouco mais do que isso. Essa manutenção do custo de mudança do código em um nível baixo é uma das vantagens que se ganha ao utilizar TDD.

Outra vantagem é a perda do medo de mudar o código. A possibilidade de se apoiar na suíte de testes toda vez que você for modificar o software lhe dá liberdade e coragem em mudar e adaptar o sistema de acordo com a necessidade do projeto, por toda sua vida. Esse tipo de coisa permite, por exemplo, que você faça mudanças arquiteturais no seu código, garantindo que ele continuará funcionando. Permite também que novos integrantes da equipe consigam contribuir mais rapidamente no projeto, visto que eles não precisam ter medo de mudar o código.

Por fim, outra vantagem notada pelos praticantes e estudiosos de TDD é a melhora no design do seu código. Existem vários exemplos mostrando uma relação direta entre testabilidade e bom design. A ideia geral é que, se seu código for difícil de testar, isso significa que ele pode estar acoplado demais ou com baixa coesão. TDD nos ajuda a detectar esse tipo de problema, sugerindo-nos melhorar o design do nosso código. Essa vantagem vai ficar mais clara quando formos desenvolver um projeto inteiro com TDD, a partir do capítulo BDD na prática: começando um projeto com BDD.

Agora que você já conhece a história por trás do TDD e as vantagens de usá-lo, vamos ver um exemplo simples de como é isso na prática e depois vamos falar sobre a continuação dessa história e como ela culminou no que hoje conhecemos como BDD (Behavior-Driven Development).

Capítulo 2

Primeiros passos com RSpec e Cucumber

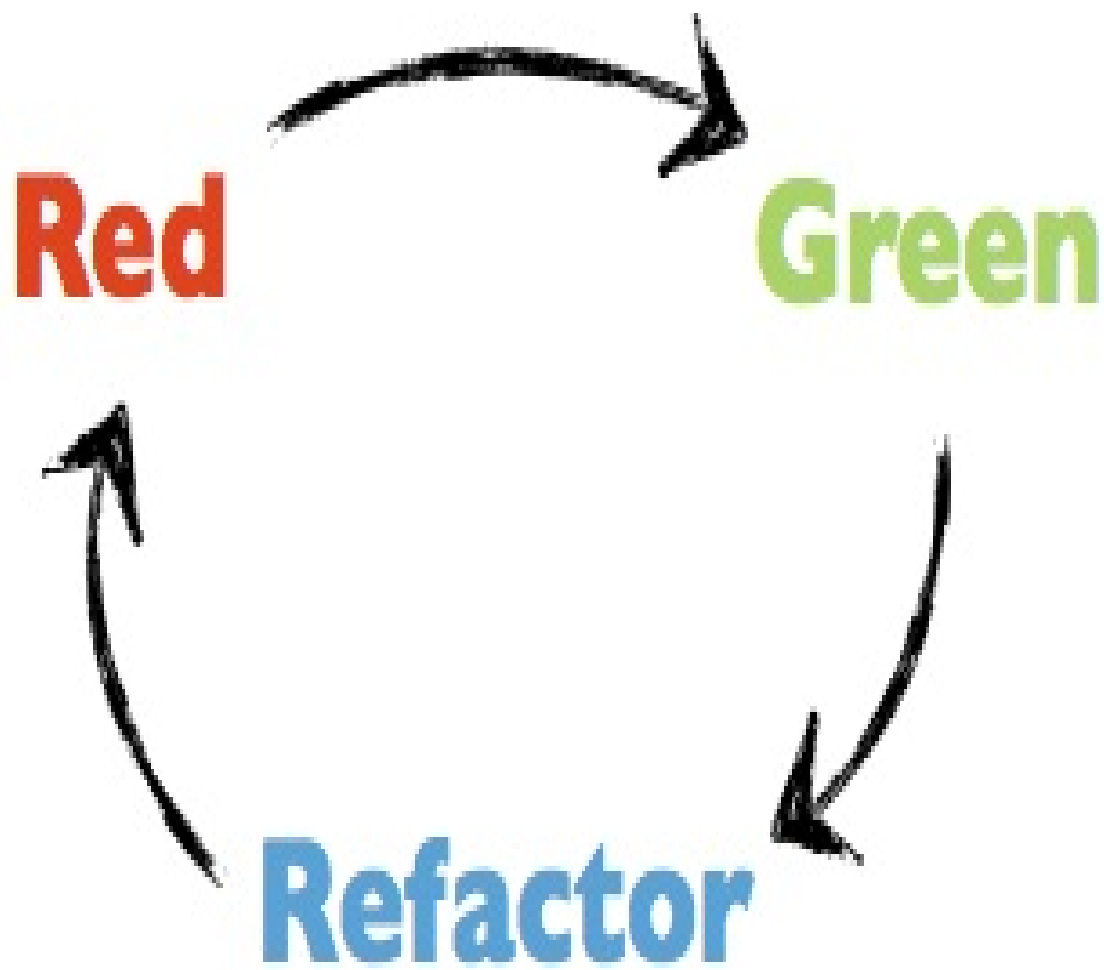
Como desenvolvedor, conheço a nossa sede por ver código. Todo mundo já ouviu a tradicional frase "Show me the code!". É por isso que, em vez de começarmos vendo os detalhes sobre RSpec e Cucumber, vamos primeiro ver uma espécie de hello world para ambas as ferramentas.

Neste primeiro momento, o mais importante é que você possa ter uma ideia da "cara" do RSpec e do Cucumber, e é isso que vamos fazer neste capítulo.

2.1 Olá RSpec

O RSpec é uma biblioteca de testes de unidade em Ruby que segue a filosofia do BDD. BDD é uma extensão do TDD — vamos falar melhor sobre ele depois. Por enquanto, nosso objetivo é ver na prática como é escrever um pequeno programa seguindo TDD e usando RSpec.

Como você viu no capítulo anterior, TDD não é considerado apenas uma prática de teste, mas sim uma prática de design. Isso se faz verdade pois, ao seguir o TDD, você pensa na API (interface) do seu software antes mesmo de construí-lo. Você descreve esse pensamento, essa especificação, em formato de teste automatizado. Tendo feito o teste, você desenvolve o pedaço de software que acabou de especificar, fazendo com que seu teste passe. Uma vez que o teste passou e está minimamente atendido, você fica livre para refatorar o seu código, reduzindo duplicação, deixando-o mais claro e fazendo outras melhorias que julgar necessárias. A esse fluxo do TDD se dá o nome de red — green — refactor. Vamos aplicá-lo para ficar mais claro.



Red: escreva um teste que quebra

Green: faça o teste passar

Refactor: melhore o seu código

Figura 2.1: Ciclo red - green - refactor

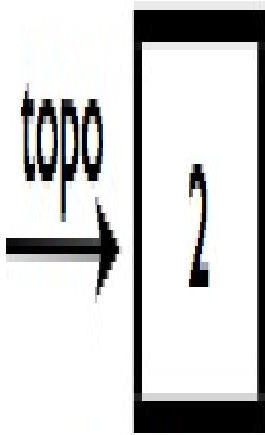
O primeiro teste com RSpec

Na minha aula de Estrutura de Dados da Poli, aprendi a implementar uma pilha. Naquele tempo, eu não conhecia TDD, então fiz na raça, sem testes (e em C). Hoje, olhando para trás, fica a curiosidade: "como seria implementar aquela pilha, só que com testes?" Hora de matar essa curiosidade!

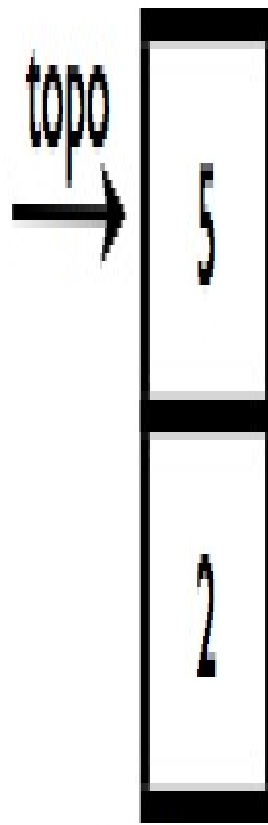
Uma pilha (Stack) é uma estrutura de dados LIFO (last in, first out), ou seja, é uma coleção de elementos na qual o último que você colocou é o primeiro a sair. Vamos especificar esse comportamento em formato de teste.

Não se importe por enquanto com a sintaxe do RSpec, o código será simples o bastante para que você possa entendê-lo mesmo sem nunca ter visto RSpec.

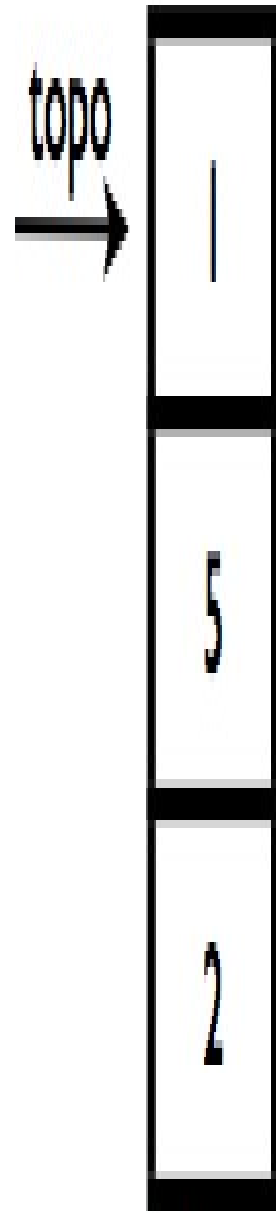
Vamos começar especificando o método push (empilha). Quando você empilhar um elemento, esse elemento deve poder ser visualizado no topo da pilha.



`stack.push(2)`



`stack.push(5)`



`stack.push(1)`

Figura 2.2: Pilha com elementos sendo empilhados

Vamos criar um arquivo chamado `stack_spec.rb` com o seguinte código, que especificará esse comportamento:

```
RSpec.describe Stack do

  describe
    "#push" do

      it
        "puts an element at the top of the stack" do

          stack =
            Stack
              .new

          stack.push(
```

```
)  
    stack.push(  
2  
)  
  
    expect(stack.top).to eq(  
2  
)  
  
end  
  
end end
```

Novamente, não se assuste com a sintaxe. Atente-se apenas ao fato de que estamos chamando o método `push` para adicionar primeiro o elemento 1 e, em seguida, o elemento 2 na pilha. Por fim, indicamos que o topo da pilha deve conter o elemento 2, seguindo a ideia de last in, first out.

Esse é um teste escrito com RSpec e, para rodá-lo, precisamos antes instalá-lo, através de sua gem. Para fazer a instalação, execute o seguinte comando no seu console:

```
$ gem install rspec
```

Com ele instalado, para rodar seus testes basta executar no seu console:

```
$ rspec stack_spec.rb
```

Ao rodar o RSpec, o teste quebra, como esperado. Esse é o passo red, do ciclo red — green — refactor. Ele quebrou porque até agora só implementamos o teste, não implementamos nenhum código que o faça passar. Nesse momento, o importante é ver que o teste quebrou e por que ele quebrou. Após ter rodado o RSpec, você deve ter visto uma mensagem de erro contendo a seguinte saída:

```
$ rspec stack_spec.rb
```

```
(...)
```

NameError:

```
uninitialized constant Stack
```

A mensagem de erro nos diz que o teste quebrou porque a constante Stack ainda não foi definida. Ou seja, precisamos criar a classe Stack. Para simplificar, escreva a definição da classe no mesmo arquivo stack_spec.rb, de modo que ele fique assim:

```
class Stack end
```

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      it
```

```
        "puts an element at the top of the stack" do
```

```
          stack =
```

```
            Stack
```

```
              .new
```

```
            stack.push(
```

```
              1
```

```
            )
```

```
            stack.push(
```

```
              2
```

```
            )
```

```
    expect(stack.top).to eq(  
2  
)
```

```
end
```

```
end end
```

Ao rodar o RSpec agora, vemos o seguinte:

```
$ rspec stack_spec.rb  
(...)
```

Failures:

1) Stack#push puts an element at the top of the stack

Failure/Error: stack.push(1)

NoMethodError:

undefined method `push' for #<Stack:0x007fa91b919b70>

Stack

#push

puts an element at the top of the stack (FAILED - 1)

Failures:

1) Stack#push puts an element at the top of the stack

Failure/Error: stack.push(1)

NoMethodError:

undefined method 'push' for #<Stack:0x00007fc9559bdf68>

./stack_spec.rb:20:in 'block (3 levels) in <top (required)>'

Finished in 0.0024 seconds (files took 0.25717 seconds to load)

1 example, 1 failure

Figura 2.3: Teste no vermelho

Evoluímos, mas, mesmo assim, o teste quebrou e a mensagem de erro agora nos diz que foi porque o método push ainda não está definido. Precisamos implementá-lo.

Para implementar o método push, precisamos pensar no que ele vai fazer. Temos que colocar um elemento dentro da pilha, de modo que o último item colocado é o que ficará no topo da pilha. Como mecanismo para guardar os elementos, podemos usar um Array e ir colocando os elementos dentro dele. Faça isso editando a classe Stack do seguinte modo:

```
class Stack
```

```
  def
```

```
    initialize
```

```
      @elements
```

```
      = []
```

```
    end
```

```
def
```

```
  push(element)
```

```
  @elements
```

```
  << element
```

```
end end
```

```
# (...)
```

Com o método push implementado, vamos rodar o RSpec novamente:

```
$ rspec stack_spec.rb
```

```
(...)
```

Failures:

1) Stack#push puts an element at the top of the stack

Failure/Error: expect(stack.top).to eq(2)

NoMethodError:

undefined method `top' for #<Stack:0x007fce55c55e38>

Evoluímos mais um pouco e o teste quebrou novamente. Não desanime! Dessa vez, a mensagem de erro foi diferente, o que quer dizer que estamos progredindo. Agora ela nos diz que o motivo da falha é que o método top ainda não foi implementado.

O que o método top precisa fazer é apenas retornar o último elemento que foi empilhado. Para sabermos qual o último elemento adicionado na nossa pilha, vamos criar uma variável de instância chamada `@last_element_index` para salvar o índice do último elemento empilhado. Modifique o código para seguir essa ideia e ficar assim:

```
class Stack
```

```
  def
```

```
    initialize
```

```
      @elements
```

```
= []
```

```
@last_element_index = -1
```

```
end
```

```
def
```

```
push(element)
```

```
@elements
```

```
<< element
```

```
@last_element_index += 1
```

```
end
```

```
def
```

```
top
```

```
@elements[@last_element_index
```

```
]
```

```
end end
```

Ao rodarmos o RSpec, dessa vez vemos o seguinte:

```
$ rspec --format documentation stack_spec.rb
```

```
Stack
```

```
#push
```

```
puts an element at the top of the stack
```

Finished in 0.00431 seconds (files took 0.2455 seconds to load)

1 example, 0 failures

Stack

#push

puts an element at the top of the stack

Finished in 0.00311 seconds

1 example, 0 failures

Figura 2.4: Teste no verde

Agora o teste passou! Chegamos ao passo green do ciclo red — green — refactor. Agora que os testes estão no verde, podemos refatorar os testes ou o código da pilha em si. Refatorar é melhorar o código sem mudar seu comportamento externo, não adicionando nenhum comportamento novo, apenas melhorando a qualidade interna do nosso software.

Note que dessa vez usamos a opção `--format documentation`. Ela é responsável por apresentar os exemplos do RSpec de forma estruturada:

(...)

Stack

`#push`

puts an element at the top of the stack

Um ponto que podemos melhorar no nosso código é o modo como estamos pegando o último elemento salvo na pilha. Em vez de criar uma variável de instância para pegar o último elemento do array interno, podemos simplesmente fazer `@elements.last`. Seguindo essa ideia, modifique o código para ficar assim:

```
class Stack
```

```
  def
```

```
    initialize
```

```
      @elements
```

```
    = []
```

```
  end
```

```
  def
```

```
    push(element)
```

```
      @elements
```

```
    << element
```

```
  end
```


def

top

@elements

.last

end end

Agora, rode o RSpec novamente para checar se o teste está passando e que o comportamento previamente especificado continua funcionando:

```
$ rspec --format documentation stack_spec.rb
```

Stack

#push

puts an element at the top of the stack

Finished in 0.0044 seconds (files took 0.24301 seconds to load)

1 example, 0 failures

Muito bom, o teste continua no verde! Isso quer dizer que conseguimos refatorar o nosso código com sucesso. Melhoramos a qualidade interna sem quebrar nada que já estava funcionando.

O passo seguinte seria escrever o próximo teste da classe Stack, talvez para o método top, ou para um possível método pop (desempilha), seguindo o ciclo red — green — refactor novamente. Esses eu deixarei como um desafio para você: apenas replique o padrão que foi mostrado.

No capítulo seguinte, vamos entrar em detalhes no RSpec, entender sua sintaxe e suas nuances. A partir do capítulo BDD na prática: começando um projeto com BDD vamos construir uma aplicação inteira fazendo TDD, usando RSpec do começo ao fim. Lá você terá bastante tempo e exemplos para exercitar o ciclo red — green — refactor e aprender como se constrói um software inteiro usando TDD.

Dando uma última olhada no código que escrevemos para fazer o teste com o RSpec, dá para perceber que fizemos a descrição de um comportamento do nosso código, através das seguintes linhas:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

it

"puts an element at the top of the stack" do

Dissemos que estávamos "Descrevendo" (Describe) a classe Stack, em seguida, o seu método #push e, por fim, falamos que ele coloca o elemento no topo da pilha. Mas repare que misturamos código com o texto. Será que não há uma alternativa de descrever nossa aplicação como um todo, apenas como um texto, um roteiro, e fazer com que isso execute? Não seria mais natural?

■

Escrevo em português ou em inglês?

É muito comum se perguntar se a descrição dos testes deve ser feita em inglês ou em português. Pelo fato de o RSpec se basear no inglês, fica mais natural manter tudo no mesmo idioma, além de que a legibilidade fica maior quando se entende o idioma. Mas nada o impede fazer as descrições em português. Durante o livro, usaremos as descrições em inglês, mas daremos sempre as explicações em português.

■

2.2 Olá Cucumber

O Cucumber é uma ferramenta de testes de aceitação, isto é, serve para automatizar testes do comportamento do software levando em conta o ponto de vista do usuário final.

Ao escrever um teste com Cucumber, você vai especificar uma funcionalidade do seu sistema, em vez de uma classe ou um método. De modo bem resumido, entenda o Cucumber como uma espécie de documentação de requisitos funcionais on steroids.

A proposta do Cucumber é ajudar a especificar o sistema com documentos escritos em uma linguagem natural (inglês, português etc.), porém com a possibilidade de executar essa documentação para verificar o comportamento especificado no sistema real. Ou seja, uma documentação executável.

O primeiro teste com Cucumber

Para ficar mais claro, vamos ver um exemplo de uma especificação escrita com Cucumber.

Imagine que você foi contratado para desenvolver um pequeno jogo da forca, que vai funcionar no shell. Você conversa com o cliente, ele lhe explica o motivo

de querer o jogo e como ele pensa que vai funcionar.

Você entende o contexto inteiro e decide documentar a primeira funcionalidade para pedir ao cliente conferir se o seu entendimento está correto. A primeira funcionalidade que você imagina é a de "Começar jogo", e você a documenta do seguinte modo:

language: pt

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

|||||

Bem-vindo ao jogo da forca!

|||||

Esse é um exemplo real de especificação de software com Cucumber. A spec tem o título da funcionalidade, uma narrativa explicando o contexto e, por fim, um cenário de teste, mostrando um exemplo de como o software vai funcionar do ponto de vista do usuário final.

■

Cenários em português

Diferentemente de quando escrevemos as especificações com o RSpec, em que misturávamos código e texto, com o Cucumber escrevemos apenas texto para descrever os cenários. Isso remove qualquer ruído que a linguagem de programação pudesse causar na descrição e, dessa forma, faz com que escrever em linguagem natural seja algo bem simples e direto. Durante o livro, para os cenários em Cucumber, usaremos sempre o português.

■

Perceba como a especificação é bem simples e clara. O cliente lê a spec e confirma que o entendimento está correto. Você pode agora ir em frente e desenvolver essa funcionalidade. Uma das vantagens de utilizar o Cucumber é que você pode usar essa mesma documentação para criar um cenário de teste de aceitação automatizado. Vamos fazer isso.

Antes de tudo, precisamos instalar o Cucumber. Como ele também é distribuído

como uma gem, basta executar no seu console:

```
$ gem install cucumber
```

Em seguida, crie um novo diretório chamado hello-cucumber e, dentro dele, execute o comando init do Cucumber, que criará os diretórios padrões utilizados pela gem:

```
$ mkdir hello-cucumber
```

```
$ cd hello-cucumber
```

```
$ cucumber --init
```

Dentro do diretório features, crie um arquivo chamado `comecar_jogo.feature` e salve dentro dele o conteúdo da especificação que fizemos há pouco. Agora você já pode rodar o Cucumber.

Para fazer isso, dentro do diretório hello-cucumber, execute o comando cucumber no seu console:

```
$ cucumber
```

Isso fará com que os cenários sejam executados e a seguinte saída apareça como resultado:

language: pt

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

Bem-vindo ao jogo da forca!

1 scenario (1 undefined)

2 steps (2 undefined)

0m0.060s

You can implement step definitions for undefined steps
with these snippets:

```
Quando("começo um novo jogo") do  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

```
Então("vejo a seguinte mensagem na tela:") do |string|  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```


Vamos analisar um pouco a saída do Cucumber. Logo depois do texto da especificação em si, vemos o seguinte:

1 scenario (1 undefined)

2 steps (2 undefined)

Perceba que ele diz que temos um cenário que está indefinido e que existem dois steps também indefinidos. Em specs de Cucumber, cada caso de teste é um cenário e cada cenário é composto por vários steps. No nosso caso, temos o

cenário Começo de novo jogo com sucesso que contém os steps Quando começo um novo jogo e Então vejo a seguinte mensagem na tela: (...).

Funcionalidade: 

Cenário: 

Step a
Step b
Step c
Step d

Cenário: 

Step a
Step b
Step c
Step d

Figura 2.5: Estrutura de uma especificação com Cucumber

Para utilizar essa documentação como um teste automatizado, temos que definir cada um desses steps, implementando o que o Cucumber chama de step definitions. Um step definition mapeia que código deve ser executado para um dado step. O legal é que, se o step ainda está indefinido, o próprio Cucumber já nos dá um snippet de step definition, como pudemos ver no final da saída:

```
$ cucumber
```

```
(...)
```

You can implement step definitions for undefined steps
with these snippets:

```
Quando("começo um novo jogo") do  
  pending # Write code here that turns the phrase above  
           # into concrete actions  
end
```

```
Então("vejo a seguinte mensagem na tela:") do |string|
```

```
pending # Write code here that turns the phrase above  
  # into concrete actions  
end
```

Vamos agora automatizar nossa especificação. Primeiro, crie um arquivo chamado `game_steps.rb` dentro do diretório `features/step_definitions`. Nele, vamos colocar os step definitions dos dois steps que temos na nossa spec. Implemente esses steps do seguinte modo nesse arquivo:

```
Quando("começo um novo jogo") do
```

```
  @game = Game
```

```
  .new
```

```
  @game.start end
```

```
Então("vejo a seguinte mensagem na tela:") do
```

```
  |text|
```

```
    expect(
```

```
      @game.output).to include(text) end
```

O que fizemos foi implementar os step definitions, definindo quais partes da API do nosso software cada step definition vai utilizar.

Para o step de começar um novo jogo, imaginamos que poderíamos ter um objeto `@game` e que para começar um novo jogo bastaria chamar o método `@game.start`.

No segundo step definition, estamos verificando se o output do `@game` contém a mensagem inicial do jogo, que está salva na variável `text`.

Especificações de Cucumber devem testar o sistema utilizando a mesma interface que o usuário utilizaria. Apesar de o nosso jogo ser uma aplicação com interface de usuário pelo console, manteremos o teste bem simples nesse "hello world", sem fazer uma interação real do teste com o console. Vamos fazer esse tipo de teste de modo completo e detalhado a partir do capítulo BDD na prática: começando um projeto com BDD.

Com os step definitions prontos, vamos rodar o Cucumber novamente:

```
$ cucumber
```

```
(...)
```

```
uninitialized constant Game (NameError)
```

(...)

1 scenario (1 failed)

2 steps (1 failed, 1 skipped)

Perceba que o teste quebrou e que nos deu uma mensagem de erro. A mensagem de erro do teste é o feedback que devemos usar para o próximo passo de implementação. Nesse caso, a mensagem está dizendo que a classe Game ainda não foi definida.

O correto, nesse momento, seria parar um pouco os testes de Cucumber e começar um ciclo completo de red — green — refactor com RSpec para a definição da classe Game, mas para manter o exemplo simples, não faremos isso agora.

Por enquanto, vamos apenas definir a classe Game dentro do arquivo `features/step_definitions/game_steps.rb` mesmo. Adicione a definição dessa classe no final desse arquivo:

Quando("começo um novo jogo") do

`@game = Game`

`.new`

```
@game.start end
```

```
Então("vejo a seguinte mensagem na tela:") do
```

```
|text|
```

```
  expect(
```

```
@game.output).to include(text) end
```

```
class Game end
```

Ao rodar o Cucumber novamente, vemos uma mensagem de erro diferente:

```
$ cucumber
```

```
(...)
```

```
undefined method `start' for #<Game:0x00007fa6c4> (NoMethodError)
```

```
(...)
```

Dessa vez, o teste quebrou porque o método `Game#start` ainda não foi definido,

vamos implementá-lo. Ele deve salvar dentro do output do jogo a mensagem inicial. Adicione esse método à classe Game do seguinte modo:

```
# (...)
```

```
class Game
```

```
def
```

```
  initialize
```

```
    @output
```

```
    = []
```

```
  end
```

```
def
```

```
  start
```

```
@output << "Bem-vindo ao jogo da forca!"
```

```
end end
```

Ao rodarmos o Cucumber, podemos ver que a mensagem de erro mudou:

```
$ cucumber
```

```
(...)
```

```
undefined method `output' for #<Game:0x00007fc5c> (NoMethodError)
```

```
(...)
```

Pela mensagem de erro, podemos ver que falta implementar o método `Game#output`, responsável por devolver o conteúdo da variável `@output`. Para implementá-lo, basta adicionar um `attr_reader :output` na classe `Game`:

```
# (...)
```

```
class Game
```

```
attr_reader :output
```

```
# (...) end
```

Ao rodar o Cucumber novamente, vemos que agora ele está no verde!

```
$ cucumber
```

```
(...)
```

```
1 scenario (1 passed)
```

```
2 steps (2 passed)
```

language: pt

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

"""

Bem-vindo ao jogo da forca!

"""

1 scenario (1 passed)

2 steps (2 passed)

0m0.066s

Figura 2.6: Spec com Cucumber no verde

Pronto, você acabou de escrever sua primeira especificação com Cucumber! A ideia era mostrar de modo rápido como é a cara do RSpec e do Cucumber, para você ter uma noção do que vem pela frente.

Repare que, durante os testes, o tempo inteiro nós descrevemos o comportamento das aplicações. Esse foco no comportamento é um grande diferencial, que, quando bem usado, pode dar uma nova vida aos nossos testes.

2.3 O que é BDD?

O termo BDD (Behavior-Driven Development) foi cunhado por Dan North por volta de 2003 (NORTH, 2006). Dan North estava tendo dificuldades em ensinar TDD para seus alunos, pois eles ficavam com dúvidas sobre por onde começar a testar, o que testar, como nomear seus testes etc.

Ele percebeu que parte do problema em entender TDD era a palavra teste em si, pois ela confundia o desenvolvedor sobre a real motivação por trás de TDD. A principal motivação do TDD não é testar o seu software, e sim especificá-lo com exemplos de como usar o seu código e deixar isso guiar o design.

Ao fazer TDD, o desenvolvedor está especificando o comportamento do seu software e definindo seu design. A suíte de testes automatizados gerada através do TDD é "apenas" uma boa consequência do processo. Baseado nessas ideias, Dan North sugeriu mudar a palavra de teste para comportamento (behavior). Para concretizar sua ideia, Dan North escreveu o JBehave, uma biblioteca de BDD em Java.

No mundo de Ruby, a ideia de BDD começou a se concretizar quando Dan North portou o JBehave para Ruby com o nome de RBehave. Em 2005, inspirado pela ideia de BDD, Steven Baker escreveu a primeira versão do RSpec. Tempos depois, o RBehave foi integrado ao RSpec com o nome de Story Runner. Finalmente, em 2008, Aslak Hellesøy reescreveu o Story Runner e o chamou de Cucumber.

Apesar de o BDD ter nascido apenas como um modo de rever a nomenclatura do TDD e o modo como se enxerga essa prática, hoje BDD é mais do que isso. Atualmente ele é uma abordagem de desenvolvimento de software (CHELIMSKY; ASTELS; DENNIS, et al., 2009) que propõe que você desenvolva seu software especificando as várias camadas de comportamento com testes automatizados seguindo uma abordagem de outside-in development. Ou seja, comece primeiro especificando o comportamento de uma funcionalidade. Depois disso, especifique as classes e métodos que serão necessários para implementá-la.

À prática de começar o desenvolvimento de uma funcionalidade com testes de aceitação e, depois fazer os testes de unidade, dá-se o nome de Acceptance Test-Driven Development (ATDD). Essa é uma das práticas nas quais o BDD se baseia. Os testes devem ser escritos usando termos do domínio de negócio, uma aplicação da linguagem ubíqua (ubiquitous language), conceito criado por Eric Evans no livro Domain-Driven Design (EVANS, 2004).

Pontos-chaves deste capítulo

Neste capítulo vimos que o TDD é uma prática de design, que nos ajuda a pensar na API (interface) da sua aplicação, mesmo antes de o código existir de fato.

Aprendemos também o fluxo red — green — refactor, onde começamos por um teste quebrado, fazemos os testes passarem com o código mais simples possível, e então refatoramos para melhorar a qualidade deste código. Para exemplificar esse fluxo, implementamos uma pilha e testamos com RSpec.

Em seguida, aprendemos que o Cucumber é uma documentação executável do seu software. Ele descreve o comportamento do ponto de vista do usuário da

aplicação, e é composto por descrições das funcionalidades escritas em linguagem natural (ex. português, inglês). Essas descrições são mapeadas para o código que será executado através de step definitions.

Por fim, vimos que o BDD (Behavior-Driven Development) foi criado por Dan North, que entendeu que o TDD se propõe a especificar o comportamento (behavior) do software. Posteriormente, o BDD passou a ser definido como uma abordagem de desenvolvimento de software, incorporando também a prática do outside-in development.

A partir do capítulo BDD na prática: começando um projeto com BDD nós vamos desenvolver uma aplicação seguindo o fluxo de outside-in development. Começaremos por testes de aceitação usando Cucumber e, em seguida, faremos testes de unidade usando RSpec. Mas antes de começarmos a desenvolver o projeto, você precisa aprender mais sobre o RSpec e o Cucumber, e é isso que faremos nos capítulos a seguir.

Capítulo 3

Conhecendo o RSpec

Neste capítulo você vai aprender como usar o RSpec para fazer testes de unidade. Veremos como o RSpec pode nos ajudar a escrever testes que sejam legíveis e que possam servir não só como testes de regressão, mas como mais uma fonte de documentação do nosso código.

3.1 Aprendendo a estrutura básica de um teste com RSpec

Para começarmos a entender a estrutura de um teste com RSpec, vamos dar uma olhada no que fizemos no capítulo anterior:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      it
```

```
        "puts an element at the top of the stack" do
```

```
          stack =
```

```
            Stack
```

```
              .new
```

```
            stack.push(
```

```
1
)
  stack.push(
2
)

  expect(stack.top).to eq(
2
)

end

end end
```

■

RSpec.describe ou describe?

Você deve ter notado que o primeiro describe possui o prefixo RSpec., porém, o segundo, não. O exemplo anterior funcionaria mesmo sem o módulo RSpec:

```
describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      # ...
```

```
    end end
```

O problema de não utilizar RSpec.describe no namespace global (a primeira chamada do arquivo) é que estamos utilizando uma versão do método que está disponível globalmente. Isso pode causar conflitos caso você (ou uma gem) também defina um método global chamado describe, então o recomendado é sempre prefixar a chamada do método describe (ou dos demais métodos do RSpec, como o `shared_examples_for` que veremos mais para a frente) com o módulo RSpec quando estiver no namespace global.

■

A primeira coisa que estamos fazendo nesse teste é chamar o método describe:

```
RSpec.describe Stack do
```

```
# (...) end
```

O método `describe` serve para agrupar os nossos testes. No que acabamos de fazer, usamos um `describe` para agrupar as specs da classe `Stack` e um `describe` aninhado para agrupar especificamente as specs do método `Stack#push`. O modo como estamos usando o `describe` aninhado para agrupar os testes relacionados somente ao comportamento do método `Stack#push` é apenas uma convenção — poderíamos ter feito sem usar um `describe` aninhado:

```
RSpec.describe Stack do
```

```
  it
```

```
    "puts an element at the top of the stack" do
```

```
      stack =
```

```
        Stack
```

```
        .new
```

```
      stack.push(
```

```
        1
```

```
      )
```

```
      stack.push(
```

2

)

expect(stack.top).to eq(

2

)

end end

Mais adiante falaremos mais dessa convenção. Agora vamos entender para que serve o método it.

O método it serve para criarmos um teste de fato. Dentro do bloco que passamos para o it é onde é escrito um exemplo de como o nosso código deve se comportar em um determinado contexto. No exemplo que fizemos agora, temos o seguinte teste:

it "puts an element at the top of the stack" do

stack =

Stack

.new

```
stack.push(  
1  
)  
  
stack.push(  
2  
)  
  
expect(stack.top).to eq(  
2) end
```

Perceba como o código escrito é apenas uma chamada do método `it`. Ele está sendo chamado com dois argumentos. O primeiro é a string que contém uma descrição do teste. O segundo é um bloco com o código do teste em si, delimitado pelo `do` e `end`.

A terceira, e última parte básica de um teste com RSpec é a asserção, que é como verificamos se o nosso código se comportou do modo esperado. No RSpec as asserções são feitas com `expectations`. Nesse teste, a `expectation` que estamos fazendo é a parte onde checamos se o topo da pilha contém o elemento 2:

```
expect(stack.top).to eq(2)
```


A estrutura de uma expectation segue a seguinte forma:

```
expect(actual).to matcher(expected)
```

ou:

```
# usando to_not
```

```
expect(actual).to_not matcher(expected)
```

```
# usando not_to
```

```
expect(actual).not_to matcher(expected)
```

actual é o objeto cujo comportamento você está testando, expect é um método do RSpec para iniciar uma expectation e matcher é um objeto que segue o protocolo do RSpec::Matchers e serve para testar de fato o comportamento esperado do seu objeto.

No caso do nosso teste anterior, o matcher usado foi o eq, que testa se o seu objeto é igual ao valor esperado usando o operador ==. Ou seja, o que fizemos no nosso teste foi verificar se o valor retornado pelo método stack.top era igual ao valor esperado, 2.

Visto tudo isso, agora você já conhece a estrutura básica de um teste com RSpec. Use o `describe` para agrupar os seus testes de modo lógico. Use o `it` para escrever um teste em si. E, por fim, use os `RSpec::Matchers` para verificar o comportamento esperado dentro do teste.

Agora vamos dar uma olhada nos matchers que vêm por padrão com o RSpec.

3.2 Por que existem tantos matchers no RSpec

O RSpec vem com muitos matchers por padrão. Antes de começarmos a vê-los, vale a pena entendermos uma das vantagens em ter tantos assim. Para isso, vamos criar um pequeno projeto e fazer um teste para ele.

Vamos desenvolver um objeto que sirva como um saco de palavras, dentro do qual você pode colocar palavras e depois verificar quantas palavras há dentro do saco. Comece criando um diretório para o nosso projeto, chamado de `bag_of_words` e entre nesse diretório:

```
$ mkdir bag_of_words
```

```
$ cd bag_of_words
```

Crie um Gemfile usando o Bundler:

```
$ bundle init
```

Coloque o RSpec no Gemfile:

```
# frozen_string_literal: true
```

```
source
```

```
"https://rubygems.org"
```

```
git_source(
```

```
:github) do
```

```
|repo_name|
```

```
"https://github.com/#{repo_name}" end
```

```
gem
```

```
"rspec"
```

E finalmente instale o RSpec usando o Bundler:

```
$ bundle install
```

■

Sobre o Bundler

Caso você ainda não conheça o Bundler, ele é um projeto muito famoso na comunidade Ruby, feito para gerenciar as dependências do seu projeto.

Para fazer isso, você deve criar um arquivo chamado Gemfile e listar nele as gems com suas versões das quais o seu projeto depende. Baseado nesse arquivo Gemfile, o Bundler instala as gems necessárias, resolvendo as versões delas de acordo com o que você configurou.

Visto que o Bundler é distribuído como uma gem, instalá-lo é bem fácil. Basta utilizar o RubyGems, fazendo o seguinte comando no console:

```
$ gem install bundler
```

■

Com o RSpec instalado, vamos criar a estrutura de diretórios do nosso projeto, começando pelo diretório lib:

```
$ mkdir lib
```

E agora, crie a estrutura de diretório e arquivos padrão do RSpec usando o seguinte comando:

```
$ rspec --init
```

```
create .rspec
```

```
create spec/spec_helper.rb
```

Perceba que, ao rodar o comando `rspec --init`, ele criou dois arquivos e um diretório. O diretório `spec` é onde devemos colocar nossos testes. O arquivo `spec/spec_helper.rb` é onde são colocadas as configurações comuns entre todos nossos testes. Finalmente, no arquivo `.rspec` é onde podemos deixar salvas as flags que serão passadas para o comando `rspec` quando formos rodar nossos testes. Com a estrutura de diretórios pronta, vamos começar a desenvolver nosso pequeno projeto.

Comece criando um arquivo de teste chamado `spec/bag_of_words_spec.rb`:

```
$ touch spec/bag_of_words_spec.rb
```

Vamos fazer um teste para especificar que é possível colocar palavras no nosso saco de palavras. Escreva o seguinte teste no arquivo `spec/bag_of_words_spec.rb`:

```
require "bag_of_words"
```

```
RSpec.describe BagOfWords do
```

```
  it
```

```
    "is possible to put words on it" do
```

```
      bag =
```

```
        BagOfWords
```

```
        .new
```

```
        bag.put(
```

```
          "hello", "world"
```

```
        )
```

```
        expect(bag.words.size).to eq(
```

```
          2
```

```
        )
```

```
    end end
```

Vamos entender o que fizemos nesse teste. Primeiro, demos require do arquivo `bag_of_words`, que ainda nem existe, mas é onde colocaremos o código da nossa classe.

■

`require "spec_helper"`

É comum encontrar o comando `require "spec_helper"` nos arquivos de teste em projetos que utilizam RSpec. Até a versão 3 do RSpec, essa era uma prática necessária para carregar as configurações contidas no `spec_helper`. A partir da versão 3, ao executar o comando `rspec --init` o arquivo `.rspec` gerado possui a seguinte linha:

`--require spec_helper`

Isso garante que o arquivo `spec_helper` seja carregado toda vez que o RSpec é executado, de forma que não é mais necessário adicionar `require "spec_helper"` no início dos arquivos.

■

Depois, no teste em si, nós simplesmente criamos um objeto `bag`, colocamos duas palavras dentro dele e testamos se o tamanho da lista retornada por `bag.words.size` é igual a 2.

Rode o teste e verifique que ele está quebrando:


```
$ bundle exec rspec --format documentation
```

(...)

LoadError:

```
cannot load such file -- bag_of_words
```

O teste quebra com uma mensagem que nos fala que o problema é que não foi possível dar require do arquivo bag_of_words. Claro, ainda não o criamos. Vamos dar o primeiro passo na tentativa de fazer o teste passar, criando esse arquivo com uma classe BagOfWords e com o método BagOfWords#put. Para isso, crie o arquivo lib/bag_of_words.rb e escreva o seguinte código nele:

```
class BagOfWords
```

```
  attr_reader :words
```

```
  def
```

```
    initialize
```

```
@words
```

```
= []
```

```
end
```

```
def
```

```
  put(*words)
```

```
# TODO
```

```
end end
```

Com essa implementação feita, com basicamente um initialize e um método put que ainda vamos implementar, podemos rodar os testes novamente e ver o feedback para o próximo passo:

```
$ bundle exec rspec --format documentation
```

BagOfWords

is possible to put words on it (FAILED - 1)

Failures:

1) BagOfWords is possible to put words on it

Failure/Error: expect(bag.words.size).to eq(2)

expected: 2

got: 0

(compared using ==)

Nosso teste quebrou com a seguinte mensagem:

Failure/Error: expect(bag.words.size).to eq(2)

expected: 2

got: 0

O próximo passo é fazer o teste passar. Basta fazermos com que o método `BagOfWords#put` concatene os seus argumentos no array de `@words` interno. Modifique esse método no arquivo `lib/bag_of_words.rb` para ficar assim:

```
class BagOfWords
```

```
  attr_reader :words
```

```
  def
```

```
    initialize
```

```
      @words
```

```
    = []
```

```
  end
```

```
def
```

```
  put(*words)
```

```
@words
```

```
  += words
```

```
end end
```

Ao rodar os testes, vemos que eles passam:

```
$ bundle exec rspec --format documentation
```

```
BagOfWords
```

```
  is possible to put words on it
```

```
Finished in 0.00324 seconds (files took 0.22571 seconds to load)
```

```
1 example, 0 failures
```

Curiosidade: a manipulação do \$LOAD_PATH feita pelo RSpec

Para entender essa curiosidade, você deve saber que, em Ruby, ao se fazer `require "file_path"`, o Ruby vai procurar o arquivo `file_path` dentro da variável global `$LOAD_PATH` e vai carregá-lo se achar.

No teste que fizemos, você deve se lembrar que `demos` `require` do arquivo `bag_of_words`:

```
# spec/bag_of_words_spec.rb
```

```
require "bag_of_words"
```

```
RSpec.describe BagOfWords do
```

```
# (...) end
```

Ao ler o código de testes de outros projetos que utilizam RSpec, é possível que você veja o diretório `lib/` sendo inserido explicitamente no `$LOAD_PATH` para que se possa dar `require` em um arquivo desse diretório. Um exemplo dessa prática poderia ser assim:

```
$LOAD_PATH.prepend File.join(__dir__, "../lib") require "bag_of_words"
```

Há tempos atrás eu me perguntei por que algumas pessoas ficavam colocando o diretório lib/ explicitamente dentro do \$LOAD_PATH e outras não. Acontece que no RSpec é desnecessário fazer isso, porque ele já faz isso por nós. Isso é feito pelo rspec-core (<https://github.com/rspec/rspec-core/blob/4a29a4b13d66fa96fbdf439a5c21ca9816eb1a3/lib/rspec/core/configuration.rb#L1521>), no arquivo lib/rspec/core/configuration.rb:

```
def
  requires=(paths)

  directories =

  [
    "lib", default_path].select { |p| File
    .directory? p }
```

```
RSpec::Core::RubyProject
  .add_to_load_path(*directories)
```

```
# ... end
```

Portanto, não esqueça, não é necessário colocar o diretório lib/ e spec/ explicitamente no \$LOAD_PATH, o RSpec já faz isso por nós.

3.3 Conhecendo os RSpec built-in matchers

Um matcher no RSpec é um objeto que serve para verificar o comportamento esperado no nosso teste. Ele é usado para montar uma expectation do RSpec de dois modos diferentes:

```
expect(actual).to matcher(expected)
```

```
expect(actual).to_not matcher(expected)
```

Um exemplo para cada um dos modos de expectation pode ser:

```
expect(1).to eq(1
```

```
)
```

```
expect(
```

```
1).to_not eq(2)
```

O RSpec já acompanha vários matchers (built-in matchers) para nos ajudar a escrever testes expressivos. Vamos dar uma olhada neles, começando pelos mais básicos, os "be matchers".

Matchers relacionados a truthy e falsy

Os be matchers servem para você testar se um objeto é avaliado como true ou false. Você pode usá-los do seguinte modo:

passa se obj é truthy (diferente de nil e false)

```
expect(obj).to be_truthy
```

passa se obj é falsy (nil ou false)

```
expect(obj).to be_falsey
```

passa se obj é nil

```
expect(obj).to be_nil
```

passa se obj é truthy (não nil e não false)

```
expect(obj).to be
```

Note que os be matchers já seriam o suficiente para fazer qualquer tipo de teste para o seu software. Por exemplo, poderíamos reescrever o teste:

it "is possible to put words on it" do

bag =

BagOfWords

.new

bag.put(

"hello", "world"

)

expect(bag.words.size).to eq(

2) end

Para:

it "is possible to put words on it" do

bag =

BagOfWords

```
.new
```

```
    bag.put(  
    "hello", "world"  
    )
```

```
    expected = (bag.words.size ==  
    2  
    )
```

```
    expect(expected).to be_truthy
```

```
end
```

Mas usar os "be matchers" para tudo, como usamos aqui, prejudicaria a semântica dos nossos testes e é por isso que temos outros matchers.

Os matchers de equidade

Existem vários matchers para comparar se dois objetos são iguais. O mais importante é o `eq`, que compara dois objetos utilizando o operador `==` (comparação genérica):

```
a = "some string"
```

```
b =
```

```
"some string"
```

```
a == b
```

```
# true
```

```
expect(a).to eq(b)
```

```
# passa pois a == b
```

Como existem diversas formas de comparar objetos no Ruby, o RSpec provê outros matchers, porém estes são menos usados:

```
expect(a).to equal(b) # passa quando a e b são referências
```

```
# para o mesmo objeto (a.equal?(b))
```

```
expect(a).to be(b)
```

```
# equivalente a `expect(a).to equal(b)`
```

```
expect(a).to eql(b)
```

```
# passa quando a e b possuem
```

```
# o mesmo hash code (a.eql?(b))
```

Matchers relacionados a arrays

O RSpec nos oferece alguns matchers específicos para verificação de arrays. O primeiro é o `MatchArray`, usado para verificar se um array é "igual" a outro, independente da ordem dos seus elementos. Segue um exemplo de uso:

```
array = [1, 2, 3, 4
```

```
]
```

```
expect(array).to match_array([
```

```
1, 2, 3, 4
```

```
])
```

```
expect(array).to match_array([
```

```
4, 3, 2, 1
```

)

```
expect(array).not_to match_array([
```

```
1, 2, 3
```

)

```
expect(array).not_to match_array([
```

```
1, 2, 3, 4, 5])
```

Outro matcher relacionado à verificação de arrays é o Include Matcher. Você pode usá-lo para verificar a relação de pertinência entre um ou mais elementos e um determinado array. Segue um exemplo de uso:

```
array = [1, 2, 3, 4
```

```
]
```

```
expect(array).to
```

```
include(1
```

```
)
```

```
expect(array).to
```

```
include(1, 2, 3
```

```
)
```

```
expect(array).to  
include(1, 2, 3, 4  
)
```

```
expect(array).not_to  
include(0  
)
```

```
expect(array).not_to  
include(5  
)
```

```
expect(array).not_to  
include(5, 6, 7, 8  
)
```

```
expect(array).not_to  
include([1, 2, 3, 4])
```

Por fim, existem também os matchers `start_with` e `end_with` que servem para verificar se um array começa ou termina com uma sequência de elementos. Segue um exemplo de uso:

```
array = [1, 2, 3, 4
```

]

expect(array).to start_with(

1

)

expect(array).to start_with(

1, 2

)

expect(array).not_to start_with(

2

)

expect(array).to end_with(

4

)

expect(array).to end_with(

3, 4

)

expect(array).not_to end_with(

3)

Matchers relacionados a hashes

Para verificar hashes, o RSpec nos oferece o Include Matcher, o mesmo matcher que vimos na verificação de arrays. Segue um exemplo de uso desse matcher com hashes:

```
hash = { a: 7, b: 5  
}
```

#você pode usar para verificar se um hash tem uma ou mais chaves

```
expect(hash).to  
include(:a  
)  
expect(hash).to  
include(:a, :b  
)
```

você pode usar para verificar se um hash tem um ou mais pares # de chave - valor

expect(hash).to

include(a: 7

)

expect(hash).to

include(b: 5, a: 7

)

expect(hash).not_to

include(:c

)

expect(hash).not_to

include(a: 11

)

expect(hash).not_to

include(a: 13, c: 11

)

expect(hash).not_to

include(:c, :d)

Matchers relacionados a strings

O RSpec oferece alguns matchers relacionados a strings, sendo que alguns deles são os mesmos usados para verificar arrays. Vamos começar olhando o Match Matcher, que serve para verificar o valor de uma string de acordo com uma expressão regular:

```
string = "hugo barauna"
```

```
expect(string).to match(
```

```
  /hugo/
```

```
)
```

```
expect(string).to match(
```

```
  /araun/
```

```
)
```

```
expect(string).not_to match(
```

```
  /barao/
```

```
)
```

```
expect(string).not_to match(
```

```
  /hugs/)
```

Agora seguem exemplos dos matchers usados por strings e arrays, começando pelo include:

```
string = "hugo barauna"
```

```
expect(string).to
```

```
include("h"
```

```
)
```

```
expect(string).to
```

```
include("hugo"
```

```
)
```

```
expect(string).to
```

```
include("hugo", "bara"
```

```
)
```

```
expect(string).not_to
```

```
include("barao"
```

```
)
```

```
expect(string).not_to
```

```
include("hugs")
```

Por fim, você também pode usar os matchers `start_with` e `end_with` com strings:

```
string = "hugo barauna"
```

```
expect(string).to start_with(
```

```
"hugo"
```

```
)
```

```
expect(string).not_to start_with(
```

```
"barauna"
```

```
)
```

```
expect(string).to end_with(
```

```
"barauna"
```

```
)
```

```
expect(string).not_to end_with(
```

```
"hugo")
```

Predicate matchers

Um dos tipos de matchers mais famosos do RSpec são os Predicate Matchers. Para entendê-los, vamos começar estudando o seguinte exemplo.

Imagine que você está desenvolvendo uma aplicação de e-commerce e tem uma classe chamada Cart. Essa classe tem um método de instância chamado empty?, que informa se o carrinho está vazio. Para especificá-lo, você poderia fazer o seguinte teste:

```
RSpec.describe Cart do
```

```
  describe
```

```
    "#empty?" do
```

```
      it
```

```
        "returns true when the cart has no products" do
```

```
          cart =
```

```
            Cart
```

```
              .new
```

```
            expect(cart.empty?).to be_truthy
```

end

end end

O RSpec oferece um matcher dinâmico que podemos usar para reescrever essa verificação da seguinte forma:

```
expect(cart).to be_empty # chama o método cart.empty?
```

O que o RSpec faz por trás é invocar o método `empty?` via metaprogramação. Toda vez que você faz uma verificação no formato `be_method_name` o RSpec executa o predicate `method_name?` no objeto sendo testado. Esse tipo de matcher é bem usado porque os predicate methods são uma convenção bastante comum na comunidade Ruby.

Existem ainda outros três predicate matchers. O segundo é o `have_method_name`, usado para métodos no formato `has_method_name?`. Um exemplo desse caso pode ser visto para o método `has_key?` de um hash:

```
hash = { key: 1 } # chama o método hash.has_key?(:key)
```

```
expect(hash).to have_key(  
  :key)
```

Você pode utilizar essa forma de predicate matcher para os seus próprios objetos também. Imagine o exemplo da classe Cart, ela poderia ter o método has_products? e você poderia especificar esse método do seguinte modo:

```
RSpec.describe Cart do
```

```
  describe
```

```
    "#has_products?" do
```

```
      it
```

```
        "returns true if it has products" do
```

```
          product =
```

```
            Product
```

```
              .new
```

```
              cart =
```

```
                Cart
```

```
                  .new(product)
```



```
# chama o método cart.has_products?
```

```
expect(cart).to have_products
```

```
end
```

```
end end
```

O terceiro e quarto predicate matchers são os matchers `be_a_method_name` e `be_an_method_name`. Para entendê-los, imagine que sua classe `Cart` tenha os métodos `thing?` e `object?`, e ambos devem retornar `true`. Você pode especificá-los do seguinte modo:

```
expect(cart).to be_a_thing # chama o método cart.thing?
```

```
expect(cart).to be_an_object
```

```
# chama o método cart.object?
```

Matchers para exceptions

O RSpec oferece um matcher para você especificar que um método levanta uma determinada exception: é o `RaiseError` Matcher. Você pode usá-lo com o método `raise_error` e seu alias `raise_exception`:

```
expect { raise }.to raise_error  
  
expect { raise }.to raise_exception
```

Perceba que, diferentemente do modo padrão de escrever uma expectation, o argumento que é passado para o método `expect` é um bloco:

```
# correto
```

```
expect { raise }.to raise_error
```

```
# correto
```

```
expect
```

```
do
```

```
  raise
```

```
end
```

```
.to raise_error
```

```
# errado
```

```
expect(raise).to raise_error
```

É possível verificar também se a exception levantada é de uma classe específica:

```
expect { raise RuntimeError }.to raise_error(RuntimeError  
)
```

```
expect { raise  
StandardError }.to_not raise_error(RuntimeError)
```

Por fim, você pode verificar também a mensagem da exception levantada:

```
expect { raise "error" }.to raise_error("error"  
)
```

```
expect { raise  
"error" }.to_not raise_error("other error")
```

Matchers para comparação de números

Para comparar se um valor é igual a um certo número você deve usar o eq matcher, tal como em:

```
expect(hugo_age).to eq(27)
```

Mas, e para comparar se um número é menor ou maior que outro? Para esses casos, você tem os seguintes matchers:

```
expect(7).to be < 10 expect(7).to be > 1
```

```
expect(  
7).to be <= 7 expect(7).to be >= 7
```

Matchers relacionados a números float

Fazer a verificação de números float pode ser problemático, pois devido à parte fracionária do número, não dá para simplesmente utilizar um matcher de equidade. Imagine, por exemplo, que você voltou para as aulas de geometria plana e quer confirmar que você sabe que o valor de Pi é 3.14:

```
expect(Math::PI).to eq(3.14) # executa Math::PI == 3.14
```

Acontece que esse teste vai falhar, porque Pi não é exatamente igual a 3.14, ele é aproximadamente 3.14. Uma aproximação de Pi com mais casas decimais poderia ser por exemplo 3.141592653589793.

Para resolver esse problema, o RSpec nos oferece o BeWithin matcher, que serve para fazer comparações entre números de forma aproximada. Poderíamos reescrever o teste do seguinte modo com esse matcher:

```
expect(Math::PI).to be_within(0.01).of(3.14)
```

O que o teste está fazendo é comparar se o valor de Math::PI é maior ou igual a $3.14 - 0.01$ ou menor ou igual ao valor de $3.14 + 0.01$. Ou seja, para uma expectation genérica no seguinte formato:

```
expect(actual).to be_within(delta).of(expected)
```

O que esse matcher faz é verificar se a seguinte expressão é verdadeira:

$$(\text{expected} - \text{delta}) \leq \text{actual} \leq (\text{expected} + \text{delta})$$

Matchers para ranges

Para versões do Ruby a partir da 1.9, o RSpec oferece um matcher para verificar se um ou mais elementos pertencem a um dado range: é o Cover Matcher. Você pode usá-lo da seguinte forma:

```
range = (1..10
```

```
)
```

```
expect(range).to cover(
```

```
1
```

```
)
```

```
expect(range).to cover(
```

```
10
```

```
)
```

```
expect(range).to cover(
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
)
```

```
expect(range).to cover(
```

```
5, 6, 7, 8)
```

Matchers para verificar a classe de um objeto

Se você estiver escrevendo algum teste que precise verificar de qual classe seu objeto é, você pode usar os matchers `BeAnInstanceOf` e `BeAKindOf`. Eles funcionam do seguinte modo:

`# be_an_instance_of` verifica se o objeto é uma instância da `# classe dada`

```
expect(  
5).to be_an_instance_of(Integer  
)
```

```
expect(  
5).not_to be_an_instance_of(Numeric  
)
```

```
expect(  
5).not_to be_an_instance_of(String)
```

`# be_a_kind_of` verifica se o objeto é uma instância da classe `# dada` ou de uma subclasse da classe dada

```
expect(
  5).to be_a_kind_of(Integer
)
expect(
  5).to be_a_kind_of(Numeric
)
expect(
  5).to be_a_kind_of(Object
)

expect(
  5).not_to be_a_kind_of(String)
```

Matcher para verificar mudança de estado ou valor

Imagine que você está desenvolvendo um jogo e precise criar um objeto que tenha uma máquina de estados, por exemplo, para trackear se o jogo já começou ou não. Você quer que o objeto tenha um método `Game#start` que, ao ser disparado, o estado interno do jogo deve mudar de `:initial` para `:started`. Você pode especificar esse comportamento do seguinte modo:


```
RSpec.describe Game do
```

```
  describe
```

```
    "#start" do
```

```
      it
```

```
        "changes the game state from :initial to :started" do
```

```
          game =
```

```
            Game
```

```
              .new
```

```
            expect {
```

```
              game.start
```

```
            }.to change { game.state }.from(
```

```
              :initial).to(:started
```

```
            )
```

```
          end
```

end end

No exemplo, usamos o change matcher em:

```
expect {  
  game.start  
}.to change { game.state }.from(  
:initial).to(:started)
```

para verificar que o valor do estado do objeto game muda de :initial para :started quando a ação game.start for executada. Como você pôde ver no exemplo, o change matcher pode ser usado quando você quer verificar que a execução de um bloco de código causa uma mudança de estado de um objeto.

Seguem mais alguns exemplos de uso do change matcher:

verifica que ao rodar Counter.increment o valor de # Counter.count é modificado em duas unidades

```
expect {
```

Counter

.increment

}.to change {

Counter.count }.by(2

)

verifica que ao tentar salvar um user com um atributo inválido # o valor de User.count não é modificado

expect {

invalid_attributes = {

name: nil

}

user =

User

.new(invalid_attributes)

user.save

}.to_not change(

User, :count

)

verifica que ao adicionar alguns jogadores ao objeto team, # o valor de team.size é modificado por pelo menos uma unidade

```
expect {  
  team.add_players(some_players)  
}.to change(team,  
:size).by_at_least(1)
```

Pense nesse tipo de matcher como um verificador que analisará o valor antes e depois de alguma ação.

Outros matchers padrão do RSpec

Pronto, já vimos a maioria dos matchers padrão que vêm com o RSpec. Mas, como são muitos e muitas variações, não vale a pena ver todos os detalhes de todos os que vêm com o RSpec de uma vez só. Por isso, convido você a depois dar uma boa navegada pela documentação do RSpec para ver mais sobre os matchers padrão.

Você pode ver essa documentação de duas formas, no Relish (<https://relishapp.com/rspec/rspec-expectations/v/3-8/docs/built-in-matchers/>), que é uma documentação mais alto nível, ou você pode ver direto no código do RSpec, no seguinte link do GitHub: <https://github.com/rspec/rspec-expectations/blob/v3.8.2/lib/rspec/matchers.rb/>. Vale a pena guardar esses links, pois junto com o que vimos neste livro, eles são uma ótima referência sobre os matchers padrão do RSpec.

Agora que você já leu bastante sobre os matchers que vêm com o RSpec, descanse um pouquinho, pois a seguir você vai aprender a escrever os seus próprios matchers!

3.4 Custom matchers

Na seção anterior, nós aprendemos sobre os matchers padrões do RSpec e vimos que existem muitos deles. Mas mesmo com o RSpec nos oferecendo tantos matchers, às vezes podemos precisar de mais. Eventualmente, temos que escrever nossos próprios matchers. Vamos ver quando isso pode ser necessário.

Imagine que você está trabalhando em uma aplicação de e-commerce. Você está encarregado de trabalhar na funcionalidade de categorização de produtos. O escopo da funcionalidade diz que:

uma categoria contém uma ou mais subcategorias;

uma subcategoria contém um ou mais produtos;

uma categoria contém todos os produtos contidos por suas subcategorias.

Vamos pensar num exemplo para esse escopo. Na nossa aplicação pode existir uma categoria chamada "eletrônicos". Essa categoria pode conter duas subcategorias, tais como "computadores" e "celulares". A subcategoria "computadores" pode conter o produto MacBook e a subcategoria "celulares" pode conter o produto iPhone. Logo, a categoria "eletrônicos" deve conter os produtos MacBook e iPhone.

Até então, você já tem o código de duas classes. O código da classe Category:

```
class Category
```

```
  attr_reader :subcategories
```

```
  attr_reader :name
```

```
  def
```

```
    initialize(name)
```

```
      @name
```

```
    = name
```

```
      @subcategories
```

```
    = []
```

```
  end
```

```
def
add_subcategories(*subcategories)

# TODO


end end
```

E o código da classe Subcategory:

```
class Subcategory

attr_reader :products
```



```
def
```

```
  initialize(name)
```

```
  @name
```

```
  = name
```

```
  @products
```

```
  = []
```

```
end
```

```
def
```

```
  add_product(product)
```

```
  @products
```

```
  << product
```

```
end end
```

Agora, imagine que você precise escrever um teste unitário para especificar que uma categoria contém todos os produtos de suas subcategorias. Poderíamos começar escrevendo-o do seguinte modo:

```
RSpec.describe Category do
```

```
  it
```

```
    "contains all the products of its subcategories" end
```

No setup desse teste, vamos criar um objeto para a categoria "electronics", criar objetos para as subcategorias "computers" e "cell phones" e adicionar um produto para cada subcategoria:

```
RSpec.describe Category do
```

```
  it
```

```
    "contains all the products of its subcategories" do
```

```
      electronics =
```

```
        Category.new("electronics"
```

```
          )
```

```
        computers =
```

```
Subcategory.new("computers"  
)  
  cell_phones =  
    Subcategory.new("cell phones"  
    )  
    computers.add_product(  
      "MacBook"  
    )  
    cell_phones.add_product(  
      "iPhone"  
    )  
  
end end
```

Podemos adicionar essas subcategorias à categoria "electronics" e verificar que os produtos delas estão contidos na "electronics":

```
RSpec.describe Category do  
  
  it  
  
    "contains all the products of its subcategories" do
```

```
electronics =  
Category.new("electronics"  
)  
computers =  
Subcategory.new("computers"  
)  
cell_phones =  
Subcategory.new("cell phones"  
)  
computers.add_product(  
"MacBook"  
)  
cell_phones.add_product(  
"iPhone"  
)  
  
electronics.add_subcategories(computers, cell_phones)  
  
electronics_products =  
electronics.subcategories.flat_map(&
```

```
:products
)

expect(electronics_products).to
include("MacBook", "iPhone"
)

end end
```

Perceba como a intenção da lógica de verificação desse teste não está 100% clara:

```
electronics_products =
  electronics.subcategories.flat_map(&
:products
)

expect(electronics_products).to
include("MacBook", "iPhone")
```

Só de bater o olho nesse código, não dá para dizer que sua intenção é verificar que a categoria "electronics" contém os produtos "MacBook" e "iPhone". Esse é o primeiro ponto a melhorar.

O segundo ponto, nós podemos identificar ao rodar o teste e ler sua mensagem de erro:

1) Category contains all the products of its subcategories

Failure/Error:

```
expect(electronics_products).to include("MacBook", "iPhone")
```

```
expected [] to include "MacBook" and "iPhone"
```

A mensagem de erro não fala explicitamente que o erro é que a categoria "electronics" não contém nenhum dos produtos que ela deveria conter. Lembre-se de que a mensagem de erro de um teste é muito importante, pois uma mensagem de erro clara pode nos ajudar a corrigir mais rápido um teste que está quebrado.

Portanto, pelo menos esses dois pontos nós podemos melhorar no nosso teste: a clareza da intenção da lógica de verificação e a mensagem de erro. Para fazer essa melhoria, podemos escrever um custom matcher do RSpec e utilizá-lo para refatorar a verificação do nosso teste para ficar assim:

```
expect(electronics).to contain_products("MacBook", "iPhone")
```

Vamos aprender como fazer um custom matcher do RSpec.

Escrevendo um custom matcher do RSpec

O RSpec nos oferece uma DSL para escrever um custom matcher de modo bem simples. Para entender essa DSL, vamos escrever um custom matcher que serve para verificar se um número é múltiplo de 7. Esse matcher poderá ser usado do seguinte modo:

```
expect(21).to be_a_multiple_of(7)
```

O primeiro passo para escrevermos o custom matcher anterior é utilizar o método `RSpec::Matchers.define` da DSL do RSpec:

```
RSpec::Matchers.define :be_a_multiple_of
```

O método `define` cria um método nomeado segundo o argumento que foi passado para ele — no nosso caso, o método `be_a_multiple_of`. Continuando a seguir a DSL, é necessário passar um bloco para esse método:

```
RSpec::Matchers.define :be_a_multiple_of do |expected| end
```

Repare no argumento que é passado para o bloco, o `expected`. Ele é o mesmo que é passado para o nosso matcher na hora em que é utilizado. Logo, quando

utilizarmos `be_a_multiple_of(7)`, o valor de `expected` será 7.

Para finalizar o nosso matcher, é necessário colocar a lógica de verificação dentro dele, ou seja, a lógica para checar se um número é múltiplo de outro. Podemos fazer isso checando se o resto da divisão entre o número testado e o argumento passado para o matcher é zero:

```
RSpec::Matchers.define :be_a_multiple_of do
```

```
  |expected|
```

```
    match
```

```
  do
```

```
    |actual|
```

```
      (actual % expected) ==
```

```
      0
```

```
  end end
```

Repare que, para implementar a lógica de verificação, utilizamos o método `match`. Chamamos esse método passando um bloco para ele com um argumento, o `actual`, que é o objeto sendo testado. No exemplo que estamos usando:


```
expect(21).to be_a_multiple_of(7)
```

actual é o 21 (objeto sendo testado) e o expected é o 7 (argumento passado para o matcher).

Pronto, isso é o mínimo necessário para escrevermos nosso próprio matcher. Com esse matcher pronto, poderíamos escrever um teste do seguinte modo:

```
RSpec::Matchers.define :be_a_multiple_of do
```

```
  |expected|
```

```
    match
```

```
  do
```

```
    |actual|
```

```
      (actual % expected) ==
```

```
      0
```

```
  end end
```

```
RSpec.describe "The be_a_multiple_of custom matcher" do
```

it

"verifies if a number is a multiple of another one" do

expect(

21).to be_a_multiple_of(7

)

expect(

15).to be_a_multiple_of(3

)

expect(

7).not_to be_a_multiple_of(3

)

end end

Agora que já sabemos como construir nosso próprio matcher, vamos voltar ao nosso objetivo inicial: escrever um matcher para verificar que uma categoria contém um ou mais produtos.

O teste que escrevemos para a categorização de produtos está até então do seguinte modo:

```
RSpec.describe Category do
```

```
  it
```

```
    "contains all the products of its subcategories" do
```

```
      electronics =
```

```
        Category.new("electronics"
```

```
        )
```

```
      computers =
```

```
        Subcategory.new("computers"
```

```
        )
```

```
      cell_phones =
```

```
        Subcategory.new("cell phones"
```

```
        )
```

```
      computers.add_product(
```

```
        "MacBook"
```

```
      )
```

```
      cell_phones.add_product(
```

```
        "iPhone"
```

```
      )
```

```
electronics.add_subcategories(computers, cell_phones)

electronics_products =
  electronics.subcategories.flat_map(&
:products
)
  expect(electronics_products).to
include("MacBook", "iPhone"
)

end end
```

Queremos modificá-lo usando um custom matcher para ficar assim:

```
RSpec.describe Category do

  it
  "contains all the products of its subcategories" do
```

```
electronics =  
Category.new("electronics"  
)  
computers =  
Subcategory.new("computers"  
)  
cell_phones =  
Subcategory.new("cell phones"  
)  
computers.add_product(  
"MacBook"  
)  
cell_phones.add_product(  
"iPhone"  
)  
  
electronics.add_subcategories(computers, cell_phones)  
  
expect(electronics).to contain_products(  
"MacBook", "iPhone"  
)
```

end end

Para escrever esse custom matcher, vamos mais uma vez utilizar a DSL do RSpec e extrair a lógica de verificação original para dentro dele:

```
RSpec::Matchers.define :contain_products do

  |*products|

    match

  do

    |category|

      subcategories_products =

        category.subcategories.flat_map(&

          :products

        )

      expect(subcategories_products & products).to eq products

    end end
```

Agora que temos o custom matcher pronto, podemos utilizá-lo no nosso teste para ficar assim:

```
RSpec.describe Category do
```

```
  it
```

```
    "contains all the products of its subcategories" do
```

```
      electronics =
```

```
        Category.new("electronics"
```

```
        )
```

```
      computers =
```

```
        Subcategory.new("computers"
```

```
        )
```

```
      cell_phones =
```

```
        Subcategory.new("cell phones"
```

```
        )
```

```
      computers.add_product(
```

```
        "MacBook"
```

```
      )
```

```
      cell_phones.add_product(
```

```
        "iPhone"
```

```
      )
```

```
electronics.add_subcategories(computers, cell_phones)

expect(electronics).to contain_products(
  "MacBook", "iPhone"
)

end end
```

Repare que, após essa refatoração de extrair uma lógica de verificação complexa para um custom matcher, o comportamento esperado no nosso teste ficou bem mais claro.

Outro ponto que ficou melhor é a mensagem de erro do nosso teste. Para vermos isso, basta rodarmos e ver a seguinte mensagem:

Failures:

1) Category contains all the products of its subcategories

Failure/Error:

```
expect(electronics).to contain_products("MacBook", "iPhone")
```



```
expected #<Category @name="electronics", @subcategories=[]>
```

```
to contain products "MacBook" and "iPhone"
```

Na mensagem de erro já está mais claro que o teste falhou porque a categoria electronics não contém os produtos "MacBook" e "iPhone", mas podemos melhorá-la ainda mais.

A mensagem de erro foi gerada automaticamente pelo RSpec a partir do nome do nosso custom matcher. Como nem sempre a mensagem padrão fica clara, é possível customizá-la. Podemos fazer isso usando o método `failure_message` da DSL de custom matcher do RSpec:

```
failure_message do
```

```
  |category|
```

```
    "expected category '#{category.name}' "
```

```
  \
```

```
    "to contain products #{products}" end
```

Esse código deve ser adicionado dentro da definição do nosso custom matcher:

```
RSpec::Matchers.define :contain_products do
  |*products|
    match
do
  |category|
    subcategories_products =
      category.subcategories.flat_map(&
:products
    )
    expect(subcategories_products & products).to eq products
end
```

```
failure_message
do
  |category|

  "expected category '#{category.name}' "
\
```

```
"to contain products #{products}"
```

```
end end
```

Dessa vez, ao rodarmos o teste, ele falha com a seguinte mensagem:

Failures:

1) Category contains all the products of its subcategories

Failure/Error:

```
expect(electronics).to contain_products("MacBook", "iPhone")
```

```
expected category 'electronics'
```

```
to contain products ["MacBook", "iPhone"]
```

Após a customização, a mensagem de erro ficou bem mais clara. Além dessa opção de customização, o RSpec tem várias outras. Como são muitas, convido você a olhar a documentação de custom matchers do RSpec (<https://relishapp.com/rspec/rspec-expectations/v/3-8/docs/custom-matchers/>) para ver as outras opções de customização.

Agora que você já teve a experiência de construir um custom matcher com a DSL do RSpec, vamos ver melhor o que de fato é um matcher para o RSpec e descobrir o protocolo de matchers por trás disso tudo.

3.5 Entendendo o protocolo interno de matcher do RSpec

Na seção anterior, nós aprendemos a construir nossos próprios matchers para o RSpec usando a DSL de custom matchers. Essa DSL serve para facilitar a criação de novos matchers, mas ela não é o único modo de se criar um matcher novo. Você pode criar seu próprio matcher do zero se entender o protocolo por trás da relação entre uma expectation e o seu matcher.

Como vimos na seção Aprendendo a estrutura básica de um teste com RSpec, uma expectation segue a seguinte estrutura básica:

```
expect(actual).to matcher(expected)
```

Para ficar mais claro como uma expectation é executada, vamos colocar mais alguns parênteses nessa estrutura:

```
expect(actual).to(matcher(expected))
```

Você pode ver pela linha desse código que o matcher utilizado não é nada mais do que um argumento para o método to. O contrato entre esse método e o seu argumento não é baseado no tipo desse objeto matcher, mas sim no seu

comportamento, ou seja, é baseado no famoso duck typing.

■

Duck typing

Duck typing é um estilo de programação que, em vez de levar em consideração o tipo de um objeto (classe), considera o que o objeto sabe fazer (a quais métodos ele responde). Vimos que o RSpec utiliza esse estilo nos seus matchers.

Para o RSpec, um matcher não é um objeto de uma classe específica, um matcher é um objeto qualquer que responde à mensagem matches?.

Resumindo, ao usar duck typing, um objeto é definido pelo que ele sabe fazer, não pela sua classe.

O termo vem do ditado "Se parece com um pato, nada como um pato e grasna como um pato, então provavelmente é um pato".

■

O protocolo mínimo de um matcher é que ele responda para o método matches? (actual), no qual actual é o objeto sendo testado. Conhecendo esse protocolo, podemos escrever o matcher mais simples do mundo, que seria o seguinte:

```
class SimplestMatcher
```

```
def  
  matches?(actual)
```

```
  true
```

```
end end
```

Para utilizá-lo, poderíamos escrever um teste do seguinte modo:

```
class SimplestMatcher
```

```
  def  
    matches?(actual)
```

```
  true
```

end end

RSpec.describe "The matcher protocol" do

context

"a minimal matcher" do

it

"has a #matches?(actual) method" do

expect(

"anything").to SimplestMatcher

.new

end

end end

Talvez você estranhe a linha `expect("anything").to SimplestMatcher.new` devido ao argumento sendo passado para o método `to` ser um objeto normal. Poderíamos deixar esse código um pouquinho mais "bonito" construindo um helper method que retorne uma instância do nosso matcher:

```
class SimplestMatcher
```

```
  def
```

```
    matches?(actual)
```

```
    true
```

```
  end end
```

```
RSpec.describe "The matcher protocol" do
```

```
  it
```

```
    "has a #matches?(actual) method" do
```

```
    expect(  
      "anything"  
    ).to simplest_matcher  
  
  end  
  
  def  
    simplest_matcher  
  
    SimplestMatcher  
      .new  
  
    end end
```

Essa ideia de ter um helper method que retorna uma instância do matcher que você quer é exatamente o que o RSpec faz com praticamente todos os seus matchers. Podemos ver isso examinando o código do RSpec para o eq matcher. Como vimos antes, esse matcher é utilizado do seguinte modo:

```
actual = "got it"
```

```
expected =
```

```
"got it"
```

```
expect(actual).to eq(expected)
```

O que o RSpec faz por trás é que ele tem dentro do módulo RSpec::Matchers um helper method chamado eq com o código adiante:

```
def
```

```
eq(expected)
```

```
BuiltIn::Eq.new(expected) end
```

Como você pode ver, esse helper method retorna uma instância de um built-in matcher do RSpec, o Eq Matcher. Dentro desse matcher tem uma implementação do método matches?(actual), seguindo o protocolo de matchers, com um código semelhante a este:

```
def
```

```
matches?(actual)
```

```
actual == expected
```

```
end
```

Pronto, agora você já conhece o contrato básico entre uma expectation e um matcher. Para um objeto ser um matcher, basta que ele responda para a mensagem `matches?(actual)`.

O último ponto para entender o contexto inteiro de uma expectation é saber para que ela usa esse método `matches?(actual)`. Internamente, o método `to` chama esse método, e se ele retornar `true` é porque a expectation passou, já se ele retornar `false`, é porque ela quebrou. Simples assim.

A real importância de conhecer esse contrato não é para você construir seus próprios matchers — isso você pode fazer usando a DSL do RSpec, pois ela automatiza um monte de coisas para você. Conhecer esse contrato é importante para que uma expectation do RSpec pare de parecer mágica e faça sentido para você. Conhecer as entranhas das suas ferramentas sempre é útil, e agora você já conhece um pouco mais dos internals do RSpec. Como diz um grande amigo meu: "não é feitiçaria, é tecnologia!".

Pontos-chaves deste capítulo

Neste capítulo nós conhecemos a estrutura básica de um teste com RSpec. Conhecemos os métodos `describe`, `it` e a construção de uma expectation. Aprendemos também que o RSpec tem muitos built-in matchers que nos possibilitam escrever testes que pareçam com documentação, que estejam próximos da semântica do domínio do problema sendo resolvido e que gerem mensagens de erro claras.

Em seguida, vimos como é simples escrever nosso próprio matcher usando a DSL de custom matchers do RSpec. Dado que é muito fácil fazer isso, não hesite em fazê-lo quando o motivo for clareza do teste. Por fim, vimos como o RSpec emprega duck typing no seu protocolo de matchers. Agora, quando alguém vier com aquele velho papo de "Ruby é magia negra", pelo menos para os matchers do RSpec você já pode dizer que não é, e pode mostrar como funciona por trás dos panos.

Vimos todo o básico necessário para escrever testes com RSpec, então já podemos ir para o próximo passo, que é como organizar, refatorar e reutilizar o código dos nossos testes. Pare um pouco para tomar um café, pois veremos tudo isso no próximo capítulo.

Capítulo 4

Organização, refatoração e reúso de testes com o RSpec

Neste capítulo você vai aprender como organizar seus testes. Verá que um teste de unidade tem uma narrativa e a vantagem de segui-la. Aprenderá também como refatorar seus testes para ficarem menos repetitivos, mas sem perder sua clareza. Vamos lá!

4.1 Reduzindo duplicação com hooks do RSpec

O RSpec oferece três tipos de hooks para podermos reduzir a duplicação dos nossos testes e rodarmos códigos arbitrários antes, depois, ou antes e depois dos nossos testes. São eles: before hook, after hook e around hook. Vamos começar analisando um exemplo de quando podemos usar o before hook para reduzir a duplicação de código dos nossos testes.

Usando o before hook para reduzir duplicação

Imagine que você está escrevendo um jogo. Parte da especificação desse jogo diz que, quando o jogador está na última fase, se ele acertar o alvo, o jogo deve fazer as seguintes ações:

parabenizar o jogador;

setar a pontuação para 100.

Como bom praticante de TDD, antes de começar a desenvolver essa funcionalidade, você especifica esse comportamento com a seguinte estrutura de testes:

RSpec.describe Game, "in the final phase" do

context

"when the player hits the target" do

it

"congratulates the player"

it

"sets the score to 100"

end end

Antes de implementarmos esses testes, vamos analisar algumas novidades na sua estrutura.

A primeira é a string sendo passada como segundo argumento do método describe:

RSpec.describe Game, "in the final phase"

Como vimos no capítulo anterior, o método `describe` serve para agrupar os testes de modo semântico. A forma mais simples de agrupar os seus testes é por classe. No entanto, há outros modos. No código anterior, estamos agrupando nossos testes por classe e por um determinado estado do objeto dessa classe, no caso, o estado em que o game está na última fase. Um dos benefícios de se utilizar essa string como segundo argumento é o modo como ela fica no output de execução dos nossos testes quando executamos o RSpec com o formato de saída `documentation`:

```
$ rspec --format documentation hooks_spec.rb
```

Game in the final phase

when the player hits the target

congratulates the player

sets the score to 100

A segunda novidade que fizemos nesse teste é o uso do método `context`. Esse método é apenas um alias do método `describe` e também serve para agrupar nossos testes de forma semântica. Nesse caso, preferimos usar o `context` porque a leitura dele nesse contexto fica melhor do que usar o método `describe`. Compare ambos os modos a seguir, primeiro usando `context`:

```
RSpec.describe Game, "in the final phase" do
```

context

"when the player hits the target" do

E agora usando o describe:

RSpec.describe Game, "in the final phase" do

describe

"when the player hits the target" do

Explicadas as novidades, vamos continuar a trabalhar nos nossos testes. Vamos começar escrevendo o primeiro:

context "when the player hits the target" do

it

"congratulates the player"

(...)

O setup desse teste envolve criar um objeto game e colocá-lo na fase final:

```
context "when the player hits the target" do
```

```
  it
```

```
    "congratulates the player" do
```

```
      game =
```

```
        Game
```

```
      .new
```

```
      game.phase =
```

```
        :final
```

```
    end
```

```
  # (...) end
```

O resto do teste envolve verificar que, dado esse setup, quando o jogador acertar o alvo, o jogo deve dar-lhe parabéns:

```
context "when the player hits the target" do
```

```
  it
```

```
    "congratulates the player" do
```

```
      game =
```

```
        Game
```

```
        .new
```

```
        game.phase =
```

```
        :final
```

```
        game.player_hits_target
```

```
        expect(game.output).to eq(
```

```
        "Congratulations!"
```

```
    )
```

end

(...) end

O teste já está pronto. Por não ser o foco deste capítulo, vamos pular a parte de escrever o código para fazê-lo passar. Vamos ao próximo teste.

context "when the player hits the target" do

(...)

it

"sets the score to 100" end

As fases de setup e exercício desse teste são iguais às do anterior, devemos criar um jogo na fase final e depois fazer com que o jogador acerte o alvo:

```
context "when the player hits the target" do
```

```
# (...)
```

```
it
```

```
"sets the score to 100" do
```

```
  game =
```

```
    Game
```

```
  .new
```

```
    game.phase =
```

```
  :final
```

```
    game.player_hits_target
```

```
end end
```

A única parte que muda para esse teste é a etapa de verificação, na qual

queremos verificar que a pontuação do jogo fica em 100:

```
context "when the player hits the target" do
```

```
# (...)
```

```
it
```

```
"sets the score to 100" do
```

```
  game =
```

```
    Game
```

```
    .new
```

```
    game.phase =
```

```
    :final
```

```
    game.player_hits_target
```

```
    expect(game.score).to eq(
```


100

)

end end

Agora vamos analisar como ficou o arquivo de testes inteiro depois de termos escrito os dois testes:

```
RSpec.describe Game, "in the final phase" do
```

```
  context
```

```
    "when the player hits the target" do
```

```
      it
```

```
        "congratulates the player" do
```

```
          game =
```

```
            Game
```

```
              .new
```

```
              game.phase =
```

```
                :final
```

```
game.player_hits_target
```

```
expect(game.output).to eq(
```

```
"Congratulations!"
```

```
)
```

```
end
```

```
it
```

```
"sets the score to 100" do
```

```
game =
```

```
Game
```

```
.new
```

```
game.phase =
```

```
:final
```

```
game.player_hits_target  
  
expect(game.score).to eq(  
100  
)
```

```
end
```

```
end end
```

Perceba que existe duplicação entre eles. Assim como você segue o DRY (don't repeat yourself) no seu código de produção, você também pode usar a mesma ideia no código dos seus testes. O RSpec facilita a redução de duplicação através do before hook. O before é um método do RSpec para o qual você pode passar um bloco de código que vai ser executado antes de cada teste ou antes de todos os testes:

```
before(:each) do
```

```
# roda esse código uma vez antes de cada teste end
```

ou

não precisa passar o :each, pois ele é a opção default do # before

before

do

roda esse código uma vez antes de cada teste end

before(:all) do

roda esse código apenas uma vez antes de todos os testes end

Vamos utilizar o before para primeiro extrair o setup comum entre nossos dois testes:

```
RSpec.describe Game, "in the final phase" do
```

```
  before
```

```
  do
```

```
    @game = Game
```

```
    .new
```

```
    @game.phase = :final
```

```
  end
```

```
  context
```

```
    "when the player hits the target" do
```

```
      it
```

```
        "congratulates the player" do
```

@game

.player_hits_target

expect(

@game.output).to eq("Congratulations!"

)

end

it

"sets the score to 100" do

@game

.player_hits_target

expect(

@game.score).to eq(100

)

end

end end

Repare que extraímos o setup para um before hook dentro do escopo do describe. Poderíamos ter extraído para o escopo do context, mas preferimos extrair para o escopo do describe porque, segundo a string desse describe (Game, "in the final phase"), todos os testes dele vão compartilhar o mesmo estado do game, ou seja, o game estar na última fase.

Repare também que, ao extrairmos o setup para o before hook, foi necessário salvar o objeto game como uma variável de instância. Fizemos isso porque esse é o modo de compartilhar variáveis entre um before hook e os testes dentro do seu escopo.

Agora que já extraímos parte do código duplicado nos nossos testes, vamos analisar se vale a pena extrair o resto da duplicação.

4.2 DRY versus clareza nos testes

O que ainda temos duplicado entre ambos os testes é a linha
`@game.player_hits_target:`

```
context "when the player hits the target" do
```

```
  it
```

```
    "congratulates the player" do
```

```
      @game
```

```
        .player_hits_target
```

```
          expect(
```

```
            @game.output).to eq("Congratulations!")
```

```
          )
```

```
        end
```


it

"sets the score to 100" do

@game

.player_hits_target

expect(

@game.score).to eq(100

)

end end

Poderíamos tranquilamente extrair essa linha para um before no escopo do contexto. Ficaria assim:

context "when the player hits the target" do

before {

@game

.player_hits_target }

it

"congratulates the player" do

expect(

@game.output).to eq("Congratulations!"

)

end

it

"sets the score to 100" do

expect(

@game.score).to eq(100

)

end end

No entanto, será que vale a pena reduzir a duplicação ao máximo nos nossos testes? A resposta é: não. A aplicação do conceito de DRY no código de testes é diferente da aplicação do conceito de DRY no código de produção.

No código de testes, uma das maiores preocupações que você deve ter é se o teste está claro, se o leitor desse código consegue ler e entender o que o teste está querendo testar. Ou melhor ainda, o leitor deve entender a relação de causa e consequência do seu teste. Podemos visualizar melhor essa relação com a seguinte máquina de estados:

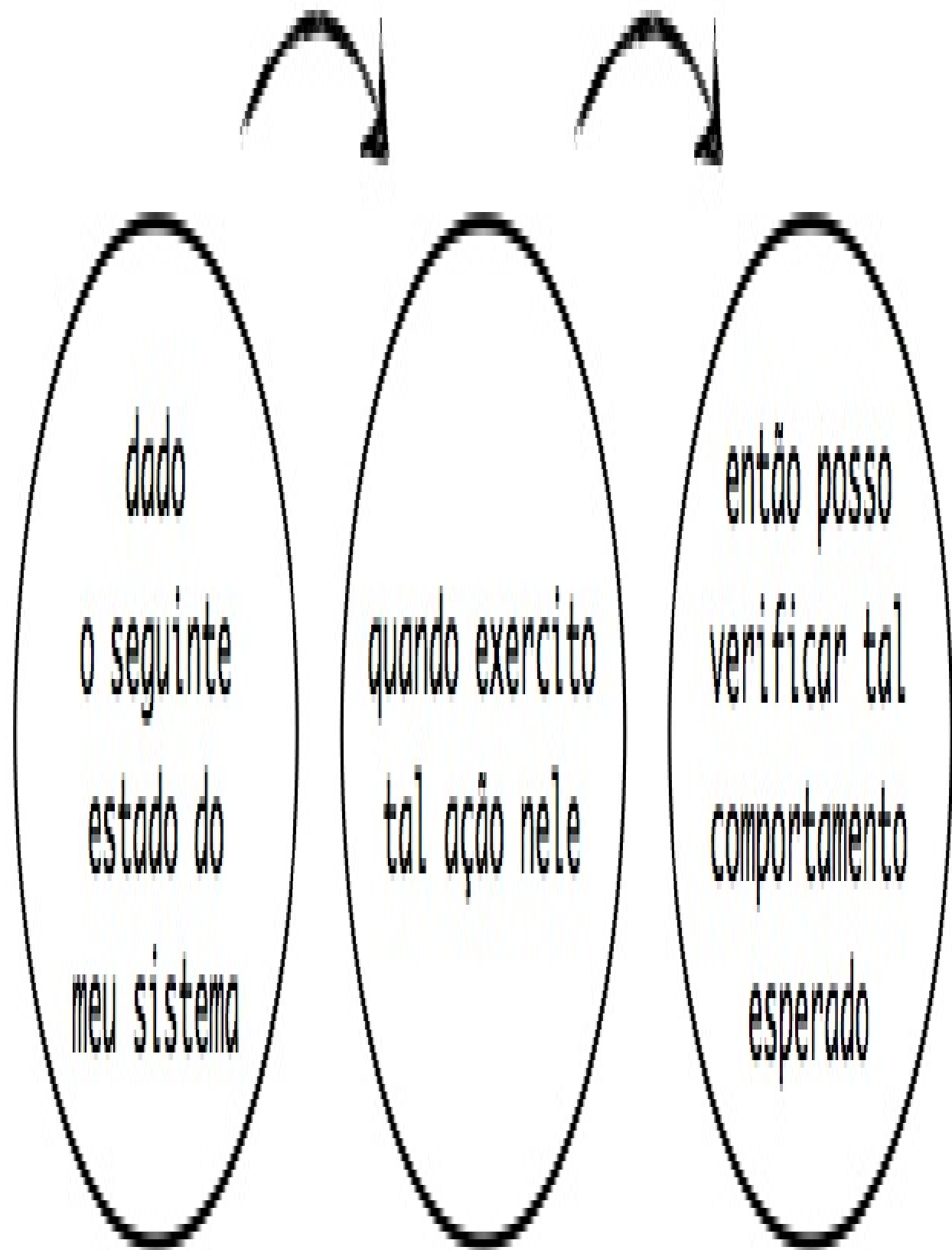


Figura 4.1: Relação de causa e consequência de um teste

Ao ler um teste, deve estar claro para o leitor a relação de causa e consequência demonstrada no diagrama anterior. Caso esteja difícil de entender essa relação, então falta clareza no teste.

Por isso, toda vez que você for refatorar seus testes, seja pelo motivo de DRY ou outra razão, você deve sempre pensar se essa refatoração não vai impactar na clareza do seu teste. No exemplo que demos ela não foi tão impactada porque os testes são pequenos:

```
context "when the player hits the target" do
```

```
  before {
```

```
    @game
```

```
    .player_hits_target }
```

```
  it
```

```
    "congratulates the player" do
```

```
      expect(
```

```
@game.output).to eq("Congratulations!"  
)
```

```
end
```

```
it  
"sets the score to 100" do
```

```
  expect(  
    @game.score).to eq(100  
  )
```

```
end end
```

Um modo de avaliar se o seu teste está perdendo a clareza é ler somente o código do teste em si, ou seja, somente o bloco de código passado para o it. Vamos fazer isso para o seguinte teste:

```
it "sets the score to 100" do
```

```
expect(  
  @game.score).to eq(100) end
```

Só de ler esse teste você poderia se perguntar:

Em que estado o objeto game precisa estar para que o score tenha o valor 100?

O que deve ser feito no objeto game para que o score seja definido como 100?

Essas perguntas são válidas para que se possa entender a relação de causa e consequência de um teste. Se ele estivesse escrito do seguinte modo:

```
it "sets the score to 100" do
```

```
  game =
```

```
    Game
```

```
  .new
```

```
  game.phase =
```

```
    :final
```

```
  game.player_hits_target
```

```
expect(  
@game.score).to eq(100) end
```

essas perguntas teriam sido respondidas apenas lendo o código do teste em si. Mas como o refatoramos, é necessário ler o código dos before hooks também.

No caso desse teste, como o arquivo de testes é muito pequeno e o código dos before hooks está bem perto do código do teste em si, ele não perde muito em clareza. No entanto, se ele estiver em um arquivo muito grande e você tiver que dar muito scroll, fazendo idas e voltas entre o teste e os before hooks, pode ser que ele tenha perdido a clareza.

Resumindo, antes de extrair tudo que for repetido entre os seus testes para before hooks, pense se ao fazer isso você não está afetando a clareza do seu teste. Não extraia tudo para before hooks só para deixar o código do seu teste "DRY", pense nas vantagens e desvantagens de fazer isso.

Lembre-se de que em código de teste, uma das características mais importantes é a clareza e a possibilidade de entender facilmente a relação de causa e consequência.

4.3 After hook

O after hook do RSpec serve para você executar código após os seus testes. Assim como o before hook, ele dá a opção de executar código depois de cada teste ou depois de todos os seus testes:

```
after(:each) do
```

```
# roda esse código uma vez depois de cada teste end
```

```
# ou
```

```
# não precisa passar o :each, pois ele é a opção default do after
```

```
after
```

```
do
```

```
# roda esse código uma vez depois de cada teste end
```

```
after(:all) do
```

```
# roda esse código uma vez só depois de todos os testes end
```

Um exemplo de onde o after hook pode ser útil é quando é necessário deletar arquivos gerados durante os testes.

Imagine que você está escrevendo um teste de um objeto que salva um arquivo de cache no file system:

```
it "caches the result in a file" do
```

```
  expect {
```

```
    my_cool_object.run
```

```
  }.to change {
```

```
    File.exists?(cache_path) }.from(false).to(true) end
```

Ao rodá-lo, um arquivo de cache é gerado. Para que seu teste não deixe sujeira para trás, você pode usar o after hook para deletar esse arquivo gerado:

```
after(:all) do
```

```
FileUtils.rm(Dir.glob("#{cache_dir}/*")) end
```

```
it
```

```
"caches the result in a file" do
```

```
  expect {
```

```
    my_cool_object.run
```

```
  }.to change {
```

```
File.exists?(cache_path) }.from(false).to(true) end
```

4.4 Around hook

O around hook do RSpec é útil quando você precisa rodar código antes e depois do seu teste:

```
RSpec.describe "An around hook example" do
```

```
  around
```

```
  do
```

```
    |example|
```

```
      puts
```

```
      "Before the example"
```

```
      example.run
```

```
      puts
```

```
      "After the example"
```

```
  end
```

it

do

puts

"Inside the example"

end end

Podemos rodar esse teste para entender o caminho da execução do teste quando usamos um around hook:

Before the example

Inside the example

After the example

Vale a pena notar que o que fizemos agora poderia ter sido feito tranquilamente usando before e after hooks:

```
RSpec.describe "An around hook example" do
```

```
  before
```

```
  do
```

```
    puts
```

```
    "Before the example"
```

```
  end
```

```
  after
```

```
  do
```

```
    puts
```

```
    "After the example"
```

```
end
```

it

do

puts

"Inside the example"

end end

O lugar onde pode valer mais a pena usar o around hook, em vez de trocá-lo por um before e um after, é quando você tem alguma vantagem de rodar o seu teste dentro de um bloco. Um exemplo disso é quando você quer rodar os seus testes dentro de uma transação do seu banco de dados, de modo que você possa dar um revert de todos os dados salvos no banco pelo seu teste simplesmente dando um rollback na transação:

class Database

def self

.transaction

puts

"open transaction"

yield

puts

"rollback transaction"

end end

RSpec.describe "around hook" do

 around(

 :each) do

 |example|

Database

 .transaction(&example)

end

it

"runs the example as a proc" do

puts

"saving a lot of data in the database"

end end

Ao executar o teste, podemos ver a seguinte saída:

open transaction

saving a lot of data in the database

rollback transaction

Repare que o método `around` passa como argumento para o seu bloco o `example` (teste criado pelo método `it`), o qual você pode passar como uma `proc` para um método que receba um bloco, como fizemos em `Database.transaction(&example)`.

4.5 Organizando seus testes

Organizar bem os seus testes é importante para a filosofia de "testes como documentação" do BDD. Dependendo de como seus testes estão organizados, pode ficar mais difícil para o leitor entendê-los, prejudicando o entendimento da relação de causa e consequência. Como primeira dica para organizar seus testes, vamos dar uma olhada nas quatro fases de um teste.

Organizando seus testes segundo as quatro fases do xUnit

Um teste do padrão xUnit tem quatro fases, são elas:

setup: quando você coloca o SUT (system under test, por exemplo, o objeto sendo testado) no estado necessário para o teste;

exercise: na qual você interage com o SUT;

verify: na qual você verifica o comportamento esperado;

teardown: na qual você coloca o sistema no estado em que ele estava antes de o teste ser executado.

Talvez você não tenha percebido, mas todos os testes que escrevemos ao longo desse livro até agora estão seguindo essas fases. Veja, por exemplo, o teste da pilha que fizemos no capítulo 1. Primeiros passos com RSpec e Cucumber:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      it
```

```
        "puts an element at the top of the stack" do
```

```
          stack =
```

```
            Stack
```

```
              .new
```

```
            stack.push(
```

```
              1
```

```
            )
```

```
            stack.push(
```

```
              2
```

```
            )
```

```
            expect(stack.top).to eq(
```

2

)

end

end end

Leia esse teste e tente identificar as quatro fases. Eu pessoalmente gosto de deixar uma quebra de linha entre cada fase do teste, de modo que fique fácil entender sua estrutura só batendo o olho. Para ficar explícito, segue o código do teste anterior com um comentário para cada fase:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      it
```

```
        "puts an element at the top of the stack" do
```

```
# setup
```

```
stack =
```

```
Stack
```

```
.new
```

```
# exercise
```

```
stack.push(
```

```
1
```

```
)
```

```
stack.push(
```

```
2
```

```
)
```

```
# verify
```

```
expect(stack.top).to eq(
```

2

)

end

end end

Nem sempre é necessário escrevermos a fase de teardown, pois a sua necessidade depende do teste sendo escrito. Um exemplo de teste que tem a fase de teardown é o que escrevemos quando vimos o after hook:

```
after(:all) do
```

```
# teardown
```

```
FileUtils.rm(Dir.glob("#{cache_dir}/*")) end
```

it

"caches the result in a file" do

expect {

exercise

my_cool_object.run

}.to change {

File.exists?(cache_path) }.from(false).to(true

)

verify with change matcher end

Só para ficar claro, não é necessário escrever um comentário explicando cada fase do teste, mas colocar uma quebra de linha entre cada fase vai ajudar bastante o leitor a entender a estrutura do seu teste visualmente. Compare um teste com e sem essas quebras de linha e reflita sobre qual é mais claro em relação às fases do teste:

sem quebra de linha

it

"puts an element at the top of the stack" do

stack =

Stack

.new

stack.push(

1

)

stack.push(

2

)

expect(stack.top).to eq(

2) end

com quebra de linha

it

"puts an element at the top of the stack" do


```
stack =  
Stack  
.new
```

```
stack.push(  
1  
)  
stack.push(  
2  
)
```

```
expect(stack.top).to eq(  
2) end
```

Estruturar seu teste seguindo as quatro fases dos testes xUnit vai ajudar o leitor a entender a relação de causa e consequência do seu teste de modo mais eficiente.

Usando um example group por método

Antes de começarmos a falar sobre como organizar seus example groups, você

precisa primeiro saber o que é um example group do RSpec. Então senta que lá vem história.

Como vimos na seção O que é BDD?, o BDD propõe uma mudança na nomenclatura original do TDD para que seja mais fácil de entender a real motivação da prática de Test-Driven Development. Em vez de teste, um outro termo muito usado no mundo do BDD é a palavra "exemplo".

Ao escrever um teste para o seu código, você está na verdade escrevendo um exemplo de como o seu código pode ser usado. Como às vezes somente um exemplo não é o bastante, você escreve vários exemplos. Se você seguir a filosofia do BDD, seus exemplos podem servir até como parte da documentação do seu código.

Ter testes como parte da documentação pode ser muito útil, por exemplo no caso em que você está escrevendo uma biblioteca. Se você usar BDD para escrever sua biblioteca, o desenvolvedor que for usá-la pode usar como referência não só a documentação da API da sua lib, mas também exemplos reais de código olhando os testes dela.

Pois bem, o RSpec utiliza essa nomenclatura de exemplos na sua estrutura interna. Você pode confirmar isso entendendo o retorno dos métodos `it` e `describe` do RSpec. O que o método `it` retorna na verdade é um objeto da classe `RSpec::Core::Example`, um objeto que representa um exemplo de uso do seu código, o teste em si. Quando você precisa organizar esses vários examples dentro de um grupo que faça sentido, você usa o método `describe`, que retorna uma classe chamada `RSpec::Core::ExampleGroup`, que representa um grupo de examples.

Agora que já sabemos de onde vêm os nomes `Example` e `ExampleGroup` que o

RSpec utiliza normalmente, vamos olhar uma convenção de como organizar example groups dos seus testes. Na seção Aprendendo a estrutura básica de um teste com RSpec nós escrevemos um teste de pilha do seguinte modo:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      it
```

```
        "puts an element at the top of the stack" do
```

```
          stack =
```

```
            Stack
```

```
              .new
```

```
                stack.push(
```

```
                  1
```

```
                )
```

```
                stack.push(
```

```
                  2
```

```
)
```

```
    expect(stack.top).to eq(
```

```
    2
```

```
)
```

```
end
```

```
end end
```

Perceba como usamos o método `describe` para agrupar os testes relacionados ao método `Stack#push`:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      # testes relacionados ao método Stack#push
```

end end

Agrupar os testes de um método de instância utilizando o describe desse modo é uma convenção muito usada para organizar seus testes em um example group por método. A mesma convenção pode ser usada para agrupar os testes de um método de classe dentro de um example group com o nome desse método:

```
RSpec.describe Stack do
```

```
  describe
```

```
    ".limit" do
```

```
      # testes relacionados ao método Stack.limit
```

```
    end end
```

Como essa organização é apenas uma convenção, você não é obrigado a usá-la. Mas cada vez mais essa convenção vem se tornando uma boa prática na comunidade, pois ela facilita ao leitor dos testes encontrar todos os testes

relacionados a um determinado método, algo que é muito útil se ele está querendo entender o comportamento de um método em específico.

■

Entendendo as notações `Class#instance_method` e `Class.class_method`

Em Ruby, existem notações muito usadas para se referenciar a um método de instância ou de classe, que são `Class#instance_method` e `Class.class_method`. Para métodos de instância você referencia utilizando o caractere `#` entre o nome da classe e o nome do método:

`Class#instance_method`

No caso da nossa pilha, temos um método de instância chamado `push`, portanto podemos nos referenciar a ele com a notação `Stack#push`. Para métodos de classe você referencia utilizando o caractere `.` entre o nome da classe e o nome do método:

`Class.class_method`

No caso da nossa pilha, ela tem um método de classe chamado `new`. Podemos nos referenciar a ele com a notação `Stack.new`.

■

Usando subject e let para organizar SUT e objetos colaboradores

Agora vamos olhar como você pode usar os métodos `subject` e `let` do RSpec para organizar seus testes. Mas antes disso, como ainda não vimos nenhum desses dois métodos, vamos primeiro olhar cada um isoladamente.

O RSpec oferece um método chamado `subject` para definirmos o SUT (system under test, o objeto sendo testado). Para entendermos como ele pode ser usado, segue um exemplo de código:

```
RSpec.describe Array, "with some elements" do
```

```
  subject(
    :array) { [1,2,3
  ] }
```

```
  it
```

```
    "has the prescribed elements" do
```

```
      expect(array).to eq [
        1,2,3
      ]
```

end end

Nesse teste, nós usamos o subject para definir um método chamado array, que, quando executado, retorna o array [1, 2, 3], que é SUT do nosso teste.

Perceba que você poderia reescrever o código usando um before hook:

```
RSpec.describe Array, "with some elements" do
```

```
  before
```

```
  do
```

```
    @array = [1,2,3
```

```
    ]
```

```
  end
```

```
    it
```


"has the prescribed elements" do

```
    expect(  
      @array).to eq [1,2,3  
    ]
```

end end

No entanto, o uso do método `subject` nesse caso é mais indicado, já que ele deixa explícito que o objeto sendo manipulado é o SUT.

A diferença entre usar o `subject` e um `before hook` para o `setup` do SUT, é que o `subject` foi feito para manipular o SUT. Já o `before hook` serve para fazer o `setup` do teste como um `todo`, ou seja, é mais genérico.

Agora que já vimos uma apresentação do `subject`, vamos falar do `let`.

O `let` serve para definir um `helper method` (método auxiliar) para o seu teste. Vamos começar a entender o `let` analisando o seguinte exemplo:

```
RSpec.describe Game do
```

```
let(  
:ui) { TwitterUi.new("sandbox_username"  
  
,  
  
"sandbox_password") } end
```

Esse código cria um método chamado ui que, quando chamado, vai retornar um objeto da classe TwitterUi. Esse método pode ser usado dentro de um teste, como a seguir:

```
RSpec.describe Game do
```

```
  before
```

```
    do
```

```
      @game = Game
```

```
        .new
```

```
    end
```

```
let(  
:ui) { TwitterUi.new("sandbox_username"  
  
"sandbox_password"  
)} }
```

```
it  
"congratulates the player "  
\
```

```
"when the player hits the target" do
```

```
@game.ui = ui # chamando o método definido pelo let
```

```
@game  
.player_hits_target
```

```
expect(  
  @game.output).to include("Congratulations!"  
)  
  
end end
```

Uma outra característica interessante do `let` é que sua execução é lazy. O bloco de código passado para ele só vai ser executado se você chamar o método definido pelo `let`. Esse comportamento é diferente do comportamento do `before hook`, que sempre executa seu bloco de código para todos os exemplos do seu escopo. Para esclarecer esse comportamento, segue um exemplo:

```
RSpec.describe "The lazy-evaluated behavior of let" do
```

```
  before {  
    @foo = "bar"  
  }
```

```
  let(  
    :broken_operation) { raise "I'm broken"  
  }
```

it

"will call the method defined by let" do

expect {

expect(

@foo).to eq("bar"

)

broken_operation

).to raise_error(

"I'm broken"

)

end

it

"won't call the method defined by let" do

expect {

expect(

```
@foo).to eq("bar"  
  
)  
  
}.not_to raise_error
```

end end

Agora que já conhecemos o `subject` e o `let`, vamos aprender um modo de usá-los para organizar nossos testes.

Como você deve saber, em Programação Orientada a Objetos, para que um objeto possa realizar sua responsabilidade, ele pode depender de outros objetos. Para essas dependências se dá o nome de colaboradores.

Já que um objeto depende de outros para realizar sua responsabilidade, uma prática muito comum é passar para o construtor de uma classe os seus colaboradores como parâmetros. Essa prática se chama injeção de dependência. Veremos um exemplo concreto dela no capítulo 10. BDD na prática: começando um projeto com BDD.

Pois bem, se um objeto depende de outros para ser construído e a etapa de construção de um objeto é a etapa mais básica no seu ciclo de vida, pode valer a pena explicitar no seu teste como é feita a construção desse objeto e quem são os seus colaboradores. Afinal, se alguém quiser entender como usar o seu código, a primeira coisa que essa pessoa deve compreender é como criar uma instância válida do seu objeto.

Em um teste com RSpec, para deixarmos explícito como criar um dado objeto e quais são seus colaboradores, podemos usar os métodos `subject` e `let`. Vamos usar um trecho de código que já vimos antes para exemplificar isso.

Imagine que você está construindo um jogo e você quer que ele se comunique com o jogador via Twitter. Você poderia ter duas classes para esse sistema, a classe `Game` e uma classe chamada `TwitterUi`, que seria responsável pela interface de escrita e leitura do Twitter. Vamos começar a escrever um teste para a classe `Game` usando `subject` para explicitar como construir um objeto `game` e o `let` para mostrar quem são os colaboradores do system under test (SUT):

```
RSpec.describe Game do
```

```
  subject(
    :game) { Game
    .new(ui) }

  let(
    :ui) { TwitterUi.new("sandbox_username"
    ,
    "sandbox_password") } end
```

Agora digamos que queiramos escrever um teste que especifique que quando o jogador acertar o alvo, o jogo deve parabenizar o jogador. Poderíamos continuar o código anterior e escrever esse teste assim:

```
RSpec.describe Game do
```

```
  subject(
    :game) { Game
    .new(ui) }

  let(
    :ui) { TwitterUi.new("sandbox_username"
    ,
```

```
    "sandbox_password"
  ) }
```

```
  it
    "congratulates the player "
    \
```

```
    "when the player hits the target" do
```

```
      game.player_hits_target
```



```
    expect(game.output).to  
    include("Congratulations!")  
  )  
  
end end
```

Perceba que, para acessar o subject, basta chamarmos o método que foi definido por ele. No caso, foi o método game.

Explicitar no teste como se cria uma instância do seu objeto e quem são os colaboradores é uma ótima forma de organizá-lo. Isso porque você já mostra para o leitor na introdução do seu teste um exemplo de algo que é essencial para usar o seu objeto, que é como construir uma instância dele.

4.6 Reúso de testes

Agora que já vimos sobre os RSpec hooks, sobre como refatorar nossos testes usando-os e também sobre como organizá-los, vamos ver um pouco sobre como reutilizar nossos testes.

O RSpec oferece uma funcionalidade bem interessante para reutilizar testes em diferentes contextos, o nome dessa funcionalidade é `shared examples`. Mas quando seria necessário reutilizar testes? Vamos ver um exemplo.

Imagine que você está desenvolvendo um sistema de publicação de conteúdo. Até então, o seu sistema só gerencia blog posts. Um requisito do seu sistema é que, se um blog post for publicado, é necessário que você salve a sua data de publicação. Sabendo disso, você começa a escrever a seguinte spec para a sua classe `BlogPost`:

```
RSpec.describe BlogPost do
```

```
  describe
```

```
    "#publish!" do
```

```
      it
```

"saves the publication date" do

blog_post =

BlogPost

.new

blog_post.publish!

today =

Time.now.strftime("%Y-%m-%d"

)

expect(blog_post.published_on).to eq(today)

end

end end

Para fazer esse teste passar, você escreve a classe BlogPost do seguinte modo:

class BlogPost

```
attr_reader :published_on
```

```
def
```

```
  publish!
```

```
    today =
```

```
    Time.now.strftime("%Y-%m-%d"
```

```
)
```

```
@published_on
```

```
= today
```

```
end end
```

Depois de um tempo, seu sistema foi sendo cada vez mais usado e você decide que agora ele cuidará também de publicação de papers acadêmicos. Um dos requisitos de publicação de papers é o mesmo da publicação de blog posts: você deve salvar a data em que o paper foi publicado. Como um bom praticante de TDD, você escreve o seguinte teste para a classe Paper:

```
RSpec.describe Paper do
```

```
  describe
```

```
    "#publish!" do
```

```
      it
```

```
        "saves the publication date" do
```

```
          paper =
```

```
            Paper
```

```
              .new
```

```
            paper.publish!
```

```
            today =
```

```
              Time.now.strftime("%Y-%m-%d"
```

```
            )
```

```
            expect(paper.published_on).to eq(today)
```

```
          end
```

```
end end
```

Para fazer esse teste passar, você escreve o código a seguir:

```
class Paper
```

```
attr_reader :published_on
```

```
def
```

```
publish!
```

```
  today =
```

```
  Time.now.strftime("%Y-%m-%d"
```

```
)
```

```
@published_on
```

```
= today
```

```
end end
```

Você para um pouco, olha as duas classes, os dois testes, e percebe que tem total duplicação entre os dois:

```
class BlogPost
```

```
attr_reader :published_on
```

```
def
```

```
publish!
```

```
  today =
```

```
  Time.now.strftime("%Y-%m-%d"
```

```
)
```

```
@published_on
```

```
= today
```

```
end end
```

```
class Paper
```

```
attr_reader :published_on
```

```
def
```

```
publish!
```

```
  today =
```

```
  Time.now.strftime("%Y-%m-%d"
```

```
)
```

```
@published_on
```

```
= today
```

```
end end
```


Para eliminar essa duplicação, você pensa em extrair um módulo que contenha as regras de negócio de um objeto publicável. Esse módulo poderia se chamar Publishable. A ideia é basicamente extrair o comportamento duplicado de lógica de publicação nas classes BlogPost e Paper para esse módulo.

Nós poderíamos escrever um teste para esse módulo do seguinte modo:

```
class PublishableObject
```

```
include Publishable end
```

```
RSpec.describe "A publishable object" do
```

```
  subject {
```

```
    PublishableObject
```

```
    .new }
```

```
  describe
```

```
"#publish!" do

  it

  "saves the publication date" do

    subject.publish!

    today =

    Time.now.strftime("%Y-%m-%d"

    )

    expect(subject.published_on).to eq(today)

  end

end end
```

Agora vamos parar um pouco para entender este teste. Primeiro, nós definimos uma classe só para ele, a `PublishableObject`, que inclui o módulo que queremos testar. Depois disso, chamamos o método `subject` sem passar para ele o nome do método que ele vai definir. Quando você usa o `subject` desse modo, você está definindo o setup do SUT, mas sem dar um nome para ele. Para usar o `subject` definido desse modo, basta chamar o método `subject` de dentro do teste, como fizemos anteriormente:

```
subject.publish!
```

Depois de analisar esses detalhes, perceba que esse teste nada mais é do que a generalização dos testes que fizemos antes para as classes BlogPost e Paper.

Para fazê-lo passar, basta escrevermos o código do módulo Publishable:

```
module Publishable
```

```
  attr_reader :published_on
```

```
  def
```

```
    publish!
```

```
      today =
```

```
      Time.now.strftime("%Y-%m-%d"
```

```
)
```

```
@published_on
```

```
= today
```

```
end end
```

Agora que já temos esse módulo pronto, podemos incluí-lo nas classes BlogPost e Paper:

```
class BlogPost
```

```
include Publishable end
```

```
class Paper
```

```
include Publishable end
```

Com isso, conseguimos reduzir a duplicação nas classes em si, mas e a duplicação nos testes? Assim como conseguimos extrair o comportamento duplicado nas nossas classes para um módulo, podemos extrair os testes

duplicados para o que o RSpec chama de shared examples.

Para fazer isso, vamos começar transformando os testes que fizemos para o módulo Publishable em shared examples. Para definir um conjunto de shared examples você deve usar o método `shared_examples_for` do módulo RSpec e passar para ele uma string nomeando esse conjunto de examples:

```
RSpec.shared_examples_for "a publishable object" do end
```

Agora, basta definir quais são os examples que pertencem a esses shared examples, passando-os dentro do bloco do `shared_examples_for`:

```
RSpec.shared_examples_for "a publishable object" do
```

```
  describe
```

```
    "#publish!" do
```

```
      it
```

```
        "saves the publication date" do
```

```
          subject.publish!
```

```
    today =  
    Time.now.strftime("%Y-%m-%d"  
    )  
    expect(subject.published_on).to eq(today)  
  
end  
  
end end
```

Para usar um conjunto de shared examples, você deve usar o método `it_behaves_like` ou o método `include_examples`, passando como argumento o nome dos shared examples. Ambos os métodos são bem parecidos e logo vamos ver a diferença entre eles. Agora vamos reescrever os testes do módulo `Publishable` utilizando os shared examples que criamos:

```
class PublishableObject  
  
include Publishable end
```

```
RSpec.describe "A publishable object" do
```

```
  subject {
```

```
    PublishableObject
```

```
  .new }
```

```
  include_examples
```

```
  "a publishable object" end
```

Pronto, isso é tudo o que é necessário para utilizarmos os shared examples que definimos. Perceba que, além de incluir os shared examples, nesse caso ainda foi necessário definir o subject. Isso porque os shared examples estão usando o subject definido pelo teste que os inclui:

```
RSpec.shared_examples_for "a publishable object" do
```

```
  describe
```

```
    "#publish!" do
```

```
      it
```

```
        "saves the publication date" do
```

```
subject.publish!
```

```
# uso do subject definido pelo teste que
```

```
# incluir estes shared examples
```

```
# (...)
```

```
end
```

```
end end
```

Na verdade, os shared examples têm acesso a tudo que é definido dentro do teste que os inclui, assim como um módulo tem acesso aos métodos da classe que o inclui.

Para finalizar a refatoração dos testes do módulo Publishable, vamos rodar o teste e ver como ficou o output do RSpec. Assumindo que esses testes estejam em um arquivo chamado `shared_examples_spec.rb`, podemos rodá-lo do seguinte

modo:

```
$ rspec --format documentation shared_examples_spec.rb
```

A publishable object

```
#publish!
```

saves the publication date

Perceba como os testes que definimos nos shared examples "a publishable object" foram incluídos diretamente no output do example group que fizemos para testar o módulo Publishable.

Agora só falta utilizarmos esses shared examples nos testes das classes BlogPost e Paper. Fazer isso é bem simples, basta utilizar o método `it_behaves_like`:

```
RSpec.describe BlogPost do
```

```
  it_behaves_like
```

```
  "a publishable object" end
```

```
RSpec.describe Paper do
```

```
it_behaves_like
```

```
"a publishable object" end
```

Perceba que para os testes dessas classes não foi necessário definirmos explicitamente o subject, como fizemos no teste do módulo Publishable. Isso porque quando você não define explicitamente o subject, o RSpec usa uma definição default. Essa definição default do subject acontece automaticamente quando o argumento passado para o método describe é uma classe. Logo, quando escrevemos o seguinte código:

```
RSpec.describe BlogPost do end
```

O RSpec na verdade define o subject de modo implícito da forma a seguir:

```
RSpec.describe BlogPost do
```

```
  subject {
```

```
    BlogPost.new } end
```

Apesar de o RSpec definir o subject de modo implícito, sempre dê preferência para defini-lo de modo explícito e dando um nome para ele. Ao fazer dessa maneira você deixará seu teste mais claro.

Por fim, vamos rodar os testes para ver que o método `it_behaves_like` gera um output um pouco diferente do método `include_examples`:

```
$ rspec --format documentation shared_examples_spec.rb
```

A publishable object

```
#publish!
```

saves the publication date

BlogPost

behaves like a publishable object

```
#publish!
```

saves the publication date

Paper

behaves like a publishable object

```
#publish!
```

saves the publication date

Perceba que, ao usar o `it_behaves_like`, ele coloca o output dos shared examples

aninhado com a frase `behaves like a publishable object`, diferentemente de quando usamos o método `include_examples`, que colocou o output dos `shared examples` diretamente, sem mostrar que eles são `shared examples`.

Agora você já sabe como reutilizar testes utilizando a funcionalidade de `shared examples` do RSpec. Essa é uma funcionalidade bem poderosa e pode ajudar bastante a reduzir testes repetitivos.

O único cuidado que você tem que ter ao utilizar essa funcionalidade é não a usar demais de modo a comprometer a clareza dos seus testes. Lembre-se, ao escrever testes, a clareza é mais importante que o DRY.

Pontos-chaves deste capítulo

Neste capítulo aprendemos sobre como refatorar, organizar e reutilizar nossos testes.

Começamos aprendendo sobre os RSpec hooks e como eles podem nos ajudar a reduzir duplicação nos nossos testes. Lembre-se, ao usar hooks, evite comprometer a clareza em favor da redução de duplicação.

Aprendemos sobre o que quer dizer clareza nos testes. Um teste é claro se ele possibilita ao seu leitor entender de modo fácil a relação de causa e consequência entre as fases de `setup`, `exercise` e `verify` do teste.

Em seguida aprendemos que um teste tem uma narrativa, que são as quatro fases

do padrão xUnit: setup, exercise, verify e teardown. Assim como uma redação tem as etapas de introdução, desenvolvimento e conclusão, um teste tem as quatro fases do xUnit. Ter uma estrutura facilita a compreensão e deixa seu teste mais claro.

Outro tópico interessante que vimos sobre organização de testes foi a ideia de usar o subject e let do RSpec para explicitar o setup do seu SUT e quem são os colaboradores do seu objeto. Ter esse tipo de informação no começo do seu teste é algo mais que pode ajudar em sua clareza.

Por fim, vimos sobre como utilizar os shared examples do RSpec para reutilizar testes. A dica aqui é a mesma de sempre: ao pensar em reduzir duplicação nos testes, faça isso com cautela, clareza é mais importante do que DRY quando se está escrevendo testes.

No capítulo seguinte vamos aprender sobre RSpec Mocks, sobre o que são test doubles e que novidades o uso desse tipo de ferramenta traz para o modo como escrevemos testes.

Não saia daí!

Capítulo 5

Mocks e stubs

Nos capítulos anteriores nós aprendemos sobre a estrutura básica de um teste com RSpec e sobre como organizar e refatorar seus testes. Neste último capítulo focado somente em RSpec, você aprenderá sobre como utilizar o RSpec Mocks.

Mock é uma ferramenta poderosa para fazer testes. O seu uso definiu toda uma escola de praticantes de TDD, a "London school of TDD" (escola londrina de TDD), ou também conhecido por mockists, termo cunhado por Martin Fowler (2007) no famoso artigo Mocks aren't Stubs.

Neste capítulo veremos algumas vantagens de usar mocks, como isso pode influenciar no design do nosso código e qual a diferença entre mocks e stubs.

Let's mock?

5.1 Por que preciso de mocks?

Antes de começarmos a falar sobre mocks de modo conceitual e como RSpec nos oferece esse tipo de ferramenta, vamos ver um exemplo real em que mocks ajudam bastante a escrever testes.

Imagine que você é um grande fã de Douglas Adams e da sua série de livros "O Guia do Mochileiro das Galáxias". Como tal, você já deve ter ouvido falar do Deep Thought, o computador que conseguiu processar a pergunta final sobre a vida, o universo e tudo mais... E a resposta é 42. Você decide, então, fazer um programa simples que simule o Deep Thought.

A especificação desse programa é que ele deve imprimir na tela a tão famosa resposta, ou seja, ele deve imprimir na tela o número 42. Você escreve a seguinte estrutura de teste para especificar esse comportamento:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" end
```

A fase de setup desse teste é bem simples, basta criarmos uma instância do

DeepThought:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

```
      computer =
```

```
        DeepThought
```

```
        .new
```

```
      end end
```

A fase de exercise desse teste também parece simples, basta chamarmos o método que imprime a resposta que queremos:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

end end

Mas e sobre a fase de verify, como podemos verificar que o objeto computer imprimiu 42 na tela? Uma ideia seria checar no objeto que representa a tela se o número 42 foi impresso. Mas isso pode ser meio estranho. Para ficar claro como isso pode ser estranho, vamos escrever uma possível implementação da classe DeepThought e analisar como poderíamos verificar o comportamento esperado no teste.

Uma possível implementação dessa classe é a seguinte:

```
class DeepThought
```

def

```
print_the_answer
```

puts

```
"42"
```

```
end end
```

Perceba que, para imprimir a resposta na tela, nós estamos usando o método `puts`, que faz parte do módulo `Kernel` do Ruby. O que o método `Kernel#puts` faz é imprimir uma string no `STDOUT`. Se fôssemos seguir o estilo de teste que temos feito até agora, poderíamos tentar ler o que foi escrito no `STDOUT` e verificar se o número 42 está lá dentro. Seria algo assim:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

```
      computer =
```

```
        DeepThought
```

```
          .new
```

```
      computer.print_the_answer
```

```
expect(  
STDOUT.read).to include("42"  
)
```

end end

Acontece que não é possível ler do STDOUT. A seguinte exception é lançada quando você tenta enviar a mensagem read para o objeto STDOUT:

```
STDOUT.read # => IOError: not opened for reading
```

Se não é possível testar checando se a string foi de fato impressa, como podemos verificar o comportamento esperado? Antes de partir para a resposta em si, vamos entender qual o estilo de teste que temos feito até agora, que é o estilo baseado em verificação de estado.

Teste com verificação de estado

Como sabemos, um teste do padrão xUnit possui a fase de verificação. Até então, todos os nossos testes seguiram um padrão na fase de verificação. O nome desse padrão é teste com verificação de estado.

Desde o início estamos utilizando esse padrão sem saber exatamente como ele

funciona, então vamos analisar um exemplo para esclarecer seu funcionamento.

Olhemos o teste da pilha que fizemos no capítulo Primeiros passos com RSpec e Cucumber:

```
RSpec.describe Stack do
```

```
  describe
```

```
    "#push" do
```

```
      it
```

```
        "puts an element at the top of the stack" do
```

```
          stack =
```

```
            Stack
```

```
              .new
```

```
                stack.push(
```

```
                  1
```

```
                )
```

```
                stack.push(
```

2

)

expect(stack.top).to eq(

2

)

end

end end

As fases do teste anterior são:

setup: instanciamos um objeto da classe Stack;

exercise: enviamos a mensagem push duas vezes para o objeto que criamos na fase de setup, ou seja, interagimos com o SUT (system under test);

verify: checamos o estado final do objeto sendo testado, isto é, checamos se o estado da pilha consiste em ter no topo o número 2.

Teste com verificação de estado é o modo como fizemos o teste anterior. Para verificar se o comportamento final era o comportamento esperado, nós verificamos o estado do SUT. Nesse caso, verificamos o estado do objeto stack.

Acontece que nem sempre observar o estado final do SUT é a melhor maneira de fazer a verificação do nosso teste. Existe um outro jeito de fazer isso, que é a verificação de comportamento.

Teste com verificação de comportamento

Alan Key, um dos pais da Orientação a Objetos, costumava falar que uma das chaves da POO é pensar mais na comunicação entre os objetos e menos nos objetos em si. Então, o comportamento de um objeto é mais baseado nas interações que ele faz com outros objetos, e menos nos dados que ele tem dentro do seu estado interno. A ideia de mocks e verificação por comportamento segue totalmente essa filosofia do Alan Key, vamos ver como.

Vamos começar voltando ao exemplo do teste que começamos a escrever para a classe DeepThought. O teste que escrevemos até agora está mais ou menos assim:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

```
      computer =
```

```
        DeepThought
```

```
.new
```

```
computer.print_the_answer
```

```
end end
```

A dúvida que temos sobre esse teste é como verificar o comportamento esperado. Seguindo a ideia de que o comportamento de um objeto é baseado nas suas interações com outros objetos, poderíamos verificar se o nosso objeto `computer` desse teste está interagindo corretamente com os outros objetos de que ele depende. Acontece que até agora ele não depende explicitamente de ninguém, no entanto, implicitamente ele depende do `STDOUT`, pois ele utiliza o método `puts`.

O que podemos fazer para melhorar isso é deixar essa dependência explícita, isto é, passar um objeto como dependência do objeto `computer` e verificar se o `computer` está interagindo corretamente com esse objeto.

O papel que essa dependência do `computer` tem é o de um objeto impressor, pois sua responsabilidade é imprimir a string que é passada para ele. Para criar essa dependência no nosso teste, vamos usar um método chamado `double` (dublê) que faz parte do `RSpec::Mocks`:

```
printer = double("printer")
```

O método `double` retorna um objeto vazio, o qual podemos programar para

responder aos métodos que quisermos e também podemos programá-lo para que ele verifique se uma determinada mensagem foi enviada para ele. Nas seções seguintes vamos explicá-lo melhor. Por enquanto, vamos deixar simples e apenas entender como ele pode nos ajudar no teste da classe DeepThought.

Usaremos o nosso double para passar como dependência explícita de um objeto da classe DeepThought:

```
RSpec.describe DeepThought do

  it

  "prints the answer to the ultimate question" do

    printer = double(
      "printer"
    )

    computer =
      DeepThought
        .new(printer)

  end end
```

Agora vamos programar o nosso double para que ele verifique se o objeto computer está interagindo corretamente com ele. O comportamento que queremos verificar é que, quando o método DeepThought#print_the_answer for chamado, a mensagem print deve ser enviada para a dependência printer com o valor "42". Podemos fazer isso do seguinte modo com o RSpec:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

```
      printer = double(
```

```
        "printer"
```

```
      )
```

```
      computer =
```

```
        DeepThought
```

```
          .new(printer)
```

```
    # programa o objeto printer para verificar se o método print
```

será chamado nesse objeto com o argumento "42" durante o

teste

```
    expect(printer).to receive(  
      :print).with("42"  
    )
```

```
    computer.print_the_answer
```

end end

Pare um pouco para analisar o código anterior e tente entender como está sendo feita a fase de verificação do teste.

O que pode parecer estranho à primeira vista é que a fase de verificação do teste está sendo declarada antes da fase de exercise:

```
it "prints the answer to the ultimate question" do
```

```
# setup
```

```
printer = double(  
  "printer"  
)
```

```
computer =  
  DeepThought  
    .new(printer)
```

```
# programa a verificação que será feita ao final do teste
```

```
expect(printer).to receive(  
  :print).with("42"  
)
```

```
# exercise
```

```
computer.print_the_answer
```

```
end
```

Para continuar entendendo como o RSpec vai fazer a verificação anterior, vamos escrever parte da implementação da classe DeepThought e ver o teste quebrar:

```
class DeepThought
```

```
  def
```

```
    initialize(printer)
```

```
  @printer
```

```
  = printer
```

```
end
```

```
  def
```

```
    print_the_answer
```

end end

Ao rodar o teste com a implementação dessa classe, ele vai quebrar com a seguinte mensagem de erro:

Failures:

1) DeepThought prints the answer to the ultimate question

Failure/Error: expect(printer).to receive(:print).with("42")

(Double "printer").print("42")

expected: 1 time with arguments: ("42")

received: 0 times

Vamos interpretar essa mensagem de erro. O que o RSpec fez foi checar ao final do teste se a mensagem print foi enviada ao objeto printer com o argumento "42". O RSpec está nos dizendo que era esperado que essa mensagem fosse enviada uma vez, mas não foi enviada nenhuma vez.

O que fizemos foi mockar o objeto printer. Um objeto que verifica automaticamente se uma ou mais mensagens foram enviadas para ele é chamado de mock object.

Para fazer esse teste passar, basta implementarmos o comportamento esperado e fazer com que a mensagem print seja enviada para a dependência printer quando o método DeepThought#print_the_answer for chamado:

```
class DeepThought
```

```
def
```

```
  initialize(printer)
```

```
  @printer
```

```
  = printer
```

```
end
```

```
def
```

```
  print_the_answer
```

```
    @printer.print("42")
```

)

end end

Com essa implementação, nosso teste finalmente passa.

O que fizemos nesse teste foi escrever a fase de verificação usando verificação de comportamento, ou seja, ela consistiu basicamente em ver se o SUT interagiu corretamente com seus colaboradores.

Esse tipo de verificação no RSpec é chamado de mock expectations. Veremos mock expectations em detalhes na seção Escrevendo mock expectations com RSpec.

Um ponto interessante para notar nesse exemplo é como o uso de mocks pode afetar o design dos nossos objetos. Nesse caso, saímos de uma implementação inicial como:

```
class DeepThought
```

```
def
```

```
  print_the_answer
```


puts

"42"

end end

para

class DeepThought

def

initialize(printer)

@printer

= printer

end

```
def
```

```
print_the_answer
```

```
@printer.print("42"
```

```
)
```

```
end end
```

Compare as duas implementações. A vantagem do design resultado do uso de mock nesse exemplo é que ele nos forçou a deixar explícita a dependência do DeepThought sobre um objeto com papel de printer. Ao deixar isso explícito, o nosso design fica mais extensível, mais maleável. Com esse design poderíamos usar diferentes tipos de printers.

Poderíamos ter um printer default, que só imprime para o STDOUT:

```
class CliPrinter
```

```
def
```

```
print(text)
```

STDOUT

.puts(text)

end end

printer =

CliPrinter

.new

computer =

DeepThought.new(printer)

Poderíamos ter também um printer que envia a mensagem por e-mail, sem ter que mudar nada na classe DeepThought:

class EmailPrinter

def

print(text)

 deliver_email(text)

end end

```
printer =  
EmailPrinter.new(to: "john.doe@gmail.com"  
)  
computer =  
DeepThought.new(printer)
```

Usar TDD com mocks para o desenvolvimento da classe DeepThought nos forçou (orientou) a chegar nesse design. Essa é uma das vantagens do uso de mocks: não só eles nos deixam testar nossos objetos baseado no comportamento deles (na interação deles com outros objetos), como também podem nos guiar para design com baixo acoplamento. Isso porque usar mocks normalmente nos força a passar as dependências de modo explícito para o construtor da nossa classe. Você pode ver um estudo intensivo do uso de mocks desse modo no famoso livro *Growing Object-Oriented Software, Guided by Tests* (FREEMAN; PRYCE, 2009).

5.2 Conhecendo o conceito de test doubles

Agora que você já tem uma ideia inicial sobre o que são mock objects, onde usá-los e por que eles são úteis, vamos seguir em frente e aprender sobre o mundo maior que abriga o conceito de mock objects — vamos aprender sobre o que são test doubles.

Test double é um objeto utilizado durante um teste para substituir um colaborador do SUT. Isso pode ser necessário em vários cenários, como:

instanciar um colaborador do SUT é muito complicado, então você usa um test double em seu lugar;

instanciar um colaborador real do SUT pode trazer consequências ruins (por exemplo, um objeto que envia e-mail para usuários de produção durante os testes), então você usa um test double no lugar;

você quer testar seu objeto fazendo verificação de comportamento, em vez de verificação de estado.

Existem vários tipos de test doubles, mock object é um deles. Outros tipos que existem são: stub, spy, fake object e dummy object. O RSpec implementa diretamente três deles, o stub, o mock e o spy. Neste livro vamos falar do mock e do stub.

5.3 Usando stubs com RSpec

Na seção Por que preciso de mocks? vimos uma introdução sobre mock objects. Nesta seção aprenderemos sobre outro tipo de test doubles, os stubs. Veremos o que eles são, para que servem e como usá-los com o RSpec.

Como nós sabemos, um objeto depende de outros objetos para realizar suas tarefas. No contexto do teste de um objeto, nós precisamos que o ambiente em volta do SUT esteja em um determinado estado para podermos verificar o comportamento esperado. Nessa configuração do ambiente pode ser necessário que o objeto colaborador se comporte de um modo específico, e é aí que entram os test stubs.

Um stub nada mais é do que forçar uma resposta específica para um determinado método de um objeto colaborador. Para ficar mais claro, vamos ver um exemplo.

Imagine que você está desenvolvendo um sistema financeiro. Uma das funcionalidades desse sistema é informar se é possível ou não fazer um empréstimo para um determinado cliente. Você imagina que teremos uma classe chamada LoanChecker (checador de empréstimo) que será responsável por dizer se é possível fazer um empréstimo para um determinado cliente. Só deve ser possível fazer empréstimo para um cliente se ele não tem nenhuma dívida no sistema.

Para especificar esse comportamento, começamos a escrever a seguinte estrutura de teste:

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt"
```

```
    end end
```

Vamos começar a pensar na fase de setup desse teste. A parte principal dessa fase é setar o estado que o cliente não tem nenhuma dívida no sistema.

Nesse sistema, além de termos a classe LoanChecker, temos também a classe Client, que representa um cliente. Essa classe tem um método chamado Client#has_debt?, que retorna true caso o client tenha alguma dívida, e false caso contrário.

No setup do teste, para configurar que um objeto client não tem débito, temos pelo menos duas opções:

Instanciar um objeto client no nosso teste e mudar o seu estado interno de modo que ele fique sem débito;

Fazer um stub no método Client#has_debt? para retornar false.

Como estamos falando de stub, usaremos a opção 2. Para forçarmos que um objeto retorne um certo valor para um método, vamos usar o método allow do RSpec, que serve para fazer stub de um método:

```
client = Client
```

```
.new
```

```
allow(client).to receive(
```

```
:has_debt?).and_return(false)
```

Vamos entender um pouco melhor essa sintaxe. O que fizemos foi permitir que o objeto client receba a mensagem has_debt? e retorne false quando a receber. Simples assim. Agora podemos usar isso no setup do nosso teste:

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```


it

"returns true when the client has no debt" do

setup

client =

Client

.new

allow(client).to receive(

:has_debt?).and_return(false

)

loan_checker =

LoanChecker

.new

exercise

loan_evaluation = loan_checker.can_lend_to?(client)

```
end
```

```
end end
```

Para finalizar nosso teste só falta a fase de verify. Para isso, basta checarmos que a variável `loan_evaluation` (avaliação do empréstimo) é `true`:

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt" do
```

```
          # setup
```

```
            client =
```

Client

.new

allow(client).to receive(
:has_debt?).and_return(false
)

loan_checker =

LoanChecker

.new

exercise

loan_evaluation = loan_checker.can_lend_to?(client)

verify

expect(loan_evaluation).to be_truthy

end

end end

Pronto, é desse modo que se usa stub. Ele é usado para a fase de setup do seu teste. O stub serve para você forçar que um objeto responda a uma mensagem do modo que você precisa. Vamos agora ver outras funcionalidades que o stub do RSpec nos oferece.

Stub com raise

No nosso exemplo anterior, nós testamos um comportamento do objeto `loan_checker`. Esse objeto depende de um colaborador, o `client`. Imagine que, para o `client` processar a mensagem `has_debt?`, ele precise acessar diversos outros sistemas, e quando algum desses sistemas não estiver no ar, ele levanta uma exceção. Vale a pena especificar qual o comportamento esperado do `loan_checker` nesse cenário.

Para fazer o setup desse cenário, podemos fazer um stub no objeto `client` para que quando ele receber a mensagem `has_debt?`, ele levante uma exceção:

```
client = Client
```

```
.new
```

```
allow(client).to receive(
```

```
:has_debt?).and_raise(RuntimeError)
```

Com esse setup, o teste desse cenário ficaria assim:

```
it "returns nil when it's not possible "
```

```
\
```

```
"to check the client's debt" do
```

```
  client =
```

```
    Client
```

```
  .new
```

```
    allow(client).to receive(
```

```
      :has_debt?).and_raise(RuntimeError
```

```
)
```

```
    loan_checker =
```

```
      LoanChecker
```

```
    .new
```

```
    loan_evaluation = loan_checker.can_lend_to?(client)
```

```
expect(loan_evaluation).to be_nil
```

```
end
```

Stub com restrições de argumentos

É possível também fazer um stub de um método somente quando ele receber uma certa configuração de argumentos. Por exemplo, imagine que quiséssemos fazer stub do método `Client#has_debt?` para retornar `true` sobre dívidas no ano de 2012 e `false` sobre dívidas no ano de 2011. Podemos fazer isso do seguinte modo com o RSpec:

```
allow(client)
```

```
  .to receive(
```

```
    :has_debt?).with(in: 2012
```

```
)
```

```
  .and_return(
```

```
    true
```

```
)
```

```
allow(client)
```

```
  .to receive(
```

```
    :has_debt?).with(in: 2011
```

```
)
```

```
.and_return(  
false)
```

Existem vários outros modos de fazer stub com restrição de argumentos. Para conhecê-los, vale a pena você dar uma olhada na documentação do RSpec: <https://relishapp.com/rspec/rspec-mocks/v/3-8/docs/basics>.

Stub de qualquer instância de uma classe

Dependendo do design do seu sistema, às vezes você pode precisar fazer stub não só em um objeto específico, mas sim em qualquer instância de uma classe. Você pode fazer isso usando o método `allow_any_instance_of` do RSpec da seguinte maneira:

```
allow_any_instance_of(Client  
  
)  
  .to receive(  
    :has_debt?  
  )  
  .and_return(  
    false  
  )
```

com o stub acima, qualquer instância que você fizer da classe # Client vai responder a mensagem has_debt? com o retorno false

```
client_instance =
```

```
  Client
```

```
  .new
```

```
another_client_instance =
```

```
  Client
```

```
  .new
```

```
expect(client_instance.has_debt?).to be_falsey
```

```
expect(another_client_instance.has_debt?).to be_falsey
```

Você deve usar essa funcionalidade com cuidado pois o seu uso pode ser um sinal de que algo pode melhorar no seu código. Vamos analisar um exemplo em que existe uma solução melhor que usar o `allow_any_instance_of`.

Imagine que você está fazendo um sistema de previsão do tempo. Esse sistema tem duas classes, a classe `Weather` (tempo) e uma classe `ApiClient` que serve para consultar um web service público na Web sobre previsão de tempo. Uma funcionalidade do seu sistema será informar a condição do tempo do dia corrente. Se a condição do tempo no web service for "sunny" (ensolarado), então o seu sistema deve retornar que o tempo será "good" (bom).

Até então o código do seu sistema está escrito da seguinte forma:

```
class Weather
```

```
  def
```

```
    today
```

```
      weather_api_client =
```

```
        ApiClient
```

```
      .new
```

```
    case
```

```
      weather_api_client.today
```

```
    when "sunny"
```

```
      "good"
```

end

end end

Para especificar esse código, você escreveu o seguinte teste:

```
RSpec.describe Weather do
```

```
  describe
```

```
    "#today" do
```

```
      it
```

```
        "returns 'good' for a sunny day" do
```

```
          allow_any_instance_of(
```

```
            ApiClient
```

```
          )
```

```
            .to receive(
```

```
              :today
```

```
)  
    .and_return(  
      "sunny"  
    )  
  
    weather =  
      Weather  
        .new  
  
        expect(weather.today).to eq(  
          "good"  
        )  
  
      end  
  
    end end
```

Esse é um exemplo em que o uso do `allow_any_instance_of` é desnecessário e indica que o design pode melhorar. Vamos entender por quê.

Quando um objeto depende de outro, é uma boa prática tornar essa dependência explícita. Caso a dependência não esteja explícita, fica difícil de trocá-la durante a execução do seu código. Imagine se no exemplo anterior nós quiséssemos trocar a dependência da classe `ApiClient` por uma outra biblioteca de consulta do web service de previsão do tempo. Para fazer isso seria necessário mudar o código do método `Weather#today`.

Uma boa prática em Orientação a Objetos é fazer o seu design de modo que seu objeto seja extensível sem que seja necessário modificar o seu código. Um exemplo de extensão seria passar uma implementação diferente de um colaborador. Para que isso seja possível, podemos fazer uma injeção de dependência, passando o colaborador como parâmetro do método construtor da sua classe. No caso da classe `Weather`, essa ideia ficaria assim:

```
class Weather
```

```
  def
```

```
    initialize(api_client)
```

```
    @weather_api_client
```

```
    = api_client
```

```
  end
```

```
def
```

```
  today
```

```
  case @weather_api_client
```

```
    .today
```

```
    when "sunny"
```

```
      "good"
```

```
    end
```

```
  end end
```

Ao mudar a implementação para usar injeção de dependência, não é mais necessário fazer stub em qualquer instância da classe ApiClient, já que podemos fazer um stub somente no objeto passado para o construtor do Weather:

```
RSpec.describe Weather do
```

```
  describe
```

```
    "#today" do
```

```
      it
```

```
        "returns 'good' for a sunny day" do
```

```
          api_client =
```

```
            ApiClient
```

```
            .new
```

```
            allow(api_client).to receive(
```

```
              :today).and_return("sunny"
```

```
            )
```

```
            weather =
```

```
              Weather
```

```
              .new(api_client)
```

```
              expect(weather.today).to eq(
```

```
                "good"
```

)

end

end end

Esse é um caso em que o uso de `allow_any_instance_of` indicou uma possível melhoria no design do seu código. Se você tiver controle sobre uma classe, sempre pense se não ficaria melhor fazer injeção de dependência. Seu design fica mais extensível e a pessoa que for dar manutenção no seu código vai agradecer.

5.4 Escrevendo mock expectations com RSpec

Na seção Por que preciso de mocks?, vimos uma introdução ao uso de mocks e que vantagens essa prática nos traz. Nesta seção veremos um pouco mais sobre mocks, mas agora focando nos modos que o RSpec nos oferece para utilizá-los. Antes disso, no entanto, vamos entender qual a relação entre os mocks e as expectations do RSpec.

Como vimos, mockar um objeto quer dizer programá-lo para verificar se ele vai receber uma certa mensagem durante o teste. No RSpec, essa construção se chama mock expectation:

```
printer = CliPrinter
```

```
.new
```

```
expect(printer).to receive(
```

```
:print)
```

Você deve se lembrar de que uma expectation no RSpec é a construção básica utilizada na fase de verify do nosso teste. Perceba como a sintaxe de mocking é semelhante à sintaxe de expectation — ambas usam o mesmo método expect:


```
# expectation comum
```

```
expect(stack.top).to eq(  
  2  
)
```

```
# mock expectation
```

```
expect(printer).to receive(  
  :print)
```

A única diferença entre essas duas expectations é que a expectation comum recebe um matcher e a mock expectation recebe o retorno do método receive. As duas expectations não são tão semelhantes à toa, na verdade o RSpec trata ambas do mesmo modo. Isso porque o método receive na verdade retorna um matcher, assim como o método eq. Ou seja, mock expectations também usam matchers que seguem o protocolo de matchers que aprendemos na seção Entendendo o protocolo interno de matcher do RSpec.

Agora que você já sabe que por debaixo dos panos os mocks do RSpec não são nada mais do que expectations, deve ficar ainda mais claro para você que os mocks servem para a fase de verify do teste, assim como qualquer outra expectation.

Mock com raise

Mockar um objeto serve para verificar a interação do SUT com esse objeto. Dependendo do seu teste, você pode precisar configurar mais aspectos dessa interação. Uma das coisas de que você pode precisar é que essa interação levante uma exceção. Você pode fazer isso usando o método `and_raise` do RSpec:

```
expect(bomb).to receive(:explode!).and_raise(RuntimeError)
```

O teste que tiver essa mock expectation só vai passar se o objeto `bomb` receber a mensagem `explode!`. Além disso, quando esse objeto receber essa mensagem, ele vai levantar uma exception da classe `RuntimeError`, seguindo a configuração que fizemos.

Mock com restrições de argumentos

Ao configurar uma mock expectation, você pode precisar verificar se o seu SUT está enviando a mensagem correta com os argumentos certos para o seu colaborador. Você pode fazer isso usando o método `with` do RSpec:

```
printer = double("printer")
```

```
expect(printer).to receive(
```

```
:print).with("42")
```

O teste que tiver essa mock expectation só vai passar se a mensagem print for enviada para o objeto printer com o argumento "42".

Veja a seguir mais alguns exemplos de mock com restrições de argumentos:

```
expect(obj)
  .to receive(
    :message
  )
  .with(
    "more_than", "one_argument"
  )
expect(obj).to receive(
  :message
).with(anything())
expect(obj).to receive(
  :message).with(an_instance_of(Money
))
expect(obj).to receive(
```

```
:message).with(hash_including(:a => "b"))
```

Existem vários outros modos de você configurar restrições de argumentos para seus mocks. Para ver a lista inteira, acesse a documentação do RSpec: <https://relishapp.com/rspec/rspec-mocks/v/3-8/docs/setting-constraints/matching-arguments>.

Mock com contagem de mensagens

Se existir algum cenário no seu teste que é importante verificar quantas vezes o SUT envia uma certa mensagem para o seu colaborador, você pode fazer isso do seguinte modo com o RSpec:

```
expect(obj).to receive(:message  
  
).once  
  
expect(obj).to receive(  
  
:message  
  
).twice  
  
expect(obj).to receive(  
  
:message).exactly(3  
  
).times
```

```
expect(obj).to receive(  
  :message).at_least(:once  
)
```

```
expect(obj).to receive(  
  :message).at_least(:twice  
)
```

```
expect(obj).to receive(  
  :message  
)  
.at_least(n).times
```

```
expect(obj).to receive(  
  :message).at_most(:once  
)
```

```
expect(obj).to receive(  
  :message).at_most(:twice  
)
```

```
expect(obj).to receive(  
  :message).at_most(n).times
```

Mock com return value

Outro aspecto que você pode configurar no seu mock object é fazê-lo retornar um valor quando receber uma certa mensagem. Isso pode ser feito usando o método `and_return` do RSpec:

```
expect(bomb).to receive(:status).and_return("turned off")
```

Poder setar o return value de um objeto assim não é muito parecido ou igual ao que o stub faz? Devo então usar mock ou stub para setar o return value?

Lembre-se sempre: poder fazer algo não quer dizer que você deve fazer isso. Só porque o RSpec nos permite setar o return value de um mock object, não quer dizer que devemos usar essa funcionalidade toda hora. Na verdade, você deve usar essa funcionalidade com atenção. Vamos falar na seção seguinte o porquê disso.

5.5 Quando usar mock e quando usar stub

Muitas pessoas confundem o propósito de mocks e stubs, e usam mocks quando deveriam usar stubs. De modo bem resumido, stub serve para setar o estado inicial do seu teste, ou seja, é utilizado na fase de setup. Mock serve para configurar a verificação de que uma mensagem deve ser enviada para um objeto colaborador, serve para a fase de verify. Vamos ver alguns exemplos para entender melhor.

Na seção Usando stubs com RSpec nós escrevemos o seguinte teste para uma classe gerenciadora de empréstimos:

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt" do
```

```
# setup
```

```
client =
```

```
Client
```

```
.new
```

```
allow(client).to receive(
```

```
:has_debt?).and_return(false
```

```
)
```

```
loan_checker =
```

```
LoanChecker
```

```
.new
```

```
# exercise
```

```
loan_evaluation = loan_checker.can_lend_to?(client)
```

```
# verify
```

```
expect(loan_evaluation).to be_truthy
```


end

end end

O teste anterior era para especificar que o empréstimo pode ser feito quando o cliente não tem nenhuma dívida no sistema. Releia a seguinte parte dessa frase: "quando o cliente não tem nenhuma dívida no sistema". Essa frase indica o estado inicial necessário do nosso teste. Para configurar esse estado, usamos um stub para que o colaborador client retornasse false para o método has_debt?.

Na fase de setup do teste anterior usamos stub no seguinte trecho:

```
allow(client).to receive(:has_debt?).and_return(false)
```

No entanto, poderíamos ter usado mock para fazer esse setup:

```
expect(client).to receive(:has_debt?).and_return(false)
```

Ao usar mock nesse caso, nosso teste continuaria passando e ele ficaria assim:

```
RSpec.describe LoanChecker, "using mocks for setup" do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt" do
```

```
          # setup
```

```
            client =
```

```
              Client
```

```
                .new
```

```
                  expect(client).to receive(
```

```
                    :has_debt?).and_return(false
```

```
                  )
```

```
                    loan_checker =
```

```
                      LoanChecker
```

```
                        .new
```

```
# exercise
```

```
loan_evaluation = loan_checker.can_lend_to?(client)
```

```
# verify
```

```
expect(loan_evaluation).to be_truthy
```

```
end
```

```
end end
```

Apesar de o teste ter continuado passando ao usarmos mock para fazer o setup, usamos mocks para o propósito errado. Mock não foi feito para setup, foi feito para verify. É nesse ponto em que você deve ficar atento quando usar o método `and_return` em um mock object, como fizemos em:

```
client = Client
```

```
.new
```

```
expect(client).to receive(  
:has_debt?).and_return(false)
```

Quando você usa o método `and_return` em um mock object, é porque você precisa que o seu mock retorne algum valor para uma certa mensagem. Se você estiver usando isso muitas vezes, pode ser porque você está usando mocks para a fase de setup, ou seja, você está usando mocks errado. Na fase de setup, use stubs.

Vamos agora rever o exemplo de uso de mocks que fizemos na seção Por que preciso de mocks?. Nessa seção fizemos o seguinte teste:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

```
      printer = double(  
        "printer"
```

```
      )
```

```
      computer =
```

```
        DeepThought
```

```
          .new(printer)
```

```
    expect(printer).to receive(  
      :print).with("42"  
    )
```

```
    computer.print_the_answer
```

```
  end end
```

Esse teste especifica que o computador DeepThought imprime a resposta para a pergunta fundamental da vida. Para verificarmos o comportamento esperado, usamos um mock para checar que essa resposta está sendo impressa. Fizemos isso do seguinte modo:

```
expect(printer).to receive(:print).with("42")
```

Perceba que o uso do mock nesse exemplo está totalmente relacionado com especificação do comportamento esperado, não com o setup do teste. Mocks servem para você verificar o comportamento do teste através da verificação da interação correta entre o SUT e seus colaboradores. É para isso que você deve usar mocks. Stubs são para setup, mocks são para verificação.

5.6 Usando o método double para fazer testes isolados

Na maioria dos testes que fizemos até então, quando o SUT tinha algum colaborador, utilizamos a implementação real desse colaborador no nosso teste. Acontece que esse não é único modo de escrever testes. Você pode substituir um colaborador durante um teste por um test double puro, sem a necessidade de instanciar uma implementação real do colaborador. Esse tipo de teste se chama teste isolado. Vamos ver como fazer isso.

Você deve lembrar do seguinte teste que fizemos na seção Usando stubs com RSpec:

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt" do
```

```
          client =
```

Client

```
.new  
  
  allow(client).to receive(  
:has_debt?).and_return(false  
)  
  
  loan_checker =
```

LoanChecker

```
.new  
  
  loan_evaluation = loan_checker.can_lend_to?(client)  
  
  expect(loan_evaluation).to be_truthy
```

end

end end

No teste anterior, o SUT é uma instância da classe LoanChecker e o colaborador é uma instância da classe Client. Em vez de utilizarmos uma instância real da classe Client, poderíamos ter usado um double puro para substituí-lo usando o método double do RSpec:

```
client = double("client"  
  
)  
  
allow(client).to receive(  
  
:has_debt?).and_return(false)
```

Ao usar esse double puro, nosso teste continuaria passando, pois estamos fazendo stub do único método desse colaborador que é usado durante esse teste, o método `has_debt?`.

O método `double` recebe como argumento opcional uma string que serve para nomear o seu double com o papel que ele representa em relação ao SUT. Como esse argumento é opcional, você pode criar um double sem passá-lo:

```
client = double  
  
allow(client).to receive(  
  
:has_debt?).and_return(false)
```

Apesar de esse argumento ser opcional, é uma boa prática sempre passar essa string para o método `double`. Primeiro porque, se o teste falhar devido a alguma interação errada com esse double, o nome que você deu pra ele vai ser usado na mensagem de erro do teste, facilitando o entendimento do erro.

O segundo motivo para você dar um nome para seus doubles é porque, ao fazer isso, você está informando ao leitor do teste qual o papel desse double em relação ao SUT. Para entender melhor essa questão de "papel do double", vamos ver um exemplo.

Na seção Por que preciso de mocks? nós vimos o seguinte teste usando um double puro:

```
RSpec.describe DeepThought do

  it
  "prints the answer to the ultimate question" do

    printer = double(
  "printer"
)

    computer =
  DeepThought
.new(printer)

    expect(printer).to receive(
  :print).with("42"
```

)

```
computer.print_the_answer
```

end end

O double no teste representa um colaborador que tem o papel de um objeto impressor. Ou seja, o DeepThought depende de um objeto que tem como responsabilidade imprimir texto.

Perceba que, ao definir o papel de um objeto colaborador, não estamos falando como ele faz a sua responsabilidade, mas sim o que ele faz. Ao fazer isso estamos diminuindo semanticamente o acoplamento entre o DeepThought e seu colaborador. Vamos entender melhor isso.

No ambiente de produção, provavelmente vamos usar como colaborador para representar esse papel um objeto que imprime texto na linha de comando, que poderia ser um objeto de uma classe CliPrinter (command line interface printer). Apesar de em produção o DeepThought usar o CliPrinter, o DeepThought não está acoplado diretamente com essa classe em si, mas sim com o que essa classe faz. Ele está acoplado com o papel do seu colaborador.

Podemos fazer testes isolados para representar que um objeto está acoplado ao papel de seu colaborador, não com a classe dele.

Em um teste isolado, nós mockamos o papel de um colaborador:

```
printer = double("printer"  
)
```

```
computer =
```

```
DeepThought
```

```
.new(printer)
```

```
expect(printer).to receive(  
:print).with("42")
```

Já em um teste não isolado, você usa a implementação real de um colaborador:

```
cli_printer = CliPrinter
```

```
.new
```

```
computer =
```

```
DeepThought.new(cli_printer)
```

O fato de você fazer um teste de modo isolado, mockando um colaborador, faz você pensar mais na interação que o SUT faz com o colaborador e menos na instância do colaborador em si. Esse tipo de pensamento leva a um design com menos acoplamento e de fácil extensão.

No nosso exemplo, podemos constatar isso pelo fato de que se fosse necessário parar de imprimir o texto na linha de comando e começar a imprimir o texto via Twitter, bastaria passar outro objeto para o DeepThought que também tivesse o papel de printer, mas que fizesse isso pelo Twitter:

```
printer = TwitterPrinter  
  
.new  
  
computer =  
  
DeepThought  
  
.new(printer)  
  
computer.print_the_answer
```

Usando o método as_null_object

Quando você está fazendo um teste isolado, o seu SUT pode depender de mais de um método do seu colaborador. Imagine o seguinte exemplo de código:

```
class DeepThought
```

```
def
```

```
initialize(printer)
```

```
@printer
```

```
= printer
```

```
@printer.print("Hello, I'm the DeepThought super computer"
```

```
)
```

```
end
```

```
def
```

```
print_the_answer
```

```
@printer.print("42"
```

```
)
```

```
end end
```

Agora imagine que você quer testar o comportamento de impressão da resposta

para a pergunta fundamental da vida. Vamos usar o teste que tínhamos feito anteriormente para isso:

```
RSpec.describe DeepThought do
```

```
  it
```

```
    "prints the answer to the ultimate question" do
```

```
      printer = double(
```

```
        "printer"
```

```
      )
```

```
      computer =
```

```
        DeepThought
```

```
          .new(printer)
```

```
      expect(printer).to receive(
```

```
        :print).with("42"
```

```
      )
```

```
      computer.print_the_answer
```

end end

Ao rodar esse teste, ele quebra com a seguinte mensagem:

Failures:

1) DeepThought prints the answer to the ultimate question

Failure/Error:

```
@printer.print("Hello, I'm the DeepThought super computer")
```

```
#<Double "printer"> received unexpected message :print
```

```
with ("Hello, I'm the DeepThought super computer")
```

O teste quebra pois print com o argumento "Hello, I'm the DeepThought super computer" é uma mensagem inesperada. Para fazer esse teste passar, poderíamos fazer um mock expectation configurando nosso mock object para receber essa mensagem:

```
printer = double("printer"
```

```
)
```

```
hello_message =
```

```
"Hello, I'm the DeepThought super computer"
```

```
expect(printer).to receive(  
:print).with(hello_message)
```

No entanto, como esse teste não é sobre o comportamento de impressão da mensagem de hello, não faz sentido usarmos mock para fazer a verificação dessa interação com o colaborador. Um stub é mais apropriado para isso, já que o que queremos nesse teste é ignorar o envio da mensagem `printer.print("Hello, I'm the DeepThought super computer")`:

```
printer = double("printer"  
)
```

```
hello_message =  
"Hello, I'm the DeepThought super computer"
```

```
allow(printer).to receive(  
:print).with(hello_message)
```

Usar stub para ignorar que uma mensagem é enviada para um colaborador faz sentido, mas o RSpec nos oferece uma ferramenta ainda melhor para isso: o método `as_null_object`. Quando você chama o método `as_null_object` em um `double`, ele vai ignorar qualquer mensagem que for enviada para ele, menos as que você mockar explicitamente. Poderíamos usá-lo para reescrever nosso

double do seguinte modo:

```
printer = double("printer").as_null_object
```

Nosso teste ficaria assim:

it "prints the answer to the ultimate question" do

```
  printer = double(
```

```
    "printer"
```

```
  ).as_null_object
```

```
    computer =
```

```
      DeepThought
```

```
    ).new(printer)
```

```
    expect(printer).to receive(
```

```
      :print).with("42"
```

```
    )
```

```
    computer.print_the_answer
```

end

Perceba como, ao usar o `as_null_object`, não foi mais necessário fazermos um stub específico para a mensagem que queremos ignorar.

Usar o método `as_null_object` é muito útil quando você quer focar somente na verificação de algumas interações entre o SUT e colaborador. Portanto, se você estiver fazendo um teste isolado, e quer ignorar todas as mensagens que seu colaborador recebe menos aquelas que são foco do seu teste, use o método `as_null_object`.

5.7 Verified doubles

Um recurso novo, introduzido no RSpec 3, ajuda a verificar que os stubs que estamos criando condizem com os métodos que uma implementação real contém. Imagine o seguinte cenário: você está criando um double da classe Client, que usamos anteriormente:

```
class Client
```

```
  def
```

```
    has_debt?
```

```
    # ...
```

```
  end end
```

Mas, por acidente, ao criar o stub você digita o nome do método errado (repare que o ? está faltando):

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt" do
```

```
          client = double(
```

```
            "client"
```

```
          )
```

```
          allow(client).to receive(
```

```
            :has_debt).and_return(false
```

```
          )
```

```
          loan_checker =
```

```
            LoanChecker
```

```
              .new
```

```
            loan_evaluation = loan_checker.can_lend_to?(client)
```

```
expect(loan_evaluation).to be_truthy
```

```
end
```

```
end end
```

O erro que o RSpec vai nos fornecer não é tão informativo:

Failures:

1) LoanChecker#can_lend_to? returns true when the client
has no debt

Failure/Error:

```
#<Double "client"> received unexpected message :has_debt?  
with (no args)
```

Nesse caso, como a classe Client já existe, podemos usar o `instance_double` do RSpec, que tem a vantagem de verificar automaticamente se os métodos dos quais fizemos stub realmente existem:

```
RSpec.describe LoanChecker do
```

```
  describe
```

```
    "#can_lend_to?" do
```

```
      it
```

```
        "returns true when the client has no debt" do
```

```
          client = instance_double(
```

```
            Client
```

```
          )
```

```
          allow(client).to receive(
```

```
            :has_debt).and_return(false
```

```
          )
```

```
          loan_checker =
```

```
            LoanChecker
```

```
              .new
```

```
            loan_evaluation = loan_checker.can_lend_to?(client)
```

```
expect(loan_evaluation).to be_truthy
```

```
end
```

```
end end
```

Com essa pequena alteração, o RSpec já informa com precisão onde ocorreu o problema:

Failures:

1) LoanChecker#can_lend_to? returns true when the client has
no debt

Failure/Error:

```
allow(client).to receive(:has_debt).and_return(false)
```

the Client class does not implement the instance

method: has_debt

Isso nos permite construir uma suíte mais resistente, que aponta as próprias

falhas. Podemos saber, por exemplo, se uma refatoração removeu um método que esperávamos que ainda existisse.

Pontos-chaves deste capítulo

Este capítulo foi o último focado somente em RSpec. Aprendemos sobre o que são mock objects, para que servem nos nossos testes e como o RSpec usa esse conceito para montar mock expectations.

Vimos também sobre stubs e como você pode usá-los em seus testes com RSpec. Vimos a diferença entre stubs e mocks — stubs são para a fase de setup, mocks são para a fase de verify. Também vimos como configurar o RSpec para que ele verifique automaticamente se o stub que criamos corresponde aos métodos de uma implementação real daquele objeto.

Por fim, vimos o que são testes isolados e como fazer esse tipo de teste utilizando o double do RSpec. Outro ponto importante que aprendemos foi como a prática desse tipo de teste usando mocks nos leva a pensar mais no papel dos nossos colaboradores, mais na interação entre os nossos objetos e menos no como nossos objetos realizam suas responsabilidades.

Apesar de você ter finalizado o último capítulo de RSpec, ainda falta mais uma ferramenta para você poder praticar BDD por completo, falta aprendermos sobre Cucumber. E é isso que veremos nos próximos capítulos.

Capítulo 6

Conhecendo o Cucumber

Na seção O que é BDD? nós vimos rapidamente que BDD é uma abordagem de desenvolvimento de software que prega que o desenvolvedor deve ir construindo e "descobrimo" o seu software e suas APIs utilizando testes para especificá-lo, começando pelas camadas mais externas e depois as mais internas. Esse tipo de abordagem se chama outside-in development.

Uma das maiores vantagens de outside-in development é que, ao começar a desenvolver o software por uma camada externa, você vai especificando o que a camada interna precisa implementar de acordo com a necessidade da camada externa. Ou seja, você só implementa aquilo que é realmente necessário para a camada externa e guiado pelas necessidades dela. No mundo Ruby, uma das ferramentas que nos ajuda a fazer outside-in development é o Cucumber.

O Cucumber é uma ferramenta que une especificação funcional e automatização de testes de aceitação. Ao ter o primeiro contato com o Cucumber, as pessoas pensam que ele é uma ferramenta de teste, e somente isso. No entanto, esse entendimento está errado. Ele é uma ferramenta de especificação e também uma ferramenta de automatização de testes.

A partir deste capítulo nós estudaremos o Cucumber: como usá-lo, por que usá-lo, quando é melhor usá-lo e as melhores práticas de uso. Siga em frente e se prepare para colocar mais um item na sua valiosa caixa de ferramentas.

6.1 Por que usar Cucumber?

Se você já tentou usar Cucumber alguma vez ou conversou com alguém que já o usou ou tentou usá-lo, você provavelmente ouviu falar que ele é legal para fazer testes de aceitação da sua aplicação. Esse é o motivo mais fácil de entender e o primeiro motivo pelo qual alguém começa a usá-lo. Acontece que, se você for usar Cucumber somente para automatizar seus testes de aceitação, você provavelmente vai achar custoso demais usá-lo e vai desistir. Isso quer dizer que você provavelmente o utilizou errado ou simplesmente ele não é útil para o tipo de aplicação que você está construindo.

O uso correto do Cucumber traz valor nos seguintes pontos:

Ele estimula a conversa com o PO (product owner) ou pessoa de negócios de modo que você entenda melhor os requisitos do seu sistema;

Ele une a especificação do seu software junto com testes automatizados, fazendo com que sua documentação não fique desatualizada em relação ao comportamento real do seu sistema;

Com o passar do tempo, a especificação de testes feita com seu uso se torna uma das principais referências do comportamento do seu sistema, tão relevante quanto o próprio código do software em si, porém mais fácil de ser lida. Isso é o que chamamos de Living Documentation (documentação viva).

Vamos falar um pouco mais sobre esses tópicos.

Cucumber como estímulo de conversa sobre os requisitos

Ao aprender sobre desenvolvimento com metodologias ágeis, uma das primeiras coisas que conhecemos é o modo ágil de capturar requisitos funcionais, a famosa User Story. Segue um exemplo de uma user story de um jogo da forca:

Jogador começa um jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Figura 6.1: Exemplo de user story

Perceba como não existem muitos detalhes na user story acima. O "documento" em si é apenas um lembrete de uma conversa que você deve ter no futuro com o PO ou equivalente sobre a especificação dessa história. Mas quando finalmente você conversar com o PO, o que você vai falar?

O que deve ser conversado para especificar uma história é o seu critério de aceite. Uma técnica útil para descobrir e definir o critério de aceite de uma história é pensar em exemplos de uso. Vamos entender isso um pouco melhor utilizando o exemplo da user story anterior.

O software da qual essa user story faz parte é um jogo da forca. O começo de um jogo da forca tem os seguintes passos:

O jogador pede para iniciar um novo jogo;

O jogo mostra para o jogador uma mensagem de início do jogo;

O jogo pergunta quantas letras deve ter a palavra a ser sorteada para adivinhação;

O jogador informa quantas letras deve ter a palavra que ele quer adivinhar;

O jogo sorteia uma palavra com o tamanho pedido.

Note que esse processo pode ter mais de um cenário. O primeiro cenário seria o

de sucesso, o jogador começa o jogo pedindo uma palavra de quatro letras e o jogo a sorteia com sucesso.

Outro cenário seria o jogador iniciar o jogo e pedir para sortear uma palavra com 100 letras. Nesse caso o jogo não consegue sortear uma palavra com esse tamanho pois ela não existe.

Perceba que ao pensar em alguns exemplos de como o jogador poderia usar essa funcionalidade nós melhoramos nosso entendimento sobre seu funcionamento. Imagine agora que a listagem desses exemplos pode ser documentada junto com o seu software e inclusive utilizada para automatizar testes de aceitação. É isso que o Cucumber oferece.

Para cada funcionalidade, o Cucumber "pede" para que você liste cenários de uso. Como faz parte da natureza do Cucumber ter documentadas funcionalidades com exemplos de uso, ele estimula você a pensar nesses cenários e conversar com as pessoas que podem ajudar a defini-los.

Funcionalidade: 

Cenário: 

Step a
Step b
Step c
Step d

Cenário: 

Step a
Step b
Step c
Step d

Figura 6.2: Estrutura de uma especificação com Cucumber

Essa técnica de especificar requisitos funcionais utilizando como base exemplos de uso se chama Specification by Example (especificação por exemplos) (ADZIC, 2011).

Por fim, vamos ver como ficaria a especificação da história anterior utilizando Cucumber:

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Cenário: Sorteio da palavra com sucesso

Após o jogador começar o jogo, ele deve escolher o tamanho

da palavra a ser adivinhada. Ao escolher o tamanho, o jogo sorteia a palavra e mostra na tela um

" _ "

para cada letra

que a palavra sorteada tem.

Cenário: Sorteio da palavra sem sucesso

Se o jogador pedir para o jogo sortear uma palavra com um tamanho que o jogo não tem disponível, o jogador deve ser avisado disso e o jogo deve pedir para o jogador sortear outra palavra.

Cucumber: especificação e testes juntos!

Uma das primeiras etapas do processo cascata de desenvolvimento de software é a especificação dos requisitos funcionais. Tradicionalmente nessa etapa os analistas de negócio documentam todos os detalhes de todas as funcionalidades a serem desenvolvidas no sistema. Acontece que esse processo tem problemas.

No mundo de desenvolvimento de business software, os requisitos podem mudar ao longo do tempo. A consequência disso é que o software que é desenvolvido não é o mesmo que foi especificado. A documentação feita inicialmente não é atualizada ao longo das mudanças feitas no software e ela se torna obsoleta.

Para resolver esse problema é necessário mudar um pouco a abordagem de especificação de software. No livro *Specification by Example* (ADZIC, 2011), Gojko Adzic comenta sobre o tipo de documentação ideal em equipes ágeis. Essa documentação deve ser:

precisa e testável;

escrita segundo um processo puxado (just-in-time);

de fácil manutenção.

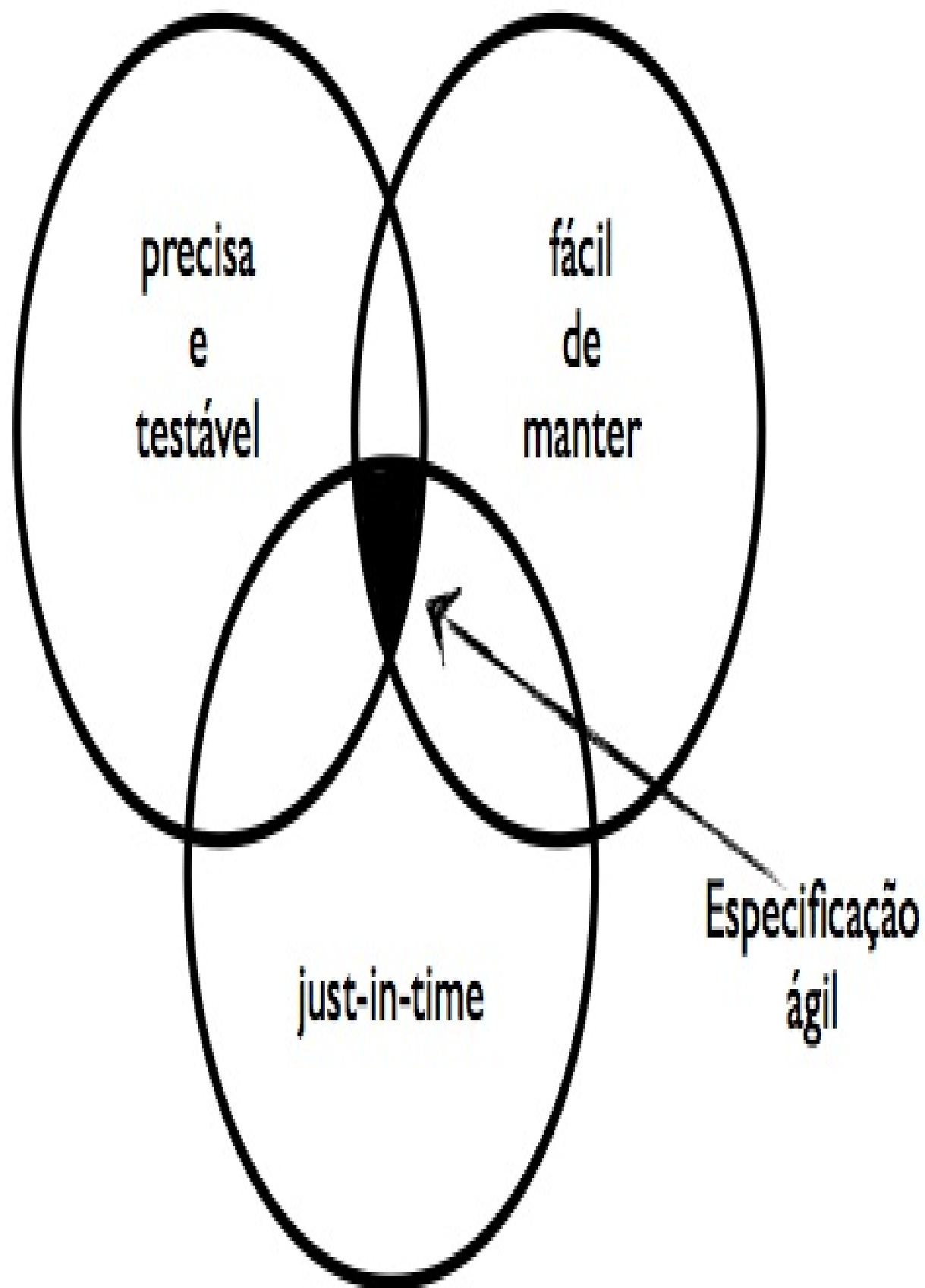


Figura 6.3: Especificação ágil

O Cucumber fornece a base necessária para construirmos esse tipo de especificação. Ao ter a especificação do seu software junto com testes que automatizam essa especificação, você ganha o benefício de diminuir bastante as chances de a sua documentação ficar obsoleta, pois, se seu software fugir do comportamento especificado, a sua especificação vai quebrar. Isso é o que chamamos de Especificação Executável.

Cucumber para construir uma documentação viva

Ao utilizar o Cucumber para especificar e testar seu software, você tem como resultado uma especificação executável. Ao conjunto das especificações executáveis do seu software se dá o nome de documentação viva (living documentation). Ter uma documentação viva no seu projeto tem uma série de benefícios. Vamos ver alguns deles.

O primeiro benefício é que, como essa documentação é executável, ela não fica desatualizada em relação ao comportamento real do software. Por outro lado, em situações em que você tem uma documentação tradicional desatualizada, as únicas pessoas que podem dizer o que realmente o software faz são aquelas que conseguem ler o código do software. O código não mente — em última instância ele é a fonte autoritativa sobre o comportamento real do software.

O problema de ter que recorrer ao código do software para saber o que ele faz é que nem todo mundo tem habilidades para fazer isso. Analistas de negócio, POs,

scrum masters, testers e outros provavelmente não conseguirão tirar suas dúvidas sobre o comportamento do software lendo o seu código. A documentação viva resolve esse problema pois ela se torna uma fonte autoritativa sobre o comportamento do sistema tão boa quanto o código em si, mas bem mais fácil de ser lida.

Existem situações em que até mesmo para desenvolvedores fica difícil de entender o que um software faz apenas lendo seu código. Isso acontece, por exemplo, em projetos de manutenção de sistemas legados. Nesse tipo de projeto é muito comum você entrar com sua equipe para melhorar a qualidade interna do sistema, mas, para fazer isso, você precisa primeiro entender o que o software faz. O problema é que provavelmente os desenvolvedores originais do sistema, os únicos capazes de entender o código com eficiência, já deixaram o projeto e você não tem acesso a eles.

Nesses cenários você precisa fazer uma engenharia reversa dos requisitos funcionais do sistema, partindo da leitura do código para conseguir gerar a especificação das funcionalidades. Esse processo poderia ser bem menos custoso se você tivesse uma documentação viva para ler. Ela estaria escrita em linguagem natural (inglês, português etc.), não em linguagem de programação e você poderia confiar nela porque ela seria composta de um conjunto de especificações executáveis.

Outro cenário em que a documentação viva pode ajudar é na relação de confiança entre os desenvolvedores e os testers. Alguma vez já aconteceu de você achar que o tester está fazendo testes desnecessários porque ele fica refazendo na mão os testes que você já fez de modo automatizado? Comigo já aconteceu bastante. Esse tipo de problema acontece porque os testes automatizados que você escreveu não são visíveis para o tester, então ele teria que confiar no que você fala para ele, não no que ele vê. Imagine se o tester pudesse ver os testes automatizados em um formato e linguagens acessíveis. Esse formato pode existir, se você escrever seus testes de forma a ter uma documentação viva.

A documentação viva não deve viver apenas no seu repositório de código. Você deve publicá-la de alguma forma visível a todos os interessados: PO, testers, analistas de negócio etc. Existem algumas ferramentas com esse objetivo:

Jam (<https://cucumber.io/jam/>), um serviço para publicar as documentações que o Cucumber gera. Ele é um substituto do descontinuado Relish, que ainda hospeda algumas documentações baseadas em Cucumber de gems como o RSpec Expectations (<https://relishapp.com/rspec/rspec-expectations/docs/>);

YARD Cucumber (<https://github.com/burtlo/yard-cucumber/>), uma extensão do YARD (ferramenta em Ruby para documentações) que inclui os testes feitos em Cucumber na documentação gerada.

O Cucumber também oferece uma configuração para gerar um relatório em HTML ao executar a suíte de testes. Basta executar:

```
$ cucumber --format html --out cucumber.html
```

Que será gerado uma versão HTML do resultado da execução da suíte:

Cucumber Features

1 scenarios (1 passed)

2 steps (2 passed)

Finished in 0m0.059s seconds

[Collapse All](#) [Expand All](#)

language: pt

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

Bem-vindo ao jogo da forca!

Figura 6.4: Relatório do Cucumber em HTML

Essa é uma parte importante do processo, pois adotar uma ferramenta adicional para escrever testes adiciona complexidade. Esse é um investimento vantajoso quando a documentação gerada é usada pelas pessoas envolvidas no projeto como um meio de entender os comportamentos do sistema, então vale a pena trabalhar em formas de tornar essa documentação mais acessível.

Quando essa documentação não é adotada uma alternativa é usar o próprio RSpec para escrever os testes de aceitação.

6.2 Visão geral de Cucumber

Até então já aprendemos que o Cucumber tem duas facetas, uma de documentação e outra de testes. A organização dos arquivos de uma especificação com Cucumber também segue essa estrutura, alguns arquivos servem mais para o propósito de documentação, enquanto outros servem mais para o propósito de teste.

Uma especificação executável com Cucumber tem a seguinte estrutura:

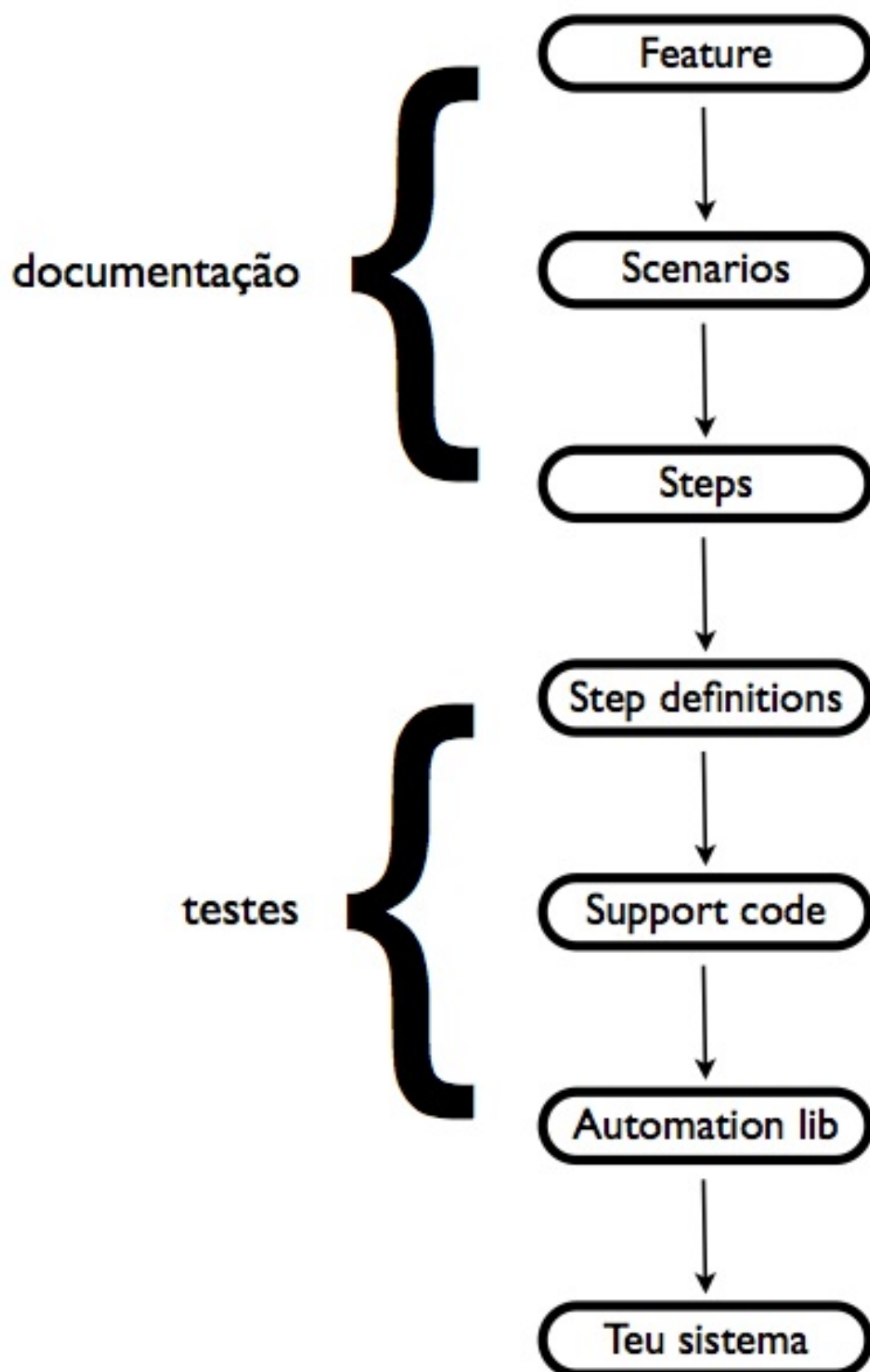


Figura 6.5: Estrutura de uma especificação executável com Cucumber

Vamos entender melhor o que vai em cada camada dessa estrutura.

Primeiro, vale a pena notar que existem dois grupos de camadas. O primeiro tem como objetivo a documentação e especificação do seu software. O segundo tem como objetivo os testes automatizados. Vamos começar pela primeira parte do grupo de documentação, a Feature (funcionalidade).

Feature

Uma Feature do Cucumber serve para especificar e testar uma funcionalidade do seu sistema. Para ficar mais claro o que é uma feature, segue o exemplo que desenvolvemos na seção Olá Cucumber:

```
# language: pt
```

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

Bem-vindo ao jogo da forca!

Essas features devem ser escritas utilizando um padrão chamado Gherkin. O Gherkin na verdade é uma linguagem em si, com gramática e tudo mais. Apesar da restrição de escrever suas features seguindo o Gherkin, você pode e deve escrevê-las utilizando linguagem natural, focando no uso dos termos do domínio do seu sistema.

Como vimos anteriormente, uma feature é composta por vários cenários, que por sua vez é composto por vários steps. Vamos dar uma olhada nessas outras partes.

Scenario

Uma funcionalidade tem escopo e critérios de aceite. Para ajudar a entender esses critérios de aceite nós usamos exemplos de uso dessa feature. É para isso que serve o Scenario (cenário) do Cucumber, para definirmos os caminhos mais importantes que o nosso software pode tomar em relação a uma feature.

Segue um exemplo de cenário:

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

""""

Bem-vindo ao jogo da forca!

""""

Sua funcionalidade pode ter vários cenários através da variação do estado inicial do sistema ou da ação que você toma nele. Cabe à equipe definir os principais cenários de modo que você tenha uma funcionalidade bem documentada e testada.

Steps

Um cenário no Cucumber é composto por vários steps (passos). Os steps servem para descrever o comportamento do seu sistema em um determinado cenário. Um step começa com as seguintes palavras reservadas do Gherkin: "Dado", "Quando", "Então", "E" e "Mas". Segue um exemplo de steps para uma funcionalidade de soma de uma calculadora:

Dado que iniciei a minha calculadora

Quando eu faço a soma

"1 + 2"

Então eu obtenho o resultado

"3"

Step definitions

As três camadas que vimos até então (feature, scenarios, steps) são escritas em linguagem natural usando o Gherkin. Você tem a liberdade de escrevê-las desse modo porque o texto deve ser expressivo o bastante para servir como documentação do seu software. As próximas três camadas são escritas na linguagem de programação que você está usando, no nosso caso, Ruby. Vamos começar falando das step definitions.

O step definition é a camada que mapeia aquilo que você escreve nos seus steps (Gherkin) com as ações que serão tomadas no seu sistema. Um step definition para o step Então eu obtenho o resultado "3" poderia ser, por exemplo:

```
Então('eu obtenho o resultado "{int}"') do
```

```
|result|
```

```
# ... end
```

O mapeamento entre um step definition e um step se dá através do match de expressão Cucumber, uma string com alguns marcadores especiais para as variáveis. No exemplo anterior, o marcador {int} vai capturar o valor 3 na variável result. Quando o Cucumber executa uma feature que você escreveu, ele vai lendo cada step, e para cada um ele procura o primeiro match entre ele e um step definition do seu conjunto. Quando ele encontra esse match, ele executa esse step definition.

Support code

Um step definition pode falar direto com a sua aplicação, mas na maioria das vezes isso não é uma boa ideia. Imagine que você tem dois step definitions que fazem algo bem parecido, para onde você vai extrair esse comportamento comum? Uma das responsabilidades do support code é essa, conter o comportamento repetido entre step definitions.

O support code deve ser uma camada entre os step definitions e sua aplicação. Um modo de entendê-lo é como uma DSL (domain specific language) da sua aplicação para os step definitions. Vamos ver um exemplo.

Imagine que você está fazendo uma aplicação web e que você tem um step definition para login no seu sistema. Esse step definition poderia ser assim:

```
Quando("logo no sistema com o username {string}") do
```

```
  |username|
```

```
    visit
```

```
    "/login"
```

```
      fill_in
```

```
        "Username", with:
```

```
          username
```

```
            click_button
```

```
    "Login" end
```

Ao utilizar uma camada de support code, esse comportamento todo de interação com o sistema ficaria dentro do support code. No step definition fica o que você quer fazer, no support code fica a forma como você quer fazer. Ao extrair o "como" para o support code, seu step definition ficaria assim:

```
Quando("logo no sistema com o username {string}") do  
  |username|  
    login_with(username)  
end
```

Automation library

Para que o seu support code consiga interagir com sua aplicação, muitas das vezes você vai querer fazer essa interação utilizando a UI (user interface) do seu sistema. Para isso você necessitará de uma biblioteca que saiba interagir com essa interface, essa é a camada de automation library.

Em aplicações web você pode usar, por exemplo, o Capybara (<https://github.com/teamcapybara/capybara>). Já em aplicação de linha de comando (CLI) você pode usar o Aruba (<https://github.com/cucumber/aruba>).

Pontos-chaves deste capítulo

Neste capítulo começamos a aprender Cucumber. O Cucumber será a nossa ferramenta para fecharmos o ciclo de outside-in development do BDD.

Aprendemos que Cucumber não é uma ferramenta de testes. Cucumber é primeiro uma ferramenta de especificação de requisitos e depois uma ferramenta de automatização de testes de aceitação. Quando usada somente para testes, ele

perde seu valor e você deixa de ganhar todos os outros benefícios que ele pode trazer.

Aprendemos também que uma das maiores vantagens do uso do Cucumber é o estímulo que ele dá para as conversas sobre os requisitos, fazendo com que capturemos nossos requisitos na forma de especificação com exemplos. Outra grande vantagem é a construção de uma documentação viva, um artefato que pode se tornar uma fonte autoritativa sobre o comportamento do sistema tão boa quanto o código, porém mais fácil de ser lida.

Por fim, conhecemos a estrutura geral de uma especificação com Cucumber. Descobrimos que para escrever tal especificação usamos o "padrão" Gherkin para a faceta de documentação e o Ruby para a faceta de automatização de testes.

Nos capítulos a seguir vamos aprender em detalhes cada uma das camadas de uma especificação executável com Cucumber, olhando as funcionalidades que ele nos provê, as boas práticas e também as más.

Capítulo 7

Especificando funcionalidades com Cucumber

Neste capítulo, conheceremos em detalhes a camada mais externa de uma especificação executável com Cucumber, a camada que serve para documentar o software sendo especificado.

Começaremos aprendendo como instalar o Cucumber e conhecendo sua estrutura de diretórios. Em seguida, aprenderemos a estrutura de uma especificação com Cucumber, ou seja, a estrutura da linguagem Gherkin.

Por fim, veremos como escrever cenários e as diversas funcionalidades que o Cucumber nos oferece para escrevê-los de um modo expressivo.

7.1 Instalando e fazendo setup do Cucumber

Antes de começarmos a escrever uma especificação com Cucumber, precisamos primeiro instalá-lo. Como já vimos antes, basta usarmos o RubyGems para isso:

```
$ gem install cucumber
```

Com o Cucumber instalado, vamos criar um diretório para o nosso projeto e entrar nesse diretório:

```
$ mkdir learning-cucumber
```

```
$ cd learning-cucumber
```

Algo legal de saber é que o próprio comando do Cucumber nos avisa o que é necessário para começar um projeto. Rode o comando cucumber no seu console e veja a seguinte saída:

```
$ cucumber
```

```
No such file or directory - features.
```

You can use `cucumber --init` to get started.

Agora utilize o comando init do Cucumber para criar a estrutura de diretórios e os arquivos padrão:

```
$ cucumber --init
```

```
create features
```

```
create features/step_definitions
```

```
create features/support
```

```
create features/support/env.rb
```

Os arquivos que ficam diretamente dentro do diretório features são as nossas especificações. Além das especificações, ainda existem mais dois diretórios dentro de features, o step_definitions e o support — mas vamos falar desses dois só nos capítulos seguintes, por enquanto vamos colocar nossa atenção no diretório features.

Rode o comando cucumber novamente e confirme que o setup está feito ao visualizar a seguinte saída:

```
$ cucumber
```

0 scenarios

0 steps

0m0.000s

Agora que você já tem o Cucumber instalado e a estrutura de diretórios básica feita, estamos prontos para escrever uma especificação com Cucumber.

7.2 Estrutura de uma feature com Cucumber

Vamos analisar uma especificação com Cucumber utilizando como exemplo a que já fizemos na seção Olá Cucumber:

```
# language: pt
```

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

|||||

Bem-vindo ao jogo da forca!

|||||

O Cucumber nomeia suas especificações como funcionalidades (features). Uma funcionalidade tem um título, uma descrição (também chamada de narrativa ou preamble) e um ou mais cenários.

language: pt

Funcionalidade: Começar jogo → título

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

→ descrição (narrativa)

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

|||||

Bem vindo ao jogo da forca!

|||||

→ cenário

Figura 7.1: Estrutura de uma feature do Cucumber

O título deve ser uma descrição curta da funcionalidade sendo especificada. Ele não precisa descrever com detalhes o que ela é, deve ser apenas o suficiente para que o leitor dessa especificação possa identificar qual é essa funcionalidade.

A descrição é um texto livre que você pode usar para escrever o que for necessário. No exemplo anterior nós usamos uma variante do tradicional formato Connextra para escrever user stories:

Para < conseguir realizar um certo objetivo >

Como < o papel de alguém usando o sistema >

Quero < usar uma certa funcionalidade do sistema >

O formato Connextra é um ótimo template para captar e resumir os elementos principais de uma especificação: quem está usando o sistema, qual funcionalidade e para quê.

Apesar de termos usado o formato Connextra para descrever nossa funcionalidade, você não é obrigado usá-lo e nem deve se ater somente a ele. Aliás, prender-se a esse formato é um problema muito comum em várias equipes. Ao fazer isso, a equipe está desperdiçando uma grande oportunidade de contextualizar melhor a funcionalidade sendo especificada. O formato Connextra é apenas um resumo dos principais pontos de uma funcionalidade, ele não é a

especificação por si só.

A última parte da estrutura de uma feature são os cenários. Uma funcionalidade pode ter um ou mais cenários. Na seção seguinte vamos falar mais deles. Por fim, vale a pena notar a primeira linha que tem no exemplo que usamos anteriormente:

```
# language: pt
```

Essa linha serve para dizer que estamos escrevendo uma feature com Cucumber utilizando o idioma português. Caso não tivéssemos colocado essa linha, o Cucumber tentaria interpretar essa feature utilizando o idioma padrão dele, que é inglês. A diferença básica entre utilizar um idioma ou outro são as palavras reservadas do Gherkin, por exemplo:

```
|
```

O Gherkin oferece dezenas de idiomas. Para descobrir quais são eles, você pode executar o seguinte comando:

```
$ cucumber --i18n-languages
```

af	Afrikaans	Afrikaans	
am	Armenian	հայերեն	
an	Aragonese	Aragonés	

ar	Arabic	العربية	
ast	Asturian	asturianu	
az	Azerbaijani	Azərbaycanca	
bg	Bulgarian	български	
bm	Malay	Bahasa Melayu	
...			

Para descobrir quais são as palavras reservadas do Gherkin para um certo idioma basta você executar `cucumber --i18n-keywords <sigla do idioma>`, por exemplo:

```
$ cucumber --i18n-keywords pt
```

```
$ cucumber --i18n-keywords en
```

```
$ cucumber --i18n-keywords ja
```

Quando estamos escrevendo uma especificação com Cucumber, devemos seguir um padrão, que é o Gherkin. O Gherkin é uma linguagem e, como toda linguagem, tem seus tokens, sua gramática e suas regras. Se você escrever uma feature com Cucumber e ela não estiver de acordo com o Gherkin, ela será considerada inválida, quebrará e você será avisado disso.

Vamos ver o que o Cucumber nos fala ao tentar executar uma especificação inválida. Crie o seguinte arquivo `learning-cucumber/features/gherkin_invalido.feature` e coloque o seguinte conteúdo nele:

```
# language: pt
```

Título da minha funcionalidade inválida

Um pouco de descrição quebrada

Cenário: Cenário de uma funcionalidade com gherkin inválido

Perceba que essa feature não está de acordo com o Gherkin porque ela não começa com o título da feature utilizando a palavra reservada Funcionalidade. Ao rodar o Cucumber recebemos a seguinte mensagem de erro:

```
$ cucumber
```

```
features/gherkin_invalido.feature: Parser errors:
```

```
(2:1): expected: #TagLine, #FeatureLine, #Comment, #Empty,
```

```
got 'Título da minha funcionalidade inválida'
```

```
(4:1): expected: #TagLine, #FeatureLine, #Comment, #Empty,
```

```
got 'Um pouco de descrição quebrada'
```

```
(6:1): expected: #TagLine, #FeatureLine, #Comment, #Empty,
```

```
got 'Cenário: Cenário de uma funcionalidade
```

```
  com gherkin inválido'
```

```
(7:0): unexpected end of file, expected: #TagLine, #FeatureLine,
```

```
  #Comment, #Empty
```

```
(Cucumber::Core::Gherkin::ParseError)
```

A mensagem de erro diz que tem um problema na linha 2 da feature. O problema, como já sabemos, é que não iniciamos a feature com a palavra reservada Funcionalidade. Delete o arquivo `features/gherkin_invalido.feature` para que ele não nos atrapalhe no futuro.

Outro detalhe interessante que vale a pena comentar é que o Gherkin aceita comentários, semelhante aos de linguagens de programação. Um comentário no Gherkin deve começar com o caractere `#`, como você pode ver na primeira linha do exemplo que fizemos:

```
# language: pt
```

Não é muito comum o uso de comentários nas especificações com Gherkin, mas se você achar necessário e útil para a clareza da sua spec, sintá-se à vontade para utilizá-los. Agora que já conhecemos a estrutura de uma feature com Cucumber, vamos entender a estrutura de um cenário.

■

Em qual idioma devo escrever minhas features?

Como expliquei na seção Olá Cucumber, durante todo este livro escreveremos nossas features em português. A decisão sobre escrever as features em português, em inglês ou em qualquer outra língua depende do seu contexto. Quando você for decidir em que idioma escrever suas features, pense principalmente no público dessas features, ou seja, nos leitores da sua documentação. Se as pessoas que forem ler as suas features forem falantes nativos de português, então não há necessidade de escrevê-las em inglês. Tome este livro como exemplo, nele eu escrevi todas as features em português porque o público do livro é de falantes da língua portuguesa.

Escrever código em inglês é uma boa prática, pois todas as palavras da sintaxe do código estão em inglês. E, ao escrever código em inglês, você o torna compreensível para um público maior, algo importante para projetos open source, por exemplo. No entanto, ao escrever features com Cucumber em um projeto privado, você provavelmente deve escolher o idioma que você, sua equipe e seu cliente falam.

A dica é: use o idioma que for mais acessível para os leitores da sua documentação.

■

7.3 Escrevendo um cenário

Uma feature com Cucumber representa a especificação de uma funcionalidade ou de algum comportamento do seu sistema. Para deixar mais claro o escopo e os critérios de aceite, você deve listar alguns exemplos chave do funcionamento da funcionalidade sendo especificada, seguindo a prática de especificação por exemplos que vimos anteriormente.

Cada exemplo de funcionamento da sua funcionalidade é representado como um cenário no Cucumber. Vamos começar entendendo a estrutura de um cenário, olhando um que já escrevemos:

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

''''''

Bem-vindo ao jogo da forca!

|||||

Cenário: Começo de novo jogo com sucesso

→ título

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

→ descrição

Quando começo um novo jogo

Então vejo a seguinte mensagem na tela:

|||||

Bem vindo ao jogo da forca!

|||||

→ steps

Figura 7.2: Estrutura de um cenário com Cucumber

Um cenário no Cucumber tem um título, uma descrição e um ou mais steps. O título deve conter um texto com um breve resumo sobre o que é esse cenário.

A descrição é um texto aberto em que você pode descrever em mais detalhes o cenário em questão. Use esse espaço para explicar o que ele está testando, especificando ou então o critério de aceite relacionado a ele. Por fim, um cenário contém um ou mais steps.

Uma feature do Cucumber tem como propósito documentar um comportamento do nosso sistema. Além da faceta de documentação, você deve se lembrar de que o Cucumber tem também uma faceta de automatização de testes de aceitação. Dentro de uma especificação feita com Cucumber, a parte automatizável são os steps.

Os steps servem para mostrar o passo a passo da regra de negócio sendo especificada. Um step no Cucumber é iniciado com as seguintes palavras reservadas: Dado, Quando, Então, E, Mas e *. Segue um exemplo de um cenário contendo quatro steps:

Cenário: Login com sucesso

Após o visitante logar com sucesso na aplicação, ele deve visualizar uma mensagem de boas-vindas.

Dado que um visitante está na home do site

E ele está deslogado

Quando ele loga com sucesso

Então o sistema deve mostrar pra ele uma mensagem de boas-vindas

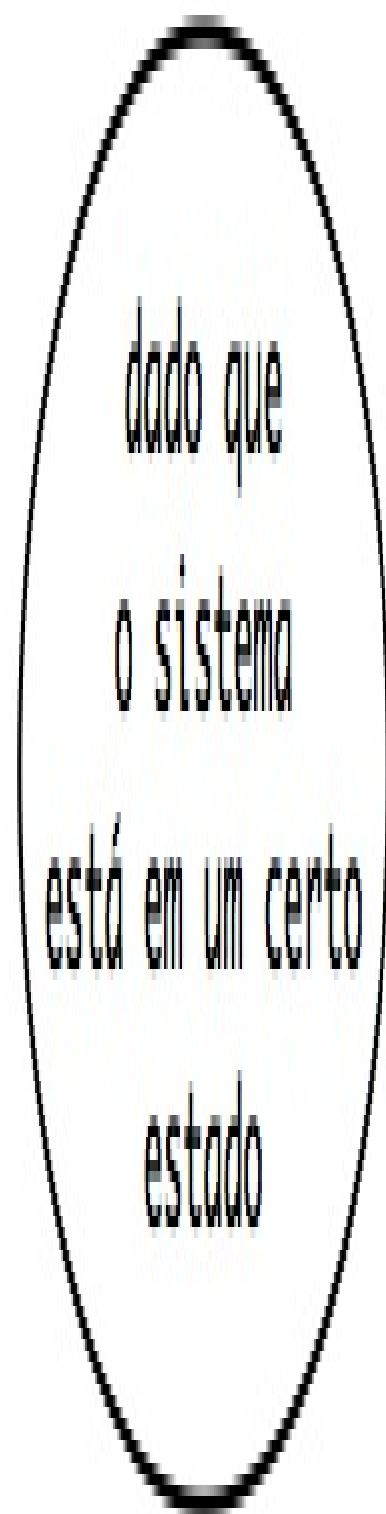
O conjunto padrão dos steps de um cenário segue o seguinte formato:

Dado que o sistema está em um determinado estado

Quando algo ou alguém realizar uma determinada ação nele

Então pode-se observar uma determinada consequência no sistema

Um outro modo de modelar o conjunto de steps de um cenário é através de uma máquina de estado:



quando ele recebe

esta ação

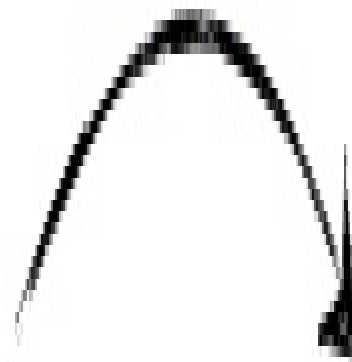


Figura 7.3: Conjunto de steps como uma máquina de estados

Agora que você já sabe o básico da estrutura de um cenário, vamos conversar sobre algumas dúvidas comuns na hora de escrever um novo cenário.

Como devo nomear o título do meu cenário?

Um modo bom de nomear um cenário é utilizar um texto que esteja relacionado ao step Quando ou Então. Ou seja, o título do seu cenário pode ser nomeado segundo a ação sendo realizada no sistema ou então segundo o comportamento esperado do sistema.

O que devo colocar nos steps Dado, Quando e Então?

Ao escrever os steps de um cenário é muito comum ficar em dúvida sobre o que colocar em cada step e como escrever cada um. Vamos tirar essas dúvidas.

Como os steps compõem a parte da especificação que será automatizada, eles devem seguir um formato um pouco mais rígido que o de texto livre usado na descrição da feature e dos cenários.

O step Dado (ou Given) deve colocar o sistema no estado necessário antes de o

usuário ou ator interagir com o sistema. Esse step deve conter as pré-condições existentes para o nosso teste. Ele é o equivalente à fase de setup do xUnit.

Nesse step você deve descrever coisas que já aconteceram no sistema ou como ele está agora, não as ações principais que serão realizadas dentro do teste. Imagine que o cenário tem uma linha do tempo. O que você descrever nesse step são coisas que aconteceram antes de a linha do tempo do cenário começar a rodar.

Segue um exemplo de mau uso desse step:

Dado que um usuário acessa a home do site

E ele se desloga

Quando ele loga com sucesso

Então o sistema deve mostrar pra ele uma mensagem de boas-vindas

Perceba que no step Dado desse exemplo, nós estamos utilizando frases que indicam ações que o usuário está realizando durante o teste. Isso está errado, pois nesse step devem ser descritas coisas que aconteceram antes do início da linha do tempo do cenário. Podemos melhorar isso utilizando frases que indiquem o estado atual do sistema, não ações sendo realizadas sobre ele:

Dado que um usuário está na home do site

E ele está deslogado

Quando ele loga com sucesso

Então o sistema deve mostrar pra ele uma mensagem de boas-vindas

O step Quando (ou When) deve conter a ação principal a ser realizada no sistema durante o teste. Ele deve descrever o que o usuário do sistema faz. Imagine esse step como a primeira coisa acontecendo na linha do tempo do nosso cenário.

Finalmente, o step Então (ou Then) deve conter as reações do sistema em decorrência das ações feitas no step Quando.

7.4 Escrevendo cenários expressivos com Cucumber

Como vimos até agora, uma feature e seus cenários devem ser escritos utilizando a linguagem Gherkin. Até então vimos apenas a estrutura básica do Gherkin, tal como as palavras reservadas "Funcionalidade", "Cenário", "Dado", "Quando" e "Então". Nesta seção veremos algumas funcionalidades mais avançadas do Gherkin e como elas nos permitem escrever cenários mais expressivos.

Scenario outline

Imagine que você está fazendo a especificação de uma calculadora aritmética usando Cucumber e escreve a seguinte especificação:

```
# language: pt
```

Funcionalidade: Soma

Cenário: Soma de 1 + 1

Quando somo 1 + 1

Então o resultado é 2

Cenário: Soma de 2 + 3

Quando somo 2 + 3

Então o resultado é 5

Cenário: Soma de 4 + 3

Quando somo 4 + 3

Então o resultado é 7

Perceba que os três cenários têm o mesmo formato. Com isso, eles se tornaram repetitivos demais. Devido à repetição excessiva, eles se tornaram cansativos de ler, e até um pouco chatos. O Cucumber oferece uma funcionalidade para lidar com esse problema de repetição; ela se chama scenario outline.

Vamos ver como reescreveríamos o exemplo anterior usando scenario outline:

```
# language: pt
```

Funcionalidade: Soma

Esquema do Cenário: Soma de dois números

Quando somo

<a> +

Então o resultado deve ser

<resultado>

Exemplos:

| a | b | resultado |

| 1 | 1 | 2 |

| 2 | 3 | 5 |

| 4 | 3 | 7 |

Repare como é simples reduzir a repetição utilizando a sintaxe de scenario outline do Gherkin. Essa sintaxe é composta por três partes: Esquema do cenário, placeholders e Exemplos.

A palavra reservada Esquema do cenário é utilizada para definir um conjunto de cenários.

Um placeholder é utilizado para definir uma parte que vai variar entre os diversos cenários dentro de um mesmo esquema. Ele é definido como um trecho de texto dentro de < >. Os placeholders do exemplo anterior são: <a>, e <resultado>.

Os Exemplos são uma tabela de dados que serão utilizados para substituir os placeholders no esquema do cenário. A primeira linha dessa tabela serve como cabeçalho, contendo o texto de cada placeholder definido. Na segunda linha em diante, temos os valores que serão utilizados como substitutos dos placeholders. Cada linha a partir da segunda gera um cenário novo. Ou seja, no caso acima nós definimos na verdade três cenários.

■

Scenario outline em inglês

Caso você esteja escrevendo specs em inglês, as palavras reservadas são outras: "Scenario outline" e "Examples". Como vimos antes, você pode descobrir todas as palavras reservados do Gherkin para o inglês utilizando o seguinte comando:

```
$ cucumber --i18n-keywords en
```

■

Agora que já sabemos a sintaxe do scenario outline, vamos ver outro exemplo. Imagine que você esteja desenvolvendo um e-commerce e que você precise especificar o cálculo de desconto para os produtos da sua loja. Para fazer isso, você escreve a seguinte especificação:

language: pt

Funcionalidade: Desconto

Esquema do Cenário: Cálculo de desconto de produto

O desconto dos produtos da nossa loja são baseados em até três variáveis: tipo do produto, preço e quantidade em estoque.

Dado que um

<produto> tem um estoque de <estoque>

unidades

E que esse produto custa

<preco>

reais

Então o desconto desse produto deve ser de

<desconto>

reais

Exemplos: Desconto para eletrodomésticos

Um eletrodoméstico deve ter um desconto de 20 reais caso tenha mais de 100 unidades em estoque e seu preço seja maior que 100 reais.

produto	estoque	preco	desconto
geladeira	90	200	0
geladeira	150	200	20
fogão	150	70	0

Exemplos: Desconto para alimentos

Um alimento deve ter um desconto de 1 real caso tenha mais de 1000 unidades em estoque.

produto	estoque	preco	desconto
arroz	900	5	0
feijão	1500	3	1

Vamos observar algumas partes importantes dessa especificação. A primeira parte que vale a pena reparar é que escrevemos uma descrição para o "Esquema

do Cenário". Assim como um cenário normal, o "Esquema do Cenário" também pode ter uma descrição e normalmente é bom você escrever uma, para enriquecer sua documentação e deixá-la o mais claro possível.

Outra parte interessante que vale a pena notar é que esse esquema de cenário tem dois conjuntos de exemplos, um para descontos de eletrodomésticos e outro para alimentos. Você pode ter quantos conjuntos de exemplos quiser. No nosso caso, quebramos em dois grupos de exemplos para deixar explícito que cada um tem sua própria regra de negócio. Perceba também que cada grupo tem seu título e sua descrição. Pode ser importante escrever título e descrição para os seus exemplos para deixar as regras de negócio claras. Lembre-se, em testes e especificações, uma das qualidades mais importantes é a clareza.

Para que você tenha uma ideia do quanto essas descrições e títulos ajudam na clareza da documentação, compare a especificação a seguir com a anterior:

language: pt

Funcionalidade: Desconto

Esquema do Cenário: Cálculo de desconto de produto

Dado que um

<produto> tem um estoque de <estoque>

unidades

E que esse produto custa

<preco>

reais

Então o desconto desse produto deve ser de

<desconto>

reais

Exemplos:

produto	estoque	preco	desconto	
geladeira	90	200	0	
geladeira	150	200	20	
fogão	150	70	0	
arroz	900	5	0	
feijão	1500	3	1	

Só com as informações dessa feature, você consegue saber quando o desconto é aplicado e com qual valor? Não dá para saber. Sem as descrições, títulos e a separação do conjunto de exemplos, fica bem difícil entender a relação de causa e consequência entre as variáveis que compõem o desconto e o valor final dele. Por isso, é importante você sempre deixar suas especificações explícitas e claras.

Background

Na seção Reduzindo duplicação com hooks do RSpec aprendemos que o RSpec nos oferece o before hook para reduzir a duplicação entre os testes como a seguir:

```
RSpec.describe Game, "in the final phase" do
```

```
  before
```

```
    do
```

```
      @game = Game
```

```
        .new
```

```
      @game.phase = :final
```

```
    end
```

context

"when the player hits the target" do

it

"congratulates the player" do

@game

.player_hits_target

expect(

@game.output).to eq("Congratulations!"

)

end

it

"sets the score to 100" do

```
@game
.player_hits_target

    expect(
@game.score).to eq(100
)

end

end end
```

No Cucumber nós temos um equivalente do before hook do RSpec, ele se chama Background (em português: Contexto). Vamos ver um exemplo simples para entendermos essa funcionalidade:

```
# language: pt
```

Funcionalidade: background

Contexto:

Dado que estou somando $1 + 1$

Cenário:

E adiciono mais 2 na soma

Então o resultado deve ser 4

Cenário:

E adiciono mais 5 na soma

Então o resultado deve ser 7

O background do Cucumber funciona adicionando steps para todos os cenários definidos na feature em questão. Logo, o exemplo que vimos é o equivalente a escrevermos do seguinte modo:

```
# language: pt
```

Funcionalidade: background

Cenário:

Dado que estou somando $1 + 1$

E adiciono mais 2 na soma

Então o resultado deve ser 4

Cenário:

Dado que estou somando $1 + 1$

E adiciono mais 5 na soma

Então o resultado deve ser 7

Imagine o background como uma ferramenta para evitar a duplicação de setup entre os vários cenários da sua feature. No entanto, assim como o before hook do RSpec, você não deve usá-lo simplesmente para aplicar o DRY nos seus cenários. A ideia do background é extrair dos seus cenários aqueles steps de setup que sejam repetitivos e que sejam detalhes desnecessários em relação ao que se está especificando. Vamos ver um exemplo para entender melhor essa ideia.

Imagine que você está especificando o processo de compra do seu e-commerce e escreve a seguinte especificação:

language: pt

Funcionalidade: Processo de compra

Cenário: compra normal

Uma compra normal quer dizer comprar qualquer produto do nosso site que não seja caneca.

Dado que estou logado

Quando faço uma compra normal

Então recebo uma mensagem de sucesso

Cenário: compra com desconto

Compra com desconto é quando alguém compra uma ou mais canecas.

Dado que estou logado

Quando compro uma caneca

Então recebo um aviso dizendo que ganhei um desconto

Cenário: sistema da transportadora está fora ar

Se o sistema da transportadora estiver fora do ar, então não deve ser possível fazer compras.

Dado que estou logado

E que o sistema da transportadora está fora do ar

Quando tento fazer uma compra normal

Então recebo uma mensagem de erro

Note o primeiro step de cada cenário, o Dado que estou logado. Esse step é necessário, pois para fazer uma compra no nosso e-commerce, o usuário deve estar logado. Mas a autenticação não faz parte do escopo dessa especificação, essa especificação é sobre o processo de compra. Logo, esse step é um detalhe desnecessário para os cenários dessa funcionalidade. Como ele é desnecessário para o entendimento de causa e consequência dessa spec e está se repetindo entre os vários cenários, ele é um ótimo candidato para ser extraído para um background:

language: pt

Funcionalidade: Processo de compra

Contexto:

Dado que estou logado

Cenário: compra normal

Uma compra normal quer dizer comprar qualquer produto do nosso site que não seja caneca.

Quando faço uma compra normal

Então recebo uma mensagem de sucesso

Cenário: compra com desconto

Compra com desconto é quando alguém compra uma ou mais canecas.

Quando compro uma caneca

Então recebo um aviso dizendo que ganhei um desconto

Cenário: sistema da transportadora está fora ar

Se o sistema da transportadora estiver fora do ar, então não deve ser possível fazer compras.

Dado que o sistema da transportadora está fora do ar

Quando tento fazer uma compra normal

Então recebo uma mensagem de erro

Essa refatoração ilustra bem o bom uso do background do Cucumber. Você deve utilizá-lo para extrair steps que sejam detalhes repetidos e desnecessários entre os cenários da sua especificação.

Data table

Cada step de um cenário contém apenas uma linha. Às vezes não fica tão bom escrever tudo em uma única linha. Um exemplo desse caso acontece quando você está fazendo um setup que envolve inserir dados no banco de dados:

Dado que tenho um cadastro de usuário com nome "Hugo Pessoa"

e idade

"27"

Nesse step, estamos colocando um registro no banco de dados com alguns atributos. Perceba que somente com dois atributos a linha já começa a ficar grande. Um modo de melhorar esse step é utilizar a funcionalidade de data table (tabela de dados) do Cucumber:

Dado que tenho um cadastro de usuário com os seguintes

atributos:

```
| nome          | idade |  
| Hugo Pessoa de Baraúna | 27    |
```

Ao executarmos esse step, a tabela de dados definida será passada como um array de arrays para a step definition, veremos isso em mais detalhes no capítulo Automatizando especificações com Cucumber.

Essa funcionalidade é bem flexível, você pode construir a sua tabela do modo que for melhor. Por exemplo, não é necessário que ela tenha um cabeçalho.

Segue um exemplo no qual não é usado um cabeçalho para as tabelas:

Cenário: multiplicação de matrizes

Dado que tenho a matriz A:

```
| 1 | 2 |  
| 3 | 4 |
```

E que tenho a matriz B:

```
| 5 | 6 |  
| 7 | 8 |
```

Quando multiplico a matriz A com a matriz B

Então o resultado é:

| 19 | 22 |

| 43 | 50 |

Doc string

Imagine que você está especificando a funcionalidade de cadastro do seu aplicativo web. Quando um visitante faz um cadastro, ele deve receber um e-mail de boas-vindas. Ao conversar com sua equipe, você descobre que é importante que a especificação contenha o texto desse e-mail. Você pode escrever essa spec do seguinte modo com Cucumber:

Cenário: Cadastro com sucesso

Quando um visitante faz um cadastro com o nome

"João da Silva"

Então ele deve receber um e-mail de boas-vindas com o seguinte

texto:

"""

Olá João da Silva,

Bem vindo ao nosso site. Espero que você tenha uma ótima

experiência aqui.

""""

Perceba o texto entre """" """" do step Então ele deve receber (...), esse texto é o que chamamos de doc String. Doc String serve para especificar um texto dentro de um step que não cabe em uma única linha. Nesse exemplo, utilizamos uma doc string para especificar o texto do e-mail de boas-vindas.

Pontos-chaves deste capítulo

Neste capítulo aprendemos a instalar o Cucumber e a escrever uma especificação com ele.

Aprendemos que, em Cucumber, uma especificação é chamada de Feature (Funcionalidade). Aprendemos que a estrutura de uma feature é composta por um título, uma descrição e um conjunto de cenários. A descrição de uma funcionalidade é um lugar importante para você escrever a documentação dessa funcionalidade em si — não deixe de utilizar esse recurso.

Aprendemos também sobre a estrutura de um cenário. Ele é composto por um título, uma descrição e um conjunto de um ou mais steps. O conjunto de steps de um cenário é a parte que será executada como um teste de fato. Por isso, essa parte tem um formato mais "fechado", devendo ser escrita para explicitar a relação de causa e consequência do escopo do teste.

Aprendemos sobre Gherkin, a linguagem que descreve o formato de uma especificação feita com Cucumber. Vimos também que podemos usar o Gherkin

em vários idiomas, por exemplo, em inglês e português.

Por fim, aprendemos várias funcionalidades do Gherkin para escrevermos cenários expressivos, tais como: scenario outline, background, data table e doc string.

No capítulo seguinte, veremos como automatizar uma especificação feita com Cucumber.

Capítulo 8

Automatizando especificações com Cucumber

No capítulo anterior aprendemos sobre a estrutura de uma feature do Cucumber. Vimos que ela tem um ou mais cenários e que cada cenário pode ter um ou mais steps. Em suma, aprendemos sobre como usar o Cucumber para documentar nosso software. Mas ainda falta aprender a usá-lo para automatizar essa documentação e torná-la uma especificação executável.

Neste capítulo, aprenderemos sobre como transformar suas especificações em documentos executáveis.

8.1 Escrevendo os primeiros step definitions

Já vimos antes que um cenário é composto por um conjunto de steps. Vimos também que esse conjunto é a parte automatizável de uma especificação com Cucumber. A seguir, vamos aprender como fazer essa automatização.

Vamos retomar o exemplo da especificação de uma calculadora. Primeiro, crie um diretório chamado calculadora para o nosso projeto e entre nele:

```
$ mkdir calculadora
```

```
$ cd calculadora
```

Execute o comando init do Cucumber para criar os arquivos e diretórios padrões:

```
$ cucumber --init
```

Agora crie um arquivo chamado soma.feature no diretório features com o seguinte conteúdo:

```
# language: pt
```

Funcionalidade: Soma

Cenário: Soma de 1 + 1

Quando somo 1 + 1

Então o resultado é 2

Esse cenário tem dois steps para automatizarmos:

Quando somo 1 + 1

Então o resultado é 2

Para automatizar um step no Cucumber, precisamos escrever o que ele chama de step definition. Um step definition é um trecho de código escrito em Ruby que mapeia um step do seu cenário para um bloco de código que será executado.

Crie agora um arquivo chamado `sum_steps.rb` dentro do diretório `features/step_definitions` para começarmos a escrever nossas step definitions:

```
$ touch features/step_definitions/sum_steps.rb
```

O primeiro step definition que vamos automatizar é o Quando somo 1 + 1. Para isso, escreva o seguinte código dentro do arquivo sum_steps.rb:

```
When("somo {int} + {int}") do
```

```
|a, b|
```

```
  @result = a + b end
```

Vamos analisar o código. Um step definition é composto por até três partes:

o método usado para sua criação. No caso, estamos usando o método When;

uma expressão Cucumber. No nosso exemplo estamos usando a expressão somo {int} + {int};

um bloco de código. No exemplo, é o bloco que contém a linha @result = a + b.

When é um método que o Cucumber nos fornece para criar um step definition. Ou seja, um step definition nada mais é do que a chamada desse tipo de método, passando dois argumentos: uma expressão Cucumber e um bloco de código.

Se rodarmos o comando cucumber agora, veremos que o primeiro step do cenário já está automatizado:

\$ cucumber

language: pt

Funcionalidade: Soma

Cenário: Soma de 1 + 1

Quando somo 1 + 1

Então o resultado é 2

1 scenario (1 undefined)

2 steps (1 undefined, 1 passed)

0m0.009s

You can implement step definitions for undefined steps

with these snippets:

```
Então("o resultado é {int}") do |int|
```

```
  pending # Write code here that turns the phrase above
```

```
    # into concrete actions
```

```
end
```

Podemos confirmar que o primeiro step já está automatizado pelo output do comando cucumber: 2 steps (1 undefined, 1 passed). Então, temos 2 steps, 1 deles já passou e o outro está indefinido. Mas como o primeiro step já passou?

Um step definition mapeia um step de um cenário através de uma expressão Cucumber. Isso quer dizer que, quando o Cucumber roda, para cada step de um cenário, ele procura por um step definition que dê match com o step em questão através da expressão Cucumber.

As expressões Cucumber oferecem por padrão marcadores para mapear os tipos básicos de dados:

{int} - identifica números inteiros, como 100 ou -10

{float} - identifica números de ponto flutuante, como 3.14, .5 ou -1.2

{word} - identifica uma palavra sem espaço, como batata (mas não batata frita)

{string} - identifica palavras cercadas por aspas simples ou duplas, como "batata frita" ou 'batata frita'

Perceba que, na hora de dar o match, o Cucumber ignora as palavras reservadas do Gherkin que definem um step, tais como: Dado, Quando e Então.

Vamos agora analisar o que nosso step definition está fazendo:

```
When("somo {int} + {int}") do
```

```
|a, b|
```

```
@result = a + b end
```

Note que ao mapear o parâmetro {int} o Cucumber automaticamente converte as variáveis correspondentes para o tipo correto (inteiro), de forma que já conseguimos realizar a soma sem nenhuma conversão adicional. O resultado da soma é armazenado na variável de instância @result.

■

Expressões regulares em step definitions

Originalmente, o Cucumber mapeava os step definitions usando expressões regulares. O exemplo anterior seria escrito desta forma:

```
When(/^somo (\d+) \+ (\d+)$/) do
```

```
|a, b|
```

```
@result = a + b end
```

Esse exemplo funciona pois a expressão regular corresponde ao step do nosso cenário. Podemos verificar isso desta forma:

```
/^somo (\d+) \+ (\d+)$/.match? "somo 1 + 1" # => true
```

O Cucumber ainda dá suporte para expressões regulares, pois é um recurso útil para mapear steps mais complexos. Porém, sempre que possível, utilize expressões Cucumber, que são muito mais fáceis de ler e de manter.

■

Com o primeiro step automatizado, podemos começar a automatizar o segundo. Primeiro, vamos rodar o Cucumber novamente e observar a sua saída:

```
$ cucumber
```

```
# language: pt
```

```
Funcionalidade: Soma
```

```
  Cenário: Soma de 1 + 1
```

```
    Quando somo 1 + 1
```

```
    Então o resultado é 2
```

```
1 scenario (1 undefined)
```

```
2 steps (1 undefined, 1 passed)
```


0m0.009s

You can implement step definitions for undefined steps
with these snippets:

```
Então("o resultado é {int}") do |int|  
  pending # Write code here that turns the phrase above  
          # into concrete actions  
end
```

Note o seguinte trecho da saída do comando:

You can implement step definitions for undefined steps
with these snippets:

```
Então("o resultado é {int}") do |int|  
  pending # Write code here that turns the phrase above  
          # into concrete actions  
end
```

Quando o Cucumber é executado e ele percebe que não existe nenhum step

definition para um certo step, ele nos retorna uma dica de template inicial de como implementar o step definition que está indefinido. O trecho de texto acima é essa dica para o step Então o resultado é 2.

Vamos começar a implementar o step definition do step 2. Copie a sugestão de step definition que o Cucumber nos deu para o arquivo `sum_steps.rb`:

```
Then("o resultado é {int}") do  
  
  |int|  
  
  pending  
  
  # Write code here that turns the phrase above  
  
  # into concrete actions end
```

Repare que, em vez de usar o método Então sugerido pela dica, usamos o método Then. Ambos os modos funcionam. Como estamos usando português para nossas features do Cucumber, então ele nos forneceu o snippet com a dica usando o método em português: Então, em vez de Then. Podemos usar qualquer um dos modos, nesses exemplos vamos utilizar os métodos que criam um step definition em inglês.

Vamos agora finalizar esse step definition. Existem dois passos que precisamos fazer para isso. O primeiro é renomear a variável sendo passada para o bloco, que atualmente está como `int`, um nome sem semântica nenhuma. O segundo passo é implementar o código de dentro do bloco desse step definition.

O valor sendo passado para a variável `int` é o resultado esperado da soma, então vamos renomear essa variável para `expected_result`:

```
Then("o resultado é {int}") do  
  |expected_result|  
  pending  
  
  # Write code here that turns the phrase above  
  
  # into concrete actions end
```

O segundo passo é implementarmos o bloco do `step definition`. O que esse `step definition` deve fazer é comparar se o resultado da soma feita no `step` anterior é igual ao esperado neste `step`. No `step definition` anterior salvamos o resultado da soma na variável de instância `@result`, então, para saber se a soma foi feita corretamente, podemos comparar a variável `@result` com a `expected_result`:

```
Then("o resultado é {int}") do  
  |expected_result|  
  fail  
  
  if @result != expected_result end
```

Após alterar o nosso step definition para ficar como acima, vamos rodar o Cucumber e ver se a spec agora está verde:

```
$ cucumber
```

```
# language: pt
```

```
Funcionalidade: Soma
```

```
  Cenário: Soma de 1 + 1
```

```
    Quando somo 1 + 1
```

```
    Então o resultado é 2
```

```
1 scenario (1 passed)
```

```
2 steps (2 passed)
```

```
0m0.007s
```

Pelo output do Cucumber, podemos ver que o teste passou:

```
1 scenario (1 passed)
```

```
2 steps (2 passed)
```

0m0.007s

Vamos agora analisar o que fizemos nesse segundo step definition. Para isso, precisamos primeiro entender como o Cucumber verifica se o teste passou ou não.

Para um cenário no Cucumber ficar no verde, é necessário que todos seus steps tenham step definitions, e que o código executado nesses step definitions não estoure nenhuma exception. Sabendo dessa informação, podemos entender melhor o que fizemos no segundo step definition.

No bloco do nosso step definition, estamos levantando uma exception utilizando o método fail do Ruby, caso o resultado da soma não seja igual ao esperado:

```
fail if @result != expected_result
```

Para saber que nossos step definitions estão corretos, vamos modificar nosso cenário de modo que ele quebre, para que possamos confirmar que nossos step definitions estão verificando aquilo que esperamos. Modifique o cenário que fizemos para fazer com que o resultado esperado da soma de "1 + 1" seja "3":

Cenário: Soma de 1 + 1

Quando somo 1 + 1

Então o resultado é 3

Depois dessa modificação, ao rodarmos o Cucumber, vemos a seguinte saída:

```
# language: pt
```

```
Funcionalidade: Soma
```

```
Cenário: Soma de 1 + 1
```

```
Quando somo 1 + 1
```

```
Então o resultado é 3
```

```
(RuntimeError)
```

```
./features/step_definitions/sum_steps.rb:6
```

```
features/soma.feature:7:in `Então o resultado é 3'
```

Failing Scenarios:

```
cucumber features/soma.feature:5 # Cenário: Soma de 1 + 1
```

```
1 scenario (1 failed)
```

```
2 steps (1 failed, 1 passed)
```

```
0m0.008s
```

O teste quebrou como esperado. Ou seja, nossos step definitions estão

funcionando de verdade. Agora que confirmamos que está tudo ok, modifique o cenário para o modo como estava antes e deixe-o no verde.

Vamos agora analisar como podemos fazer uma melhoria no step definition que finalizamos. Nesse step definition estamos levantando uma exception explicitamente caso o resultado esperado seja diferente do que foi computado.

```
Then("o resultado é {int}") do
  |expected_result|
    fail
  if @result != expected_result end
```

Podemos fazer uma refatoração aqui, usando os matchers do RSpec para definir essa lógica de verificação. Modifique o step definition para utilizarmos o RSpec:

```
Then("o resultado é {int}") do
  |expected_result|
    expect(
      @result).to eq(expected_result) end
```

Rode novamente o Cucumber para garantir que tudo continua no verde:

\$ cucumber

(...)

1 scenario (1 passed)

2 steps (2 passed)

Com tudo no verde, conseguimos automatizar o nosso cenário 100%. Agora temos o que chamamos de uma especificação executável.

Mais alguns detalhes sobre step definitions

Vimos que, para criar um step definition, é necessário utilizarmos um método do Cucumber. Até então utilizamos os métodos When e Then. Além deles, temos também o método Given, que é relativo à palavra reservada Dado ou Given do Gherkin, But e And.

Esses métodos (Given, When, Then, But e And) são na verdade aliases de um método interno do Cucumber chamado `register_step_definition`, que, como o seu nome diz, serve para registrar um step definition.

Como todos eles são aliases do mesmo método, na prática não importa qual você utiliza para registrar um step definition. Isso quer dizer que poderíamos ter escrito nossos step definition do seguinte jeito, que nosso cenário continuaria passando:


```
Given("somo {int} + {int}") do
```

```
|a, b|
```

```
@result = a + b end
```

```
Given("o resultado é {int}") do
```

```
|expected_result|
```

```
  expect(
```

```
@result).to eq(expected_result) end
```

No entanto, se tivéssemos feito do modo que acabamos de ver, perderíamos um pouco a intenção e a semântica dos step definitions. Por exemplo, ficaria mais difícil de perceber que o step definition `Given("somo {int} + {int}")` está sendo usado para mapear o step `Quando somo 1 + 1`. Por isso, utilize sempre o método para registrar um step definition de acordo com o step que você quer mapear.

Pronto, agora você já sabe o básico da construção de step definitions e da automatização de especificações com Cucumber. A seguir veremos como automatizar steps que utilizam funcionalidades mais avançadas do Gherkin, aquelas que aprendemos na seção `Escrevendo cenários expressivos com Cucumber`.

8.2 Escrevendo step definitions para cenários expressivos

Na seção Escrevendo cenários expressivos com Cucumber aprendemos sobre as funcionalidades que o Gherkin nos oferece para escrevermos especificações mais expressivas.

A escrita de step definitions para cenários expressivos tem algumas facilidades para os casos de Parâmetros personalizados, data table e doc string. Nesta seção aprenderemos sobre essas facilidades.

Parâmetros personalizados nos step definitions

As expressões Cucumber permitem que criemos parâmetros mais complexos, além dos tipos básicos de dados como inteiro ou string. Imagine que temos o seguinte cenário:

```
# language: pt
```

Funcionalidade: Boleto do cartão de crédito

Cenário: fatura do cliente em aberto

Dado que a fatura do cliente vence dia 13-01-2019

O step definition seria escrito dessa forma:

```
require "date"
```

```
Dado("que a fatura do cliente vence "
```

```
\
```

```
"dia {int}-{int}-{int}") do
```

```
  |day, month, year|
```

```
    date =
```

```
    Date.new(year, month, day) end
```

Se fosse comum usar datas nos nossos step definitions, mapeá-las e convertê-las seria uma tarefa bastante repetitiva. Podemos criar um tipo personalizado do Cucumber para cuidar disso. Vamos criar o arquivo `features/support/parameter_types.rb` e adicionar o seguinte código:

```
require "date"
```

```
ParameterType
```

```
(
```

```
  name:      "date"
```

```
,
```

```
  regexp:
```

```
    /(\d+)-(\d+)-(\d+)/,
```

```
  type:      Date
```

```
,
```

```
  transformer:
```

```
    ->(day, month, year) {
```

```
      Date
```

```
        .new(year.to_i, month.to_i, day.to_i)
```

```
}  
)
```

O método `ParameterType` do Cucumber recebe vários argumentos:

`name`: o nome do parâmetro, que será usado no step definition

`regexp`: uma expressão regular que corresponde ao parâmetro

`type`: o tipo de destino do parâmetro

`transformer`: uma proc que converte a string do step na variável que será fornecida ao step definition

Note que usamos capture groups na expressão regular para separar o dia, mês e ano em variáveis distintas no transformer. Essa é uma funcionalidade das expressões regulares que nos permite capturar o conteúdo dos placeholders (como `\d+`, que representa um ou mais números) quando eles estão cercados por parênteses:

```
match_data = /(\d+)-(\d+)-(\d+)/.match("13-01-2019"  
)
```

```
match_data.captures
```

```
# => ["13", "01", "2019"]
```

Observe que o método `captures` retornaria um array vazio caso removêssemos os

parênteses:

```
match_data = /\d+-\d+-\d+/.match("13-01-2019")
```

```
match_data.captures
```

```
# => []
```

O Cucumber consegue identificar que usamos esse recurso, e já oferece as variáveis de forma separada no transformer. Com isso, podemos atualizar nosso step definition:

```
Dado("que a fatura do cliente vence dia {date}") do
```

```
|date|
```

```
# ... end
```

Uma vez definido um parâmetro personalizado, o Cucumber passa a sugerir a utilização do mesmo quando encontra um step que corresponde à expressão regular. No exemplo anterior, se o step definition ainda não existisse, ao executar o comando cucumber veríamos a seguinte saída:

```
$ cucumber
```

language: pt

Funcionalidade: Boleto do cartão de crédito

Cenário: fatura do cliente em aberto

Dado que a fatura do cliente vence dia 13-01-2019

1 scenario (1 undefined)

1 step (1 undefined)

0m0.062s

You can implement step definitions for undefined steps
with these snippets:

```
Dado("que a fatura do cliente vence dia {date}") do |date|
```

```
  pending # Write code here that turns the phrase above
```

```
    # into concrete actions
```

```
end
```

Parâmetros personalizados é um recurso bastante poderoso para simplificar os step definitions. Podemos criar parâmetros para as entidades do nosso domínio, como usuário, venda, ou qualquer classe mais complexa que usamos em nossos

testes.

Step definitions para cenários com data tables

Imagine que temos que automatizar o seguinte cenário que está utilizando o recurso de data tables do Gherkin:

```
# language: pt
```

Funcionalidade: Data table

Cenário: tabela simples

Dado que existe o seguinte cadastro de usuário:

nome	idade
Hugo Pessoa de Baraúna	27

Ao rodarmos o Cucumber, vemos o seguinte snippet como parte do output do comando:

You can implement step definitions for undefined steps
with these snippets:

```
Dado("que existe o seguinte cadastro de usuário:") do |table|  
  # table is a Cucumber::MultilineArgument::DataTable  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

Perceba que no snippet desse output o Cucumber já percebeu que esse step está usando data tables e nos sugeriu um step definition que passa como argumento do seu bloco um objeto da classe `Cucumber::MultilineArgument::DataTable`. Esse objeto possui vários métodos bem interessantes, vamos dar uma olhada.

O primeiro método que veremos é o `raw`. Ele retorna a tabela no formato de um array de arrays. Para o exemplo anterior, teríamos o seguinte:

```
table.raw
```

```
# => [["nome", "idade"], ["Hugo Pessoa de Baraúna", "27"]]
```

Um outro método interessante é o `hashes`. Ele retorna um array de hashes, sendo que cada hash tem como keys o cabeçalho da tabela e como values as linhas da

tabela a partir da segunda linha:

```
table.hashes
```

```
# => [{"nome"=>"Hugo Pessoa de Baraúna", "idade"=>"27"}]
```

É possível converter os dados utilizando o `DataTable#map_column` para transformar o campo `idade` em inteiro:

```
table = table.map_column("idade"
```

```
) { |idade| idade.to_i }
```

```
table.hashes
```

```
# => [{"nome"=>"Hugo Pessoa de Baraúna", "idade"=>27}]
```

Além desses métodos, um objeto da classe

`Cucumber::MultilineArgument::DataTable` possui vários outros que são úteis na automatização de specs que estejam usando data tables. Dê uma olhada na documentação dessa classe para conhecer o que mais ela oferece:

<https://www.rubydoc.info/gems/cucumber/3.1.2/Cucumber/MultilineArgument/D>

Vamos ver um último exemplo do uso de data tables com Cucumber. Imagine que estamos construindo um jogo de tabuleiro simples. O objetivo desse jogo é achar em qual coordenada do tabuleiro está a bandeira.

Um tabuleiro vazio pode ser representado do seguinte modo utilizando data table:

	_		_	
--	---	--	---	--

	_		_	
--	---	--	---	--

As coordenadas do tabuleiro funcionam como uma matriz:

	[0, 0]		[0, 1]	
--	--------	--	--------	--

	[1, 0]		[1, 1]	
--	--------	--	--------	--

Se o jogador errar a coordenada onde está a bandeira, por exemplo, na coordenada [0, 0], então o jogo marca "x" nessa coordenada:

	x		_	
--	---	--	---	--

	_		_	
--	---	--	---	--

Se ele acertar, por exemplo na coordenada [1, 1], então o jogo marca "o".

| x | _ |

| _ | o |

Sabendo disso, podemos escrever a seguinte especificação:

Cenário: jogador acha a bandeira

Dado que o jogador começou a jogar

E que a bandeira está na posição [1, 1]

Quando o jogador checa a posição [0, 0]

Então o tabuleiro deve ficar assim:

| x | _ |

| _ | _ |

Quando o jogador checa a posição [1, 1]

Então o tabuleiro deve ficar assim:

| x | _ |

| _ | o |

Vamos implementar os step definitions desse cenário. O primeiro step a ser

automatizado é o Dado que o jogador começou a jogar. O que esse step definition deve fazer é apenas iniciar o tabuleiro do jogo. Podemos fazer isso salvando o tabuleiro como um array de arrays (uma matriz) e em cada elemento colocaremos a marca "_":

Dado("que o jogador começou a jogar") do

@board

```
= [  
  [  
    "_", "_"  
  ],  
  [  
    "_", "_"  
  ]  
]  
  
end
```

Feito isso, vamos implementar o segundo step, o E que a bandeira está na posição [1, 1]. Para esse step, podemos simplesmente salvar a posição onde está a bandeira:

```
Dado("que a bandeira está na posição [{int}, {int}]") do
```

```
|m, n|
```

```
@flag_coordinates = [m, n] end
```

Agora vamos automatizar o step Quando o jogador checa a posição [0, 0]. Esse step definition deve checar se o jogador acertou onde está a bandeira. Se sim, deve colocar a marca "o" na coordenada, caso contrário, deve colocar a marca "x". Podemos implementá-lo do seguinte modo:

```
Quando("o jogador checa a posição [{int}, {int}]") do
```

```
|m, n|
```

```
  player_found_the_flag =
```

```
    @flag_coordinates
```

```
  == [m, n]
```

```
  mark = player_found_the_flag ?
```

```
    "o" : "x"
```

```
@board[m][n] = mark end
```

Falta automatizar apenas mais um step, o seguinte:

Então o tabuleiro deve ficar assim:

```
| x | _ |  
| _ | _ |
```

O que o step definition deve fazer é checar se a tabela do step está igual à tabela que representa o tabuleiro. O Cucumber tem um método bem interessante para comparar duas data tables dentro de um step definition, o diff!. Ele pertence à classe `Cucumber::MultilineArgument::DataTable`. Vamos implementar esse step definition usando esse método e analisar um pouco o seu uso:

Então("o tabuleiro deve ficar assim:") do

```
|expected_board|
```

```
  expected_board.diff!(
```

```
    @board) end
```

No step definition apresentada, `expected_board` é a data table que veio do step, um objeto da classe `Cucumber::MultilineArgument::DataTable` que representa o tabuleiro esperado. `@board` é o tabuleiro atual do jogo, no formato de array de arrays. O que o método `diff!` faz é comparar o data table com esse array de

arrays. Se eles forem iguais, o step definition passa, caso contrário, ele levanta uma exception e quebra.

Um detalhe legal desse método é que, quando o step falha, ele reporta no output do console em que pontos as tabelas estavam diferente. Por exemplo, no final do nosso cenário, nosso board estará assim:

```
| x | _ |  
| _ | o |
```

Se tivéssemos feito nossa especificação errada e o último step estivesse do seguinte modo:

Então o tabuleiro deve ficar assim:

```
| x | _ |  
| _ | x |
```

Ao rodar essa especificação, ela iria quebrar e veríamos o seguinte output no console:

language: pt

Funcionalidade: Funcionamento do jogo da bandeira

Cenário: jogador acha a bandeira

Dado que o jogador começou a jogar

E que a bandeira está na posição [1, 1]

Quando o jogador checa a posição [0, 0]

Então o tabuleiro deve ficar assim:

	x		-	
	-		-	

Quando o jogador checa a posição [1, 1]

Então o tabuleiro deve ficar assim:

	x		-	
	-		x	

Tables were not identical:

		x			-	
	(-)	-		(-)	x	
	(+)	-		(+)	o	

(Cucumber::MultilineArgument::DataTable::Different)

./features/step_definitions/flag_board_steps.rb:22:in `o tabuleiro deve ficar assim:'

flag_game.feature:15:in `Então o tabuleiro deve ficar assim:'

Figura 8.1: Output de data table com erro

Perceba como no output podemos ver em amarelo as células da segunda linha, porque essa linha tem erro, e logo abaixo, em preto, tem a linha com os valores que de fato foram achados. Esse relatório no output ajuda bastante a identificar onde está o erro na sua data table.

Os exemplos que vimos de data tables não esgotam a possibilidade do que você pode fazer com elas, mas já dão uma ideia do poder e flexibilidade dessa funcionalidade. Basicamente, toda vez que você tiver alguma estrutura que possa ser representada de forma tabular, vale a pena avaliar se o uso de data tables pode deixar o seu cenário mais direto e mais fácil de ser lido.

Step definitions para cenários com doc string

Imagine que temos a seguinte especificação para automatizar:

Cenário: Cadastro

com sucesso

Quando um visitante faz um cadastro com o nome "João da Silva"

Então ele recebe um email com o seguinte texto:

```
"""
```

```
"
```

Olá João da Silva,

Bem-vindo ao nosso site. Espero que você tenha uma ótima
experiência aqui.

```
"
```

```
"""
```

Repare que essa especificação está usando uma doc string para especificar o conteúdo do e-mail que deve ser enviado. Vamos ver como podemos receber e manipular uma doc string em um step definition.

Um step definition que contém uma doc string, a envia como último argumento do seu bloco. Portanto, o step definition para o step anterior teria a seguinte estrutura:

Então("ele recebe um email "

\

"com o seguinte texto:") do

|email_body|

email_body contém o texto da doc string end

Sabendo disso, poderíamos terminar esse step definition do seguinte modo:

Então("ele recebe um email "

\

"com o seguinte texto:") do

|email_body|

expect(last_email.body).to eq(email_body)

end

Assim como data tables, doc string não foi feito para um uso específico. Os casos de uso podem variar desde a especificação de um texto de e-mail, passando por texto de mensagens na UI, até a especificação da resposta JSON de uma API. O importante aqui é você saber que essa ferramenta existe e entender como usá-la.

8.3 Support code

No começo da seção Visão geral de Cucumber nós vimos que uma especificação executável com Cucumber tem a seguinte estrutura:

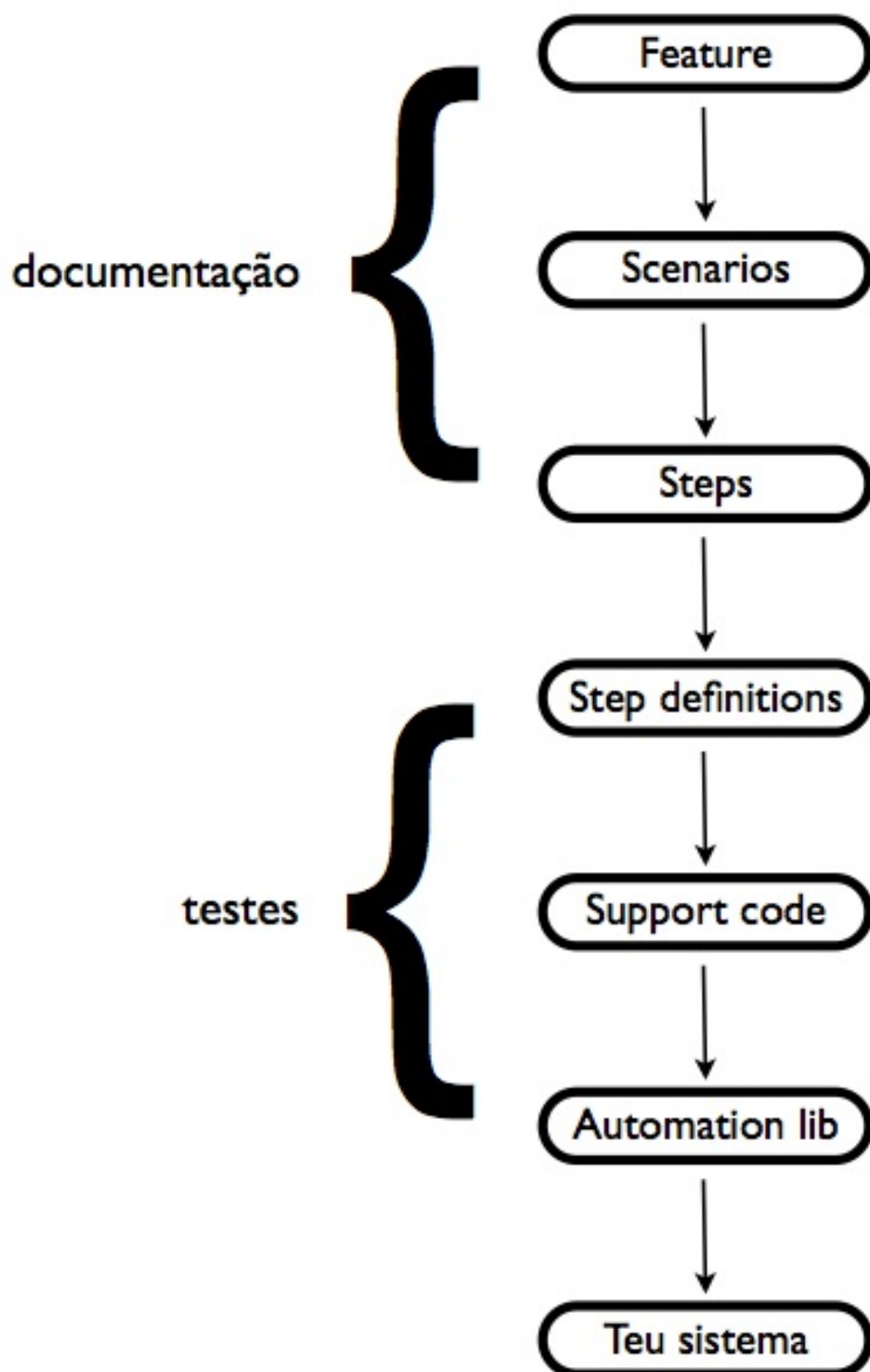


Figura 8.2: Estrutura de uma especificação executável com Cucumber

No capítulo anterior vimos com detalhes todas as camadas dessa estrutura relacionadas ao uso do Cucumber para documentação: feature (Gherkin), cenários e steps.

Nas seções anteriores deste capítulo vimos a primeira camada relacionada ao uso do Cucumber para testes: step definitions. Nesta seção aprenderemos sobre mais uma camada, a camada de support code.

Support code é a camada de código que fica entre os step definitions e a interação com o SUT (system under test, sistema sob teste). Um outro modo de entender a camada de support code é enxergá-la como o lugar para onde deve ser extraído o código repetido entre mais de um step definition. Vamos ver um exemplo para ficar mais claro.

Imagine que estamos construindo uma aplicação web e estamos especificando a funcionalidade de cadastro de conta. A regra de negócio é: quando uma conta for criada, um e-mail de notificação deve ser enviado para o endereço dessa conta. Segue a especificação dessa funcionalidade:

```
# language: pt
```

Funcionalidade: Cadastro na aplicação

Cenário: cadastro de conta com sucesso

Ao se criar uma conta, o dono da conta deve receber um email de notificação.

Quando eu crio uma conta

Então devo receber um email de notificação

Digamos que a tela dessa funcionalidade fosse assim:

Cadastro de conta



http://web-app.com/cadastro



Cadastro de conta

Email

Senha

Confirmação de senha

Criar conta



Figura 8.3: Página de cadastro

Para a implementação dos steps definitions dessa feature, poderíamos fazer o seguinte:

Quando("eu crio uma conta") do

visita a página de cadastro

visit

"/cadastro"

preenche o formulário de cadastro

fill_in

"Email", with: "email.qualquer@gmail.com"

fill_in

"Senha", with: "12345678"

fill_in

"Confirmação de senha", with: "12345678"

click_button

"Criar conta" end

Então("devo receber um email de notificação") do

busca o último email enviado pela aplicação

sent_emails =

ActionMailer::Base

.deliveries

last_email = sent_emails.last

```
# verifica para quem foi enviado o último email
```

```
expect(last_email.to).to eq
```

```
"email.qualquer@gmail.com" end
```

Nesses step definitions estamos usando o Capybara para interagir com a aplicação (métodos como `visit` e `fill_in`) e supondo que a aplicação envia e-mails utilizando o ActionMailer do Ruby on Rails. Como o foco aqui não é aprendermos a usar o Capybara ou o ActionMailer, deixaremos esses detalhes de implementação de lado. Vamos analisar esses step definitions por outro ponto de vista, o ponto de vista da lógica que eles contêm.

Step definitions idealmente devem conter apenas o que você quer fazer na aplicação, não como você faz. Vamos analisar os step definitions anteriores e checar se eles estão seguindo esse guia.

No primeiro step definition temos o seguinte:

```
Quando("eu crio uma conta") do
```

```
# visita a página de cadastro
```

visit

["/cadastro"](#)

preenche o formulário de cadastro

fill_in

"Email", with: "email.qualquer@gmail.com"

fill_in

"Senha", with: "12345678"

fill_in

["Confirmação de senha", with: "12345678"](#)

click_button

["Criar conta" end](#)

O "o que" esse step definition quer fazer é: criar uma conta. Para criar uma conta nessa aplicação, existem diversos passos, que seria o "como":

entrar na URL `"/cadastro"`;

preencher os campos do formulário de cadastro;

clicar no botão de criar conta.

Um step definition não deve conter tanta lógica assim, mesmo que ela seja pequena. Um lugar melhor para conter esse tipo de lógica é a camada de support code. Então, vamos refatorar nosso step definition e mover essa lógica para o lugar correto.

Digamos que a estrutura de diretórios das features dessa aplicação seja a seguinte:

`features/`

`cadastro.feature`

`step_definitions/`

`signup_steps.rb`

`support/`

`env.rb`

Os arquivos da camada de support code vivem no diretório `features/support/`.

Agora precisamos criar o arquivo que vai conter a lógica que estamos extraindo do nosso step definition. Como esse arquivo vai conter a lógica referente à criação de conta, vamos chamá-lo de `sign_up_helpers.rb`:

```
$ touch features/support/sign_up_helpers.rb
```

Com o arquivo criado, vamos finalmente extrair a lógica do step definition para esse novo arquivo. Como é a primeira vez que estamos fazendo isso, vamos fazer passo a passo.

A primeira coisa que temos que saber sobre o support code do Cucumber é que ele é composto por módulos. Portanto, vamos criar um módulo nesse arquivo com o nome `SignUpHelpers`:

```
# features/support/sign_up_helpers.rb
```

```
module SignUpHelpers end
```

Com o módulo criado, podemos mover a lógica do step definition para dentro dele. Como estamos extraindo a lógica de sign up, vamos criar um método chamado `sign_up` nesse módulo e mover o código para ele:

```
# features/support/sign_up_helpers.rb
```

```
module SignUpHelpers
```

```
  def
```

```
    sign_up
```

```
    # visita a página de cadastro
```

```
      visit
```

```
        "/cadastro"
```

```
    # preenche o formulário de cadastro
```

```
      fill_in
```

```
        "Email", with: "email.qualquer@gmail.com"
```


fill_in

"Senha", with: "12345678"

fill_in

"Confirmação de senha", with: "12345678"

click_button

"Criar conta"

end end

Agora só falta mais um passo para que possamos usar esse método no nosso step definition: incluir esse módulo no objeto World (mundo) do Cucumber. O World serve para disponibilizar o código do nosso support code para as step definitions. Daqui a pouco entraremos em mais detalhes sobre o que é o World, por enquanto vamos apenas usá-lo. Para você fazer isso, basta usar o método World e passar como argumento o módulo que você quer disponibilizar:

World(SignUpHelpers)

Faremos isso no final do arquivo `sign_up_helpers.rb`, de modo que ele fique assim:

```
module SignUpHelpers
```

```
  def
```

```
    sign_up
```

```
      # visita a página de cadastro
```

```
        visit
```

```
          "/cadastro"
```

```
      # preenche o formulário de cadastro
```

```
        fill_in
```

```
          "Email", with: "email.qualquer@gmail.com"
```

```
fill_in
```

```
"Senha", with: "12345678"
```

```
fill_in
```

```
"Confirmação de senha", with: "12345678"
```

```
click_button
```

```
"Criar conta"
```

```
end end
```

```
World(SignUpHelpers)
```

Agora sim, já podemos reescrever nosso step definition para utilizar o método que acabamos de adicionar no support code. Ao fazer essa refatoração, nosso step definition ficaria assim:

```
Quando("eu crio uma conta") do
```

```
    sign_up
```

```
end
```

Compare o step definition anterior com a versão de antes de extrairmos a lógica do como para o support code:

```
Quando("eu crio uma conta") do
```

```
    # visita a página de cadastro
```

```
        visit
```

```
        "/cadastro"
```

```
    # preenche o formulário de cadastro
```

```
        fill_in
```

"Email", with: "email.qualquer@gmail.com"

fill_in

"Senha", with: "12345678"

fill_in

"Confirmação de senha", with: "12345678"

click_button

"Criar conta" end

Perceba o quão mais simples ficou o step definition. O número de linhas caiu de várias para apenas uma! Step definitions devem ser simples assim, contendo poucas linhas e delegando sua lógica para o support code.

Vamos agora aplicar a mesma organização entre step definition e support code para o outro step definition que ainda está muito "gordo". O código dela ainda está do seguinte modo:

Então("devo receber um email de notificação") do

```
# busca o último email enviado pela aplicação
```

```
sent_emails =
```

```
ActionMailer::Base
```

```
.deliveries
```

```
last_email = sent_emails.last
```

```
# verifica para quem foi enviado o último email
```

```
expect(last_email.to).to eq
```

```
"email.qualquer@gmail.com" end
```

O que esse step definition quer fazer é apenas verificar se o e-mail foi enviado para a pessoa correta. Ele está fazendo isso resgatando o último e-mail enviado pelo sistema e comparando o destinatário dele com o destinatário esperado.

Para simplificar esse step definition, vamos extrair a lógica de procurar o último e-mail enviado para um método chamado `last_sent_email` e utilizá-lo nesse step definition:

Então("devo receber um email de notificação") do

```
expect(last_sent_email.to).to eq
```

```
"email.qualquer@gmail.com" end
```

Como esse método é relacionado a e-mails, vamos criar um novo arquivo para ele dentro do support code chamado de features/support/email_helpers.rb e extrair a lógica do step definition para esse método:

```
module EmailHelpers
```

```
  def
```

```
    last_sent_email
```

```
      sent_emails =
```

```
        ActionMailer::Base
```

```
          .deliveries
```

```
            sent_emails.last
```

```
    end end
```

World(EmailHelpers)

Após essa refatoração, o nosso arquivo de step definitions ficou simples assim:

Quando("eu crio uma conta") do

sign_up

end

Então("devo receber um email de notificação") do

expect(last_sent_email.to).to eq

"email.qualquer@gmail.com" end

Após esse exemplo prático de extração de lógica de step definition para módulos do support code, já deve estar mais claro para você o que é o support code e para que ele serve.

O support code deve conter a lógica necessária para interagir com o SUT. Do ponto de vista dos step definitions, ele é uma espécie de DSL da nossa aplicação.

Ou seja, as ações que podem ser realizadas na nossa aplicação são "descritas" pelo support code, de modo que os step definitions possam utilizar essas descrições para executar ações no nosso sistema.

Agora que vimos tudo isso, vamos dar uma olhada melhor no que é o World do Cucumber.

8.4 Cucumber World

Na seção anterior vimos como usar o método World para disponibilizar um módulo do support code para os step definitions. Nesta seção, veremos mais detalhes do que é o Cucumber World.

No começo deste capítulo vimos os seguintes step definitions para a soma de uma calculadora:

```
When("somo {int} + {int}") do
```

```
  |a, b|
```

```
    @result = a + b end
```

```
Then("o resultado é {int}") do
```

```
  |expected_result|
```

```
    expect(
```

```
    @result).to eq(expected_result) end
```

Perceba que no código estamos compartilhando uma variável de instância entre as dois step definitions, a variável `@result`. Como isso funciona no Cucumber por debaixo dos panos?

Todo código de dentro de um step definition roda no contexto de um objeto chamado `World`. O `World` default do Cucumber é apenas uma instância da classe `Object`. Dentro do código do Cucumber existe um método chamado `create_world` responsável por criar esse objeto:

```
# fragmento editado do código do Cucumber def
```

```
  create_world
```

```
  @current_world = Object.new end
```

Se formos checar o `self` de um step definition qualquer, esse `self` é o objeto `World` do Cucumber:

```
Given("anything") do
```

```
  puts
```

```
  self.class
```

=> Object end

Portanto, é possível compartilhar variáveis de instâncias entre os step definitions de um cenário porque todos eles compartilham o mesmo contexto (self), que é um objeto World.

Uma informação importante à qual vale ficarmos atentos é que o Cucumber cria um World novo para a execução de cada cenário das suas features. Ou seja, o objeto World não é compartilhado entre os cenários, um novo World é criado e destruído para cada cenário sendo executado.

Uma última questão que falta esclarecermos sobre o World é como ele é usado para integrar o nosso support code com os step definitions.

Como vimos antes, o support code é composto por módulos Ruby. Na seção anterior fizemos um módulo chamado EmailHelpers no support code e o disponibilizamos para os step definitions através da chamada do método World(EmailHelpers). Agora que já sabemos que o World do Cucumber é apenas um objeto, podemos entender melhor o que significa chamar o método World.

O que esse método faz é bem simples, ele apenas estende o objeto World com o módulo que foi passado como argumento. Em código, o que ele faz seria mais ou menos o seguinte:

```
def World
```

```
(*world_modules)
```

```
world_modules.each  
do |module|  
  
  @current_world.extend(module)  
  
end end
```

Pronto, isso é tudo que você precisa saber sobre o Cucumber World. Ele é um objeto composto pelos módulos do seu support code, que serve para dar o contexto dos step definitions e também para interagir com o SUT.

8.5 Usando Cucumber hooks

Assim como o RSpec, o Cucumber também tem hooks. Os mais comuns são o Before e After, mas além desses também existem o Around e o AfterStep. Vamos dar uma olhada apenas nos dois primeiros.

O hook Before recebe um bloco que é executado antes de cada cenário. Um hook bem simples seria assim:

Before do

```
puts
```

```
"Estou sendo executado antes de cada cenário" end
```

Algo importante de saber é que o World do Before hook é o mesmo dos seus step definitions. Logo, um possível uso desse hook pode ser para inicializar alguma variável que será compartilhada entre todos os step definitions. Um exemplo seria fazer o login de um usuário no hook e usar esse usuário nos seus step definitions:

Before do

```
@user = login_user end
```

```
When("eu compro um produto") do
```

```
@user.buy_product end
```

Assim como o support code, os hooks ficam dentro do diretório features/support. Portanto, poderíamos ter colocado o hook acima em um arquivo chamado features/support/hooks.rb.

Conhecendo o Before, entender o que o After hook faz fica bem fácil. Ele executa o bloco passado para ele depois da execução de cada cenário:

```
After do
```

```
  puts
```

```
"Estou sendo executado depois de cada cenário" end
```

Assim como no RSpec, esses hooks podem ser úteis para as fases de setup e teardown da sua especificação executável. Um exemplo de uso dos hooks seria limpar os dados inseridos no banco de dados após a execução de um cenário.

Outra coisa interessante de hooks é que eles podem ser executados apenas para alguns cenários das suas features, utilizando o que o Cucumber chama de tags. Vamos dar uma olhada nisso.

Usando as tags do Cucumber

No Cucumber, é possível você criar tags para marcar seus cenários e features. Para usar uma tag, basta definir um nome e usar esse nome com @ na frente, colocando essa tag na linha anterior da definição de um cenário ou feature:

@tag_1

Funcionalidade: Hello World

@tag_2

Cenário: Hello world com sucesso

Quando rodo meu programa de hello world

Entao ele imprime na tela

```
"hello world"
```

Na execução das suas features, as tags servem para duas coisas: para filtrar que somente algumas features ou cenários sejam executados e para rodar um hook somente para algumas tags. Vamos ver um exemplo de cada um dos usos.

Utilizando o exemplo anterior, imagine que queiramos executar um hook somente para cenários e features marcadas com a tag "@tag_2". Poderíamos fazer isso do seguinte modo:

```
Before "@tag_2" do
```

```
  puts
```

```
"executado hook da tag_2" end
```

A mesma coisa serve para qualquer tag e qualquer hook. Por exemplo, poderíamos fazer o seguinte para rodar um hook somente depois da execução de features ou cenários com a tag "@tag_1":

```
After "@tag_1" do
```

```
  puts
```

"executado hook da tag_1" end

Isso cobre o uso de tags com hooks. Agora vamos ver o uso de tags para filtrar a execução de cenários.

Imagine que você tem a seguinte feature:

Funcionalidade: Hello World

@happy_path

Cenário: Hello world com sucesso

Quando rodo meu programa de hello world

Entao ele imprime na tela

"hello world"

Cenário: Hello world sem sucesso

Quando rodo meu programa de hello world

Então ele não imprime na tela o texto

"hello world"

Vamos supor que você só quer executar os cenários marcados com a tag @happy_path. Você pode fazer isso usando a flag --tags do comando Cucumber:

```
$ cucumber --tags @happy_path
```

É possível também rodar todos os cenários que não têm uma determinada tag. Para isso, basta você colocar "not" antes do nome da tag. Por exemplo, se quisermos rodar todos os cenários que não têm a tag @happy_path podemos fazer o seguinte:

```
$ cucumber --tags 'not @happy_path'
```

Configurando o ambiente de testes

Já vimos bastante coisa relacionada à execução e automatização de especificações executáveis com Cucumber, mas vale a pena ver mais uma coisa, que também fica dentro do diretório features/support/, o arquivo env.rb.

Quando você executa o comando cucumber, a primeira coisa que ele faz é carregar todos os arquivos dentro de features/support/. Depois, ele carrega os arquivos dentro de features/step_definitions e finalmente faz o parsing das features.

Ao carregar os arquivos dentro de features/support, ele faz isso sem seguir uma ordem específica, com exceção do features/support/env.rb.

O Cucumber carrega esse arquivo antes de tudo, de modo que, se você precisar fazer alguma coisa relacionada a preparar o ambiente dos seus testes, é nesse arquivo que você deve fazê-lo. Um exemplo do que poderia ser feito nesse arquivo é dar um require de alguma lib da qual seus testes dependam.

Pontos-chaves deste capítulo

Neste capítulo aprendemos como automatizar especificações escritas com Cucumber. Aprendemos que, para fazer isso, precisamos escrever step definitions, e que um step é mapeado com um step definition via sua expressão Cucumber.

Além da camada de step definitions, conhecemos também a camada de support code. Essa camada é muitas vezes ignorada por algumas pessoas que estão começando a usar Cucumber. Usá-la é importante, pois sem ela seus step definitions ficam difíceis de serem mantidos.

Por fim, vimos mais alguns itens relacionados com a execução de especificações feitas com Cucumber, tais como hooks e tags.

O capítulo seguinte é o último sobre Cucumber. Nele veremos boas práticas de Cucumber, algo importante e que pode diferenciar um usuário iniciante de um usuário mais avançado.

Capítulo 9

Boas práticas no uso de Cucumber

Nos capítulos anteriores vimos sobre como instalar e fazer o setup do Cucumber. Vimos também como é a estrutura de uma feature, de seus cenários, e como escrever step definitions.

Neste capítulo, veremos boas práticas no uso de Cucumber de modo geral. Ao pensar em boas práticas na escrita de especificações com Cucumber você deve pensar sempre em duas coisas:

Cucumber é primeiro uma ferramenta de documentação, e depois uma ferramenta de automatização de testes;

ao escrever especificações com Cucumber, tenha como principal objetivo a clareza.

Quando uma especificação com Cucumber não está boa, normalmente é porque um desses dois guias não foi bem seguido. Para avaliar se uma especificação está boa, lembre-se de levar os dois pontos em consideração.

A seguir, veremos exemplos mais concretos de como escrever especificações de boa qualidade com Cucumber

9.1 Use e abuse das descrições da funcionalidade e dos cenários

Quando comecei a aprender Cucumber, vi vários exemplos do seu uso na web. A maioria dos exemplos seguia o seguinte estilo:

language: pt

Funcionalidade: Login

Para utilizar o meu sistema

Como um visitante

Quero fazer login no sistema

Contexto:

*

existe um usuário cadastrado com a seguinte credencial:

```
| login | senha |  
| hugobarauna | 123 |
```

Cenário: Login com sucesso

Dado que estou na página de login

Quando preencho o campo "Login" com "hugobarauna"

E preencho o campo "Senha" com "123"

E cliço no botão "Logar"

Então eu vejo uma mensagem de sucesso

Cenário: Login com senha errada

Dado que estou na página de login

Quando preencho o campo "Login" com "hugobarauna"

E preencho o campo "Senha" com "senha errada"

E cliço no botão "Logar"

Então eu vejo uma mensagem de erro com o seguinte text:

""""

Login ou senha estão errados

""""

Esse estilo de especificação é bem comum: feature e cenários sem descrição,

steps com muitos detalhes etc. Apesar de ele servir para a parte de automatização de testes, ele foca pouco na parte de documentação, o que é ruim. Vamos analisar melhor esse estilo e ver o que poderíamos melhorar.

A descrição da funcionalidade foi feita utilizando o formato Connextra. Como vimos na seção Estrutura de uma feature com Cucumber, embora esse formato seja um bom template para a descrição de uma feature, você não precisa se ater somente a ele.

A fim de enriquecer sua documentação, você pode escrever mais coisas na descrição da sua funcionalidade. Você pode explicar o requisito sendo especificado de modo mais amplo. Pode, por exemplo, escrever o porquê da existência dessa funcionalidade. Pode escrever para que ela serve, por que ela é útil. Pode escrever também os critérios de aceite dessa funcionalidade. Em suma, você pode (e deve) escrever bem mais coisas na descrição do que apenas utilizar o template Connextra.

Vamos analisar agora os cenários da especificação anterior. Perceba que nenhum dos dois cenários tem uma descrição. Quando um cenário não tem descrição, fica mais difícil de entender com qual regra de negócio ele está relacionado. Sempre escreva uma pequena descrição em cada cenário explicando o comportamento sendo testado por ele, tal como o exemplo a seguir:

Cenário: sistema da transportadora está fora ar

Se o sistema da transportadora estiver fora do ar, então não deve ser possível fazer compras.

Dado que o sistema da transportadora está fora do ar

Quando tento fazer uma compra normal

Então recebo uma mensagem de erro

Cucumber em projetos com domínio de negócio complexo

Nem todo sistema tem um domínio de negócio complexo. Um sistema de publicação de blogs pode ser complexo tecnicamente, mas não tem um domínio complexo. Uma aplicação de compartilhamento de fotos poder ser complexa tecnicamente, mas não tem um domínio complexo.

Para entendermos melhor o que é um domínio de negócio complexo, seguem algumas características comuns desse tipo de domínio:

Você nunca ouviu falar dos termos que as pessoas de negócio do projeto falam nas conversas sobre os requisitos;

cada regra de negócio parece simples inicialmente, mas tem vários desdobramentos internos;

o problema a ser resolvido pelo sistema não é um problema que uma pessoa "comum" teria. Costumam ser problemas de nichos bem específicos ou relacionados a processos internos de empresas e outros tipos de instituições;

entender as regras de negócio e seu contexto não é simples, podendo ser tão custoso quando desenvolver o código do sistema em si.

Alguns exemplos de domínios que podem se encaixar como um domínio de

negócios complexo:

um sistema de uma corretora de seguros de carro online;

um sistema de gerenciamento de identidade e acesso em sistemas corporativos (IAM);

um sistema de cálculo de impostos de produtos de supermercado;

um sistema para automatizar um processo de negócio de uma empresa multinacional.

Ao entrar em um projeto com esse tipo de domínio, não é possível entender as regras de negócio por trás do código apenas através de sua leitura. Darei um exemplo.

Já participei de um projeto que tinha uma funcionalidade chamada "Análise de segregação de função" e uma classe chamada SegregationOfDuty. Você já ouviu falar em "Análise de segregação de função"? Eu não conhecia até então. Por isso, só de ler o código era impossível de entender o porquê daquele código, pois eu não tinha o contexto de negócio.

Era necessário primeiro entender o que era uma "Segregação de Função" no contexto de negócio, para depois entender a classe SegregationOfDuty.

Nesse tipo de contexto, uma documentação com Cucumber bem-feita tem muito valor, pois ela pode ajudá-lo a entender as regras de negócio por trás de uma parte do código. São situações em que fica ainda mais importante você focar suas specs com Cucumber primeiro em documentação e depois em testes.

9.2 Evite detalhes desnecessários

Uma má prática comum que acontece em especificações com Cucumber é a existência de detalhes desnecessários. Eles podem comprometer a clareza e desviar a atenção do leitor para algo que não é o foco da especificação em questão. Vamos ver um exemplo de uma especificação com detalhes desnecessários.

```
# language: pt
```

Funcionalidade: Notificação de compra

Ao usuário realizar uma compra, ele deve receber uma notificação por email com os detalhes da sua compra.

Contexto:

*

existe um cadastro de usuário com os seguintes dados:

| email | senha |
| user@mail.org | abcdefg |

Cenário: Notificação com sucesso

Dado que estou logado com as seguintes credenciais:

| email | senha |
| user@mail.org | abcdefg |

Quando eu compro o produto "macbook pro"

Então devo receber um email com os detalhes da compra

Essa especificação é sobre a funcionalidade de notificação de compra por e-mail. O assunto dessa especificação é bem simples: quando um usuário faz uma compra, ele deve receber por e-mail os detalhes dessa compra. Qualquer detalhe que não esteja diretamente relacionado com a relação de causa e consequência desse assunto é um candidato a detalhe desnecessário.

Seguem alguns exemplos de detalhes que não fazem parte do foco dessa especificação mas que estão escritos nela:

no step de contexto dessa especificação, estamos salvando no banco um cadastro de usuário com e-mail e senha. Como esse detalhe é desnecessário, ele já foi extraído para fora do cenário, está no contexto. Mas podemos fazer melhor, veremos como daqui a pouco;

no step Dado, estamos garantindo que existe um usuário logado com o e-mail "user@mail.org" e com a senha "abcdefg". O e-mail e a senha são detalhes desnecessários pois eles não impactam diretamente se o usuário vai receber a

notificação ou não. O endereço de e-mail poderia ser qualquer um, assim como a senha;

no step Quando, o usuário compra o produto "macbook pro". O nome do produto também é um detalhe desnecessário nesse cenário, pois a notificação por e-mail independe do nome do produto.

Um modo melhor de escrever a especificação anterior seria:

language: pt

Funcionalidade: Notificação de compra

Ao usuário realizar uma compra, ele deve receber uma notificação por email com os detalhes da sua compra.

Cenário: Notificação com sucesso

Dado que estou logado

Quando eu compro um produto

Então devo receber um email com os detalhes da compra

Perceba que, após refatorada, a spec ficou bem mais simples do que a versão anterior. Ela ficou menor, mais concisa e mais direta. Não existe nenhum detalhe nessa spec que não faça parte de seu assunto principal.

Escrever specs mais diretas e com menos detalhes é um bom caminho, porém temos que ter duas coisas em mente:

não deixar a spec genérica demais;

specs mais fáceis de serem lidas podem incorrer na necessidade de step definitions mais flexíveis e mais inteligentes.

Sobre o primeiro ponto da lista, vamos imaginar um exemplo genérico ao extremo:

Funcionalidade: Funcionamento do sistema

Cenário: Sistema funciona com sucesso

Dado que o sistema está ligado

Quando eu faço uma compra

Então ele funciona

Essa especificação é tão genérica que ela não serve nem para documentação e nem para testes. Ela não contém nenhuma regra de negócio explícita. Muito menos é possível criar algum teste automatizado baseado nesses steps.

Portanto, ao tirar os detalhes desnecessários da sua spec, fique atento a se você

não está tirando detalhes que eram necessários para explicar a regra de negócio em questão, ou então detalhes que são importantes para a automatização das specs.

Vamos agora falar do segundo ponto: specs mais fáceis de serem lidas podem incorrer na necessidade de step definitions mais flexíveis e mais inteligentes. Vamos continuar usando o exemplo da spec que refatoramos. Imagine que agora você precisa automatizá-la:

language: pt

Funcionalidade: Notificação de compra

Ao usuário realizar uma compra, ele deve receber uma notificação por email com os detalhes da sua compra.

Cenário: Notificação com sucesso

Dado que estou logado

Quando eu compro um produto

Então devo receber um email com os detalhes da compra

Vamos pensar na automatização de step por step, começando pelo primeiro. O primeiro step é: Dado que estou logado.

Para escrever o step definition desse step eu preciso logar com um usuário. Mas qual usuário? Para logar, eu preciso primeiro ter um cadastro de usuário. É devido a essa dependência técnica que a spec antes tinha o detalhe desnecessário da criação do usuário no banco de dados:

Contexto:

*

existe um cadastro de usuário com os seguintes dados:

| email | senha |
| user@mail.org | abcdefg |

Agora digamos que já sabemos com qual usuário queremos logar. Qual a senha desse usuário? Qual seu e-mail? Na spec anterior, tanto a criação do usuário com esses atributos, e-mail e senha, quanto o uso desses atributos no step de login estavam explícitos:

Contexto:

*

existe um cadastro de usuário com os seguintes dados:

| email | senha |
| user@mail.org | abcdefg |

Cenário: Notificação com sucesso

Dado que estou logado com as seguintes credenciais:

| email | senha |
| user@mail.org | abcdefg |

No entanto, essa spec é sobre o envio de notificação de compra por e-mail, não é sobre a criação de usuário, ou sobre o fato de a autenticação ser por e-mail e senha. Por isso, esses detalhes são desnecessários.

Apesar de serem desnecessários para a documentação e explicação da regra de negócio, eles são importantes para a automatização dos steps. O que precisamos fazer para resolver esse problema é escrever um step definition mais inteligente. Vamos ver o que isso quer dizer.

Voltando ao step que queremos automatizar: Dado que estou logado. Como nem o e-mail e nem a senha importam nesse step, o que podemos fazer no step definition é já criar um usuário qualquer e logar com esse usuário. Nosso step definition poderia ficar do seguinte modo:

Dado("que estou logado") do

sign_in

end

Perceba que o step definition está bem simples. Ela está delegando toda a lógica para o método sign_in do support code. Vamos implementar uma ideia do que seria esse método.

O que esse método deve fazer é:

criar um usuário;

logar com esse usuário.

Nesse step definition em específico não importa quem é esse usuário ou qual seu e-mail e senha. Apesar disso, vamos fazer esse método genérico o suficiente para que ele possa ser usado de dois modos: quando são importantes o e-mail e senha do usuário a ser logado, ou quando não.

Para cumprir esses requisitos, poderíamos implementar esse método do seguinte modo:

module AuthenticationHelpers

```
def sign_in(email = "user@mail.org")
)
  user =
  User.find_or_create_by!(email: email) do
    |user|
      user.password =
"123"

  end
```

```
visit
"login"
```

```
fill_in
"Email", with:
user.email
fill_in
```

"Senha", with:

user.password

click_button

"Logar"

end end

World(AuthenticationHelpers)

Perceba como esse método é "inteligente". Se você passar um e-mail, ele usa esse input, caso contrário, ele cria um usuário default e usa esse usuário para logar.

Escrever esse tipo de support code permite a escrita de steps sem detalhes desnecessários. Como esses detalhes são necessários somente na automatização dos steps, então estamos empurrando-os para a camada de support code. Vamos agora implementar a mesma ideia para o step seguinte.

O step seguinte a ser automatizado é o: Quando eu compro um produto. O que precisamos fazer nesse step é navegar até a página de um produto e clicar no botão de comprar. Mas qual produto? Para essa spec não importa, já que ela não é sobre comprar produto, ela é sobre a notificação de e-mail após a compra do produto.

No entanto, na hora de implementar o step definition, precisamos ter um produto na base de dados, para poder comprá-lo. Vamos implementar esse step definition seguindo a mesma linha de raciocínio que fizemos anteriormente.

Esse step definition poderia ser implementada do seguinte modo:

Quando("eu compro um produto") do

buy_product

end

O método buy_product precisa fazer o seguinte:

cadastrar um produto na base de dados;

entrar na página desse produto;

clicar no botão de comprar.

Podemos implementá-lo do seguinte modo no support code:

module ProductHelpers

def

buy_product(product_name)

product =

Product.find_or_create_by!(name:

product_name)

visit

"/produtos/#{product_name}"

click_button

"Comprar"

end end

World(ProductHelpers)

Pronto, implementamos mais um step definition inteligente o suficiente para nos possibilitar escrever um cenário sem detalhes desnecessários.

Resumindo, para evitar detalhes desnecessários nas suas specs com Cucumber, você deve estar sempre atento a qual é o assunto ou objetivo principal da sua feature ou cenário. Sabendo disso, você deve analisar se os seus steps têm algum detalhe escrito que não está relacionado com o assunto principal. Se tiver, empurre esses detalhes para a camada de support code e utilize o support code para escrever step definitions mais simples.

9.3 Cenários imperativos versus cenários declarativos

Como vimos na seção anterior, um problema comum que acontece em especificações com Cucumber são os detalhes desnecessários. Um caso específico de cenário com esse tipo de problema é o cenário imperativo. Para entender o que é um cenário imperativo, vamos ver um exemplo que segue esse estilo:

Funcionalidade: Login

Cenário: Login com sucesso

Após o visitante logar com sucesso na aplicação, ele deve
visualizar uma mensagem de boas-vindas.

Dado que existe um usuário

"user@mail.org" com a senha "123"

Quando ele visita a página de login

E preenche o campo

"E-mail" com "user@mail.org"

E preenche o campo

"Senha" com "123"

E clica no botão

"Logar"

Então o sistema deve mostrar para ele a seguinte mensagem:

Bem-vindo user@mail.org

Perceba o quanto de detalhes desnecessários existem nesse cenário: e-mail e senha do usuário, preencher o campo e-mail, preencher o campo senha, clicar no botão de login etc.

A intenção principal desse cenário era mostrar apenas uma regra de negócio: quando um usuário login, então o sistema deve mostrar pra ele uma mensagem de sucesso. No entanto, o excesso de detalhe de interação com a interface gerou muito ruído no texto, dificultando o entendimento do assunto principal.

Esse cenário é imperativo pois, em vez de ele falar o que ele quer do comportamento esperado, ele explica como ele quer.

Uma característica muito comum de cenários imperativos é que eles costumam focar demais em cada detalhe da interface com usuário. No cenário mostrado, há detalhes como a label de cada campo a ser preenchido no formulário e o botão que é clicado no formulário.

De modo geral, ao escrever um cenário, você está especificando uma regra de negócio, não o comportamento ou funcionamento da UI (user interface, interface com o usuário).

O contrário de um cenário imperativo é um cenário declarativo. Essa nomenclatura é um paralelo aos estilos de linguagem de programação "imperativo" e "declarativo". Em linguagens imperativas nós escrevemos em detalhes como o nosso programa deve fazer o que queremos, instrução por instrução. Já em linguagens declarativas (tais como CSS ou SQL), escrevemos o que queremos como resultado, sem entrar em detalhes de como o resultado esperado é realizado.

Refatorando o cenário anterior para seguir o estilo declarativo, ele poderia ficar assim:

Cenário: Login com sucesso

Após o visitante logar com sucesso na aplicação, ele deve visualizar uma mensagem de boas-vindas.

Quando um visitante loga na aplicação

Então o sistema deve mostrar pra ele uma mensagem de sucesso

Perceba a diferença entre esse estilo e o estilo imperativo. Retiramos todos os detalhes desnecessários que tiravam o foco da intenção principal. Detalhes como as especificações de UI por exemplo (labels e campos do formulário).

Um dos benefícios de escrever cenários no estilo declarativo é que eles ficam menos frágeis. Por menos frágil, entenda que o texto do seu cenário tem menos probabilidade de mudar se algum detalhe de requisito não relacionado ao foco do cenário for modificado.

Por exemplo, imagine que a equipe de produto pediu para mudar o campo do formulário que antes era "Senha" para "Segredo". Se você estivesse seguindo o estilo imperativo, você teria que modificar o texto do seu cenário e talvez o step definition.

Por outro lado, se você estivesse seguindo o estilo declarativo e também estivesse enviado toda a lógica do step definition para o support code, o único lugar que você precisaria modificar seria o support code. Antes, ele estaria assim:

```
def
```

```
sign_in
```

```
# (...)
```

```
fill_in
```

```
"Senha", with: user.password end
```

E depois ele mudaria para ficar assim:

```
def
```

```
sign_in
```

```
# (...)
```

```
fill_in
```

```
"Segredo", with: user.password end
```

Fazer modificações no support code é muito melhor do que fazer modificações na camada de features ou step definitions, pois na camada de support code você está mexendo com Ruby puro: módulos e métodos. Nessa camada é muito mais fácil você modificar código, renomear métodos etc.

Por exemplo, se você quiser fazer uma modificação em uma expressão Cucumber de um step definition, você precisa checar todos os steps de todos os cenários que estão utilizando esse step definition e avaliar se é necessário atualizá-los também.

Se você quiser fazer uma modificação na camada de features (Gherkin), pode ser necessário envolver pessoas com outros papéis além do desenvolvedor, tais como PO (product owner), analistas de negócio etc.

De modo geral, quanto mais você empurrar o "como" para a camada de support code, melhor, pois o "como" costuma variar mais do que o "o que", e o custo de modificação da camada de support code é o menor de todas as camadas.

Outro problema em cenários imperativos é que eles não levam à criação da linguagem ubíqua do seu projeto. Linguagem ubíqua (ubiquitous language) é um termo inventado por Eric Evans no livro Domain Driven Design (EVANS, 2004). Uma definição bem simples para esse termo seria: a linguagem formada pelos termos específicos do domínio de negócio do problema sendo resolvido.

Desenvolver uma linguagem ubíqua é importante para que todo mundo no projeto fale a mesma língua, principalmente pessoas do negócio e desenvolvedores. Um problema comum de quando não se usa tal linguagem é o de pessoas de negócio e desenvolvedores utilizando palavras diferentes para se referir ao mesmo conceito do domínio do problema.

Esse tipo de problema aumenta o custo de comunicação entre as pessoas de negócio e as pessoas técnicas, pois elas precisam ficar constantemente traduzindo conceitos de uma linguagem para outra. Levando isso ao extremo, é como se os dois grupos de pessoas falassem idiomas diferentes.

Cenários imperativos atrapalham o desenvolvimento da linguagem ubíqua pois, em vez de os cenários serem escritos com os termos do problema sendo resolvido, eles são escritos com os termos da solução. Por exemplo, em vez de falar de "login com sucesso" (termo do problema), fala-se de "campo de e-mail", "campo de senha", "botão de login", termos da solução do problema, que nesse

caso é uma aplicação web.

Ao escrever cenários, tente escrevê-los utilizando a linguagem de negócio, focando no "o quê" e não no "como". Desse modo você deixa seus cenários menos frágeis e dá maiores chances de se criar uma linguagem ubíqua documentada pelas suas features. Também não se esqueça de que, ao escrever cenários declarativos, você deve empurrar o "como" para a camada de support code, devido à maior facilidade de manutenção dessa camada.

9.4 Organizando os arquivos da minha especificação executável

Já estamos chegando no final do último capítulo específico de Cucumber e ainda não falamos sobre como organizar os arquivos das várias camadas da sua especificação executável. Vamos dar uma olhada nesse assunto.

O Cucumber não força muitas opiniões ou convenções sobre a organização dos seus arquivos, mas ele tem algumas. Vamos rever algumas das convenções e regras que ele tem:

todos os arquivos da sua especificação executável ficam dentro de um diretório chamado `features/`;

os nomes dos arquivos das features devem terminar com a extensão `.feature`;

os arquivos de step definitions vivem no diretório `features/step_definitions` e seus nomes terminam com o sufixo `_steps.rb`;

os arquivos de support code vivem no diretório `features/support`.

Dito isso, ainda ficam algumas perguntas no ar, tais como:

como devo nomear os arquivos das minhas features?

como devo nomear os arquivos dos meus step definitions?

como devo nomear os arquivos do support code?

Não existem regras ou respostas diretas para essas perguntas, mas existem algumas dicas. Vamos dar uma olhada nelas.

Organizando minhas features

Os arquivos de features são provavelmente os mais fáceis de serem nomeados. Você deve nomear seu arquivo de acordo com o título da funcionalidade sendo especificada. Se o título for muito longo, use somente parte dele, o bastante para que o nome fique claro.

Algo que vale a pena ter em mente é que você não precisa ter um mapeamento de um para um entre uma "user story" (do XP, Scrum etc.) e um arquivo de feature do Cucumber. Uma feature no Cucumber é uma funcionalidade do ponto de vista de negócio, não do ponto de vista de desenvolvimento. É comum que uma feature possa conter várias user stories. Portanto, se você estiver usando user stories para captar seus requisitos, esse é um ponto importante para lembrar.

Agora imagine que sua aplicação está ficando cada vez maior e agora você tem trinta features nela. Colocar trinta arquivos direto no diretório features/ pode ficar muito desorganizado. A solução para isso é criar subdiretórios dentro dele.

Por exemplo, digamos que sua aplicação esteja dividida em: gerenciamento de conteúdo, pagamento, cadastro de produtos e relatórios. Você poderia organizar suas features com a seguinte estrutura de diretórios:

features/

content_management/

payment/

product_registry/

reports/

Por fim, outra dica interessante na organização das suas features é você lembrar que elas vão formar uma documentação da sua aplicação. Uma documentação não é apenas um monte de arquivos isolados, ela é formada por um conjunto de arquivos que tenham uma ordem e que façam sentido em conjunto.

Algumas ferramentas open source são bons exemplos de documentação viva escrita em Cucumber, como o RSpec Expectations (<https://relishapp.com/rspec/rspec-expectations/docs/>), VCR (<https://relishapp.com/vcr/vcr/docs/>) ou o próprio Cucumber (<https://relishapp.com/cucumber/cucumber/docs/>).

Organizando meus step definitions

Uma pergunta muito comum ao começar a usar Cucumber é como organizar os step definitions. O consenso atual na comunidade é ter um arquivo de step definition para cada conceito do seu domínio. Mas o que isso quer dizer na prática?

Na prática, quer dizer que você deve ter um arquivo de step definition para mais ou menos cada classe do seu domínio. Por exemplo, se você estiver fazendo um blog, você poderia ter os seguintes arquivos de step definition: `user_steps.rb`, `blog_steps.rb`, `comment_steps.rb`.

Um bom modo de pensar em quantos e quais step definitions devem pertencer a um único arquivo é imaginar esse arquivo como uma classe e as step definitions que ele contém como métodos. O arquivo deve ser coeso, deve conter step definitions que formem um conjunto semântico.

Outra dica importante sobre organização de step definitions é: não se restrinja a reutilizar sempre os mesmos step definitions entre todos os seus arquivos de features. Ao escrever features, você pode se encontrar na situação de querer escrever um step que faz a mesma coisa que outro step de outra feature ou cenário, mas você quer escrevê-lo de um modo um pouco diferente.

Nesse caso você tem dois caminhos: ou criar um novo step definition para esse novo step, ou reutilizar um step definition que já está pronto e não escrever o step de modo diferente. Quando isso acontecer, não se force a sempre reutilizar os step definitions que já estão prontos.

Utilizar os mesmos step definitions nas suas features tem a vantagem de o forçar a desenvolver um conjunto de termos do seu domínio, a sua linguagem ubíqua.

Por outro lado, escrever um step um pouco diferente não necessariamente vai prejudicar a sua linguagem ubíqua. Se você precisar de um step novo, mas que seja parecido com um antigo, simplesmente crie um novo e faça reuso da lógica do step definition que era semelhante através do support code.

Organizando meu support code

A organização do support code segue a mesma ideia da organização dos arquivos de step definitions, você pode ter um arquivo do support code para cada conceito do seu domínio.

Em aplicações web, um outro modo bem interessante de organizar seu support code é através do padrão "page object". Não vamos entrar em detalhes sobre esse padrão, mas se você estiver interessado em ler sobre ele, eu recomendo o seguinte link: <http://robots.thoughtbot.com/better-acceptance-tests-with-page-objects>

Pontos-chaves deste capítulo

Nos capítulos anteriores aprendemos sobre como usar Cucumber de modo geral. Neste capítulo aprendemos sobre como usar o Cucumber de um modo melhor, seguindo boas práticas.

Vimos que escrever features e cenários sem descrição é ruim. A descrição da feature e dos cenários serve para documentação, é um texto aberto que você pode escrever o que for necessário para que o leitor possa entender a especificação da funcionalidade, seu contexto de negócio e suas regras.

Quando você não escreve essas descrições, você está perdendo a oportunidade de unir documentação e testes de aceitação em um lugar só.

Vimos também como evitar detalhes desnecessários nos nossos cenários, empurrando as dependências técnicas para a camada de support code.

Vimos a diferença entre os estilos de cenário imperativo e declarativo. O estilo declarativo foca mais no uso de termos do domínio do negócio, dando mais chances de emergir uma linguagem ubíqua do projeto. Além disso, usar o estilo declarativo leva a cenários mais simples, diretos e menos chatos de serem lidos.

Finalmente, terminamos nossa introdução ao RSpec e Cucumber. Prepare-se, pois nos capítulos seguintes vamos construir um projeto do zero, fazendo BDD com RSpec e Cucumber, e colocando em prática tudo que aprendemos até agora.

Capítulo 10

BDD na prática: começando um projeto com BDD

Não é segredo que para desenvolver uma habilidade o mais importante é praticar, praticar e praticar! Com BDD não é diferente, não basta ler sobre RSpec, Cucumber, suas sintaxes e suas funcionalidades. É necessário praticar, é necessário escrever testes! Por isso, neste e nos próximos capítulos vamos desenvolver uma aplicação de 0 a 100 fazendo BDD com Cucumber e RSpec.

Neste capítulo, vamos começar o desenvolvimento da aplicação, definir o seu escopo, fazer seu setup, especificar e testar a primeira funcionalidade. Além disso, você terá o primeiro contato com o fluxo de fazer BDD com Cucumber e RSpec, o famoso outside-in development. Ou seja, começar o teste com uma especificação de Cucumber, depois fazer testes de RSpec, fazer o RSpec passar e, por fim, fazer o Cucumber passar.

Durante o desenvolvimento dessa aplicação, você aprenderá as respostas para algumas dúvidas clássicas sobre TDD, tais como:

Por onde devo começar a testar?

O que devo testar?

Quantos testes devo escrever?

Ao decorrer deste e dos próximos capítulos, as respostas para essas e outras perguntas ficarão mais claras para você.

10.1 Definindo o escopo da nossa aplicação: Jogo da Forca

A aplicação que vamos desenvolver é um jogo da forca com interface pela linha de comando. Antes de começarmos a fazer o código, é necessário primeiro saber os requisitos do jogo. Você conhece o jogo da forca? Ele é originalmente um jogo de papel e caneta que funciona do seguinte modo:

Uma palavra secreta é definida;

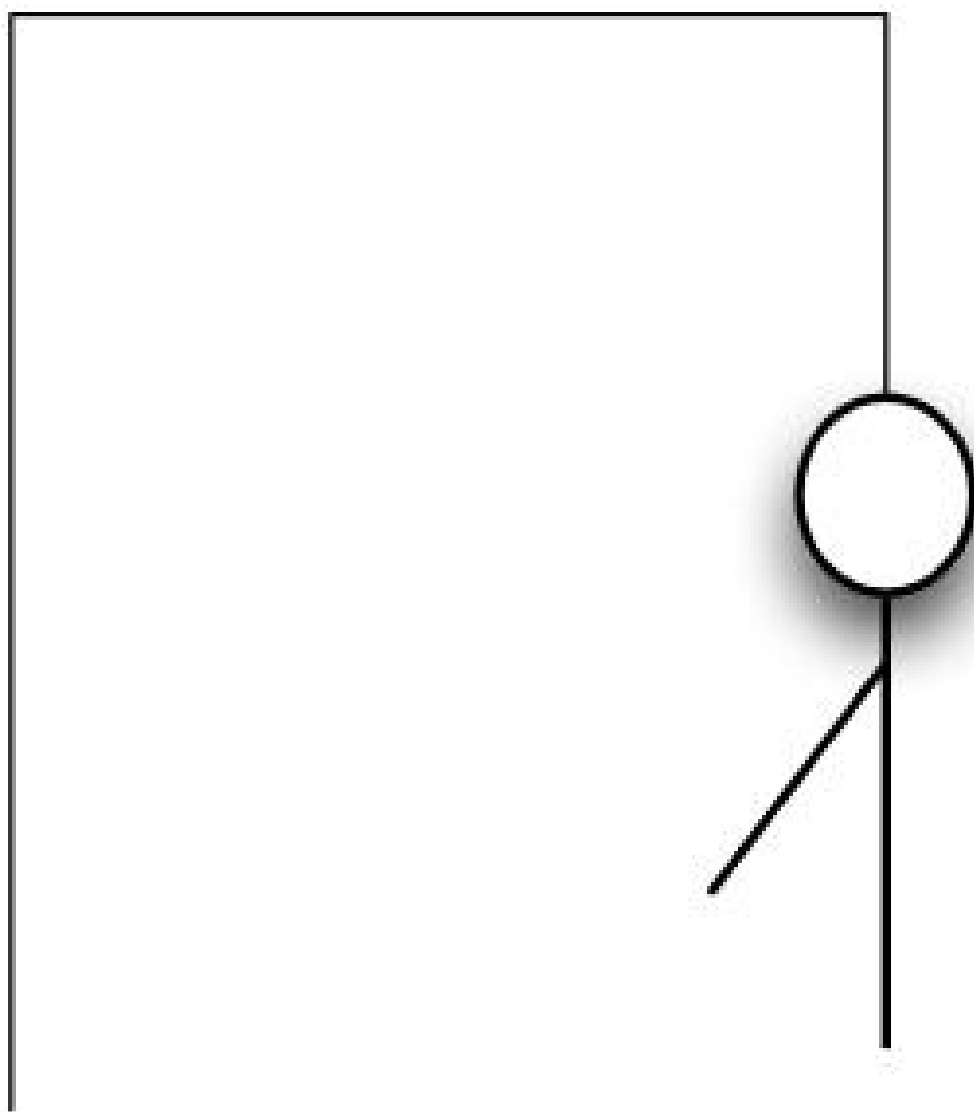
Você tenta adivinhar uma letra da palavra;

Se você acertar a letra, é mostrada a posição dessa letra dentro da palavra secreta;

Se você errar a letra, uma parte do boneco aparece na forca;

Você ganha se acertar todas as letras;

Você perde se errar o bastante para o corpo inteiro do boneco aparecer na forca.



_ i _ e

Figura 10.1: Jogo da forca com duas letras adivinhadas e três erros cometidos

Agora que já sabemos o fluxo e as regras de negócio do jogo, qual o próximo passo antes de começar a programar? Uma boa ideia é transformar esse escopo em uma lista de funcionalidades que o nosso jogo terá. A lista de funcionalidades do nosso jogo será a seguinte:

Jogador começa um novo jogo: jogo mostra a mensagem inicial e pede para o jogador sortear uma palavra;

Jogador adivinha uma letra da palavra: jogador tenta adivinhar uma letra e o jogo mostra se ele acertou ou errou;

Fim do jogo: o jogo termina quando ou jogador acerta todas as letras ou erra o bastante para o corpo inteiro do boneco aparecer na forca.

Baseados nessa lista de funcionalidades, já podemos começar a desenvolver nosso projeto.

10.2 Especificando uma funcionalidade com Cucumber

Fazendo o setup do projeto

Antes de tudo, vamos criar um diretório onde ficará o código do nosso projeto. Para isso, abra o seu terminal e digite os seguintes comandos:

```
$ mkdir forca
```

```
$ cd forca
```

Atualmente, todo projeto em Ruby usa o Bundler para gerenciar suas dependências. Como vamos usar Cucumber e RSpec, vamos criar um Gemfile e listar as dependências do nosso projeto.

Instale o Bundler e crie um Gemfile, fazendo os seguintes comandos no terminal:

```
$ gem install bundler
```

```
$ bundle init
```

Depois, adicione o Cucumber e RSpec no Gemfile:

```
# frozen_string_literal: true
```

```
source
```

```
"https://rubygems.org"
```

```
git_source(
```

```
:github) do
```

```
|repo_name|
```

```
"https://github.com/#{repo_name}" end
```

```
gem
```

```
"cucumber"
```

```
gem
```

"rspec"

Finalmente, para instalar as dependências, basta fazer `bundle install`. Após tudo isso, você já deve ter o Cucumber e o RSpec instalados.

Com o Cucumber instalado, podemos rodar o comando `init` para criar a estrutura padrão de arquivos e diretórios:

```
$ bundle exec cucumber --init
```

Agora estamos prontos para criar a primeira especificação do nosso jogo da forca.

Definindo os steps do primeiro cenário

Baseado na lista de funcionalidades que definimos, vamos começar especificando a funcionalidade Jogador começa um novo jogo. Para isso, crie um arquivo chamado `features/comecar_jogo.feature` e escreva nele o seguinte:

```
# language: pt
```

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo na tela:

""""

Bem-vindo ao jogo da forca!

""""

Com essa especificação escrita, ao rodarmos o Cucumber fazendo bundle exec cucumber, veremos a seguinte saída na tela:

```
$ bundle exec cucumber
```

```
# language: pt
```

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então vejo na tela:

""""

Bem-vindo ao jogo da forca!

""""

1 scenario (1 undefined)

2 steps (2 undefined)

0m0.066s

You can implement step definitions for undefined steps with these snippets:

```
Quando("começo um novo jogo") do  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

```
Então("vejo na tela:") do |string|  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

Perceba que o Cucumber nos mostra que 1 cenário foi executado (1 scenario (1 undefined)), mas que temos 2 steps indefinidos (2 steps (2 undefined)). Além disso, ele já nos dá um guia inicial de como definir esses steps:

```
Quando("começo um novo jogo") do  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

```
Então("vejo na tela:") do |string|  
  pending # Write code here that turns the phrase above
```



```
# into concrete actions  
  
end
```

Para começar a definir esses steps, vamos nos basear nesse guia. Crie um arquivo chamado `features/step_definitions/game_steps.rb` e escreva o seguinte nele:

Quando "começo um novo jogo" do

```
    pending  
end
```

Então "vejo na tela:" do

```
|string|  
    pending  
end
```

Se você rodar o Cucumber agora, verá que dessa vez ele fala que temos 1 cenário pending, 1 step pending e 1 step skipped:

```
$ bundle exec cucumber
```

(...)

1 scenario (1 pending)

2 steps (1 skipped, 1 pending)

0m0.055s

Está na hora de começarmos a implementar nosso primeiro step.

Implementando o primeiro step

O primeiro step que vamos implementar é o Quando começo um novo jogo. Seguindo o que definimos na descrição desse step, só precisamos colocar na sua implementação o código que vai começar um novo jogo. Para isso, vamos imaginar que o nosso sistema terá uma classe `Game`, que será responsável pelo jogo rodando, e um método `Game#start`, que começará um novo jogo. Seguindo essa ideia, escreva o seguinte código no arquivo `features/step_definitions/game_steps.rb`:

Quando "começo um novo jogo" do

game =

`Game`

```
.new  
  
  game.start  
  
end
```

Repare que estamos usando uma classe Game e um método Game#start antes mesmo de eles serem implementados. Isso não é um problema, faz parte do fluxo do BDD usar um código que você gostaria que existisse e deixar os testes nos guiar sobre quando devemos implementar esse código.

Seguindo o fluxo de BDD, ao rodar o Cucumber, o teste falha e recebemos a seguinte mensagem:

```
$ bundle exec cucumber
```

Quando começo um novo jogo

```
uninitialized constant Game (NameError)
```

```
./features/step_definitions/game_steps.rb:2:
```

```
  in `"começo um novo jogo"
```

```
features/comecar_jogo.feature:12:
```

```
  in `Quando começo um novo jogo'
```

O feedback que recebemos do teste é que a constante Game ainda não foi definida. Vamos defini-la criando a classe Game. Crie o diretório lib e, dentro

dele, crie o arquivo lib/game.rb com o seguinte conteúdo:

```
class Game end
```

Depois disso, ao rodar o Cucumber novamente, ele continua nos falando que a constante Game ainda não foi definida. Isso porque não configuramos o Cucumber para carregar nossa nova classe. Para que ele possa carregá-la, crie o arquivo features/support/env.rb e dentro dele escreva:

```
require "game"
```

Agora, rodando o Cucumber, recebemos a seguinte mensagem:

```
$ bundle exec cucumber
```

Quando começo um novo jogo

```
undefined method `start' for #<Game:0x00007fac> (NoMethodError)
```

```
./features/step_definitions/game_steps.rb:3:
```

```
  in `"começo um novo jogo"
```

```
features/comecar_jogo.feature:12:
```

```
  in `Quando começo um novo jogo'
```

O erro mudou, e isso é bom porque é um indicador de que estamos avançando. Essa nova mensagem de erro nos fala que não existe um método chamado start na classe Game. Neste momento, temos duas opções:

Implementar o método start na classe Game;

Escrever um teste para especificar o método start antes de começarmos a implementá-lo.

Como bons praticantes de BDD (e também porque você está lendo um livro sobre BDD), vamos escolher a segunda opção: escrever um teste antes de começar a implementar. Vamos usar o RSpec para escrever esse teste.

10.3 Usando RSpec no nosso primeiro teste

Antes de começar a usar o RSpec, precisamos inicializá-lo no nosso projeto. Para isso, execute o seguinte comando no terminal:

```
$ bundle exec rspec --init
```

Após inicializar o RSpec, crie o arquivo de teste spec/game_spec.rb. O primeiro teste que vamos escrever é para o método `Game#start`. Ao executar esse método, deve ser impressa na tela a mensagem inicial do jogo.

Para que possamos testar esse comportamento, podemos usar o RSpec para verificar se a mensagem inicial foi impressa na tela:

```
require "game"
```

```
RSpec.describe Game do
```

```
  describe
```

```
"#start" do
```

```
  it
```

```
    "prints the initial message" do
```

```
      game =
```

```
        Game
```

```
        .new
```

```
      game.start
```

```
      initial_message =
```

```
        "Bem-vindo ao jogo da forca!"
```

```
      expect(
```

```
        STDOUT).to include
```

```
        (initial_message)
```

```
    end
```

end end

O que fizemos nesse teste foi:

criamos uma instância da classe Game (fase de setup do teste);

depois chamamos o método start (fase de exercício do teste);

verificamos se o método start imprimiu na tela a mensagem inicial do jogo (fase de verificação do teste).

Ao rodarmos o RSpec, executando bundle exec rspec, veremos a seguinte mensagem:

```
$ bundle exec rspec
```

```
F
```

Failures:

1) Game#start prints the initial message

Failure/Error: game.start

NoMethodError:


```
undefined method `start' for #<Game:0x00007fcd7aa1e768>
```

```
# ./spec/game_spec.rb:8:in `block (3 levels)'
```

Finished in 0.00662 seconds (files took 0.24057 seconds to load)

1 example, 1 failure

Ou seja, nosso teste falhou, está vermelho. A mensagem de falha nos diz que o método start não foi definido na classe Game. Agora chegou a hora de implementarmos esse método. A única coisa que ele deverá fazer é imprimir na tela a mensagem inicial do jogo. Para fazer isso, escreva o seguinte na classe Game:

```
class Game
```

```
  def
```

```
    start
```

```
      initial_message =
```

```
        "Bem-vindo ao jogo da forca!"
```

```
        puts initial_message
```

end end

Ao rodarmos o RSpec novamente, vemos a seguinte mensagem:

```
$ bundle exec rspec
```

1) Game#start prints the initial message

Failure/Error: expect(STDOUT).to include(initial_message)

IOError:

not opened for reading

O problema agora está no modo como estamos testando se a mensagem inicial foi impressa com sucesso. Estamos fazendo isso tentando ler do STDOUT, mas no Ruby não é possível ler do STDOUT — a mensagem "IOError: not opened for reading" está nos dizendo isso. Como podemos testar esse comportamento? Não parece tão simples.

Quando um teste começa a ficar difícil de escrever, é provável que ele "queira" nos dar um feedback de que alguma coisa pode ser melhorada. Portanto, vamos pensar no que o nosso método start está fazendo, como estamos tentando testá-lo e o que podemos melhorar.

Para que o método start funcione, ele precisa fazer uma chamada ao método

puts. No Ruby, o método puts é definido no módulo Kernel. Logo, ao fazer puts initial_message dentro do método start, ele está na verdade dependendo do método Kernel#puts para funcionar. No entanto, essa dependência não está explícita, e é por isso que está difícil de testar esse método.

TDD "nos força" a deixar explícitas as dependências de nossos objetos. No caso da classe Game, podemos deixar explícita a dependência de Kernel#puts, injetando-a na classe Game.

Vamos fazer isso adicionando um parâmetro no construtor da classe Game. Esse parâmetro se chamará output, representando a dependência de um objeto que possa imprimir strings na tela. Além disso, para facilitar a instanciação de um objeto da classe Game, vamos setar o STDOUT como valor default desse parâmetro.

Abra a classe Game e adicione esse novo parâmetro no construtor escrevendo o seguinte:

```
class Game
```

```
  def initialize(output = STDOUT  
  )
```

```
    @output
```

```
= output
```

```
end
```

```
def
```

```
start
```

```
  initial_message =
```

```
  "Bem-vindo ao jogo da forca!"
```

```
@output
```

```
.puts initial_message
```

```
end end
```

Repare que, com a dependência output salva na variável de instância @output, mudamos também a implementação do método start para fazer @output.puts em vez de somente puts. Agora abra o arquivo game_spec.rb e o modifique para ficar assim:

```
RSpec.describe Game do
```

```
  describe
```

```
    "#start" do
```

```
      it
```

```
        "prints the initial message" do
```

```
          output = double(
```

```
            "output"
```

```
          )
```

```
          game =
```

```
            Game
```

```
              .new(output)
```

```
            initial_message =
```

```
              "Bem-vindo ao jogo da forca!"
```

```
            expect(output).to receive(
```

```
              :puts
```

```
            ).with(initial_message)
```

```
game.start
```

```
end
```

```
end end
```

Finalmente, ao rodar os testes, vemos que eles passam!

```
$ bundle exec rspec
```

```
.
```

```
Finished in 0.03037 seconds (files took 0.23934 seconds to load)
```

```
1 example, 0 failures
```

Após comemorar que nosso primeiro teste passou, vale a pena entender melhor o que fizemos para ele passar.

A dificuldade estava em testar que o método `start` estava imprimindo a mensagem inicial com sucesso. Na implementação anterior do método `start`, ele dependia do método `Kernel#puts`, mas essa dependência não estava explícita na

hora de construir um objeto da classe Game. Para resolver o problema, deixamos explícito que o objeto game depende de algum objeto que sabe imprimir strings. Fizemos isso ao injetar essa dependência no construtor da classe Game:

```
class Game
```

```
def initialize(output = STDOUT  
)
```

```
@output
```

```
= output
```

```
end end
```

Uma vantagem de deixar essa dependência explícita é que isso vira um ponto de extensão da classe. Ela não depende mais diretamente do STDOUT, ela depende de um objeto qualquer que sabe imprimir strings. Ou seja, estamos usando duck typing aqui. No ambiente de produção, esse objeto é o STDOUT, mas nos testes podemos trocá-lo por outro, desde que esse objeto "se pareça" com um que imprime strings. E foi o que fizemos, valemo-nos de duck typing e de injeção de dependência.

Para testar se o método start está funcionando, não precisamos mais ver se a string foi impressa na tela, basta verificarmos se a interação entre o objeto game

e a dependência dele foi feita corretamente. Foi isso que fizemos ao usar um mock nesse teste, estamos testando se o objeto game chama o método puts da sua dependência quando o método game.start é executado:

```
it "prints the initial message" do
```

```
  game =
```

```
    Game
```

```
    .new(output)
```

```
    expect(output).to receive(
```

```
      :puts
```

```
    ).with(initial_message)
```

```
    game.start
```

```
  end
```

Resumindo, deixamos de testar o estado final do sistema (string impressa na tela) para testar a interação entre o objeto e sua dependência. Fizemos isso usando RSpec Mocks. Essa técnica de fazer TDD testando a comunicação entre os objetos, e não o estado deles, é bem explicada no famoso livro Growing Object-Oriented Software, Guided by Tests, escrito por Steve Freeman e Nat Pryce (2009) e também no livro em português Test-Driven Development: Teste e Design no Mundo Real, escrito pelo Mauricio Aniche (2012).

Por fim, você pode rodar o Cucumber novamente e ver que o primeiro step que antes não passava, agora está passando:

```
$ bundle exec cucumber
```

```
(...)
```

```
1 scenario (1 pending)
```

```
2 steps (1 pending, 1 passed)
```

```
0m0.015s
```

Estamos prontos para ir para o segundo step!

10.4 Usando Aruba para testar uma aplicação CLI

O nosso primeiro step (Quando começo um novo jogo) já está passando, vamos agora pensar no segundo, que é:

Então vejo na tela:

```
*****
```

```
Bem-vindo ao jogo da forca!
```

```
*****
```

Precisamos testar que, após o jogador iniciar o jogo, essa mensagem inicial é impressa na tela.

No ambiente de produção, o jogo vai imprimir no STDOUT, ou seja, no console. Mas, como vimos no teste de unidade da classe Game, não é possível verificar o que foi impresso no STDOUT. Para resolver esse problema no teste com RSpec, nós usamos mocks. Desse modo, conseguimos testar a classe Game isoladamente, independente da implementação real de suas dependências. No teste com Cucumber faremos diferente.

Testes com Cucumber são testes de aceitação. Nesse tipo de teste, idealmente, o

software deve ser testado do mesmo modo que o usuário final o utiliza. Por isso, não queremos usar mocks aqui. Não queremos fazer um teste isolado, queremos fazer um teste que exercite todas as camadas do nosso software de modo integrado.

Acontece que, até agora, o nosso jogo ainda não tem uma interface com o usuário. Então antes de escrever testes de Cucumber que testem a partir da interface com o usuário, precisamos primeiro criar essa interface. Para fazer isso, crie o arquivo bin/forca com o seguinte conteúdo:

```
#!/usr/bin/env ruby
```

```
$LOAD_PATH.prepend File.join(__dir__, "../lib")
```

```
require "game"
```

```
game =
```

```
  Game
```

```
  .new
```

```
  game.start
```

E defina esse arquivo como um executável:

```
$ chmod +x bin/forca
```

Esse arquivo será o binário do nosso jogo. É ele quem iniciará a aplicação. Assim, bastará que o jogador rode esse arquivo para ver a seguinte saída:

```
$ bin/forca
```

Bem-vindo ao jogo da forca!

■

Como lidar com binários em Ruby no Windows

No Windows você não pode fazer `chmod +x` para tornar um arquivo de Ruby executável, é necessário criar um arquivo `.bat` para que o arquivo original seja tratado como executável. Para fazer isso, além de ter criado o arquivo `bin/forca`, como você fez anteriormente, será necessário criar também um arquivo `bin/forca.bat` com o seguinte conteúdo:

```
@ "ruby.exe" "%~dpn0" %*
```

■

Agora que o nosso jogo tem uma interface com o usuário, precisamos atualizar o step definition Quando "começo um novo jogo", para que ela utilize essa interface. Para implementar step definitions que interagem com uma aplicação de linha de comando, utilizaremos uma gem chamada Aruba.

O Aruba contém uma coleção de step definitions de Cucumber feitos para facilitar o desenvolvimento de testes que interagem com uma aplicação CLI (Command Line Interface). Ao utilizar o Aruba, ficará fácil verificar o que foi impresso na tela. Sem ele, isso seria difícil, pois teríamos que executar nosso jogo em outro processo, que não o processo rodando nosso teste, e conversar com ele através do seu STDIN e STDOUT. Isso tudo, e mais um pouco, o Aruba é quem fará por nós.

Para instalar o Aruba, adicione-o no Gemfile:

```
gem "cucumber"
```

```
gem
```

```
"rspec"
```

```
gem
```

```
"aruba"
```

E atualize o seu bundle, com o comando `bundle install`.

Para fazer seu setup, é necessário dar um `require` dele no arquivo `features/support/env.rb`. Abra o arquivo `features/support/env.rb` e o edite para adicionar esse `require`:

```
require "game"
```

```
require "aruba/cucumber"
```

■

Para usuários de Windows

Alguns usuários de Windows podem ter um problema de encoding ao rodar o Cucumber com Aruba, e receberão algo parecido com a seguinte mensagem de erro:

Incompatible character encodings:

IBM437 and UTF-8 (Encoding::CompatibilityError)

Se você tiver esse problema, basta alterar o `default_encoding` para `utf-8` no arquivo `features/support/env.rb`. Faça isso adicionando a seguinte linha no topo

desse arquivo:

```
Encoding.default_external = "utf-8"
```

■

Agora precisamos modificar o step definition Quando "começo um novo jogo" do para utilizar o Aruba. Mas antes disso, vamos ver como ele está implementado até então:

Quando "começo um novo jogo" do

```
game =
```

```
Game
```

```
.new
```

```
game.start
```

```
end
```

O modo como tínhamos implementado esse step era via uma chamada direta de uma classe do nosso sistema para começar o jogo. Na implementação com Aruba faremos diferente, queremos usá-lo para começar o jogo do mesmo modo que o jogador o faria, ou seja, executando o binário bin/forca.

Para refatorar esse step definition a fim de usar o Aruba, abra o arquivo `features/step_definitions/game_steps.rb` e edite a step definition de começo de novo jogo para ficar assim:

Quando "começo um novo jogo" do

```
steps
%{
  * I run `bin/forca` interactively
}
end
```

Para começar um jogo executando o binário `forca`, usamos a step definition `* I run `command` interactively` do Aruba.

Com isso, você já pode rodar o Cucumber novamente e verificar que o step continua passando:

```
$ bundle exec cucumber
```

```
(...)
```

```
1 scenario (1 pending)
```


2 steps (1 pending, 1 passed)

Finalmente, só está faltando o último step pendente, o Então "vejo na tela:". Esse step deve verificar se uma string foi impressa na tela. Para isso, vamos utilizar o step definition Then `/^the stdout should contain "([^\"]*)"$/` do Aruba.

Edite esse step no arquivo `features/step_definitions/game_steps.rb` para ficar assim:

Então "vejo na tela:" do

|text|

steps

%{

 * the stdout should contain "

#{text}

"

}

end

Por fim, ao rodar o Cucumber, veremos que nosso primeiro cenário está no verde:

\$ bundle exec cucumber

1 scenario (1 passed)

2 steps (2 passed)

language: pt

Funcionalidade: Começar jogo

Para poder passar o tempo

Como jogador

Quero poder começar um novo jogo

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então o jogo termina com a seguinte mensagem na tela:

"""

Bem-vindo ao jogo da forca!

"""

1 scenario (1 passed)

2 steps (2 passed)

0m0.080s

Figura 10.2: Primeiro cenário no verde

Finalmente, conseguimos terminar o primeiro cenário inteiro! Estamos quase prontos para ir para o próximo passo. Mas antes de seguirmos em frente, vamos dar uma olhada em um outro modo de como poderíamos utilizar os step definitions do Aruba.

10.5 Reutilizando step definitions de um modo melhor

Testes de aceitação interagem com a aplicação pela sua camada mais externa, a UI (interface com o usuário). Como na nossa aplicação a UI é uma linha de comando, utilizamos o Aruba para fazer a interação com ela.

O Aruba oferece dois modos para o utilizarmos: via step definitions ou via support code. Vamos ver cada um desses modos.

O modo de utilizar via step definitions é o que fizemos no seguinte trecho de código:

Quando "começo um novo jogo" do

steps

%{

* I run `bin/forca` interactively

}

end

O primeiro detalhe que vale a pena notar nesse código é que está sendo usada uma feature do Cucumber para reutilizar um step definition, o método `steps`.

Esse método serve para executar um conjunto de steps no formato de uma String contendo Gherkin. No caso, passamos uma string contendo o step `* I run bin/forca interactively`, que é um step definition fornecida pelo Aruba. Para ficar mais claro, segue a mesma chamada que fizemos do método `steps` de um modo mais direto:

```
steps("* I run `bin/forca` interactively")
```

O método `steps` foi feito para você poder reutilizar lógica de uma step definition dentro de outra. Isso pode parecer uma boa ideia à primeira vista, mas existe um modo melhor para fazer reuso de lógica entre step definitions, que é o support code. Vamos ver como usar o support code do Aruba.

O Aruba oferece duas APIs públicas para seu uso: um conjunto de step definitions de Cucumber e um conjunto de helper methods que formam seu support code. O conjunto de step definitions que ele oferece você pode ver em <https://github.com/cucumber/aruba/blob/v0.14.9/lib/aruba/cucumber/command.rb>. Já o support code está disponível em <https://github.com/cucumber/aruba/blob/v0.14.9/lib/aruba/api/commands.rb>.

Todos os step definitions do Aruba têm uma estrutura comum: cada step definition é simples e delega a parte pesada para seu support code. Vamos olhar o código de um deles, por exemplo, o que mapeia o step que usamos em `* I run `bin/forca` interactively`:

```
When(/^I run `([^\`]*)` interactively$/) do
```

```
|cmd|
```

```
# ...
```

```
  run_command(sanitize_text(cmd))
```

```
end
```

Perceba como esse step definition delega toda sua lógica para o método run. Esse método faz parte do módulo `Aruba::Api::Commands`, que contém o support code do Aruba. Essa ideia de step definitions simples que delegam a lógica para o support code é a mesma que vimos na seção Support code.

Já que o Aruba foi feito utilizando essa boa prática, em vez de usarmos suas funcionalidades através dos seus step definitions, é muito melhor o usarmos através do seu support code, o módulo `Aruba::Api::Commands` (<https://github.com/cucumber/aruba/blob/v0.14.9/lib/aruba/api/commands.rb>).

Antigamente, utilizar o método steps do Cucumber era considerado usual para fazer o reúso de step definitions. Mas, ao longo do tempo, a comunidade está cada vez mais caminhando para reutilizar lógica através do support code.

A utilização do método steps para fazer um step definition chamar outro step definition segue uma ideia equivalente a fazer um método chamar outro método. O problema disso é que ela nos leva a visualizar o step definition como uma unidade básica de organização de código, semelhante a um método, o que ele

não é. Step definition é um modo de mapear um step (Gherkin) com um trecho de código para ser executado.

Visualizar step definitions como métodos traz problemas, pois não é possível tratá-los 100% como métodos. Por exemplo, imagine que você quer renomear um step definition. Seu nome pode ser variável, pois é composto pela sua regex e os "argumentos" que são passados para ela. Devido ao fato de ser variável, mudar seu nome e todo os lugares que o referenciam não é tão simples quanto fazer um "find and replace". Mudar o nome de um método é bem mais simples do que mudar o nome de um step definition.

Levando esses pontos em consideração, devemos fazer reúso de lógica de step definitions através do support code, que é formado por módulos e métodos Ruby, algo que já sabemos como organizar, separar e reutilizar.

Aplicando essa ideia no uso do Aruba, não vamos utilizar suas step definitions, como fizemos em:

Quando "começo um novo jogo" do

steps

%{

* I run `bin/forca` interactively

}

end

Agora devemos utilizar seu support code. Modifique o step definition que acabamos de ver para utilizar o método run do support code do Aruba:

Quando "começo um novo jogo" do

```
run_command(  
  "bin/forca") end
```

Para modificar o outro step definition, vamos utilizar a documentação do Aruba (<https://app.cucumber.pro/projects/aruba/documents/branch/master/>). Hoje ele está assim:

Então "vejo na tela:" do

```
|text|  
  
  steps  
  
    %{  
      * the stdout should contain "  
      #{text}  
      "  
    }  
  
  end
```

Podemos reescrever o step definition desta forma:

Então "vejo na tela:" do

|text|

expect(last_command_stopped.stdout).to

include(text) end

Estamos usando o `last_command_stopped` do Aruba para referenciar a execução do forca, e o método `stdout` que dá acesso a tudo que foi impresso na tela.

Com isso, terminamos a refatoração dos nossos step definitions, para fazer reuso do Aruba através do seu support code. Esse caso do reuso do Aruba é um bom exemplo de como devemos organizar nossas step definitions, que devem ser simples e a lógica pesada deve ir para o support code.

No capítulo seguinte continuaremos o desenvolvimento dessa primeira funcionalidade. Mas antes disso, vale a pena lembrarmos dos pontos-chaves que acabamos de aprender.

Pontos-chaves deste capítulo

Começamos o capítulo entendendo o escopo da nossa aplicação e especificando a primeira funcionalidade com Cucumber.

Com a primeira spec de Cucumber no vermelho, seguimos o fluxo de outside-in development, fazendo um teste de RSpec, implementando código para vê-lo passar e, por fim, fazer o Cucumber passar.

Ao fazer os primeiros testes de unidade com RSpec, vimos que TDD nos forçou a deixar explícitas as dependências de um objeto, quando foi necessário injetar a dependência output na classe Game. Esse é um dos pontos nos quais TDD nos leva a fazer um código melhor, porque ao deixar as dependências explícitas, fica mais fácil de mudar o comportamento do nosso software simplesmente trocando a implementação de alguma dependência.

Vimos também que ao usarmos o Cucumber como ferramenta de testes de aceitação, devemos enxergar o software sob teste como uma caixa preta. Para seguir essa ideia no caso de uma aplicação de linha de comando, usamos o Aruba, que é uma extensão do Cucumber para testar aplicações CLI.

Por fim, vimos como usar o Aruba através do seu support code, em vez de utilizar o método steps do Cucumber para reutilizar step definitions. Prefira sempre fazer reuso através da camada de support code. Isso vai ajudar na manutenção de nossa especificação executável e nos fará ter menos dores de cabeça no futuro.

Capítulo 11

Começando o segundo cenário

No capítulo anterior, definimos o escopo do nosso projeto, fizemos o setup e começamos a desenvolver e testar a primeira funcionalidade do jogo da forca. Neste capítulo, continuaremos o desenvolvimento dessa funcionalidade, definindo o seu segundo cenário e fazendo as mudanças e refatorações necessárias para implementá-lo. Vamos lá!

11.1 Definindo o segundo cenário

Até agora já implementamos o primeiro cenário da funcionalidade "Começar jogo". Mas, para começar o nosso jogo, não basta que o jogador veja a mensagem inicial do jogo — é preciso também sortear a palavra que deverá ser adivinhada, a partir de um tamanho de palavra definido pelo jogador. Nesse sorteio, teremos os seguintes cenários:

Sorteio da palavra com sucesso: quando o jogador escolhe o tamanho da palavra a ser sorteada e o jogo consegue sortear uma palavra com esse tamanho;

Sorteio da palavra sem sucesso: quando o jogador escolhe o tamanho da palavra a ser sorteada, mas o jogo não tem uma palavra com esse tamanho.

Vamos começar definindo o cenário de sucesso. Para isso, abra o arquivo `comecar_jogo.feature` e adicione o seguinte conteúdo nele:

@wip

Cenário: Sorteio da palavra com sucesso

Após o jogador começar o jogo, ele deve escolher o tamanho da palavra a ser adivinhada. Ao escolher o tamanho, o jogo

sorteia a palavra e mostra na tela um

" _ "

para cada letra que

a palavra sorteada tem.

Dado que comecei um jogo

Quando escolho que uma palavra com

"4"

letras deve ser sorteada

Então vejo na tela:

""""

_ _ _ _

""""

Perceba que colocamos a tag @wip (work in progress) nesse cenário, para conseguir executá-lo sozinho. Ao executarmos esse cenário no terminal, vemos o seguinte:

```
$ bundle exec cucumber --tags @wip
```

(...)

1 scenario (1 undefined)

3 steps (1 skipped, 2 undefined)

Ou seja, dos 3 steps do nosso cenário, 2 ainda estão indefinidos. Vamos começar a trabalhar nisso.

11.2 Reduza duplicação através de support code

O primeiro step indefinido é o Dado que comecei um jogo. É muito parecido com o step Quando começo um novo jogo que implementamos no capítulo anterior. Na verdade, queremos que ambos os steps façam a mesma coisa: iniciem um jogo.

Poderíamos ter reutilizado o mesmo step, mas não o fizemos de propósito, porque escrevemos o texto do novo step pensando principalmente do ponto de vista de quem o está lendo. Seus steps devem ser escritos otimizados mais para leitura do que para implementação. Lembre-se, o Cucumber não é só uma ferramenta de automatização de testes de aceitação, mas sim uma ferramenta de especificação de software e principalmente um modo de comunicar o que o seu software faz.

Para definir o novo step Dado que comecei um jogo, poderíamos simplesmente copiar e colar a definição do step Quando começo um novo jogo, mas isso geraria duplicação no nosso código. Para evitar isso, vamos extrair a parte que ficaria duplicada para o support code do Cucumber.

Crie o arquivo `features/support/game_helpers.rb`, e implemente o helper method `start_new_game`, que será responsável por iniciar um novo jogo usando o Aruba:

```
module GameHelpers
```

```
def
  start_new_game
    run_command(
      "bin/forca"
    )
end end
```

World(GameHelpers)

Repare que, além de termos criado o módulo GameHelpers, também incluímos os métodos desse módulo no contexto dos step definitions, usando o método World do Cucumber, como vimos na seção Cucumber World.

Agora edite o step Quando "começo um novo jogo" do arquivo features/step_definitions/game_steps.rb para que ele chame o nosso novo helper method start_new_game:

Quando "começo um novo jogo" do

```
start_new_game
```

```
end
```

Ao rodar o Cucumber, você verá que o cenário Começo de novo jogo com sucesso continua no verde. Com isso, podemos adicionar o novo step no arquivo `features/step_definitions/game_steps.rb`:

Dado "que comecei um jogo" do

```
start_new_game
```

```
end
```

Rodando o Cucumber para o novo cenário, vemos que um step já está passando, mas que o Quando escolho que uma palavra com "4" letras deve ser sorteada continua indefinido:

```
$ bundle exec cucumber --tags @wip
```

```
(...)
```

```
3 steps (1 skipped, 1 undefined, 1 passed)
```

Vamos implementar esse último step.

11.3 Implementando o fluxo do jogo no binário

O próximo step a ser implementado é o Quando escolho que uma palavra com "4" letras deve ser sorteada. Para implementá-lo, vamos adicionar um step definition no arquivo `features/step_definitions/game_steps.rb` que usará o Aruba para simular um jogador digitando o tamanho da palavra a ser sorteada no jogo.

O Aruba possui um método chamado `type` que recebe como um argumento uma string e simula um usuário digitando um texto no console. Podemos usar esse método para implementar nosso step do seguinte modo:

Quando "escolho que uma palavra com {string} letras "

\

"deve ser sorteada" do

|number_of_letters|

type(number_of_letters)

end

Ao rodarmos o Cucumber deste cenário, vemos o seguinte:

```
$ bundle exec cucumber --tags @wip
```

Então vejo na tela:

```
""
```

```
----
```

```
""
```

expected "Bem-vindo ao jogo da forca!\n" to include "----"

Diff:

```
@@ -1,2 +1,2 @@
```

```
- ----
```

```
+ Bem-vindo ao jogo da forca!
```

(RSpec::Expectations::ExpectationNotMetError)

./features/step_definitions/game_steps.rb:6:

```
in `"veja na tela:"
```

features/comecar_jogo.feature:27:in `Então vejo na tela:'

Failing Scenarios:

cucumber features/comecar_jogo.feature:19

O cenário falhou, mas por quê? O motivo é que ainda precisamos construir o

resto da funcionalidade. O que está faltando é:

o jogador poder escolher o tamanho da palavra a ser sorteada;

o jogo sorteia essa palavra;

o jogo imprime na tela um caractere _ para cada letra da palavra sorteada.

Precisamos modificar o binário do nosso jogo para implementar esses passos. Até então o binário só inicia o jogo e imprime a mensagem inicial:

```
#!/usr/bin/env ruby
```

```
$LOAD_PATH.prepend File.join(__dir__, "../lib")
```

```
require "game"
```

```
game =
```

```
Game
```

```
.new
```

game.start

Precisamos modificá-lo para que ele peça para o jogador o tamanho da palavra a ser sorteada, depois a sorteie e continue rodando até que o jogo acabe. Ou seja, precisamos modificar o binário para que ele rode o fluxo do jogo como um todo. Nossa ideia não é implementar o código do fluxo do jogo em si no binário, mas sim que o binário dependa de métodos do nosso sistema que serão responsáveis por controlar esse fluxo.

Para controlar o fluxo do jogo, podemos imaginar que a classe Game terá dois novos métodos, ended? e next_step. O método ended? vai dizer se o jogo terminou ou não. E o método next_step vai executar o próximo passo do jogo, baseado no seu estado atual.

Vamos adicionar o uso desses novos métodos no binário do nosso jogo. Modifique o arquivo bin/forca para que ele fique assim:

```
#!/usr/bin/env ruby
```

```
$LOAD_PATH.prepend File.join(__dir__, "../lib"  
)
```

```
require "game"
```



```
game =  
Game  
  .new  
  game.start  
  
until  
  game.ended?  
    game.next_step  
end
```

O que fizemos nesse código foi o seguinte: após o jogo ser inicializado e mostrar a mensagem inicial, ele entra em um loop e fica rodando o próximo passo do jogo até o jogo terminar, utilizando o método `Game#ended?` para controlar o loop e o método `Game#next_step` para executar o passo seguinte do jogo.

Logo após o jogo começar, a palavra ainda não foi sorteada, então o próximo passo é pedir para o jogador o tamanho da palavra a ser sorteada e sorteá-la. Em seguida, o próximo passo é o jogo pedir para o jogador adivinhar uma letra da palavra. Os passos seguintes consistem no jogador tentar adivinhar as letras da palavra, até que ele adivinhe todas e ganhe, ou perca porque errou demais.

Antes de começarmos a implementar o próximo passo do fluxo do jogo, vamos rodar o novo binário.

```
$ bin/forca
```

Bem-vindo ao jogo da forca!

Traceback (most recent call last):

```
bin/forca:10:in `<main>': undefined method `ended?' for
```

```
#<Game:0x00007ff39709b6 @output=#<IO:<STDOUT>>> (NoMethodError)
```

O erro foi que o método `Game#ended?` ainda não foi construído. Vamos rodar o Cucumber e ver se aparece a mesma mensagem de erro:

```
$ bundle exec cucumber
```

Então vejo na tela:

```
""""
```

```
 _ _ _ _
```

```
""""
```

```
expected "Bem-vindo ao jogo da forca!\n" to include "_ _ _ _"
```

Diff:

```
@@ -1,2 +1,2 @@
```

- _ _ _ _

+Bem-vindo ao jogo da forca!

(RSpec::Expectations::ExpectationNotMetError)

./features/step_definitions/game_steps.rb:6:in `"veja na tela:"

features/comecar_jogo.feature:27:in `Então vejo na tela:'

Failing Scenarios:

cucumber features/comecar_jogo.feature:19

Cenário: Sorteio da palavra com sucesso

2 scenarios (1 failed, 1 passed)

5 steps (1 failed, 4 passed)

Não apareceu a mesma mensagem de erro. Rodando o binário, o feedback que recebemos é que o método `Game#ended?` ainda precisa ser implementado. Ao rodar com Cucumber, não recebemos o mesmo feedback.

Executando o binário, nosso jogo quebra, rodando com Cucumber o primeiro cenário chega até a passar e o segundo cenário quebra com uma mensagem de erro diferente da que aparece quando quebra rodando na mão. Isso não está legal, pois como o Cucumber deve testar o software do mesmo ponto de vista do usuário final, se algo está quebrando do ponto de vista do usuário (rodando diretamente o binário), ele deveria quebrar nos testes do Cucumber também. Precisamos consertar esse problema.

11.4 Modificando nosso cenário para receber o feedback correto

Vimos que rodar o jogo na mão está quebrando e rodá-lo com Cucumber não está quebrando. Isso é ruim pois testes do Cucumber devem nos dizer se o software está quebrando do ponto de vista do usuário.

O usuário final e o Cucumber estão iniciando o jogo do mesmo modo, usando o binário força, com a diferença que o Cucumber está fazendo isso via Aruba.

Para que o teste com Cucumber tenha um resultado final mais parecido com rodar o jogo na mão, vamos verificar que nenhuma mensagem de erro foi exibida. Vamos usar uma funcionalidade que todo processo tem, que é o STDERR, uma saída especial do processo onde todas as mensagens de erro são imprimidas.

As mensagens de erro do STDERR são exibidas normalmente para o usuário com as demais mensagens, porém podemos capturá-las de forma separada para identificar que uma determinada mensagem é relacionada a um problema na aplicação.

Também vamos conferir se o processo do jogo foi finalizado com sucesso. Vamos usar o exit status do processo, um número que indica se um processo foi finalizado sem erro. Podemos aproveitar os matchers que o Aruba já oferece para verificar esses cenários.

Vamos mudar nosso arquivo de feature features/comecar_jogo.feature, colocando um step que vai fazer isso. Vamos trocar o antigo step Então vejo na tela: por um novo step, o Então o jogo termina com a seguinte mensagem na tela:. Modifique os cenários do arquivo features/comecar_jogo.feature para utilizar esse novo step:

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o jogador.

Quando começo um novo jogo

Então o jogo termina com a seguinte mensagem na tela:

""""

Bem-vindo ao jogo da forca!

""""

@wip

Cenário: Sorteio da palavra com sucesso

Após o jogador começar o jogo, ele deve escolher o tamanho da palavra a ser adivinhada. Ao escolher o tamanho, o jogo sorteia a palavra e mostra na tela um

" _ "

para cada letra que a palavra sorteada tem.

Dado que comecei um jogo

Quando escolho que uma palavra com

"4"

letras deve ser sorteada

Então o jogo termina com a seguinte mensagem na tela:

""

_ _ _ _

""

Ao rodar o Cucumber, veremos que esse step ainda está indefinido. Vamos defini-lo. Ao adicionar o novo step definition no arquivo `features/step_definitions/game_steps.rb` ele ficará assim:

Dado "que comecei um jogo" do

start_new_game

end

Quando "começo um novo jogo" do

start_new_game

end

Quando "escolho que uma palavra com {string} letras "

\

"deve ser sorteada" do

|number_of_letters|

type(number_of_letters)

end

Então "o jogo termina com a seguinte mensagem na tela:" do

|text|

```
expect(last_command_stopped).not_to have_output_on_stderr
```

```
expect(last_command_stopped).to be_successfully_executed
```

```
expect(last_command_stopped.stdout).to
```

```
include(text) end
```

Deletamos o antigo step Então vejo na tela: e adicionamos o novo step Então o jogo termina com a seguinte mensagem na tela:. Usamos os seguintes matchers do Aruba:

`expect(last_command_stopped).not_to have_output_on_stderr`: verifica que nada foi impresso no STDERR, isto é, que nenhuma mensagem de erro foi exibida;

`expect(last_command_stopped).to be_successfully_executed`: verifica que o exit status do processo sob teste foi de sucesso.

Ao rodar o Cucumber novamente, vemos o seguinte:

```
$ bundle exec cucumber
```

Então o jogo termina com a seguinte mensagem na tela:

(...)

expected `forca` not to have output on stdout

but was:

```
bin/forca:10:in `<main>': undefined method `ended?'
```

```
for #<Game:0x00007fd7 @output=#<IO:<STDOUT>>> (NoMethodError)
```

(...)

2 scenarios (2 failed)

5 steps (2 failed, 3 passed)

A mensagem de erro nos diz que o programa quebrou com a mensagem de exceção "undefined method `ended?'", o mesmo feedback que recebemos ao rodar o jogo na mão. Agora sim os testes com Cucumber refletem a realidade! Com o feedback correto do Cucumber, podemos seguir em frente.

11.5 Usando subject e let do RSpec para evitar duplicação nos testes

Ao rodar o Cucumber, vemos que ambos os cenários quebram, e quebram com a seguinte mensagem: "undefined method `ended?' for #<Game:>". Ou seja, precisamos implementar o método Game#ended?.

Como vimos no arquivo bin/forca, o método Game#ended? será usado para sinalizar se o jogo deve continuar rodando ou não. O jogo deve continuar rodando ou até que o jogador peça para ele terminar, ou até que ele ganhe ou até que ele perca. Vamos começar fazendo um teste para cobrir o cenário de quando o jogo acabou de começar.

Abra o arquivo spec/game_spec.rb e escreva esse teste do seguinte modo:

```
describe "#ended?" do
```

```
  it
```

```
    "returns false when the game just started" do
```

```
      game =
```

Game

.new

expect(game).not_to be_ended

end end

Ao rodarmos o RSpec, esse teste quebra porque falta implementar o método `Game#ended?`, vamos implementá-lo.

Esse método terá que retornar `true` ou `false` conforme o jogo terminou ou não. Vamos salvar a informação se o jogo terminou ou não em uma variável de instância chamada `@ended`, e utilizá-la para implementar o método `Game#ended?`.

Seguindo essa ideia, adicione o método `ended?` no arquivo `lib/game.rb` e modifique o método `initialize` para ficar assim:

```
def initialize(output = STDOUT  
  
)
```

```
@output
```

```
= output
```

```
@ended = false end
```

```
def
```

```
ended?
```

```
@ended end
```

Agora, ao rodar o RSpec, vemos que o novo teste passa. Até então, nossos testes da classe Game estão assim:

```
RSpec.describe Game do
```

```
  describe
```

```
    "#start" do
```

```
      it
```

```
        "prints the initial message" do
```

```
          output = double(
```

```
            "output"
```

```
)
```

```
  game =
```

```
    Game
```

```
      .new(output)
```

```
        initial_message =
```

```
        "Bem-vindo ao jogo da forca!"
```

```
        expect(output).to receive(
```

```
          :puts
```

```
        ).with(initial_message)
```

```
        game.start
```

```
      end
```

```
    end
```

```
  describe
```

```
"#ended?" do

  it

  "returns false when the game just started" do

    game =

  Game

  .new

    expect(game).not_to be_ended

  end

end end
```

Perceba que a criação do objeto game está duplicada entre os dois testes. Vamos extrair a instanciação do objeto game para um subject. Além disso, vamos aproveitar para extrair a definição do colaborador output para um let.

Pessoalmente, eu gosto de deixar a definição de todos os colaboradores do objeto sob teste no começo do teste usando let. Desse modo, fica bem claro quais são as dependências do objeto sob teste.

Após essa redução de duplicidade, nosso teste ficará assim:

```
RSpec.describe Game do
```

```
  let(
    :output) { double("output"
  ) }
```

```
  subject(
    :game) { Game
    .new(output) }
```

```
  describe
    "#start" do
```

```
    it
    "prints the initial message" do
```

```
      initial_message =
      "Bem-vindo ao jogo da forca!"
```

```
    expect(output).to receive(
      :puts
    ).with(initial_message)

    game.start

  end

end

describe
  "#ended?" do

    it
      "returns false when the game just started" do

        expect(game).not_to be_ended
      end
    end
  end
end
```


end

end end

Depois dessa refatoração, você pode verificar que os testes de RSpec continuam passando. Então, podemos rodar os testes do Cucumber novamente:

```
$ bundle exec cucumber
```

Então o jogo termina com a seguinte mensagem na tela:

(...)

expected `forca` not to have output on stdout

but was:

```
bin/forca:11:in `<main>': undefined method `next_step'
```

```
for #<Game:0x00007f8a22 @output=#<IO:<STDOUT>>, @ended=false>
```

```
(NoMethodError)
```

(...)

2 scenarios (2 failed)

Podemos ver pela mensagem de erro que o teste agora está falhando por outro motivo, falta implementarmos o método `Game#next_step`. Vamos escrever um teste para esse método e implementá-lo.

O primeiro teste que vamos escrever é para o cenário onde o jogo acabou de começar. Nesse contexto, o método `Game#next_step` deve pedir para o jogador o tamanho da palavra a ser sorteada. Para fazer a verificação desse teste, basta checarmos se o objeto `game` vai pedir para o seu colaborador `output` para imprimir a pergunta ao jogador pedindo o tamanho da palavra a ser sorteada.

Seguindo essa ideia, escreva esse teste no arquivo `spec/game_spec.rb`:

```
describe "#next_step" do
```

```
  context
```

```
    "when the game just started" do
```

```
      it
```

```
        "asks the player for the length "
```

```
        \
```

```
"of the word to be raffled" do
```

```
  question =
```

```
  "Qual o tamanho da palavra a ser sorteada?"
```

```
  expect(output).to receive(
```

```
    :puts
```

```
  ).with(question)
```

```
  game.next_step
```

```
end
```

```
end end
```

Vamos fazê-lo passar. Implemente o método `next_step` no arquivo `lib/game.rb`, que imprimirá a frase pedindo o tamanho da palavra:

```
def
```

```
next_step
```

```
@output.puts("Qual o tamanho da palavra a ser sorteada?") end
```

Ao rodar o RSpec, podemos ver que os testes estão passando. Podemos voltar para o Cucumber.

11.6 Refatorando o código para poder implementar o segundo cenário

Ao rodarmos o Cucumber, vemos o seguinte:

```
$ bundle exec cucumber
```

```
(...)
```

```
expected "forca" to be successfully executed
```

```
(RSpec::Expectations::ExpectationNotMetError)
```

```
2 scenarios (2 failed)
```

```
5 steps (2 failed, 3 passed)
```

```
0m42.117s
```

Mesmo com os métodos `Game#ended?` e `Game#next_step` implementados, os dois cenários do Cucumber continuam quebrando. O erro aconteceu pois o processo do nosso jogo nunca termina. Para entendê-lo melhor, rode o nosso jogo:

\$ bin/forca

Qual o tamanho da palavra a ser sorteada?

Qual o tamanho da palavra a ser sorteada?

(...)

Qual o tamanho da palavra a ser sorteada?

Qual o tamanho da palavra a ser sorteada?

Qual o tamanho da palavra a ser sorteada?

O problema que está ocorrendo é que nosso jogo está em um loop infinito, pois em nenhum momento o método `Game#ended?` está retornando `true`. Por conta disso, depois de alguns segundos o Aruba encerra o processo, o que explica por que o `forca` não foi executado com sucesso.

Vamos melhorar a mensagem de erro que o teste nos dá para que seja mais fácil identificar esse problema no futuro. Primeiro, vamos configurar o tempo que o Aruba espera antes de encerrar o processo, pois o tempo padrão (quinze segundos) é muito longo. Três segundos são o suficiente para nosso cenário. Modifique o `features/support/game_helpers.rb`:

`module GameHelpers`

```
def
  start_new_game
    run_command(
      "bin/forca", exit_timeout: 3
    )
  end end
```

World(GameHelpers)

Agora vamos adicionar uma verificação para ter certeza de que o processo não foi encerrado pelo Aruba. Modifique o step definition o jogo termina com a seguinte mensagem na tela no arquivo `features/step_definitions/game_steps.rb`:

Então "o jogo termina com a seguinte mensagem na tela:" do

|text|

```
  expect(last_command_stopped).to have_finished_in_time
```

```
# ... end
```

Ao rodar o Cucumber, podemos ver que a mensagem de erro é bem mais informativa:

```
$ bundle exec cucumber
```

```
(...)
```

```
expected "forca" to have finished in time
```

```
(RSpec::Expectations::ExpectationNotMetError)
```

Agora precisamos que o jogador tenha um modo de finalizar o jogo no meio. O modo como ele vai poder fazer isso é digitando "fim" em qualquer momento em que o jogo pedir alguma entrada.

Para implementar essa ideia, vamos começar adicionando um novo step na nossa feature, o E termino o jogo, que vai simular o jogador terminando o jogo no meio.

Podemos adicioná-lo primeiro no cenário Começo de novo jogo com sucesso:

Cenário: Começo de novo jogo com sucesso

Ao começar o jogo, é mostrada a mensagem inicial para o

jogador.

Quando começo um novo jogo

E termino o jogo

Então o jogo termina com a seguinte mensagem na tela:

""""

Bem-vindo ao jogo da forca!

""""

E em seguida, no cenário que fará com sucesso o sorteio da palavra:

@wip

Cenário: Sorteio da palavra com sucesso

Após o jogador começar o jogo, ele deve escolher o tamanho da

palavra a ser adivinhada. Ao escolher o tamanho, o jogo

sorteia a palavra e mostra na tela um

" _ "

para cada letra que

a palavra sorteada tem.

Dado que comecei um jogo

Quando escolho que uma palavra com

"4"

letras deve ser sorteada

E termino o jogo

Então o jogo termina com a seguinte mensagem na tela:

```
""
```

```
 _ _ _ _
```

```
""
```

Agora, estamos prontos para implementar esse step. Adicione o seguinte step definition no arquivo `features/step_definitions/game_steps.rb`:

Quando "termino o jogo" do

```
type(  
  "fim") end
```

Com esse step definido, ao rodar o Cucumber, vemos que o processo do jogo

continua rodando sem parar. Isso ainda está acontecendo porque o método `Game#next_step` ainda não está pedindo uma entrada do jogador na qual ele possa digitar "fim". Vamos especificar esse novo comportamento através de um teste.

Até então, nós já temos um teste para o método `Game#next_step`:

```
describe "#next_step" do
```

```
  context
```

```
    "when the game just started" do
```

```
      it
```

```
        "asks the player for the length "
```

```
        \
```

```
        "of the word to be raffled" do
```

```
          question =
```

```
            "Qual o tamanho da palavra a ser sorteada?"
```

```
          expect(output).to receive(
```

```
:puts
```

```
).with(question)
```

```
game.next_step
```

```
end
```

```
end end
```

Esse teste verifica se o jogo pergunta para o jogador o tamanho da palavra a ser sorteada, quando o método `Game#next_step` é executado e o jogo acabou de começar. Precisamos modificá-lo para fazer mais do que isso, ele precisa verificar também se o jogo está lendo o tamanho da palavra que o jogador vai digitar. Para ler a entrada do jogador, podemos criar um novo colaborador que terá essa responsabilidade; vamos chamá-lo de `input`. Mudaremos, portanto, o teste acima para verificar se o objeto `game` pede para o objeto `input` ler a entrada do jogador:

```
RSpec.describe Game do
```

```
  let(  
    :output) { double("output"  
  ) }
```

```
let(  
:input) { double("input"  
) }
```

```
subject(  
:game) { Game  
.new(output, input) }
```

```
# (...)
```

```
describe  
"#next_step" do
```

```
context  
"when the game just started" do
```

```
it  
"asks the player for the length "  
\
```

```
"of the word to be raffled" do
```

```
  question =
```

```
    "Qual o tamanho da palavra a ser sorteada?"
```

```
    expect(output).to receive(
```

```
      :puts
```

```
    ).with(question)
```

```
    expect(input).to receive(
```

```
      :gets
```

```
    )
```

```
    game.next_step
```

```
  end
```

```
end
```

```
end end
```

Para fazer esse teste passar, podemos injetar a dependência input, cuja implementação default vai ser o STDIN, e utilizá-la no método next_step para ler a entrada do jogador. Essa implementação no arquivo lib/game.rb ficará assim:

```
class Game
```

```
  def initialize(output = STDOUT, input = STDIN  
  )
```

```
    @output
```

```
    = output
```

```
    @input
```

```
    = input
```

```
    @ended = false
```

```
end
```

```
# (...)
```

```
def
```

```
  next_step
```

```
    @output.puts("Qual o tamanho da palavra a ser sorteada?"
```

```
  )
```

```
    word_length =
```

```
    @input
```

```
    .gets
```

```
end end
```

Ao rodar os testes de RSpec, vemos que agora eles passam. Aproveitando que os

testes estão passando, vamos refatorar.

11.7 Extraíndo uma classe através de refatoração

Um problema da implementação atual é que agora nossa classe Game tem mais uma dependência. Essa nova dependência é mais um motivo para nosso código quebrar e um nível a mais de complexidade. Agora para instanciarmos um objeto da classe Game, precisamos antes instanciar dois objetos:

```
let(:output) { double("output"  
  
  ) }  
  
let(  
  
  :input) { double("input"  
  
  ) }  
  
  
subject(  
  
  :game) { Game.new(output, input) }
```

Os papéis desses objetos são muito relacionados: um tem como responsabilidade mostrar coisas para o usuário, o outro tem como responsabilidade pegar entradas de dados do usuário.

Como os papéis estão bem relacionados, podemos juntar esses dois objetos em

um só, que vai ter a responsabilidade de poder interagir com o usuário. Um objeto que fará a interface com nosso usuário. Vamos refatorar nosso teste e código para trocar a utilização dos objetos input e output por um único objeto que chamaremos de ui (user interface).

A primeira coisa que temos que mudar no teste da classe Game é trocar a criação de input e output por ui. Até então, essa parte do teste está assim:

```
RSpec.describe Game do
```

```
  let(
    :output) { double("output"
  ) }
```

```
  let(
    :input) { double("input"
  ) }
```

```
  subject(
    :game) { Game
    .new(output, input) }
```

```
# (...) end
```

Vamos modificá-la para ficar como o seguinte:

```
RSpec.describe Game do
```

```
  let(
    :ui) { double("ui"
  ) }
```

```
  subject(
    :game) { Game
    .new(ui) }
```

```
# (...) end
```

A segunda mudança que temos que fazer nos testes da classe Game é mudar chamadas de método do input e do output para o novo ui. Nesses testes, temos verificação de chamada de método desses objetos nos seguintes trechos:

it "prints the initial message" do

initial_message =

"Bem-vindo ao jogo da forca!"

expect(output).to receive(

:puts

).with(initial_message)

game.start

end

e

it "asks the player for the length of the word to be raffled" do

question =

"Qual o tamanho da palavra a ser sorteada?"

expect(output).to receive(

:puts

```
).with(question)
```

```
    expect(input).to receive(  
      :gets  
    )
```

```
    game.next_step  
  end
```

Ou seja, onde antes estava "expect(output).to receive(:puts)" e "expect(input).to receive(:gets)", precisamos mudar para "expect(ui).to receive(:puts)" e "expect(ui).to receive(:gets)". E, já que estamos mudando de input e output para ui, vale a pena repensar os nomes de métodos puts e gets para o novo ui.

Os nomes de método puts e gets estão muito relacionados com o STDOUT e STDIN, que eram as implementações antigas das dependências do game. A partir de agora, estamos criando uma nova abstração como dependência, o ui. Essa nova dependência não necessariamente sempre vai ter como implementação real o STDIN e STDOUT por trás. Logo, faz sentido criarmos nomes de métodos mais abstratos, que estejam mais relacionados com o papel que a dependência vai ter para o game do que com a implementação real que vai ser feita. Vamos mudar então de gets e puts para read e write.

Seguindo essa ideia, o que antes era "expect(input).to receive(:gets)" vai ficar "expect(ui).to receive(:read)", e o que antes era "expect(output).to receive(:puts)" vai ficar "expect(ui).to receive(:write)".

Logo, os trechos de código anteriores ficarão assim:

```
it "prints the initial message" do
```

```
  initial_message =
```

```
    "Bem-vindo ao jogo da forca!"
```

```
    expect(ui).to receive(
```

```
      :write
```

```
    ).with(initial_message)
```

```
    game.start
```

```
  end
```

```
e
```

```
it "asks the player for the length of the word to be raffled" do
```

```
  question =
```

"Qual o tamanho da palavra a ser sorteada?"

```
expect(ui).to receive(  
  :write  
) .with(question)
```

```
expect(ui).to receive(  
  :read  
)
```

```
game.next_step  
end
```

Por fim, nosso teste inteiro ficará desse modo:

```
RSpec.describe Game do
```

```
  let(  
    :ui) { double("ui"  
  ) }
```



```
subject(  
:game) { Game  
.new(ui) }
```

```
describe  
"#start" do
```

```
  it  
  "prints the initial message" do
```

```
    initial_message =  
    "Bem-vindo ao jogo da forca!"
```

```
    expect(ui).to receive(  
:write  
) .with(initial_message)
```

```
    game.start
```

```
end
```

end

describe

"#ended?" do

it

"returns false when the game just started" do

expect(game).not_to be_ended

end

end

describe

"#next_step" do

context

"when the game just started" do

it

"asks the player for the length "

\

"of the word to be raffled" do

question =

"Qual o tamanho da palavra a ser sorteada?"

expect(ui).to receive(

:write

).with(question)

expect(ui).to receive(

:read

)

```
game.next_step
```

```
end
```

```
end
```

```
end end
```

Para fazer o teste passar, precisamos alterar a classe Game para deixar de utilizar `input.gets` e `output.puts` para utilizar `ui.read` e `ui.write`. Precisamos também mudar a injeção de dependência para a nova dependência `ui`. Aplicando essas mudanças no arquivo `lib/game.rb`, o código ficará assim:

```
class Game
```

```
def initialize(ui = CliUi
```

```
.new)
```

```
@ui
```

```
= ui
```

```
@ended = false
```

```
end
```

```
def
```

```
start
```

```
  initial_message =
```

```
  "Bem-vindo ao jogo da forca!"
```

```
@ui
```

```
.write(initial_message)
```

```
end
```

```
def
```

```
ended?
```

```
@ended
```

```
end
```

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
word_length =
```

```
@ui
```

```
.read
```

```
end end
```

Com essa implementação, nossos testes passam.

Note que a implementação default da dependência ui é um objeto de uma nova classe, a classe CliUi (command line user interface).

Crie um novo arquivo lib/cli_ui.rb e escreva o código dessa classe nele:

```
# Classe responsável por representar a interface com o usuário. # class CliUi
```

```
  def
```

```
    write(text)
```

```
      puts text
```

```
  end
```

```
  def
```

```
    read
```

```
user_input = gets
```

```
user_input
```

```
end end
```

Para que a classe Game enxergue a classe CliUi, adicione o seguinte require no começo do arquivo lib/game.rb:

```
require_relative "cli_ui"
```

```
class Game
```

```
# (...) end
```

Ao rodar os testes do RSpec, vemos que eles estão passando. Com a refatoração terminada e com os testes no verde, podemos ir para o próximo passo.

11.8 Possibilitando ao jogador terminar o jogo no meio

Com os testes passando, podemos voltar para a ideia original, que era a possibilidade de o jogador terminar o jogo no meio digitando "fim". Para isso, adicione o seguinte teste no arquivo spec/game_spec.rb:

```
describe "#next_step" do
```

```
  # (...)
```

```
    it
```

```
      "finishes the game when the player asks to" do
```

```
        player_input =
```

```
          "fim"
```

```
allow(ui).to receive(  
:read  
)and_return(player_input)  
  
game.next_step  
  
expect(game).to be_ended  
  
end end
```

Ao rodarmos os testes, eles falham com a seguinte mensagem:

Failures:

1) Game#next_step finishes the game when the player asks to

Failure/Error:

```
@ui.write("Qual o tamanho da palavra a ser sorteada?")  
#<Double "ui"> received unexpected message :write  
with ("Qual o tamanho da palavra a ser sorteada?")  
# ./lib/game.rb:19:in `next_step'
```

O problema foi que o double ui recebeu uma mensagem write com o argumento Qual o tamanho da palavra a ser sorteada? de modo inesperado. O método next_step de fato chama ui.write, mas a única coisa que fizemos no setup do nosso teste com o double ui foi allow(ui).to receive(:read).and_return(player_input). Fizemos desse modo, pois essa é a única parte da interação com o ui que nos importa nesse teste. Vamos ignorar as outras interações.

Para ignorar as mensagens enviadas para ui, podemos usar a feature de as_null_object do RSpec. Edite a definição da dependência ui no arquivo spec/game_spec.rb para ficar da seguinte forma:

RSpec.describe Game do

```
  let(
    :ui) { double("ui"
  ).as_null_object }
```

```
# (...) end
```

Agora, ao rodar os testes, recebemos a mensagem de erro esperada:

```
$ bundle exec rspec
```

1) Game#next_step finishes the game when the player asks to

Failure/Error: expect(game).to be_ended

expected `#<Game:0x00007ff54029fee0>.ended?`

to return true, got false

./spec/game_spec.rb:41:in `block (3 levels)'

Para fazer o teste passar, precisamos modificar o método next_step para que ele sete a flag @ended como true, caso o jogador tenha digitado a entrada "fim". Com essa modificação, o método Game#next_step ficará assim:

```
def
```

```
  next_step
```

```
    @ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
  )
```

```
    player_input =
```

```
    @ui
```

```
      .read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

```
end end
```

Ao rodar os testes do RSpec agora, vemos que eles estão passando. E melhor ainda, se rodarmos os testes do Cucumber, vemos que o primeiro cenário voltou a passar!

```
$ bundle exec cucumber
```

```
(...)
```

```
2 scenarios (1 failed, 1 passed)
```

Agora finalmente podemos voltar a trabalhar no cenário que definimos no começo deste capítulo, o cenário Sorteio da palavra com sucesso. Mas faremos isso só no próximo capítulo.

Pontos-chaves deste capítulo

Neste capítulo, nós continuamos a especificar a primeira funcionalidade do programa, Começar Jogo. Ao definir um novo cenário para essa funcionalidade, usamos o support code do Cucumber para evitar duplicação na camada de step definitions.

Outro ponto importante é que tivemos que adaptar nossos testes com Cucumber para que eles refletissem a realidade do nosso programa, que estava quebrando na "vida real", mas passando nos testes. Ter o feedback correto dos testes é muito importante, pois caso isso não seja feito, as pessoas podem começar a perder a confiança na suíte de testes.

Nos testes com RSpec, vimos como usar o `subject` e o `let` para evitar duplicação de código nos testes.

Por fim, refatoramos a classe `Game`, extraíndo uma nova dependência, a classe `CliUi`. Desse modo ficou mais fácil de fazer nossos testes. Além disso, ao retirarmos essa responsabilidade da classe `Game`, a deixamos menos acoplada com o input e output com usuário. Essa refatoração resultou em um design com menor acoplamento.

Essa melhoria no design pode nos ajudar com requisitos futuros, por exemplo, se a interface com usuário mudasse de linha de comando para Twitter, agora podemos fazer isso simplesmente trocando a classe `CliUi` por uma classe `TwitterUi`.

Com tudo isso feito, estamos prontos para, no capítulo seguinte, finalizar a especificação e implementação da primeira funcionalidade.

Capítulo 12

Finalizando a primeira funcionalidade

No capítulo anterior, nós definimos o segundo cenário da primeira funcionalidade. No entanto, não pudemos implementá-lo, pois foi necessário fazer uma série de modificações e refatorações para que pudéssemos começar a trabalhar nele.

Neste capítulo, vamos implementar esse e o terceiro cenário, finalizando essa funcionalidade. Ao final do capítulo, o jogador já vai poder iniciar um jogo e sortear uma palavra para adivinhar.

Sigam-me os bons!

12.1 Deixando o segundo cenário no verde

Antes de irmos para o próximo passo, vamos ver como estão nossos testes. Ao rodarmos o RSpec, podemos ver que está tudo verde. Porém, ao rodar o Cucumber, vemos o seguinte:

```
$ bundle exec cucumber
```

```
Dado que comecei um jogo
```

```
Quando escolho que uma palavra com "4" letras deve ser sorteada
```

```
E termino o jogo
```

```
Então o jogo termina com a seguinte mensagem na tela:
```

```
""""
```

```
----
```

```
""""
```

```
expected "Bem-vindo ao jogo da forca!\n
```

```
Qual o tamanho da palavra a ser sorteada?\n
```

```
Qual o tamanho da palavra a ser sorteada?\n"
```

```
to include "_ _ _ _"
```

2 scenarios (1 failed, 1 passed)

O primeiro cenário está passando e o segundo está quebrando. Por quê? Ele quebrou, pois falta imprimir na tela um "_" para cada letra da palavra sorteada. Vamos fazer isso!

Para fazer esse cenário passar é necessário que o nosso jogo sorteie uma palavra e imprima na tela um _ para cada letra da palavra sorteada. Isso deve acontecer como o passo após o jogador dizer qual o tamanho da palavra a ser sorteada. Logo, esse comportamento fará parte do método `Game#next_step`.

Podemos estruturar os testes para esse comportamento do seguinte modo no arquivo `spec/game_spec.rb`:

```
describe "#next_step" do
```

```
# (...)
```

```
  context
```

```
    "when the player asks to raffle a word" do
```

it

"raffles a word with the given length"

it

"prints a '_' for each letter in the raffled word"

end

(...) end

Para implementar o primeiro teste (it "raffles a word with the given length") podemos checar se, dado que o jogador insere que o tamanho da palavra deve ser "3", então a palavra sorteada deverá ter 3 letras:

describe "#next_step" do

```
# (...)
```

```
context
```

```
"when the player asks to raffle a word" do
```

```
it
```

```
"raffles a word with the given length" do
```

```
word_length =
```

```
3
```

```
allow(ui).to receive(
```

```
:read
```

```
).and_return(word_length.to_s)
```

```
game.next_step
```

```
expect(game.raffled_word.length).to eq(word_length)
```

end

it

"prints a '_' for each letter in the raffled word"

end

(...) end

Ao rodar o RSpec, o teste quebra e vemos o seguinte:

```
$ bundle exec rspec
```

Failures:

- 1) Game#next_step when the player asks to raffle a word
raffles a word with the given length

Failure/Error:

```
expect(game.raffled_word.length).to eq(word_length)
```

NoMethodError:

```
undefined method `raffled_word' for #<Game:0x00007f9a2b96>
```

Ou seja, falta implementarmos o método que retornará a palavra sorteada, o método `Game#raffled_word`. Faça isso adicionando o seguinte no começo do arquivo `lib/game.rb`:

```
class Game
```

```
  attr_accessor :raffled_word end
```

Ao rodarmos o `RSpec` novamente, ele quebra com a mensagem de erro adiante:

```
$ bundle exec rspec
```

Failures:

1) Game#next_step when the player asks to raffle a word
raffles a word with the given length

Failure/Error:

```
expect(game.raffled_word.length).to eq(word_length)
```

NoMethodError:

```
undefined method `length' for nil:NilClass
```

Não foi possível verificar quantas letras a palavra sorteada tem, pois não existe uma palavra sorteada ainda. Para o teste passar, precisamos que o método next_step sorteie uma palavra com o tamanho dado pelo jogador. Modifique o método Game#next_step para que ele faça isso:

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
player_input =
```

```
@ui
```

```
.read.strip
```



```
if player_input == "fim"
```

```
  @ended = true
```

```
else
```

```
  word_length = player_input.to_i
```

```
  words =
```

```
    %w(hi mom game fruit)
```

```
  @raffled_word
```

```
=
```

```
  words.detect { |word| word.length == word_length }
```

```
end end
```

Implementamos a seguinte condição nesse código: se o jogador não digitou

"fim", então ele digitou o tamanho da palavra a ser sorteada. Após isso, usamos esse input para sortear uma palavra e salvá-la em `@raffled_word`.

Ao rodar os testes, vemos que agora eles passam! Mas antes de seguir em frente, vamos refatorar nosso código para ficar mais clara a sua intenção.

Para entender o que o trecho de código a seguir faz:

```
word_length = player_input.to_i  
  
words =  
  
%w(hi mom game fruit) @raffled_word  
  
=  
  
words.detect { |word| word.length == word_length }
```

Precisamos parar um pouco, pensar e só então é possível entender que o que ele está fazendo é sortear a palavra. A intenção desse código (o que ele faz, e não como ele faz) não está clara.

A intenção do nosso código deve ser clara, caso contrário, sua leitura pode ser prejudicada. Para deixá-la mais explícita, vamos refatorar o código, extraindo esse trecho para um método privado chamado `raffle_word`:

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

```
else
```

```
raffle_word(player_input.to_i)
```

```
end end
```

```
private
```

```
def
```

```
  raffle_word(word_length)
```

```
    words =
```

```
    %w(hi mom game fruit)
```

```
  @raffled_word
```

```
  =
```

```
    words.detect { |word| word.length == word_length }
```

```
end
```

Ao rodar o RSpec, vemos que ele continua no verde, ou seja, tivemos sucesso na refatoração, pois nenhum teste quebrou. Vamos para o próximo spec pendente, que é o it "prints a '_' for each letter in the raffled word".

Esse teste vai verificar se o jogo está imprimindo um caractere _ para cada letra da palavra sorteada. Implemente esse teste do seguinte modo:

```
it "prints a '_' for each letter in the raffled word" do
```

```
word_length =  
"3"  
  
allow(ui).to receive(  
:read  
)and_return(word_length)  
  
expect(ui).to receive(  
:write).with(" _ _ _"  
)  
  
game.next_step  
end
```

Ao rodar o RSpec, ele falhará com a seguinte mensagem:

```
$ bundle exec rspec
```

Failures:

1) Game#next_step when the player asks to raffle a word

prints a '_' for each letter in the raffled word

Failure/Error: expect(ui).to receive(:write).with("__ _")

#<Double "ui"> received :write with unexpected arguments

expected: ("__ _")

got: ("Qual o tamanho da palavra a ser sorteada?")

A mensagem nos diz que o objeto ui deveria ter recebido a mensagem write com o argumento "__ _", mas não recebeu. Vamos escrever um método chamado print_letters_feedback, que vai ser responsável por imprimir o feedback das letras adivinhadas, que será inicialmente um _ para cada letra da palavra sorteada. Esse método deve ser chamado logo após a palavra ser sorteada. Adicionando essa chamada, nosso código ficará assim:

def

next_step

@ui.write("Qual o tamanho da palavra a ser sorteada?"

)

player_input =

@ui

```
.read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

```
else
```

```
    raffle_word(player_input.to_i)
```

```
    print_letters_feedback
```

```
end end
```

O método `print_letters_feedback` será privado e a sua implementação ficará dessa forma:

```
def
```

```
    next_step
```

```
# (...) end
```

```
private
```

```
# (...)
```

```
def
```

```
  print_letters_feedback
```

```
    letters_feedback =
```

```
    """
```

```
    @raffled_word.length.times do
```

```
      letters_feedback <<
```

```
      "_ "
```


end

```
letters_feedback.strip!
```

```
@ui.write(letters_feedback) end
```

Ao rodar a suíte, vemos que o teste it "prints a '_' for each letter in the raffled word" passa, mas outro quebrou:

```
$ bundle exec rspec
```

Failures:

1) Game#next_step when the game just started asks the player
for the length of the word to be raffled

Failure/Error:

```
@raffled_word.length.times do
```

```
  letters_feedback << "_ "
```

```
end
```

NoMethodError:

undefined method `length' for nil:NilClass

./lib/game.rb:43:in `print_letters_feedback'

./lib/game.rb:28:in `next_step'

./spec/game_spec.rb:31:in `block (4 levels)'

Como podemos ver na mensagem de erro do teste, o problema ocorreu na chamada do método length, dentro do método print_letters_feedback:

def

print_letters_feedback

letters_feedback =

""

@raffled_word.length.times do

letters_feedback <<

" _ "

end

letters_feedback.strip!

@ui.write(letters_feedback) end

Esse teste quebrou porque, nele, a variável de instância `@raffled_word` está nil. Para entender por que isso aconteceu, vamos dar uma olhada no teste que quebrou:

describe "#next_step" do

context

"when the game just started" do

it

"asks the player for the length "

\

"of the word to be raffled" do

question =

"Qual o tamanho da palavra a ser sorteada?"

expect(ui).to receive(

:write

).with(question)

expect(ui).to receive(

:read

)

game.next_step

end

end

```
# (...) end
```

Lendo o teste, vemos que em nenhum momento é setado o que o método `ui.read` vai retornar. Mas, para que uma palavra seja sorteada e a variável de instância `@raffled_word` seja setada, é necessário que esse método retorne algo, que será o tamanho da palavra a ser sorteada. Esse é o problema que precisamos consertar.

Para que o `ui.read` retorne o tamanho da palavra, vamos mudar a linha que estava `expect(ui).to receive(:read)` para `expect(ui).to receive(:read).and_return(word_length)`:

```
describe "#next_step" do
```

```
  context
```

```
    "when the game just started" do
```

```
      it
```

```
        "asks the player for the length "
```

```
        \
```

```
        "of the word to be raffled" do
```

```
question =
```

```
"Qual o tamanho da palavra a ser sorteada?"
```

```
expect(ui).to receive(
```

```
:write
```

```
).with(question)
```

```
word_length =
```

```
"3"
```

```
expect(ui).to receive(
```

```
:read
```

```
).and_return(word_length)
```

```
game.next_step
```

```
end
```

```
end
```

```
# (...) end
```

Ao rodar o RSpec novamente, podemos ver que ele está no verde. E, ao rodar o Cucumber, podemos ver que ele também está no verde! Terminamos o segundo cenário da primeira funcionalidade do nosso jogo! Como o cenário está no verde, já podemos tirar a tag @wip dele.

Com tudo no verde, vamos para o próximo cenário.

12.2 Finalizando a primeira funcionalidade

A primeira funcionalidade do nosso jogo é a "Começar jogo", que consiste nos seguintes cenários:

Começo de novo jogo com sucesso;

Sorteio da palavra com sucesso;

Sorteio da palavra sem sucesso.

Os dois primeiros cenários já estão no verde. Vamos começar a especificar e implementar o último cenário.

Adicione o seguinte cenário no arquivo `features/comecar_jogo.feature`:

@wip

Cenário: Sorteio da palavra sem sucesso

Se o jogador pedir para o jogo sortear uma palavra com um tamanho que o jogo não tem disponível, o jogador deve ser avisado

disso e o jogo deve pedir para o jogador sortear outra palavra.

Dado que comecei um jogo

Quando escolho que uma palavra com

"9"

letras deve ser sorteada

E termino o jogo

Então o jogo termina com a seguinte mensagem na tela:

""

Não temos uma palavra com o tamanho desejado,

é necessário escolher outro tamanho.

Qual o tamanho da palavra a ser sorteada?

""

Ao rodar o Cucumber, vemos que ele quebra para esse novo cenário:

```
$ bundle exec cucumber -t @wip
```

(...)

1 scenario (1 failed)

Como de costume, quando temos um teste novo de Cucumber quebrando, o próximo passo é escrever um teste de RSpec e fazê-lo passar.

Vamos escrever um teste que vai verificar o cenário quando o jogador pede para ser sorteada uma palavra com um tamanho que o jogo não tem. Para implementá-lo, adicione o seguinte no arquivo spec/game_spec.rb:

```
describe "#next_step" do
```

```
# (...)
```

```
  context
```

```
    "when the player asks to raffle a word" do
```

```
# (...)
```

it

"tells if it's not possible to raffle "

\

"with the given length" do

word_length =

"9"

allow(ui).to receive(

:read

).and_return(word_length)

error_message =

"Não temos uma palavra com o tamanho desejado,\n"

\

"é necessário escolher outro tamanho."

```
        expect(ui).to receive(  
:write  
) .with(error_message)
```

```
        game.next_step
```

```
    end
```

```
end end
```

Ao rodar o RSpec, podemos confirmar que o novo teste está vermelho. Vamos fazê-lo passar. A implementação atual do `Game#next_step` está assim:

```
def
```

```
  next_step
```

```
    @ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
  )
```

```
    player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

```
else
```

```
    raffle_word(player_input.to_i)
```

```
    print_letters_feedback
```

```
end end
```

Precisamos alterá-la para que, se não for possível sortear uma palavra com o tamanho pedido, uma mensagem de erro seja impressa para o jogador. Para fazer isso, basta checar o retorno do método `raffle_word`. Se for `false`, a mensagem de erro deve ser impressa. Altere o método `Game#next_step` do seguinte modo para implementar essa ideia:

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

```
else
```

```
if
```

```
raffle_word(player_input.to_i)
```

```
  print_letters_feedback
```

```
else
```

```
  error_message =
```

```
    "Não temos uma palavra com o tamanho desejado,\n"
```

```
  \
```

```
    "é necessário escolher outro tamanho."
```

```
@ui
```

```
  .write(error_message)
```

```
end
```

end end

Repare que foi necessário fazer um if ... else dentro do else já existente para fazer o teste passar. Condicionais aninhadas são um mau sinal, mostra que o código está complexo. Precisamos refatorar isso, mas por enquanto vamos deixar assim.

Com a implementação anterior, ao rodarmos o RSpec, vemos que ele passa com tudo no verde. E, ao rodar o Cucumber, vemos que ele também está 100% verde!

```
$ bundle exec cucumber
```

```
(...)
```

```
3 scenarios (3 passed)
```

Finalmente a funcionalidade de começar o jogo está totalmente pronta! Mas, antes de continuar a comemoração, não se esqueça de tirar a tag @wip do cenário que acabamos de finalizar.

Com todos os testes no verde, estamos prontos para refatorar. Faremos isso no próximo capítulo.

Enquanto você descansa um pouco e se prepara, aproveite também para brincar

com o que temos pronto até agora. Basta rodar o binário do jogo bin/forca e explorar a primeira funcionalidade.

Pontos-chaves deste capítulo

Neste capítulo nós conseguimos finalizar a especificação e implementação da primeira funcionalidade.

Durante a implementação, vimos como refatorar nosso código para deixar mais clara a sua intenção, através da extração de métodos privados.

No final da implementação, foi necessário fazer mais condicionais aninhadas do que gostamos, deixando nosso código mais complexo. No capítulo seguinte, vamos refatorar nosso código para eliminar essa e outras dívidas técnicas que possam existir.

Capítulo 13

Refatorando nosso código

No capítulo anterior, nós terminamos a especificação e implementação da primeira funcionalidade. No entanto, algumas partes do código que desenvolvemos até agora podem ser melhoradas.

Neste capítulo, vamos identificar que partes podem ser melhoradas, refatorar nosso código e melhorar a arquitetura do nosso projeto. Tudo isso com suporte da nossa suíte de testes, que garantirá que não quebraremos nenhuma funcionalidade enquanto aprimoramos a qualidade de nosso código.

13.1 Identificando os pontos a serem refatorados

No final da implementação do cenário anterior, notamos que uma parte do nosso código não ficou tão boa quanto queremos — essa parte está dentro do método `Game#next_step`:

```
def
```

```
  next_step
```

```
    @ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
    )
```

```
    player_input =
```

```
      @ui
```

```
        .read.strip
```

```
    if player_input == "fim"
```

```
@ended = true
```

```
else
```

```
if
```

```
raffle_word(player_input.to_i)
```

```
  print_letters_feedback
```

```
else
```

```
  error_message =
```

```
    "Não temos uma palavra com o tamanho desejado,\n"
```

```
  \
```

```
    "é necessário escolher outro tamanho."
```

```
@ui
```

```
.write(error_message)
```

```
end
```

```
end end
```

Esse método está com muitas condicionais aninhadas, o que dificulta seu entendimento e aumenta sua complexidade. Esse é um possível ponto a ser refatorado.

Outro ponto que vale a pena ser avaliado é o conjunto de métodos privados da classe Game:

```
class Game
```

```
# (...)
```

```
private
```

```
def
    raffle_word(word_length)

        words =

            %w(hi mom game fruit)

    @raffled_word

    =

        words.detect { |word| word.length == word_length }

end
```

```
def
    print_letters_feedback

        letters_feedback =

        ""
```

```
@raffled_word.length.times do
```

```
  letters_feedback <<
```

```
  "_ "
```

```
end
```

```
  letters_feedback.strip!
```

```
@ui
```

```
.write(letters_feedback)
```

```
end end
```

Métodos privados em excesso podem ser um indício de que a classe está com mais de uma responsabilidade e que uma delas está escondida de sua API pública através de métodos privados. Talvez esses métodos privados sejam mais um ponto possível de refatoração.

Por fim, para termos mais feedback sobre a qualidade de nosso código, podemos olhar também para os testes. Testes podem dar um ótimo feedback, pois neles podemos ver um exemplo de como nosso código é utilizado. Logo, no teste fica bem claro se é simples ou não de utilizar o código que fizemos.

Vamos dar uma olhada nos testes da classe Game:

Game

#start

prints the initial message

#ended?

returns false when the game just started

#next_step

finishes the game when the player asks to

when the game just started

asks the player for the length of the word to be raffled

when the player asks to raffle a word

raffles a word with the given length

prints a '_' for each letter in the raffled word

tells if it's not possible to raffle with the given length

A primeira coisa que podemos notar nos testes da classe Game é que a maioria deles é relacionada ao comportamento do método next_step. Esse método implementa o seguinte comportamento:

- mostrar uma informação para o jogador;
- pedir um input do jogador;
- fazer algum processamento interno do jogo;
- depois mostrar outra informação para o jogador;
- e assim consecutivamente.

Esse comportamento é basicamente o fluxo do jogo.

Tirando os testes do next_step, só sobram dois, um do start e um do ended?. O do start testa se é impressa a mensagem inicial para o jogador, ou seja, também está relacionado à interação com o usuário e ao fluxo do jogo. O do ended? verifica se o jogo acabou ou não, isto é, é relacionado com o estado do objeto game.

A maioria dos testes da classe Game é sobre o fluxo do jogo e interação com usuário, só um é sobre o estado do objeto game. Então, um objeto da classe Game cuida do fluxo do jogo, da interação com o usuário e também de guardar o estado do jogo. Isso está estranho, pode ser um indício de falta de coesão e de que a classe está implementando mais de uma responsabilidade.

Vamos começar a atacar cada um dos pontos levantados e refatorar onde for necessário.

13.2 Extraíndo uma classe de um método privado

O primeiro ponto que vamos analisar serão os métodos privados, mais especificamente, o método `Game#raffle_word`.

Ao ler o código da classe `Game`, podemos notar que ela está fazendo pelo menos 3 coisas:

Interage com o usuário e executa as ações necessárias baseado no input do usuário. Toda a parte do `next_step`, por exemplo;

Conhece o estado do jogo: método `ended?` e variável de instância `@ended`;

Sorteia uma palavra: método `raffle_word`.

Se cada uma dessas partes tiver um novo requisito, a classe vai ter que ser modificada, logo o princípio da responsabilidade única (Single Responsibility Principle — SRP) está sendo quebrado (MARTIN, 2005). Por exemplo, imagine se o processo de sortear uma palavra não seja mais a partir de um array hard-coded no código, mas sim sortear através da comunicação com um web service sorteador de palavras. Se isso acontecesse, seria necessário mudar a classe `Game`.

Outro motivo para refatorarmos o método `Game#raffle_word` é que, pelo fato de ele ser um método privado, não pudemos fazer testes de unidade para ele. Visto que ele implementa um comportamento importante, ele merece testes. Se você

sentir a necessidade de testar um método privado, pode ser um sinal de que vale a pena extraí-lo para sua própria classe. E é isso que vamos fazer.

Antes de extrair o método `Game#raffle_word` para sua própria classe, vamos modificar os testes da classe `Game` como se ela já existisse. Desse modo, podemos deixar os testes nos guiar nas modificações que serão necessárias.

A primeira coisa que temos que fazer é injetar uma nova dependência na classe `Game`. Essa nova dependência vai ter o papel de sortear palavras, então vamos chamá-la de `word raffler`. Vamos modificar o setup do teste `spec/game_spec.rb` para seguir essa ideia. Modifique o subject desse teste e adicione um novo let de modo a ficar assim:

`RSpec.describe Game do`

```
  let(
    :ui) { double("ui"
    ).as_null_object }

  let(
    :word_raffler) { double("word raffler"
    ).as_null_object }

  subject(
    :game) { Game
```

```
.new(ui, word_raffler) }
```

```
# (...) end
```

Feito isso, precisamos atualizar os testes que são impactados por essa mudança. Os testes impactados são os testes relativos ao comportamento do método `#next_step` no contexto de sorteio da palavra:

```
context "when the player asks to raffle a word" do
```

```
  it
```

```
    "raffles a word with the given length"
```

```
  it
```

```
    "prints a '_' for each letter in the raffled word"
```

```
  it
```

```
    "tells if it's not possible to raffle with the given length" end
```

Vamos atualizar um por um. O primeiro teste está atualmente assim:

```
it "raffles a word with the given length" do
```

```
  word_length =
```

```
    3
```

```
    allow(ui).to receive(
```

```
      :read
```

```
    ).and_return(word_length.to_s)
```

```
    game.next_step
```

```
    expect(game.raffled_word.length).to eq(word_length)
```

```
  end
```

Esse teste é sobre verificar se foi sorteada uma palavra com o tamanho pedido. Imaginando que a API do colaborador `word_raffler` vai ter um método chamado `raffle` que sorteia uma palavra com o tamanho pedido, para fazer a verificação desse teste, basta que chequemos se o objeto `game` enviou a mensagem `raffle` para o objeto `word_raffler`. Podemos fazer isso usando mock expectations do seguinte modo:

it "raffles a word with the given length" do

word_length =

3

allow(ui).to receive(

:read

).and_return(word_length.to_s)

expect(word_raffler).to receive(

:raffle

).with(word_length)

game.next_step

end

Seguindo a atualização dos testes, o próximo é o seguinte:

it "prints a '_' for each letter in the raffled word" do

```
word_length =  
"3"  
  
allow(ui).to receive(  
:read  
)and_return(word_length)  
  
expect(ui).to receive(  
:write).with("__ _"  
)  
  
game.next_step  
end
```

Esse teste verifica se foi impresso na tela um caractere "" para cada letra da palavra sorteada. Como ele está verificando a impressão de três caracteres "", precisamos que o word_raffler sorteie uma palavra com três letras. Podemos fazer isso fazendo um stub do método word_raffler.raffle. Faça isso nesse teste deste modo:

it "prints a '_' for each letter in the raffled word" do


```
word_length =  
"3"  
  
allow(ui).to receive(  
:read  
)and_return(word_length)  
allow(word_raffler).to receive(  
:raffle).and_return("mom"  
)  
  
expect(ui).to receive(  
:write).with("__ _"  
)  
  
game.next_step  
end
```

Nesse caso usamos stub e não mock, pois o teste não é sobre verificar que a mensagem raffle foi enviada para o objeto word_raffler. Usar stubs serve para setar o retorno de alguma dependência, já usar mocks serve para testar se a interação com uma dependência foi feita corretamente.

O último teste que falta atualizarmos é o seguinte:

```
it "tells if it's not possible to raffle "
```

```
\
```

```
"with the given length" do
```

```
  word_length =
```

```
  "9"
```

```
  allow(ui).to receive(
```

```
    :read
```

```
  ).and_return(word_length)
```

```
  error_message =
```

```
  "Não temos uma palavra com o tamanho desejado,\n"
```

```
\
```

```
  "é necessário escolher outro tamanho."
```

```
expect(ui).to receive(  
  :write  
) .with(error_message)
```

```
game.next_step  
end
```

Esse teste verifica o comportamento esperado de quando não é possível sortear uma palavra com o tamanho pedido. Vamos imaginar que quando o método `word_raffler.raffle` não consegue sortear uma palavra, ele retorna `nil`. Com essa definição, podemos atualizar nosso teste com um stub desse método do seguinte modo:

```
it "tells if it's not possible to raffle "
```

```
\
```

```
"with the given length" do
```

```
  word_length =
```

```
  "9"
```

```
allow(ui).to receive(  
  :read  
)and_return(word_length)  
allow(word_raffler).to receive(  
  :raffle).and_return(nil  
)
```

```
error_message =
```

```
"Não temos uma palavra com o tamanho desejado,\n"
```

```
\
```

```
"é necessário escolher outro tamanho."
```

```
expect(ui).to receive(  
  :write  
)with(error_message)
```

```
game.next_step
```

end

Perceba que ao fazer as suposições dos métodos do colaborador `word_raffler`, estamos especificando o protocolo de comunicação entre o objeto `game` e o objeto `word_raffler`. Isso é muito legal, pois estamos desenhando a API desse novo objeto baseada nas necessidades de quem vai utilizá-lo.

Feitas todas essas atualizações, vamos rodar o `RSpec` e ver o que precisamos mudar no nosso código:

```
$ bundle exec rspec
```

(...)

Failure/Error:

```
def initialize(ui = CliUi.new)

  @ui = ui

  @ended = false

end
```

ArgumentError:

```
wrong number of arguments (given 2, expected 0..1)
```

(...)

7 examples, 7 failures

Todos os testes quebraram. A mensagem de erro nos mostra que o problema é que o construtor da classe Game só aceita um argumento. Vamos modificá-lo para podermos injetar a nova dependência. Edite o método initialize da classe Game a fim de injetar a dependência word_raffler, ele deve ficar assim:

```
class Game
```

```
# (...)
```

```
def initialize(ui = CliUi.new, word_raffler = WordRaffler  
.new)
```

```
@ui
```

```
= ui
```

```
@word_raffler
```

```
= word_raffler
```

```
@ended = false
```

```
end
```

```
# (...) end
```

Ao rodar os testes novamente, vemos o seguinte:

```
$ bundle exec rspec
```

Failures:

- 1) Game#next_step when the player asks to raffle a word
raffles a word with the given length

Failure/Error:

```
expect(word_raffler).to receive(:raffle).with(word_length)
```

```
(Double "word raffer").raffle(3)
```

```
expected: 1 time with arguments: (3)
```

```
received: 0 times
```

7 examples, 1 failure

Agora somente um teste quebrou. O problema foi que o teste está verificando se a mensagem raffle foi enviada para o objeto word_raffler, mas ela não foi enviada. Vamos modificar a classe Game para que ela use a nova dependência e envie essa mensagem.

Até então o método Game#next_step está sorteando a palavra do seguinte modo:

```
def
```

```
next_step
```

```
# (...)
```



```
if # (...)
```

```
else
```

```
if raffle_word(player_input.to_i) # sorteio da palavra
```

```
else
```

```
end
```

```
end end
```

Modifique esse método para que ele sorteie a palavra utilizando o objeto salvo em `@word_raffler`:

```
def
```

```
  next_step
```

```
  # (...)
```

```
  if # (...)
```

```
  else
```

```
    if @raffled_word = @word_raffler
```

```
      .raffle(player_input.to_i)
```

```
    else
```

```
  end
```

```
end end
```

Ao fazer essa modificação, esse método deve ficar assim:

```
def
```

```
  next_step
```

```
    @ui.write("Qual o tamanho da palavra a ser sorteada?"  
  )
```

```
    player_input =
```

```
    @ui
```

```
    .read.strip
```

```
    if player_input == "fim"
```

```
      @ended = true
```

else

if @raffled_word = @word_raffler

.raffle(player_input.to_i)

print_letters_feedback

else

error_message =

"Não temos uma palavra com o tamanho desejado,\n"

\

"é necessário escolher outro tamanho."

@ui

```
.write(error_message)
```

```
end
```

```
end end
```

Agora que extraímos a responsabilidade do método `Game#raffle_word` para uma outra classe, remova esse método do arquivo `lib/game.rb`.

Finalmente, ao rodarmos o RSpec, podemos ver que está tudo no verde!

Vamos rodar o Cucumber e ver como ele está:

```
$ bundle exec cucumber
```

```
(...)
```

```
expected `forca` not to have output on stdout
```

```
but was:
```

```
lib/game.rb:6:in `initialize':
```

```
uninitialized constant Game::WordRaffler (NameError)
```

```
from bin/forca:7:in `new'
```

```
from bin/forca:7:in `<main>'
```

```
(...)
```

3 scenarios (3 failed)

Todos os testes do Cucumber quebraram. Os testes de unidade da classe Game passaram porque eles estão testando essa classe isolada das suas dependências, através de test doubles (mocks e stubs). Já os do do Cucumber quebraram porque eles testam o software como ele é, incluindo a integração entre os objetos.

A mensagem de erro do Cucumber está nos dizendo que não existe uma constante Game::WordRaffler, ou seja, que a classe WordRaffler não foi implementada. Vamos continuar nossa refatoração e implementá-la.

Crie o arquivo spec/word_raffler_spec.rb e escreva os seguintes testes para verificar o comportamento que definimos no protocolo de comunicação com o o word raffler:

```
require "word_raffler"
```

```
RSpec.describe WordRaffler do
```

it

"raffles a word from a given list of words" do

words =

%w(me you nice)

raffler =

WordRaffler

.new(words)

expect(raffler.raffle(
3)).to eq("you"
)

expect(raffler.raffle(
2)).to eq("me"
)

expect(raffler.raffle(
4)).to eq("nice"
)

end

it

"returns nil if it doesn't have a word "

\

"with the given length" do

words =

%w(me you nice)

raffler =

WordRaffler

.new(words)

expect(raffler.raffle(

9

)).to be_nil

end end

Para fazer o teste passar, crie um arquivo lib/word_raffler.rb com o seguinte conteúdo:

```
# Classe responsável por sortear uma palavra de uma dada lista # de palavras. #  
class WordRaffler
```

```
  def initialize(words = %w(hi mom game fruit)  
  )
```

```
    @words  
    = words
```

```
  end
```

```
  def  
    raffle(word_length)
```

@words

```
.detect { |word| word.length == word_length }
```

end end

Ao rodar o RSpec, podemos ver que está no verde. Perceba que a lógica do método raffle dessa nova classe é exatamente igual à do antigo método privado Game#raffle_word. Só extraímos o antigo método para essa nova classe.

Outra coisa que fizemos foi deixar como comentário no começo da classe a sua responsabilidade. Gosto de fazer isso para deixar bem claro qual a responsabilidade da classe em questão. Escrever esse texto nos força a pensar sobre as responsabilidades dessa classe. Isso é bom pois ao longo do tempo podemos verificar se a classe está acumulando mais responsabilidades do que o texto no começo dela está falando.

A última coisa que falta fazer para terminar essa refatoração é dar um require da nova classe dentro da classe Game. Para fazer isso, adicione o require_relative "word_raffler" no começo da classe Game para ficar assim:

```
require_relative "cli_ui"
```

```
require_relative
```

```
"word_raffler"
```

```
class Game
```

```
# (...) end
```

Ao rodar o Cucumber agora, podemos ver que está totalmente verde.
Finalizamos o primeiro ponto de refatoração! Vamos para o próximo.

13.3 Distribuindo responsabilidades para outras classes

Na seção anterior, identificamos que a classe Game tinha pelo menos três responsabilidades:

Interagir com o usuário e executar as ações necessárias baseado no input do usuário. Toda a parte do next_step por exemplo;

Conhecer o estado do jogo: método ended? e variável de instância @ended;

Sortear uma palavra: método raffle_word.

A responsabilidade de sortear uma palavra nós já extraímos, deletando um método privado e criando uma nova classe só para essa responsabilidade.

O próximo ponto será extrair a lógica de fluxo do jogo e interação com usuário da classe Game para uma nova classe. Essa nova classe pode se chamar GameFlow (fluxo do jogo).

Comece criando um arquivo de teste chamado spec/game_flow_spec.rb. Moveremos os testes dos métodos start e next_step do arquivo spec/game_spec.rb para esse novo arquivo. Além de mover os testes, teremos que fazer algumas adaptações neles. Vamos mover de pouco em pouco os testes e entender as modificações necessárias.

O primeiro teste que vamos mover será o do método start, que ficará assim no novo arquivo spec/game_flow_spec.rb:

```
require "game_flow"
```

```
RSpec.describe GameFlow do
```

```
  let(
    :ui) { double("ui"
  ).as_null_object }
```

```
  let(
    :game) { double("game"
  ).as_null_object }
```

```
  subject(
    :game_flow) { GameFlow
    .new(game, ui) }
```

```
  describe
```

```
"#start" do
```

```
  it
```

```
    "prints the initial message" do
```

```
      initial_message =
```

```
      "Bem-vindo ao jogo da forca!"
```

```
      expect(ui).to receive(
```

```
        :write
```

```
      ).with(initial_message)
```

```
      game_flow.start
```

```
    end
```

```
  end
```

```
# (...) end
```

Além de mover esse teste da classe Game para o teste da nova classe GameFlow, tivemos que fazer uma adaptação, mudando onde antes era game para ficar game_flow.

Outro ponto que vale notar é a construção do objeto game_flow:

```
let(:ui) { double("ui"  
  
).as_null_object }  
  
let(  
  
:game) { double("game"  
  
).as_null_object }  
  
subject(  
  
:game_flow) { GameFlow.new(game, ui) }
```

Perceba que ele depende tanto de game quando de ui. Isso porque a responsabilidade desse novo objeto é cuidar da interação com o usuário necessária durante o fluxo do jogo e, ao longo do fluxo do jogo, enviar as mensagens necessária para o objeto game, que é quem vai cuidar das regras do jogo em si.

Após movermos os testes do método start, vamos mover os do método next_step, e olhar como cada uma ficará. O primeiro teste ficará assim:

```
RSpec.describe GameFlow do
```

```
# (...)
```

```
  describe
```

```
    "#next_step" do
```

```
      context
```

```
        "when the game just started" do
```

```
          it
```

```
            "asks the player for the length "
```

```
            \
```

```
            "of the word to be raffled" do
```



```
question =
```

```
"Qual o tamanho da palavra a ser sorteada?"
```

```
expect(ui).to receive(
```

```
:write
```

```
).with(question)
```

```
word_length =
```

```
"3"
```

```
expect(ui).to receive(
```

```
:read
```

```
).and_return(word_length)
```

```
game_flow.next_step
```

```
end
```

```
end
```

```
# (...)
```

```
end end
```

Para esse teste, a única adaptação que tivemos que fazer foi mudar onde antes era game para game_flow.

O teste seguinte que vamos mover ficará dessa forma:

```
RSpec.describe GameFlow do
```

```
# (...)
```

```
  describe
```

```
    "#next_step" do
```

context

"when the player asks to raffle a word" do

it

"raffles a word with the given length" do

word_length =

3

allow(ui).to receive(

:read

).and_return(word_length.to_s)

expect(game).to receive(

:raffle

).with(word_length)

game_flow.next_step

end

end

#(...)

end end

Nesse teste tivemos que mudar onde estava word_raffler para game. Isso porque quem usará diretamente o word_raffler é só o objeto game. O game_flow pede para o game sortear uma palavra e o objeto game usa o word_raffler para isso. Devido a essa nova cadeia de responsabilidades, será necessário que o game implemente um método raffle, que até então não existia.

O próximo teste a ser movido ficará assim:

RSpec.describe GameFlow do

(...)

describe

"#next_step" do

context

"when the player asks to raffle a word" do

it

"prints a '_' for each letter in the raffled word" do

word_length =

"3"

allow(ui).to receive(

:read

).and_return(word_length)

allow(game).to receive(

:raffle).and_return("mom"

)

allow(game).to receive(

```
:raffled_word).and_return("mom"  
)
```

```
    expect(ui).to receive(  
:write).with(" _ _ _"  
)
```

```
    game_flow.next_step
```

```
end
```

```
end
```

```
#(...)
```

```
end end
```

Nesse teste tivemos que fazer um stub no objeto game, porém, antes de mover o teste, nós fazíamos esse stub no objeto word_raffler. Estamos fazendo o stub no objeto game porque ele que é a dependência direta do game_flow e não o word_raffler.

O último teste que vamos mover analisando com detalhes é o seguinte:

```
RSpec.describe GameFlow do
```

```
  # (...)
```

```
    describe
```

```
      "#next_step" do
```

```
        it
```

```
          "finishes the game when the player asks to" do
```

```
            player_input =
```

```
              "fim"
```

```
    allow(ui).to receive(  
      :read  
    ).and_return(player_input)
```

```
    expect(game).to receive(  
      :finish  
    )
```

```
    game_flow.next_step
```

```
  end
```

```
  #(...)
```

```
end end
```

Esse teste verifica se o jogo é terminado quando o jogador digita "fim". Antes nós fazíamos essa verificação olhando o estado do objeto game, fazendo `expect(game).to be_ended`. Após mover o teste para o arquivo

spec/game_flow_spec.rb, não devemos mais testar o estado do objeto game, já que não é ele o objeto sob teste. Então, para fazer esse teste estamos verificando se a mensagem certa está sendo enviada para o objeto game:

```
expect(game).to receive(:finish)
```

Esse teste espera que objeto game tenha um método Game#finish, que ainda não existe, mas vamos implementá-lo depois.

O resultado final dos testes que movemos para o arquivo spec/game_flow_spec.rb ficará assim:

```
require "game_flow"
```

```
RSpec.describe GameFlow do
```

```
  let(
    :ui) { double("ui"
  ).as_null_object }
```

```
  let(
    :game) { double("game"
```

```
).as_null_object }
```

```
subject(  
:game_flow) { GameFlow  
.new(game, ui) }
```

```
describe  
"#start" do
```

```
it  
"prints the initial message" do
```

```
initial_message =  
"Bem-vindo ao jogo da forca!"
```

```
expect(ui).to receive(  
:write  
).with(initial_message)
```

```
game_flow.start
```

end

end

describe

"#next_step" do

context

"when the game just started" do

it

"asks the player for the length "

\

"of the word to be raffled" do

question =

"Qual o tamanho da palavra a ser sorteada?"

```
        expect(ui).to receive(  
:write  
) .with(question)
```

```
        word_length =  
"3"
```

```
        expect(ui).to receive(  
:read  
) .and_return(word_length)
```

```
        game_flow.next_step  
  
end
```

```
end
```

```
context  
  
"when the player asks to raffle a word" do
```

it

"raffles a word with the given length" do

word_length =

3

allow(ui).to receive(

:read

).and_return(word_length.to_s)

expect(game).to receive(

:raffle

).with(word_length)

game_flow.next_step

end

it

"prints a '_' for each letter in the raffled word" do

word_length =

"3"

allow(ui).to receive(

:read

).and_return(word_length)

allow(game).to receive(

:raffle).and_return("mom"

)

allow(game).to receive(

:raffled_word).and_return("mom"

)

expect(ui).to receive(

:write).with("_ _ _"

)

game_flow.next_step

end

it

"tells if it's not possible to raffle "

\

"with the given length" do

word_length =

"9"

allow(ui).to receive(

:read

).and_return(word_length)

allow(game).to receive(

:raffle).and_return(nil

)

error_message =

"Não temos uma palavra com o tamanho desejado,\n"

\

"é necessário escolher outro tamanho."

```
    expect(ui).to receive(  
      :write  
    ).with(error_message)
```

```
    game_flow.next_step
```

end

```
it
```

"finishes the game when the player asks to" do

```
    player_input =  
    "fim"
```



```
        allow(ui).to receive(
:read
).and_return(player_input)

        expect(game).to receive(
:finish
)

        game_flow.next_step

    end

end

end end
```

Antes de rodar esses novos testes, vamos verificar como ficaram os testes da classe Game após termos movido a maior parte deles. Abra o arquivo spec/game_spec.rb, ele deve estar assim:

```
require "game"
```

```
RSpec.describe Game do
```

```
  let(  
    :ui) { double("ui"  
    ).as_null_object }
```

```
  let(  
    :word_raffler) { double("word raffler"  
    ).as_null_object }
```

```
  subject(  
    :game) { Game  
    .new(ui, word_raffler) }
```

```
    describe
```

```
      "#ended?" do
```

```
        it
```

```
"returns false when the game just started" do
```

```
  expect(game).not_to be_ended
```

```
end
```

```
end end
```

Como era de se esperar, todos os testes sobre o fluxo do jogo foram extraídos, ficando apenas o teste do método ended?, que é sobre guardar o estado do jogo. Após ter extraído a maior parte dos testes, será necessário fazer adaptações nos testes da classe Game também. Por exemplo, não precisaremos mais do objeto ui nesse teste, já que toda interação com o usuário será responsabilidade do game_flow.

Vamos deletar a definição do test double ui e adaptar o subject para retirar a dependência de ui. Após essas adaptações, o teste ficará desta maneira:

```
require "game"
```

```
RSpec.describe Game do
```

```
let(  
:word_raffler) { double("word raffler"  
).as_null_object }
```

```
subject(  
:game) { Game  
.new(word_raffler) }
```

```
describe  
"#ended?" do
```

```
it  
"returns false when the game just started" do
```

```
  expect(game).not_to be_ended
```

```
end
```

```
end end
```

Vamos checar se a refatoração da classe Game não quebrou nada. Ao rodar os testes da classe Game, vemos que eles estão passando:

```
$ bundle exec rspec spec/game_spec.rb
```

1 example, 0 failures

Não podemos esquecer também que a classe Game precisa fazer mais algumas coisas das quais a GameFlow depende, que são os métodos raffle e finish. Adicione os seguintes testes para eles no arquivo spec/game_spec.rb:

```
describe "#raffle" do
```

```
  it
```

```
    "raffles a word with the given length" do
```

```
      expect(word_raffler).to receive(
        :raffle).with(3
      )
```

```
      game.raffle(
```

3

)

end

it

"saves the raffled word" do

raffled_word =

"mom"

allow(word_raffler)

.to receive(

:raffle

)

.and_return(raffled_word)

game.raffle(

3

```
)
```

```
expect(game.raffled_word).to eq(raffled_word)
```

```
end end
```

```
describe
```

```
"#finish" do
```

```
it
```

```
"sets the game as ended" do
```

```
game.finish
```

```
expect(game).to be_ended
```

```
end end
```

Para fazê-los passar, adicione os seguintes métodos na classe Game:

```
def
```

```
  raffle(word_length)
```

```
@raffled_word = @word_raffler.raffle(word_length) end
```

```
def
```

```
  ended?
```

```
@ended end
```

```
def
```

```
  finish
```

```
@ended = true end
```

Vamos modificar o construtor para retirar a injeção de dependência ui, já que o Game não depende mais dela. Modifique o construtor para ele ficar assim:

```
def initialize(word_raffler = WordRaffler
```



```
.new)
```

```
@word_raffler
```

```
= word_raffler
```

```
@ended = false end
```

Ao rodar os testes da classe Game podemos verificar que eles estão passando:

```
$ bundle exec rspec spec/game_spec.rb
```

4 examples, 0 failures

Com os testes da classe Game no verde, podemos ir em frente.

Após termos movido os testes de next_step e start para o novo arquivo spec/game_flow_spec.rb, precisamos mover também a lógica necessária para a nova classe. Vamos criar o arquivo lib/game_flow.rb e mover a lógica necessária da classe Game para essa nova classe.

Mova o método start da classe Game para a classe GameFlow, de modo a ficar assim:

```
require_relative "cli_ui"
```

```
require_relative
```

```
"game"
```

```
# Esta classe é responsável pelo fluxo do jogo. # class GameFlow
```

```
def initialize(game = Game.new, ui = CliUi  
.new)
```

```
@game
```

```
= game
```

```
@ui
```

```
= ui
```

```
end
```

```
def
  start
    initial_message =
      "Bem-vindo ao jogo da forca!"
```

```
@ui
  .write(initial_message)
```

```
end
```

```
# (...) end
```

Perceba que, além de mover o método start, também implementamos o construtor da classe GameFlow, explicitando a dependência de game e de ui.

Agora mova o método next_step para a nova classe:

```
class GameFlow
```

```
# (...)
```

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
    player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

```
else
```

```
if @raffled_word = @word_raffler
```

```
.raffle(player_input.to_i)
```

```
  print_letters_feedback
```

```
else
```

```
  error_message =
```

```
    "Não temos uma palavra com o tamanho desejado,\n"
```

```
  \
```

```
    "é necessário escolher outro tamanho."
```

```
@ui
```

```
.write(error_message)
```

```
end
```

```
end
```

```
end end
```

Precisamos fazer alguns ajustes após mover esse método. Na linha onde está:

```
if @raffled_word = @word_raffler.raffle(player_input.to_i)
```

Não podemos mais usar `@word_raffler`, portanto, mude-a para ficar como o que segue:

```
if @game.raffle(player_input.to_i)
```

Agora mova o método `print_letters_feedback` da classe `Game` para a classe `GameFlow` de modo que ela fique assim:

```
# (...) class GameFlow # (...)
```

```
private
```

```
def
```

```
  print_letters_feedback
```

```
    letters_feedback =
```

```
    """
```

```
    @raffled_word.length.times do
```

```
      letters_feedback <<
```

```
      " _ "
```

end

letters_feedback.strip!

@ui.write(letters_feedback) end

Como o game flow não tem acesso à variável de instância @raffled_word, já que isso faz parte do estado interno do objeto game, precisaremos mudar a seguinte linha:

@raffled_word.length.times do end

Modifique-a de modo que ela fique assim:

@game.raffled_word.length.times do end

Após essas modificações, esses métodos ficarão desta forma:

class GameFlow


```
# (...)
```

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
    player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@ended = true
```

else

if @game

.raffle(player_input.to_i)

print_letters_feedback

else

error_message =

"Não temos uma palavra com o tamanho desejado,\n"

\

"é necessário escolher outro tamanho."

@ui

.write(error_message)

end

end

end

private

def

print_letters_feedback

letters_feedback =

"""

@game.raffled_word.length.times do

```
letters_feedback <<  
" "  
  _  
  
end  
  
letters_feedback.strip!  
  
@ui  
.write(letters_feedback)  
  
end end
```

Após toda essa refatoração, ao rodarmos o RSpec inteiro, vemos o seguinte:

```
$ bundle exec rspec
```

Failures:

1) GameFlow#next_step when the player asks to raffle a word

finishes the game when the player asks to

Failure/Error: expect(game).to receive(:finish)

(Double "game").finish(*(any args))

expected: 1 time with any arguments

received: 0 times with any arguments

./spec/game_flow_spec.rb:70:in `block (4 levels)'

12 examples, 1 failure

Muito bom, só um teste está quebrando! Ele quebrou porque, quando game_flow recebeu o input "fim" do jogador, ele não enviou a mensagem finish para o seu colaborador game. O que ele está fazendo até agora nessa situação é setar a variável de instância @ended:

def

next_step

(...)

```
if player_input == "fim"
```

```
  @ended = true
```

```
else
```

```
  # (..)
```

```
end end
```

A variável de instância `@ended` faz parte do estado interno do objeto `game`, não do `game_flow`, então precisamos modificar isso. O que precisamos fazer é, em vez de setar essa variável, enviar a mensagem `finish` para o objeto `game`. Faça essa modificação nessa linha do método `next_step` de modo que ela fique assim:

```
def
```

```
  next_step
```

```
  # (...)
```

```
  if player_input == "fim"
```

```
    @game
```

```
    .finish
```

```
  else
```

```
    # (.)
```

```
  end end
```

Depois de tudo isso, finalmente ao rodarmos o RSpec, tudo está no verde!

```
$ bundle exec rspec
```

12 examples, 0 failures

RSpec no verde, vamos ver o Cucumber. Ao rodarmos o Cucumber, vemos o seguinte:

```
$ bundle exec cucumber
```

(...)

expected `forca` not to have output on stdout

but was:

lib/game.rb:14:in `start':

undefined method `write' for nil:NilClass (NoMethodError)

from bin/forca:8:in `<main>'

(...)

3 scenarios (3 failed)

Todos os testes estão quebrando. O problema apontado pela mensagem de erro é

que não existe o método start na classe Game. Claro! Porque extraímos esse e outros métodos para a classe GameFlow. Precisamos atualizar o binário do nosso jogo para resolver esse problema.

Edite o arquivo bin/forca para ficar desta maneira:

```
#!/usr/bin/env ruby
```

```
$LOAD_PATH.prepend File.join(__dir__, "../lib")
```

```
require "game_flow"
```

```
game_flow =
```

```
  GameFlow
```

```
  .new
```

```
  game_flow.start
```

```
until
```

```
game_flow.ended?  
  game_flow.next_step  
end
```

Ao rodar o Cucumber de novo, vemos o seguinte:

```
$ bundle exec cucumber
```

```
(...)
```

```
expected `forca` not to have output on stdout
```

```
but was:
```

```
forca:10:in `
```

```
<main>
```

```
': undefined method `ended?'
```

```
for
```

```
#<GameFlow:0x00007fabed1aea28> (NoMethodError)
```

```
(...)
```

```
3 scenarios (3 failed)
```

Infelizmente, todos os testes ainda estão quebrando. Mas felizmente, a

mensagem de erro mudou, o que quer dizer que estamos avançando. A mensagem de erro indica que o problema é que a classe GameFlow não tem o método ended?. De fato, ela não o tem, porque quem guarda o estado do jogo é a classe Game. Porém, podemos fazê-la delegar esse método para sua dependência game. Vamos usar o Forwardable da biblioteca padrão do Ruby para fazer isso. Edite o começo do arquivo lib/game_flow.rb para ficar assim:

```
# (...)
```

```
require "forwardable"
```

```
# Esta classe é responsável pelo fluxo do jogo. # class GameFlow
```

```
  extend
```

```
  Forwardable
```

```
  delegate
```

```
  :ended? => :@game
```

(...) end

O que fizemos agora foi usar o Forwardable para delegar o método ended? para o objeto dependência game.

Ao rodar o Cucumber, ele finalmente está no verde! Cucumber e RSpec no verde, mais um passo da refatoração finalizado, missão cumprida.

No começo do capítulo identificamos três pontos de refatoração:

extração da responsabilidade de sortear palavra;

extração da responsabilidade de fluxo do jogo;

redução de condicionais aninhadas no método next_step.

Conseguimos refatorar os dois primeiros pontos. O último ponto vamos deixar na nossa lista de dívidas técnicas e atacá-lo nos próximos capítulos, durante o desenvolvimento da próxima funcionalidade. Mas antes de irmos em frente vamos pensar um pouco nos pontos-chaves deste capítulo.

Pontos-chaves deste capítulo

Neste capítulo nós refatoramos o nosso código, melhorando a arquitetura e a qualidade. Começamos identificando os pontos de melhoria através do feedback dos nossos testes e da detecção de alguns code smells comuns, tais como

métodos privados fazendo coisas demais e muitas condicionais aninhadas.

Antes da refatoração, nosso projeto tinha apenas duas classes: Game e CliUi. Ao final do processo de refatoração, ficamos com quatro classes no total, adicionando duas novas classes: WordRaffler e GameFlow. Fizemos isso para reduzir o número de responsabilidades que a classe Game estava abrigando, aumentando a sua coesão.

A primeira refatoração que fizemos foi extrair uma classe de um método privado. No caso, extraímos a classe WordRaffler. A segunda refatoração que fizemos foi baseada no feedback dos testes da classe Game. Nós notamos que havia muitos testes para um método só, o método next_step, o que justificou extrair uma classe nova que abrigaria a responsabilidade relacionada a esse método, no caso, a classe GameFlow.

Nós fizemos todo o processo de refatoração com suporte da nossa suíte de testes. A cada pequeno passo, nós rodamos a suíte para saber se havíamos quebrado algo e o que precisaria ser adaptado. Fazer refatoração sem uma suíte de testes automatizados é muito arriscado, você pode quebrar alguma parte do sistema sem nem mesmo perceber.

Portanto, uma das vantagens do TDD é que a suíte de testes que construímos durante o design do nosso código serve também como uma suíte de testes de regressão. E é ela que nos dá confiança e produtividade na melhoria contínua da qualidade do nosso código.

Capítulo 14

Especificando a segunda funcionalidade

No capítulo anterior, nós refatoramos o nosso código para melhorar sua qualidade, facilitando a adição das funcionalidades que estariam por vir. Neste capítulo começaremos a desenvolver uma dessas novas funcionalidades, a de adivinhar letra.

Para isso, continuaremos seguindo o fluxo de outside-in development com BDD, começando com testes de aceitação com Cucumber e depois indo para os testes de unidade com RSpec.

Hora de colocar a mão na massa! (Opa, no código).

14.1 Documentando especificação e critério de aceite com Cucumber

Na seção Definindo o escopo da nossa aplicação: Jogo da Forca definimos que o nosso jogo teria três funcionalidades:

Jogador começa um novo jogo: jogo mostra a mensagem inicial e pede para o jogador sortear uma palavra;

Jogador adivinha uma letra da palavra: jogador tenta adivinhar uma letra e o jogo mostra se ele acertou ou errou;

Fim do jogo: o jogo termina quando o jogador acerta todas as letras ou erra o bastante de modo que o corpo inteiro do boneco apareça na forca.

A primeira funcionalidade nós já desenvolvemos e refatoramos nos capítulos anteriores. Vamos agora começar a desenvolver a segunda. Antes de começar, precisamos primeiro pensar na sua especificação e nos seus critérios de aceite. Como aprendemos nos capítulos anteriores, vamos utilizar o Cucumber para documentar a especificação e os critérios de aceite dessa funcionalidade.

Para começar a fazer isso, crie o arquivo `features/adivinhar_letra.feature` e comece a preenchê-lo com a especificação dessa funcionalidade, que será assim:

```
# language: pt
```


Funcionalidade: Adivinhar letra

Após a palavra do jogo ser sorteada, o jogador pode começar a tentar adivinhar as letras da palavra.

Cada vez que ele acerta uma letra, o jogo mostra para ele em que posição dentro da palavra está a letra que ele acertou.

Cada vez que o jogador erra uma letra, uma parte do boneco da forca aparece na forca. O jogador pode errar no máximo seis vezes, que correspondem às seis partes do boneco: cabeça, corpo, braço esquerdo, braço direito, perna esquerda, perna direita.

Através dessa especificação já podemos ter um entendimento melhor de como será essa funcionalidade. Mas só isso não é o bastante para começarmos a desenvolvê-la. Além dessa descrição mais genérica, é necessário definir seu critério de aceite. Para defini-lo, podemos escrever alguns exemplos sobre como deve ser o comportamento dessa funcionalidade em cenários específicos. Os exemplos que utilizaremos para nos guiar como critério de aceite serão os seguintes:

Cenário: Sucesso ao adivinhar letra

Se o jogador adivinhar a letra com sucesso, o jogo mostra uma mensagem de sucesso e mostra em que posição está a letra que o jogador adivinhou.

Cenário: Erro ao adivinhar letra

Se o jogador errar ao tentar adivinhar a letra, o jogo mostra uma mensagem de erro e mostra quais as partes o boneco da forca já perdeu.

Cenário: Jogador advinha com sucesso duas vezes

Quanto mais o jogador for acertando, mais o jogo vai mostrando para ele as letras que ele adivinhou.

Cenário: Jogador erra três vezes ao adivinhar letra

Quanto mais o jogador for errando, mais partes do boneco da forca são perdidas.

Esses cenários já são o bastante para cobrir os critérios de aceite dessa funcionalidade. Ao definir os exemplos para uma funcionalidade, você deve ter no mínimo o cenário de sucesso e o cenário de erro. Foi o que fizemos ao escrever os dois primeiros cenários acima.

Além dos cenários básicos de sucesso e erro, definimos também mais dois cenários, para deixar claro o que acontece quando o jogador acerta ou erra mais de uma vez ao tentar adivinhar uma letra da palavra sorteada.

Agora que já temos a especificação da funcionalidade, assim como a definição dos cenários que compõem o seu critério de aceite, podemos começar a implementar o teste de aceitação do nosso primeiro cenário.

14.2 Definindo o teste de aceitação do primeiro cenário

No nosso primeiro cenário, Sucesso ao adivinhar letra, vamos simular um jogador iniciando o jogo, escolhendo que a palavra sorteada deverá ter três letras e depois adivinhando uma letra dessa palavra com sucesso. Para fazer isso, escreva os seguintes steps no arquivo `features/adivinhar_letra.feature`:

@wip

Cenário: Sucesso ao adivinhar letra

Se o jogador adivinhar a letra com sucesso, o jogo mostra uma mensagem de sucesso e mostra em que posição está a letra que o jogador adivinhou.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem

"3"

letras

Quando tento adivinhar que a palavra tem a letra

"a"

E termino o jogo

Então o jogo mostra que eu adivinhei uma letra com sucesso

E o jogo termina com a seguinte mensagem na tela:

""

a _ _

""

Antes de rodarmos o Cucumber, vamos analisar um pouco mais esses steps. Dentro desses steps, temos a seguinte sequência:

jogador começa um jogo;

ele escolhe que o tamanho da palavra a ser sorteada será de 3 letras;

ele tenta adivinhar que a palavra tem a letra "a";

ele acerta a letra e o jogo mostra que ele acertou.

Ao ler esses steps e pensar em como vamos implementá-los, podemos nos fazer a seguinte pergunta: "como que sabemos que, nesse teste, quando jogador tentar adivinhar que a palavra tem a letra 'a', ele vai acertar?". No teste está definido que ele vai acertar, mas o motivo de ele acertar e de a palavra com 3 letras ter a letra "a" dentro dela não está explícito! Deixar coisas implícitas em um teste é um problema. Isso ocorre por dois motivos:

Para podermos implementar um teste de modo que ele seja determinístico, é necessário explicitar o que ele precisa para funcionar (setup do teste);

Quando alguém for lê-lo, deve ser possível que ele entenda a relação de causa e consequência do seu teste. Ou seja, deve ser possível que ele entenda o seguinte fluxo: dado que o sistema está em um determinado estado, quando realizo uma determinada ação nesse sistema, então observo uma dada consequência. Quando não é possível entender o fluxo do seu teste, é provável que ele esteja sofrendo do smell *Obscure test* (teste obscuro). Você pode ler sobre esse e outros test smells no livro *xUnit Test Patterns* (MESZAROS, 2007).

Do modo como definimos nosso teste, ele está pecando nos dois pontos. Para que o jogador adivinhe com sucesso a letra "a" da palavra sorteada, devemos garantir no setup do nosso teste que a palavra de três letras a ser sorteada terá a letra "a".

O outro problema é que para o leitor não está claro por que o jogador consegue adivinhar com sucesso a letra "a", já que o teste não fala qual é a palavra sorteada.

Vamos resolver esses problemas adicionando mais um step para deixar explícito que, se uma palavra de três letras for sorteada, ela terá a letra "a". Vamos adicionar esse step usando a funcionalidade de background (contexto) do Cucumber. Adicione o seguinte step antes do cenário Sucesso ao adivinhar letra no arquivo `features/adivinhar_letra.feature`:

Contexto:

*

o jogo tem as possíveis palavras para sortear:

| número de letras | palavra sorteada |

| 3 | avo |

Após adicionar esse contexto na nossa funcionalidade, o seu arquivo `features/adivinhar_letra.feature` deve estar assim:

```
# language: pt
```

Funcionalidade: Adivinhar letra

Após a palavra do jogo ser sorteada, o jogador pode começar a tentar adivinhar as letras da palavra.

Cada vez que ele acerta uma letra, o jogo mostra para ele em que posição dentro da palavra está a letra que ele acertou.

Cada vez que o jogador erra uma letra, uma parte do boneco da forca aparece na forca. O jogador pode errar no máximo seis

vezes, que correspondem às seis partes do boneco: cabeça, corpo, braço esquerdo, braço direito, perna esquerda, perna direita.

Contexto:

*

o jogo tem as possíveis palavras para sortear:

| número de letras | palavra sorteada |

| 3 | avo |

@wip

Cenário: Sucesso ao adivinhar letra

Se o jogador adivinhar a letra com sucesso, o jogo mostra uma mensagem de sucesso e mostra em que posição está a letra que o jogador adivinhou.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem "3" letras

Quando tento adivinhar que a palavra tem a letra "a"

E termino o jogo

Então o jogo mostra que eu adivinhei uma letra com sucesso

E o jogo termina com a seguinte mensagem na tela:

```
*****
```

```
a _ _
```

```
*****
```

Repare que no cenário Sucesso ao adivinhar letra, quando o jogador escolhe que a palavra sorteada deverá ter 3 letras, e adivinha com sucesso que ela tem a letra "a", o leitor já poderá entender a relação de causa e consequência desse teste. Ao lê-lo, ele pode verificar que o jogador conseguiu adivinhar com sucesso a letra "a", porque no contexto desse teste definimos que, se a palavra sorteada tiver três letras, então ela será a palavra "avo", que possui a letra "a".

Agora que nosso cenário está claro, vamos rodar o Cucumber e descobrir qual o próximo passo:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 undefined)
```

```
7 steps (3 skipped, 4 undefined)
```

```
0m0.032s
```

You can implement step definitions for undefined steps
with these snippets:

```
Dado("o jogo tem as possíveis palavras para sortear:") do |table|  
  # table is a Cucumber::MultilineArgument::DataTable  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

```
Dado("que escolhi que a palavra a ser sorteada  
  tem {string} letras") do |string|  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

```
Quando("tento adivinhar que a palavra tem  
  a letra {string}") do |string|  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

```
Então("o jogo mostra que eu adivinhei uma letra com sucesso") do  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end
```

Pela saída do Cucumber, podemos ver que, dos sete steps usados nesse cenário, temos três que já estão definidos e outros quatro que ainda falta definirmos. Vamos começar definindo o step o jogo tem as possíveis palavras para sortear.

14.3 Melhore a testabilidade do seu software

O próximo step que vamos implementar é o o jogo tem as possíveis palavras para sortear. Ele serve para definir a lista de palavras possíveis que o nosso jogo pode sortear. Mas até então o nosso software ainda não tem essa interface para definir a lista de palavra sorteáveis. Precisamos implementá-la.

A definição padrão da lista de palavras sorteáveis está em um array hard-coded na classe WordRaffler :

```
class WordRaffler
```

```
def initialize(words = %w(hi mom game fruit)
```

```
)
```

```
@words
```

```
= words
```

```
end
```

```
# (...) end
```

Repare nesse código que, apesar de o comportamento normal do nosso jogo ser utilizar somente as palavras "hi, mom, game, fruit" como opções para o sorteio, a definição dessa lista pode ser injetada no WordRaffler. Essa lista de 4 palavras é apenas uma lista default, caso uma outra lista não seja passada como argumento.

Fizemos o design dessa classe usando injeção de dependência porque os testes dela nos "obrigaram". Ou seja, para que fosse possível testá-la, era necessário poder passar uma lista de palavras como argumento, como você pode ver nos testes da classe WordRaffler:

```
RSpec.describe WordRaffler do
```

```
  it
```

```
    "raffles a word from a given list of words" do
```

```
      words =
```

```
        %w(me you nice)
```

```
raffler =  
WordRaffler  
.new(words)  
  
expect(raffler.raffle(  
3)).to eq("you"  
)  
expect(raffler.raffle(  
2)).to eq("me"  
)  
expect(raffler.raffle(  
4)).to eq("nice"  
)  
  
end
```

```
# (...) end
```

Como você verá mais à frente, esse design nos permitirá de modo mais fácil

criar uma interface para setar a lista de palavras sorteáveis do nosso jogo durante os testes de Cucumber. Precisaremos disso para implementar o step o jogo tem as possíveis palavras para sortear.

Os testes da classe WordRaffler foram a motivação para termos feito a lista de palavras sorteáveis como uma injeção de dependência da classe WordRaffler. Essa consequência "acidental" dos nossos testes para melhorar a testabilidade da classe WordRaffler nos levou a um design melhor, e que vai nos possibilitar construir um modo de setarmos as palavras sorteáveis do nosso jogo durante os testes de Cucumber.

Como você já ouviu falar, usar Test Driven Development faz com que seu software tenha um design melhor. O que aconteceu aqui é um exemplo claro e simples disso.

Voltando à implementação dos nossos steps, vamos rodar o Cucumber mais uma vez para nos lembrar qual o próximo passo:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 undefined)
```

```
7 steps (3 skipped, 4 undefined)
```

```
0m0.047s
```

You can implement step definitions for undefined steps
with these snippets:

```
Dado("o jogo tem as possíveis palavras para sortear:") do |table|  
  # table is a Cucumber::MultilineArgument::DataTable  
  pending # Write code here that turns the phrase above  
           # into concrete actions  
end
```

(...)

Pela saída do Cucumber, podemos nos lembrar de que o próximo passo é definir o step o jogo tem as possíveis palavras para sortear. Vamos fazer isso.

Construindo uma interface para definir as palavras sorteáveis

O step o jogo tem as possíveis palavras para sortear vai implementar a lógica necessária para definir as palavras sorteáveis durante um determinado cenário de Cucumber. Antes de começarmos a implementá-lo, vamos pensar um pouco no que ele vai fazer e como.

Nossos testes executam nosso jogo pela mesma interface do usuário, via linha de comando. Logo, precisamos que a interface de linha comando do nosso jogo

possibilite definir a lista de palavras sorteáveis. Um modo de fazer isso seria passando para o binário do nosso jogo um argumento com essa lista. Esse argumento seria usado do seguinte modo:

```
$ bin/forca "oi ola voce"
```

Nessa ideia estamos passando a lista de palavras sorteáveis como argumento para o binário do nosso jogo. Vamos implementar isso.

Para isso, precisaremos editar o binário bin/forca para lidar com esse argumento e passar para o WordRaffler essa lista de palavras como a lista de palavras sorteáveis do jogo. Esse argumento só será usado normalmente no ambiente de teste, logo, o binário vai ter que lidar com os dois casos, quando for passado o argumento e quando não for.

Até então, o binário do nosso jogo está assim:

```
#!/usr/bin/env ruby
```

```
$LOAD_PATH.prepend File.join(__dir__, "../lib"  
)
```

```
require "game_flow"
```

```
game_flow =
```

```
  GameFlow
```

```
  .new
```

```
  game_flow.start
```

```
until
```

```
  game_flow.ended?
```

```
    game_flow.next_step
```

```
end
```

Precisamos modificá-lo para que ele lide com o novo argumento que estamos adicionando. Para que possamos setar a lista de palavras sorteáveis do nosso jogo, precisamos passar essa lista para o construtor do WordRaffler. O WordRaffler é uma dependência da classe Game, que uma é dependência da classe GameFlow. Portanto, teremos que construir um WordRaffler com a lista de palavras que virá como argumento pela linha de comando, passar esse objeto word_raffler como argumento para o Game e passar esse Game como argumento para o GameFlow.

Edite o arquivo bin/forca para que ele faça isso e fique do seguinte modo:

```
# (...)
```

```
def
```

```
  create_game
```

```
    word_raffler = create_word_raffler
```

```
  Game.new(word_raffler) end
```

```
def
```

```
  create_word_raffler
```

```
if ARGV
```

```
  .first
```

```
    words =
```

```
  ARGV
```

```
  .pop
```

```
    words = words.split
```

```
WordRaffler
```

```
.new(words)
```

```
else
```

```
WordRaffler
```

```
.new
```

```
end end
```

```
game = create_game
```

```
game_flow =
```

```
GameFlow
```

```
.new(game)
```

```
game_flow.start
```

```
until
```

```
game_flow.ended?
```

```
  game_flow.next_step
```

end

Repare na maneira como estamos criando o objeto WordRaffler. Estamos checando se está vindo algum argumento da linha de comando e, se estiver, estamos usando esse argumento como lista de palavras sorteáveis do jogo.

Com o binário modificado, podemos começar a implementar o step o jogo tem as possíveis palavras para sortear para que ele faça uso desse novo argumento. Esse step definition vai ter que usar a lista de palavras sorteadas desta tabela:

Contexto:

*

o jogo tem as possíveis palavras para sortear:

número de letras	palavra sorteada
------------------	------------------

3	avo	
---	-----	--

E passar as palavras da segunda coluna para o WordRaffler. Vamos implementar esse step definition do seguinte modo, no arquivo features/step_definitions/game_steps.rb:

Dado "o jogo tem as possíveis palavras "

```
\

"para sortear:" do

|words_table|

  words = words_table.rows.map(&

:last).join(" ")

)

  set_rafflable_words(words)

end
```

Note que o método `set_rafflable_words` ainda não existe, estamos usando um código que gostaríamos que existisse. Fizemos desse jeito para deixar o step definition bem simples e delegar a maior parte da lógica para esse método, que fará parte do support code.

A ideia do método `set_rafflable_words` é que ele salve a lista de palavras sorteáveis em uma variável de instância, para que ela seja usada pelo step que inicia o jogo de fato. Vamos implementar esse método no arquivo `features/support/game_helpers.rb`:

```
module GameHelpers
```

```
def
```

```
  set_raffable_words(words)
```

```
  @raffable_words
```

```
  = words
```

```
end
```

```
# (...) end
```

Com esse helper method definido, precisamos atualizar os steps que iniciam um novo jogo para utilizar a lista de palavras sorteáveis definida por esse método, caso ela exista.

Podemos ver no arquivo `features/step_definitions/game_steps.rb` que existem os seguintes step definitions sobre começar um novo jogo:

Dado "que comecei um jogo" do

```
  start_new_game
```

```
end
```

Quando "começo um novo jogo" do

```
start_new_game
```

```
end
```

Ambos os step definitions estão delegando toda lógica para o support code, sendo assim, basta que modifiquemos o método `start_new_game` para utilizar a lista definida pelo método `set_raffable_words`. Repare como foi uma boa decisão delegar a lógica desses step definitions para o support code, pois agora só precisamos mudar em um único lugar, ao invés de ter que mudar todos os step definitions que têm a ver com iniciar um novo jogo.

Vamos modificar o método `start_new_game` no arquivo `features/support/game_helpers.rb` para que ele utilize a lista de palavras sorteáveis:

```
module GameHelpers
```

```
def
```

```
set_raffable_words(words)
```



```
@raffable_words
```

```
= words
```

```
end
```

```
def
```

```
start_new_game
```

```
if @raffable_words.nil
```

```
?
```

```
    set_raffable_words(
```

```
"hi mom game fruit"
```

```
)
```

```
end
```

```
run_command(
```

```
"bin/forca \"#{@raffable_words}\"", exit_timeout: 3
```

```
)
```

```
end end
```

O que mudamos no método `start_new_game` foram duas coisas. Ele define a lista de palavras sorteáveis usando o método `set_raffable_words` e, ao executar nosso jogo pela linha de comando via Aruba, está passando essa lista de palavras como argumento.

Vamos rodar o Cucumber, e verificar se está tudo ok:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 undefined)
```

```
7 steps (2 skipped, 3 undefined, 2 passed)
```

```
0m0.043s
```

You can implement step definitions for undefined steps

with these snippets:

```
Dado("que escolhi que a palavra a ser sorteada  
    tem {string} letras") do |string|  
  pending # Write code here that turns the phrase above  
    # into concrete actions  
end  
  
(...)
```

■

Testabilidade

Testabilidade é um requisito não funcional que mede "quão fácil" é testar um software. Um dos fatores que compõem a testabilidade de um software é sua controlabilidade, ou seja, em que nível é possível controlar o estado do software para que seja mais fácil testá-lo.

O step o jogo tem as possíveis palavras para sortear foi feito para aumentar a controlabilidade do nosso jogo.

No ambiente de produção o jogador não poderá definir, ou saber, quais são as palavras disponíveis para sorteio dentro do nosso jogo. Mas, no ambiente de teste é necessário que se possa controlar quais palavras estão disponíveis, para

que seja possível testar o programa de modo determinístico.

Por fim, melhorar a testabilidade do nosso jogo nos possibilita escrever um teste de Cucumber melhor, com o qual o leitor possa entender a relação de causa e consequência entre o estado inicial do sistema e a lógica de verificação (etapa de setup e de verificação).

■

O step o jogo tem as possíveis palavras para sortear passou. O próximo step a ser definido é o E que escolhi que a palavra a ser sorteada tem "3" letras.

A implementação desse step consiste apenas em digitar o tamanho da palavra a ser sorteada. Como fizemos antes, vamos utilizar o Aruba para simular essa interação com a linha de comando. Adicione o seguinte step definition no arquivo `features/step_definitions/game_steps.rb`:

Dado "que escolhi que a palavra a ser sorteada "

\

"tem {string} letras" do

|number_of_letters|

type(number_of_letters)

end

Ao rodar o Cucumber novamente, vemos que esse step passou. O próximo step pendente é o Quando tento adivinhar que a palavra tem a letra "a". Assim como o step acima, esse é apenas mais uma interação de escrita com a linha de comando, então podemos utilizar o Aruba para implementá-lo. Adicione a seguinte implementação desse step definition no arquivo `features/step_definitions/game_steps.rb`:

Quando "tento adivinhar que a palavra "

\

"tem a letra {string}" do

|letter|

type(letter)

end

Ao rodar o Cucumber novamente, vemos que só sobrou um step pendente:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 undefined)
```

```
7 steps (1 skipped, 1 undefined, 5 passed)
```

0m0.175s

You can implement step definitions for undefined steps
with these snippets:

```
Então("o jogo mostra que eu adivinhei uma letra com sucesso") do  
  pending # Write code here that turns the phrase above  
           # into concrete actions  
end
```

O step pendente é o Então o jogo mostra que eu adivinhei uma letra com sucesso. O jogo mostrará para o jogador que ele adivinhou uma letra com sucesso, imprimindo na tela a mensagem "Você adivinhou uma letra com sucesso.". O que a definição desse step deve fazer é apenas verificar se essa mensagem foi impressa na tela. Vamos fazer isso usando o Aruba, adicionando o seguinte step definition no arquivo features/step_definitions/game_steps.rb:

```
Então "o jogo mostra que eu adivinhei uma letra com sucesso" do
```

```
  expect(last_command_stopped.stdout).to
```

```
  include
```

```
(
```

"Você adivinhou uma letra com sucesso."

)

end

Ao rodar o Cucumber, não temos mais steps pendentes e o cenário finalmente quebra:

```
$ bundle exec cucumber -t @wip
```

(...)

Diff:

```
@@ -1,2 +1,8 @@
```

```
-Você adivinhou uma letra com sucesso.
```

```
+Bem-vindo ao jogo da forca!
```

```
+Qual o tamanho da palavra a ser sorteada?
```

(...)

1 scenario (1 failed)

Como você pôde ver pela mensagem de falha do Cucumber, o cenário falhou

porque o jogo não imprimiu na tela a mensagem "Você adivinhou uma letra com sucesso.". Era o que esperávamos. Agora que finalizamos a especificação e o teste de aceitação do primeiro cenário da funcionalidade Adivinhar letra, podemos continuar o fluxo de outside-in development e partir para um teste de RSpec. Mas vamos deixar para fazer isso só no capítulo seguinte.

Pontos-chaves deste capítulo

Neste capítulo nós começamos a especificação da funcionalidade Adivinhar letra. O desenvolvimento dessa especificação foi um pouco diferente do que tínhamos feito até então. Primeiro, começamos com uma descrição da funcionalidade em texto aberto para deixar claro e documentado o entendimento do escopo dessa funcionalidade.

Após isso, definimos quais seriam os cenários de teste que iriam compor o critério de aceite dessa funcionalidade. Escrevemos não só o título dos cenários de teste como também uma descrição de cada um deles. Só depois disso tudo que começamos a implementar de fato o primeiro cenário de teste de aceitação.

A ideia de ter feito esse fluxo na especificação (requisitos em texto aberto, definição dos cenários e por último implementação dos cenários) era de mostrar que vale a pena pensar um pouco no escopo da funcionalidade antes até de implementar o primeiro teste de aceitação. Desse modo, o requisito da funcionalidade fica mais claro para você e para futuros membros do projeto, que poderão ler uma documentação sucinta do requisito juntamente com os seus testes automatizados. Juntando assim escopo, critério de aceite e testes automatizados.

Durante a implementação do primeiro cenário, tivemos também uma boa

discussão sobre testabilidade, quando foi necessário criar um modo de definir a lista de palavras sorteáveis do nosso jogo no contexto de um teste de aceitação. Há quem fale que você nunca deve mudar seu código só por causa de um teste. Mas cada caso é um caso. No nosso, ao melhorarmos a controlabilidade do nosso software, o deixamos mais maleável e também mais fácil de testar. Seguir o feedback dos testes costuma levar a um design melhor do seu software, mas cabe a você decidir se uma mudança no código de produção devido a um teste vai de fato melhorar seu design ou se é só para fazer o teste passar.

Para finalizar, vale a pena lembrar que Testabilidade é também um requisito de software, assim como um requisito funcional. Se seu software tiver uma testabilidade baixa, será difícil de testá-lo. Se for difícil de testá-lo, fica difícil de praticar TDD e, ainda pior, é provável que o motivo de ser difícil de testar seu software é porque o design está ruim.

Capítulo 15

Finalizando a segunda funcionalidade

No capítulo anterior, fizemos a especificação da funcionalidade "Adivinhar letra" e o teste de aceitação do seu primeiro cenário. Neste capítulo vamos continuar trabalhando nessa funcionalidade, finalizando seus testes e sua implementação.

Ao final deste capítulo, o jogador do nosso jogo já vai poder adivinhar as letras da palavra sorteada.

Quem viver, verá!

15.1 Refatorando nosso jogo para ter uma máquina de estados

Já finalizamos o primeiro cenário da funcionalidade Adivinhar letra, que foi o Sucesso ao adivinhar letra.

Seu teste de aceitação estava falhando porque o jogo não imprimiu a mensagem "Você adivinhou uma letra com sucesso.". Dado que o teste de Cucumber está quebrando, o próximo passo é fazermos um teste de unidade com RSpec para especificar o comportamento necessário.

O comportamento referente a "jogo imprime uma mensagem de sucesso quando o jogador adivinha uma letra" faz parte do fluxo do jogo. Então, vamos escrever o teste desse comportamento como parte dos testes do método `GameFlow#next_step`.

O que esse teste precisa fazer é:

dado que o jogo está no estado que já tem uma palavra sorteada;

quando o jogador adivinhar uma letra com sucesso;

então o jogo imprime a mensagem "Você adivinhou uma letra com sucesso".

Para implementar essas ações no nosso teste, adicione o seguinte código no arquivo spec/game_flow_spec.rb:

```
describe "#next_step" do
```

```
  (...)
```

```
  context
```

```
    "when the player guess a letter with success" do
```

```
      it
```

```
        "prints a success message" do
```

```
          allow(game).to receive(
            :raffled_word).and_return("hey"
          )
```

```
          success_message =
```

```
            "Você adivinhou uma letra com sucesso."
```

```
          expect(ui).to receive(
```

```
:write
```

```
).with(success_message)
```

```
game.next_step
```

```
end
```

```
end end
```

Ao rodar o RSpec, o teste quebra como esperado. Para fazê-lo passar, será necessário mudar o comportamento do método `GameFlow#next_step`. Vamos analisar como esse método está hoje e pensar em como podemos modificá-lo:

```
def
```

```
next_step
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@game
```

```
.finish
```

```
else
```

```
if @game
```

```
.raffle(player_input.to_i)
```

```
  print_letters_feedback
```

```
else
```

```
  error_message =
```

```
"Não temos uma palavra com o tamanho desejado,\n"
```

\

"é necessário escolher outro tamanho."

@ui

.write(error_message)

end

end end

O que esse método está fazendo até então é lidar com o fluxo do jogo no estado em que o jogo ainda não tem uma palavra sorteada. Nesse estado, o fluxo consiste em perguntar para o jogador o tamanho da palavra a ser sorteada e sorteá-la. O que precisamos fazer é o passo seguinte do jogo, no estado no qual a palavra já foi sorteada e o jogo deve pedir pro jogador adivinhar uma letra.

Para implementar esse próximo passo do jogo, seria necessário adicionar mais uma condicional no método `next_step`, checando se existe uma palavra sorteada no jogo ou não. Adicionar mais uma condicional nesse método não é uma boa ideia. Antes mesmo de pensar em adicioná-la, nós já tínhamos identificado que o aninhamento de condicionais atual nesse método já era uma dívida técnica. Não

queremos piorar a dívida técnica aumentando mais ainda esse aninhamento de condicionais.

Uma outra solução para controlar o fluxo do jogo seria salvar o estado do jogo no objeto game e utilizar essa informação para controlar o fluxo. Se o objeto game estiver no estado :initial (estado inicial), o próximo passo do fluxo do jogo deve ser o de sorteio de palavra. Se o estado do game for :word_raffled (palavra sorteada), o próximo passo do fluxo do jogo é pedir para o jogador adivinhar uma letra.

Vamos implementar essa solução, começando por novos testes da classe Game. Como vamos mudar a classe Game, mas o teste novo que adicionamos da classe GameFlow ainda está quebrado, vamos marcar esse teste como pendente, para que a suíte possa rodar no verde enquanto modificamos a classe Game. Para fazer isso, podemos usar o método xit do RSpec. Edite o novo teste no arquivo spec/game_flow_spec.rb para ficar assim:

```
context "when the player guess a letter with success" do
```

```
  xit
```

```
  "prints a success message" do
```

```
    # (...)
```

end end

Após essa edição, ao rodar a suíte de teste do RSpec, você pode ver que ela inteira passou e que temos um teste pendente:

```
$ bundle exec rspec
```

```
(...)
```

```
13 examples, 0 failures, 1 pending
```

Com a suíte no verde, vamos fazer um teste para verificar que um jogo novo começa no estado :initial. Adicione o seguinte teste no arquivo spec/game_spec.rb:

```
context "when just created" do
```

```
  it
```

```
    "has the :initial state" do
```

```
      expect(game.state).to eq(
```

```
        :initial
```

```
      )
```

```
end end
```

Para fazer esse teste passar, podemos inicializar o estado de um objeto game como :initial no seu construtor e adicionar um attr_writer: state na classe:

```
class Game
```

```
  attr_accessor :raffled_word
```

```
  attr_accessor :state
```

```
  def initialize(word_raffler = WordRaffler  
    .new)
```

```
    @word_raffler
```

```
    = word_raffler
```

```
@ended = false
```

```
@state = :initial
```

```
end end
```

Ao rodar o RSpec, você pode ver que o teste passou. Vamos para o próximo.

O próximo teste será sobre a transição de estado do objeto game. Quando uma palavra for sorteada com sucesso, o game deve ir do estado :initial para estado :word_raffled. Especifique esse comportamento adicionando o seguinte teste no arquivo spec/game_spec.rb:

```
describe "#raffle" do
```

```
# (...)
```

it

"makes a transition from :initial "

\

"to :word_raffled on success" do

allow(word_raffler).to receive(
:raffle).and_return("mom"
)

expect { game.raffle(
3
) }

.to change { game.state }
.from(
:initial
)

.to(
:word_raffled
)

end end

Note como nossa spec ficou bonita ao usarmos o matcher expect change do RSpec! Após apreciarmos um pouco a sintaxe do RSpec, vamos fazer o teste passar.

Para fazer isso, basta que modifiquemos o método Game#raffle para fazer a transição do estado quando o sorteio da palavra for feito com sucesso pelo word_raffler. Seguindo essa ideia, modifique o método Game#raffle para ficar assim:

def

raffle(word_length)

@raffled_word = @word_raffler

.raffle(word_length)

if @raffled_word

@state = :word_raffled

```
end end
```

Ao rodarmos o RSpec, podemos ver que o teste passou, vamos para o próximo.

O próximo teste é para o cenário de quando o sorteio da palavra não tem sucesso. Nele, o game deve continuar no estado :initial. Especifique esse comportamento escrevendo o seguinte teste no arquivo spec/game_spec.rb:

```
describe "#raffle" do
```

```
# (...)
```

```
  it
```

```
    "stays on the :initial state "
```

```
  \
```

```
    "when a word can't be raffled" do
```

```
allow(word_raffler).to receive(  
:raffle).and_return(nil  
)
```

```
game.raffle(  
3  
)
```

```
expect(game.state).to eq(  
:initial  
)
```

```
end end
```

Ao rodarmos o RSpec, vemos que esse teste já está no verde, devido ao modo como implementamos a classe Game.

Antes de voltarmos para o teste pendente da classe GameFlow, vamos dar uma olhada no código da classe Game. Devido à nova propriedade de estado da Game, existe um método que vale a pena ser modificado. É o Game#finish:


```
def
```

```
  finish
```

```
@ended = true end
```

Esse método é relacionado com a manutenção do estado do game. Como agora temos uma variável de instância para controlar o estado do jogo como um todo, vale a pena a utilizarmos para controlar se o game está no estado final ou não. Para fazer isso, vamos modificar o método `Game#finish`, o método `Game#ended?` e deletar a inicialização da variável de instância `@ended` do construtor da classe `Game`.

O código da modificação dessas partes é o seguinte:

```
class Game
```

```
  # (...)
```

```
  def initialize(word_raffler = WordRaffler
```

```
    .new)
```

```
@word_raffler
```

```
= word_raffler
```

```
@state = :initial
```

```
end
```

```
# (...)
```

```
def
```

```
ended?
```

```
@state == :ended
```

end

def

finish

@state = :ended

end end

Ao rodar o RSpec, podemos perceber que os testes continuam no verde e concluímos que nossa refatoração não quebrou nada. Finalmente, podemos voltar para o teste que originou todas essas modificações na classe Game, que é o seguinte teste do GameFlow:

context "when the player guess a letter with success" do

xit

"prints a success message" do

```
allow(game).to receive(  
:raffled_word).and_return("hey"  
)
```

```
success_message =  
"Você adivinhou uma letra com sucesso."
```

```
expect(ui).to receive(  
:write  
) .with(success_message)
```

```
game.next_step
```

```
end end
```

15.2 Refatorando o fluxo do jogo para usar a máquina de estados

Antes de continuarmos a trabalhar no teste:

```
context "when the player guess a letter with success" do
```

```
 xit
```

```
  "prints a success message" do
```

```
    allow(game).to receive(
      :raffled_word).and_return("hey"
    )
```

```
    success_message =
```

```
    "Você adivinhou uma letra com sucesso."
```

```
    expect(ui).to receive(
      :write
```

```
).with(success_message)
```

```
game.next_step
```

```
end end
```

Vamos ver se o novo modo de controlar o estado do game não influencia nesse teste da classe GameFlow, já que o game é uma dependência do game_flow.

Esse teste cobre o cenário de adivinhação de letra com sucesso. Esse cenário só deve acontecer depois que a palavra do jogo já foi sorteada, ou seja, quando o estado do game for :word_raffled. Então precisamos setar esse estado no colaborador game. Podemos fazer isso do seguinte modo:

```
allow(game).to receive(:state).and_return(:word_raffled)
```

Além desse stub serão necessários mais alguns passos no setup desse teste. Como esse cenário envolve adivinhar uma letra com sucesso, precisamos decidir de onde virá essa informação. O "chute" da letra, input do jogador, virá do ui.read. Podemos fazer esse setup no teste da seguinte forma:

```
letter_guess = "e"
```

```
allow(ui).to receive(  
:read).and_return(letter_guess)
```

Por fim, precisamos fazer o setup sobre de onde vem a informação de que o chute da letra foi uma adivinhação com sucesso. Como quem guarda a palavra sorteada é o game, ele pode ser responsável por saber se uma dada letra existe na palavra sorteada ou não. Ou seja, ele será o responsável por dizer se a adivinhação da letra teve sucesso ou não.

Vamos imaginar um método chamado `game.guess_letter`, que recebe uma letra e, caso a adivinhação tenha sucesso, ele retorna `true`, caso contrário, retorna `false`. Como o cenário do teste que estamos escrevendo é sobre adivinhação de letra com sucesso, se esse método existisse, iríamos usá-lo para fazer o setup do nosso teste do seguinte modo:

```
allow(game).to receive(:guess_letter).and_return(true)
```

Juntando todos os passos de setup, nosso teste ficará assim:

```
context "when the player guess a letter with success" do
```

```
  it
```

```
    "prints a success message" do
```

```
    allow(game).to receive(
:state).and_return(:word_raffled
)

    allow(game).to receive(
:guess_letter).and_return(true
)

    success_message =
"Você adivinhou uma letra com sucesso."

    expect(ui).to receive(
:write
).with(success_message)

    game_flow.next_step

end end
```

Ao rodarmos o RSpec, vemos que esse teste quebra com a seguinte mensagem:

```
$ bundle exec rspec
```


(...)

Failures:

1) GameFlow#next_step when the player guess a letter with success
prints a success message

Failure/Error:

```
expect(ui).to receive(:write).with(success_message)
```

```
#<Double "ui"> received :write with unexpected arguments
```

```
expected: ("Você adivinhou uma letra com sucesso.")
```

```
got: ("Qual o tamanho da palavra a ser sorteada?") (1 time)
```

```
("") (1 time)
```

Pela mensagem de erro podemos ver que o problema foi que a mensagem de adivinhação de letra com sucesso não foi impressa. Vamos fazer esse teste passar.

Para isso, precisamos primeiro modificar a classe GameFlow para que ela tome suas ações baseada no novo atributo de estado do objeto game. O método que cuida do próximo passo do fluxo do jogo, GameFlow#next_step está assim:

```
def
```

```
    next_step
```

```
    @ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
    player_input =
```

```
    @ui
```

```
    .read.strip
```

```
    if player_input == "fim"
```

```
        @game
```

```
        .finish
```

```
    else
```

```
        if @game
```

```
.raffle(player_input.to_i)
  print_letters_feedback
```

```
else
```

```
  error_message =
```

```
    "Não temos uma palavra com o tamanho desejado,\n"
```

```
  \
```

```
    "é necessário escolher outro tamanho."
```

```
@ui
```

```
.write(error_message)
```

```
end
```

```
end end
```

O passo implementado agora é o primeiro passo do jogo e só deve ser realizado quando o jogo estiver no estado inicial. Vamos extrair a lógica apresentada para um método privado e só chamá-lo quando `@game.state` for igual a `:initial`:

```
def
```

```
  next_step
```

```
  case @game
```

```
    .state
```

```
      when :initial
```

```
        ask_to_raffle_a_word
```

```
      end end
```

```
  private
```

```
  # (...) def
```

```
ask_to_raffle_a_word
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@game
```

```
.finish
```

```
else
```

```
if @game
```

```
.raffle(player_input.to_i)
```

```
print_letters_feedback
```

```
else
```

```
    error_message =
```

```
        "Não temos uma palavra com o tamanho desejado,\n"
```

```
        \
```

```
        "é necessário escolher outro tamanho."
```

```
@ui
```

```
    .write(error_message)
```

```
end
```

```
end end
```

Ao rodarmos o RSpec depois dessa modificação, podemos ver que quase todos

os testes da classe GameFlow quebraram:

```
$ bundle exec rspec
```

```
(...)
```

16 examples, 6 failures

Failed examples:

```
rspec ./spec/game_flow_spec.rb:20
```

```
rspec ./spec/game_flow_spec.rb:32
```

```
rspec ./spec/game_flow_spec.rb:41
```

```
rspec ./spec/game_flow_spec.rb:52
```

```
rspec ./spec/game_flow_spec.rb:66
```

```
rspec ./spec/game_flow_spec.rb:77
```

Essa modificação quebrou esses testes porque neles não está sendo feito nenhum setup do estado da dependência game. Para todos os testes da classe GameFlow que tínhamos feito até então, é necessário que o game esteja no estado :initial. Então, todos esses testes compartilham um mesmo setup. Podemos colocá-lo no let que define o colaborador game no começo desse teste. Esse let está assim:

```
let(:game) { double("game").as_null_object }
```

Vamos adicionar o stub do state para retornar :initial, de modo a ficar assim:

```
let(:game) { double("game", state: :initial).as_null_object }
```

Ao rodar o RSpec depois dessa modificação, podemos verificar que todos os testes que tinham quebrado voltaram a passar. Ficamos somente com um teste quebrado, que é o teste em que estamos trabalhando:

```
context "when the player guess a letter with success" do
```

```
  it
```

```
    "prints a success message" do
```

```
      # (...)
```

```
    end end
```


Para fazermos esse teste passar, basta que o método `GameFlow#next_step` lide com o estado `:word_raffled` do objeto `game`. Primeiro vamos adicionar esse estado no método `Game#next_step` no arquivo `lib/game_flow.rb`:

```
def
  next_step

  case @game
    .state

  when :initial

    ask_to_raffle_a_word

  when :word_raffled

    ask_to_guess_a_letter

  end end
```

Nesse código, estamos chamando o método `ask_to_guess_a_letter` para executar o próximo passo quando o jogo estiver no estado `:word_raffled`. Agora falta

implementarmos esse método.

Como especificamos no nosso teste, o que esse método precisa fazer até agora é executar a adivinhação de letra do jogador e imprimir uma mensagem de sucesso se essa adivinhação for correta. Para fazer isso adicione o seguinte método privado na classe GameFlow:

```
private # (...) def
```

```
  ask_to_guess_a_letter
```

```
    letter =
```

```
      @ui
```

```
        .read.strip
```

```
      if @game
```

```
        .guess_letter(letter)
```

```
        @ui.write("Você adivinhou uma letra com sucesso."
```

```
      )
```

```
    end end
```

Ao rodar o RSpec, você pode ver que, depois dessa modificação, finalmente os testes voltaram a ficar 100% no verde! Vamos aproveitar que os testes estão no verde e reorganizar os testes da classe GameFlow.

15.3 Organizando seus testes otimizando para leitura

Ao ler o arquivo `spec/game_flow_spec.rb` você pode ver que o método `next_step` tem muitos testes:

GameFlow

`#next_step`

when the game just started

asks the player for the length of the word to be raffled

when the player asks to raffle a word

raffles a word with the given length

prints a '_' for each letter in the raffled word

tells if it's not possible to raffle with the given length

finishes the game when the player asks to

when the player guess a letter with success

prints a success message

Já que o comportamento do método `next_step` é fazer uma série de ações baseado no estado do game, vamos reorganizar esses testes para ficarem agrupados segundo o estado do game. Seguindo essa ideia, eles ficariam

organizados assim:

GameFlow

#next_step

when the game is in the 'initial' state

asks the player for the length of the word to be raffled

and the player asks to raffle a word

tells if it's not possible to raffle with the given

length prints a '_' for each letter in the raffled word

raffles a word with the given length

when the game is in the 'word raffled' state

and the player guess a letter with success

prints a success message

finishes the game when the player asks to

Para fazer essa reorganização, vamos começar pelos testes do grupo do estado inicial. Para fazer isso comece modificando no arquivo spec/game_flow_spec.rb onde antes era "when the game just started" para "when the game is in the 'initial' state":

describe "#next_step" do

context

"when the game is in the 'initial' state" do

it

"asks the player for the length "

\

"of the word to be raffled" do

(...)

end

end

Depois disso, coloque o trecho de código do context "when the player asks to raffle a word" dentro do context "when the game is in the 'initial' state":

```
describe "#next_step" do
```

```
  context
```

```
    "when the game is in the 'initial' state" do
```

```
      it
```

```
        "asks the player for the length "
```

```
        \
```

```
        "of the word to be raffled" do
```

```
          # (...)
```

```
        end
```

```
      context
```

```
        "when the player asks to raffle a word" do
```

it

"raffles a word with the given length" do

(...)

end

it

"prints a '_' for each letter in the raffled word" do

(...)

end

it

"tells if it's not possible to raffle "

\

"with the given length" do

(...)

end

end

end

E para finalizar essa parte do agrupamento do estado inicial, modifique o texto de onde era context "when the player asks to raffle a word" para context "and the player asks to raffle a word".

Agora vamos organizar os testes do agrupamento do estado word raffled. Crie um novo context para esse agrupamento e coloque o context "when the player guess a letter with success" dentro desse novo context, de modo a ficar assim:

```
context "when the game is in the 'word raffled' state" do
```

```
  context
```

```
    "and the player guess a letter with success" do
```

```
      it
```

```
        "prints a success message" do
```

```
          allow(game).to receive(
            :state).and_return(:word_raffled
          )
```

```
          allow(game).to receive(
            :guess_letter).and_return(true
          )
```

```
          success_message =
```

```
            "Você adivinhou uma letra com sucesso."
```

```
    expect(ui).to receive(
      :write
    ).with(success_message)

    game_flow.next_step

  end

end end
```

Ao rodar o RSpec, podemos ver que os testes continuam passando. Fizemos toda essa reorganização dos contexts para que a leitura dos nossos testes ficasse mais fácil e intuitiva. Para checar se a nova estrutura do nosso teste ficou boa, podemos rodar o RSpec com o output formatter documentation e ler o resultado. Faça isso executando o seguinte comando:

```
$ bundle exec rspec --format documentation
spec/game_flow_spec.rb
```

Você verá este output:

GameFlow

#start

prints the initial message

#next_step

finishes the game when the player asks to

when the game is in the 'initial' state

asks the player for the length of the word to be raffled

and the player asks to raffle a word

tells if it's not possible to raffle with the given length

prints a '_' for each letter in the raffled word

raffles a word with the given length

when the game is in the 'word raffled' state

and the player guess a letter with success

prints a success message

Finished in 0.00442 seconds (files took 0.23084 seconds to load)

7 examples, 0 failures

Figura 15.1: RSpec com output formatado para documentação

Pronto, com os testes organizados de modo a ser fácil de lê-los, podemos ir em frente. Vamos finalizar os testes de unidade necessários para a adivinhação de letra com sucesso.

15.4 Interface discovery utilizando test doubles

Antes de continuar o desenvolvimento do nosso jogo, vamos falar de uma técnica muito interessante chamada Interface discovery.

Interface discovery é uma técnica que consiste em descobrir (definir) a API (interface) de um objeto colaborador durante o processo de TDD utilizando Mock Objects. Ela foi definida inicialmente no paper Endo-testing: unit testing with mock objects (MACKINNON; FREEMAN; CRAIG, 2001), e depois mais explorada no paper Mock roles, not Objects (FREEMAN; MACKINNON; PRYCE; WALNES, 2004) e também no tradicional livro Growing Object-Oriented Software, Guided by Tests (FREEMAN; PRYCE, 2009).

Utilizamos uma variante dessa técnica na seção Refatorando o fluxo do jogo para usar a máquina de estados, quando, durante o processo de TDD do GameFlow, definimos que ele precisaria depender de um método `Game#guess_letter` que ainda não existia, para saber se o jogador tinha adivinhado uma letra com sucesso. Nesta seção, utilizaremos de novo essa variante, mas de modo mais explícito, de forma que seu uso fique claro.

Voltando ao desenvolvimento do nosso jogo, vamos relembrar a funcionalidade que estamos desenvolvendo olhando o cenário de Cucumber que está quebrando:

Cenário: Sucesso ao adivinhar letra

Se o jogador adivinhar a letra com sucesso, o jogo mostra uma mensagem de sucesso e mostra em que posição está a letra que o jogador adivinhou.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem

"3"

letras

Quando tento adivinhar que a palavra tem a letra

"a"

E termino o jogo

Então o jogo mostra que eu adivinhei uma letra com sucesso

E o jogo termina com a seguinte mensagem na tela:

""

a _ _

""

Para essa funcionalidade funcionar, o seguinte fluxo deve ser implementado:

O jogo deve estar no estado :word_raffled;

O jogo pede para o jogador adivinhar uma letra;

Jogador adivinha uma letra com sucesso;

Jogo mostra mensagem de sucesso;

Jogo mostra letras adivinhadas.

Já trabalhamos no primeiro, terceiro e quarto passos desse fluxo, ainda faltam o segundo e quinto.

Vamos escrever um teste de unidade para especificar que quando o jogo está no estado :word_raffled, o próximo passo do jogo é perguntar para o jogador qual letra ele acha que a palavra tem. Faça isso escrevendo o seguinte teste no arquivo spec/game_flow_spec.rb:

```
context "when the game is in the 'word raffled' state" do
```

```
  it
```

```
    "asks the player to guess a letter" do
```

```
      allow(game).to receive(
        :state).and_return(:word_raffled
      )
```

```
question =  
"Qual letra você acha que a palavra tem?"  
  
expect(ui).to receive(  
  :write  
) .with(question)  
  
game_flow.next_step  
  
end  
  
context  
"and the player guess a letter with success" do  
  
  # (...)  
  
end end
```

Rode os testes de RSpec e verifique que esse teste quebrou.

Para fazê-lo passar, basta fazermos com que o jogo faça essa pergunta quando ele estiver tratando o estado :word_raffled. Hoje já temos um método que está tratando esse estado, o método GameFlow#ask_to_guess_a_letter. Vamos modificá-lo para que ele faça a pergunta para o jogador:

```
def
```

```
ask_to_guess_a_letter
```

```
@ui.write("Qual letra você acha que a palavra tem?"
```

```
)
```

```
letter =
```

```
@ui
```

```
.read.strip
```

```
if @game
```

```
.guess_letter(letter)
```

```
@ui.write("Você adivinhou uma letra com sucesso."
```

```
)
```

```
end end
```

Ao rodar o RSpec agora, podemos ver que o teste passou. Com os testes no verde, podemos parar para refatorá-los um pouco.

Ao ler os testes dentro do context context "when the game is in the 'word raffled' state", podemos ver que todos eles estão setando o estado do game para :word_raffled na fase de setup do teste. Para reduzir essa duplicação, vamos extrair isso para um before block. Ao fazer isso seus testes ficarão assim:

```
context "when the game is in the 'word raffled' state" do
```

```
  before
```

```
  do
```

```
    allow(game).to receive(
```

```
    :state).and_return(:word_raffled
```

```
  )
```

```
end
```

it

"asks the player to guess a letter" do

question =

"Qual letra você acha que a palavra tem?"

expect(ui).to receive(

:write

).with(question)

game_flow.next_step

end

context

"and the player guess a letter with success" do

it

```
"prints a success message" do
```

```
    allow(game).to receive(  
      :guess_letter).and_return(true  
    )
```

```
    success_message =
```

```
    "Você adivinhou uma letra com sucesso."
```

```
    expect(ui).to receive(  
      :write  
    ).with(success_message)
```

```
    game_flow.next_step
```

```
end
```

```
end end
```

Usando Interface discovery

A seguir, vamos utilizar a técnica de Interface discovery, mas diferentemente de como fizemos na seção Refatorando o fluxo do jogo para usar a máquina de estados, vamos deixar explícito que a estamos utilizando, para você entender melhor o uso dessa técnica na prática.

O próximo comportamento a ser especificado é o de que, quando o jogador adivinhar uma letra com sucesso, o jogo deve mostrar pra ele as letras adivinhadas.

Para que o GameFlow imprima as letras adivinhadas, essa informação precisa estar salva em algum lugar e precisamos definir qual método a retornará. Como o responsável por todo o estado do jogo é o objeto Game, então vamos imaginar que é esse objeto que terá um método que retornará um array das letras adivinhadas. Vamos chamá-lo de `Game#guessed_letters`.

É nesse momento que estamos descobrindo uma parte nova da interface do objeto game. O objeto game é um colaborador do `game_flow`, e durante o desenvolvimento de um novo comportamento do `game_flow`, descobrimos que ele vai depender de um novo método do game, chamado `guessed_letters`. Apesar de termos descoberto essa nova dependência agora, não precisamos implementar esse método neste momento, porque para os testes da classe `GameFlow` podemos representar essa dependência usando o `test double game`.

Essa é a técnica de Interface discovery, descobrir um novo método que vai fazer parte da API de um objeto colaborador durante o processo de TDD e representar essa dependência nos testes utilizando um `test double`.

Dada essa definição de que vai existir um método chamado `Game#guessed_letters`, podemos escrever nosso próximo teste fazendo o stub do seguinte modo:

```
context "and the player guess a letter with success" do
```

```
# (...)
```

```
  it
```

```
    "prints the guessed letters" do
```

```
      allow(game).to receive(
        :guess_letter).and_return(true
      )

      allow(game).to receive(
        :raffled_word).and_return("hey"
      )

      allow(game).to receive(
        :guessed_letters).and_return(["e"
```



```
)
```

```
    expect(ui).to receive(  
:write).with("_ e _"  
)
```

```
    game_flow.next_step
```

```
end end
```

Ao rodar o RSpec, vemos a seguinte saída:

```
$ bundle exec rspec
```

```
1) GameFlow#next_step when the game is in the 'word raffled'  
state and the player guess a letter with success prints the  
guessed letters
```

```
Failure/Error: expect(ui).to receive(:write).with("_ e _")
```

```
#<Double "ui"> received :write with unexpected arguments  
expected: ("_ e _")
```

got: ("Qual letra você acha que a palavra tem?") (1 time)

("Você adivinhou uma letra com sucesso.") (1 time)

A mensagem diz que o teste quebrou porque o ui não recebeu a mensagem write com o argumento "_" e "_", ou seja, quebrou porque nosso jogo não imprimiu as letras adivinhadas.

Para fazer o teste passar, basta que nosso jogo imprima as letras adivinhadas quando o jogador adivinhar uma letra com sucesso. Vamos fazer isso adicionando @ui.write(guessed_letters) no método que trata adivinhação de letra, o GameFlow#ask_to_guess_a_letter:

def

ask_to_guess_a_letter

@ui.write("Qual letra você acha que a palavra tem?"

)

letter =

@ui

.read.strip

```
if @game
```

```
.guess_letter(letter)
```

```
@ui.write("Você adivinhou uma letra com sucesso."
```

```
)
```

```
@ui
```

```
.write(guessed_letters)
```

```
end end
```

Agora precisamos implementar o método `guessed_letters`, que é o argumento sendo passado para `@ui.write`. Ele vai retornar uma string representando as letras que foram adivinhadas até então. No caso do nosso teste, ele deve retornar `"_ e _"`.

Para implementar esse método, podemos iterar sobre cada letra da palavra sorteada e checar se essa letra é uma letra que já foi adivinhada. Se já foi, devemos colocá-la na string de retorno. Se não, devemos colocar um `"_"` na string. Implemente esse método do seguinte modo no arquivo `lib/game_flow.rb`:

```
private # (...) def
```

```
  guessed_letters
```

```
letters =  
@game.raffled_word.chars.map do  
  |letter|  
  
  @game.guessed_letters.include?(letter) ? letter : "_"  
  
end  
  
letters.join(  
  " ") end
```

Ao rodarmos o RSpec agora, podemos ver que o teste passou! Vamos checar se podemos refatorar algo no nosso código ou teste.

Nos testes sobre quando o jogador adivinha uma letra com sucesso, faz parte do setup fazer o stub `allow(game).to receive(:guess_letter).and_return(true)` para setar a adivinhação com sucesso. Esse setup está duplicado entre esses testes:

```
context "and the player guess a letter with success" do
```

it

"prints a success message" do

allow(game).to receive(
:guess_letter).and_return(true
)

(...)

end

it

"prints the guessed letters" do

allow(game).to receive(
:guess_letter).and_return(true
)

allow(game).to receive(
:raffled_word).and_return("hey"

```
)  
  allow(game).to receive(  
:guessed_letters).and_return(["e"  
])
```

```
# (...)
```

```
end end
```

Vamos extrair esse setup para um before block desse context. Modifique esses testes para ficarem assim:

```
context "and the player guess a letter with success" do
```

```
  before
```

```
do
```

```
  allow(game).to receive(  
:guess_letter).and_return(true  
)
```

end

it

"prints a success message" do

(...)

end

it

"prints the guessed letters" do

allow(game).to receive(
:raffled_word).and_return("hey"
)

allow(game).to receive(

```
:guessed_letters).and_return(["e"  
])
```

```
# (...)
```

```
end end
```

Ao rodar novamente o RSpec, vemos que os testes continuam no verde, logo nossa refatoração foi ok e não quebrou nada. Além dessa refatoração no teste, podemos refatorar também o código de produção.

Se você checar os métodos `GameFlow#print_letters_feedback` e `GameFlow#guessed_letters`, verá que o segundo é, na verdade, praticamente uma generalização do primeiro. Logo, onde antes usávamos `print_letters_feedback`, podemos agora usar o `guessed_letters`, reduzindo a duplicação de comportamento. O único lugar onde está sendo usado o `print_letters_feedback` é no seguinte trecho do método `ask_to_raffle_a_word`:

```
def
```

```
ask_to_raffle_a_word
```

```
# (...)
```



```
if player_input == "fim"
```

```
# (...)
```

```
else
```

```
if @game
```

```
.raffle(player_input.to_i)
```

```
  print_letters_feedback
```

```
else
```

```
# (...)
```

```
end
```

```
end end
```

Para utilizar o novo `guessed_letters`, podemos modificar esse trecho para ficar desta forma:

```
def
```

```
ask_to_raffle_a_word
```

```
# (...)
```

```
if player_input == "fim"
```

```
# (...)
```

else

if @game

.raffle(player_input.to_i)

@ui

.write(guessed_letters)

else

(...)

end

end end

Faça essa modificação e delete o antigo método

GameFlow#print_letters_feedback. Ao fazer isso, o teste it "prints a '_' for each letter in the raffled word" quebra. Isso porque, agora que o comportamento especificado nesse teste também depende do método Game#guessed_letters, é necessário que façamos um stub desse método no let que define o double game. Esse let está assim:

```
let(:game) { double("game", state: :initial).as_null_object }
```

Vamos modificá-lo:

```
let(:game) do
```

```
  double(
```

```
    "game"
```

```
    ,
```

```
    state: :initial
```

```
    ,
```

```
    guessed_letters:
```

```
    []
```

```
).as_null_object
```

```
end
```

Ao rodar o RSpec, vemos que está tudo no verde novamente, logo nossa refatoração teve sucesso. Vamos voltar ao cenário de adivinhação de letra com sucesso como um todo.

Os testes que fizemos do GameFlow para especificar o cenário de adivinhação de letra com sucesso foram:

```
context "when the game is in the 'word raffled' state" do
```

```
  it
```

```
    "asks the player to guess a letter" do
```

```
      context
```

```
        "and the player guess a letter with success" do
```

```
          it
```

```
            "prints a success message" do
```

it

"prints the guessed letters"

end end

Já fizemos todos os testes de unidade necessários do GameFlow para cobrir o cenário de adivinhação de letra com sucesso. No entanto, a funcionalidade ainda não está pronta. Para verificarmos isso, podemos rodar o Cucumber e ver que ele ainda tem testes no vermelho:

```
$ bundle exec cucumber
```

```
(...)
```

```
game_flow.rb:63:in `block in guessed_letters':
```

```
undefined method `guessed_letters' for #<Game:0x00007fc1201323>
```

```
(NoMethodError)
```

```
4 scenarios (2 failed, 2 passed)
```

Como você pode ver na mensagem de erro do Cucumber, o teste está quebrando porque o método `Game#guess_letter` ainda não foi implementado. Isso porque até então não tínhamos feito o stub da sua dependência nos testes de unidade do

GameFlow, mas agora precisamos fazer uma implementação concreta desse método.

Implementando métodos descobertos via interface discovery

Percebemos que nossos testes de Cucumber estão quebrados porque falta implementar o método `Game#guess_letter`. Ele foi definido durante os testes de unidade do GameFlow utilizando a técnica de Interface discovery. Além desse método, precisamos implementar também o `Game.guessed_letters`. Vamos começar especificando o `Game#guess_letter`.

O método `Game#guess_letter` receberá como argumento uma letra. Se a palavra sorteada contiver essa letra, ele retorna, `true`, caso contrário, retorna `false`. Escreva um teste para especificar esse comportamento no arquivo `spec/game_spec.rb` do seguinte modo:

```
describe "#guess_letter" do
```

```
  it
```

```
    "returns true if the raffled word "
```

```
  \
```

```
    "contains the given letter" do
```

```
game.raffled_word =  
"hey"  
  
expect(game.guess_letter(  
"h"  
)).to be_truthy  
  
end end
```

Ao rodar esse teste, vemos que ele quebra porque o método `Game#guess_letter` ainda não foi definido. Para fazer o teste passar, implemente esse método do seguinte modo no arquivo `lib/game.rb`:

```
def  
  guess_letter(letter)  
  
    @raffled_word.include?(letter) end
```

Rodando o teste, podemos ver que ele passa. Vamos para o próximo. Agora precisamos escrever um teste que verifica que o método `guess_letter` deve retornar `false`, caso a palavra sorteada não contenha a letra dada como argumento. Escreva-o:


```
describe "#guess_letter" do
```

```
  it
```

```
    "returns true if the raffled word "
```

```
    \
```

```
    "contains the given letter" do
```

```
      # (...)
```

```
    end
```

```
  it
```

```
    "returns false if the raffled word "
```

```
    \
```

```
    "doesn't contain the given letter" do
```

```
game.raffled_word =  
"hey"  
  
expect(game.guess_letter(  
"z"  
)).to be_falsey  
  
end end
```

Ao rodá-lo, vemos que ele já está no verde. Isso é porque a implementação que fizemos desse método já cobria esse cenário de insucesso também.

Por último, vamos escrever um teste de sanidade para esse método. Vamos especificar o que acontece se ele recebe uma string vazia ou só com espaços em branco. Nesse caso, o método deve retornar false também. Escreva o seguinte teste para especificar esse comportamento:

```
describe "#guess_letter" do  
  
  it
```

"returns true if the raffled word "

\

"contains the given letter"

it

"returns false if the raffled word "

\

"doesn't contain the given letter"

it

"returns false if the given letter is a blank string" do

game.raffled_word =

"hey"

expect(game.guess_letter(

```
"""
```

```
)).to be_falsey
```

```
    expect(game.guess_letter(
```

```
" "
```

```
)).to be_falsey
```

```
end end
```

Ao rodarmos esse teste, vemos que ele quebra. Ele quebrou porque o que a implementação do nosso método faz é:

```
def
```

```
  guess_letter(letter)
```

```
  @raffled_word.include?(letter) end
```

Acontece que `@raffled_word.include?("")` retorna true. Por isso, nosso teste quebrou. Vamos alterar nosso método para retornar false caso a string dada como argumento seja vazia ou só tenha espaços em branco. Modifique nosso método para ficar assim:

```
def
```

```
guess_letter(letter)
```

```
return false if
```

```
letter.strip.empty?
```

```
@raffled_word.include?(letter) end
```

Ao rodar o RSpec agora, vemos que os testes passam. Como não há nada para refatorar até agora, podemos ir para o próximo teste, que é o do método `Game.guessed_letters`.

Esse método deve retornar um array com as letras adivinhadas até então. Portanto, se o jogo tem como palavra sorteada a palavra "hey", e a letra "e" já foi adivinhada com sucesso, o método `Game#guessed_letters` deve retornar o array ["e"]. Escreva um teste para especificar esse comportamento no arquivo `spec/game_spec.rb`:

```
describe "#guessed_letters" do
```

```
  it
```

```
    "returns the guessed letters" do
```

```
game.raffled_word =  
"hey"  
  
game.guess_letter(  
"e"  
)  
  
expect(game.guessed_letters).to eq([  
"e"  
)  
  
end end
```

Ao rodar esse teste, vemos que ele quebra porque o método `guessed_letters` ainda não foi definido. Vamos defini-lo.

Vamos imaginar que as letras adivinhadas fiquem salvas em uma variável de instância chamada `@guessed_letters`. Bastaria que o método `guessed_letters` fosse definido como um `attr_reader` na classe `Game`. Para fazer isso, adicione esse `attr_reader` no começo do arquivo `lib/game.rb`:

```
class Game
```

```
attr_accessor :raffled_word
```

```
attr_accessor :state
```

```
attr_reader :guessed_letters
```

```
# (...) end
```

Precisamos agora inicializar essa variável. Quando um novo jogo é criado, ela deve vir com um array vazio. Escreva um teste para isso do seguinte modo:

```
describe "#guessed_letters" do
```

```
  it
```

```
    "returns the guessed letters" do
```

```
# (...)
```

```
end
```

```
it
```

```
"returns an empty array when there's no guessed letters" do
```

```
  expect(game.guessed_letters).to eq([])
```

```
end end
```

Para fazer esse teste passar, basta inicializarmos essa variável com um array vazio no construtor da classe Game:

```
def initialize(word_raffler = WordRaffler  
  .new)
```



```
@word_raffler
```

```
= word_raffler
```

```
@state = :initial
```

```
@guessed_letters = [] end
```

Ao rodarmos o RSpec agora, vemos que um teste passou mas um desses novos testes quebrou, com a seguinte mensagem:

```
$ bundle exec rspec
```

Failures:

1) Game#guessed_letters returns the guessed letters

Failure/Error: expect(game.guessed_letters).to eq(["e"])

expected: ["e"]

got: [] (using ==)

Para que o método guessed_letters retorne as letras adivinhadas, é necessário que

alguém atualize o array `@guessed_letters`. Isso pode ser feito pelo método `Game#guess_letter`, quando a adivinhação da letra tiver sucesso. Vamos especificar esse novo comportamento do método `Game#guess_letter`. Adicione este teste no arquivo `spec/game_spec.rb`:

```
describe "#guess_letter" do
```

```
  it
```

```
    "returns true if the raffled word "
```

```
    \
```

```
    "contains the given letter" do
```

```
      # (...)
```

```
    end
```

```
  it
```

```
    "saves the guessed letter when the guess is right" do
```

```
game.raffled_word =
```

```
"hey"
```

```
expect { game.guess_letter(
```

```
"h"
```

```
) }
```

```
.to change { game.guessed_letters }.from([]).to([
```

```
"h"
```

```
])
```

```
end
```

```
# (...) end
```

Ao rodar o RSpec, vemos que esse teste quebra. Para fazê-lo passar, basta alterarmos o método `Game#guess_letter` para salvar a letra adivinhada com sucesso:

```
def
```

```
  guess_letter(letter)
```

```
  return false if
```

```
    letter.strip.empty?
```

```
  if @raffled_word.include
```

```
    ?(letter)
```

```
  @guessed_letters
```

```
    << letter
```

```
  true
```

```
else
```

```
  false
```

```
end end
```

Ao rodarmos o RSpec agora, vemos que todos os testes estão no verde! Falta fazermos somente mais um teste para o método `Game#guess_letter`, que é o cenário onde o jogador adivinha com sucesso a mesma letra mais de uma vez. Nesse cenário, a variável `@guessed_letters` não deve salvar a letra adivinhada de modo repetido, mas sim uma única vez. Especifique esse comportamento escrevendo o seguinte teste no arquivo `spec/game_spec.rb`:

```
describe "#guess_letter" do
```

```
# (...)
```

```
  it
```

```
    "saves the guessed letter when the guess is right" do
```

```
# (...)
```

end

it

"does not save a guessed letter more than once" do

game.raffled_word =

"hey"

game.guess_letter(

"h"

)

expect { game.guess_letter(

"h"

) }

.to_not change { game.guessed_letters }.from([

"h"

])

end

(...)

Ao rodar o RSpec, vemos que esse teste quebra com a seguinte mensagem:

```
$ bundle exec rspec
```

Failures:

1) Game#guess_letter does not save a guessed letter
more than once

Failure/Error:

```
expect { game.guess_letter("h") }  
  .to_not change { game.guessed_letters }.from(["h"])
```

expected `game.guessed_letters` not to have changed,

but did change from ["h"] to ["h", "h"]

O problema que aconteceu é que a letra adivinhada com sucesso está sendo salva mais de uma vez no array @guessed_letters. Altere o método Game#guessed_letters para não duplicar uma letra adivinhada com sucesso fazendo um uniq! na @guessed_letters:

```
def
```

```
  guess_letter(letter)
```

```
  return false if
```

```
    letter.strip.empty?
```

```
    if @raffled_word.include
```

```
      ?(letter)
```

```
      @guessed_letters
```

```
      << letter
```

```
    @guessed_letters
```



```
.uniq!
```

```
true
```

```
else
```

```
false
```

```
end end
```

Agora ao rodar o RSpec, podemos ver que está tudo no verde! Vamos rodar o Cucumber.

Ao rodar o Cucumber, você pode ver que ainda tem teste quebrado. Esse problema está acontecendo porque quando o jogo está no estado :word_raffled ele não está lidando com o input "fim", que é quando o jogador quer finalizar o jogo no meio — comportamento que estamos utilizando nos testes do Cucumber.

Atualmente já temos um teste no spec/game_flow_spec.rb para lidar com o caso de o jogador querer finalizar o jogo no meio:

```
describe "#next_step" do
```

```
# (...)
```

```
it
```

```
"finishes the game when the player asks to" do
```

```
  player_input =
```

```
    "fim"
```

```
    allow(ui).to receive(
```

```
      :read
```

```
    ).and_return(player_input)
```

```
    expect(game).to receive(
```

```
      :finish
```

```
    )
```

```
game_flow.next_step
```

```
end end
```

No entanto, esse teste só está sendo aplicado para quando o game está no estado :initial. Precisamos fazer com que esse teste verifique esse comportamento enquanto o game estiver também no estado :word_raffled. Para fazer isso, modifique esse teste para ficar assim:

```
it "finishes the game when the player asks to" do
```

```
  player_input =
```

```
    "fim"
```

```
    allow(ui).to receive(
```

```
      :read
```

```
    ).and_return(player_input)
```

```
    allow(game).to receive(
```

```
      :state).and_return(:initial
```

```
    )
```

```
    expect(game).to receive(
      :finish
    )

    game_flow.next_step

    allow(game).to receive(
      :state).and_return(:word_raffled
    )

    expect(game).to receive(
      :finish
    )

    game_flow.next_step
  end
```

Ao rodar o RSpec, podemos ver que ele quebra com a seguinte mensagem:

```
$ bundle exec rspec
```

Failures:

1) GameFlow#next_step finishes the game when the player asks to

Failure/Error: expect(game).to receive(:finish)

```
(Double "game").finish(*(any args))
```

expected: 1 time with any arguments

received: 0 times with any arguments

O erro ocorreu porque se o jogador digitar "fim", e o estado do game for :word_raffled, a mensagem finish não está sendo enviada para game quando o método GameFlow#next_step é executado.

Antes de fazer esse teste passar, vamos primeiro checar como que o input "fim" está sendo tratado no caso de o game estar no estado :initial. Esse tratamento está sendo feito no método GameFlow#ask_to_raffle_a_word:

```
def
```

```
ask_to_raffle_a_word
```

```
@ui.write("Qual o tamanho da palavra a ser sorteada?"
```

```
)
```

```
player_input =
```

```
@ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@game
```

```
.finish
```

```
else
```

```
# (...)
```

```
end end
```

Para tratar o input "fim" no método `GameFlow#ask_to_guess_a_letter`, que é o método usado quando o game está no estado `:word_raffled`, seria necessário duplicar esta parte do método `ask_to_raffle_a_word`:

```
player_input = @ui
```

```
.read.strip
```

```
if player_input == "fim"
```

```
@game.finish # (...)
```

Em vez de fazer isso, vamos fazer uma refatoração para evitar essa duplicação. Vamos extrair esse comportamento de fazer uma pergunta para o jogador e tratar o input "fim" para um outro método privado, que poderá ser reutilizado por `ask_to_raffle_a_word` e `ask_to_guess_a_letter`. Como vamos fazer uma refatoração e temos um teste quebrado, marque esse teste quebrado como pendente, utilizando o `xit` do `RSpec`, em vez do `it`:

```
xit "finishes the game when the player asks to" do
```

```
# (...) end
```

Agora vamos extrair, do método `ask_to_raffle_a_word` para um novo método chamado `ask_the_player`, a lógica de fazer uma pergunta para o jogador e tratar o input "fim":

```
def
```

```
ask_to_raffle_a_word
```

```
question =
```

```
"Qual o tamanho da palavra a ser sorteada?"
```

```
ask_the_player(question)
```

```
do
```

```
|length|
```

```
if @game
```

```
.raffle(length.to_i)
```

```
@ui
```

```
.write(guessed_letters)
```

```
else
```

```
error_message =
```

```
"Não temos uma palavra com o tamanho desejado,\n"
```

```
\
```


"é necessário escolher outro tamanho."

@ui

.write(error_message)

end

end end

O que o método `ask_the_player` vai fazer é imprimir uma pergunta para o jogador e tratar o input de resposta. Quando o input for diferente de "fim", esse método dá um `yield` do input do jogador. Caso contrário, ele finaliza o jogo. Implemente esse método do seguinte modo no arquivo `lib/game_flow.rb`:

private # (...)

def

ask_the_player(question)

@ui

.write(question)

player_input =

@ui

.read.strip

if player_input == "fim"

@game

.finish

else

yield

player_input.strip

end end

Ao rodar o RSpec, vemos que não quebramos nenhum teste que já estava passando, logo nossa refatoração teve sucesso. Vamos agora reativar o teste que tínhamos marcado como pendente:

```
it "finishes the game when the player asks to" do
```

```
  player_input =
```

```
    "fim"
```

```
    allow(ui).to receive(
```

```
      :read
```

```
    ).and_return(player_input)
```

```
    allow(game).to receive(
```

```
      :state).and_return(:initial
```

```
    )
```

```
    expect(game).to receive(
```

```
      :finish
```

```
    )
```

```
    game_flow.next_step
```

```
allow(game).to receive(
:state).and_return(:word_raffled
)

expect(game).to receive(
:finish
)

game_flow.next_step

end
```

Ao rodar o RSpec, vemos que esse teste continua quebrando. Para fazê-lo passar, basta que usemos o novo método `ask_the_player` no método `ask_to_guess_a_letter` para que ele lide com input "fim" também. Modifique o método `ask_to_guess_a_letter` para ficar assim:

```
def

ask_to_guess_a_letter

question =

"Qual letra você acha que a palavra tem?"

ask_the_player(question)

do
```

```
|letter|
```

```
if @game
```

```
.guess_letter(letter)
```

```
@ui.write("Você adivinhou uma letra com sucesso."
```

```
)
```

```
@ui
```

```
.write(guessed_letters)
```

```
end
```

```
end end
```

Ao rodar o RSpec novamente, vemos que ele está no verde. E o melhor, ao rodarmos o Cucumber agora, vemos que ele também está no verde!

```
$ bundle exec cucumber
```

```
(...)
```

4 scenarios (4 passed)

18 steps (18 passed)

Finalmente o cenário do Cucumber Sucesso ao adivinhar letra está no verde!
Como ele já está no verde, retire a tag @wip dele.

Agora estamos prontos para ir para os próximos cenários da funcionalidade Adivinhar letra e finalmente finalizá-la.

15.5 Finalizando a funcionalidade Adivinhar letra

Para a funcionalidade Adivinhar letra definimos os seguintes cenários com Cucumber:

Sucesso ao adivinhar letra;

Erro ao adivinhar letra;

Jogador adivinha com sucesso duas vezes;

Jogador erra três vezes ao adivinhar letra.

O primeiro cenário nós já implementamos, está no verde. Vamos para o segundo, o Erro ao adivinhar letra.

Para esse cenário de erro, tínhamos definido a seguinte especificação:

Cenário: Erro ao adivinhar letra

Se o jogador errar ao tentar adivinhar a letra, o jogo mostra uma mensagem de erro e mostra quais as partes o boneco da forca já perdeu.

O que precisamos fazer agora é escrever o teste de aceitação para esse cenário. Sua estrutura é muito semelhante com a do cenário de adivinhação de letra com sucesso. As únicas diferenças são que o jogador tem que tentar adivinhar uma letra que não tem na palavra, o jogo deve mostrar que ele errou e deve mostrar quais são as partes que o boneco da forca já perdeu. Seguindo essa ideia, escreva esse teste do seguinte modo:

@wip

Cenário: Erro ao adivinhar letra

Se o jogador errar ao tentar adivinhar a letra, o jogo mostra uma mensagem de erro e mostra quais as partes o boneco da forca já perdeu.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem

"3"

letras

Quando tento adivinhar que a palavra tem a letra

"z"

E termino o jogo

Então o jogo mostra que eu errei a adivinhação da letra

E o jogo termina com a seguinte mensagem na tela:

```
.....
```

O boneco da força perdeu as seguintes partes do corpo: cabeça

```
.....
```

Ao rodar o Cucumber, nós vemos o seguinte:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 undefined)
```

```
7 steps (1 skipped, 1 undefined, 5 passed)
```

```
0m0.039s
```

You can implement step definitions for undefined steps

with these snippets:

```
Então("o jogo mostra que eu errei a adivinhação da letra") do
```

```
  pending # Write code here that turns the phrase above
```

```
# into concrete actions  
  
end
```

Como você pode ver pela saída do Cucumber, o step Então o jogo mostra que eu errei a adivinhação da letra está indefinido. O que esse step precisa fazer é apenas checar se a mensagem "Você errou a letra." foi impressa na tela. Defina esse step no arquivo `features/step_definitions/game_steps.rb`:

Então "o jogo mostra que eu errei a adivinhação da letra" do

```
output = last_command_stopped.stdout  
  
expect(output).to  
include("Você errou a letra.") end
```

Ao rodar o Cucumber agora, vemos que ele falha com a seguinte mensagem:

```
$ bundle exec cucumber -t @wip  
  
(...)
```

Diff:

```
@@ -1,2 +1,6 @@
```

-Você errou a letra.

+Bem-vindo ao jogo da forca!

+Qual o tamanho da palavra a ser sorteada?

+ _ _ _

+Qual letra você acha que a palavra tem?

+Qual letra você acha que a palavra tem?

1 scenario (1 failed)

7 steps (1 failed, 1 skipped, 5 passed)

Ele falhou porque a mensagem "Você errou a letra." não foi impressa na tela. Precisamos fazer com que nosso jogo lide com o cenário de adivinhação com erro. Para isso, vamos escrever um teste de unidade para o GameFlow.

Precisamos fazer um teste que verifique que, dado que o jogo está no estado :word_raffled, quando o jogador erra ao adivinhar uma letra, então a mensagem de erro é impressa. Escreva esse teste do seguinte modo no arquivo spec/game_flow_spec.rb:

describe "#next_step" do

(...)

context

"when the game is in the 'word raffled' state" do

before

do

allow(game).to receive(

:state).and_return(:word_raffled

)

end

(...)

context

"and the player fails to guess a letter" do

before

do

```
    allow(game).to receive(  
      :guess_letter).and_return(false  
    )
```

end

it

"prints an error message" do

```
    error_message =  
    "Você errou a letra."
```

```
    expect(ui).to receive(  
      :write  
    ).with(error_message)
```

```
    game_flow.next_step
```

```
end
```

```
end
```

```
end end
```

Rode o RSpec e verifique que o teste quebrou. Para fazer esse teste passar, basta que alteremos o método `GameFlow#ask_to_guess_a_letter` para lidar com o caso quando o método `Game#guess_letter` retorna `false`. Altere esse método para lidar com esse caso de modo que ele fique assim:

```
def
```

```
  ask_to_guess_a_letter
```

```
    question =
```

```
      "Qual letra você acha que a palavra tem?"
```

```
      ask_the_player(question)
```

```
  do
```

```
    |letter|
```

```
if @game
```

```
.guess_letter(letter)
```

```
@ui.write("Você adivinhou uma letra com sucesso."
```

```
)
```

```
@ui
```

```
.write(guessed_letters)
```

```
else
```

```
@ui.write("Você errou a letra."
```

```
)
```

```
end
```

```
end end
```

Ao rodar o RSpec agora, podemos ver que ele está no verde. Vamos rodar o

Cucumber:

```
$ bundle exec cucumber -t @wip
```

(...)

E o jogo termina com a seguinte mensagem na tela:

""""

O boneco da forca perdeu as seguintes partes do corpo: cabeça

""""

(...)

Diff:

```
@@ -1,2 +1,7 @@
```

-O boneco da forca perdeu as seguintes partes do corpo: cabeça

+Bem-vindo ao jogo da forca!

+Qual o tamanho da palavra a ser sorteada?

+ _ _ _

+Qual letra você acha que a palavra tem?

+Você errou a letra.

+Qual letra você acha que a palavra tem?

1 scenario (1 failed)

7 steps (1 failed, 6 passed)

O step que antes estava quebrando agora está passando, mas o step seguinte quebrou. O step que quebrou foi o que verifica se o jogo imprimiu na tela as partes que o boneco da forca perdeu. Para implementar esse comportamento, vamos antes especificá-lo com testes de unidade da classe GameFlow.

Esse teste deve verificar que, quando o jogador faz uma adivinhação de letra incorreta, o jogo imprime a lista das partes que o boneco da forca já perdeu. Vamos começar implementando esse teste do seguinte modo:

```
context "and the player fails to guess a letter" do
```

```
  before
```

```
do
```

```
  allow(game).to receive(
```

```
:guess_letter).and_return(false
```

```
)
```

end

(...)

it

"prints the list of the missed parts" do

missed_parts_message =

"O boneco da forca perdeu as "

\

"seguintes partes do corpo: cabeça"

expect(ui).to receive(

:write

).with(missed_parts_message)

game_flow.next_step

end end

A estrutura do teste está completa, porém, nem todas as informações necessárias para o setup do teste estão definidas. De onde vem a informação de que a parte que o boneco perdeu até então foi a cabeça?

Podemos imaginar que o objeto game poderá guardar essa informação como parte do estado do jogo. Desse modo, o objeto game poderia ter um método chamado `Game#missed_parts` que retornaria um array com as partes que o boneco já perdeu. Supondo que esse método existirá, podemos fazer stub dele no nosso teste:

it "prints the list of the missed parts" do

```
  allow(game).to receive(
    :missed_parts).and_return(["cabeça"]
  )
```

```
  missed_parts_message =
```

```
    "O boneco da forca perdeu as "
```

```
  \
```

"seguintes partes do corpo: cabeça"

```
expect(ui).to receive(  
  :write  
) .with(missed_parts_message)
```

```
game_flow.next_step  
end
```

Agora nosso teste deve estar completo. Rode o RSpec e verifique que esse teste quebrou. Para fazê-lo passar, basta modificarmos o método `GameFlow#ask_to_guess_a_letter` para utilizar o método `Game#missed_parts`:

```
def  
  ask_to_guess_a_letter  
    question =  
      "Qual letra você acha que a palavra tem?"  
  
    ask_the_player(question)  
  do  
    |letter|
```

```
if @game
```

```
.guess_letter(letter)
```

```
# (...)
```

```
else
```

```
@ui.write("Você errou a letra."
```

```
)
```

```
missed_parts_message =
```

```
"O boneco da forca perdeu as "
```

```
\
```

```
"seguintes partes do corpo: "
```

```
missed_parts_message <<
```

```
@game.missed_parts.join(", "
```

```
)
```

```
@ui
```

```
.write(missed_parts_message)
```

```
end
```

```
end end
```

Ao rodar o RSpec, podemos ver que está tudo no verde. No entanto, antes de rodar o Cucumber, precisamos implementar esse novo método que descobrimos que tem que fazer parte da API do objeto Game, o método Game#missed_parts.

Para começarmos a especificá-lo, adicione a seguinte estrutura de testes no arquivo spec/game_spec.rb:

```
describe "#missed_parts" do
```

```
  it
```

```
    "returns an empty array when there's no missed parts"
```

```
it
```

```
"returns the missed parts "
```

```
\
```

```
"for each fail in guessing a letter" end
```

O primeiro teste é para verificar o comportamento desse método quando o jogo acabou de começar. O segundo teste é para testar o comportamento desse método quando já aconteceram alguns insucessos na adivinhação de letra. Vamos escrever o primeiro teste:

```
it "returns an empty array when there's no missed parts" do
```

```
  expect(game.missed_parts).to eq([])
```

```
end
```

Ao rodarmos esse teste, vemos que ele falha porque o método `missed_parts` ainda não foi definido. Vamos defini-lo colocando um `attr_reader` para o atributo `missed_parts` e inicializando esse atributo no construtor da classe `Game`:

```
class Game
```

```
# (...)
```

```
attr_reader :missed_parts
```

```
def initialize(word_raffler = WordRaffler  
.new)
```

```
@word_raffler  
= word_raffler
```

```
@state = :initial
```

```
@guessed_letters  
= []
```

```
@missed_parts
```



```
= []
```

```
end
```

Ao rodar o RSpec, vemos que ele passou. Vamos para o próximo teste.

O próximo teste é para exercitar que, se algumas tentativas de adivinhação de letra já foram feitas sem sucesso, o método `missed_parts` deve retornar uma parte do boneco para cada tentativa errada. Escreva esse teste do seguinte modo:

```
it "returns the missed parts "
```

```
\
```

```
"for each fail in guessing a letter" do
```

```
  game.raffled_word =
```

```
    "hey"
```

```
  3.times do
```

```
game.guess_letter(  
  "z"  
)
```

end

```
missed_parts = [  
  "cabeça", "corpo", "braço esquerdo"  
]  
  
expect(game.missed_parts).to eq(missed_parts)  
  
end
```

Ao rodarmos o RSpec podemos ver que esse teste falha. Isso acontece porque o método `game.guess_letter` não está atualizando a variável de instância `@missed_parts` quando a adivinhação de letra foi errada. Vamos escrever um teste para especificar esse comportamento também:

```
describe "#guess_letter" do
```

```
# (...)
```

```
it
```

```
"updates the missed parts when the guess is wrong" do
```

```
  game.raffled_word =
```

```
    "hey"
```

```
    game.guess_letter(
```

```
      "z"
```

```
)
```

```
    expect(game.missed_parts).to eq([
```

```
      "cabeça"
```

```
    ])
```

```
end end
```

Ao rodar o RSpec, vemos que ele continua no vermelho. Para fazer os dois testes

que estão no vermelho passarem, precisamos fazer com que o método `Game#guess_letter` atualize a variável de instância `@missed_parts` de acordo com o número de erros de adivinhação de letra. Vamos começar primeiro salvando quais são as possíveis partes que o boneco pode perder em uma constante dentro da classe `Game`:

```
class Game
```

```
  HANGMAN_PARTS
```

```
  = [
```

```
    "cabeça", "corpo", "braço esquerdo"
```

```
    ,
```

```
    "braço direito", "perna esquerda", "perna direita"
```

```
  ]
```

```
  # (...) end
```

Agora que temos essa constante, para cada erro que é feito ao adivinhar uma letra, o método `guess_letter` precisa colocar um elemento da constante `HANGMAN_PARTS` na variável de instância `@missed_parts`. Para fazer isso, será necessário saber o número de erros de adivinhação de letra. Vamos criar uma variável de instância para salvar essa informação e inicializá-la com zero no construtor da classe `Game`:

```
class Game
```

```
  def initialize(word_raffler = WordRaffler  
    .new)
```

```
    @word_raffler  
    = word_raffler
```

```
    @state = :initial
```

```
    @guessed_letters  
    = []
```

```
    @missed_parts
```

```
= []
```

```
@wrong_guesses = 0
```

```
end end
```

Agora podemos utilizar essa variável para atualizar a `@missed_parts` quando for feita uma adivinhação errada:

```
def
```

```
  guess_letter(letter)
```

```
  return false if
```

```
    letter.strip.empty?
```

```
  if @raffled_word.include
```

```
    ?(letter)
```

```
  # (...)
```

```
else
```

```
@missed_parts << HANGMAN_PARTS[@wrong_guesses  
]
```

```
@wrong_guesses += 1
```

```
false
```

```
end end
```

Ao rodar o RSpec agora, podemos ver que ele está no verde. E o melhor, ao rodar o Cucumber, vemos que ele também está no verde:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

1 scenario (1 passed)

7 steps (7 passed)

Retire a tag @wip desse cenário e vamos para os dois últimos cenários desta funcionalidade.

Jogador pode tentar adivinhar mais de uma letra

Os dois cenários que estão faltando para finalizarmos esta funcionalidade são:

Cenário: Jogador advinha com sucesso duas vezes

Quanto mais o jogador for acertando, mais o jogo vai mostrando para ele as letras que ele adivinhou.

Cenário: Jogador erra três vezes ao adivinhar letra

Quanto mais o jogador for errando, mais partes do boneco da forca são perdidas.

A ideia desses cenários é testar o que acontece quando o jogador acerta ou erra mais de uma vez ao tentar adivinhar uma letra. Vamos começar criando o teste de aceitação para o primeiro cenário.

O teste de aceitação para esse cenário consistirá em simular que o jogador acerta duas vezes a adivinhação de letra e o jogo deve mostrar para o jogador as duas letras que ele adivinhou. Escreva esse teste do seguinte modo:

@wip

Cenário: Jogador advinha com sucesso duas vezes

Quanto mais o jogador for acertando, mais o jogo vai mostrando para ele as letras que ele adivinhou.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem

"3"

letras

Quando tento adivinhar que a palavra tem a letra

"a"

E tento adivinhar que a palavra tem a letra

"v"

E termino o jogo

Então o jogo termina com a seguinte mensagem na tela:

```

a v _

```

Ao rodar o Cucumber vemos que esse teste passa:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 passed)
```

```
7 steps (7 passed)
```

Esse teste passou de primeira porque o comportamento especificado por ele é apenas uma extensão do comportamento de adivinhação de letra com sucesso, e ele já foi implementado antes.

Retire a tag @wip desse teste e vamos para o próximo.

O último cenário servirá para verificar que, quando o jogador erra mais de uma vez ao tentar adivinhar uma letra, o jogo deve mostrar para ele as partes que o boneco da força perdeu de acordo como número de erros. Escreva esse teste do seguinte modo:

@wip

Cenário: Jogador erra três vezes ao adivinhar letra

Quanto mais o jogador for errando, mais partes do boneco da
força são perdidas.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem

"3"

letras

Quando tento adivinhar que a palavra tem a letra

"z"

E tento adivinhar que a palavra tem a letra

"y"

E termino o jogo

Então o jogo termina com a seguinte mensagem na tela:

""""

boneco da forca perdeu as seguintes partes do corpo: cabeça, corpo

""""

Ao rodarmos o Cucumber, vemos que esse teste também passa de primeira! Isso aconteceu pelo mesmo motivo do anterior, pois ele é apenas uma extensão do comportamento de erro ao adivinhar de letra. Como esse teste está passando, você já pode tirar a tag @wip dele também.

Para garantir que está tudo ok, rode a suíte inteira do RSpec e do Cucumber e verifique que elas estão no verde.

Pronto, depois de um árduo trabalho construindo essa funcionalidade, finalmente a finalizamos. Agora falta bem pouco para terminarmos o nosso jogo: falta apenas a funcionalidade de finalizar jogo, seja pelo jogador adivinhando todas as letras ou errando demais.

Pontos-chaves deste capítulo

Neste capítulo conseguimos finalizar a parte principal do nosso jogo, a funcionalidade que permite ao jogador tentar adivinhar uma letra. Foi um capítulo longo, não só devido ao desenvolvimento da funcionalidade em si, mas também devido às várias atividades que fizemos e aprendemos.

Começamos refatorando nosso jogo para controlar seu estado baseado em uma máquina de estados. Em cima do resultado desse refactoring, foi possível continuar o desenvolvimento do fluxo do jogo sem deixar o método

GameFlow#next_step com mais condicionais aninhadas.

Após o refactoring, reorganizamos a estrutura dos testes de unidade da classe GameFlow para que a leitura dos testes ficasse mais fácil e intuitiva. É muito importante que ao escrever um teste você pense não só no que o teste está verificando, mas também no teste como uma forma de documentação do seu software. Se for fácil de ler o seu teste e entender a relação de causa e consequência dele, ele será um dos melhores artefatos de documentação do seu sistema, pois é uma documentação executável, logo, bastante fidedigna ao comportamento real do seu software.

Por fim, aprendemos e utilizamos uma variação da técnica Interface Discovery, que utiliza o processo de TDD aliada a test doubles para descobrir e definir a API dos seus objetos. Encorajo você fortemente a ler sobre a versão canônica dessa técnica no paper Mock roles, not Objects (FREEMAN; MACKINNON; PRYCE; WALNES, 2004), para que você entenda o motivo de mock objects e como os seus criadores pensaram que eles devem ser utilizados. Lembre-se, mocks e stubs são coisas diferentes e têm aplicações diferentes (FOWLER, 2007).

Capítulo 16

Finalizando nosso jogo

Este é o último capítulo do desenvolvimento do nosso jogo. Até então, o jogador já pode começar um jogo, tentar adivinhar uma letra, adivinhar com sucesso e adivinhar sem sucesso. Faltava apenas a funcionalidade que permite ao jogador ganhar ou perder o jogo. Vamos desenvolvê-la e finalmente terminar o nosso projeto.

Ao final deste capítulo você terá desenvolvido uma aplicação completa usando BDD do começo ao fim. Parabéns! Isso pode ser um pequeno passo para o homem, mas um grande passo no desenvolvimento de software com qualidade.

16.1 Especificando o fim do jogo

Existem 3 modos de o nosso jogo terminar:

Jogador termina jogo no meio;

Jogador vence o jogo;

Jogador perde o jogo.

O primeiro modo nós já desenvolvemos nos capítulos anteriores. Vamos focar agora no segundo e terceiro modos.

Para que o jogador possa vencer o jogo, ele precisa adivinhar todas as letras do jogo antes que todas as partes do boneco da forca apareçam. Para que o jogador perca o jogo, basta que ele erre 6 vezes ao tentar adivinhar uma letra.

Vamos criar um arquivo de Cucumber com o nome `features/fim_do_jogo.feature` para documentar a especificação que definimos acima:

```
# language: pt
```


Funcionalidade: Fim do jogo

O jogo termina em dois cenários:

1. O jogador adivinhou todas as letras da palavra,
então ele ganha! :)
2. O jogador errou 6 vezes ao tentar adivinhar as letras da
palavra, então ele perde. :(

Cenário: Jogador vence o jogo

Para que o jogador possa vencer o jogo, ele precisa
adivinhar todas as letras do jogo antes que todas as partes
do boneco da forca apareçam.

Cenário: Jogador perde o jogo

Para que o jogador perca o jogo, basta que ele erre 6 vezes
ao tentar adivinhar uma letra.

Com a especificação e critério de aceite definidos, vamos escrever testes com o Cucumber para implementar esses critérios de aceite. Vamos começar pelo cenário onde o jogador vence o jogo.

16.2 Jogador vence o jogo

Para fazermos um teste do cenário onde o jogador vence o jogo, basta simularmos que o jogador consegue adivinhar todas as letras da palavra e verificarmos se uma mensagem de sucesso é impressa na tela. Escreva esse teste do seguinte modo no arquivo `features/fim_do_jogo.feature`:

Contexto:

*

o jogo tem as possíveis palavras para sortear:

| número de letras | palavra sorteada |

| 3 | avo |

@wip

Cenário: Jogador vence o jogo

Para que o jogador possa vencer o jogo ele precisa adivinhar todas as letras do jogo antes que todas as partes do boneco da forca apareçam.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem "3" letras

Quando tento adivinhar que a palavra tem a letra "a"

E tento adivinhar que a palavra tem a letra "v"

E tento adivinhar que a palavra tem a letra "o"

Então o jogo termina com a seguinte mensagem na tela:

```
""""
```

```
Você venceu! :)
```

```
""""
```

Perceba que além dos steps do cenário em si, foi necessário colocar os steps do Contexto, para que fosse possível saber que, quando o jogo sortear uma palavra com 3 letras, a palavra sorteada será "avo". Esse Contexto é o mesmo que usamos na funcionalidade Adivinhar letra.

Ao rodar o Cucumber dessa spec vemos que ele falha com a seguinte mensagem:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

Então o jogo termina com a seguinte mensagem na tela:

```
''''''
```

Você venceu! :)

```
''''''
```

```
expected "forca \"avo\""" to have finished in time
```

```
(RSpec::Expectations::ExpectationNotMetError)
```

Pela mensagem de erro podemos ver que o problema apontado é que após a execução de todos os steps do nosso teste, o processo do jogo não terminou. Isso porque nesse cenário não estamos utilizando aquele step E termino o jogo que utilizamos nas outras specs de Cucumber do nosso sistema. Não precisamos utilizar esse step porque o comportamento esperado é que, ao acertar todas as letras da palavra, o jogo imprima uma mensagem de sucesso e termine o seu processo sem ser necessário que o jogador digite mais nada.

Dado que essa spec de Cucumber está no vermelho, vamos escrever uma spec de RSpec para especificar o comportamento esperado.

Vamos começar primeiro com a estrutura dos testes que queremos escrever. Quando o jogador ganhar o jogo, o jogo estará no estado :ended e o jogo deve imprimir uma mensagem de sucesso. Escreva a seguinte estrutura de spec para esse comportamento no arquivo spec/game_flow_spec.rb:

```
describe "#next_step" do
```

```
# (...)
```

```
context
```

```
"when the game is in the 'ended' state" do
```

```
  before { allow(game).to receive(  
:state).and_return(:ended  
) }
```

```
  it
```

```
"prints a success message when the player wins"
```

```
end
```

```
# (...) end
```

A mensagem de sucesso quando o jogador vencer o jogo será: "Você venceu! :)".

Para verificar que a mensagem de sucesso foi impressa, basta verificar que a mensagem write foi enviada para o objeto ui com o argumento correto:

```
it "prints a success message when the player wins" do
```

```
  expect(ui).to receive(
    :write).with("Você venceu! :)")
)
```

```
  game_flow.next_step
```

```
end
```

Falta apenas uma informação no setup desse teste. Para que o game_flow saiba que o jogador venceu o jogo, não basta saber que o jogo acabou e que o objeto game está no estado :ended, é necessário saber que o jogo acabou e o jogador venceu. Vamos imaginar que exista um método chamado Game#player_won? que retorna essa informação e vamos fazer stub dele no setup do nosso teste:

```
it "prints a success message when the player wins" do
```

```
  allow(game).to receive(
    :player_won?).and_return(true
```

```
)
```

```
    expect(ui).to receive(  
      :write).with("Você venceu! :)")  
  )
```

```
    game_flow.next_step  
  end
```

Ao rodar o RSpec, vemos que esse teste quebra. Para fazê-lo passar, basta que modifiquemos o método `GameFlow#next_step` para lidar com o estado `:ended` do objeto `game`, checando se o jogador ganhou para poder imprimir a mensagem de sucesso:

```
def  
  next_step  
  
  case @game  
  .state  
  
  when :initial
```

```
ask_to_raffle_a_word
```

```
when :word_raffled
```

```
ask_to_guess_a_letter
```

```
end
```

```
print_game_final_result
```

```
if @game.ended? end
```

```
private
```

```
# (...)
```

```
def
```

```
print_game_final_result
```

```
if @game
```



```
.player_won?
```

```
@ui.write("Você venceu! :)")
```

```
)
```

```
end end
```

Ao rodar o RSpec agora, podemos ver que o teste está no verde. Agora é necessário implementarmos o método de que fizemos stub, o `Game#player_won?`.

Esse método deve ter o seguinte comportamento:

deve retornar false se o jogo não estiver no estado `:ended`, já que se o jogador ganhou, o jogo tem que estar no estado final;

deve retornar true caso o jogador tenha adivinhado todas as letras com sucesso;

deve retornar false caso o jogador não tenha adivinhado todas as letras com sucesso.

Vamos começar escrevendo um teste para o cenário de sucesso, quando o método retorna true. Nesse teste queremos simular que o jogador adivinhou todas as letras com sucesso para verificar que o retorno do método `player_won?` será true. Para simular esse cenário, devemos primeiro colocar o objeto `game` no estado `:word_raffled`, depois fazer com que todas as letras sejam adivinhadas com sucesso. Implemente esse comportamento em um novo teste no arquivo `spec/game_spec.rb`:

```
describe "#player_won?" do
```

```
  it
```

```
    "returns true when the player "
```

```
    \
```

```
    "guessed all letters with success" do
```

```
      game.state =
```

```
        :word_raffled
```

```
      game.raffled_word =
```

```
        "hi"
```

```
      game.guess_letter(
```

```
        "h"
```

```
      )
```

```
      game.guess_letter(
```

```
        "i"
```

```
)
```

```
expect(game.player_won?).to be_truthy
```

```
end end
```

Ao rodá-lo, vemos que ele quebra. Para fazê-lo passar, precisamos implementar o método `Game#player_won?`, fazê-lo checar que o estado do objeto `game` é igual a `:ended` e fazê-lo checar se o jogador adivinhou todas as letras com sucesso. Escreva esse método do seguinte modo no arquivo `lib/game.rb`:

```
def
```

```
  player_won?
```

```
    return false if @state != :ended
```

```
    raffled_word_letters =
```

```
    @raffled_word
```

```
    .to_s.chars.uniq
```

```
@guessed_letters.sort == raffled_word_letters.sort end
```

Ao rodar o teste, vemos que ele continua quebrando. Ele está retornando false porque o estado do jogo está chegando como :word_raffled, não como :ended. Isso acontece porque até então ninguém está fazendo a transição do estado :word_raffled para :ended quando o jogador adivinha todas as letras com sucesso. Essa transição deveria ser feita pelo método Game#guess_letter, quando o jogador adivinhar todas as letras com sucesso. Vamos escrever um teste para esse comportamento no arquivo spec/game_spec.rb:

```
describe "#guess_letter" do
```

```
# (...)
```

```
  it
```

```
    "makes a transition to the 'ended' state "
```

```
  \
```

```
    "when all the letters are guessed with success" do
```

```
      game.state =
```

```
:word_raffled
```

```
game.raffled_word =
```

```
"hi"
```

```
expect
```

```
do
```

```
game.guess_letter(
```

```
"h"
```

```
)
```

```
game.guess_letter(
```

```
"i"
```

```
)
```

```
end.to change { game.state }.from(:word_raffled).to(:ended
```

```
)
```

```
end end
```

Ao rodar, vemos que ele quebra. Para fazê-lo passar, vamos fazer o método `Game#guess_letter` atualizar o estado para `:ended` quando o jogador conseguir adivinhar todas as letras:

```
def
```

```
  guess_letter(letter)
```

```
  return false if
```

```
    letter.strip.empty?
```

```
    if @raffled_word.include
```

```
      ?(letter)
```

```
      @guessed_letters
```

```
      << letter
```

```
      @guessed_letters
```

```
      .uniq!
```

```
@state = :ended if
```

```
all_letters_were_guessed?
```

```
true
```

```
else
```

```
# (...)
```

```
end end
```

Note que utilizamos um método chamado `all_letters_were_guessed?` que ainda não existe. Esse método deve retornar `true` quando todas as letras já foram adivinhadas. Implemente-o como um método privado no arquivo `lib/game.rb`:

```
private
```

```
def
```

```
  all_letters_were_guessed?
```

```
    raffled_word_letters =
```

```
    @raffled_word
```

```
    .to_s.chars.uniq
```

```
    @guessed_letters.sort == raffled_word_letters.sort end
```

Ao rodarmos os testes agora, vemos que todos passam. Inclusive o teste do método `Game#player_won?!` Vamos para a refatoração.

Perceba que os métodos `player_won?` e `all_letters_were_guessed?` têm muita duplicação:

```
def
```

```
  player_won?
```

```
    return false if @state != :ended
```



```
raffled_word_letters =  
@raffled_word  
.to_s.chars.uniq  
  
@guessed_letters.sort == raffled_word_letters.sort end
```

private

```
def  
all_letters_were_guessed?  
  raffled_word_letters =  
  @raffled_word  
  .to_s.chars.uniq
```

```
@guessed_letters.sort == raffled_word_letters.sort end
```

Para retirarmos essa duplicação, basta reutilizarmos o método `all_letters_were_guessed?` dentro do método `player_won?`:

```
def
  player_won?

  return false if @state != :ended

  all_letters_were_guessed?
end
```

Rode o RSpec e verifique que ele continua no verde. Com o RSpec inteiro no verde, podemos escrever os outros testes do método `player_won?`.

Vamos escrever o teste para o caso quando o jogador terminou o jogo mas não conseguiu adivinhar todas as letras. Para o jogador terminar o jogo sem ter adivinhado todas as letras, ele tem que ter errado seis vezes:

```
describe "#player_won?" do
```

```
  # (...)
```

it

"returns false when the player didn't guess all letters" do

game.state =

:word_raffled

game.raffled_word =

"hi"

6.times { game.guess_letter("z"
) }

expect(game.player_won?).to be_falsey

end end

Ao rodar o RSpec, podemos ver que esse teste passa antes mesmo de termos implementando o código para fazê-lo passar. No fluxo de TDD, normalmente

quando escrevemos um teste, primeiro ele deve quebrar, e só depois que implementarmos código de produção, ele deve passar. Nesse caso, o teste já passou de primeira, sem a necessidade de escrevermos código novo.

Quando acontecer algo assim, você deve se certificar de que seu teste está realmente testando o que você quer e que seu teste está passando pelo motivo correto. No caso desse teste em questão, ele está passando, mas não pelo motivo correto.

Ele está passando porque o estado do objeto game nesse teste ficou setado como :word_raffled, pois não aconteceu a transição para o estado :ended. Ou seja, quem está fazendo o return false do método player_won? nesse teste não é a checagem de que o jogador adivinhou todas as letras com sucesso, mas sim a primeira linha do método que verifica o estado do objeto game:

```
def
```

```
  player_won?
```

```
    return false if @state != :ended
```

```
    all_letters_were_guessed?
```

```
  end
```

O que precisa ser feito é especificar e implementar que quando o jogador erra

seis vezes, o método `guess_letter` fará a transição do estado `:word_raffled` para `:ended`. Mas faremos isso só quando formos testar o cenário de Cucumber quando o jogador perde o jogo. Por enquanto, vamos nos ater ao cenário onde o jogador vence o jogo.

Ficou faltando fazermos um último teste do método `player_won?`, que é o teste que verifica que o retorno é `false` quando o jogo não está no estado `:ended`:

```
describe "#player_won?" do
```

```
  # (...)
```

```
  it
```

```
    "returns false when the game is not in the 'ended' state" do
```

```
      game.state =
```

```
        :initial
```

```
      expect(game.player_won?).to be_falsey
```

```
game.state =  
:word_raffled  
  
expect(game.player_won?).to be_falsey  
  
end
```

Ao rodarmos o RSpec, vemos que continua todo no verde. Vamos rodar o Cucumber e verificar se finalizamos o primeiro cenário da funcionalidade de "fim do jogo":

```
$ bundle exec cucumber -t @wip  
  
(...)
```

```
1 scenario (1 passed)
```

```
7 steps (7 passed)
```

O Cucumber está no verde, terminamos esse cenário! Retire a tag @wip dele e rode a suíte inteira de Cucumber para verificar se o sistema inteiro continua funcionando:

```
$ bundle exec cucumber
```

(...)

9 scenarios (9 passed)

47 steps (47 passed)

Como podemos ver pela saída do Cucumber, está tudo no verde. Sabendo que todos os testes de Cucumber e RSpec estão no verde, podemos ir para o próximo cenário, que será sobre quando o jogador perde o jogo.

16.3 Jogador perde o jogo

Na seção Especificando o fim do jogo nós já especificamos o critério de aceite do cenário para quando o jogador perde o jogo:

Cenário: Jogador perde o jogo

Para que o jogador perca o jogo, basta que ele erre 6 vezes ao tentar adivinhar uma letra.

Agora precisamos definir os steps desse cenário. Para implementá-lo, basta simularmos o jogador errando seis vezes ao tentar adivinhar uma letra da palavra sorteada e verificarmos que a mensagem "Você perdeu. :(" é impressa na tela:

@wip

Cenário: Jogador perde o jogo

Para que o jogador perca o jogo, basta que ele erre 6 vezes ao tentar adivinhar uma letra.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada tem

"3"

letras

Quando tento adivinhar que a palavra tem a letra

"z" "6"

vezes

Então o jogo termina com a seguinte mensagem na tela:

""

Você perdeu. :(

""

Note que, para fazer o jogador errar ao tentar adivinhar uma letra, estamos simulando que ele digitou seis vezes a letra "z". Sabemos que a letra "z" não está na palavra sorteada nesse cenário, porque esse cenário também roda no mesmo contexto do cenário que o jogador vence o jogo, no qual a palavra sorteada com três letra é "avo".

Ao rodarmos o Cucumber para esse cenário, vemos que um step está indefinido:

```
$ bundle exec cucumber -t @wip
```

(...)

1 scenario (1 undefined)

5 steps (1 skipped, 1 undefined, 3 passed)

0m0.042s

You can implement step definitions for undefined steps with these snippets:

```
Quando("tento adivinhar que a palavra tem  
    a letra {string} {string} vezes") do |string, string2|  
    pending # Write code here that turns the phrase above  
           # into concrete actions  
end
```

Esse step que está indefinido é muito parecido com um que já temos pronto:

Quando "tento adivinhar que a palavra "

\

"tem a letra {string}" do

```
|letter|  
  
  type(letter)  
  
end
```

Para implementá-lo, basta que ele faça o que esse step faz um número n de vezes. Faça isso do seguinte modo no arquivo `features/step_definitions/game_steps.rb`:

```
Quando 'tento adivinhar que a palavra tem a letra {string} '  
  
\  
  
  \"{int}\" vezes' do  
  
    |letter, number_of_guesses|  
  
      number_of_guesses.times  
  
do  
  
  type(letter)  
  
end end
```

Rode o Cucumber para esse cenário e verifique a saída:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
expected "forca \"avo\"" to have finished in time
```

```
(RSpec::Expectations::ExpectationNotMetError)
```

Failing Scenarios:

```
cucumber features/fim_do_jogo.feature:31
```

```
# Cenário: Jogador perde o jogo
```

```
1 scenario (1 failed)
```

```
5 steps (1 failed, 4 passed)
```

Pela saída do Cucumber podemos ver que o teste não passou porque, após todos os steps serem executados, o processo do nosso jogo não terminou. Esse mesmo comportamento aconteceu quando escrevemos o cenário de quando o jogador vence o jogo. Isso está ocorrendo porque ao jogador errar seis vezes o jogo não está finalizando sozinho e nem imprimindo a mensagem de derrota. Vamos escrever testes de unidade para especificar esse comportamento.

Vamos começar especificando o fluxo do jogo para quando o jogador perde. Escreva o seguinte teste no arquivo `spec/game_flow_spec.rb` para especificar que, quando o jogador perde, uma mensagem de derrota é impressa:

```
context "when the game is in the 'ended' state" do
```

```
  before { allow(game).to receive(  
:state).and_return(:ended  
) }
```

```
# (...)
```

```
  it
```

```
    "prints a defeat message when the player loses" do
```

```
      allow(game).to receive(  
:player_won?).and_return(false  
)
```

```
      expect(ui).to receive(  
:write).with("Você perdeu. :(")  
)
```

```
game_flow.next_step
```

```
end end
```

Note que, para saber que jogador perdeu, estamos acreditando que o contrato do método `Game#player_won?` é retornar `false` nesse caso.

Ao rodar o RSpec, vemos que esse teste quebra. Para fazê-lo passar, basta alterarmos o método privado `GameFlow#print_game_final_result`. Até então ele está assim:

```
def
```

```
print_game_final_result
```

```
if @game
```

```
.player_won?
```

```
@ui.write("Você venceu! :)")
```

```
)
```

end end

Modifique esse método para lidar com o cenário quando @game.player_won? retornar false:

def

print_game_final_result

if @game

.player_won?

@ui.write("Você venceu! :)"

)

else

@ui.write("Você perdeu. :("

)

end end

Ao rodarmos o RSpec, podemos ver que agora está no verde. Antes de seguirmos em frente e rodarmos o Cucumber, lembre-se de que na seção anterior falamos que ainda era necessário especificar e implementar que o método `Game#guess_letter` deve fazer uma transição de estado para `:ended`, quando o jogador errar seis vezes ao tentar adivinhar uma letra. Vamos fazer isso.

Escreva o seguinte teste para especificar esse comportamento no arquivo `spec/game_spec.rb`:

```
describe "#guess_letter" do
```

```
  # (...)
```

```
    it
```

```
      "makes a transition to the 'ended' state when the player "
```

```
      \
```

```
      "misses 6 times trying to guess a letter" do
```

```
        game.state =
```



```
:word_raffled
```

```
game.raffled_word =
```

```
"hi"
```

```
expect
```

```
do
```

```
6.times { game.guess_letter("z"
```

```
) }
```

```
end.to change { game.state }.from(:word_raffled).to(:ended
```

```
)
```

```
end end
```

Ao rodarmos o RSpec, vemos que esse teste quebra. Para fazê-lo passar, precisamos alterar o método `Game#guess_letter` para fazer a transição de estado quando o jogador errar seis vezes. Até então esse método está do seguinte modo:

```
def
```

```
  guess_letter(letter)
```

```
  return false if
```

```
    letter.strip.empty?
```

```
  if @raffled_word.include
```

```
    ?(letter)
```

```
    # (...)
```

```
  else
```

```
    @missed_parts << HANGMAN_PARTS[@wrong_guesses
```

```
  ]
```

```
    @wrong_guesses += 1
```

```
false
```

```
end end
```

Note que já existe um contador de quantas vezes o jogador errou. Basta fazermos a transição de estado quando esse contador for igual a seis:

```
def
```

```
  guess_letter(letter)
```

```
  return false if
```

```
    letter.strip.empty?
```

```
  if @raffled_word.include
```

```
    ?(letter)
```

```
  # (...)
```

else

@missed_parts << HANGMAN_PARTS[@wrong_guesses
]

@wrong_guesses += 1

@state = :ended if @wrong_guesses == 6

false

end end

Ao rodarmos o RSpec agora, vemos que ele está no verde. Aproveite e rode o Cucumber do cenário em que estamos trabalhando:

```
$ bundle exec cucumber -t @wip
```

```
(...)
```

```
1 scenario (1 passed)
```

```
5 steps (5 passed)
```

O Cucumber está no verde também, finalizamos o último cenário do nosso jogo! Agora que esse cenário está no verde, retire a tag @wip dele.

Apenas para verificar que está tudo ok, rode o RSpec e Cucumber da suíte inteira e confirme que está tudo no verde.

Sim, terminamos de desenvolver nosso jogo. Espero que tenha sido tão legal para você quanto foi para mim!

16.4 Próximos passos

Você acabou de construir uma aplicação inteira usando BDD e outside-in development do começo ao fim. Espero que o desenvolvimento dessa aplicação tenha deixado mais claro para você como é desenvolver software utilizando BDD.

Você agora já deve entender o tradicional fluxo "red, green, refactor" do TDD. Deve entender também a técnica "outside-in development", que é o modo padrão de desenvolver software segundo a filosofia do BDD (Behavior-Driven Development).

O próximo passo para você ganhar confiança e experiência no desenvolvimento de software orientado a testes é praticar. Praticar, praticar e praticar. O desenvolvimento que fizemos neste livro foi apenas um dos seus primeiros exercícios nesta técnica. Incentivo você a utilizar BDD em todos os seus projetos, pois só assim você poderá dominar essa técnica.

Vou deixar também uma lição de casa para você já ter pelo menos um próximo exercício a fazer. No livro *Test-Driven Development: Teste e Design no Mundo Real*, o Mauricio Aniche (2012) fala que muitos testes para um único método pode ser um indício de problema de design. Devemos ouvir o feedback dos nossos testes e utilizá-lo para melhorar o design do nosso software. Nosso jogo tem um caso com muitos testes para um método só, no arquivo `spec/game_flow_spec.rb`:

```
$ rspec -fd -e "#next_step" spec/game_flow_spec.rb
```

GameFlow

#next_step

finishes the game when the player asks to

when the game is in the 'initial' state

asks the player for the length of the word to be raffled

and the player asks to raffle a word

raffles a word with the given length

prints a '_' for each letter in the raffled word

tells if it's not possible to raffle with the given

length

when the game is in the 'word raffled' state

asks the player to guess a letter

and the player guess a letter with success

prints a success message

prints the guessed letters

and the player fails to guess a letter

prints an error message

prints the list of the missed parts

when the game is in the 'ended' state

prints a success message whe the player win

prints a defeat message when the player loses

Perceba quão grande é o número de testes que fizemos só para esse método, o `GameFlow#next_step`. Seu dever de casa será refatorar a classe `GameFlow` para evitar que seja necessário fazer tantos testes para um método só.

Capítulo 17

Referências bibliográficas

ADZIC, Gojko. Specification by example: How successful teams deliver the right software. Manning Publications, 2011.

ANICHE, M. Test-Driven Development: Teste e Design no Mundo Real. Casa do Código, 2012.

BOEHM, Barry W. Software engineering economics. Prentice Hall, 1981.

CHELIMSKY, David; ASTELS, Dave; DENNIS, Zach; HELLESOY, Aslak; HELMKAMP, Bryan; NORTH, Dan. The RSpec book: Behaviour-Driven Development with RSpec, Cucumber, and Friends. Pragmatic Bookshelf, 2009.

EVANS, Eric. Domain-Driven Design: Tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

FOWLER, M. Mocks aren't stubs. Jan. 2007. Disponível em: <https://martinfowler.com/articles/mocksArentStubs.html>

FREEMAN, S.; MACKINNON, T.; PRYCE, N.; WALNES, J. Mock roles, not objects. 2004. Disponível em: <http://www.jmock.org/oopsla2004.pdf>.

FREEMAN, S.; PRYCE, N. Growing object-oriented software, guided by tests. Addison-Wesley Professional, 2009.

MACKINNON, T.; FREEMAN, S.; CRAIG, P. Endo-testing: Unit testing with mock objects. 2001. Disponível em:
<http://www.ccs.neu.edu/research/demeter/related-work/extreme-programming/MockObjectsFinal.PDF>

MARTIN, Robert C. Principles of OOD. 2005. Disponível em:
<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.

MESZAROS, G. xUnit test patterns: refactoring test code. Addison-Wesley Professional, 2007.

NORTH, D. Introducing BDD. Mar. 2006. Disponível em:
<http://dannorth.net/introducing-bdd/>.