



Cucumber e RSpec | Construa aplicações Ruby e Rails com testes e especificações

Cucumber e RSpec

Construa aplicações Ruby e Rails com testes e especificações



Casa do
Código

HUGO BARAÚNA

Dedicatória

Dedico a minha mãe e minha noiva.

Agradecimentos

Agradeço a minha mãe, minha noiva, amigos e Plataformatec.

Sobre o autor

Hugo Baraúna é co-fundador da Plataformatec, empresa de consultoria em desenvolvimento de software especializada em Ruby e Rails. A Plataformatec é referência nacional e internacional no mundo Ruby, devido principalmente a seus projetos open source e sua colaboração com a comunidade. Ele atua tanto na direção da empresa quanto como desenvolvedor, tendo participado de projetos de consultoria, coaching e testes de carga.

Hugo se formou em Engenharia de Computação pela Politécnica da USP em 2010. Durante a faculdade, passou pelo laboratório USP-Microsoft e por empresas como Procwork e IBM.

Para ele, só é possível fazer produtos e serviços de qualidade quando se ama o que faz.

Prefácio

Sumário

1	Visão geral sobre TDD	1
1.1	TDD e sua história	2
1.2	E por qual motivo eu deveria usar TDD?	3
2	Primeiros passos com RSpec e Cucumber	5
2.1	Olá RSpec	5
2.2	Olá Cucumber	13
2.3	O que é BDD?	21
3	Introdução ao básico do RSpec	23
3.1	Aprendendo a estrutura básica de um teste com RSpec	23
3.2	Porquê existem tantos matchers no RSpec	26
3.3	Conhecendo os RSpec built-in matchers	33
3.4	Matchers relacionados a truthy e falsy	33
3.5	Os matchers de equidade	34
3.6	Matchers relacionados a arrays	35
3.7	Custom matchers	46
3.8	Entendendo o protocolo interno de matcher do RSpec	53
3.9	Pontos-chave deste capítulo	56
4	Organização, refatoração e reuso de testes com o RSpec	59
4.1	Reduzindo duplicação com hooks do RSpec	59
4.2	DRY versus clareza nos testes	65
4.3	After hook	67
4.4	Around hook	69

4.5	Organizando seus testes	70
4.6	Reuso de testes	79
4.7	Pontos-chave deste capítulo	86
5	TDD na prática, começando um projeto com TDD	87
5.1	Definindo o escopo da nossa aplicação: Jogo da Força	88
5.2	Especificando uma funcionalidade com Cucumber	89
5.3	Usando RSpec no nosso primeiro teste	94
5.4	Usando Aruba para testar uma aplicação CLI	99
5.5	Pontos-chave deste capítulo	104
6	Começando o segundo cenário	107
6.1	Definindo o segundo cenário	107
6.2	Reduza duplicação através de support code	108
6.3	Implementando o fluxo do jogo no binário	110
6.4	Modificando nosso cenário para receber o feedback correto	113
6.5	Usando subject e let do RSpec para evitar duplicação nos testes	116
6.6	Refatorando o código para poder implementar o segundo cenário	119
6.7	Extraindo uma classe através de refatoração	123
6.8	Possibilitando ao jogador terminar o jogo no meio	127
6.9	Pontos-chave deste capítulo	129
7	Finalizando a primeira funcionalidade	131
7.1	Continuando o segundo cenário	131
7.2	Deixando o segundo cenário no verde	132
7.3	Finalizando a primeira funcionalidade	139
7.4	Pontos-chave deste capítulo	142
8	Refatorando nosso código	143
8.1	Identificado os pontos a serem refatorados	143
8.2	Extraindo uma classe de um método privado	146
8.3	Distribuindo responsabilidades para outras classes	153
8.4	Pontos-chave deste capítulo	167

9	Especificando a segunda funcionalidade	169
9.1	Documentando especificação e critério de aceite com Cucumber . . .	169
9.2	Definindo o teste de aceitação do primeiro cenário	171
9.3	Melhore a testabilidade do seu software	175
9.4	Pontos-chave deste capítulo	184
10	Finalizando a segunda funcionalidade	187
10.1	Refatorando nosso jogo para ter uma máquina de estados	187
10.2	Refatorando o fluxo do jogo para usar a máquina de estados	193
10.3	Organizando seus testes otimizando para leitura	198
10.4	Interface discovery utilizando test doubles	201
10.5	Finalizando a funcionalidade Adivinhar letra	219
10.6	Pontos-chave deste capítulo	229
11	Finalizando nosso jogo	231
11.1	Especificando o fim do jogo	231
11.2	Jogador vence o jogo	232
11.3	Jogador perde o jogo	241
11.4	Próximos passos	246
	Bibliografia	249

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 ainda não faz isso, você 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 irá aprender a fazer TDD, o *Test-Driven Development*, usando algumas das ferramentas mais famosas na comunidade Ruby: o RSpec e o Cucumber. Você irá aprender sobre as ferramentas em si e também sobre 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 irá mudar o modo como você escreve software... para melhor.

1.1 TDD E SUA HISTÓRIA

A história de TDD começa principalmente no começo da década de 90, quando Kent Beck escreve 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 das 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 de 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, PHP e outras. 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 depois fazem o código que vai fazer com esse teste passe. A esse uso das bibliotecas xUnit, se deu o nome de 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 visto 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, o uso de TDD e testes automatizados sempre foi a regra, e não a exceção. Como eu disse antes, todo projeto open source em Ruby que se preza tem testes automatizados.

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, ou seja, Rails é tão bom que as pessoas começaram a aprender Ruby só por causa dele. Como um dos

softwares mais famosos escritos em Ruby, 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 você colocar seus testes automatizados, ou seja, Rails te convida a fazer TDD.

Daí pra 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* [2], Barry Boehm sugeriu que o custo de alteração em um projeto de software cresce exponencialmente à medida que se avança nas fases do desenvolvimento.

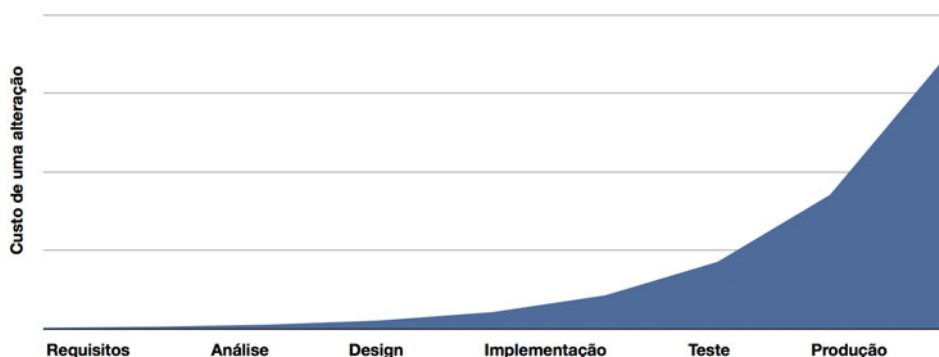


Figura 1.1: Custo de alteração em um projeto de software

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, onde uma das práticas que ajuda a diminuir esse custo é o TDD.

Uma das consequências de fazer 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 então dirá se você quebrou algum comportamento previamente implementado.

Segundo o XP, o desenvolvedor deve ao longo do projeto refatorar constantemente o código para deixar o design o melhor possível. Com a refatoração constante, o design do software pode se manter bom, de modo que o custo de alteração do sistema não cresça exponencialmente. Na prática, isso quer dizer que se no começo do projeto leva uma semana para adicionar uma funcionalidade, um ano depois, deve 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 poder se apoiar na suíte de teste toda vez que você for modificar o software te da 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 possa fazer mudanças arquiteturais no seu código, garantindo que ele irá continuar funcionando. Permite também que novos integrantes da equipe possam conseguir 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, significa que ele pode estar acoplado demais ou com baixa coesão. TDD nos ajuda a detectar esse tipo de problema, nos sugerindo melhorar o design do nosso código. Essa vantagem vai ficar mais clara quando formos desenvolver um projeto inteiro com TDD, a partir do capítulo 5.

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 ao invés 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, que é uma extensão do TDD. Vamos falar melhor sobre BDD 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 1, 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. Uma vez o teste feito, você desenvolve o pedaço de software que você 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 você julgar necessárias. A esse fluxo do TDD se dá o nome de red - green - refactor. Vamos aplicá-lo para ficar mais claro.

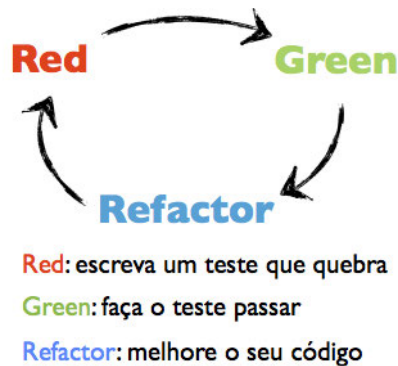


Figura 2.1: Ciclo red - green - refactor

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 pra 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, então esse elemento deve poder ser visualizado no topo da pilha.

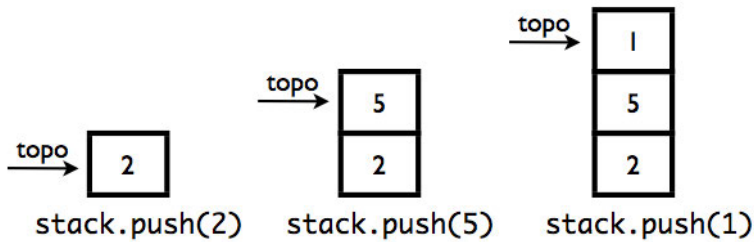


Figura 2.2: Pilha com elementos sendo empilhados

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

```
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
```

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 e por fim, indicamos que o `topo` da pilha deve conter o elemento `2`, seguindo assim 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 --color 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 porquê ele quebrou. Após ter rodado o RSpec, você deve ter visto uma mensagem de erro contendo a seguinte saída:

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

uninitialized constant Stack (NameError)
```

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

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 --color 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>
```

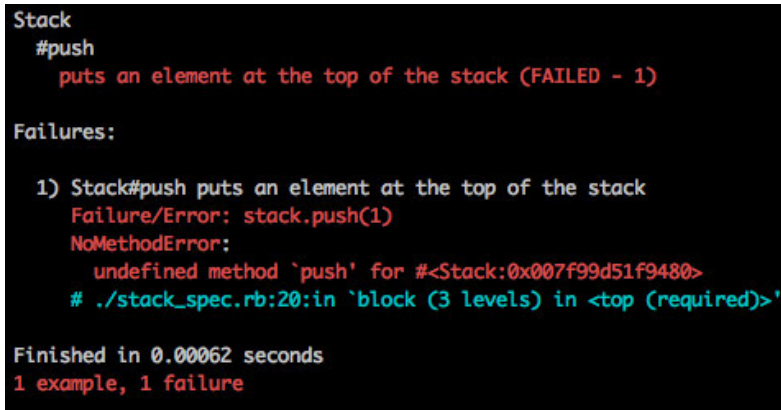
A screenshot of a terminal window with a black background and red and white text. It shows the output of an RSpec test. The first line is 'Stack', followed by '#push' and 'puts an element at the top of the stack (FAILED - 1)'. Then 'Failures:' is shown, followed by a list of failures. The first failure is '1) Stack#push puts an element at the top of the stack', followed by 'Failure/Error: stack.push(1)', 'NoMethodError:', and 'undefined method `push' for #<Stack:0x007f99d51f9480>'. Below this is a line indicating the file and line number: '# ./stack_spec.rb:20:in `block (3 levels) in <top (required)>''. At the bottom, it says 'Finished in 0.00062 seconds' and '1 example, 1 failure'.

Figura 2.3: Teste no vermelho

Evoluímos, mas o teste ainda quebrou e a mensagem de erro agora nos diz que foi porque o método `push` ainda não está definido, precisamos implementar esse método ainda.

Para implementar o método `push`, precisamos pensar no que ele irá 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 --color 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 quer 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, no array interno `@elements`, adicionado a pilha. 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
```

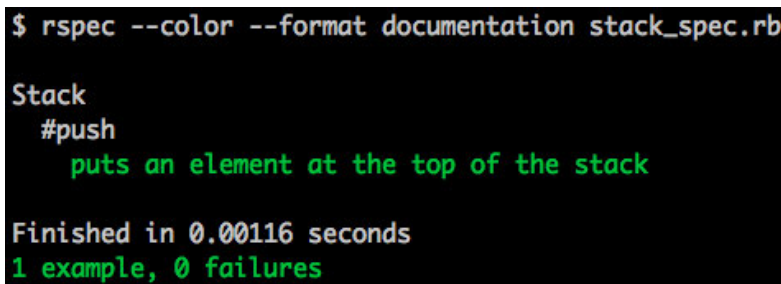
Ao rodarmos o RSpec, dessa vez vemos o seguinte:

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

```
Stack
  #push
    puts an element at the top of the stack
```

```
Finished in 0.00549 seconds
```

```
1 example, 0 failures
```

A screenshot of a terminal window with a black background. The text is displayed in a monospaced font. The command '\$ rspec --color --format documentation stack_spec.rb' is at the top. Below it, the test output is shown: 'Stack' followed by an indented '#push' which has an indented 'puts an element at the top of the stack' below it. The text 'puts an element at the top of the stack' is highlighted in green. At the bottom, it says 'Finished in 0.00116 seconds' and '1 example, 0 failures', with the last line also highlighted in green.

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

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

Finished in 0.00116 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 o teste e/ou o código para melhorá-los. Refatorar é melhorar o código sem mudar seu comportamento externo, não adicionando nenhum comportamento novo, apenas melhorando a qualidade interna do nosso software.

Um ponto que podemos melhorar no nosso código é o modo como estamos pegando o último elemento salvo na pilha. Ao invés de criar uma variável de instância para pegar o último elemento do array interno, podemos simplesmente fazer `elements[-1]`. Passando `-1` para índice do array, indicamos que estamos querendo o último elemento adicionado.

Seguindo essa ideia, modifique o código para ficar assim:

```
class Stack
  def initialize
    @elements = []
  end
```



```
def push(element)
  @elements << element
end

def top
  @elements[-1]
end
end
```

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

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

```
Stack
  #push
    puts an element at the top of the stack
```

```
Finished in 0.00549 seconds
1 example, 0 failures
```

Sucesso, 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 3 iremos entrar em detalhes no RSpec, entender essa sintaxe estranha e as muitas nuances do framework. E a partir do capítulo 5 iremos 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:

```
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 do que está testando deve ser feito 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, ou seja, 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 irá especificar uma funcionalidade do seu sistema, ao invés de especificar uma classe ou um método. De modo bem resumido, entenda o Cucumber como uma espécie de documentação de requisitos funcionais *on steroids*.

Eu digo isso, pois 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.

Você foi contratado para desenvolver um pequeno jogo da força, que irá funcionar no shell. Você conversa com o cliente, ele te explica o motivo de querer o jogo e como ele pensa que ele irá funcionar.

Você entende o contexto inteiro e decide documentar a primeira funcionalidade para então pedir ao cliente para 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 irá funcionar do ponto de vista do usuário final.

CENÁRIOS EM PORTUGUÊS

Diferentemente de quando escrevemos as especificações com o RSpec, onde 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 então 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
```

Agora crie um novo diretório chamado `hello-cucumber` e dentro dele crie um outro diretório chamado `features`. Dentro desse diretório `features`, crie um arquivo chamado `features/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)
0m0.002s
```

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

```
Quando /^começo um novo jogo$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Então /^vejo a seguinte mensagem na tela:$/ do |string|
  pending # express the regexp above with the code you wish you had
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: (...)"*.

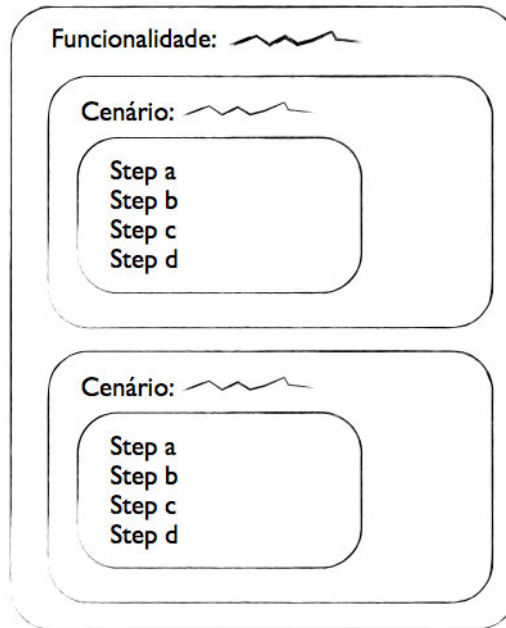


Figura 2.5: Estrutura de uma especificação com Cucumber

Para utilizar essa documentação como um teste automatizado, temos que definir cada um desses steps, implementando o que o Cucumber chama de *step definitions*. Basicamente é dizer ao Cucumber o que ele precisa fazer naquele passo. 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 # express the regexp above with the code you wish you had
end
```

```
Então /^vejo a seguinte mensagem na tela:$/ do |string|
```

```
pending # express the regexp above with the code you wish you had
end
```

Vamos agora automatizar nossa especificação. Primeiro crie um diretório dentro do `features` e vamos chamá-lo de `step_definitions`. Dentro desse diretório crie um arquivo chamado `game_steps.rb`. Nesse arquivo vamos colocar os step definitions dos dois que temos na nossa spec. Implemente esses steps do seguinte modo nesse arquivo:

```
# encoding: UTF-8

Quando /^começo um novo jogo$/ do
  @game = Game.new
  @game.start
end

Então /^veja 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. O nosso jogo será um jogo de console, mas por enquanto vamos manter o teste bem simples nesse “hello world”, sem uma interação real do teste com o console. Iremos fazer esse tipo de teste de modo completo e detalhado a partir do capítulo 5.

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 agora 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 não vamos fazer isso agora, para manter o exemplo simples. O ciclo inteiro de BDD com Cucumber e RSpec será feito extensivamente a partir do capítulo 5.

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:

```
# encoding: UTF-8

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:0x007fde41b5a938> (NoMethodError)

(...)
```


Dessa vez o teste quebrou porque o método `Game#start` não foi definido ainda. Vamos implementar esse método, que vai salvar dentro do `output` do jogo a mensagem inicial. Adicione esse método à classe `Game` do seguinte modo:

```
# encoding: UTF-8

# (...)

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:0x007fbcd268410> (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`:

```
# encoding: UTF-8

# (...)

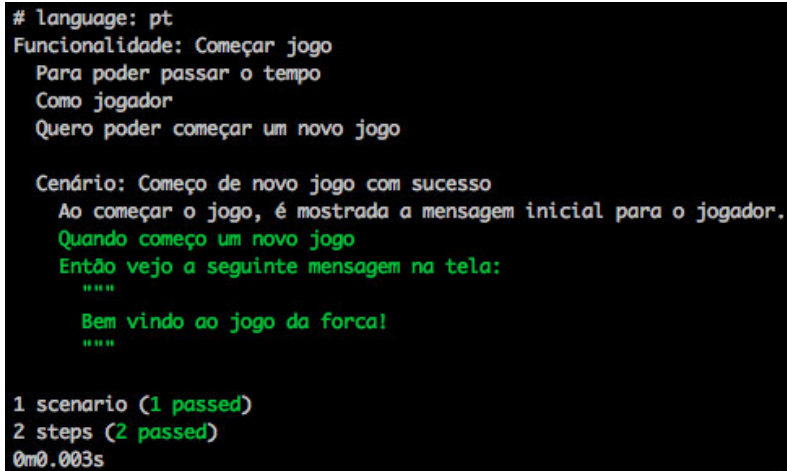
class Game
  attr_reader :output

# (...)
end
```

Ao rodar o Cucumber agora, 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)
0m0.003s
```

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 ideia 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 [11]. 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 do mesmo.

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 do 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, ele agora é uma abordagem de desenvolvimento de software [3] que propõe que você desenvolva seu software especificando as várias camadas de comportamento com testes automatizados. Comece primeiro especificando o comportamento de uma funcionalidade. Depois disso, especifique as classes e métodos que serão necessárias para implementar essa funcionalidade.

À prática de começar o desenvolvimento de uma funcionalidade com testes de aceitação e depois fazer os testes de unidade, se dá o nome de *acceptance test-driven development* (ATDD). Essa é uma das práticas nas quais o BDD se baseia. Uma das outras práticas é o famoso *domain-driven design* [4].

A partir do capítulo 5 nós iremos desenvolver uma aplicação seguindo o fluxo de BDD. Começaremos por testes de aceitação com Cucumber e em seguida faremos testes de unidade com 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

Introdução ao básico do RSpec

Neste capítulo você irá aprender como usar o RSpec para fazer testes de unidade. Iremos ver como o RSpec pode nos ajudar a escrever testes que sejam legíveis e que possam servir não só como testes de regressão mas também 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:

```
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
```

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

```
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:

```
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 para frente vamos falar mais dessa convenção. Vamos agora 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 acima 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 argumento 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 onde verificamos se o nosso código se comportou do modo esperado. No RSpec as asserções são feitas com *expectations*. No teste acima 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)
```

Onde `actual` é o objeto do qual você está testando algum comportamento, `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.

DIFERENTES SINTAXES DE EXPECTATION: EXPECT X SHOULD

Se você já viu ou escreveu algum teste com RSpec antes de ler este livro, é provável que você tenha visto a sintaxe antiga do RSpec para criar expectations, que era usando o método `should`.

Antigamente, o padrão de expectations do RSpec era do seguinte modo:

```
stack.top.should eq(2)
```

ao invés de:

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

Na versão 2.11 do RSpec a nova sintaxe `expect` foi introduzida, e desde então é a sintaxe recomendada pelo equipe core do RSpec.

Se você tiver interesse, você pode ver o porquê dessa mudança em um post (<http://myronmars.to/n/dev-blog/2012/06/rspecs-new-expectation-syntax>) no blog do Myron Marston, atual mantenedor do RSpec.

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 vem por padrão com o RSpec.

3.2 PORQUÊ EXISTEM TANTOS MATCHERS NO RSpec

Antes de começarmos a ver os diversos matchers que o RSpec nos oferece por padrão, vale a pena entendermos uma das vantagens em ter tantos matchers 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, onde você pode colocar palavras dentro dele e depois verificar quantas palavras tem dentro do saco. Comece criando um diretório para o nosso projeto, chamado de `bag_of_words`:

```
$ mkdir bag_of_words
```

```
$ cd bag_of_words
```

Crie um Gemfile usando o bundler:

```
$ bundle init
```

Coloque o RSpec no Gemfile:

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

```
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 você 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 `lib`:

```
$ mkdir lib
```

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


```
$ rspec --init
```

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

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. E 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 "spec_helper"
require "bag_of_words"

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 `spec_helper`, aquele arquivo gerado automaticamente pelo comando `rspec --init`. Nesse arquivo ficam as configurações gerais da nossa suíte de testes, então fizemos `require` dele para garantir que essas configurações sejam aplicadas ao nosso teste.

Depois fizemos `require` do arquivo `bag_of_words`, que ainda nem existe, mas é onde colocaremos o código da nossa classe.

Por fim, 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

(...)
cannot load such file -- bag_of_words (LoadError)
```

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 a implementação acima 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

```

Pela mensagem podemos ver que o esperado era que o `size` do array `words` fosse 2, mas veio 0. A mensagem está correta e está de acordo com nosso objetivo final, mas ela pode ficar melhor. O que queremos testar é que existem 2 palavras na `bag`, e não que o array `words` tem 2 elementos. Perceba que os dois modos de pensar dão no mesmo resultado, mas pensar em “palavras dentro do saco” está mais perto do domínio do nosso problema do que “número de elementos de um array”. Podemos melhorar a expressividade do nosso teste usando um matcher mais apropriado para esse contexto, que nesse caso seria o *Have Matcher*.

Seguindo essa idéia, reescreva a expectation do nosso teste para ficar assim:

```

require "spec_helper"
require "bag_of_words"

describe BagOfWords do
  it "is possible to put words on it" do
    bag = BagOfWords.new

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

    expect(bag).to have(2).words
  end
end

```

E rode o teste novamente:

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

```
(...)
```

Failures:

```
1) BagOfWords is possible to put words on it
   Failure/Error: expect(bag).to have(2).words
     expected 2 words, got 0
```

Perceba como a mensagem de erro mudou. A mensagem agora fala: “expected 2 words, got 0”. A mensagem agora faz bem mais sentido em relação ao domínio do nosso problema! Perceba também como fica diferente ler a nova expectation comparada a antiga:

```
# antiga
expect(bag.words.size).to eq(2)

# nova
expect(bag).to have(2).words
```

Não só a mensagem de erro ficou mais expressiva como também o próprio código do nosso teste também ficou! Essa é a vantagem de se usar o matcher correto. É por isso que o RSpec vem com mais de 30 matchers, para que nossos testes sejam expressivos e para que possamos aplicar a idéia do BDD de escrever testes que sirvam como especificação e documentação do nosso código.

Bom, agora que já estamos usando o matcher correto, vale a pena fazermos 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
```

Agora ao rodar os testes, vemos que eles passam:

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

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

```
Finished in 0.00139 seconds
1 example, 0 failures
```

Agora estamos prontos para dar uma olhada nos built-in matchers do RSpec (que são os matchers padrão do RSpec).

Curiosidade: a manipulação do `$LOAD_PATH` feita pelo RSpec

Para entender essa curiosidade, você deve saber que em Ruby ao 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 acima, você deve se lembrar que `demos require` do `spec_helper` e do arquivo `bag_of_words`:

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

```
require "spec_helper"
require "bag_of_words"
```

```
describe BagOfWords do
  # (...)
end
```

Ao ler o código de testes de outros projetos com RSpec, é possível que você veja que a pessoa está inserindo o diretório `lib/` explicitamente no `$LOAD_PATH` para conseguir dar `require` em um arquivo desse diretório. Um exemplo desse tipo de código poderia ser assim:

```
require "spec_helper"

$LOAD_PATH.unshift File.join(File.dirname(__FILE__), "..", "lib")
require "bag_of_words"
```

Há tempos atrás eu me perguntei porque 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/tree/v2.13.1/>), no arquivo `lib/core/load_path.rb`:

```
require 'rspec/core/ruby_project'

RSpec::Core::RubyProject.add_to_load_path('lib', 'spec')
```

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 acima pode ser:

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

Como mencionado na seção anterior, o RSpec vem com muitos built-in matchers para nos ajudar a escrever testes expressivos. Vamos dar uma olhada neles, começando pelos mais básicos, os *"be matchers"*.

3.4 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:

```
expect(obj).to be_true   # passa se obj é truthy (não nil ou não false)
expect(obj).to be_false  # passa se obj é falsy (nil ou false)
expect(obj).to be_nil    # passa se obj é nil
expect(obj).to be        # passa se obj é truthy (não nil ou não false)
```

Note que os `be matchers` já seriam o suficiente para fazer qualquer tipo de teste pro seu software. Por exemplo, poderíamos re-escrever o teste:

```
it "is possible to put words on it" do
  bag = BagOfWords.new

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

  expect(bag).to have(2).words
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_true
end
```

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

3.5 OS MATCHERS DE EQUIDADE

Os `equality matchers` servem para verificar se um objeto é igual a outro objeto. Existe mais de um matcher para esse tipo de verificação porque no Ruby existe mais de um modo de checar a equidade entre dois objetos, que são os seguintes:

```
a.equal?(b) # object identity - a e b se referem ao mesmo objeto
a.eql?(b)   # object equivalence - a e b tem o mesmo valor
a == b      # object equivalence - a e b tem o mesmo valor com conversão
             # de tipo
```

Os matchers que o RSpec tem para verificar equidade são:

```
expect(a).to equal(b) # passa se a.equal?(b)
expect(a).to be(b)    # passa se a.eql?(b)
expect(a).to eql(b)   # passa se a.eql?(b)
expect(a).to eq(b)    # passa se a == b
```

3.6 MATCHERS RELACIONADOS A ARRAYS

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

```
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 a 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 desse matcher:

```
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 desses matchers:

```
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 entender esses matchers 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?`. Para especificar esse método, você poderia fazer o seguinte teste:

```
describe Cart do
  describe "#empty?" do
    it "returns true when the cart has no products" do
      cart = Cart.new

      expect(cart.empty?).to be_true
    end
  end
end
```

Acontece que toda vez que um objeto tem um predicate method (método que termina com `?`), o RSpec gera um matcher específico para esse método dinamicamente. No caso do exemplo acima, ao invés de fazer:

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

Você pode utilizar um predicate matcher para re-escrever o teste do seguinte modo:

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

O que o RSpec faz é gerar para todo predicate method chamado `method_name?` um matcher chamado `be_method_name`. Foi o que fizemos acima quando usamos o matcher `be_empty` referente ao predicate method `empty?`. Esse tipo de matcher é bem usado porque os predicate methods são uma convenção bastante usada na comunidade Ruby.

Os predicate matchers ainda tem outras três formas. A segunda forma é quando o RSpec gera um matcher `have_method_name` para métodos da forma `has_method_name?`. Um exemplo desse caso pode ser visto para o método `has_key?` de um hash:

```
hash = { key: 1 }
expect(hash).to have_key(:key) # chama o método hash.has_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:

```
describe Cart do
  describe "#has_products?" do
    it "returns true if it has products" do
      product = Product.new
      cart = Cart.new(product)

      expect(cart).to have_products # chama o método cart.has_products?
    end
  end
end
```

A terceira e quarta formas dos predicate matchers são os matchers `be_a_method_name` e `be_an_method_name`. Para entender essas últimas formas, imagine que sua classe `Cart` tem os métodos `thing?` e `object?`, e ambos devem retornar `true`. Você pode especificar esses métodos 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 message' }.to raise_error('error message')
expect { raise 'wrong message' }.to_not raise_error('error message')
```

Matchers para comparação de números

Para comparar se um valor é igual a um certo número você deve usar o *eq matcher*: `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 a 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 o teste acima 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.

Com o problema acima em mente, o RSpec nos oferece o *BeWithin matcher*, que serve para fazer comparações entre números de forma aproximada. Poderíamos re-escrever o teste acima do seguinte modo com esse matcher:

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

O que o teste acima 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:

```
(expected - delta) <= actual <= (expected + 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(Fixnum)

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(Fixnum)
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)
```

Matchers para verificar tamanho de coleção

Se você quer checar o tamanho de uma coleção, você pode usar o *Have Matcher*. Na seção 3.2 nós vimos um exemplo de uso desse matcher quando escrevemos o seguinte teste:

```
describe BagOfWords do
  it "is possible to put words on it" do
    bag = BagOfWords.new

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

    expect(bag).to have(2).words
  end
end
```

Vamos entender melhor o que a linha `expect(bag).to have(2).words` faz por debaixo dos panos:

- 1) a primeira coisa que o *Have Matcher* faz é determinar se o objeto sendo testado é uma coleção ou se ele contém uma coleção. Para fazer essa checagem ele vê o método que foi passado para o `have`, no caso acima foi o método `words`, e checa se o objeto sendo testado responde para esse método: `bag.respond_to?(:words)`;
- 2) como o objeto `bag` responde para o método `words`, o *Have Matcher* infere que o objeto `bag` contém uma coleção e é sobre essa coleção que se quer verificar o tamanho;
- 3) finalmente, o *Have Matcher* verifica o tamanho dessa coleção de acordo com o tamanho passado na expectation. Como no caso acima o tamanho passado para o matcher foi `2 (have(2).words)`, ele executa `bag.words.size == 2`.

O *Have Matcher* precisa fazer toda a lógica acima porque ele serve para testar o tamanho de uma coleção em dois cenários:

- quando você quer verificar o tamanho de uma coleção contida por um objeto, como no caso acima;
- quando você quer verificar o tamanho de uma coleção diretamente.

Abaixo segue um exemplo de uso desse matcher para verificar o tamanho de uma coleção diretamente:

```
collection = [1, 2, 3]
```

```
expect(collection).to have(3).items
```

Um outro detalhe interessante sobre esse matcher é que quando você está fazendo um teste de uma coleção diretamente, você pode passar qualquer método para ele:

```
collection = [1, 2, 3]
```

```
# todos os modos abaixo funcionam
expect(collection).to have(3).items
expect(collection).to have(3).children
expect(collection).to have(3).things
```

Apesar de qualquer método que você passar para o *Have Matcher* funcionar quando o objeto sendo testado é uma coleção, é importante que você escolha o método que melhor represente a semântica do domínio do seu problema.

Por exemplo, se você estiver fazendo alguma aplicação que é sobre processamento de texto, faz muito mais sentido você escrever um teste como:

```
words = "hugo"

expect(words).to have(4).letters # letters é um termo do domínio
                                # do problema
```

do que um teste como:

```
words = "hugo"

expect(words).to have(4).items # itens não é um termo do domínio
                               # do problema
```

Escolher o método correto como mostramos acima tem duas vantagens. A primeira é que ao escrever código de teste que melhor representa o seu domínio, você está seguindo a prática de “linguagem ubíqua” do Domain Driven Design - DDD, deixando seu teste mais parecido com uma documentação e também facilitando a criação de um dicionário de termos padrão para o seu projeto.

A segunda vantagem pode ser vista pela variação entre a mensagem de erro no teste. Ela é sempre um ótimo indício se você está no caminho de escrever um bom teste.

Por exemplo, o teste a seguir quebra com a seguinte mensagem:

```
words = "hugo"

expect(words).to have(3).items

# Failure/Error: expect(words).to have(3).items
#           expected 3 items, got 4
```

Já o mesmo teste escrito de um modo melhor, quebra com a seguinte mensagem:

```
words = "hugo"

expect(words).to have(3).letters
```



```
# Failure/Error: expect(words).to have(3).letters
#           expected 3 letters, got 4
```

Perceba como a mensagem de erro ficou melhor quando usamos um método mais apropriado para descrever a intenção do nosso teste.

Por fim, você pode usar mais duas formas desse matcher, que são as formas `have_at_least` e `have_at_most`. Segue um exemplo usando essas duas formas:

```
collection = [1, 2, 3, 4, 5]

expect(collection).to have_at_least(3).elements
expect(collection).to have_at_most(5).elements
```

Matcher para verificar mudança de estado ou valor

Imagine que você está desenvolvendo um jogo e precise criar um objeto que tem 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:

```
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
```

Ne exemplo acima 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 acima, 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 vem com o RSpec. Mas, como são muitos e muitas variações, não vale a pena ver todos os detalhes de todos os matchers que vem 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://www.relishapp.com/rspec/rspec-expectations/v/2-14/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/v2.13.0/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 vem com o RSpec, descanse um pouquinho, pois a seguir você irá aprender a escrever os seus próprios matchers!

3.7 CUSTOM MATCHERS

Na seção anterior nós aprendemos sobre os matchers padrão do RSpec e vimos que existem muitos deles. Mas mesmo com o RSpec nos oferecendo tantos matchers, as vezes podemos precisar de mais. As vezes 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 o escopo acima. 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
```

```
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 esse teste do seguinte modo:

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

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

```
describe Category do  
  it "contains all the products of its subcategories" do  
    eletronics = Category.new("eletronics")  
    computers = Subcategory.new("computers")  
    cell_phones = Subcategory.new("cell phones")  
    computers.add_product("MacBook")  
    cell_phones.add_product("iPhone")  
  end  
end
```

Agora podemos adicionar essas subcategorias a categoria "eletronics" e verificar que os produtos delas estão contidos na categoria "eletronics":

```

describe Category do
  it "contains all the products of its subcategories" do
    eletronics = Category.new("eletronics")
    computers  = Subcategory.new("computers")
    cell_phones = Subcategory.new("cell phones")
    computers.add_product("MacBook")
    cell_phones.add_product("iPhone")

    eletronics.add_subcategories(computers, cell_phones)

    eletronics_products = eletronics.subcategories.map { |sub|
      sub.products
    }
    eletronics_products.flatten!
    expect(eletronics_products).to include("MacBook", "iPhone")
  end
end

```

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

```

eletronics_products = eletronics.subcategories.map { |sub|
  sub.products
}
eletronics_products.flatten!
expect(eletronics_products).to include("MacBook", "iPhone")

```

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

O segundo ponto a melhorar 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(eletronics_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 "eletronics" não contém nenhum dos produtos que ela deveria conter. Lembre-se que a mensagem de erro de um teste é muito importante, pois uma mensagem de erro clara pode nos ajudar a resolver mais rápido um teste que está quebrado.

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

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

Vamos aprender a 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 acima 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`. Esse argumento é o mesmo que é passado para o nosso matcher na hora 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 pra 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

describe "The be_a_multiple_of custom matcher" do
  it "can be used to verify 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:

```
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)
```

```

    eletronic_products = eletronic.subcategories.map { |sub|
      sub.products
    }
    eletronic_products.flatten!
    expect(eletronic_products).to include("MacBook", "iPhone")
  end
end

```

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

```

describe Category do
  it "contains all the products of its subcategories" do
    eletronic = Category.new("eletronic")
    computers = Subcategory.new("computers")
    cell_phones = Subcategory.new("cell phones")
    computers.add_product("MacBook")
    cell_phones.add_product("iPhone")

    eletronic.add_subcategories(computers, cell_phones)

    expect(eletronic).to contain_products("MacBook", "iPhone")
  end
end

```

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

```

RSpec::Matchers.define :contain_products do |*products|
  match do |category|
    subcategories_products = category.subcategories.map { |sub|
      sub.products
    }
    subcategories_products.flatten!

    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:

```

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

```



```

eletronics = Category.new("eletronics")
computers  = Subcategory.new("computers")
cell_phones = Subcategory.new("cell phones")
computers.add_product("MacBook")
cell_phones.add_product("iPhone")

eletronics.add_subcategories(computers, cell_phones)

expect(eletronics).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 esse teste e ver a seguinte mensagem:

Failures:

```

1) Category contains all the products of its subcategories
   Failure/Error: expect(eletronics).to
     contain_products("MacBook", "iPhone")

     expected #<Category:0x007fcb1c0fc460 @name="eletronics",
       @subcategories=[]>
       to contain products "MacBook" and "iPhone"

```

Na mensagem de erro já está mais claro que o teste falhou porque a categoria `eletronics` não contém os produtos "MacBook" e "iPhone", mas ainda podemos melhorar essa mensagem ainda mais.

A mensagem de erro acima 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_for_should` da DSL de custom matcher do RSpec:

```

failure_message_for_should do |category|
  "expected category #{category.name} to contain products #{products}"
end

```

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

```
RSpec::Matchers.define :contain_products do |*products|
  match do |category|
    subcategories_products = category.subcategories.map { |sub|
      sub.products
    }
    subcategories_products.flatten!

    expect(subcategories_products & products).to eq products
  end

  failure_message_for_should do |category|
    "expected category #{category.name} to contain products #{products}"
  end
end
```

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

Failures:

- 1) Category contains all the products of its subcategories

```
Failure/Error: expect(eletronics).to
  contain_products("MacBook", "iPhone")
```

```
expected category eletronics 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://www.relishapp.com/rspec/rspec-expectations/v/2-14/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 pro RSpec e descobrir o protocolo de matchers por trás disso tudo.

3.8 ENTENDENDO O PROTOCOLO INTERNO DE MATCHER DO RSpec

Na seção anterior nós aprendemos a construir nossos próprios matchers pro RSpec usando a DSL de custom matchers. Essa DSL server para facilitar a criação de novos

matchers, mas ela não é o único modo de criar um matcher novo. Você pode criar seu próprio matcher se entender o protocolo por trás da relação entre uma expectation e o seu matcher.

Como vimos na seção 3.1, 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 de código acima 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*.

O protocolo mínimo de um matcher é que ele responda para o método `matches?(actual)`, onde `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 utilizar esse matcher, poderíamos escrever um teste do seguinte modo:

```
class SimplestMatcher
  def matches?(actual)
    true
  end
end

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

describe 'The matcher procotol' do
  it 'has a #matches?(actual) method' do
    expect('anything').to simple_matcher
  end

  def simplest_matcher
    SimplestMatcher.new
  end
end
```

Essa idéia 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 seguinte código:

```
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*. Esse matcher tem dentro dele uma implementa-

ção 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, se ele retornar `true` é porque a `expectation` passou, se ele retornar `false`, é porque ela quebrou. Simples assim.

A real importância de conhecer esse contrato é não 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!”.

3.9 PONTOS-CHAVE 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` para nos possibilitar a escrever testes que pareçam com documentação, que estejam próximos da semântica do domínio do problema sendo resolvido e para gerar 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 existe 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.

Neste capítulo 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 só isso no próximo capítulo.

CAPÍTULO 4

Organização, refatoração e reuso de testes com o RSpec

Neste capítulo você irá aprender a como organizar seus testes. Irá aprender que um teste de unidade tem uma narrativa e qual a vantagem de segui-la. Irá aprender também a como refatorar seus testes para ficarem menos repetitivos, mas sem perder a clareza dos mesmos.

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 o 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:

```
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 continuarmos falando sobre os testes em si, vamos analisar algumas novidades nele.

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

```
describe Game, "in the final phase"
```

Como vimos no capítulo 3, 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 que vimos, estamos agrupando nossos testes por classe e um determinado estado do objeto dessa classe, no caso, o estado em que o `game` está na última fase. Uma dos benefícios de 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`:

```
describe Game, "in the final phase" do
  context "when the player hits the target" do
```

E agora usando o `describe`:

```
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"

  # (...)
end
```

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 parabéns para o jogador:

```
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
```

Beleza, esse teste já está pronto. Vamos pular a parte de escrever o código para fazer esse teste passar, porque não é o foco agora, vamos escrever o próximo teste.

O próximo teste é o:

```
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 a do teste 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, onde 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:

```
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("Congratuations!")
    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 os testes. Assim como você segue o DRY (*don't repeat yourself*) no seu código de produção, você também pode usar a mesma idéia 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 uma vez só antes de todos os testes
end
```

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

```
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("Congratuations!")
    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 irão 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 de 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("Congratuations!")
  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("Congratuations!")
  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, ou seja, 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 acima. 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 irá impactar na clareza do seu teste. No exemplo que demos acima ela não ficou 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("Congratuations!")
  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 o score ser setado como 100?
- o que tem que ser feito no objeto game para que ele sete o score para 100?

Essas perguntas são válidas para ser possível entender a relação de causa e consequência de um teste. Se o teste estivesse sido 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, o teste não perde muito em clareza. No entanto, se o teste 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 o seu teste 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. Ou seja, 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 que em código de teste, uma das características mais importantes é a clareza e a possibilidade de entender fácil 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 te 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 pode ser útil o *after hook* é quando é necessário deletar arquivos gerados durante os testes.

Imagine que você está escrevendo um teste de um objeto que salva um cache em um arquivo 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 rodar esse teste, 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:

```
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 o teste acima 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 acima poderia ter sido feito tranquilamente usando *before* e *after* hooks:

```
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* ao invés 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

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 acima, 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 entender os seus testes, 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:

- 1) **setup**: onde você coloca o SUT (*system under test*, por exemplo o objeto sendo testado) no estado necessário para o teste;
- 2) **exercise**: onde você interage com o SUT;
- 3) **verify**: onde você verifica o comportamento esperado;
- 4) **teardown**: onde você coloca o sistema no estado que ele estava antes do 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 2:

```
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 o teste acima e tente identificar as 4 fases. Eu pessoalmente gosto de deixar uma quebra de linha entre cada fase do teste, de modo que fique fácil entender a estrutura do mesmo só batendo o olho.

Para ficar explícito, segue o código do teste acima com um comentário para cada fase:

```
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 teste 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 com change matcher
end

```

Só para ficar claro, não é necessário você escrever um comentário explicando cada fase do teste, mas colocar uma quebra de linha entre cada fase irá 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 irá 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 sinta que lá vem história.

Como vimos na seção 2.3, 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. Ao invés 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 as 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 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 vem os nomes `Example` e `ExampleGroup` que o `RSpec` utiliza normalmente, vamos olhar uma convenção de como organizar example groups dos seus testes.

Na seção 3.1 nós escrevemos um teste de pilha do seguinte modo:

```
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`:

```
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 class dentro de um example group com o nome desse método:

```
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 esse leitor estiver querendo entender o comportamento de um método em específico.

NOTAÇÃO	CLASS#INSTANCE_METHOD	E
CLASS.CLASS_METHOD		

Em Ruby, existe uma notação muito usada para se referenciar a um método de instância ou de classe, que é a notação `Class#instance_method` e `Class.class_method`.

Para métodos de instância você se 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 se 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 `initialize`, podemos nos referenciar a ele com a notação `Stack.initialize`.

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 definir o **SUT** (*system under test*, o objeto sendo testado). Para entendermos como ele pode ser usado, segue um exemplo de código com o seu uso:

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



```

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

it "should have the prescribed elements" do
  array.should == [1,2,3]
end
end

```

No teste acima 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 re-escrever o código acima usando um `before` hook:

```

describe Array, "with some elements" do
  before do
    @array = [1,2,3]
  end

  it "should have the prescribed elements" do
    @array.should == [1,2,3]
  end
end

```

No entanto, o uso do método `subject` no caso acima 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 do teste como um todo, ou seja, o `before` hook 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:

```

describe Game do
  let(:ui) { TwitterUi.new("sandbox_username", "sandbox_password") }
end

```

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

```

describe Game do
  before do

```

```
@game = Game.new
end

let(:ui) { TwitterUi.new('sandbox_username', 'sandbox') }

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. Ou seja, 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:

```
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 como usá-los para organizar nossos testes.

Como você deve saber, em programação orientada a objetos, para que um objeto possa realizar sua responsabilidade, ele as vezes depende de outros objetos. Esses objetos dos quais ele depende são chamados seus 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 objetos das quais ela depende como parâmetros. Essa prática se chama injeção de dependência, nós veremos um exemplo concreto dela no capítulo 5.

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, então 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 entender 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 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):

```
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 acima e escrever esse teste assim:

```
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("Congratuations!")
  end
end
```

```
end  
end
```

Perceba que para acessar o `subject` basta chamarmos o método que foi definido por ele. No caso acima, 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 organizar seu teste. 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 REUSO DE TESTES

Agora que já vimos sobre os RSpec hooks, sobre como refatorar nossos testes usando os eles e também sobre como organizar nossos testes, 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 onde que 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 data de publicação dele. Sabendo disso, você começa e escrever a seguinte spec para a sua classe `BlogPost`:

```
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 que o paper foi publicado. Como um bom praticante de TDD, você escreve o seguinte teste para a classe `Paper`:

```
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 seguinte código:

```
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 conte-nha as regras de negócio de um objeto publicável. Esse módulo poderia se chamar `Publishable`. A idéia 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

describe "A published 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 o teste acima. Primeiro nós defini-mos uma classe só para esse teste, a `PublishableObject`, que inclui o módulo

que queremos testar. Depois disso, nós 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 você chamar o método `subject` de dentro do teste, como fizemos acima:

```
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 fazer esse teste 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 acima para o módulo `Publishable` em *shared examples*. Para definir um conjunto de *shared examples* você deve usar o método `shared_examples_for` e passar para ele uma string nomeando esse conjunto de *examples*:

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

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

```
shared_examples_for "a publishable object" do
  describe "#publish" do
    it "saves the date when the object is published" 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 logo vamos ver a diferença entre eles. Agora vamos re-escrever os testes do módulo `Publishable` utilizando os *shared examples* que criamos acima:

```
class PublishableObject
  include Publishable
end

describe "A published object" do
  subject { PublishableObject.new }

  include_examples "a publishable object"
end
```

Pronto, isso é tudo 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:

```
shared_examples_for "a publishable object" do
  describe "#publish" do
    it "saves the date when the object is published" do
```



```

    subject.publish! # uso do subject definido pelo teste que incluir
                     # estes shared examples

    # (...)
  end
end
end

```

Na verdade, os *shared examples* tem 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`.

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

```

describe BlogPost do
  it_behaves_like "a publishable object"
end

describe Paper do
  it_behaves_like "a publishable object"
end

```

Perceba que para os testes das classes acima 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 faz isso para

você automaticamente. Ele define o `subject` automaticamente quando o argumento passado para o método `describe` é uma classe. Então quando escrevemos o seguinte código:

```
describe BlogPost do
end
```

O RSpec na verdade define o `subject` de modo implícito do seguinte modo:

```
describe BlogPost do
  subject { BlogPost.new }
end
```

Apesar do RSpec definir o `subject` de modo implícito, sempre dê preferência para definir o `subject` de modo explícito e dando um nome para ele. Ao fazer desse modo você deixará seu teste mais claro.

Por fim, vamos rodar os testes das classes `BlogPost` e `Paper` 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
```

```
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`.

Bom, agora você já sabe como reutilizar testes utilizando a funcionalidade de *shared examples* do RSpec. Essa é uma funcionalidade bem poderosa e pode te ajudar bastante a reduzir testes repetitivos.

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

4.7 PONTOS-CHAVE 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. Algo que você precisa lembrar disso é sempre pesar a redução de duplicação versus a perda de clareza nos seus testes.

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*. Seguir uma narrativa no seu teste facilita a compreensão do mesmo.

Por exemplo, você lembra da estrutura de redação dissertativa que você aprendeu na escola? A estrutura era introdução, desenvolvimento e conclusão. O equivalente dessa estrutura de redação para um teste são as quatro fases do xUnit. Siga a estrutura das quatro fases e o seu teste ficará 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 na clareza do mesmo.

Por fim, vimos sobre como utilizar os `shared examples` do RSpec para reutilizar testes. A dica aqui é 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 iremos aprender sobre RSpec Mocks, sobre o que são *test doubles* e que novidades o uso desse tipo de ferramenta trás para o modo como escrevemos testes.

Não saia daí!

CAPÍTULO 5

TDD na prática, começando um projeto com TDD

Não é segredo que para desenvolver uma habilidade o mais importante é praticar, praticar e praticar! Com TDD 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 o a 100 fazendo TDD com Cucumber e RSpec.

Neste capítulo iremos 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 TDD com Cucumber e RSpec, o famoso *outside-in development*. Ou seja, começar o teste com uma especificação de Cucumber, depois fazer testes de RSpec, depois fazer o RSpec passar e por fim fazer o Cucumber passar.

Por fim, 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.

5.1 DEFININDO O ESCOPO DA NOSSA APLICAÇÃO: JOGO DA FORCA

A aplicação que iremos 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?

O jogo da forca é originalmente um jogo de papel e caneta que funciona do seguinte modo:

- 1) Uma palavra secreta é definida
- 2) Você tenta adivinhar uma letra da palavra
- 3) Se você acertar a letra, é mostrada a posição dessa letra dentro da palavra secreta
- 4) Se você errar a letra, uma parte do boneco aparece na forca
- 5) Você ganha se acertar todas as letras
- 6) Você perde se errar o bastante para o corpo inteiro do boneco aparecer na forca

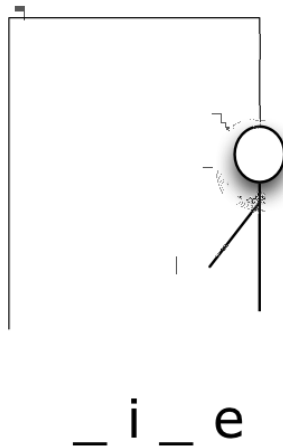


Figura 5.1: Jogo da forca com duas letras adivinhadas e três erros cometidos

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 pro 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.

Baseado nessa lista de funcionalidades, já podemos começar a desenvolver nosso projeto.

5.2 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 iremos 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`:

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

ruby "1.9.3"

group "test" do
  gem "cucumber"
  gem "rspec"
end
```

E 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 rodá-lo pela primeira vez:

```
$ bundle exec cucumber
You don't have a 'features' directory. Please create one to get started.
See http://cukes.info/ for more information.
```

Após rodar o comando, ele nos avisa que é necessário criar o diretório `features`. Como vimos no capítulo ??, é nesse diretório que fica o conjunto de especificações da nossa aplicação. Então, vamos 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 o seguinte nele:

```
# 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)
0m0.002s
```

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

```
Quando /^começo um novo jogo$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Então /^veja na tela:$/ do |string|
  pending # express the regexp above with the code you wish you had
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 # express the regexp above with the code you wish you had
end
```

```
Então /^veja na tela:$/ do |string|
  pending # express the regexp above with the code you wish you had
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:

```
# encoding: UTF-8
```

```
Quando /^começo um novo jogo$/ do
  pending
end
```

```
Então /^veja 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)
0m0.013s
```

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 deles serem implementados. Isso não é um problema, faz parte do fluxo do TDD usar um código que você gostaria que existisse e deixar os testes nos guiar sobre quando devemos implementar esse código.

Seguindo o fluxo de TDD, 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:4:in
    `/^começo um novo jogo$/'
  features/comecar_jogo.feature:9:in `Quando começo um novo jogo'
```

O feedback que recebemos do teste é que a constante `Game` ainda não foi definida. Vamos então defini-la criando a classe `Game`. Crie o diretório `lib` e dentro dele crie o arquivo `lib/game.rb` com o seguinte conteúdo:

```
# encoding: UTF-8
```

```
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 File.join(File.dirname(__FILE__), "..", "..", "lib", "game")
```

Agora, rodando o Cucumber, recebemos a seguinte mensagem:

```
$ bundle exec cucumber
```

```
Quando começo um novo jogo  
  undefined method `start' for #<Game:0x007f93c911bdb8> (NoMethodError)  
  ./features/step_definitions/game_steps.rb:5:in  
    `/~começo um novo jogo$/'  
  features/comecar_jogo.feature:9:in `Quando começo um novo jogo'
```

O erro mudou, isso é bom porque é um indicador 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:

- 1) Implementarmos o método `start` na classe `Game`
- 2) Escrevermos um teste para especificar o método `start` antes de começar a implementá-lo

Como bons praticantes de TDD (e também porque você está lendo um livro sobre TDD), vamos escolher a segunda opção: escrever um teste antes de começar a implementar. Vamos usar o RSpec para escrever esse teste.

5.3 USANDO RSpec NO NOSSO PRIMEIRO TESTE

Antes de começar a usar o RSpec, precisamos inicializá-lo no nosso projeto, executando o seguinte comando no terminal:

```
$ bundle exec rspec --init
```

Após inicializar o RSpec, crie o diretório `spec` e 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 impresso 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:

```
# encoding: UTF-8

require 'spec_helper'
require 'game'

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!"
      STDOUT.should 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:0x007fd44b8f5f08>
# ./spec/game_spec.rb:10:in `block (3 levels) in <top (required)>'

Finished in 0.00036 seconds
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`. Então, 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: STDOUT.should 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. Então, 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 que alguma coisa pode ser melhorada. Então, 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 essa dependência 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` ao invés de somente `puts`.

Agora abra o arquivo `game_spec.rb` e o modifique para ficar assim:

```
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!"
      output.should_receive(:puts).with(initial_message)
```

```
      game.start
    end
  end
end
```

Finalmente, ao rodar os testes, vemos que eles passam!

```
$ bundle exec rspec
```

```
.
```

```
Finished in 0.00055 seconds
1 example, 0 failures
```

Após comemorar que nosso primeiro teste passou (\o/), 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 objeto, desde que esse objeto “se pareça” com um objeto que imprime strings. E foi o que que fizemos, nos valem 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)

  output.should_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 [7] e também no livro em português *Test-Driven Development: Teste e Design no Mundo Real*, escrito pelo Mauricio Aniche [1].

Por fim, você pode rodar o Cucumber novamente e ver que o primeiro step, que antes não passava, pois faltava implementar o método `start`, agora passa:

```
$ bundle exec cucumber
(...)

1 scenario (1 pending)
2 steps (1 pending, 1 passed)
0m0.015s
```

Estamos prontos para ir pro segundo step!

5.4 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!
  """
```

Ou seja, precisamos testar que após o jogador iniciar o jogo, essa mensagem inicial é impressa na tela.

No ambiente de produção, o jogo irá 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` iremos fazer 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 continuar escrevendo 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

$: .unshift File.join(File.dirname(__FILE__), "..", "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 ele interaja com o jogo do mesmo modo que o jogador interagiria. Para implementar *step definitions* que interagem com uma aplicação de linha de comando, utilizaremos uma gem chamada `Aruba`.

O `Aruba` nada mais é do que 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 irá fazer pra gente.

Para instalar o `Aruba`, adicione ele no `Gemfile`:

```
group "test" do
  gem "cucumber"
  gem "rspec"
  gem "aruba"
end
```

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`. Então abra o arquivo `features/support/env.rb` e o edite para adicionar esse `require`:

```
require File.join(File.dirname(__FILE__), "..", "..", "lib", "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\$/” 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 *forca*.

Para refatorar esse step definition a fim de usar o Aruba, abra o arquivo `features/step_definitions/game_steps.rb` e edite o *step definition* de começo de novo jogo para ficar assim:

```
Quando /^começo um novo jogo$/ do
  steps %{
```

```

    When I run `forca` interactively
  }
end

```

Para começar um jogo executando o binário *forca*, usamos o step definition “When I run 'command' interactively” do Aruba.

PARA USUÁRIOS DE WINDOWS

No windows você deve fazer com que o Cucumber rode o binário *forca.bat*, não o *forca*, portanto mude o step definition acima para:

```

Quando /^começo um novo jogo$/ do
  steps %{
    When I run `forca.bat` interactively
  }
end

```

Lembre-se de sempre usar *forca.bat* quando for referenciar o binário *forca* nos seus step definitions que usam 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 /^veja na tela:$/ do |text|
  steps %{
    Then the stdout should contain "#{text}"
  }
end

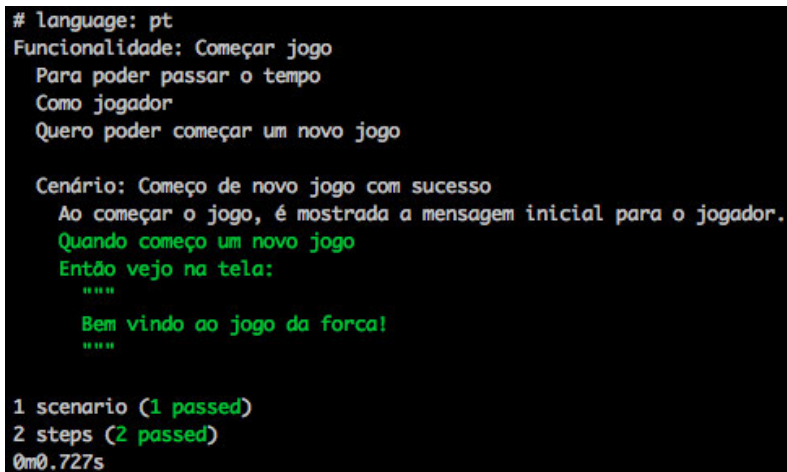
```

```
}  
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 vejo na tela:  
    """  
    Bem vindo ao jogo da forca!  
    """  
  
1 scenario (1 passed)  
2 steps (2 passed)  
0m0.727s
```

Figura 5.2: Primeiro cenário no verde

Finalmente, conseguimos terminar o primeiro cenário inteiro!

No capítulo seguinte continuaremos o desenvolvimento dessa primeira funcionalidade. Mas antes disso, vale a pena lembrarmos dos pontos-chave que acabamos de aprender.

5.5 PONTOS-CHAVE 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 força a deixar explícito 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.

Por fim, vimos que ao usarmos o Cucumber como ferramenta de testes de aceitação, devemos fazer nossos testes enxergando o software sob teste como uma caixa preta. Para fazer testes com essa visão no caso de uma aplicação de linha de comando, usamos o Aruba, que é uma extensão do Cucumber para testar aplicações CLI.

CAPÍTULO 6

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 nós iremos continuar o desenvolvimento dessa funcionalidade, definindo o segundo cenário dela e fazendo as mudanças e refatorações necessárias para implementar esse segundo cenário.

Vamos lá!

6.1 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 a palavra a ser sorteada deverá ter "4" letras

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.

6.2 REDUZA DUPLICAÇÃO ATRAVÉS DE SUPPORT CODE

O primeiro step indefinido é o “Dado que comecei um jogo”. Esse step é muito parecido com o step “Quando começo um novo jogo” que implementamos no

capítulo anterior. Na verdade, nós queremos que ambos os steps façam a mesma coisa, ou seja, inicie um jogo.

Poderíamos ter reutilizado o mesmo step, mas não o fizemos de propósito. Fizemos assim 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, iremos 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
    steps %{
      When I run `forca` interactively
    }
  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 no capítulo ??.

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 a palavra a ser sorteada deverá ter 4 letras"* continua indefinido:

```
$ bundle exec cucumber --tags @wip

(...)

3 steps (1 skipped, 1 undefined, 1 passed)
```

Vamos implementar esse último step.

6.3 IMPLEMENTANDO O FLUXO DO JOGO NO BINÁRIO

O próximo step a ser implementado é o *"Quando escolho que a palavra a ser sorteada deverá ter 4 letras"*. 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:

```
Quando /^escolho que a palavra a ser sorteada deverá ter "(.*?)" letras\$/ do |number_of_letters|
  steps %{
    When I type "#{number_of_letters}"
  }
end
```

Ao rodarmos o Cucumber deste cenário, vemos o seguinte:

```
$ bundle exec cucumber --tags @wip
```

Então vejo na tela:

```
"" ""
- - - -
```

```

"""
expected "Qual o tamanho da palavra a ser sorteada?\n"
  to include "_ _ _ _"
Diff:
@@ -1,2 +1,2 @@
- _ _ _ _
+Qual o tamanho da palavra a ser sorteada?
  (RSpec::Expectations::ExpectationNotMetError)
features/comecar_jogo.feature:25:in `Então vejo na tela:'

Failing Scenarios:
cucumber features/comecar_jogo.feature:18

```

O cenário falhou, mas por quê? O motivo é que ainda precisamos construir o resto da funcionalidade. O que está faltando é:

- 1) o jogador poder escolher o tamanho da palavra a ser sorteada;
- 2) o jogo sorteia essa palavra;
- 3) 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

$: .unshift File.join(File.dirname(__FILE__), "..", "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 imagina que a classe `Game` terá dois novos métodos, `ended?` e `next_step`. O método `ended?` irá dizer se o jogo terminou ou não. E o método `next_step` vai executar o próximo passo do jogo, baseado no estado atual do mesmo.

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

$:.unshift File.join(File.dirname(__FILE__), "..", "lib")

require 'game'

game = Game.new
game.start

while not 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 num 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 pro 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 jogado 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!
```

```
bin/forca:10:in `<main>': undefined method `ended?' for
#<Game:0x007fe84890a208 @output=#<IO:<STDOUT>>> (NoMethodError)
```

Ou seja, 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/comecar_jogo.feature:26:in `Então vejo na tela:'
```

```
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!

6.4 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 `forca`, com a diferença que o Cucumber está fazendo isso via o Aruba.

Para que o teste com Cucumber tenha um resultado final mais parecido com rodar o jogo na mão, o que podemos fazer é checar se o binário rodou com sucesso, ou seja verificar o *exit status* do processo do jogo. O Aruba tem um step definition que verifica o *exit status* do processo sob teste, portanto, podemos usá-lo.

Vamos mudar nosso arquivo de feature `features/comecar_jogo.feature`, colocando um step que vai fazer isso. Iremos trocar o antigo step *"Então vejo na tela:"* por um novo step, o *"Então o jogo termina com a seguinte mensagem na tela:"*. O antigo step só verificava se uma dada string foi impressa na tela. Nesse novo step, além de verificar se a string foi impressa, verificaremos também se o processo do jogo terminou com sucesso, através da checagem do seu *exit status* com o Aruba.

Modifique os cenários do arquivo `features/comecar_jogo.feature` para utilizar esse novo step e ficar assim:

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 a palavra a ser sorteada deverá ter "4" letras

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:

```
# encoding: UTF-8

Dado /^que comecei um jogo$/ do
  start_new_game
end

Quando /^começo um novo jogo$/ do
  start_new_game
end

Quando /^escolho que a palavra a ser sorteada deverá ter "(.*?)" letras\
$/ do |number_of_letters|
  steps %{
    When I type "#{number_of_letters}"
  }
end

Então /^o jogo termina com a seguinte mensagem na tela:$/ do |text|
  steps %{
    Then it should pass with:
      """
      #{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 o step definition “Then it should pass with:” do Aruba. Esse step definition verifica se um texto foi impresso no `STDOUT` e também verifica se 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:

```
(...)
```

```
Exit status was 1 but expected it to be 0. Output:
```



```
forca/bin/forca:10:in `<main>':
undefined method `ended?' for #<Game: ...> (NoMethodError)
```

```
2 scenarios (2 failed)
```

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.

6.5 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:

```
it "returns false when the game just started" do
  game = Game.new
  game.should_not be_ended
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` dependendo se 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:

```
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!"
      output.should_receive(:puts).with(initial_message)

      game.start
    end
  end

  describe "#ended?" do
    it "returns false when the game just started" do
      game = Game.new
      game.should_not 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:

```
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!"
    output.should_receive(:puts).with(initial_message)

    game.start
  end
end

describe "#ended?" do
  it "returns false when the game just started" do
    game.should_not be_ended
  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:

```

(...)

Exit status was 1 but expected it to be 0. Output:

forca/bin/forca:11:in `<main>':
undefined method `next_step' for #<Game: ... > (NoMethodError)

2 scenarios (2 failed)

```

Pela mensagem de erro, podemos ver que falta implementarmos o método `Game#next_step`, vamos escrever um teste para esse método e implementá-lo.

O primeiro teste que iremos escrever é pro cenário onde o jogo acabou de começar. Nesse contexto, o método `Game#next_step` deve pedir pro jogador o tamanho da palavra a ser sorteada. Para fazer a verificação desse teste, basta checarmos se o

objeto `game` irá pedir pro seu colaborador `output` para imprimir a pergunta pro 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?"
      output.should_receive(:puts).with(question)

      game.next_step
    end
  end
end
```

Vamos fazer esse teste 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. Ou seja, podemos voltar para o Cucumber.

6.6 REFATORANDO O CÓDIGO PARA PODER IMPLEMENTAR O SEGUNDO CENÁRIO

Ao rodarmos o Cucumber, vemos o seguinte:

```
$ bundle exec cucumber
```

```
(...)
```

```
process still alive after 3 seconds (ChildProcess::TimeoutError)
```

```
2 scenarios (2 failed)
```

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 porque em nenhum momento o método `Game#ended?` está retornando `true`.

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 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 adicionar 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 a palavra a ser sorteada deverá ter `"4"` letras

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
  steps %{
    When I 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, onde ele pode 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?"
      output.should_receive(:puts).with(question)

      game.next_step
    end
  end
end

```

Esse teste verifica se o jogo pergunta pro jogador o tamanho da palavra a ser sorteada, quando o método `Game#next_step` é executado e o jogo acabou de começar. Precisamos modificar esse teste para fazer mais do que isso, ele precisa verificar também se o jogo está lendo o tamanho da palavra que o jogador irá digitar. Para ler a entrada do jogador, podemos criar um novo colaborador que terá essa responsabilidade, vamos chamá-lo de `input`. Vamos então mudar o teste acima para verificar se o objeto `game` pede para o objeto `input` ler a entrada do jogador:

```

1 describe Game do
2   let(:output) { double("output") }
3   let(:input)  { double("input") }

```

```

4
5  subject(:game) { Game.new(output, input) }
6
7  # (...)
8
9  describe "#next_step" do
10    context "when the game just started" do
11      it "asks the player for the length of the word to be raffled" do
12        question = "Qual o tamanho da palavra a ser sorteada?"
13        output.should_receive(:puts).with(question)
14
15        input.should_receive(:gets)
16
17        game.next_step
18      end
19    end
20  end
21 end

```

Para fazer esse teste passar, podemos injetar a dependência `input`, a qual a 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. Vamos aproveitar que os testes estão passando e vamos refatorar.

6.7 EXTRAINDO 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 a trocar a criação de `input` e `output` por `ui`. Até então, essa parte do teste está assim:

```
describe Game do
  let(:output) { double("output") }
  let(:input)  { double("input") }

  subject(:game) { Game.new(output, input) }

  # (...)
end
```

Vamos modificá-la para ficar assim:

```
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 chamados 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!"
  output.should_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?"
  output.should_receive(:puts).with(question)

  input.should_receive(:gets)

  game.next_step
end
```

Ou seja, onde antes estava “`output.should_receive(:puts)`” ou “`input.should_receive(:gets)`”, precisamos mudar para “`ui.should_receive(:puts)`” e “`ui.should_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, onde antes era “`input.should_receive(:gets)`” vai ficar “`ui.should_receive(:read)`” e onde antes era “`output.should_receive(:puts)`” vai ficar “`ui.should_receive(:write)`”.

Logo, os trechos de código anteriores ficarão assim:

```
it "prints the initial message" do
  initial_message = "Bem vindo ao jogo da forca!"
  ui.should_receive(:write).with(initial_message)

  game.start
end

e assim:

it "asks the player for the length of the word to be raffled" do
  question = "Qual o tamanho da palavra a ser sorteada?"
  ui.should_receive(:write).with(question)

  ui.should_receive(:read)

  game.next_step
end
```

Por fim, nosso teste inteiro ficará assim:

```
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!"
      ui.should_receive(:write).with(initial_message)

      game.start
    end
  end

  describe "#ended?" do
    it "returns false when the game just started" do
      game.should_not 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?"
      ui.should_receive(:write).with(question)

      ui.should_receive(:read)

      game.next_step
    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`:

```
# encoding: UTF-8

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.

6.8 POSSIBILITANDO AO JOGADOR TERMINAR O JOGO NO MEIO

Com os testes passando, podemos voltar para a ideia original, que era a possibilidade do jogador terminar o jogo no meio digitando *"fim"*. Para isso, adicione o seguinte teste no arquivo `lib/game.rb`:

```
describe "#next_step" do
  # (...)

  it "finishes the game when the player asks to" do
    player_input = "fim"
```

```

    ui.stub(read: player_input)

    game.next_step

    game.should 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: game.next_step
     Double "ui" received unexpected message :write with
       ("Qual o tamanho da palavra a ser sorteada?")
     # ./lib/game.rb:21:in `next_step'
     # ./spec/game_spec.rb:42

```

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 `ui.stub(read: player_input)`. Fizemos desse modo, pois essa é a única parte da interação com o `ui` que nos importa nesse teste.

O que queremos é ignorar o resto das mensagens enviadas para o double `ui`. Podemos fazer isso usando a feature de `as_null_object` do RSpec. Edite a definição da dependência `ui` no arquivo `spec/game_spec.rb` para ficar assim:

```

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: game.should be_ended

```

```
expected ended? to return true, got false
# ./spec/game_spec.rb:44
```

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)
```

Ou seja, agora finalmente podemos voltar a trabalhar no cenário que definimos no começo deste capítulo, o cenário *"Sorteio da palavra com sucesso"*. Mas vamos fazer isso só no próximo capítulo.

6.9 PONTOS-CHAVE DESTE CAPÍTULO

Nesse 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`, extraindo 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 7

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 nesse novo cenário.

Neste capítulo, iremos implementar esse e o terceiro cenário, finalizando assim essa funcionalidade. Ao final do capítulo o jogador já vai poder iniciar um jogo e sortear uma palavra para ele adivinhar.

Sigam-me os bons!

7.1 CONTINUANDO O SEGUNDO CENÁRIO

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 a palavra a ser sorteada deverá ter "4" letras
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)
```

Ou seja, 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!

7.2 DEIXANDO O SEGUNDO CENÁRIO NO VERDE

Para fazer esse segundo 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 passo seguinte após 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 inputa o tamanho da palavra como "3", então a palavra sorteada do 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"
      ui.stub(read: word_length)

      game.next_step

      game.raffled_word.should have(word_length).letters
    end

    it "prints a '_' for each letter in the raffled word"
  end
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: game.raffled_word.should have(word_length).letters

NoMethodError:

undefined method `raffled_word' for #<Game:0x007fc3facf85d0>

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 seguinte mensagem de erro:

```
$ 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: game.raffled_word.should have(word_length).letters
```

```
NoMethodError:
```

```
undefined method `letters' for nil:NilClass
```

Ou seja, 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 no código acima: se o adivinhar_letra.feature não digitou "fim", então ele digitou o tamanho da palavra a

ser sorteada. Após isso, usamos esse input para sortear uma palavra e salvar ela 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 claro a intenção dele.

Ao ler o trecho de código a seguir:

```
word_length = player_input.to_i
words = %w(hi mom game fruit)
@raffled_word = words.detect { |word| word.length == word_length }
```

Podemos pensar um pouco e entender que o que ele está fazendo é sortear a palavra. Mas a intenção desse código (o **que** ele faz, e não **como** ele faz) poderia estar mais explícita para quem o lerá. Para deixar a intenção desse trecho de código mais explícita, vamos refatorá-lo, 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, refatoramos sem quebrar nada. Vamos para o próximo spec pendente, que é o *it “prints a ‘_’ for each letter in the raffled word”*.

Esse teste irá 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"
```

```

ui.stub(read: word_length)

ui.should_receive(:write).with("_ _ _")

game.next_step
end

```

Ao rodar o RSpec, ela 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: ui.should_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 a chamada desse método, nosso código ficará assim:

```

1 def next_step
2   @ui.write("Qual o tamanho da palavra a ser sorteada?")
3   player_input = @ui.read.strip
4
5   if player_input == "fim"
6     @ended = true
7   else
8     raffle_word(player_input.to_i)
9     print_letters_feedback
10  end
11 end

```

O método `print_letters_feedback` será um método privado e a implementação dele ficará assim:

```

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"*, 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: game.next_step
NoMethodError:
  undefined method `length' for nil:NilClass

# ./lib/game.rb:43:in `print_letters_feedback'
# ./lib/game.rb:30:in `next_step'
# ./spec/game_spec.rb:34:

```

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 porque 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?"
      ui.should_receive(:write).with(question)

      ui.should_receive(:read)

      game.next_step
    end
  end
end

# (...)
end

```

Lendo o teste, vemos que em nenhum momento é setado o que o método `ui.read` irá 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 `ui.should_receive(:read)` para `ui.should_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?"
ui.should_receive(:write).with(question)

word_length = "3"
ui.should_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.

7.3 FINALIZANDO A PRIMEIRA FUNCIONALIDADE

A primeira funcionalidade do nosso jogo é a “Começar jogo”, que consiste nos seguintes cenários:

- 1) Começo de novo jogo com sucesso;
- 2) Sorteio da palavra com sucesso;
- 3) 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 pro jogo sortear uma palavra com um tamanho
  que o jogo não tem disponível, o jogador deve ser avisado disso
  e o jogo deve pedir pro jogador sortear outra palavra.
```



```

Dado que comecei um jogo
Quando escolho que a palavra a ser sorteada deverá ter "20" letras
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 fazer isso.

Vamos escrever um teste que vai verificar o cenário quando o jogador pede para ser sorteado uma palavra com um tamanho que o jogo não tem. Para implementar esse teste, 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 = "20"
      ui.stub(read: word_length)

      error_message = "Não temos uma palavra com o tamanho " <<
                      "desejado,\n" <<
                      "é necessário escolher outro tamanho."

      ui.should_receive(:write).with(error_message)

      game.next_step
    end
  end
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 se não for possível sortear uma palavra com o tamanho pedido, então uma mensagem de erro deve ser impressa para o jogador. Para fazer isso, basta checar o retorno do método `raffle_word`, se for `false`, então a mensagem de erro deve ser impressa pro jogador. 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 mal sinal, mostra que o código está complexo. Precisamos refatorar isso, mas por enquanto vamos deixar assim.

Com a implementação acima, 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. Iremos fazer isso só no próximo capítulo.

Enquanto você descansa um pouco e se prepara para o próximo capítulo, aproveite também para brincar com o que temos pronto até agora. Basta rodar o binário do jogo `bin/forca` e explorar a primeira funcionalidade.

7.4 PONTOS-CHAVE DESTE CAPÍTULO

Nesse 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.

Por fim, 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 esse e outros débitos técnicos que nosso código possa ter.

CAPÍTULO 8

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 nós 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 irá garantir que não iremos quebrar nenhuma funcionalidade enquanto melhoramos a qualidade de nosso código.

8.1 IDENTIFICADO 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 o entendimento do método e aumenta sua complexidade. Esse é um possível ponto a ser refatorado.

Outra 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!
  end
end

```

```
@ui.write(letters_feedback)
end
end
```

Métodos privados em excesso pode ser um indício que a classe está com mais de uma responsabilidade e que uma das suas responsabilidades está escondida de sua API pública através de métodos privados. Talvez esses métodos privados são 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 o nosso código é utilizado assim como seria utilizado por outra parte do nosso software, então no teste fica bem claro, por exemplo, 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 impresso a mensagem inicial para o jogador, ou seja, também está relacionado a interação com o usuário e ao fluxo do jogo. O `do ended?` verifica se o jogo acabou ou não, ou seja, 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`. Ou seja, 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 que a classe está implementando mais de uma responsabilidade.

Vamos começar a atacar cada um dos pontos levantados e refatorar onde for necessário.

8.2 EXTRAINDO UMA CLASSE DE UM MÉTODO PRIVADO

O primeiro ponto que iremos 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 [9]) está sendo quebrado. Por exemplo, imagine se o processo de sortear uma palavra não seja mais sortear 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 dele ser um método privado, não pudemos fazer testes de unidade para ele. Visto que

esse método 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 extrair esse método para sua própria classe. E é isso que vamos fazer.

Comece modificando uma parte dos testes da classe `game` para ficar assim:

```
1 describe Game do
2   let(:ui) { double("ui").as_null_object }
3   let(:word_raffler) { double("raffler").as_null_object }
4
5   subject(:game) { Game.new(ui, word_raffler) }
6
7   # (...)
8
9   describe "#next_step" do
10    # (...)
11
12    context "when the player asks to raffle a word" do
13      it "raffles a word with the given length" do
14        word_length = "3"
15        ui.stub(read: word_length)
16
17        word_raffler.should_receive(:raffle).with(word_length.to_i)
18
19        game.next_step
20      end
21
22      it "prints a '_' for each letter in the raffled word" do
23        word_length = "3"
24        ui.stub(read: word_length)
25        word_raffler.stub(raffle: "mom")
26
27        ui.should_receive(:write).with("_ _ _")
28
29        game.next_step
30      end
31
32      it "tells if it's not possible to raffle with the given length" do
33        word_length = "20"
34        ui.stub(read: word_length)
35        word_raffler.stub(raffle: nil)
36
```



```
37     error_message = "Não temos uma palavra com o tamanho " <<
38                     "desejado,\n" <<
39                     "é necessário escolher outro tamanho."
40
41     ui.should_receive(:write).with(error_message)
42
43     game.next_step
44   end
45 end
46
47 # (...)
48 end
49 end
```

Vamos entender o que fizemos no código acima. Primeiro, na linha 3 nós deixamos explícito que o objeto `game` tem um novo colaborador, o objeto `word_raffler`, que vai ter o papel de sorteador de palavras. Além de deixar dependência explícita, também a injetamos no objeto `game` na linha 5.

Após explicitar essa dependência, foi necessário identificar quais os testes que seriam afetados por essa refatoração e adaptá-los. O teste da linha 13 testa se uma palavra vai ser sorteada com o tamanho dado. Como ele é relacionado ao sorteio da palavra tivemos que adaptar esse teste. Para adaptá-lo, na linha 17 usamos mocks do RSpec para verificar se a mensagem certa estava sendo enviada para a nova dependência `word_raffler`.

O teste da linha 22 testa se foi impresso um “_” para cada letra da palavra sorteada. Como esse teste também está atrelado ao processo de sortear palavra, também tivemos que adaptá-lo. Na linha 25, foi necessário fazer um stub do método `word_raffler.raffle` para retornar a palavra “*mom*”, que seria a palavra sorteada. 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 só para setar o retorno de alguma dependência, já usar mocks serve para testar se a interação com uma dependência foi feita corretamente.

No teste da linha 32 é testado o comportamento de quando não é possível sortear uma palavra com o tamanho pedido. Para adaptá-lo, bastou fazer um stub do método `word_raffler.raffle` para retornar `nil`. Esse retorno de `nil` quer dizer que o `word_raffler` não pôde sortear uma palavra com o tamanho pedido. Ao fazermos esse stub estamos inclusive definindo o protocolo de comunicação entre o objeto `game` e o objeto `word_raffler`.

Após todas essas modificações nos testes, vamos rodá-los e ver o que precisaremos mudar:

```
$ bundle exec rspec
```

```
(...)
```

```
Failure/Error: subject(:game) { Game.new(ui, word_raffler) }
```

```
ArgumentError:
```

```
  wrong number of arguments (2 for 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:
```

```

word_raffler.should_receive(:raffle).with(word_length.to_i)

(Double "raffler").raffle(3)
  expected: 1 time
  received: 0 times
# ./spec/game_spec.rb:45

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 possa usar a nova dependência e não ter mais o método privado. Edite o método `Game#next_step` no arquivo `lib/game.rb` para ficar assim:

```

1 def next_step
2   @ui.write("Qual o tamanho da palavra a ser sorteada?")
3   player_input = @ui.read.strip
4
5   if player_input == "fim"
6     @ended = true
7   else
8     if @raffled_word = @word_raffler.raffle(player_input.to_i)
9       print_letters_feedback
10    else
11      error_message = "Não temos uma palavra com o tamanho " <<
12                    "desejado,\n" <<
13                    "é necessário escolher outro tamanho."
14
15      @ui.write(error_message)
16    end
17  end
18 end

```

Na linha 8 modificamos o código para usar o objeto `@word_raffler` ao invés de chamar o método privado. Aproveite e delete o antigo método privado.

Ao rodarmos o RSpec, podemos ver que está tudo no verde! Vamos rodar o Cucumber:

```
$ bundle exec cucumber
```

Exit status was 1 but expected it to be 0. Output:

```
forca/lib/game.rb:8:in `initialize':  
  
  uninitialized constant Game::WordRaffler (NameError)  
  
from forca/bin/forca:7:in `new'  
from forca/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 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` com os seguintes testes para essa nova classe:

```
# encoding: UTF-8  
  
require 'spec_helper'  
require 'word_raffler'  
  
describe WordRaffler do  
  it "raffles a word from a given list of words" do  
    words = %w(me you nice)  
    raffler = WordRaffler.new(words)  
  
    raffler.raffle(3).should == "you"  
    raffler.raffle(2).should == "me"  
    raffler.raffle(4).should == "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)
```

```

    raffler.raffle(20).should be_nil
  end
end

```

Para fazer o teste passar, crie um arquivo `lib/word_raffler.rb` com o seguinte conteúdo:

```

# encoding: UTF-8

# 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 a do antigo método privado `Game#raffle_word`. Ou seja, 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. Ao escrevermos esse texto, nos forçamos a pensar sobre as responsabilidades dessa classe e podemos verificar se ela está acumulando mais de uma.

A última coisa que falta fazer para terminar essa refatoração é dar um `require` da nova classe para a classe `Game`. Para fazer isso, adicione o `require_relative` `'word_raffler'` no começo da classe `Game` para ficar assim:

```

# encoding: UTF-8

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.

8.3 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 da classe `Game`. A ideia é que essa classe fique somente com a responsabilidade de guardar o estado do jogo e as regras de negócio dele de modo geral.

Para começar, vamos retirar da classe `Game` a lógica relacionado ao fluxo do jogo e interação com usuário. Comece criando um arquivo de teste chamado `spec/game_flow_spec.rb` e mova os testes do método `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 a pouco os testes e entender as modificações necessárias.

O primeiro teste que iremos mover será o do método `start`, que ficará assim no novo arquivo `spec/game_flow_spec.rb`

```
1 # encoding: UTF-8  
2  
3 require 'spec_helper'  
4 require 'game_flow'  
5
```

```

6 describe GameFlow do
7   let(:ui) { double("ui").as_null_object }
8   let(:game) { double("game").as_null_object }
9
10  subject(:game_flow) { GameFlow.new(game, ui) }
11
12  describe "#start" do
13    it "prints the initial message" do
14      initial_message = "Bem vindo ao jogo da forca!"
15      ui.should_receive(:write).with(initial_message)
16
17      game_flow.start
18    end
19  end
20
21  # (...)
22 end

```

Além de mover esse teste da classe `Game` para o teste da nova classe `GameFlow`, na linha 17 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` na linha 10, perceba que ele depende tanto de `game` quanto de `ui`. Isso porque a responsabilidade desse novo objeto é cuidar da interação como 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 vamos olhar como cada uma ficará. O primeiro teste ficará assim:

```

1 describe GameFlow do
2   # (...)
3
4   describe "#next_step" do
5     context "when the game just started" do
6       it "asks the player for the length of the word to be raffled" do
7         question = "Qual o tamanho da palavra a ser sorteada?"
8         ui.should_receive(:write).with(question)
9
10        word_length = "3"
11        ui.should_receive(:read).and_return(word_length)

```

```
12
13     game_flow.next_step
14   end
15 end
16
17   # (...)
18 end
19 end
```

Para esse teste, a única adaptação que tivemos que fazer foi na linha 13, onde mudamos de `game` para `game_flow`.

O teste seguinte que iremos mover ficará assim:

```
1 describe GameFlow do
2   # (...)
3
4   describe "#next_step" do
5     context "when the player asks to raffle a word" do
6       it "raffles a word with the given length" do
7         word_length = "3"
8         ui.stub(read: word_length)
9
10        game.should_receive(:raffle).with(word_length.to_i)
11
12        game_flow.next_step
13      end
14    end
15
16    #(...)
17  end
18 end
```

Na linha 10 desse 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:

```
1 describe GameFlow do
2   # (...)
```



```

3
4 describe "#next_step" do
5   context "when the player asks to raffle a word" do
6     it "prints a '_' for each letter in the raffled word" do
7       word_length = "3"
8       ui.stub(read: word_length)
9       game.stub(raffle: "mom", raffled_word: "mom")
10
11       ui.should_receive(:write).with("_ _ _")
12
13       game_flow.next_step
14     end
15   end
16
17   #(...)
18 end
19 end

```

Na linha 9 desse 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:

```

1 describe GameFlow do
2   # (...)
3
4   describe "#next_step" do
5     it "finishes the game when the player asks to" do
6       player_input = "fim"
7       ui.stub(read: player_input)
8
9       game.should_receive(:finish)
10
11       game_flow.next_step
12     end
13
14     #(...)
15   end
16 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 `game.should 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`, como você pode ver na linha 9. Esse teste espera que objeto `game` tenha um método `Game#finish`, que ainda precisa ser implementado.

O resultado final dos testes que movemos para o arquivo `spec/game_flow_spec.rb` ficará assim:

```
# encoding: UTF-8

require 'spec_helper'
require 'game_flow'

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!"
      ui.should_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?"
        ui.should_receive(:write).with(question)

        word_length = "3"
        ui.should_receive(:read).and_return(word_length)

        game_flow.next_step
      end
    end
  end
end
```

```

    end
  end

  context "when the player asks to raffle a word" do
    it "raffles a word with the given length" do
      word_length = "3"
      ui.stub(read: word_length)

      game.should_receive(:raffle).with(word_length.to_i)

      game_flow.next_step
    end

    it "prints a '_' for each letter in the raffled word" do
      word_length = "3"
      ui.stub(read: word_length)
      game.stub(raffle: "mom", raffled_word: "mom")

      ui.should_receive(:write).with("_ _ _")

      game_flow.next_step
    end

    it "tells if it's not possible to raffle with the given length" do
      word_length = "20"
      ui.stub(read: word_length)
      game.stub(raffle: nil)

      error_message = "Não temos uma palavra com o tamanho " <<
        "desejado,\n" <<
        "é necessário escolher outro tamanho."

      ui.should_receive(:write).with(error_message)

      game_flow.next_step
    end
  end

  it "finishes the game when the player asks to" do
    player_input = "fim"
    ui.stub(read: player_input)
  end
end

```

```
game.should_receive(:finish)

game_flow.next_step
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:

```
1 # encoding: UTF-8
2
3 require 'spec_helper'
4 require 'game'
5
6 describe Game do
7   let(:ui) { double("ui").as_null_object }
8   let(:word_raffler) { double("raffler").as_null_object }
9
10  subject(:game) { Game.new(ui, word_raffler) }
11
12  describe "#ended?" do
13    it "returns false when the game just started" do
14      game.should_not be_ended
15    end
16  end
17 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`. Então delete a linha 7 e adapte o `subject` na linha 10 para retirar a dependência de `ui`. Após essas adaptações, o teste ficará assim:

```
# encoding: UTF-8

require 'spec_helper'
```

```
require 'game'

describe Game do
  let(:word_raffler) { double("raffler").as_null_object }

  subject(:game) { Game.new(word_raffler) }

  describe "#ended?" do
    it "returns false when the game just started" do
      game.should_not 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
    word_raffler.should_receive(:raffle).with(3)

    game.raffle(3)
  end

  it "saves the raffled word" do
    raffled_word = "mom"
    word_raffler.stub(raffle: raffled_word)

    game.raffle(3)

    game.raffled_word.should == raffled_word
  end
end
```

```
describe "#finish" do
  it "sets the game as ended" do
    game.finish

    game.should be_ended
  end
end
```

Para fazer esses testes passarem, adicione os seguintes métodos na classe `Game`:

```
def raffle(word_length)
  @raffled_word = @word_raffler.raffle(word_length)
end

def ended?
  @ended
end
```

E mude o construtor da classe `Game` para ficar assim:

```
def initialize(word_raffler = WordRaffler.new)
  @word_raffler = word_raffler
  @ended = false
end
```

O que mudamos no construtor foi tirar a injeção de dependência `ui`, já que `Game` não depende mais de `ui`.

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:

```
# encoding: UTF-8

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` e o método privado `print_letters_feedback` para a nova classe, de modo a ficar assim:

```
1 class GameFlow
2   # (...)
3
4   def next_step
5     @ui.write("Qual o tamanho da palavra a ser sorteada?")
6     player_input = @ui.read.strip
7
8     if player_input == "fim"
9       @ended = true
10    else
11      if @raffled_word = @word_raffler.raffle(player_input.to_i)
12        print_letters_feedback
13      else
14        error_message = "Não temos uma palavra com o tamanho " <<
15                      "desejado,\n" <<
```

```

16             "é necessário escolher outro tamanho."
17
18         @ui.write(error_message)
19     end
20 end
21 end
22
23 private
24 def print_letters_feedback
25     letters_feedback = ""
26
27     @raffled_word.length.times do
28         letters_feedback << "_ "
29     end
30
31     letters_feedback.strip!
32
33     @ui.write(letters_feedback)
34 end
35 end

```

Vamos precisar fazer alguns ajustes após mover esses métodos. Na linha 11 não podemos mais usar `@word_raffler`, então mude essa linha para ficar assim: `@game.raffle(player_input.to_i)`.

Na linha 27, não temos mais acesso a variável de instância `@raffled_word`, já que isso faz parte do estado interno do objeto `game`. Então mude essa linha para ficar assim: `@game.raffled_word`

Após essas modificações, esses métodos ficarão assim:

```

1 class GameFlow
2     # (...)
3
4     def next_step
5         @ui.write("Qual o tamanho da palavra a ser sorteada?")
6         player_input = @ui.read.strip
7
8         if player_input == "fim"
9             @ended = true
10        else
11            if @game.raffle(player_input.to_i)
12                print_letters_feedback

```



```

13     else
14         error_message = "Não temos uma palavra com o tamanho " <<
15                         "desejado,\n" <<
16                         "é necessário escolher outro tamanho."
17
18         @ui.write(error_message)
19     end
20 end
21 end
22
23 private
24 def print_letters_feedback
25     letters_feedback = ""
26
27     @game.raffled_word.length.times do
28         letters_feedback << "_ "
29     end
30
31     letters_feedback.strip!
32
33     @ui.write(letters_feedback)
34 end
35 end

```

Após toda essa refatoração, ao rodarmos o RSpec inteiro, vemos o seguinte:

```
$ bundle exec rspec
```

Failures:

```

1) GameFlow#next_step finishes the game when the player asks to
   Failure/Error: game.should_receive(:finish)
     (Double "game").finish(any args)
       expected: 1 time
       received: 0 times
     # ./spec/game_flow_spec.rb:73

```

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`. Para corrigir esse problema, edite o método `GameFlow#next_step` para ficar assim:

```

1 def next_step
2   @ui.write("Qual o tamanho da palavra a ser sorteada?")
3   player_input = @ui.read.strip
4
5   if player_input == "fim"
6     @game.finish
7   else
8     if @game.raffle(player_input.to_i)
9       print_letters_feedback
10    else
11      error_message = "Não temos uma palavra com o tamanho " <<
12                      "desejado,\n" <<
13                      "é necessário escolher outro tamanho."
14
15      @ui.write(error_message)
16    end
17  end
18 end

```

A única coisa que fizemos foi mudar a linha 6 para chamar o método `game.finish`.

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
```

```
(...)
```

```
Exit status was 1 but expected it to be 0. Output:
```

```
forca/bin/forca:8:in `<main>':
  undefined method `start' for #<Game>
```

```
(...)
```

```
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 assim:

```
#!/usr/bin/env ruby

$:.unshift File.join(File.dirname(__FILE__), "..", "lib")

require 'game_flow'

game_flow = GameFlow.new
game_flow.start

while not game_flow.ended?
  game_flow.next_step
end
```

Ao rodar o Cucumber de novo, vemos o seguinte:

```
$ bundle exec cucumber

(...)
Exit status was 1 but expected it to be 0. Output:

Exit status was 1 but expected it to be 0. Output:

forca/bin/forca:10:in `':
  undefined method `ended?' for #<GameFlow> (NoMethodError)

(...)
3 scenarios (3 failed)
```

Infelizmente, todos os testes ainda estão quebrando. 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 tem, porque quem guarda o estado do jogo é a classe `Game`. Mas, podemos fazê-la delegar esse método para sua dependência `game`. Para fazer isso, edite o começo do arquivo `lib/game_flow.rb` para ficar assim:

```
1 # (...)
2
3 require 'forwardable'
4
5 # Esta classe é responsável pelo fluxo do jogo.
6 #
7 class GameFlow
8   extend Forwardable
9   delegate :ended? => :@game
10
11   # (...)
12 end
```

Você pode ver na linha 9 que o que fizemos foi apenas usar a biblioteca padrão do Ruby, `Forwardable`, para delegar o método `ended?` para o objeto dependência `game`.

Ao rodar o Cucumber agora, 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:

- 1) extração da responsabilidade de sortear palavra
- 2) extração da responsabilidade de fluxo do jogo
- 3) redução de condicionais aninhadas no método `next_step`

Conseguimos refatorar os dois primeiros pontos. O último ponto, vamos deixar na nossa lista débitos técnicos e atacá-lo nos próximos capítulos, durante o desenvolvimento da próxima funcionalidade. Mas antes de irmos em frente vamos pensar um pouco nos pontos-chave deste capítulo.

8.4 PONTOS-CHAVE 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 assim 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 essa suíte que nos dá confiança e produtividade na melhoria contínua da qualidade do nosso código.

CAPÍTULO 9

Especificando a segunda funcionalidade

No capítulo anterior nós refatoramos o nosso código para melhorar a qualidade do mesmo, facilitando assim a adição das funcionalidades que estariam por vir. Neste capítulo começaremos a desenvolver uma dessas novas funcionalidades, a funcionalidade de *"adivinhar letra"*.

Para desenvolver essa funcionalidade, continuaremos seguindo o fluxo de *outside-in development* com TDD, 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).

9.1 DOCUMENTANDO ESPECIFICAÇÃO E CRITÉRIO DE ACEITE COM CUCUMBER

Na seção [5.1](#) nós definimos que o nosso jogo teria três funcionalidades:

- 1) **Jogador começa um novo jogo:** jogo mostra a mensagem inicial e pede pro jogador sortear uma palavra;
- 2) **Jogador adivinha uma letra da palavra:** jogador tenta adivinhar uma letra e o jogo mostra se ele acertou ou errou;
- 3) **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.

A primeira funcionalidade nós já desenvolvemos e refatoramos nos capítulos anteriores. Vamos agora começar a desenvolver a segunda funcionalidade. Antes de começar a desenvolver essa funcionalidade, precisamos primeiro pensar na especificação dela 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 da especificação acima 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 o critério de aceite dessa funcionalidade. Para defini-lo, podemos escrever alguns exemplos sobre como deve ser o comportamento dessa funcionalidade em cenários específicos. Os exemplos que iremos utilizar 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 adivinha com sucesso duas vezes

Quanto mais o jogador for acertando, mais o jogo vai mostrando pra 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.

Os cenários acima 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.

9.2 DEFININDO O TESTE DE ACEITAÇÃO DO PRIMEIRO CENÁRIO

No nosso primeiro cenário, "*Sucesso ao adivinhar letra*", iremos 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 desse cenário 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 deverá ter "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:

- 1) jogador começa um jogo;
- 2) ele escolhe que o tamanho da palavra a ser sorteada será de 3 letras;
- 3) ele tenta adivinhar que a palavra tem a letra "a";
- 4) 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, ao jogador tentar adivinhar que a palavra tem a letra ‘a’, ele vai acertar?”. No teste está definido que ele irá acertar, mas o porquê ele acerta e o porquê da palavra com 3 letras ter a letra “a” dentro dela não está explícito! Deixar coisas implícitas em um teste é um problema. Isso é um problema por dois motivos:

- 1) Para podermos implementar um teste de modo que ele seja determinístico, é necessário explicitar o que o teste precisa para ele funcionar (setup do teste);
- 2) Quando alguém for ler seu teste, 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 do seu teste: dado que o sistema está em um determinado estado, quando realizo uma determinada ação nesse sistema, então observo uma data consequência. Quando não é possível entender esse 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* [10].

Do modo como definimos nosso teste, ele está pecando nos dois pontos acima.

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 do nosso teste não está claro porque 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 no nosso teste para deixar explícito que se uma palavra de três letra 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, mais 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 deverá ter "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 do teste já poderá entender a relação de causa e consequência desse teste. Ao ler o teste, 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)
0m0.008s
```

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::Ast::Table
  pending # express the regexp above with the code you wish you had
end
```

```
Dado /~que escolhi que a palavra a ser sorteada deverá ter "(.*)"
```

```

    letras$/ do |arg1|
      pending # express the regexp above with the code you wish you had
    end

Quando /^tento adivinhar que a palavra tem a letra "(.*?)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Então /^o jogo mostra que eu adivinhei uma letra com sucesso$/ do
  pending # express the regexp above with the code you wish you had
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"*.

9.3 MELHORE A TESTABILIDADE DO SEU SOFTWARE

O próximo step que iremos implementar é o *"o jogo tem as possíveis palavras para sortear"*. Esse step irá servir 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 implementar isso.

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 no código acima que apesar do 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 palavra 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`:

```
describe WordRaffler do
  it "raffles a word from a given list of words" do
    words = %w(me you nice)
    raffler = WordRaffler.new(words)

    raffler.raffle(3).should == "you"
    raffler.raffle(2).should == "me"
    raffler.raffle(4).should == "nice"
  end

  # (...)
end
```

Como você verá mais a 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 irá nos possibilitar construir um modo de setarmos as palavras sorteáveis do nosso jogo durante os testes de Cucumber.

Como você já ouvir falar, usar Test Driven Development faz com que seu software tenha um design melhor. O que aconteceu acima é um exemplo claro e simples disso.

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 a *controlabilidade* dele, 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 seja possível 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, no qual o leitor do teste 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).

Voltando a 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)
0m0.008s
```

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::Ast::Table
  pending # express the regexp above with the code you wish you had
end

(...)
```

Pela saída do Cucumber, podemos nos lembrar 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"* irá 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.

Nosso jogo é executado pelos testes 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"
```

Na ideia acima estamos passando a lista de palavras sorteáveis como argumento para o binário do nosso jogo. Vamos implementar essa ideia.

Para implementar essa ideia, 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

$: .unshift File.join(File.dirname(__FILE__), "..", "lib")

require 'game_flow'

game_flow = GameFlow.new
game_flow.start

while not 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`. Logo, 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 assim:

```
# (...)

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

while not game_flow.ended?
  game_flow.next_step
end
```

Repare o modo como estamos criando o objeto `WordRaffler`. Estamos checando se está vindo algum argumento da linha de comando, e se estiver vindo, 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_raffable_words(words)
end
```

Note que o método `set_raffable_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_raffable_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 NewGameHelpers
  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, logo basta que modifiquemos o método `start_new_game` para utilizar a lista definida pelo método `set_rafflable_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 tem 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 NewGameHelpers
  def set_rafflable_words(words)
    @raffable_words = words
  end

  def start_new_game
    set_rafflable_words(%w[hi mom game fruit]) if @raffable_words.nil?

    steps %{
      When I run `forca "#{@raffable_words}"` interactively
    }
  end
end
```

O que mudamos no método `start_new_game` foram duas coisas. Ele define a lista de palavras sorteáveis usando o método `set_rafflable_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)
0m0.019s
```

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

```
Dado /^que escolhi que a palavra a ser sorteada deverá ter "(.*)" \
  letras\
$/ do |number_of_letters|
  pending # express the regexp above with the code you wish you had
end

(...)
```

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 deverá ter "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 deverá ter "(.*)" \
  letras\
$/ do |number_of_letters|
  steps %{
    When I 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 step 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 "(.*)"$/ do |letter|
  steps %{
    When I 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)
0m0.323s
```

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

```
Então /^o jogo mostra que eu adivinhei uma letra com sucesso$/ do
  pending # express the regexp above with the code you wish you had
end
```

O step pendente é o *"Então o jogo mostra que eu adivinhei uma letra com sucesso"*. O jogo irá mostrar pro jogador que ele adivinhou uma letra com sucesso imprimindo na tela a mensagem "Você adivinhou uma letra com sucesso.". Então 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
  steps %{
    Then the stdout should contain:
      """
      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:
- 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 pode 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.

9.4 PONTOS-CHAVE 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 é só para fazer o teste passar.

Para finalizar, vale a pena se 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 10

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!

10.1 REFATORANDO NOSSO JOGO PARA TER UMA MÁQUINA DE ESTADOS

Na capítulo anterior anterior finalizamos o primeiro cenário da funcionalidade “*Adivinhar letra*”, que foi o cenário “*Sucesso ao adivinhar letra*”.

O teste de aceitação desse cenário 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:

- 1) dado que o jogo está no estado que já tem uma palavra sorteada
- 2) quando o jogador adivinhar uma letra com sucesso
- 3) 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
      game.stub(:raffled_word, "hey")

      success_message = "Você adivinhou uma letra com sucesso."
      ui.should_receive(:write).with(success_message)

      game.next_step
    end
  end
end
```

Ao rodar o RSpec, o teste quebra como esperado. Para fazer esse teste passar será necessário mudar o comportamento do método `Game#next_step`. Vamos analisar como esse método está hoje e pensar 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 onde o jogo ainda não tem uma palavra sorteada. Nesse estado, o fluxo consiste em perguntar pro jogador o tamanho da palavra a ser sorteada e sortear essa palavra. 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 adicionar essa nova condicional, nós já tínhamos identificado que o aninhamento de condicionais atual nesse método já é um débito técnico. Não queremos piorar o débito técnico 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), então o próximo passo do fluxo do jogo deve ser o de sorteio de palavra. Se o estado do `game` for `:word_raffled` (palavra sorteada), então o próximo passo do fluxo do jogo é pedir pro 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
(...)
```

```
14 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
  its(:state) { should == :initial }
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 :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 teste.

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
    word_raffler.stub(raffle: "word")

    expect do
      game.raffle(3)
    end.to change { game.state }.from(:initial).to(:word_raffled)
  end
end
```

Note como nosso spec ficou bonito ao usarmos o matcher *expect change* do RSpec! Após apreciarmos um pouco a sintaxe do RSpec, vamos fazer o teste passar.

Para fazer o teste passar, 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)
  if @raffled_word = @word_raffler.raffle(word_length)
    @state = :word_raffled
  end
end
```

Ao rodarmos o RSpec, podemos ver que o test passou, vamos para o próximo teste.

O próximo teste é para o cenário quando o sorteio da palavra não tem sucesso. Nesse cenário 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
    word_raffler.stub(raffle: nil)

    game.raffle(3)

    game.state.should == :initial
  end
end
```

```
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 a nova propriedade de estado da `Game`, existe um método que vale a pena ser modificado. É o método `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 pro

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
    game.stub(raffled_word: "hey")

    success_message = "Você adivinhou uma letra com sucesso."
    ui.should_receive(:write).with(success_message)

    game.next_step
  end
end
```

10.2 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
    game.stub(raffled_word: "hey")

    success_message = "Você adivinhou uma letra com sucesso."
    ui.should_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 serve para testar 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`. Setar esse estado vai fazer parte do setup desse teste. Podemos fazer isso no setup desse teste do seguinte modo:

```
game.stub(state: :word_raffled)
```

Além desse `stub` serão necessários mais alguns passos no setup desse teste.

Como o cenário desse teste 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 do seguinte modo:

```
letter_guess = "e"
ui.stub(read: letter_guess)
```

Por fim, precisamos fazer o setup sobre de onde vem a informação 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 que existisse 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, o usaríamos para fazer o setup do nosso teste do seguinte modo:

```
game.stub(guess_letter: true)
```

Juntando todos os passos de setup acima, nosso teste ficará assim:

```
context "when the player guess a letter with success" do
  it "prints a success message" do
    game.stub(state: :word_raffled, guess_letter: true)

    success_message = "Você adivinhou uma letra com sucesso."
    ui.should_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: ui.should_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?"), ("")
```

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 fazer o teste passar precisamos primeiro modificar a classe `GameFlow` para que ela tome suas ações baseado no novo atributo de estado do objeto `game`. Hoje 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 acima é o primeiro passo do jogo e só deve ser realizado quando o jogo estiver no estado inicial. Vamos extrair a lógica acima 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
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:83
rspec ./spec/game_flow_spec.rb:23
rspec ./spec/game_flow_spec.rb:35
rspec ./spec/game_flow_spec.rb:44
rspec ./spec/game_flow_spec.rb:54
rspec ./spec/game_flow_spec.rb:70

```

Essa modificação quebrou esses testes porque nesses testes 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`. Ou seja, todos esses testes compartilham um mesmo setup. Podemos colocar esse setup no `let` que define o colaborador `game` no começo desse teste. Hoje 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 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 fazermos 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
```

No código acima, 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`.

10.3 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
  finishes the game when the player asks to
  when the player guess a letter with success
    prints a success message
  when 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 just started
    asks the player for the length of the word to be raffled
```

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, esses testes 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
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
end

```

E para finalizar esse 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
      game.stub(state: :word_raffled, guess_letter: true)

      success_message = "Você adivinhou uma letra com sucesso."
      ui.should_receive(:write).with(success_message)

      game_flow.next_step
    end
  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 o seguinte output:

```
GameFlow
#start
  prints the initial message
#next_step
  finishes the game when the user asks to
  when the game is in the 'initial' state
    asks the user for the length of the word to be raffled
    and the user 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 user guess a letter with success
        prints a success message

Finished in 0.00807 seconds
7 examples, 0 failures
```

Figura 10.1: 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.

10.4 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* [8], e depois mais explorada no paper *Mock roles, not Objects* [6] e também no tradicional livro *Growing Object-Oriented Software, Guided by Tests* [7].

Utilizamos uma variante dessa técnica na seção 10.2, 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 dessa técnica, mais de modo mais explícito, de modo que fique claro o seu uso. Para conhecer a versão original dessa técnica, recomendo que você leia o paper *Mock roles, not Objects* [6].

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 deverá ter "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:

- 1) O jogo deve estar no estado `:word_raffled`
- 2) O jogo pede para o jogador adivinhar uma letra
- 3) Jogador adivinha uma letra com sucesso
- 4) Jogo mostra mensagem de sucesso
- 5) Jogo mostra letras adivinhadas

Já trabalhamos no primeiro, terceiro e quarto passos do fluxo acima, ainda faltam o segundo e quinto passos. Vamos trabalhar nisso.

Vamos escrever um teste de unidade para especificar que quando o jogo está no estado `:word_raffled`, o próximo passo do jogo é perguntar pro 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
    game.stub(state: :word_raffled)
```

```

    question = "Qual letra você acha que a palavra tem?"
    ui.should_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 fazer esse teste 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 `Game#ask_to_guess_a_letter`. Vamos modificar esse método para que ele faça a pergunta pro 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 { game.stub(state: :word_raffled) }

  it "asks the player to guess a letter" do
    question = "Qual letra você acha que a palavra tem?"
    ui.should_receive(:write).with(question)

    game_flow.next_step
  end
end

```



```

end

context "and the player guess a letter with success" do
  it "prints a success message" do
    game.stub(guess_letter: true)

    success_message = "Você adivinhou uma letra com sucesso."
    ui.should_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 10.2, 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 quando o jogador adivinhar uma letra com sucesso, o jogo deve mostrar pra ele as letras adivinhadas. Vamos especificar esse comportamento.

Para que o `GameFlow` imprima as letras adivinhadas, essa informação precisa estar salva em algum lugar, precisamos definir qual método retornará essa informação. 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 chamar esse método 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 que vai existir um método chamado

`Game#guessed_letters`, podemos escrever nosso próximo teste stubando esse método do seguinte modo:

```
context "and the player guess a letter with success" do
  # (...)

  it "prints the guessed letters" do
    game.stub(guess_letter: true, raffled_word: "hey",
              guessed_letters: ["e"])

    ui.should_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: ui.should_receive(:write).with("_ e _")
Double "ui" received :write with unexpected arguments
expected: ("_ e _")
got: ("Qual letra você acha que a palavra tem?"),
     ("Você adivinhou uma letra com sucesso.")
```

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 fazer com 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 `Game#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
end

```

Agora precisamos implementar o método `guessed_letters`, que é o argumento sendo passado para `@ui.write`. Esse método irá 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 ela já foi adivinhada, devemos colocá-la na string de retorno. Se não foi adivinhada, 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.each_char do |letter|
    if @game.guessed_letters.include?(letter)
      letters << letter + " "
    else
      letters << "_ "
    end
  end

  letters.strip!
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 `game.stub(guess_letter: 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

```

```
    game.stub(guess_letter: true)
    # (...)
end

it "prints the guessed letters" do
  game.stub(guess_letter: true, raffled_word: "hey",
            guessed_letters: ["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 { game.stub(guess_letter: true) }

  it "prints a success message" do
    # (...)
  end

  it "prints the guessed letters" do
    game.stub(raffled_word: "hey", guessed_letters: ["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. Então onde antes usávamos `print_letters_feedback`, podemos agora usar o `guessed_letters`, reduzindo assim 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
end

```

Para utilizar o novo `guessed_letters`, podemos modificar esse trecho para ficar assim:

```

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
end
end

```

Ao fazer isso, o teste `'it "prints a '_' for each letter in the raffled word"` quebra. Ele quebra 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`. Hoje esse `let` está assim:

```
let(:game) { double("game", state: :initial).as_null_object }
```

Vamos modificá-lo para ficar assim:

```

let(:game) { double("game",
                    state: :initial,
                    guessed_letters: []).as_null_object }

```

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
  end
end
```

Ou seja, já fizemos todos os testes de unidade necessários do `GameFlow` para cobrir o cenário de adivinhação de letra com sucesso. No entanto, a funcionalidade não está pronta ainda. Para verificarmos isso, podemos rodar o Cucumber e ver que ele ainda tem testes no vermelho:

```
$ bundle exec cucumber
(...)
```

```
game_flow.rb:69:in `ask_to_guess_a_letter':
  undefined method `guess_letter' for #<Game:0x007fbdf1c96740>
  (NoMethodError)
```

```
7 scenarios (2 failed, 5 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. Ele não foi implementando ainda porque a dependência dele até então tinha sido stubada 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`. Esse método foi definido durante os

testes de unidade do `GameFlow` utilizando a técnica de Interface discovery. Assim como esse método, definimos outro método do mesmo modo, que ainda precisa ser implementado, 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, esse método retorna, `true`, caso contrário, ele 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"

    game.guess_letter("h").should be_true
  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 esse teste do seguinte modo:

```
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"

      game.guess_letter("z").should be_false
    end
  end
end
```

```
end
end
```

Ao rodar esse teste, 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:

```
it "returns false if the given letter is an blank string" do
  game.raffled_word = "hey"

  game.guess_letter("").should be_false
  game.guess_letter(" ").should be_false
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 == ''

  @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. Então, se o jogo tem como palavra sorteada a palavra “hey”, e a letra “e” já foi adivinhada com sucesso, logo 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("e")

    game.guessed_letters.should = ["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`, então 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, essa variável 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
    game.guessed_letters.should = []
  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: game.guessed_letters.should == ["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 o seguinte 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 do
      game.guess_letter("h")
    end.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 == ''
```

```

if @raffled_word.include?(letter)
  @guessed_letters << letter
  return true
else
  return 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 do
      game.guess_letter("h")
    end.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 do
  result should not 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 == ''

  if @raffled_word.include?(letter)
    @guessed_letters << letter
    @guessed_letters.uniq!
    return true
  else
    return 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.

Hoje já temos um teste no `spec/game_flow_spec.rb` para lidar com o caso do jogador querer finalizar o jogo no meio:

```
describe "#next_step" do
  # (...)

  it "finishes the game when the player asks to" do
    player_input = "fim"
    ui.stub(read: player_input)

    game.should_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"
  ui.stub(read: player_input)

  game.stub(state: :initial)
  game.should_receive(:finish)
  game_flow.next_step

  game.stub(state: :word_raffled)
  game.should_receive(:finish)
  game_flow.next_step
end
```

Ao rodar o RSpec, podemos ver que esse teste quebra com a seguinte mensagem:

```
$ bundle exec rspec
Failures:
```

```
1) GameFlow#next_step finishes the game when the player asks to
   Failure/Error: game.should_receive(:finish)
     (Double "game").finish(any args)
       expected: 1 time
       received: 0 times
```

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 do `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
# (...)

```

Ao invés de fazer isso, vamos fazer uma refatoração para evitar essa duplicação. Vamos extrair esse comportamento de fazer uma pergunta pro 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, ao invés do `it`:

```

xit "finishes the game when the player asks to" do
  # (...)
end

```

Agora vamos extrair do método `ask_to_raffle_a_word` a lógica de fazer uma pergunta pro jogador e tratar o input `fim` para um método chamado `ask_the_player`:

```

def ask_to_raffle_a_word
  ask_the_player("Qual o tamanho da palavra a ser sorteada?") do |length|
    if @game.raffle(length.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 o método `ask_the_player` irá fazer é imprimir uma pergunta pro jogador e tratar o input de resposta. Quando o input for diferente de `"fim"`, então 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"
  ui.stub(read: player_input)

  game.stub(state: :initial)
  game.should_receive(:finish)
  game_flow.next_step

  game.stub(state: :word_raffled)
  game.should_receive(:finish)
  game_flow.next_step
end
```

Ao rodar o RSpec, vemos que esse teste continua quebrando. Para fazer ele 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
  ask_the_player("Qual letra você acha que a palavra tem?") 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
(...)
```

```
7 scenarios (7 passed)
21 steps (21 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.

10.5 FINALIZANDO A FUNCIONALIDADE ADIVINHAR LETRA

Para a funcionalidade ‘Adivinhar letra’ letra definimos os seguintes cenários com Cucumber:

- Sucesso ao adivinhar letra;
- Erro ao adivinhar letra;
- Jogador advinha 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 força já perdeu.

O que precisamos fazer agora é escrever o teste de aceitação para esse cenário. A estrutura do teste desse cenário 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 força 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 força já perdeu.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada deverá ter "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)
```

```
0m0.324s
```

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

```
Então /^o jogo mostra que eu errei a adivinhação da letra$/ do
  pending # express the regexp above with the code you wish you had
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` do seguinte modo:

```
Então /^o jogo mostra que eu errei a adivinhação da letra$/ do
  steps %{
    Then the stdout should contain:
      """
      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 { game.stub(state: :word_raffled) }
    # (...)

    context "and the player fails to guess a letter" do
      before { game.stub(guess_letter: false) }

      it "prints a error message" do
        error_message = "Você errou a letra."
        ui.should_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
  ask_the_player("Qual letra você acha que a palavra tem?") 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 { game.stub(guess_letter: false) }
  # (...)

  it "prints the list of the missed parts" do
    missed_parts_message = "O boneco da forca perdeu as " <<
                          "seguintes partes do corpo: cabeça"
    ui.should_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 essas informações 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 stubar esse novo método no nosso teste:

```
it "prints the list of the missed parts" do
  game.stub(missed_parts: ["cabeça"])

  missed_parts_message = "0 boneco da forca perdeu as " <<
                        "seguintes partes do corpo: cabeça"
  ui.should_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
  ask_the_player("Qual letra você acha que a palavra tem?") do |letter|
    if @game.guess_letter(letter)
      # (...)
    else
      @ui.write("Você errou a letra.")

      missed_parts_message = "0 boneco da forca perdeu as " <<
                          "seguintes partes do corpo: "
      missed_parts_message << @game.missed_parts.join(", ")
      @ui.write(missed_parts_message)
    end
  end
end
```

Agora 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 especificar esse método, 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 testar 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 escrevendo o primeiro teste:

```
it "returns an empty array when there's no missed parts" do
  game.missed_parts.should == []
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
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, então 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

game.missed_parts.should == ["cabeça", "corpo", "braço esquerdo"]
end

```

Ao rodarmos o RSpec podemos ver que esse teste falha. Ele falha 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")

    game.missed_parts.should == ["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. Para fazer esse teste passar, 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 feito uma adivinhação errada:

```
def guess_letter(letter)
  return false if letter.strip == ''

  if @raffled_word.include?(letter)
    # (...)
  else
    @missed_parts << HANGMAN_PARTS[@wrong_guesses]
    @wrong_guesses += 1
    return 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 pra 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 então 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 pra ele as letras que ele adivinhou.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada deverá ter "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 esse teste 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, então o jogo deve mostrar pra ele as partes que o boneco da forca 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 forca são perdidas.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada deverá ter "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! Ele passou de primeira 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.

10.6 PONTOS-CHAVE 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 as várias atividades que fizemos e aprendemos ao longo deste capítulo.

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 pensar 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 ele 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* [6], para que você entenda o porquê de mock objects e como os seus criadores pensaram que eles devem ser utilizados. Lembre-se, mocks e stubs são diferentes e tem diferentes casos de uso [5].

CAPÍTULO 11

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. Falta apenas a funcionalidade que permite ao jogador ganhar ou perder o jogo. Vamos desenvolver essa funcionalidade e finalmente terminar o nosso jogo.

Ao final deste capítulo você terá desenvolvido uma aplicação completa usando TDD do começo ao fim. Parabéns!

Isso pode ser um pequeno passo para homem, mas um grande passo no desenvolvimento de software com qualidade!

11.1 ESPECIFICANDO O FIM DO JOGO

Existem 3 modos do nosso jogo terminar:

- 1) Jogador termina jogo no meio;

2) Jogador vence o jogo;

3) 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

0 jogo termina em dois cenários:

1. 0 jogador adivinhou todas as letras da palavra,
então ele ganha! :)
2. 0 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 agora escrever testes com o Cucumber para implementar esses critérios de aceite. Vamos começar pelo cenário onde o jogador vence o jogo.

11.2 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 força apareçam.

Dado que comecei um jogo

E que escolhi que a palavra a ser sorteada deverá ter "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:

```
# features/step_definitions/game_steps.rb:43
"""
Você venceu! :)
"""

process still alive after 3 seconds (ChildProcess::TimeoutError)
```

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. Ele não terminou 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 jogador acertar todas as letras da palavra, o jogo imprime uma mensagem de sucesso e termina 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 { game.stub(state: :ended) }

    it "prints a success message when the player win"
  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 win" do
  ui.should_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` esta no estado `:ended`, é necessário saber que o jogo acabou e o jogador venceu. Vamos imaginar então que exista um método chamado `Game#player_won?` que retorna essa informação e vamos stubar esse método no setup do nosso teste:

```

it "prints a success message when the player win" do
  game.stub(player_won?: true)

  ui.should_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 que stubamos, 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 então 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 num 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")

    game.player_won?.should be_true
  end
end
```

Ao rodar esse teste, 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.to_a.uniq.sort
  @guessed_letters.sort == raffled_word_letters
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 esse teste, 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 == ''

  if @raffled_word.include?(letter)
    @guessed_letters << letter
    @guessed_letters.uniq!

    @state = :ended if all_letters_were_guessed?

    return true
  else
    @missed_parts << HANGMAN_PARTS[@wrong_guesses]
    @wrong_guesses += 1
    return false
  end
end
```

Note que utilizamos um método chamado `all_letters_were_guessed?` que ainda não existe. Esse método deve retornar `true` quando todas as le-

tras já foram adivinhadas. Implemente ele como um método privado no arquivo `lib/game.rb`:

```
private
def all_letters_were_guessed?
  raffled_word_letters = @raffled_word.to_s.chars.to_a.uniq.sort

  @guessed_letters.sort == raffled_word_letters
end
```

Ao rodarmos os testes agora, vemos que todos passaram. Inclusive o teste do método `Game#player_won?`! Vamos para a refatoração.

Perceba que os métodos `player_won?` e `all_letters_were_guessed?` tem muita duplicação:

```
def player_won?
  return false if @state != :ended

  raffled_word_letters = @raffled_word.to_s.chars.to_a.uniq.sort

  @guessed_letters.sort == raffled_word_letters
end

private
def all_letters_were_guessed?
  raffled_word_letters = @raffled_word.to_s.chars.to_a.uniq.sort

  @guessed_letters.sort == raffled_word_letters
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 guessed all letters" do
    game.state = :word_raffled
    game.raffled_word = "hi"

    6.times { game.guess_letter("z") }

    game.player_won?.should be_false
  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 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 está passando 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
    game.player_won?.should be_false

    game.state = :word_raffled
    game.player_won?.should be_false
  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` desse cenário 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.

11.3 JOGADOR PERDE O JOGO

Na seção 11.1 nós já especificamos o critério de aceite do cenário 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 implementar esse cenário, 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 deverá ter "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)
```

```
0m0.131s
```

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

```
Quando /^tento adivinhar que a palavra tem a letra "(.*)" "(.*)"
    vezes$/ do |arg1, arg2|
    pending # express the regexp above with the code you wish you had
end
```

Esse step que está indefinido é muito parecido com um step que já temos pronto:

```
Quando /^tento adivinhar que a palavra tem a letra "(.*)"$/ do |letter|
  steps %{
    When I type "#{letter}"
  }
end
```

Para implementar esse step indefinido, podemos reutilizar o step definition já pronto, simplesmente o chamando o número de vezes que for necessário. Faça isso do seguinte modo no arquivo `features/step_definitions/game_steps.rb`:

```
Quando /^tento adivinhar que a palavra tem a letra "(.*)" \
"(.*)" vezes$/ do |letter, number_of_guesses|

  number_of_guesses.to_i.times do
    steps %{
      * tento adivinhar que a palavra tem a letra "#{letter}"
    }
  end
end
```

Perceba que nosso novo step definition irá simplesmente chamar o outro step N vezes. No caso do nosso cenário, ele irá chamar seis vezes.

Rode o Cucumber para esse cenário e verifique a saída:

```
$ bundle exec cucumber -t @wip
(...)
```

```
process still alive after 3 seconds (ChildProcess::TimeoutError)
```

Failing Scenarios:

```
cucumber features/fim_do_jogo.feature:32 # Scenario: 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 do 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 acontecendo 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 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 { game.stub(state: :ended) }

  # (...)

  it "prints a defeat message when the player lose" do
    game.stub(player_won?: false)

    ui.should_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 testa 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 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 o estado `: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 " <<
    "miss 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 == ''

  if @raffled_word.include?(letter)
```

```

    # (...)
  else
    @missed_parts << HANGMAN_PARTS[@wrong_guesses]
    @wrong_guesses += 1
    return false
  end
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 == ''

  if @raffled_word.include?(letter)
    # (...)
  else
    @missed_parts << HANGMAN_PARTS[@wrong_guesses]
    @wrong_guesses += 1

    @state = :ended if @wrong_guesses == 6

    return false
  end
end
end

```

Ao rodarmos o RSpec agora, vemos que ele está no verde. Aproveite e rode o Cucumber do cenário 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.

11.4 PRÓXIMOS PASSOS

Você acabou de construir uma aplicação inteira usando TDD 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 TDD.

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 TDD em todos os seus próximos 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* [1], o Mauricio Aniche 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 a 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 lose
```

Perceba quão grande é o número de testes que fizemos só para esse método o método `GameFlow#next_step`. Seu dever de casa será refatorar a classe `GameFlow` para evitar que seja necessário fazer tantos testes para um método só. Faça isso e mande o resultado da sua refatoração com um link para seu Github para o endereço da lista de e-mail deste livro, que é casadocodigo-tdd-ruby@googlegroups.com.

Referências Bibliográficas

- [1] M. Aniche. *Test-Driven Development: Teste e Design no Mundo Real*. Casa do Código, 2012.
- [2] Barry W Boehm. Software engineering economics. 1981.
- [3] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. The rspec book. 2009.
- [4] Eric Evans. Domain-driven design: tackling complexity in the heart of software. 2004.
- [5] M. Fowler. Mocks aren't stubs. <http://martinfowler.com/articles/mocksArentStubs.html>.
- [6] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, not objects. 2004.
- [7] S. Freeman and N. Pryce. Growing object-oriented software, guided by tests. 2009.
- [8] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. 2001.
- [9] R.C. Martin. Principles of ood. <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.
- [10] G. Meszaros. xunit test patterns: Refactoring test code. 2007.
- [11] D. North. Introducing bdd. <http://dannorth.net/introducing-bdd/>.