

Pesquisa e Inovação III

Introdução ao Spock Framework

Sumário

Preparando o ambiente para trabalhar com Spock	6
Configurando o Groovy nas principais IDEs Java	6
Eclipse	6
IntelliJ IDEA	6
NetBeans	6
Configurando as dependências Spock do projeto com Maven	7
Experimentando o Spock on-line	8
Primeiros testes automatizados com Spock	9
Primeiro teste: A classe Math calcula potência corretamente?	9
Executando um teste	10
O que acontece quando um teste falha	10
Teste que falha por exceção	11
Teste que não passa em uma verificação	12
Anatomia de um teste Spock	17
Testes são chamados de Specifications	17
Criar as Specifications com o sufixo Test possui algum efeito colateral?	17
Toda classe de teste deve ser subclasse de spock.lang.Specification	17
Uma classe de teste pode conter vários métodos de teste	17
Fixture Methods (métodos de montagem)	18
setup()	19
cleanup()	19
setupSpec()	19
cleanupSpec()	19
Blocks (Blocos)	20
setup:	21
given:	21
when:	21
expect:	22
then:	22
cleanup:	22
and:	22
where:	22

Ordem e restrição de combinações de blocos	22
Blocos para Four-phase test	23
Blocos para Given-When-Then	23
Por que a anatomia do Spock é essa	24
Spock é uma ferramenta para aplicação de BDD?	24
Data-Driven Testing - Facilitando os testes de múltiplos cenários	25
Testando um Analisador de IMC	25
Um método, vários cenários	26
Classe de testes para vários cenários SEM Data-Driven Testing	26
Classe de testes para vários cenários com Data-Driven Testing	27
Testes com falha no Data-Driven Testing	29
Exception Conditions - Testes à espera de exceções	31
Às vezes o certo é dar errado	31
Blocos try-catch	32
Testando vários cenários de erro	34
Verificando detalhes das exceções	35
Testando com a ajuda de Mocks	37
O que são dublês e por que usá-los	37
Tipos de dublês	38
Dependências adicionais para Mocks	39
Configurando retornos de métodos em Mocks	39
Só é possível indicar um retorno para métodos dos Mocks? E se precisar de algo mais dinâmico?	42
Configurando métodos de Mocks para aceitarem qualquer valor	43
Configurando Mocks para lançarem exceções	44
Criando um Mock que lança uma exceção simples	44
Criando um Mock que lança uma exceção com muitos detalhes	47
Configurando retornos de métodos aninhados em Mocks	48
Verificando a quantidade de execuções de métodos dos Mocks	52
Testes de integração em projetos Spring	55
Spring TestContext framework e Spring Module do Spock	55
Testes de integração em um projeto Spring Boot	55
Resumo do Projeto Spring Boot	55
Classe Lutador (JPA Entity)	56
Interface LutadorRepository (Spring Repository)	57

Classe LutadorService (Spring Service)	57
Classe LutadorController (Spring REST Controller)	58
Endpoint GET /melhor-lutador da LutadorController	58
Endpoint GET / daLutadorController	59
Endpoint POST / {JSON} daLutadorController	59
Classe LutadoresApp (Spring Application)	60
Arquivo application.properties do projeto Spring Boot	60
Cenários do teste de integração Spring MVC	61
Classe LutadorControllerTest (teste de integração com Spring Module)	61
Cenário Deveria retornar status 204 e sem corpo quando não existirem lutadores	62
Cenário Deveria retornar status 400 e mensagem esperada quando falha em criar Lutador	62
Cenário Deveria retornar status 201 com JSON correto quando cria um Lutador	63
Cenário Deveria retornar 200 trazer todos os lutadores e no formato correto quando existirem Lutadores	63
Testes de integração em um projeto Spring MVC	64
Resumo do Projeto Spring MVC	64
Arquivo application.properties do projeto Spring MVC	65
Arquivo home.jsp do projeto Spring MVC	65
Cenários do teste de integração Spring MVC	66
Classe LutadorMvcControllerTest (teste de integração com Spring Module)	66
Cenário Deveria retornar status 200 e redirecionar para JSP correto	67
Cenário Deveria retornar status 400 quando parâmetro o usuario não for enviado	67
Ciclo de vida do contexto Spring nos testes de integração	68
Profiles e properties do projeto Spring nos testes de integração	68
Definindo um arquivo de properties para os testes	69
Escolhendo o profile para os testes	69
Ordem de busca de properties conforme profile	70
Apêndice A - Guia de Groovy para desenvolvedores Java	71
Como funciona o Groovy	71
Características e recursos do Groovy	72
Diferenças na sintaxe	72
A tipagem pode ser estática ou dinâmica	74
Operador == faz o papel do .equals()	76
Concatenação de valores String mais dinâmica e simples	76

String podem ter múltiplas linhas sem concatenações	77
Interoperabilidade com classes Java	78
NullPointerException é muito fácil de ser evitado	79
Manipulação de coleções é muito simples	79
Técnicas de iteração simples e embarcadas para números, coleções e Strings	82
Verificação de "verdadeiro/falso" expandida, porém facilitada	84
Métodos embarcados para as conversões mais comuns	86
Manipulação de datas é muito simples	87
Acesso direto a métodos privados	88
Alteração do comportamento de métodos em classes e objetos em tempo de execução	89
getters e setters podem invocados apenas com o nome do atributo	90

Referências	92
--------------------	-----------

Preparando o ambiente para trabalhar com Spock

A configuração de um ambiente Spock consiste basicamente em preparar o ambiente para a linguagem Groovy e incluir algumas dependências **Maven**, **Gradle** ou **SBT** no projeto onde o Spock será usado. A seguir será demonstrado como realizar essas duas ações.

Configurando o Groovy nas principais IDEs Java

Como o Spock é uma ferramenta para a escrita de testes em projetos para a plataforma Java, é natural que sejam usadas as principais IDEs dessa plataforma. Não é necessária a instalação de plugin específico para Spock, mas devemos habilitar a capacidade de editar e compilar arquivos Groovy na IDE. **Eclipse**, **IntelliJ Idea** e **NetBeans** possuem suporte para Groovy conforme detalhado a seguir.

Eclipse

Existe um conjunto de plugins para Groovy, o **Groovy-Eclipse** que, historicamente, possui versão compatível com a última ou penúltima versão do Eclipse. Para instalá-lo, o processo é o mesmo de instalação de qualquer plugin: basta adicionar o *Update Site* conforme a versão do Eclipse desejada. As versões disponíveis constam na sua *wiki* (<https://github.com/groovy/groovy-eclipse/wiki>). Dentre os plugins que serão listados para instalação, os necessários para o uso do Spock são:

- Groovy-Eclipse feature
- Groovy Compiler 2.4 (*ou versão mais atual, conforme versão do Spock que adotar*)
- Groovy-Eclipse M2E integration

IntelliJ IDEA

O IntelliJ IDEA, mesmo na versão **Community Edition**, já conta com plugin para Groovy embarcado. Assim, não é necessária a instalação de qualquer complemento nesta IDE. Este plugin, como o do Eclipse está em constante atualização.

Caso você opte por usar esta IDE, talvez não consiga executar individualmente os testes escritos em Spock como testes JUnit mesmo que seus arquivos compilem. A IDE pode informar que não se trata de uma classe *JUnit*. Caso isso ocorra, basta marcar o diretório **src/test/groovy** como "*Test Sources Root*" (ou "*Test Sources*" em versões até 2017). Caso esse diretório não exista, crie o **groovy** dentro de **src/test**.

NetBeans

O NetBeans, desde a versão **9**, já vem com um plugin para Groovy pré-instalado, bastando solicitar que seja ativado. Até o fechamento desta obra, o NetBeans só permitia a

execução de uma versão atual de um teste individual em Groovy após um Maven *clean*.

Configurando as dependências Spock do projeto com Maven

Caso seu projeto use o Maven como gerenciador de dependências, é necessário apenas configurar **um plugin** e **uma dependência**. No código do exemplo a seguir é demonstrado como seria a configuração do plugin obrigatório para que os testes escritos em Groovy sejam compilados e executados na etapa de **test** do build do Maven.

```
<plugin>

  <groupId>org.codehaus.gmavenplus</groupId>

  <artifactId>gmavenplus-plugin</artifactId>

  <version>1.5</version>

  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>addTestSources</goal>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>

</plugin>
```

No código do exemplo a seguir é mostrado como incluir a **dependência mínima** da versão mais recente do Spock framework.

```
<dependency>
  <groupId>org.spockframework</groupId>
  <artifactId>spock-core</artifactId>
  <version>1.1-groovy-2.4</version> <!-- ou a versão que preferir -->
  <scope>test</scope>
</dependency>
```

A dependência recém-apresentada já é o suficiente para usar o Spock. Porém, questões como correção de bugs ou compatibilidade com outras bibliotecas podem forçar o uso de uma versão do Groovy **diferente** da que está configurada para a versão do Spock que configurou. Nesses casos, deve-se incluir uma dependência específica, como no próximo código, que faria o Maven usar a versão 2.4.12 do Groovy.

```
<!-- ou a versão que precisar. No caso do uso de Eclipse, pode ser necessário -->

<dependency>
  <groupId>org.codehaus.groovy</groupId>
  <artifactId>groovy-all</artifactId>
  <version>2.4.12</version>
  <scope>test</scope>
</dependency>
```

Experimentando o Spock on-line

Caso você queira experimentar ou demonstrar o Spock sem configurar absolutamente nada, é possível usar uma ferramenta on-line chamada **Spock Web Console**, acessível via navegador em <http://meetspock.appspot.com/>. Não é preciso nenhum *plugin* instalado no navegador para usá-la. Com ela, você pode criar e executar testes em Spock, conforme o exemplo de execução na figura a seguir.

The screenshot displays the Spock Web Console interface. At the top, there's a blue header with the text "Spock Web Console". Below this, a code editor shows a Groovy script for a Spock test. The script defines a class `MathTest` that extends `Specification` and contains a test method `'2 ao quadrado deve ser 2'()` which expects `Math.pow(2, 2) == 3`. Below the code editor, there's a bar with "Actions" and buttons for "Run Script", "New Script", "Publish Script", and "View Recent Scripts". The "Run Script" button has been clicked, and the results are shown in a panel below. This panel has tabs for "Result", "Output", and "Stack Trace". The "Result" tab is active, showing that the test `MathTest - 2 ao quadrado deve ser 2` has FAILED. The failure message is "Condition not satisfied:" followed by a comparison `Math.pow(2, 2) == 3` where the left side evaluates to `4.0` and the right side to `false`. The stack trace indicates the failure occurred at `MathTest.2 ao quadrado deve ser 2(Script1.groovy:6)`.

```
1 import spock.lang.*
2
3 class MathTest extends Specification {
4   def '2 ao quadrado deve ser 2'() {
5     expect:
6       Math.pow(2, 2) == 3
7   }
8 }
9
10
```

Actions > Run Script New Script Publish Script View Recent Scripts

Result Output Stack Trace

MathTest
- 2 ao quadrado deve ser 2 FAILED
Condition not satisfied:
Math.pow(2, 2) == 3
| |
4.0 false
at MathTest.2 ao quadrado deve ser 2(Script1.groovy:6)

Primeiros testes automatizados com Spock

Primeiro teste: A classe Math calcula potência corretamente?

Se tiver criado um projeto **Maven**, basta criar suas classes Groovy de teste no diretório **src/test/groovy**. É nele que criaremos nosso primeiro teste com Spock.

Nosso primeiro teste vai testar a class **Math** do Java, para verificar se seu método **pow()** calcula corretamente uma potência. É claro que esse método da API do Java funciona e não precisaríamos testá-lo. Esse primeiro exemplo é apenas para que nosso primeiro contato com o Spock seja rápido e de fácil compreensão. Vejamos o código a seguir, que pode ser criado em um arquivo chamado **MathTest.groovy**, já que faremos testes para a classe **Math**.

```
import spock.lang.Specification

class MathTest extends Specification {

    def '2 ao quadrado deve ser 4'() {

        expect:

        Math.pow(2,2) == 4

    }

}
```

O pacote dos testes Spock pode ser qualquer um e, assim como em Java, apenas indica o caminho relativo de diretórios onde o arquivo está no diretório de códigos-fontes do projeto. Apenas para deixar claro quando estamos descrevendo códigos de testes ou não, os de teste estarão sempre em subpacotes de unit..

O próximo detalhe a destacar é que a classe de teste é uma subclasse de **spock.lang.Specification**. **Todo teste com Spock deve ser uma subclasse de Specification** para que funcione com todas as funcionalidades do framework.

Note ainda que a classe não é public. Na verdade, ela **é** pública. É que o termo public é opcional em Groovy e, se omitido, significa que a classe será pública em tempo de execução.

Algo que pode parecer estranho para quem nunca teve contato com Groovy é que o nome do método é uma **frase entre aspas**. Isso é permitido em Groovy e facilita muito a criação de testes, afinal, cada método de teste em uma classe de testes automatizados deveria executar um **cenário de teste**. Qual nome de método deixa mais claro o cenário a ser testado: **def doisAoQuadradoDeveSerQuatro()** ou **def '2 ao quadrado deve ser 4'()**?

Essa sintaxe é muito usada em testes com Spock, sendo predominante na documentação oficial.

O bloco **expect:** indica onde está a verificação (teste lógico) que determina o resultado do teste. Sem ele, o teste não teria sido executado. Outra característica muito importante é que as verificações em Spock normalmente contêm literalmente testes lógicos e não métodos, como se faz em **JUnit**. No exemplo há somente 1 verificação, porém, pode haver quantas forem necessárias e o teste só passará se todas passarem. O exemplo a seguir demonstra como seria um teste que faz mais de uma verificação nesse bloco.

```
class MathTest extends Specification {
    def 'sqrt deve calcular a raiz quadrada'() {
        expect:
        Math.sqrt(4) == 2
        Math.sqrt(25) == 5
        Math.sqrt(144) == 12
    }
}
```

Nesse exemplo, queríamos saber se o método sqrt da classe Math realmente calcula a raiz quadrada corretamente e, para isso, fizemos 3 verificações no bloco expect.

Caso você tenha pouco conhecimento sobre Groovy, o *Apêndice A - Guia de Groovy para desenvolvedores Java* possui um conteúdo que dá o caminho das pedras dessa linguagem.

Executando um teste

Para executar **todas** as classes de teste de um projeto, basta executar o comando de seu gerenciador de dependências que invoca os testes do projeto. No Maven, por exemplo, todos os *goals* menos o compile promovem a execução dos testes do projeto.

A execução de apenas uma classe de testes pode ser feita usando o comando específico de seu gerenciador de dependência recomendado para isso. No **Maven**, o comando seria como o do exemplo a seguir.

```
mvn -Dtest=MathTest test
```

Caso use uma IDE completa (Eclipse, IntelliJ ou NetBeans), ela permitirá a execução da classe **MathTest** como se fosse uma classe **JUnit**. Ou seja, basta pedir a execução da classe que o teste unitário nela será executado. Essas IDEs permitem ainda a execução de apenas um dos métodos na classe de teste.

O que acontece quando um teste falha

Um teste pode falhar por exceção não esperada em tempo de execução ou porque ao menos uma das verificações nele não passou. Vejamos o que ocorre em cada uma dessas situações.

Teste que falha por exceção

Tomemos como exemplo o código a seguir, que possui uma sutil diferença em relação ao anterior, na qual trocamos a base por uma divisão entre **1** e **0**.

```
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        expect:
            Math.pow((1/0), 2) == 4
    }
}
```

Esse código vai compilar, mas provocará uma exceção em tempo de execução, pois 1/0 em Groovy, assim como em Java, resulta em uma **java.lang.ArithmeticException: Division by zero**. Ao ser executado, o teste geraria uma saída no log como a do exemplo a seguir:

```
2 ao quadrado deve ser 4(MathTest) Time elapsed: 0.054 sec <<< FAILURE!
Condition failed with Exception:

Math.pow((1/0), 2) == 5
    |
    java.lang.ArithmeticException: Division by zero

    at MathTest.2 ao quadrado deve ser 4(MathTest.groovy:6)
Caused by: java.lang.ArithmeticException: Division by zero
    at java.math.BigDecimal.divide(BigDecimal.java:1742)
```

Como é possível notar, o log de erro do Spock **indica o local exato que provocou a exceção**. Com outras ferramentas de testes, saberíamos a linha, mas não o ponto exato. Note ainda como o log fica mais fácil de ser lido, pois o nome do método é, literalmente, o nome do cenário.

O formato e os detalhes da saída gerada pelo Spock ajudam muito naquelas situações em que ocorre o clássico **NullPointerException**. Vamos simular esse tipo de exceção a partir do código de teste a seguir, no qual declaramos um **Double base = null** propositalmente para que a invocação de seu **.intValue()** provoque uma exceção.

```
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        Double base = null
        expect:
            Math.pow(base.intValue(), 2) == 4
    }
}
```

Ao ser executado, esse teste resultaria em um log como este a seguir.

```
2 ao quadrado deve ser 4(MathTest) Time elapsed: 0.058 sec <<< FAILURE!
Condition failed with Exception:

Math.pow(base.intValue(), 2) == 4
    |
    java.lang.NullPointerException: Cannot invoke method intValue() on null object

    at MathTest.2 ao quadrado deve ser 4(MathTest.groovy:7)
Caused by: java.lang.NullPointerException: Cannot invoke method intValue() on null object
    at java.math.BigDecimal.divide(BigDecimal.java:1742)
```

Note que o exato objeto que provocou o **NullPointerException** é indicado, evitando ajustes e/ou novas execuções para descobrir quem é o culpado pela exceção.

Como foi demonstrado nos exemplos, o ponto específico onde ocorreu a exceção ficaria indicado, o que facilita muito o tratamento do problema. Com outras ferramentas, teríamos que, ou executar várias vezes mexendo um pouco no código (*tentativa e erro*) ou teríamos que executar o teste em *modo debug* caso a IDE usada possuía esse recurso.

Teste que não passa em uma verificação

Comparação entre números

Tomemos como exemplo o código a seguir, que é muito parecido com o anterior, porém, contém um erro proposital no resultado esperado (5 em vez de 4 como resultado esperado para 2 ao quadrado).

```
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        expect:
        Math.pow(2, 2) == 5
    }
}
```

Esse código vai compilar e executar sem exceção alguma. Porém, a verificação configurada no bloco expect vai **falhar**, pois 2 ao quadrado não é igual a 5. Ao ser executado, o teste geraria uma saída no log como a do exemplo a seguir:

```
Running unit.br.com.apostilapockframework.topico03.MathTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.185 sec <<< FAILURE!
2 ao quadrado deve ser 4(MathTest) Time elapsed: 0.059 sec <<< FAILURE!
Condition not satisfied:
```

```
Math.pow(2, 2) == 5
    |      |
    4.0    false
```

```
at MathTest.2 ao quadrado deve ser 4(MathTest.groovy:8)
```

Como é possível notar novamente, o log de erro do Spock **indica o local exato da falha de verificação e os valores envolvidos no teste**. Ficaram claros não só a linha da falha, mas também os valores comparados. Note que o resultado de **pow(2,2)** ficou explícito no log. Isso nos poupa de criar uma mensagem de erro recuperando, manualmente, o resultado do método.

Comparação entre alfanuméricos

Quando criamos testes que comparam valores alfanuméricos (**String**), além de apontar que existe a diferença, o Spock gera um pequeno relatório estatístico das diferenças. Para ver como isso funciona, vamos criar outra classe de teste, a **StringTest**. Nela, vamos testar os métodos **toUpperCase()** e **substring()** da biblioteca padrão do Java.

```
class StringTest extends Specification {

    def "toUpperCase() deveria por tudo em caixa alta"() {
        expect:
        "bom dia".toUpperCase() == "Bom Dia"
    }

    def "substring() deveria extrair a parte desejada do texto"() {
        expect:
        "sejam bem vindos!".substring(1,5) == "sejam"
    }

}
```

No último código, montamos 2 testes que vão falhar. O objetivo é que você veja como é o log que o Spock gera em caso de falha na comparação entre diferentes objetos String. A execução desse teste geraria logs como dos textos a seguir.

```

toUpperCase()          deveria          por          tudo          em          caixa
alta(unit.br.com.apostilapockframework.topico03.StringTest) Time elapsed: 0.208 sec <<<
FAILURE!
org.spockframework.runtime.SpockComparisonFailure: Condition not satisfied:

"bom dia".toUpperCase() == "Bom Dia"
    |          |
    BOM DIA false

                4 differences (42% similarity)
                B(OM) D(IA)
                B(om) D(ia)

```

```

substring()          deveria          extrair          a          parte          desejada          do
texto(unit.br.com.apostilapockframework.topico03.StringTest) Time elapsed: 0.006 sec <<<
FAILURE!

org.spockframework.runtime.SpockComparisonFailure: Condition not satisfied:

"sejam bem vindos!".substring(0,4) == "sejam"

                |          |
                seja      false

                        1 difference (80% similarity)

                        seja(-)

                        seja(m)

```

Note nos 2 logs anteriores que, além de serem indicados o local exato da diferença e quais os valores esperado e recuperado, são descritos uma porcentagem de semelhança e os detalhes de cada posição no texto onde foram encontradas diferenças.

Comparação entre coleções

Ao comparar o conteúdo ou o tamanho de coleções, o Spock imprime no log não só a linha e o ponto da diferença como os conteúdos das coleções envolvidas. Vamos criar mais um teste, o **CollectionsTest** (veja no próximo código-fonte), no qual teremos 3 testes entre coleções. Caso fique com dúvidas sobre a lista e/ou o mapa criados com Groovy, não se esqueça de que pode recorrer ao *Apêndice A - Guia de Groovy para desenvolvedores Java*.

```

class CollectionsTest extends Specification {

    def 'listas devem ser iguais'() {
        expect:
        def lista1 = ['café','leite','açúcar']
        def lista2 = ['café','leite','adoçante']
        lista1 == lista2
    }

    def 'mapas devem ser iguais'() {
        expect:
        def mapa1 = [PA:'Pará', SP:'São Paulo', MG:'Minas Gerais']
        def mapa2 = [DF:'Distrito Federal', GO:'Goiás', PR:'Paraná']
        mapa1 == mapa2
    }

    def 'listas devem ser ter o mesmo tamanho'() {
        expect:
        def lista1 = ['uva','morango','amora']
        def lista2 = ['abacate','maçã verde']
        lista1.size() == lista2.size()
    }
}

```

No primeiro teste, comparamos o conteúdo de 2 listas (**List**). No segundo, comparamos o conteúdo de 2 mapas (**Map**). Já no terceiro teste, comparamos o tamanho de 2 listas (**List**). Todos eles vão falhar. O log produzido será com o exemplo a seguir.

listas devem ser iguais (...) Condition not satisfied:

```

lista1 == lista2
|      |  |
|      |  [café, leite, adoçante]
|      false
|      [café, leite, açúcar]

```

mapas devem ser iguais (...) Condition not satisfied:

```

mapa1 == mapa2
|      |  |
|      |  [DF:Distrito Federal, GO:Goiás, PR:Paraná]
|      false
|      [PA:Pará, SP:São Paulo, MG:Minas Gerais]

```

listas devem ser ter o mesmo tamanho (...) Condition not satisfied:

```

lista1.size() == lista2.size()
|      |  |  |  |
|      3  |  |  2
|      |  [abacate, maçã verde]
|      false
|      [uva, morango, amora]

```

Note no log dos 3 testes com coleções que, além de indicar o ponto exato da falha do teste, o Spock imprimiu os conteúdos das coleções comparadas.

Conforme demonstram os exemplos que acabamos de ver, independente do que estamos comparando, o Spock nos fornece muitos detalhes. Isso reduz muito a necessidade de *tentativa e erro* e da execução dos testes em *modo debug* da IDE para identificar e corrigir o problema.

Anatomia de um teste Spock

Testes são chamados de Specifications

Uma classe de teste no Spock é chamada em sua documentação oficial de **Specification**. Por isso, ela apresenta as classes de teste com o sufixo Spec no nome. A classe **MathTest** criada no tópico anterior, por exemplo, poderia ter o nome alterado para **MathSpec**, se fôssemos seguir a convenção da documentação do Spock.

Criar as Specifications com o sufixo **Test** possui algum efeito colateral?

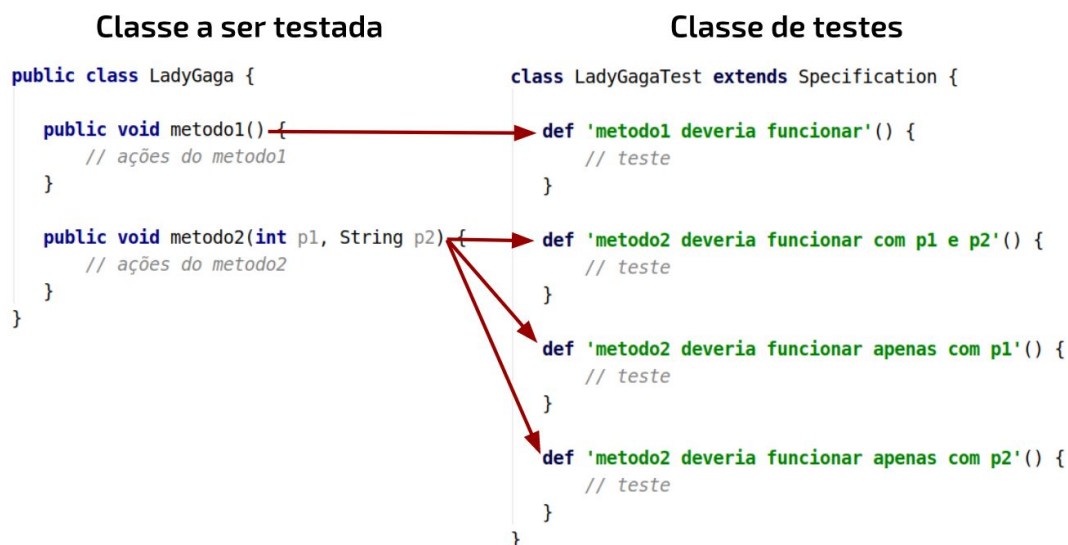
Não. Inclusive todos os testes nesta apostila terão o sufixo **Test** no nome. A razão disso é que, além de continuarem funcionando perfeitamente, seguem a convenção do Maven e afins, que executam os testes em classes que terminam com esse termo e estão no diretório **src/test**.

Toda classe de teste deve ser subclasse de **spock.lang.Specification**

Nos exemplos abordados no tópico anterior vimos isso e, como já foi dito lá, essa herança é realmente **necessária**. As classes de testes podem estender diretamente ou indiretamente (ou seja, estender outra que estenda) a **spock.lang.Specification**. Ao estender essa classe, todos os recursos do framework Spock ficam disponíveis para a classe de teste.

Uma classe de teste pode conter vários métodos de teste

Em geral costuma-se criar uma classe de testes para cada classe no projeto. E, para cada método da classe testada, criam-se vários métodos de teste, de acordo com a quantidade de cenários necessários. O Spock permite a criação de quantos métodos de teste forem necessários na mesma classe de teste. Um exemplo desse tipo de situação está na figura a seguir.



A figura ilustra a seguinte situação: tínhamos que testar a **MinhaClasse**, que possui 2 métodos (**metodo1** e **metodo2**). Foi criada a classe de testes **MinhaClasseTest** com 4 métodos (cenários) de teste, sendo apenas 1 para testar o **metodo1** e 3 para testar o **metodo2**.

Fixture Methods (métodos de montagem)

Imagine os testes de uma classe que acessa um banco de dados: antes da execução dos testes, seria bem útil configurar um banco de dados temporário, para evitar que a execução de testes crie "lixo" em uma base de dados real. Ao final da execução dos testes, esse banco de dados deveria ser completamente eliminado. Situações como a desse exemplo são comuns em testes unitários mais complexos e em testes de integração.

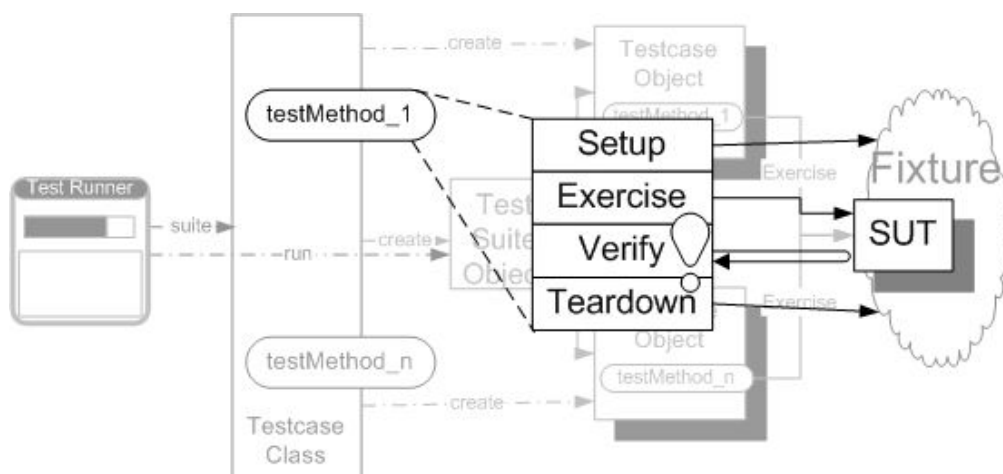
Nessas situações, onde é necessário preparar um ambiente para a execução dos testes e/ou executar ações após a execução, o uso de **fixture methods** (ou **métodos de montagem**, em tradução livre) é bastante recomendado. Eles são uma forma de implementar o ciclo de vida de testes proposto no padrão **Four-phase test**.

Padrão Four-phase test

"*Teste de quatro fases*", em tradução livre. Foi proposto por Gerard Meszaros em seu livro clássico *XUnit Test Patterns - Refactoring Test Code*, de 2007. Esse padrão propõe que a execução dos cenários de testes possua 4 (quatro) fases:

- **Setup**, em que se configura o necessário para a execução e validação dos testes.
- **Exercise**, em que é executada a ação que é alvo do teste.
- **Verify**, em que é verificado se a ação fez o que era esperado.
- **Teardown**, em que todos os recursos alocados pelo teste (objetos em memória, arquivos em disco, conexões com outros sistemas etc.) devem ser liberados.

O conceito de *Four-Phase* pode ser encontrado em [http://xunitpatterns.com/Four Phase Test.html](http://xunitpatterns.com/Four%20Phase%20Test.html) e ilustrado na figura a seguir.



Fonte: <http://xunitpatterns.com/Four%20Phase%20Test.html>

Os *fixture methods* do Spock framework são: **setup**, **setupSpec**, **cleanup** e **cleanupSpec**.

setup()

Caso exista na classe de testes, é executado **antes de cada método de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado 4 vezes, **antes** de cada um deles.

cleanup()

Caso exista na classe de testes, é executado **depois de cada método de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado 4 vezes, **depois** de cada um deles.

setupSpec()

Caso exista na classe de testes, é executado **apenas uma vez antes de todos os métodos de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado apenas 1 vez, **antes** da execução do primeiro teste. Caso exista também um `setup()` na mesma classe, sua primeira execução será **depois** do `setupSpec()`.

cleanupSpec()

Caso exista na classe de testes, é executado **apenas uma vez depois de todos os métodos de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado apenas 1 vez, **depois** da execução do último teste. Caso exista também um `cleanup()` na mesma classe, sua última execução será **antes** do `cleanupSpec()`.

A relação entre as fases do padrão *Four-phase test* e os *fixture methods* do Spock está representada na figura a seguir.

Fase	Método
Setup	setupSpec()
	setup()
Exercise	'método de teste'()
Verify	
Teardown	cleanup()
	cleanupSpec()

Para deixar bem clara a ordem e quantidade de execuções dos **fixture methods**, observe a classe de testes de exemplo a seguir.

```

class MinhaClasseTest extends Specification {
  def setupSpec() {}

  def setup() {}

  def 'metodo1 deveria funcionar'() {}

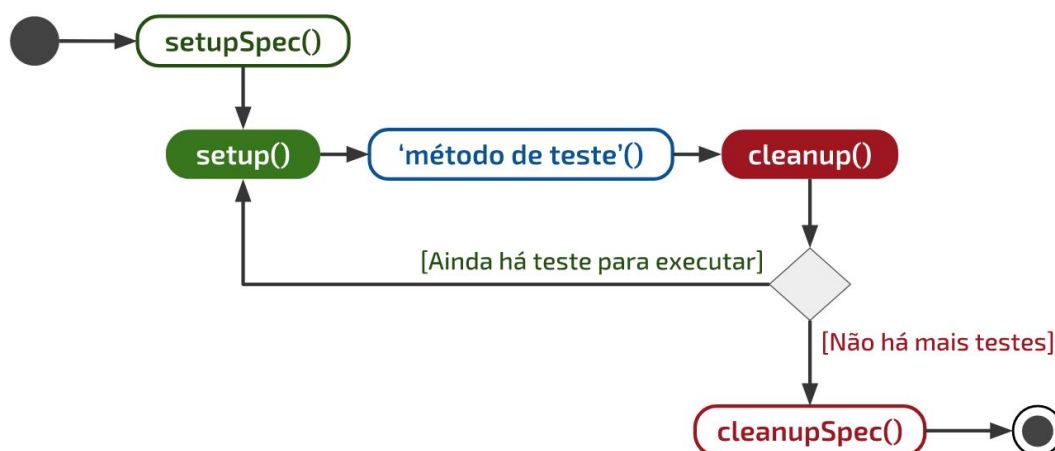
  def 'metodo2 deveria funcionar'() {}

  def cleanup() {}

  def cleanupSpec() {}
}

```

Ao ser executada, os métodos seguiriam a ordem de execução indicada no diagrama de atividade a seguir.



Importante: mesmo que os testes falhem por exceções ou por validações, todos os *fixture methods* são executados.

Blocks (Blocos)

Imagine a situação em que uma classe de acesso a banco de dados possui vários métodos e cada um faz operações tão diferentes entre si que, para testá-los, seria necessário um banco de dados temporário diferente para cada. Ou seja, os diversos cenários de teste precisariam de *setups* e/ou de *cleanups* diferentes. Considerando apenas o recurso dos *fixture methods*, a saída seria criar várias classes de testes, uma para cada cenário de teste. Porém, para situações como esta, o Spock possui um interessante recurso chamado **Blocks** (**Blocos**, em tradução livre), com o qual é possível **dividir as etapas de execução de um teste dentro de método de teste**.

Quanto à sua sintaxe, seu nome é **sempre** acompanhado por dois pontos (:) e cada bloco pode conter quantas linhas de código forem necessárias.

Os blocos do Spock indicam as principais etapas do ciclo de vida de um teste, desde a preparação até o pós-teste. São eles: **setup**, **given**, **when**, **expect**, **then**, **cleanup** e **and**.

Conforme a combinação de blocos utilizada, o teste acaba adotando o padrão **Four-phase test** ou o **Given-When-Then (GWT)**.

Padrão Given-When-Then

"Dado-quando-então", em tradução livre. Foi proposto por Dan North e Ivan Moore no artigo *Introducing BDD*, publicado na revista *Better Software* em março de 2006. Esse padrão propõe que a execução dos cenários de testes possua 3 (três) fases:

- **Given**, em que são definidas as precondições do cenário.
- **When**, em que é identificado o evento (a ação) do teste.
- **Then**, em que é verificado se o evento ocorreu conforme o esperado.

setup:

Caso seja necessária a configuração de um ambiente para a execução de um cenário de teste, ela pode ser feita no bloco **setup**. Ele deve ser sempre o primeiro bloco e não pode ser usado em conjunto com o **given**. Este bloco só pode estar presente 1 (uma) vez por método.

Se usado, torna-se necessária a presença de um bloco **expect**.

given:

Caso seja necessária a configuração de um ambiente para a execução de um cenário de teste, ela pode ser feita no bloco **given**. Ele deve ser sempre o primeiro bloco e não pode ser usado em conjunto com o **setup**. Este bloco só pode estar presente 1 (uma) vez por método.

Se usado, torna-se necessária a presença de um bloco **expect** ou dos blocos **when** e **then**.

when:

É o bloco onde o evento a ser testado deve ser executado. Pode existir mesmo sem um bloco de preparação/configuração anterior. Se o bloco **given** existir, deve estar necessariamente após ele. Não pode ser usado em conjunto com o **setup**. Este bloco só pode estar presente 1 (uma) vez por método.

Se usado, torna-se necessária a presença de um bloco **then**.

expect:

É o bloco onde o evento a ser testado deve ser executado. Pode ser usado se o bloco de preparação/configuração usado foi o **setup:** ou o **given:** e deve estar, necessariamente, após um deles. Este bloco só pode estar presente 1 (uma) vez por método.

É o **único** bloco que pode estar **sozinho** num método de testes, como foi possível feito no tópico anterior.

then:

É o bloco onde o evento a ser testado deve ser executado. Só pode ser usado se o bloco de preparação/configuração usado foi o **given:** e deve estar, necessariamente, após um **when:**. Este bloco só pode estar presente 1 (uma) vez por método.

cleanup:

Este bloco será executado após o bloco de execução do evento usado no teste (**expect:** ou **then:**) mesmo que eles falhem por verificação ou exceção. Deve estar, necessariamente, após o método de execução do evento. Este bloco só pode estar presente 1 (uma) vez por método.

and:

Este bloco simplesmente dá continuidade ao bloco imediatamente anterior, podendo ser usado após qualquer um dos outros blocos. É usado apenas para dividir e organizar um bloco em partes menores. Exemplos: se estiver após um **setup:** funciona como um **setup:**; se estiver após um **expect:**, funciona como um **expect:**. Este bloco pode estar presente várias vezes por método.

where:

Este bloco deve ser usado somente no caso de **Data-Driven Testing**. Esse tipo especial de teste será abordado no tópico a seguir.

Ordem e restrição de combinações de blocos

Os blocos não podem ficar em qualquer ordem e possuem algumas restrições quanto às combinações possíveis. Para ajudar a entender essas questões, vide a tabela a seguir.

Bloco	Ordem	Não pode ser usado em conjunto com
setup:	Sempre o 1º bloco	given:,when: e then:
given:	Sempre o 1º bloco	setup:
when:	1º bloco ou após given: (e seus and:, caso existam)	setup:
then:	Sempre após when: (e seus and:, caso existam)	setup:
expect:	Sozinho no método ou após setup: (e seus and:, caso existam)	given: e when:
cleanup:	Sempre após expect: ou then: (e seus and:, caso existam)	-
and:	Após qualquer outro bloco	-

Blocos para Four-phase test

Caso você adote o padrão **Four-phase test**, a combinação de blocos *recomendada* é **setup** > **and** > **expect**. A figura a seguir ilustra a relação entre as fases do padrão e os blocos do Spock.

Fase	Bloco
Setup	setup:
Exercise	and:
Verify	expect:
Teardown	cleanup:

Blocos para Given-When-Then

Caso você adote o padrão **Given-When-Then**, a combinação de blocos *recomendada* é **given** > **when** > **then**. A figura a seguir ilustra a relação entre as fases do padrão e os blocos do Spock.

Fase	Bloco
Given	given:
When	when:
Then	then:

Por que a anatomia do Spock é essa

Talvez você tenha se perguntado: por que o Spock adota o termo *Specification* e não *Test* como o JUnit, por exemplo? Ou ainda: por que blocos explícitos para os padrões *Given-When-Then* e *Four-phase test*? É porque o Spock foi desenvolvido sob influência de uma técnica chamada **BDD**. Essa influência levou à anatomia dos testes do Spock abordada neste tópico.

Behaviour-Driven Development - BDD

"Desenvolvimento Dirigido a Comportamento", em tradução livre. Foi proposta por Dan North e Ivan Moore no artigo *Introducing BDD*. Essa técnica fornece várias definições objetivas sobre como devem ser os testes em um projeto de desenvolvimento software. Algumas dessas definições são:

- Nomes de métodos de teste devem ser **frases**, algo comum no Spock.
- Usar **linguagem ubíqua** (linguagem compreensível por programadores e não programadores, como analistas de negócio, por exemplo). O padrão *Given-When-Then* é um exemplo dessa linguagem e o Spock o implementa com os blocos de mesmo nome.
- Critérios de aceitação devem ser executáveis.
- "Comportamento" é uma palavra mais útil que "teste" e os "comportamentos" são criados em **especificações (Specifications)**, por isso as classes de teste do Spock possuem esse sufixo, por padrão).

O artigo completo pode ser encontrado em <https://dannorth.net/introducing-bdd/> (versão traduzida para português em <http://broncodev.com/2016-10-11-introduzindo-o-bdd/>).

Como esta obra não tem o BDD como foco e tem como principal público-alvo profissionais sem experiência em testes, usaremos o termo **teste** em vez de **especificação**.

Spock é uma ferramenta para aplicação de BDD?

É possível usar o Spock na adoção BDD, como ensina e exemplifica Konstantinos Kapelonis em seu livro *Java Testing with Spock*. Porém, na mesma obra, o autor afirma que o Spock não é uma ferramenta completa de BDD, indicando outras mais adequadas para a adoção dessa técnica.

Data-Driven Testing - Facilitando os testes de múltiplos cenários

Testando um Analisador de IMC

Imagine que precisamos desenvolver um componente que informa a condição de saúde com base no *índice de massa corporal* (o famoso IMC). Esse componente recebe o IMC e o sexo de uma pessoa e devolve uma frase com a condição de saúde correspondente. Para facilitar nosso exemplo, não vamos usar a verdadeira tabela do IMC segundo a Organização Mundial de Saúde, mas uma versão resumida e com valores arredondados como a seguir.

Condição	IMC em mulheres	IMC em homens
Abaixo do peso	< 19	< 21
Peso normal	>= 19 .. < 26	>= 21 .. < 27
Acima do peso ideal	>= 26	>= 27

Para implementar esse componente, vamos criar um projeto Java e depois 2 tipos **enum** em Java: **Sexo** e **FaixaImc**, descritos nos códigos a seguir.

```
public enum Sexo {  
    FEMININO, MASCULINO;  
}
```

O **enum** **Sexo** serve para garantir que só sejam informados os valores que definimos previamente. Neste caso, **MASCULINO** e **FEMININO**.

```
public enum FaixaImc {  
  
    ABAIXO("Abaixo do peso", 19, 21),  
    NORMAL("Peso Normal", 26, 27),  
    ACIMA("Acima do peso", 100, 100);  
    // 100 é um IMC virtualmente impossível  
  
    protected String descricao;  
    protected double limiteMaximoFeminino;  
    protected double limiteMaximoMasculino;  
  
    // Construtor para os 3 atributos  
  
    public static FaixaImc getFaixa(double imc, Sexo sexo) {  
        FaixaImc retorno = values()[0];  
  
        for (int i=1; i < values().length; i++) {
```

```

        FaixaImc atual = values()[i];
        FaixaImc anterior = values()[i-1];

        double limiteInferior = (sexo == Sexo.FEMININO) ? anterior.limiteMaximoFeminino :
anterior.limiteMaximoMasculino;
        double limiteSuperior = (sexo == Sexo.FEMININO) ? atual.limiteMaximoFeminino :
atual.limiteMaximoMasculino;

        if (imc >= limiteInferior && imc < limiteSuperior) {
            retorno = atual;
        }
    }

    return retorno;
}

// getDescricao()
}

```

O *enum* **FaixaImc** contém as condições de saúde das 3 faixas de nossa tabela de IMC. O método que precisaremos testar é o **getFaixa()**, que tem como argumentos o IMC e o sexo de uma pessoa. Ele procura e retorna a condição de saúde após análise das faixas de IMC e sexo. Em ambos os *enums*, o pacote é apenas uma sugestão.

Um método, vários cenários

Agora que temos o problema e uma proposta de implementação da solução, **o que precisamos testar?** A partir de **2 valores de entrada** (IMC e sexo), precisamos testar **1 valor de saída** (condição de saúde). Só precisamos testar o método **getFaixa()** do *enum* **FaixaImc**.

O que será determinante para o sucesso de nossos testes será **a quantidade de cenários a serem testados**. Conseguiu contar quantos cenários devem ser testados para esse componente? São **3** condições possíveis para **MASCULINO** e **3** para **FEMININO**. Logo, temos que testar **6 cenários**, no mínimo.

Classe de testes para vários cenários SEM Data-Driven Testing

Vamos criar uma classe de testes, chamá-la de **FaixaImcTest** e criar os 6 cenários mínimos usando os recursos que aprendemos até aqui sobre o Spock. Veja o código-fonte a seguir.

```

class FaixaImcTest extends Specification {

    def 'IMC deve estar na faixa correta'() {
        expect:
        FaixaImc.getFaixa(18, Sexo.FEMININO) == FaixaImc.ABAIXO
        FaixaImc.getFaixa(21, Sexo.FEMININO) == FaixaImc.NORMAL
        FaixaImc.getFaixa(27, Sexo.FEMININO) == FaixaImc.ACIMA
        FaixaImc.getFaixa(20, Sexo.MASCULINO) == FaixaImc.ABAIXO
    }
}

```

```
FaixaImc.getFaixa(23, Sexo.MASCULINO) == FaixaImc.NORMAL
FaixaImc.getFaixa(28, Sexo.MASCULINO) == FaixaImc.ACIMA
}
}
```

Ao executar essa classe de teste, certamente os 6 cenários de teste passarão. Porém temos muita **repetição de código**. Vamos tentar melhorar isso? E se reescrevêssemos o método de teste de nossa classe **FaixaImcTest** e ele ficasse como o próximo código?

```
def 'IMC deve estar na faixa correta'() {
    setup:
    def cenarios = [
        [imc:18, sexo: Sexo.FEMININO, condicao: FaixaImc.ABAIXO],
        [imc:21, sexo: Sexo.FEMININO, condicao: FaixaImc.NORMAL],
        [imc:27, sexo: Sexo.FEMININO, condicao: FaixaImc.ACIMA],
        [imc:20, sexo: Sexo.MASCULINO, condicao: FaixaImc.ABAIXO],
        [imc:23, sexo: Sexo.MASCULINO, condicao: FaixaImc.NORMAL],
        [imc:28, sexo: Sexo.MASCULINO, condicao: FaixaImc.ACIMA]
    ]

    expect:
    cenarios.each{
        assert FaixaImc.getFaixa(it.imc, it.sexo) == it.condicao
    }
}
```

O que achou? Note que organizamos melhor o teste, dividindo-o explicitamente nas fases setup (em que criamos uma tabela de cenários com as entradas e saídas em uma lista de mapas) e expect (em que iteramos na tabela e cenários e testamos cada item). Caso tenha ficado com dúvidas na lista, nos mapas ou na iteração usados no último código, não se esqueça de que pode recorrer ao *Apêndice A - Guia de Groovy para desenvolvedores Java*.

Classe de testes para vários cenários com Data-Driven Testing

Percebeu que foi usado o termo "tabela de cenários" para descrever a lista de mapas com as entradas e saídas do teste anterior? Para poder descrever uma tabela de cenários literalmente como uma tabela podemos usar o recurso de **Data-Driven Testing** do Spock.

Como o nome sugere, esse recurso pode ser usado quando temos situações parecidas com o exemplo do IMC: onde há vários valores de saída a partir de várias combinações de valores de entrada. Outros exemplos de componentes onde esse recurso ajudaria muito nos testes:

- Miniguia astral (entradas: dia e hora do nascimento / saídas: signo e ascendente).
- Gerador de contracheque (entradas: vencimento e benefícios / saídas: descontos legais e valor líquido).

- Simulador de resultados de um campeonato esportivo (entradas: resultados das partidas / saída: campeão).

Vejamos como ficaria nosso método de teste usando esse recurso no código a seguir.

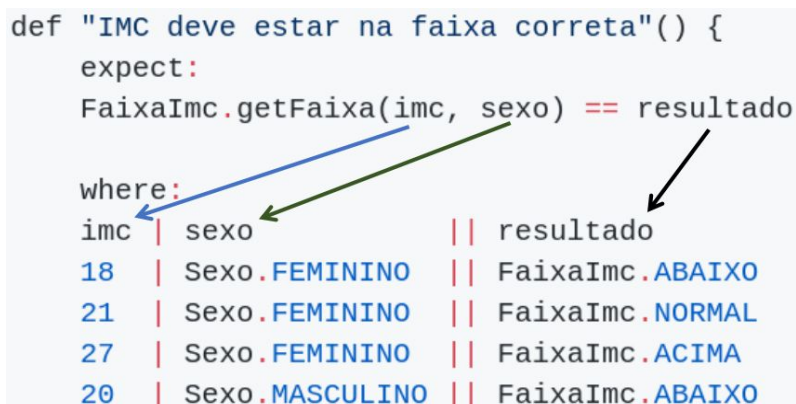
```
def 'IMC deve estar na faixa correta'() {
  expect:
  FaixaImc.getFaixa(imc, sexo) == resultado

  where:
  imc | sexo      || resultado
  18 | Sexo.FEMININO || FaixaImc.ABAIXO
  21 | Sexo.FEMININO || FaixaImc.NORMAL
  27 | Sexo.FEMININO || FaixaImc.ACIMA
  20 | Sexo.MASCULINO || FaixaImc.ABAIXO
  23 | Sexo.MASCULINO || FaixaImc.NORMAL
  28 | Sexo.MASCULINO || FaixaImc.ACIMA
}
```

Veja como a configuração da tabela de cenários ficou mais alto nível, sendo compreensível até por não programadores. Ela foi escrita, literalmente, em forma de tabela, o que reduziu a repetição e volume de código na classe de teste.

Quando usamos o recurso do **Data-Driven Testing**, o bloco **expect:** vem sempre **antes** do bloco **where:**. No **expect** indicamos o código a ser usado para o teste de cada cenário e no **where** configuramos quantos cenários acharmos necessários.

Um detalhe importante são os nomes das variáveis de entrada e saída: os **imc** e **sexo** usados pelo método **getFaixa()** e o resultado no bloco **expect:** são os mesmos das colunas **imc**, **sexo** e **resultado** da tabela no bloco **where:**. Essa relação está ilustrada na figura a seguir.



```
def "IMC deve estar na faixa correta"() {
  expect:
  FaixaImc.getFaixa(imc, sexo) == resultado

  where:
  imc | sexo      || resultado
  18 | Sexo.FEMININO || FaixaImc.ABAIXO
  21 | Sexo.FEMININO || FaixaImc.NORMAL
  27 | Sexo.FEMININO || FaixaImc.ACIMA
  20 | Sexo.MASCULINO || FaixaImc.ABAIXO
```

Na tabela de cenários, usamos o separador *pipe* (|) como separador de coluna. Usamos 2 *pipes* (||) antes da coluna de **resultado** para melhor organização, mas poderíamos ter usado apenas 1 também.

Certo, mas será que nosso enum **FaixaImc** já funciona perfeitamente? Afinal, nos testes anteriores, os cenários continham valores de IMC um tanto confortáveis, ou seja, que

não ficavam nos limites entre as faixas. Vamos acrescentar cenários que testem os limites das faixas. Vide o próximo código.

```
def 'IMC deve estar na faixa correta'() {
    expect:
    FaixaImc.getFaixa(imc, sexo) == resultado

    where:
    imc | sexo      || resultado
    18  | Sexo.FEMININO || FaixaImc.ABAIXO
    21  | Sexo.FEMININO || FaixaImc.NORMAL
    27  | Sexo.FEMININO || FaixaImc.ACIMA
    20  | Sexo.MASCULINO || FaixaImc.ABAIXO
    23  | Sexo.MASCULINO || FaixaImc.NORMAL
    28  | Sexo.MASCULINO || FaixaImc.ACIMA
    18.99 | Sexo.FEMININO || FaixaImc.ABAIXO
    19   | Sexo.FEMININO || FaixaImc.NORMAL
    25.99 | Sexo.FEMININO || FaixaImc.NORMAL
    26   | Sexo.FEMININO || FaixaImc.ACIMA
    20.99 | Sexo.MASCULINO || FaixaImc.ABAIXO
    21   | Sexo.MASCULINO || FaixaImc.NORMAL
    26.99 | Sexo.MASCULINO || FaixaImc.NORMAL
    27   | Sexo.MASCULINO || FaixaImc.ACIMA
}
```

Veja como fica mais simples acrescentar novos cenários, apenas adicionando novas linhas à tabela do bloco **where:**.

Testes com falha no Data-Driven Testing

Vamos supor que a **FaixaImc** tem um pequeno erro e falhe em alguns cenários. Nesse caso, o log de saída do teste terá a mestra estrutura que já vimos, como o texto de exemplo a seguir.

```
FaixaImc.getFaixa(imc, sexo) == resultado
```

```
  |      | | |
ABAIXO    19 | | NORMAL
           | false
           FEMININO
```

```
at unit.br.com.apostilaspoockframework.FaixaImcTest.imc deve estar na faixa correta mesmo
no limite(FaixaImcTest.groovy:52)
```

No log de exemplo, o teste falhou porque, para o cenário da 8ª linha da tabela de cenários, o resultado esperado para os IMC **19** e sexo **FEMININO** era **NORMAL**, porém o retorno do método **getFaixa()** foi **ABAIXO**.

Atenção! A falha na execução de um cenário não impede a execução dos seguintes. Por exemplo: se, dentre os 14 cenários, os das linhas 1 e 3 da tabela falharem, mesmo assim os cenários das linhas 2 e depois de 4 a 14 serão executados normalmente.

O bloco where sempre fica após a execução do teste

O bloco where sempre fica **após** os blocos de teste, ou seja, sempre após expect ou then, dependendo da opção pelo **Four-phase test** ou **Given-When-Then (GWT)**. Poderíamos, por exemplo, refazer o teste da tabela de IMC no padrão **GWT** conforme o código a seguir.

```
def 'IMC deve estar na faixa correta'() {  
  when:  
    def calculo = new Object() {  
      def getFaixa(imc, sexo) {  
        FaixaImc.getFaixa(imc, sexo)  
      }  
    }  
  
    then:  
      calculo.getFaixa(imc, sexo) == resultado  
  
  where:  
    imc | sexo      || resultado  
    18  | Sexo.FEMININO || FaixaImc.ABAIXO  
    21  | Sexo.FEMININO || FaixaImc.NORMAL  
    27  | Sexo.FEMININO || FaixaImc.ACIMA  
    // demais cenários  
}
```

No último exemplo, para aproveitar a mesma *enum* **FaixaImc** e usar o padrão **GWT**, criamos um objeto a partir de uma classe anônima com um método **getFaixa()** que apenas retorna o resultado do método homônimo da *enum*. Quanto ao teste em si, a diferença foi a troca do **expect:** por **when+then**. Note que aqui também o **where:** é após o bloco de teste que, no caso, é o **then:**.

Exception Conditions - Testes à espera de exceções

Às vezes o certo é dar errado

Vamos continuar com o exemplo do tópico anterior: o componente que dá informações baseadas no IMC. Vamos supor que precisamos agora calcular o IMC, algo que não fizemos anteriormente. O cálculo do IMC é simples. Veja na fórmula a seguir.

$$\text{IMC} = \text{peso} / \text{altura} \times \text{altura}$$

A partir dessa fórmula, poderíamos criar uma calculadora de IMC com uma classe simples como a do código a seguir.

```
public class CalculadoraImc {  
    public double calcularImc(double peso, double altura) {  
        return peso / altura * altura;  
    }  
}
```

Chamamos a classe de **CalculadoraImc**. Ela possui apenas um método, que é público, chamado **calcularImc** que, ao receber um **peso** e uma **altura**, retorna o IMC a partir da fórmula que adotamos.

Já vimos como testar o resultado do método nos tópicos anteriores. Mas, **e se algo der errado?** Aliás, reflita: *o que pode dar errado nesse cálculo? Ou ainda: esse cálculo deve ser efetuado para quaisquer valores de peso e altura?*

Vamos supor que algum desenvolvedor fez essas reflexões e chegou às seguintes conclusões sobre o método **calcularImc()**:

- Ele **não** pode receber peso com **valores menores ou iguais a 0 (zero)**. Afinal, seria um peso irreal.
- Ele **não** pode receber altura com **valores menores ou iguais a 0 (zero)**. Afinal, seria uma altura irreal.

Então, o desenvolvedor fez algumas mudanças no código do método **calcularImc()** para garantir que valores inválidos de peso e altura não provocariam a execução do cálculo e, conseqüentemente, a devolução de um IMC sem sentido. A nova versão do método está no código a seguir.

```
public double calcularImc(double peso, double altura) {  
    if (peso <= 0) {  
        throw new IllegalArgumentException("Peso inválido: "+peso);  
    }  
    if (altura <= 0) {  
        throw new IllegalArgumentException("Altura inválida: "+altura);  
    }  
    return peso / altura * altura;  
}
```

Em sua nova versão, o método **calcularImc()** não retorna o cálculo do IMC caso o peso ou altura recebidos sejam menores ou iguais a 0. Em vez disso, lança uma **IllegalArgumentException**, que é uma exceção da biblioteca padrão do Java.

Blocos try-catch

Caso você não se sinta seguro com seu conhecimento sobre tratamento de exceções e blocos **try-catch**, segue um breve resumo.

Quando podem ser usados?

Quando algum método anuncia uma **exceção checada** em sua assinatura. Isso obriga quem o invoca a tratar essa exceção, o que pode ser feito com esses blocos.

Quando algum método anuncia uma **exceção não checada** em sua assinatura, o que **não** obriga quem o invoca a tratá-la, o desenvolvedor pode usar esse bloco apenas se achar necessário. Isso costuma ser feito em situações como: registrar a ocorrência da exceção em um log de erro; evitar que um bloco de linhas código seja interrompido bruscamente; envelopar uma exceção em outra, normalmente por uma criada no projeto.

Como funcionam?

Dentro do bloco **try** há uma ou mais linhas que podem lançar exceções. Se nenhuma exceção ocorrer, todas as linhas desse bloco serão executadas.

Caso alguma exceção ocorra no bloco **try**, as linhas de código do bloco catch serão executadas. Na declaração do **catch** devem constar uma ou mais classes de exceções e o objeto no qual é instanciada a exceção. A esse objeto chamamos de **exceção capturada**. Vejamos o código a seguir.

```
try {  
    // linha a  
    // linha b  
} catch (IOException ex) {  
    // linha c  
}
```


No código de exemplo, se nenhuma exceção ocorrer no bloco try, as *linhas a e b* serão executadas. Caso contrário, a *linha c* será executada, pois está no bloco catch.

O bloco catch, se executado, possui à sua disposição na *linha c* o objeto `ex` (do tipo **IOException**), que é a exceção capturada. Caso fosse necessário tratar mais de uma exceção ao mesmo tempo, poderíamos anunciar mais de um tipo de exceção no bloco catch, como no próximo exemplo:

```
} catch (IOException | NumberFormatException ex) {  
    // linha c  
}
```

Outra opção é ter mais de um bloco catch, como no exemplo a seguir.

```
} catch (IOException ex1) {  
    // linha c  
}  
} catch (NumberFormatException ex2) {  
    // linha d  
}
```

Nesse último exemplo, caso a exceção que ocorra no bloco try seja uma **IOException**, a *linha c* seria executada. Caso ocorra uma **NumberFormatException**, a *linha d* seria executada.

Como foi dito, isso é apenas um resumo. Para saber mais sobre os blocos try-catch e tudo mais que envolve exceções com Java, uma ótima referência é o livro da Casa do Código *Java SE 8 Programmer I - O guia para sua certificação Oracle Certified Associate*, de Guilherme Silveira e Mário Amaral.

Para garantir que a execução do método **calcularImc()** lance uma exceção, vamos criar uma classe de teste chamada **CalculadoraImcTest** e usar os recursos que aprendemos até o momento para tentar testar a situação de erro desse método em caso de peso inválido.

```
class CalculadoraImcTest extends Specification {  
  
    def 'lançar exceção com peso invalido'() {  
        when:  
        boolean houveExcecao  
        try {  
            new CalculadoraImc().calcularImc(0, 1.70)  
            houveExcecao = false  
        } catch (IllegalArgumentException e) {  
            houveExcecao = true  
        }  
  
        then:  
        houveExcecao  
    }  
}
```

O teste recém-criado passará caso a execução do **calcularImc()** lance uma exceção do tipo **IllegalArgumentException**. Porém ficou um código um tanto verboso: tivemos que usar blocos **try-catch** para a montagem do teste. Caso o bloco **try** chegue ao seu final sem exceções, a variável **houveExcecao** recebe false. Caso uma **IllegalArgumentException** ocorra, **houveExcecao** recebe **true**. O valor de **houveExcecao** é que determina o resultado do teste.

É em situações como essa, em que precisamos testar se houve erro, que podemos usar o recurso *Exception Conditions* do Spock. Veja como faríamos o mesmo teste usando esse recurso no código a seguir.

```
def 'lançar exceção com peso invalido'() {  
    when:  
        new CalculadoraImc().calcularImc(0, 1.70)  
  
    then:  
        thrown(IllegalArgumentException)  
}
```

Aqui usamos o método **thrown()** da superclasse **Specification**. Esse método é uma das formas de usar **Exception Conditions**. Ele verifica se a exceção do argumento foi lançada durante a execução do teste. Ao ser executado, esse teste passaria normalmente.

Mas, vamos supor que ele tivesse sido executado quando a classe **CalculadoraImc** ainda estava na primeira versão apresentada neste tópico. Como não havia a possibilidade de ocorrer uma exceção, esse teste falharia e geraria um log como do texto a seguir.

```
lançar          exceção          com          peso          invalido  
(unit.br.com.apostilaspoockframework.topico05.CalculadoraImcTest) Time elapsed: 0.001 sec <<<  
FAILURE!  
  
org.spockframework.runtime.WrongExceptionThrownError: Expected exception of type  
'java.lang.IllegalArgumentException', but no exception was thrown
```

Note no log de exemplo que o erro foi que não aconteceu o erro esperado na execução do **calcularImc()**. Não foi lançada uma **IllegalArgumentException**.

Um detalhe importante: o método **thrown** **só pode estar dentro de blocos then**. Se estiver dentro de um **expect**, a classe de testes sequer compila.

Testando vários cenários de erro

Um componente pode ter vários cenários que provoquem erro. E é o caso de nosso método **calcularImc()**, que pode lançar exceções para valores de peso ou altura inválidos. Para testar os vários cenários, poderíamos simplesmente criar vários métodos de teste, um para cada cenário. Porém, isso é desnecessário com o Spock, já que é possível repetir os blocos **when:** e **then:** para testar diferentes cenários que esperam exceções. Vejamos como

testar mais de um cenário que provoque exceção para valores inválidos de peso no próximo código.

```
def 'lançar exceção com peso invalido'() {  
  when:  
    new CalculadoraImc().calcularImc(0, 1.70)  
  
  then:  
    thrown(IllegalArgumentException)  
  
  when:  
    new CalculadoraImc().calcularImc(-1, 1.70)  
  
  then:  
    thrown(IllegalArgumentException)  
}
```

Verificando detalhes das exceções

Até agora vimos como verificar se uma determinada exceção foi lançada com a execução de um método. Mas pode ser necessário testar se a exceção que ocorreu possui detalhes desejados.

Na última versão do método de cálculo de IMC apresentada, as exceções continham mensagens específicas para erros de peso e altura. Vamos supor que seja necessário garantir que as mensagens sejam exatamente aquelas. Poderíamos criar um novo método de teste na **CalculadoraImcTest**, como o do código a seguir.

```
def 'lançar exceção c/ mensagem correta p/ peso inválido'() {  
  when:  
    def peso = -1  
    new CalculadoraImc().calcularImc(peso, 1.70)  
  
  then:  
    def ex = thrown(IllegalArgumentException)  
    ex.message == "O peso $peso é inválido"  
}
```

Note que a diferença para o que fizemos até agora é que usamos o retorno do método **thrown()** para guardar a exceção lançada no objeto ex. Com o ex à disposição, podemos testar quaisquer detalhes que precisarmos da exceção. No exemplo, verificamos se a mensagem da exceção contém um determinado texto.

De propósito, testamos uma mensagem diferente da que realmente é usada na exceção lançada no método de cálculo do IMC. Ao executar o teste, ele falhará e produzirá um log de erro, como o do próximo texto.

```

lancar      exceção      c/      mensagem      correta      p/      peso
inválido(unit.br.com.apostilaspoackframework.topico05.CalculadoraImcTest)  Time elapsed: 0.015
sec <<< FAILURE!
org.spockframework.runtime.SpockComparisonFailure: Condition not satisfied:

ex.message == "O peso $peso é inválido"
| |          |          |
| |          |          -1
| |          false
| |          14 differences (30% similarity)
| |          (P--)eso (-----)inválido(: -1.0)
| |          (O p)eso (-1 é )inválido(-----)
| |          Peso inválido: -1.0
java.lang.IllegalArgumentException: Peso inválido: -1.0

```

Para que o último teste passe, basta ajustar o bloco then para esperar pela mensagem correta, como no próximo código:

```

then:
def ex = thrown(IllegalArgumentException)
ex.message == "Peso inválido: ${peso.toDouble()}"

```

Nos exemplos deste tópico verificamos o conteúdo da mensagem da exceção lançada, mas poderíamos ter testado qualquer outro atributo ou retorno de método de ex que fosse necessário. Por exemplo, poderíamos verificar a exceção anterior na pilha de exceções, por meio do método **getCause()**.

Testando com a ajuda de Mocks

Até aqui já vimos como criar vários tipos de testes com Spock: testes simples, testes que exigem vários cenários e testes que esperam por exceções. Neste tópico veremos como trabalhar com *dublês*, que são objetos que imitam o comportamento de componentes que dão suporte a um teste, mas que não são seus protagonistas.

O que são dublês e por que usá-los

Para a execução de alguns tipos de testes, fica complexo e até arriscado usar todos os componentes reais envolvidos na funcionalidade a ser testada. Imagine que você precisa testar funcionalidades como: um método que exclui centenas de registros em um banco de dados; um método que envia centenas de e-mails; um método que traz centenas de objetos JSONs de uma REST API. Em todos esses exemplos estamos sujeitos a uma série de problemas.

Ou seja, há testes nos quais usar todos os componentes reais envolvidos pode torná-los mais **complexos** do que deveriam. Outro problema no uso de certos tipos de componentes é o fato de estarem expostos à **dependência de fatores externos**. Existem ainda componentes cujo uso em testes ser até **perigoso**. São exemplos de componentes com uma ou mais dessas características aqueles que:

- Exigem muitas configurações;
- Exigem muitas dependências;
- Têm necessidade de estar em uma rede;
- Têm necessidade de acesso à internet;
- São afetados pela latência de rede;
- Têm a necessidade de licença de uso de software;
- Dependem da disponibilidade de algum serviço externo;
- Dependem do desempenho de algum serviço externo;
- Podem afetar dados sensíveis em bases de dados;
- Podem realizar ações que geram inconvenientes, como o envio de excesso de e-mails para clientes reais, por exemplo;
- Podem levar ao aumento custos financeiros pois podem consumir serviços que são tarifados por tempo e/ou volume de uso.

Porém, imagine que nos exemplos citados no início deste tópico:

- Os registros não precisam ser realmente excluídos do banco. Só é necessário garantir que houve a solicitação para excluí-los;
- Os e-mails não precisam ser realmente enviados. Só é necessário garantir que ocorreram as solicitações de envio;
- Não é necessário obter os JSON realmente da API. Só é necessário ter objetos JSONs na quantidade e no formato desejados.

Note que as funcionalidades que poderiam acrescentar complexidade e/ou perigo aos testes **não são o alvo dos testes**. Para situações assim, podemos substituir alguns componentes por **dublês**, que são objetos que imitam outros.

Existem vários tipos de dublês, sendo os principais dummy, fake, stub, spy e mock. Esses termos foram sugeridos por Gerard Meszaros, em sua obra *xUnit Test Patterns: Refactoring Test Code*, de 2007: ([http://xunitpatterns.com/Mocks, Fakes, Stubs and Dummies.html](http://xunitpatterns.com/Mocks,_Fakes,_Stubs_and_Dummies.html)) e também usados por Martin Fowler em um artigo seu no mesmo ano (<https://martinfowler.com/articles/mocksArentStubs.html>).

Tipos de dublês

Dummy: é o tipo de dublê mais simples. É usado apenas para preencher passagens de parâmetros. São valores simples como pequenos alfanuméricos, números, uma data qualquer, textos em branco ou até valores nulos.

Fake: esse tipo de dublê é usado principalmente para contornar problemas de dependências. São objetos simples, que não guardam estado. Um exemplo seria criar uma classe **FakeDriver** que imita uma **Driver** (um Driver JDBC), possuindo métodos públicos com a mesma assinatura. Assim, dispensamos toda uma lista de dependências e podemos realizar o teste cujo foco não era o Driver JDBC e sim o que acontece após invocar alguma funcionalidade dele. Os *Fakes* sempre apresentam o mesmo comportamento para suas funcionalidades. Por exemplo, um objeto *Fake* de uma classe **FakeDriver** sempre retornaria o mesmo valor em um método `connect()` independentemente dos argumentos usados. Devido à sua simplicidade, são normalmente criados manualmente, ou seja, sem auxílio de bibliotecas de *Mocks*.

Stub: é muito parecido com o *Fake*. A diferença é que podem ser configurados diferentes comportamentos para os métodos conforme os argumentos usados. Ou seja, um mesmo objeto *Stub* pode ter diferentes comportamentos para diferentes cenários de teste. São normalmente criados com auxílio de alguma biblioteca de *Mocks*.

Spy: são dublês que permitem saber detalhes que ocorreram durante a execução de seus métodos. Por exemplo, um Driver JDBC deveria receber 3 vezes a solicitação para invocar seu método para executar instruções SQL caso o método que está sendo testado receba uma lista com 6 elementos. Se usarmos um *Spy* para imitar o Driver JDBC, podemos obter dele a informação de quantas vezes qualquer um de seus métodos foi executado

durante um teste. São normalmente criados com auxílio de alguma biblioteca de *Mocks*.

Mocks: são os dublês mais completos. Permitem a configuração de diferentes comportamentos de acordo com as entradas (como um *Stub*) assim como permitem saber quantas vezes seus métodos foram executados (como um *Spy*). São normalmente criados com auxílio de alguma biblioteca de *Mocks*.

Dependências adicionais para Mocks

Para que seja possível trabalhar com *Mocks* no Spock, é necessário incluir a dependência de 2 bibliotecas: **objenesis** e **byte-buddy**. Elas serão usadas de forma transparente pelo Spock. A seguir, o código que deve ser incluído no pom.xml do projeto para adicioná-las.

```
<dependency>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy</artifactId>
  <version>1.6.5</version> <!-- ou a versão que precisar -->
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.objenesis</groupId>
  <artifactId>objenesis</artifactId>
  <version>2.5.1</version> <!-- ou a versão que precisar -->
  <scope>test</scope>
</dependency>
```

A versões dessas dependências podem ser as de que precisar ou preferir. A sugestão é deixá-las no escopo test do projeto, para evitar seu uso por engano fora dos testes.

Configurando retornos de métodos em Mocks

O tipo de Mock mais simples de que podemos precisar é aquele em que apenas programamos os retornos de seus métodos. Imagine uma situação em que precisamos invocar métodos de um objeto cuja criação é muito difícil de reproduzir usando sua classe real (é instanciada via inversão de controle, por exemplo), porém, o que realmente precisamos para um teste é apenas usar o retorno de alguns de seus métodos. Para situações como essa, é possível criar e configurar o retorno de métodos de Mocks.

Vamos usar como exemplo a mesma *Calculadora de IMC* dos tópicos anteriores. Vamos supor que foi necessário evoluir o projeto e foi criada uma interface chamada **PedidoImc** cuja implementação em tempo de execução ocorre por meio de mecanismos complexos de reproduzir, como inversão de controle, por exemplo. Seu código-fonte está a seguir.

```
public interface PedidoImc {  
    String getNome();  
    double getPeso();  
    double getAltura();  
    Sexo getSexo();  
}
```

E, para representar o resultado de um cálculo de IMC, foi criada também uma classe, a **ResultadoImc**, cujo código está a seguir.

```
public class ResultadoImc {  
    private String nome;  
    private double imc;  
    private String condicao;  
  
    public ResultadoImc(String nome, double imc) {  
        super();  
        this.nome = nome;  
        this.imc = imc;  
    }  
    // construtor padrão, getters e setters  
}
```

O atributo nome é o nome da pessoa de quem foi solicitado o cálculo. O imc é o valor do IMC calculado. O **condicao** é a descrição da condição de saúde da pessoa correspondente ao seu IMC.

As interface e classe recém-criadas agora são usadas na classe **CalculadoraImc**, a mesma dos tópicos anteriores. Observe o método **verificarCondicaoImc()** criado nessa classe.

```
public class CalculadoraImc {  
    // mesmo método calcularImc() dos tópicos anteriores  
  
    public ResultadoImc verificarCondicaoImc(PedidoImc pedido) {  
        ResultadoImc resultado = new ResultadoImc();  
        resultado.setNome(pedido.getNome());  
  
        try {  
            double imc = this.calcularImc(pedido.getPeso(), pedido.getAltura());  
            resultado.setImc(imc);  
            resultado.setCondicao(  
                FaixaImc.getFaixa(imc, pedido.getSexo()).getDescricao());  
        } catch (IllegalArgumentException e) {  
            resultado.setCondicao("Impossível calcular IMC: "+e.getMessage());  
        }  
        return resultado;  
    }  
}
```


O método **verificarCondicaoImc()** recebe um objeto do tipo **PedidoIMC** e retorna um objeto do tipo **ResultadoImc**. Esse método preenche o atributo nome do resultado com o valor do atributo de mesmo nome do pedido. Caso o IMC seja calculado, seu valor é atribuído ao atributo imc do resultado e o atributo **condicao** é preenchido conforme a descrição obtida em **FaixaImc**. Caso o IMC não seja calculado devido a valores inválidos de **peso** ou **altura**, a **condicao** recebe um texto padrão sobre a impossibilidade do cálculo.

Vamos então criar nosso primeiro Mock com Spock para podermos testar o método **calcularImc()** da **CalculadoraImc**. Ele será criado na classe de teste já existente **CalculadoraImcTest**, como pode ser visto no método do código-fonte a seguir.

```
class CalculadoraImcTest extends Specification {

    def 'deveria ser um homem acima do peso'() {
        given:
            def pedido = Mock(PedidoImc)

            pedido.getNome() >> "Zé Fofinho"
            pedido.getSexo() >> Sexo.MASCULINO
            pedido.getPeso() >> 72
            pedido.getAltura() >> 1.50

        when:
            def resultado = new CalculadoraImc().verificarCondicaoImc(pedido)

        then:
            resultado.imc == 32
            resultado.condicao == FaixaImc.ACIMA.descricao
    }

    // demais testes
}
```

A criação do Mock ocorreu no bloco **given;**, onde usamos o método **Mock()** que vem da superclasse **spock.lang.Specification** do Spock. Esse método exige como argumento um tipo indicando para qual classe será criada um dublê. Logo, no código do exemplo, criamos um Mock para a interface **PedidoImc**, que chamamos de pedido.

Logo após a criação do Mock, programamos os retornos dos métodos de que precisaremos em nosso teste. Por meio do operador **>>** configuramos os retornos dos métodos **getNome()**, **getSexo()**, **getPeso()** e **getAltura()**. Se não tivéssemos configurado explicitamente os retornos desses 4 métodos, o pedido retornaria valores padrões de acordo com os tipos de retorno de cada método.

O que métodos de Mocks retornam?

Quando acabamos de criar um Mock, podemos invocar todos os métodos da classe original. Mas, se não programarmos seus retornos, o que eles retornam? Eles retornam valores padrão conforme seus tipos.

Tipos primitivos Métodos com retornos de tipos primitivos retornam o valor padrão de cada tipo primitivo conforme define a linguagem Java:

- Para **int**, **short** e **byte** é **0**;
- Para **long** é **0L**;
- Para **double** é **0.0**;
- Para **float** é **0.0f**;
- Para **char** é **'\u0000'**;
- Para **boolean** é **false**.

Classes (inclusive Wrappers) Métodos com retorno de classes retornam null por padrão. Métodos com retorno de classes *wrappers* (Integer, Double, Boolean etc.) também retornam null por padrão.

Vetores e coleções Métodos com retorno de vetores e coleções retornam null por padrão, mesmo que se trate de um vetor de um tipo primitivo.

Métodos sem retorno (void) Por padrão, métodos sem retorno (retorno void) são executados sem erro. A execução é instantânea, como se o método nem tivesse sido executado.

Assim, com o Mock criado e configurado, durante a invocação do método **verificarCondicaoImc()**, o argumento do tipo **PedidoImc** retorna os valores programados no Mock do teste. Ao final da execução, este método retorna uma instância de **ResultadoImc** que pode ser analisada normalmente no bloco then do teste.

Só é possível indicar um retorno para métodos dos Mocks? E se precisar de algo mais dinâmico?

Na verdade, o operador **>>** pode ser usado também para indicar **todo o corpo do método de um Mock**. No código do exemplo anterior, poderíamos, por exemplo, ter definido o retorno do método que indica o nome a partir do retorno do método que indica o sexo, como no código a seguir.

```
def 'deveria ser um homem acima do peso'() {
  given:
  def pedido = Mock(PedidoImc)

  pedido.getNome() >> {
    if (this.getSexo() == Sexo.Masculino) {
      "Zé Fofinho"
    } else {
      "Maria Fofinha"
    }
  }
}

// demais código do teste
```

Nesse código, indicamos não apenas um valor de retorno do método **getNome()**, mas todo o seu corpo, no qual definimos o retorno conforme o que retornar o método **getSexo()** do próprio Mock.

Configurando métodos de Mocks para aceitarem qualquer valor

Imagine a situação em que você precisa configurar o retorno de um método que possui um ou mais argumentos. Só que o valor de algum dos argumentos (ou todos) não é relevante para o teste ou é impossível de prever, ou podem receber qualquer valor. Nesse caso, podemos configurar o Spock para determinar o retorno do método com um ou mais argumento recebendo qualquer valor válido.

Vamos supor que a interface **PedidoImc** possua mais 2 métodos com argumentos, como no código a seguir.

```
public interface PedidoImc {
  String getNome(String pronomeTratamento);
  String getNome(String pronomeTratamento, String apelido);
  // demais métodos
}
```

Para configurar o retorno das 2 novas versões do método **getNome()** sem nos importarmos com os valores dos argumentos, poderíamos configurar um Mock de **PedidoImc** usando o operador ***underline*** (_) como no código de teste Spock a seguir.

```
def pedido = Mock(PedidoImc)

pedido.getNome(_) >> "Sr. Zé Importante"
pedido.getNome(_, _) >> "Sr. Zé Imitador, vulgo 'ator'"
pedido.getNome(_, 'alpinista') >> "Sr. Zé Luiz, vulgo 'alpinista'"
pedido.getNome('Sr.', _) >> "Sr. Zé Boleiro, vulgo 'messi'"
```

Nesse código realizamos 4 configurações de retornos para as 2 novas versões método **getNome()**. Foram todas aplicadas no mesmo objeto Mock. Os retornos ficaram assim configurados:

- Na 1ª configuração: ao ser invocado com qualquer valor no único argumento, retorna "Sr. Zé Importante".
- Na 2ª configuração: ao ser invocado com quaisquer valores no primeiro e segundo argumentos, retorna "Sr. Zé Imitador, vulgo 'ator'".
- Na 3ª configuração: ao ser invocado com qualquer valor no primeiro argumento e 'alpinista' no segundo, retorna "Sr. Zé Luiz, vulgo 'alpinista'".
- Na 4ª configuração: ao ser invocado com 'Sr.' no primeiro argumento e qualquer valor no segundo argumento, retorna "Sr. Zé Boleiro, vulgo 'messi'".

Configurando Mocks para lançarem exceções

Imagine situações em que o comportamento de um componente pode lançar exceções em situações complexas de reproduzir a qualquer momento, como problemas de rede ou indisponibilidade de serviço externo, por exemplo. Para situações como esta, é possível configurar um Mock para lançar exceções na invocação de métodos.

Criando um Mock que lança uma exceção simples

Vamos supor que você precisa de um componente de envio de e-mails. Para facilitar esse tipo de tarefa, você optou por usar a biblioteca *Apache Commons Email* (<https://commons.apache.org/proper/commons-email/>) e criou a classe **EnvioEmailService**, cujo código-fonte está a seguir.

```
import org.apache.commons.mail.Email;
import org.apache.commons.mail.EmailException;

public class EnvioEmailService {

    private Email email;

    private int enviados;

    public EnvioEmailService(Email email) {
        this.email = email;
    }

    public void enviarEmails(String assunto, String conteudo, List<String> destinatarios) throws EmailException {
        this.email.setSubject(assunto);
        this.email.setMsg(conteudo);
        this.email.addTo(destinatarios.toArray(new String[destinatarios.size()]));

        this.email.send();
    }
}
```

```
        this.enviados++;  
    }  
  
    // getEnviados()  
}
```

A classe **EnvioEmailService** possui um atributo do tipo `org.apache.commons.mail.Email` que é atribuído no construtor da classe. É em objetos dessa classe da *Apache Commons Email* que são configurados os dados de acesso ao servidor de e-mail.

O método **enviarEmails()** usa seus argumentos para invocar os métodos do atributo de instância `email` e, caso tudo dê certo, o atributo `enviados` aumenta seu valor em **1**. Dentre os métodos invocados, o método **send()** anuncia uma **`org.apache.commons.mail.EmailException`**. Ao estudar a API da *Apache Commons Email*, verificamos que essa exceção sempre traz o motivo do erro. Exemplos: para problemas de autenticação, a causa é uma **`javax.mail.AuthenticationFailedException`**; para problemas de conexão, a causa é uma **`com.sun.mail.util.MailConnectException`**. Existem várias outras causas de erro para o envio de e-mails, mas vamos usar apenas essas duas em nosso exemplo. Caso o e-mail seja enviado em erros, o atributo de instância `enviados` aumenta seu valor em **1**.

Se não fosse possível criar Mocks para configurar o lançamento das exceções no envio de e-mail, teríamos que ficar solicitando de fato o envio de e-mails. Os possíveis problemas dessa abordagem seriam vários: estaríamos sujeitos à latência de rede; poderíamos bloquear uma conta ou mesmo cair em uma *black list* ao tentar muitas autenticações sem sucesso; seria complicado simular problemas de conexão de rede durante os testes; poderíamos acabar tendo que pagar pelo excessivo envio de e-mails, dependendo do contrato com o provedor.

Vamos então criar uma classe de teste chamada **EnvioEmailServiceTest**, na qual vamos usar Mocks que lançam **`org.apache.commons.mail.EmailException`** com a devida causa para evitar que, para testar nossa **EnvioEmailService**, precisamos enviar e-mails de fato. Vejamos o código dessa classe de teste a seguir.

```
import org.apache.commons.mail.Email  
// demais imports  
  
class EnvioEmailServiceTest extends Specification {  
  
    Email email  
    EnvioEmailService service  
  
    def setup() {  
        this.email = Mock(Email)  
        this.service = new EnvioEmailService(this.email)  
    }  
}
```

A parte do código da **EnvioEmailServiceTest** por enquanto, não possui testes. Nela, declaramos dois atributos: o email, do tipo **org.apache.commons.mail.Email**, que será instanciado com um Mock, e o service do tipo da classe que queremos testar, a **EnvioEmailService**. O Mock é criado no método **setup()**, onde é usado como argumento do construtor usado para instanciar o service. Vale lembrar que o **setup()** será invocado sempre antes de cada método de teste.

Agora vamos criar um teste que espera por exceção ocasionada por problema de conexão com o servidor de e-mail. Para isso, vamos configurar o Mock para a lançar exceção esperada para que ela ocorra quando usamos o service. Vejamos no próximo código.

```
import javax.mail.AuthenticationFailedException

class EnvioEmailServiceTest extends Specification {

  // atributos e setup()

  def 'deveria lançar erro de autenticação'() {
    given:
      def msgErro = 'Falha de autenticação! Sorry :('

      this.email.send() >> {
        throw new EmailException(new AuthenticationFailedException(msgErro))
      }

    when:
      this.service.enviarEmails('assunto', 'conteudo', ['em@t.com'])

    then:
      def ex = thrown(EmailException)
      ex.cause.class == AuthenticationFailedException.class
      ex.cause.message == msgErro

    !this.service.enviados
  }
}
```

No bloco **given** apenas definimos a mensagem de erro que será usada no Mock e para posterior comparação com o resultado da execução do teste. Em seguida, configuramos nosso Mock, o email para que, quando invocar o método **send()** lance uma **EmailException** cuja causa é uma **javax.mail.AuthenticationFailedException**. Observe que definimos um bloco de código com o operador **>>**. É que, com Spock, podemos definir todo o corpo do método do Mock, e foi isso que foi feito no exemplo.

No bloco **when:** apenas invocamos o método que queremos testar. E no bloco **then:** testamos:

- se foi lançada uma **EmailException**;
- se a causa da exceção foi uma **AuthenticationFailedException**;
- se a mensagem da causa foi a definida no Mock (lá no bloco **given:**);

- se o atributo `enviados` continua zerado.

Ratificando: sem o uso de Mock nosso teste seria mais lento na execução (pois estaríamos sujeitos à latência de rede e desempenho do servidor de e-mail) além de correr o risco de bloquear uma conta devido às falhas de autenticação ou mesmo entrar em uma *black list* do serviço de e-mail. E, com uso de Mock, não precisamos configurar a autenticação do objeto do tipo Email, o que nos poupou algumas linhas de código.

Criando um Mock que lança uma exceção com muitos detalhes

No exemplo anterior, a exceção lançada era bem simples, bastando criar uma **AuthenticationFailedException** indicando sua mensagem de erro. E a mensagem que indicamos acabou se tornando literalmente a mensagem da exceção. Porém, algumas exceções não são tão simples assim. A seguir, vamos ver como testar outro possível problema em nosso serviço de envio de e-mails.

O outro tipo de exceção que vamos testar agora é a **com.sun.mail.util.MailConnectException**. Diferente da situação do tópico anterior, ela exige mais detalhes para ser criada e sua mensagem de erro não pode ser definida na criação. Vejamos como criar o Mock para essa exceção no método de teste que também ficará na **EnvioEmailServiceTest**, cujo código está a seguir.

```
def 'deveria lançar erro de conexão'() {
    given:
    def servidor = 'teste.com'
    def porta = 7777
    def timeout = 1

    this.email.send() >> {
        def sce = new SocketConnectException("", null, servidor, porta, timeout)
        throw new EmailException(new MailConnectException(sce))
    }

    when:
    this.service.enviarEmails('assunto', 'conteudo', ['em@t.com'])

    then:
    def ex = thrown(EmailException)
    ex.cause.class == MailConnectException.class
    ex.cause.message.contains(servidor)
    ex.cause.message.contains(porta.toString())

    !this.service.enviados
}
```

No bloco `given` indicamos o servidor, porta e *timeout* que serão usados na criação do Mock do teste. Em seguida, configuramos o método **send()** para lançar uma **EmailException** na qual definimos que a causa é uma **com.sun.mail.util.SocketConnectException**. Para definir essa causa, não poderíamos usar um construtor que apenas define a mensagem, pois ele não existe na **SocketConnectException**. Tínhamos que definir uma **mensagem de**

detalhe (que não se torna a mensagem da exceção, por isso ignoramos), uma **causa** (que ignoramos, por isso null), um **servidor**, uma **porta** e um **timeout**, esses três últimos, definidos há pouco e que poderiam ter quaisquer valores válidos.

No bloco **when:** apenas invocamos o método que queremos testar com quaisquer argumentos válidos. E no bloco then testamos:

- se foi lançada uma **EmailException**;
- se a causa da exceção foi uma **SocketConnectException**;
- se a mensagem da causa contém o servidor e a porta definidos no bloco **given:**. A mensagem da **SocketConnectException** é bem longa, mas contém o servidor e a porta que foram usados na tentativa de conexão com o servidor;
- se o atributo enviados continua zerado.

Configurando retornos de métodos aninhados em Mocks

Já vimos como configurar os retornos dos métodos de um Mock. Porém, há situações em que precisamos de retornos de métodos de objetos que já foram obtidos de retornos de Mocks. Um exemplo disso seria se precisássemos salvar os IMCs das pessoas em uma tabela de um banco de dados relacional e usássemos JPA. Imagine que a tabela para registro do IMC atual das pessoas tivesse sido mapeada pela classe **ImcAtual**, cujo código está a seguir.

```
import javax.persistence.Column;
import javax.persistence.Entity;

@Entity
public class ImcAtual {

    @Column(length=50)
    private String nome;

    @Column
    private double imc;

    public ImcAtual(String nome, double imc) {
        super();
        this.nome = nome;
        this.imc = imc;
    }
    // construtor padrão, getters e setters
}
```

E, para salvar um único IMC por pessoa, criamos uma outra classe chamada **ImcAtualService**. Ela deve garantir que uma pessoa só possua um registro de IMC, sempre com o mesmo valor calculado. Seu código está a seguir.


```

import java.sql.SQLException;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;

public class ImcAtualService {

    private static final String CONSULTA_IMC =
        "SELECT i FROM ImcAtual i WHERE nome = ?0";

    private EntityManager entityManager;

    private ImcAtual imcExistente(ResultadoImc resultado) {
        try {
            return this.entityManager
                .createQuery(CONSULTA_IMC, ImcAtual.class)
                .setParameter(0, resultado.getNome())
                .getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }

    public int registrarImc(List<ResultadoImc> resultados) throws SQLException {

        int inseridos = 0;
        for (ResultadoImc resultado : resultados) {
            ImcAtual imcExistente = this.imcExistente(resultado);
            if (imcExistente != null) {
                imcExistente.setImc(resultado.getImc());
                continue;
            }
            ImcAtual imcAtual = new ImcAtual(resultado.getNome(), resultado.getImc());
            this.entityManager.persist(imcAtual);
            inseridos++;
        }

        return inseridos;
    }
}

```

O método **imcExistente()** tenta retornar uma instância de **ImcAtual** a partir de um nome. Caso nenhum registro seja encontrado, o método retorna null. O método **registrarImc()** retorna o número de registros novos inseridos, porém atualiza os existentes. Ele invoca o **imcExistente()** e, caso encontre registro com o nome, apenas atualiza seu IMC. Caso contrário, cria um novo registro.

Essa classe seria fácil de testar não fosse um detalhe: seus métodos usam um atributo de instância privado do tipo **javax.persistence.EntityManager**, chamado **entityManager**. As dificuldades surgem devido aos seguintes fatos:

- Configurar uma Persistence Unit do JPA não é simples, pois exigiria uma série de dependências e configurações, além de tornar nosso teste dependente da

disponibilidade e desempenho de um banco de dados. Como o objeto do tipo **EntityManager** não é o foco do teste, podemos simplesmente criar um Mock dele.

- No método **imcExistente()** usamos 3 métodos de forma aninhada sobre o **entityManager**: primeiro **createQuery()** e, sobre o retorno deste, **setParameter()** e, sobre o retorno deste, **getSingleResult()**. Para conseguirmos testar essa situação, teremos que criar vários Mocks e configurar seus retornos para que imitem essa chamada aninhada de métodos.

Vejam agora como poderia ficar a classe de testes. Seguindo nosso padrão, vamos chamá-la de **ImcAtualServiceTest**. Vejamos seu código na sequência. Como ele ficaria um tanto grande, vamos transcrevê-lo e explicá-lo por partes.

```
import javax.persistence.TypedQuery
// demais imports

class ImcAtualServiceTest extends Specification {
  String nomeEncontrado = 'Zé Cadastrado'
  String nomeNaoEncontrado1 = 'João Não Cadastrado'
  String nomeNaoEncontrado2 = 'Maria Não Cadastrada'

  EntityManager entityManager
  ImcAtualService service

  def setup() {
    this.entityManager = Mock(EntityManager)

    def resultadoEncontrado = Mock(TypedQuery)
    def resultadoNaoEncontrado = Mock(TypedQuery)
    def instrucaoConsulta = Mock(TypedQuery)

    resultadoEncontrado.getSingleResult() >> new ImcAtual(nome: this.nomeEncontrado)

    instrucaoConsulta.setParameter(0, this.nomeEncontrado) >> resultadoEncontrado
    instrucaoConsulta.setParameter(0, this.nomeNaoEncontrado1) >> resultadoNaoEncontrado
    instrucaoConsulta.setParameter(0, this.nomeNaoEncontrado2) >> resultadoNaoEncontrado

    this.entityManager.createQuery(ImcAtualService.CONSULTA_IMC, ImcAtual.class) >>
    instrucaoConsulta

    this.service = new ImcAtualService(entityManager:this.entityManager)
  }
  // métodos de teste
}
```

Neste código temos apenas os atributos de instância, que serão usados em todos os testes, e o método **setup()** que, como vimos anteriormente, será executado sempre antes de qualquer outro método de teste. Os atributos são:

- 3 nomes, sendo 1 que será usado para simular situações de nome já existente no banco de dados (**nomeEncontrado**) e 2 para simular nomes que não estão (**nomeNaoEncontrado1** e **nomeNaoEncontrado2**).

- 1 objeto do tipo **EntityManager** que será instanciado por um Mock;
- 1 objeto do tipo **ImcAtualService** que será um objeto real (não Mock) usado nos testes dessa classe.

No método **setup()** configuramos um Mock simples para o **entityManager** e 3 Mocks para objetos do tipo **javax.persistence.TypedQuery** (**resultadoEncontrado**, **resultadoNaoEncontrado** e **instrucaoConsulta**).

Na sequência configuramos os Mocks que serão usados de forma aninhada durante os testes:

- Método **getSingleResult()** a partir de **resultadoEncontrado**: retornará uma instância de **ImcAtual** com seu nome igual ao valor de **nomeEncontrado**;
- Método **instrucaoConsulta()**, com 0 e **nomeEncontrado** como argumentos, a partir de **instrucaoConsulta**: retornará o **resultadoEncontrado**;
- Método **instrucaoConsulta()**, com 0 e **nomeNaoEncontrado1** como argumentos, a partir de **instrucaoConsulta**: retornará o **resultadoNaoEncontrado**;
- Método **instrucaoConsulta()**, com 0 e **nomeNaoEncontrado2** como argumentos, a partir de **instrucaoConsulta**: retornará o **resultadoNaoEncontrado**;
- Método **createQuery()** a partir de **entityManager**: retornará o **instrucaoConsulta**.

Não foi necessário indicar o retorno do **getSingleResult()** a partir de **resultadoNaoEncontrado**. Como já explicado neste tópico, métodos de Mocks que não retornam primitivos, retornam null por padrão.

A instrução final no método **setup()** é instanciar o service e preenchendo o seu **entityManager** com o Mock do **EntityManager** recém-criado. Caso não sabia como funciona o construtor usado para instanciar um **ImcAtual** e um **ImcAtualService**, veja o item **É possível atribuir valores de atributos na criação de objetos** no *Apêndice A - Guia de Groovy para desenvolvedores Java*.

Com os Mocks devidamente configurados, podemos testar o método privado **imcExistente()** como no método de teste a seguir.

```
def 'deveria retornar null ou o ImcAtual por nome'() {
    expect:
    !this.service.imcExistente(this.nomeNaoEncontrado1)
    !this.service.imcExistente(this.nomeNaoEncontrado2)
    this.service.imcExistente(this.nomeEncontrado)?.getNome() == this.nomeEncontrado
}
```

Como não havia necessidade de nenhum preparo adicional, esse método poderia conter apenas o bloco **expect:**. Nesse bloco, verificamos se a invocação do método **imcExistente()**:

- Com os nomes que não deveriam ser encontrados pelo **EntityManager** retorna null;

- Com o nome que deveria ser encontrado pelo **EntityManager** retorna uma instância de **ImcAtual** cujo nome é igual ao atributo **nomeEncontrado**.

No próximo teste verificaremos se o método **registrarImc()** realmente só retorna a quantidade de registros inseridos. Vejamos esse teste no próximo código-fonte.

```
def 'deveria inserir somente se ainda não existe'() {  
    when:  
    def resultadosE2 = [  
        new ResultadoImc(nome:nomeEncontrado, imc: 25),  
        new ResultadoImc(nome:nomeNaoEncontrado1, imc: 29),  
        new ResultadoImc(nome:nomeNaoEncontrado2, imc: 35)  
    ]  
  
    def resultadosE1 = [  
        new ResultadoImc(nome:nomeEncontrado, imc: 25),  
        new ResultadoImc(nome:nomeNaoEncontrado1, imc: 29)  
    ]  
  
    def resultadosE0 = [  
        new ResultadoImc(nome:nomeEncontrado, imc: 25)  
    ]  
  
    then:  
    this.service.registrarImc(resultadosE2) == 2  
    this.service.registrarImc(resultadosE1) == 1  
    this.service.registrarImc(resultadosE0) == 0  
}
```

Nesse teste, preparamos 3 listas de **ResultadoImc** que serão usadas nos testes no bloco **when::**

- **resultadosE2**: lista que contém 1 nome que deve ser encontrado e 2 que não devem;
- **resultadosE1**: lista que contém 1 nome que deve ser encontrado e outro que não;
- **resultadosE0**: lista que contém apenas 1 nome que deve ser encontrado.

No bloco then verificamos se a invocação do método **registrarImc()** retorna:

- Valor 2 quando invocado com **resultadosE2**;
- Valor 1 quando invocado com **resultadosE1**;
- Valor 0 quando invocado com **resultadosE0**.

Verificando a quantidade de execuções de métodos dos Mocks

Uma das motivações para o uso de Mocks expostas neste tópico foi a de que há situações em que não precisamos que um componente faça sua ação real e sim que seja

solicitado a fazê-la. Mantendo a classe **ImcAtualService** como exemplo, imagine que queremos ter certeza de que o método **persist()** do **EntityManager** dela seja invocado uma certa quantidade de vezes durante a execução do método **registrarImc()**. Como os Mocks do Spock também são **Spies**, é possível saber quantas vezes um método de um Mock foi executado.

Não precisamos mexer em nada na **ImcAtualService**. Basta criarmos um novo teste na classe de testes **ImcAtualServiceTest**, que está no código-fonte a seguir.

```
def 'deveria criar somente como novo por nome'() {
  given:
  def resultados = [
    new ResultadoImc(nome:nomeEncontrado, imc: 25),
    new ResultadoImc(nome:nomeNaoEncontrado1, imc: 29),
    new ResultadoImc(nome:nomeNaoEncontrado2, imc: 35)
  ]

  when:
  this.service.registrarImc(resultados)

  then:
  2 * this.entityManager.persist(_)
}
```

No cenário programado no último método de teste criamos uma lista de 3 objetos do tipo **ResultadoImc**, em que 1 possui um nome já cadastrado e 2 não cadastrados. Portanto, o método **persist()** do **EntityManagerdeveria** ser invocado 2 vezes quando usamos o método **registrarImc()** usando essa lista. Nos blocos **given:** e **when:** não há novidades: configuração e execução do método de teste, respectivamente.

A grande novidade está no bloco **then:**. A linha que existe nele passará no teste se o método **persist()** for invocado **2 vezes** com qualquer argumento durante a execução do código do bloco **when:**. Isso foi possível porque o Mock **entityManager** da classe de testes sempre é atribuído ao atributo de mesmo nome no objeto **service** da classe de testes no **setup()**.

É interessante destacar a peculiaridade e assertividade nos logs do Spock caso esse tipo de verificação falhe. Vamos supor que nosso teste esperasse (erroneamente) por 1 execução do método **persist()**. Nesse caso, o log de erro ficaria como no log de exemplo a seguir.

```
FAILURE! - in unit.br.com.apostilaspoockframework.topico07.ImcAtualServiceTest
deveria          criar          somente          como          novo          por
nome(unit.br.com.apostilaspoockframework.topico07.ImcAtualServiceTest)  Time elapsed: 0.01 sec
<<< FAILURE!
org.spockframework.mock.TooManyInvocationsError: Too many invocations for:

1 * this.entityManager.persist(_) (2 invocations)
Matching invocations (ordered by last occurrence):
1 * <EntityManager>.persist(br.com.apostilaspoockframework.topico07.entity.ImcAtual@2bfc2f8b)
<-- this triggered the error
1 * <EntityManager>.persist(br.com.apostilaspoockframework.topico07.entity.ImcAtual@61853c7e)
```

Note no log que o Spock indica que houve mais invocações do que o esperado (*"Too many invocations"*), quantas invocações ocorreram e ainda os objetos envolvidos em cada execução. Caso a classe **ImcAtual** tivesse sobrescrito o **toString()** padrão do Java, o log poderia ter ficado ainda mais esclarecedor, podendo indicar, por exemplo, atributos dos objetos usados como argumento.

Testes de integração em projetos Spring

Neste tópico, focaremos na criação de testes de integração com Spring. Serão usados 2 estudos de caso: um para o Spring Boot e outro para Spring MVC. Criar testes de integração para ambos é feito de forma muito parecida, pois ambos usam várias bibliotecas Spring em comum. O estudo de caso mais completo será do Boot. A seguir, no estudo do MVC será demonstrado como testar apenas algumas de suas funcionalidades específicas.

Spring TestContext framework e Spring Module do Spock

Para facilitar a criação de testes de integração para os frameworks Spring foi criado um framework de testes chamado **Spring TestContext**. Ele contém uma série de bibliotecas que permite que um teste de integração inicie um contexto completo de uma aplicação Spring, dando a possibilidade de testar todos os componentes de um projeto de forma integrada.

Para fazer uso de todas as funcionalidades do *Spring TestContext* em testes com Spock, existe um módulo dele chamado **Spring Module**. Para usá-lo é preciso adicionar uma dependência ao projeto, que consta no código a seguir:

```
<dependency>
  <groupId>org.spockframework</groupId>
  <artifactId>spock-spring</artifactId>
  <version>1.1-groovy-2.4</version> <!-- ou a versão que preferir -->
  <scope>test</scope>
</dependency>
```

Essa dependência, portanto, é necessária para os estudos de caso abordados neste tópico. O **Spring TestContext framework** é muito grande. O objetivo aqui é abordá-lo de forma bem básica para compreender como ele foi habilitado pelo **Spring Module** do Spock. Caso queira se aprofundar, existe um tutorial completo sobre o *Spring TestContext framework* em sua página no site do Spring em <https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html>.

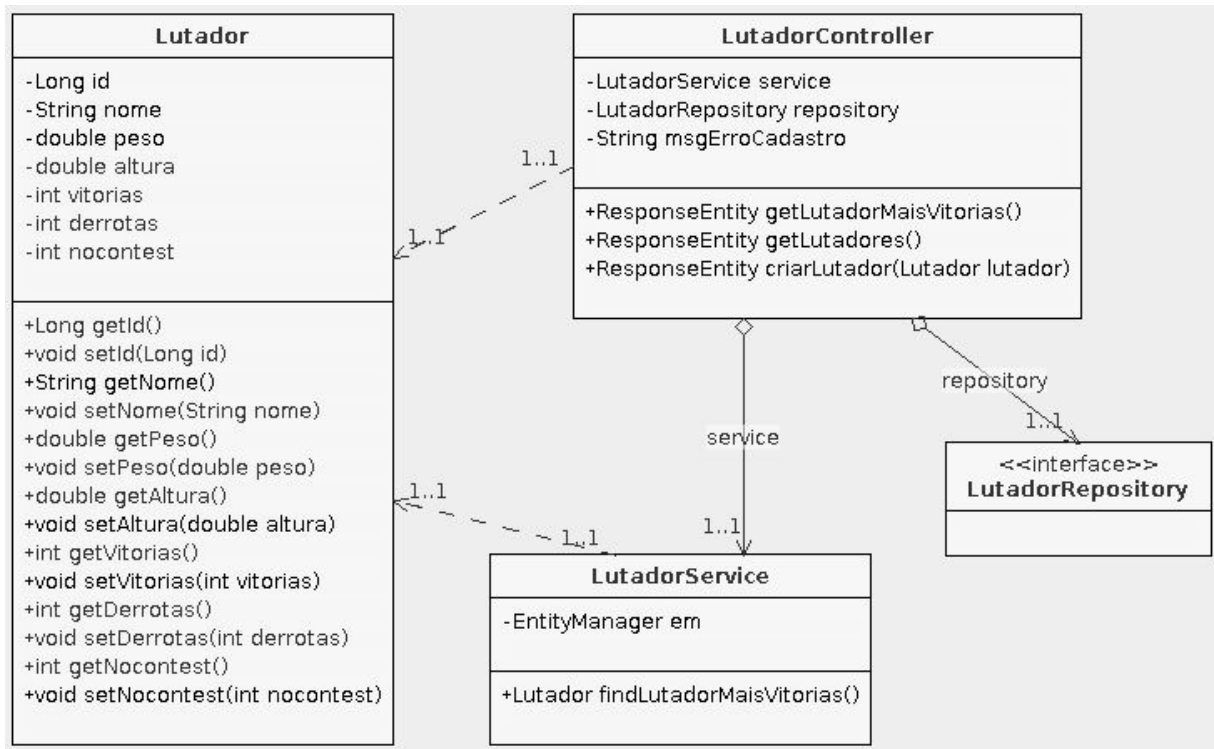
Existem mais módulos Spock para facilitar a criação de testes de integração para outras tecnologias como *Guice*, *Tapestry*, *Unitils* e *Grails*, mas que não fazem parte desta apostila.

Testes de integração em um projeto Spring Boot

O Spring Boot é um framework muito usado para criar REST APIs, possuindo uma série de componentes que abstraem boa parte do trabalho repetitivo e perigoso que envolve criar esse tipo de API.

Resumo do Projeto Spring Boot

O estudo de caso aqui será um pequeno projeto de cadastro de *Lutadores de MMA*. Esse projeto será constituído por um conjunto básico de classes no estilo **Entity** (JPA) + **Repository** (Spring) + **Service** (Spring) + **Spring REST Controller** (Spring), conforme o diagrama de classes da figura a seguir.



Classe Lutador (JPA Entity)

```

// pacote e imports JPA

@Entity
public class Lutador {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @Column(nullable=false)
    private String nome;

    private double peso;

    private double altura;

    private int vitorias;

    private int derrotas;

    private int nocontest;

    // construtores, getters e setters
    
```


A classe **Lutador** é uma entidade JPA simples, com um id (identificador) de valor gerado com autoincremento e alguns atributos simples, sem relacionamento com nenhuma outra entidade. O atributo nome é obrigatório. Os demais também são, porém, como são de tipos numéricos primitivos, serão inicializados com 0 (zero) automaticamente. Aliás, se você não sabe o que é *nocontest*: resumidamente, é quando não há vencedor nem derrotado no combate. Isso pode ocorrer quando a luta termina prematuramente ou quando tem seu resultado alterado posteriormente.

Interface **LutadorRepository** (*Spring Repository*)

```
// pacote e imports Spring

public interface LutadorRepository extends CrudRepository<Lutador, Long>{

}
```

A interface **LutadorRepository** é uma extensão de **CrudRepository**, funcionando como um *Repository* cuja classe de entidade é **Lutador** e a de identificação (chave primária) é Integer.

Classe **LutadorService** (*Spring Service*)

```
// pacote e imports Spring e JPA

@Service
public class LutadorService {

    @Autowired
    private EntityManager em;

    public Lutador findMelhorLutador() {
        return (Lutador) this.em.createQuery(
            "from Lutador order by vitorias desc, nocontest asc, derrotas asc")
            .setMaxResults(1)
            .getSingleResult();
    }
}
```

A classe **LutadorService** está anotada com **@Service**, tornando-a um componente do tipo *Service* do Spring (ou seja, terá uma única instância em toda a aplicação).

Possui apenas um atributo, o *em* do tipo **EntityManager**, para que possamos realizar operações da JPA nos métodos. Ele será injetado em tempo de execução devido à anotação **@Autowired** do Spring.

Seu método **findMelhorLutador()** retorna um **Lutador**. A ideia é retornar o lutador com mais vitórias. Em caso de mais de um lutador com o mesmo número de vitórias, o primeiro critério de desempate é a menor quantidade de *nocontest* e o segundo é a menor quantidade de derrotas.

Classe **LutadorController** (*Spring REST Controller*)

```
// pacote e imports Spring e JPA

@RestController
public class LutadorController {

    @Autowired
    private LutadorService service;

    @Autowired
    private LutadorRepository repository;

    @Value("${msg.erro.cadastro}")
    private String msgErroCadastro;

    // métodos de EndPoints
```

A classe **LutadorController** está anotada com **@RestController**, do Spring. Isso a torna capaz de abrigar Endpoints da REST API da aplicação Spring Boot.

Os atributos **LutadorService** service e **LutadorRepository** repository estão anotados com **@Autowired** para que sejam injetados pelo Spring em tempo de execução, usando a classe e a interface já explicadas neste tópico.

O atributo String **msgErroCadastro** está anotado com a **@Value** do Spring. Isso significa que o Spring vai procurar a chave **msg.erro.cadastro** no arquivo **src/main/resources/application.properties** (falaremos sobre esse arquivo logo mais). Caso encontrada, seu valor será injetado em **msgErroCadastro** em tempo de execução.

Endpoint **GET /melhor-lutador** da **LutadorController**

```
public class LutadorController {

    // atributos

    @GetMapping("/melhor-lutador")
    public ResponseEntity getLutadorMaisVitorias() {
        try {
            return ResponseEntity.ok(this.service.findMelhorLutador());
        } catch (NoResultException e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

O método **getLutadorMaisVitorias()** está anotado com a **@GetMapping** do Spring, o que o configura como método de um Endpoint de URI **/melhor-lutador** cujo verbo HTTP é GET.

Esse método usa a o service, que é uma instância de **LutadorService**, para tentar recuperar o melhor lutador. Caso encontre, devolve uma resposta com status 200 (*Ok*) e com um JSON que representa uma instância de **Lutador**.

Caso não seja encontrado o melhor lutador (caso de base de dados vazia), o método fará o Endpoint retornar um status **404** (*Not Found*) e corpo vazio.

Endpoint **GET** / daLutadorController

```
public class LutadorController {  
  
    // atributos  
    // método getLutadorMaisVitorias()  
  
    @GetMapping  
    public ResponseEntity getLutadores() {  
  
        return this.repository.count() == 0L  
            ? ResponseEntity.noContent().build()  
            : ResponseEntity.ok(this.repository.findAll());  
  
    }  
}
```

O método **getLutadores()** está anotado com a **@GetMapping** do Spring, o que o configura como método de um Endpoint de URI **/** cujo verbo HTTP é GET.

Esse método usa a repository, que é uma instância de **LutadorRepository**, para tentar recuperar todos os lutadores do banco de dados. Caso encontre lutadores, devolve uma resposta com status **200** (*Ok*) e com um JSON que representa uma lista de objetos do tipo **Lutador**.

Caso não seja encontrado o melhor lutador (caso de base de dados vazia), o método fará o Endpoint retornar um status **204** (*No Content*) e corpo vazio.

Endpoint **POST** / {JSON} daLutadorController

```
public class LutadorController {  
  
    // atributos  
    // método getLutadorMaisVitorias()  
    // método getLutadores()  
  
    @PostMapping  
    public ResponseEntity criarLutador(@RequestBody Lutador lutador) {  
        try {  
            this.repository.save(lutador);  
            return ResponseEntity.status(HttpStatus.CREATED).body(lutador);  
        }  
    }  
}
```

```
} catch (Exception e) {  
    return ResponseEntity.badRequest().body(msgErroCadastro);  
}  
}
```

O método **criarLutador()** está anotado com a **@PostMapping** do Spring, o que o configura como método de um Endpoint de URI / cujo verbo HTTP é POST. É necessário enviar um corpo na requisição: um JSON que represente uma instância de **Lutador**.

Esse método usa o repository, que é uma instância de **LutadorRepository**, para tentar inserir o lutador no banco de dados. Caso a operação ocorra sem erros, devolve uma resposta com status **201** (*Created*) e com um JSON que representa a instância do **Lutador** recém-salvo no banco.

Caso ocorra algum erro na operação, o método fará o Endpoint retornar um status **400** (*Bad request*) e o conteúdo de **msgErroCadastro** no corpo da resposta.

Classe LutadoresApp (Spring Application)

```
@SpringBootApplication  
public class LutadoresApp {  
  
    public static void main(String[] args) {  
        SpringApplication.run(LutadoresApp.class);  
    }  
  
}
```

A classe **LutadoresApp** está anotada com **@SpringBootApplication** do Spring, o que a torna a classe de aplicação do projeto. Ou seja, ao iniciá-la invocando seu método **main()**, a REST API da aplicação entra em funcionamento.

Arquivo application.properties do projeto Spring Boot

Além das classes, o projeto também possui um **application.properties** que, por padrão, fica em **/src/main/resources**. É o nome padrão que o arquivo de definição de *properties* (propriedades) de um projeto que usa Spring. Ele será usado para configurar o acesso ao banco de dados e uma mensagem que deverá ser obtida no *REST Controller* também no teste de integração. Seu conteúdo está a seguir.

```
spring.datasource.url=jdbc:h2:~/lutadores  
spring.datasource.username=sa  
spring.jpa.hibernate.ddl-auto=create  
  
msg.erro.cadastro=Erro ao tentar o cadastro de Lutador
```

O banco de dados que foi configurado é o **H2** em memória, mas poderia ser o banco de sua preferência. Já a chave **msg.erro.cadastro** é usada para injetar o texto de seu valor na

LutadorController e também será usada da mesma forma na classe de testes, sobre a qual falaremos a seguir.

Cenários do teste de integração Spring MVC

A classe do teste de integração é a **LutadorControllerTest**, que deve ficar no diretório padrão de testes, o **src/test/groovy**. Nela, vamos testar os seguintes cenários:

- Deveria retornar status 204 e sem corpo quando não existirem lutadores;
- Deveria retornar status 400 e mensagem esperada quando falha em criar Lutador;
- Deveria retornar status 201 com JSON correto quando cria um Lutador;
- Deveria retornar 200 trazer todos os lutadores e no formato correto quando existirem Lutadores.

A seguir, vamos analisar o código dessa classe.

Classe LutadorControllerTest (teste de integração com *Spring Module*)

```
// pacote e imports Spring e JPA
@SpringBootTest(classes = LutadoresApp)
class LutadorControllerTest extends Specification {

    @Autowired
    LutadorController controller

    @Autowired
    LutadorRepository repository

    @Autowired
    LutadorService service

    @Value('${msg.erro.cadastro}')
    private String msgErroCadastro;

    // métodos de teste
}
```

A classe **LutadorControllerTest** está anotada com **@org.springframework.boot.test.context.SpringBootTest** do Spock. Essa anotação indica que a classe deve iniciar um contexto do Spring Boot para a realização de testes de integração entre os componentes da aplicação. Ou seja, **é essa anotação que permite testes de integração Spring Boot** em uma classe de testes Spock.

O atributo classes da **@SpringBootTest** deve ser uma classe marcada como aplicação Spring Boot, ou seja, uma classe anotada com **@SpringBootApplication**. Por isso, usamos a **LutadoresApp** nesse atributo.

Essa classe possui 3 atributos em comum com a **LutadorController**, inclusive com as mesmas anotações: **repository**, **service** e **msgErroCadastro**. Isso demonstra como a

integração será testada só com essas declarações: durante a execução dos testes, o Spock vai usar o Spring para injetar esses atributos tal como seria feito com o projeto Spring Boot.

O atributo controller está anotado com **@Autowired** para que também tenha seu valor injetado pelo Spring durante a execução dos testes. Isso é necessário para que seus atributos gerenciados pelo Spring estejam prontos nos testes para que os métodos funcionem durante a execução da aplicação Spring Boot. Percebeu como o teste promoverá a **integração** entre os componentes?

Cenário *Deveria retornar status 204 e sem corpo quando não existirem lutadores*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos

    def 'deveria retornar status 204 e sem corpo quando não existirem lutadores'() {
        when:
            def resposta = controller.getLutadores()

        then:
            resposta.statusCode == HttpStatus.NO_CONTENT
            !resposta.body
    }

    // demais métodos de teste
}
```

O teste '**deveria retornar status 204 e sem corpo quando não existirem lutadores**()' verifica se a resposta Endpoint do método **getLutadores()** da **LutadorController** terá o status **204** (*No Content*) com corpo vazio. Esse teste é o primeiro a ser executado, logo, a base de dados de testes deve estar vazia.

Cenário *Deveria retornar status 400 e mensagem esperada quando falha em criar Lutador*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos

    def 'deveria retornar status 400 e mensagem correta quando falha em criar Lutador'() {
        when:
            def resposta = controller.criarLutador(new Lutador())

        then:
            resposta.statusCode == HttpStatus.BAD_REQUEST
            resposta.body == msgErroCadastro
    }
}
```

```
// demais métodos de teste
}
```

O teste **'deveria retornar status 400 e mensagem correta quando falha em criar Lutador()'** verifica se a resposta Endpoint do método **criarLutador()** da **LutadorController** terá o status **400** (*Bad Request*) e com corpo contendo o valor do atributo **msgErroCadastro** em caso de lutador inválido (no caso foi usado um lutador sem valor em nenhum de seus atributos). Vale lembrar que esse atributo está anotado da mesma forma que o atributo homônimo na **LutadorController**. Ou seja, por meio de um teste de integração, verificamos se foi usada a injeção planejada na classe que faz o papel de *REST Controller*.

Cenário *Deveria retornar status 201 com JSON correto quando cria um Lutador*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos

    def 'deveria retornar status 201 com JSON correto quando cria um Lutador'() {
        given:
            def lutador = new Lutador(nome: 'Zé Ruela', peso: 80, altura: 1.9)

        when:
            def resposta = controller.criarLutador(lutador)

        then:
            resposta.statusCode == HttpStatus.CREATED
            resposta.body.properties == lutador.properties
    }

    // demais métodos de teste
}
```

O teste **'deveria retornar status 201 com JSON correto quando cria um Lutador()'** verifica se a resposta Endpoint do método **criarLutador()** da **LutadorController** terá o status **201** (*Created*) e com um JSON de resposta cujos atributos são os mesmos de uma instância de Lutador. Isso tudo para o caso de um lutador válido.

Cenário *Deveria retornar 200 trazer todos os lutadores e no formato correto quando existirem Lutadores*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos
```

```

def 'deveria retornar 200 trazer todos os lutadores e no formato correto quando existirem
Lutadores'() {
    given:
    def quantidade = repository.count()

    when:
    def resposta = controller.getLutadores()

    then:
    resposta.statusCode == HttpStatus.OK
    resposta.body.size() == quantidade

    and:
    for (lutador in resposta.body) {
        lutador instanceof Lutador
    }
}

// demais métodos de teste
}

```

O teste **'deveria retornar 200 trazer todos os lutadores e no formato correto quando existirem Lutadores()'** primeiro usa a instância de **LutadorRepository** para consultar junto ao banco de dados quantos lutadores estão na tabela.

A primeira verificação do teste é se o Endpoint do método **getLutadores()** da **LutadorController** terá o status **200 (Ok)**. Em seguida, é verificado se o corpo da resposta é um JSON cuja quantidade de elementos corresponde à quantidade de lutadores no banco de dados.

Ao final, é feita uma verificação que itera no JSON da resposta para verificar, item a item, se todos os elementos são compatíveis com uma instância de Lutador.

Testes de integração em um projeto Spring MVC

O Spring MVC é um framework muito usado para criar projetos Web *full stack*, ou seja, projetos que possuem desde a camada de acesso ao banco de dados até os artefatos de *front-end*. Assim como o Boot, possui uma série de componentes que abstraem boa parte do trabalho repetitivo e perigoso que envolve criar esse tipo de sistema.

Resumo do Projeto Spring MVC

Neste estudo de caso, vamos apenas testar um *Controller* do Spring MVC. Os demais componentes (Service, Repository, Value etc.) poderiam ser testados exatamente como fizemos no projeto com Spring Boot.

A classe que será usada como exemplo é a **LutadorMvcController**, cujo código está a seguir.


```
// pacote e imports do Spring
@Controller
public class LutadorMvcController {

    @GetMapping("/home")
    public ModelAndView home(@RequestParam("usuario") String usuario)    {
        // ações com o argumento 'usuario'
        return new ModelAndView("home");
    }
}
```

A classe **LutadorMvcController** está anotada com **@Controller** do Spring, o que a transforma em um *Controller* do Spring MVC. É um tipo de componente normalmente usado para gerar respostas HTTP com páginas HTML no corpo da resposta, mas que também pode gerar arquivos binários ou até textos simples.

O método **home()** está anotado com a **@GetMapping** do Spring, o que o configura como método de um Endpoint de URI **/home** cujo verbo HTTP é GET. Esse método espera um parâmetro de requisição chamado **usuario** e retorna um **ModelAndView** cuja *view* será a página **src/main/webapp/WEB-INF/home.jsp**.

O único argumento do método está anotado com a **@RequestParam** do Spring. Isso indica que quem quiser chegar a esse método terá que, *obrigatoriamente*, informar o parâmetro de requisição **usuario**.

Arquivo application.properties do projeto Spring MVC

Além da classe *Controller*, o projeto também possui um application.properties que, por padrão, fica em **/src/main/resources**. Ele será usado para configurar o diretório e extensão padrões para as *views* dos *Controllers* do projeto. Seu conteúdo está a seguir.

```
spring.mvc.view.prefix=/WEB-INF/
spring.mvc.view.suffix=.jsp
```

O valor **/WEB-INF/** da chave **spring.mvc.view.prefix** é a partir de **src/main/webapp**. A chave **spring.mvc.view.suffix** indica a extensão padrão para os arquivos de *view* no projeto.

Arquivo home.jsp do projeto Spring MVC

O conteúdo do arquivo **src/main/webapp/WEB-INF/home.jsp** é até irrelevante para nosso estudo de caso. Porém, para deixar você mais seguro, segue um exemplo do que poderia estar em seu conteúdo.

```
<html>
<body>
    <h2>Bem vindo ao sistema de Lutadores!</h2>
    <div>Que belo sistema de lutadores ;)</div>
</body>
</html>
```

Cenários do teste de integração Spring MVC

A classe do teste de integração é a **LutadorMvcControllerTest**, que deve ficar no diretório padrão de testes, o **src/test/groovy**. Nela, vamos testar os seguintes cenários:

- Deveria retornar status 200 e redirecionar para JSP correto;
- Deveria retornar status 400 quando parâmetro o usuario não for enviado.

A seguir, vamos analisar o código dessa classe.

Classe LutadorMvcControllerTest (teste de integração com Spring Module)

```
// pacote e imports Spring
@WebMvcTest(LutadorMvcController)
@ContextConfiguration(classes=LutadoresApp)
class LutadorMvcControllerTest extends Specification {

    @Autowired
    MockMvc mvc

    @Autowired
    LutadorMvcController controller

    @Value('${spring.mvc.view.prefix}')
    String viewPrefix

    // métodos de teste
}
```

A classe **LutadorMvcControllerTest** está anotada com **@org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest** do Spock. Essa anotação indica que a classe deve iniciar um contexto do Spring MVC para a realização de testes de integração entre os componentes da aplicação. Ou seja, **é essa anotação que permite testes de integração Spring MVC** em uma classe de testes Spock.

O atributo value da **@WebMvcTest** deve ser uma classe marcada como *Controller* Spring MVC, ou seja, uma classe anotada com **@Controller**. Por isso usamos a **LutadorMvcController** nesse atributo.

A anotação **org.springframework.test.context.ContextConfiguration** indica onde estão as configurações do Spring no projeto. Como criamos este estudo de caso no mesmo projeto do estudo de caso com Spring Boot, era o caso de usar a **LutadoresApp**. Se estivéssemos lidando com um projeto Spring MVC à parte, deveríamos indicar aqui uma classe com a anotação **@Configuration** do Spring.

O atributo MockMvc mvc é **ator chave nos testes de integração para Spring MVC**. É com ele que testamos o comportamento dos métodos dos *Controllers* de forma integrada. Por isso ele deve estar anotado com **@Autowired**, para que forneça acesso a todos os componentes Spring do projeto durante a execução dos testes.

O atributo **LutadorMvcController** controller é simplesmente o *Controller* que queremos testar. O atributo **String viewPrefix**, anotado com **@Value** do Spring será usado para verificar se a *view* do retorno do método a ser testado no *Controller* está configurada corretamente.

Cenário *Deveria retornar status 200 e redirecionar para JSP correto*

```
// pacote e imports Spring
class LutadorMvcControllerTest extends Specification {

    // atributos

    def 'deveria retornar status 200 e redirecionar p/ JSP correto'() {
        given:
            def destino = "home"

        when:
            def resposta = mvc.perform(MockMvcRequestBuilders.get("/home").param("usuario", "Lady Gaga"))

        then:
            resposta
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.forwardedUrl("${viewPrefix}${destino}.jsp"))
    }
}
```

Aqui, primeiro definimos qual o destino esperado. Em seguida, solicitamos a realização da chamada ao método **home()** de forma indireta, indicando o verbo HTTP, a URI e o parâmetro de requisição. Na etapa de verificação, primeiro verificamos se o status da resposta é **200** (*Ok*). Em seguida, verificamos se a *view* corresponde a **/WEB-INF/+home+.jsp**.

Cenário *Deveria retornar status 400 quando parâmetro o usuario não for enviado*

```
// pacote e imports Spring
class LutadorMvcControllerTest extends Specification {

    // atributos

    def 'deveria retornar status 400 quando parâmetro o "usuario" não for enviado'() {
        when:
            def resposta = mvc.perform(MockMvcRequestBuilders.get("/home"))

        then:
            resposta.andExpect(MockMvcResultMatchers.status().is(400))
    }
}
```

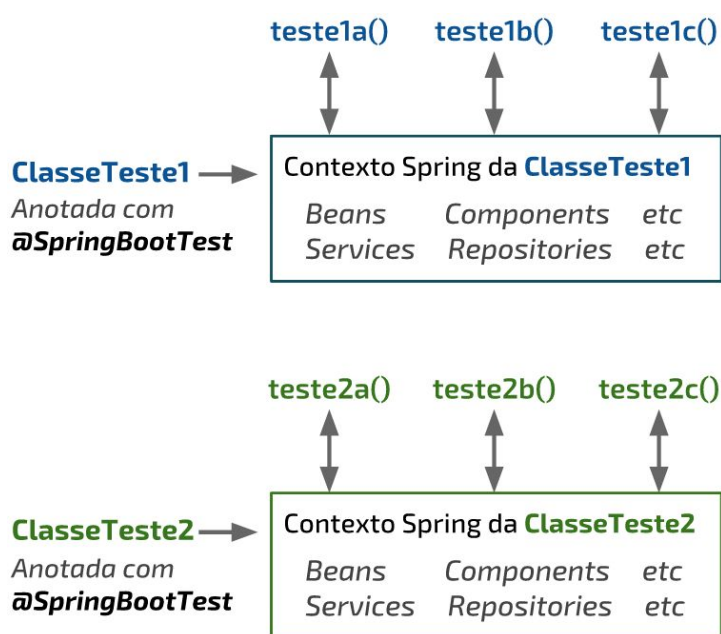
Neste teste, solicitamos a realização da chamada ao método **home()** de forma indireta, indicando o verbo HTTP, a URI e a ausência de parâmetros de requisição. Em seguida, verificamos se o status da resposta é **400** (*Bad Request*). Isso porque o parâmetro de requisição está como obrigatório em **LutadorMvcController**.

Ciclo de vida do contexto Spring nos testes de integração

Quando criamos uma classe de testes de integração usando *Spring TestContext framework* e *Spring Module do Spock*, é importante ter ciência de que **cada classe de testes possui seu contexto Spring próprio**. Isso significa que:

- Os **componentes são compartilhados entre os métodos de teste de uma mesma classe**, de acordo com o escopo de cada componente. Por exemplo: *Services* são *Singletons*, portanto terão a mesma instância compartilhada entre todos os métodos de testes de uma mesma classe. No caso de componentes de escopo *request*, terão uma instância diferente por método de uma mesma classe;
- Os **componentes não são compartilhados entre classes** de testes de integração Spring. Inclusive é possível ver a mensagem de inicialização do Spring Boot no log do Maven sempre que uma classe de testes de integração é iniciada.

Esse comportamento está ilustrado na figura a seguir.



Profiles e properties do projeto Spring nos testes de integração

Algumas das funcionalidades que mais dão flexibilidade de configuração em projetos com Spring são as *profiles* (perfis) e *properties* (propriedades), que podem ser configuradas

de forma muito simples. A seguir, veremos como usar essas funcionalidades em testes de integração com Spock.

Definindo um arquivo de *properties* para os testes

É possível definir as *properties* (propriedades ou configurações) de um projeto que usa Spring escrevendo-as em um arquivo chamado **application.properties** (formato *Properties*, ou seja, *chave x valor*) ou **application.yml** (formato *YAML*, ou seja, combinação de listas, mapas e valores simples). Nesse arquivo normalmente são definidas configurações como:

- Dados de acesso ao banco de dados;
- Dados de acesso a outros sistemas (por exemplo: servidor de e-mail);
- Textos de mensagens de validação.

Por padrão, o Spring procura um desses arquivos no diretório **/src/main/resources**. Porém, imagine que você precisa definir configurações apenas para os testes, tais como o uso de um banco de dados em memória, por exemplo. Para situações assim, é possível definir um arquivo de *properties* exclusivo para os testes de integração. Para isso, basta criar um **application.properties** (ou .yml) no diretório **/src/test/resources**, que ele será automaticamente usado pelos testes de integração.

Atenção: caso exista um arquivo de configuração em **src/test**, os arquivos em **src/main** serão ignorados durante os testes de integração.

Escolhendo o *profile* para os testes

Um dos recursos mais interessantes do Spring é a facilidade em ter diferentes *profiles* (perfis) em um mesmo projeto. Isso permite, por exemplo, que a aplicação acesse diferentes bancos de dados, de acordo com o *profile* selecionado. Para que um projeto Spring tenha vários perfis configurados, basta que possua vários arquivos de *properties* com nome terminando em **-nome_do_perfil**. Por exemplo, nosso projeto teria vários *profiles* caso ele tivesse os seguintes arquivos em **src/main/resources** (ou em **src/test/resources**):

- application.properties (profile padrão)
- application**-test**.properties (profile test)
- application**-dev**.properties (profile dev)
- application**-homol**.properties (profile homol)
- application**-prod**.properties (profile prod)

Quando criamos uma classe de testes de integração usando *Spring TestContext framework* e *Spring Module do Spock*, é possível indicar o *profile* que queremos usar na execução de uma classe de testes. Para isso, basta usar a anotação **@org.springframework.test.context.ActiveProfiles** sobre a definição da classe de testes.

Poderíamos reconfigurar a **LutadorControllerTest** deste tópico para tentar usar um *profile* chamado **test** durante a execução dos testes de integração. Então ela ficaria como no código a seguir.

```
import org.springframework.test.context.ActiveProfiles
// demais imports

@ActiveProfiles("test")
// demais anotações da classe
class LutadorControllerTest extends Specification {
    ...
}
```

Com a inclusão da anotação **@ActiveProfiles("test")** sobre a classe, todos os testes nela serão executados considerando as configurações no arquivo **application-test.properties** (ou **application-test.yml**) em **src/test/resources** ou em **src/main/resources**.

Ordem de busca de *properties* conforme *profile*

Imagine que você possui várias *properties* e *profiles* configurados. Ter configurações espalhadas em tantos arquivos pode causar confusões durante os testes.

Para que você tenha certeza de qual arquivo de configuração será usado na busca de suas configurações, basta saber qual a sequência de arquivos nos quais uma propriedade é buscada pelo Spring. Para melhor demonstrar isso, imagine que você solicitou a busca pela *property* **msg.erro.sistema** e sua classe de testes está configurada para o *profile* **test**. O Spring tentaria localizar essa *property* nas seguintes sequências durante a execução do teste de integração:

Cenário 1: existe pelo menos 1 arquivo de *properties* em **src/test/resources**

1. Arquivo **application-test.properties** (ou .yml) em **src/test/resources**;
2. Arquivo **application.properties** (ou .yml) em **src/test/resources**.

Caso a *property* **msg.erro.sistema** não seja encontrada em nenhum desses 2 arquivos, uma exceção específica do Spring será lançada.

Cenário 2: não existe nenhum arquivo de *properties* em **src/test/resources**

1. Arquivo **application-test.properties** (ou .yml) em **src/main/resources**;
2. Arquivo **application.properties** (ou .yml) em **src/main/resources**.

Caso a *property* **msg.erro.sistema** não seja encontrada em nenhum desses 2 arquivos, uma exceção específica do Spring será lançada.

Apêndice A - Guia de Groovy para desenvolvedores Java

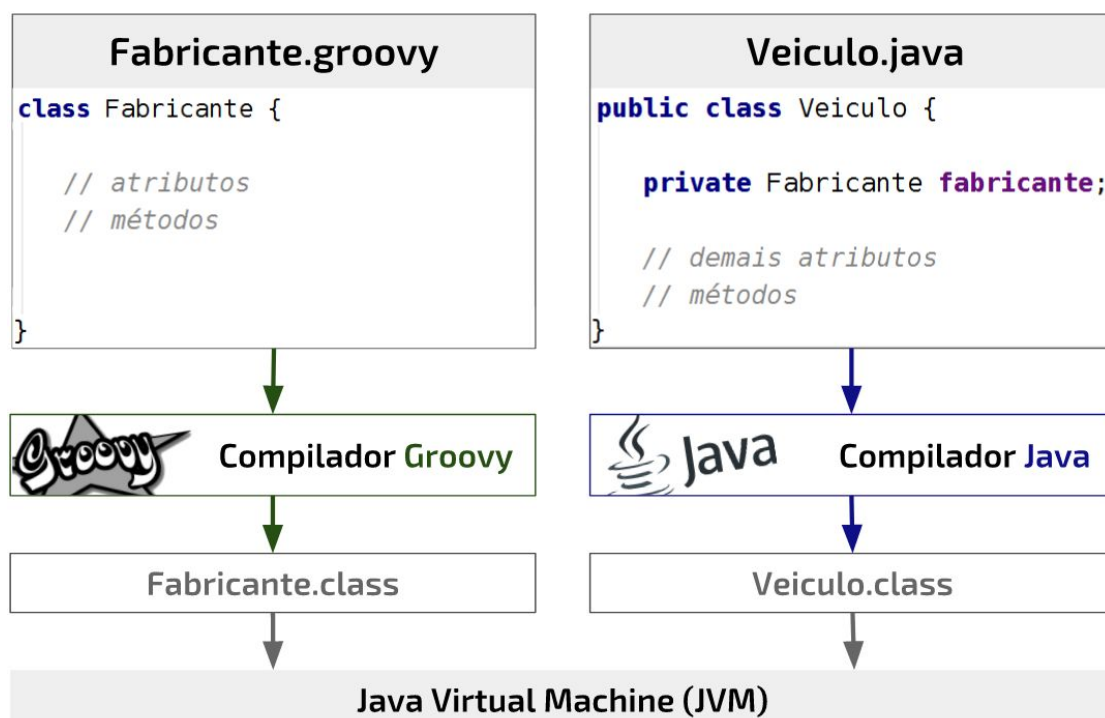
Groovy é a linguagem oficial do Spock framework, por isso é fundamental ter uma proficiência mínima com ela. Este apêndice é para você que possui pouca ou nenhuma experiência com a essa linguagem. É um guia focado em desenvolvedores Java, para que façam o "de x para" (em Java é daquele jeito e em Groovy é desse) e tenham maior facilidade na escrita dos testes com Spock.

Como funciona o Groovy

A linguagem Groovy foi criada em 2003 por James Strachan e hoje possui seu código aberto sendo mantida pela *Apache Software Foundation* desde 2015. É uma das linguagens cujo compilador gera arquivos executáveis pela *Máquina Virtual Java*, a **JVM**. Ou seja, arquivos escritos nessa linguagem, quando compilados, geram arquivos **.class** executáveis por qualquer JVM de versão de Java compatível. Na prática, isso significa que:

- Se uma IDE possui suporte a Java e Groovy, as classes escritas nessas duas linguagens conseguem trabalhar entre si de forma transparente;
- Bibliotecas escritas em Groovy podem ser usadas por projetos Java e vice-versa.

Esse funcionamento é representado na figura a seguir.



É por isso que o Spock, que possui como linguagem de programação o Groovy, pode ser usado para criar testes para projetos em Java, Scala, Kotlin, Jython, JRuby ou qualquer outra linguagem que compile para a JVM. Consequentemente, também é possível criar testes para projetos que usam qualquer framework Java como Spring MVC/Boot, Play, Struts, VRaptor etc. e até mesmo para projetos Android.

Uma característica que reduz a "curva de aprendizado" dessa linguagem para desenvolvedores Java é a proximidade entre as sintaxes e a possibilidade de escrever um código Groovy exatamente como escreveria em Java, que vai compilar e funcionar perfeitamente na imensa maioria dos casos. Ou seja, na dúvida em determinado trecho de código Groovy, faça exatamente como faria em Java.

Características e recursos do Groovy

A seguir, alguns dos recursos e características do Groovy que podem ajudar muito na criação de testes com Spock.

Diferenças na sintaxe

A sintaxe do Groovy é muito parecida com do Java. Porém, algumas diferenças na sintaxe do Groovy podem melhorar a produtividade no desenvolvimento. A seguir, várias dessas diferenças serão apresentadas.

Ponto e vírgula são opcionais no final da linha de código

Caso o desenvolvedor queira abrir mão de usar ponto e vírgula (;) ao final de cada linha, o compilador Groovy vai considerar a quebra de linha como o fim da linha de código. Porém, se ela for utilizada, será o final explícito da linha de código, como é em Java. É possível usar linhas com e sem ponto e vírgula no final em um mesmo arquivo Groovy sem problema algum.

System.out opcional

Caso seja necessário imprimir algo na saída padrão, é possível usar os **System.out.print()** e **System.out.println()** como em Java. Porém, como o **System.out** é opcional em Groovy, podem ser substituídos apenas por **print()** e **println()**, respectivamente.

Parênteses são opcionais na invocação de métodos

Em Groovy, não é obrigatório o uso de parênteses na invocação de métodos, sejam eles com ou sem parâmetros. Como exemplo, temos o código a seguir, que imprime textos na saída padrão usando o `println` com e sem parênteses.

```
println("Não sou dono do mundo")
println "Mas sou filho do dono"
```

Outro exemplo: o código a seguir exemplifica a invocação de métodos da classe **java.lang.Math** sem parênteses, primeiro com um depois com dois parâmetros.

```
double raiz144 = Math.sqrt 144 // retornará a raiz de 144: 12
double doisAoCubo = Math.pow 2,3 // retornará 2 ao cubo: 8
```


Ratificando: o não uso de parênteses é **opcional**. Caso você ache esse tipo de sintaxe confusa, pode usar os parênteses tal como usa na invocação de métodos em Java.

Por padrão, as classes são públicas

Quantas vezes você criou uma classe que não deveria ser public? Em Groovy, as classes são públicas por padrão. Ou seja, a classe **Futebolista** a seguir será compilada como pública pela JVM:

```
// Futebolista.groovy
class Futebolista { // em tempo de execução: "public class Futebolista"
    // atributos, métodos etc.
}
```

Por padrão, os atributos são privados

Quantas vezes você criou um atributo de instância que não deveria ser **private**? Em Groovy, os atributos são **privados por padrão**. Ou seja, na classe Groovy **Futebolista** a seguir, os atributos **nome** e **valorPasse** serão considerados privados pela JVM:

```
// Futebolista.groovy
class Futebolista {
    String nome // em tempo de execução: "private String nome"
    double valorPasse // em tempo de execução: "private double valorPasse"
}
```

Por padrão, os métodos são públicos

Você criou muito mais métodos **public**, **private** ou **default** em suas classes? Em Groovy, os métodos são **públicos por padrão**. Ou seja, a classe Groovy **Futebolista** a seguir teria um método considerado público (**treinar()**) e outro privado (**fugirDaConcentracao()**) pela JVM:

```
// Futebolista.groovy
class Futebolista {
    String nome
    double valorPasse

    void treinar() { // em tempo de execução: "public void treinar()"
        // treinando...
    }

    private void fugirDaConcentracao() {
        // fazendo a festa fora da concentração...
    }
}
```

É possível atribuir valores de atributos na criação de objetos

Em Java, quando queremos criar uma instância que já tenha determinados valores em atributos, é necessário criar construtores para isso. Em Groovy, toda classe possui um construtor que aceita um **Map** como argumento (porém, com colchetes opcionais). Assim, é possível indicar os argumentos que quiser e seus valores. Por exemplo, seria possível instanciar um objeto do tipo **Futebolista** como no código a seguir:

```
def jogadorA = new Futebolista(nome: 'Ze Boleiro', valorPasse: 100)
def jogadorB = new Futebolista(valorPasse: 100, nome: 'Ze Boleiro')
def jogadorC = new Futebolista(nome: 'Ze Boleiro')
def jogadorD = new Futebolista(valorPasse: 100)

def jogadorE = new Futebolista([nome: 'Ze Boleiro', valorPasse: 100])

Map mapaLoko = // Map criado em algum lugar...
def jogadorF = new Futebolista(mapaLoko)
```

Métodos podem ter uma frase entre aspas como nome

Em classes de testes automatizados, normalmente um método testa um cenário. Imagine um cenário chamado "**o valor do passe do jogador não pode ser menor ou igual a 0(zero)**". Se fôssemos nomear um método da maneira convencional, ou seja, uma palavra só em **camelCase**, seu nome seria algo como...]

```
def valorPasseJogaodorNaoAceitaZeroOuMenos() { ... }
```

Não acha que é um nome longo e difícil de ler? Uma alternativa seria reduzir o nome e pôr a descrição do cenário no **JavaDoc** do método. Porém, caso o teste falhe, é seu nome que é impresso no log da execução dos testes.

Usando Groovy, o nome do método poderia ser...

```
def 'valor do passe do jogador não pode ser menor ou igual a 0'() { ... }
```

Usando esse recurso, o cenário do teste fica literalmente descrito no nome do método. Outra vantagem é que se o teste falhar, o log de execução conterà os cenários e não nomes de métodos longos e/ou difíceis de ler.

Podem ser usadas aspas simples ou duplas e até mesmo caracteres especiais (letras acentuadas, cê-cedilha etc.) podem ser usados.

A tipagem pode ser estática ou dinâmica

Para reduzir o tempo de programação, em Groovy é possível criar objetos e métodos com o operador **def**, usando o recurso da **tipagem dinâmica**. Em caso de métodos, é possível omitir o tipo dos parâmetros. Vejamos exemplos a seguir.

No exemplo a seguir, foi usada a tipagem dinâmica para ambos os métodos. Em tempo de execução, a JVM vai considerar que eles retornam **Object**.

```
def metodoX(p1, p2) { // em tempo de execução: "public Object metodoX(Object p1, Object p2)"
}

def metodoY(def n1, def n2) { // em tempo de execução: "public Object metodoY(Object n1, Object n2)"
}
```

Se o método for **static**, até mesmo o **def** é opcional, como nos exemplos a seguir, cujos métodos serão considerados *públicos, estáticos* e com retorno **Object** em tempo de execução:

```
static metodoX(p1, p2) { // em tempo de execução: "public static Object metodoX(Object p1, Object p2)"
    // return ...;
}

static def metodoY(def n1, def n2) { // em tempo de execução: "public static Object metodoY(Object n1, Object n2)"
    // return ...;
}
```

Se o desenvolvedor preferir, pode deixar a declaração do método ou dos argumentos estática, como nos exemplos a seguir:

```
Double metodoX(n1, n2) { // em tempo de execução: "public Double metodoY(Object n1, Object n2)"
}

Double metodoY(Double n1, n2) { // em tempo de execução: "public Double metodoY(Double n1, Object n2)"
}

void metodoZ(Double n1, Double n2) { // em tempo de execução: "public void metodoZ(Double n1, Double n2)"
}
```

Objetos também podem ser definidos com esse recurso, como no exemplo a seguir onde primeiro é criado um **BigDecimal** declarado com tipagem dinâmica e, depois um outro com tipagem estática.

```
def numero0 = new BigDecimal(0) // em tempo de execução: BigDecimal numero0 = ...
BigDecimal numero1 = new BigDecimal(1)
```

Uma boa notícia para o desenvolvedor é que as principais IDEs Java do mercado - Eclipse, IntelliJ e NetBeans - sugerem os métodos e atributos do tipo usado na instanciação.

Ou seja, no último código, após **numero0**. essas IDEs iriam sugerir métodos e atributos da classe **BigDecimal**.

Operador == faz o papel do .equals()

Em Java, o operador **==** verifica se os membros comparados são, literalmente, o mesmo objeto, a mesma instância. Para saber se possuem *conteúdo* igual, deve-se usar o método **.equals()**. Inclusive é comum que iniciantes em Java tentem comparar duas Strings de mesmo conteúdo com **==** e acabam recebendo um false como resultado.

Acreditando que a maioria dos desenvolvedores faz muito mais comparação de conteúdo do que de referência de instância em seus projetos, os arquitetos da linguagem Groovy definiram que, nela, o **==** correspondesse ao **.equals()** do Java. Vejamos o exemplo a seguir, onde **texto1** e **texto2** têm seus conteúdos comparados:

```
String texto1 = // valor recebido em tempo de execução
String texto2 = // valor recebido em tempo de execução
def teste = texto1 == texto2
// se o conteúdo de "texto1" e "texto2" forem iguais, "teste" será "true"
```

Caso exista a necessidade de saber se dois objetos são a mesma instância, deve-se usar o método **.is()**, conforme o exemplo a seguir:

```
String texto1 = // valor recebido em tempo de execução
String texto2 = // valor recebido em tempo de execução
String texto3 = texto1
def teste1e2 = texto1.is(texto2) // "false"
def teste1e3 = texto1.is(texto3) // "true"
```

Concatenação de valores String mais dinâmica e simples

Em Groovy, um objeto **String** pode ser instanciado com aspas simples, como no exemplo a seguir:

```
String poema = 'Nas curvas do teu corpo capotei meu coração'
```

Porém, ao se fazer isso, perde-se a possibilidade de usar a capacidade de **concatenações dinâmicas do Groovy**, que só é possível quando se usa aspas duplas. Basta usar o operador **\$** e o nome do objeto a ser concatenado no texto. Vejamos o exemplo a seguir, onde **time1**, **time2** e dias serão concatenados no conteúdo de **anuncio**:

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def dias = 7
def anuncio = "O jogo será entre $time1 e $time2. Faltam $dias dias, não percam!"
// "O jogo será entre Tabajara e Sayajins. Faltam 7 dias, não percam!"
```

A montagem de um texto pode incluir resultados da invocação de métodos e/ou de operações matemáticas. Assim como o recurso anterior, é possível **executar métodos e operações matemáticas e concatenar seus resultados** em **String** Groovy. Nesse caso, a expressão deve estar dentro de **`$()`**.

No exemplo a seguir, o resultado de **`toUpperCase()`** em **`time1`** e de **`substring(4)`** sobre **`time2`**, bem como o resultado de valor dividido por 2 serão concatenados no conteúdo de **`anuncio`**.

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def valor = 20
def anuncio = "O jogo será entre ${time1.toUpperCase()} e ${time2.substring(4)}. Mulheres
pagam ${valor/2}, não percam!"
// "O jogo será entre TABAJARA e jins. Mulheres pagam 10, não percam!"
```

Caso exista a necessidade de montar um texto que deva estar tão associado a um objeto ao ponto que seu conteúdo acompanhe alterações feitas nesse objeto, pode-se lançar mão de outro tipo de concatenação mais dinâmica ainda, **verificando o valor atual de um objeto sempre que a String for invocada**. Para isso, o objeto deve estar dentro de **`${->}`**.

No código do próximo exemplo, associamos **`time1`** e **`time2`** a **`anuncio`** de tal forma que, sempre que **`anuncio`** for usada, seu conteúdo pode variar, de acordo com o conteúdo de **`time1`** e **`time2`**.

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def frase = "O jogo será entre ${-> time1} e ${-> time2}"
// nesse momento, "anuncio" é
// "O jogo será entre Tabajara e Sayajins"

time1 = 'Remo'
time2 = 'Paysandu'
// a partir desse momento, "anuncio" passa a ser
// "O jogo será entre Remo e Paysandu"
```

String podem ter múltiplas linhas sem concatenações

Quem já precisou escrever uma instrução SQL ou criar um *Mock* de um JSON em Java já deve ter feito todo tipo de malabarismo para que uma **String** de várias linhas não fique tão ilegível no código. Em Groovy, existe o recurso de múltiplas linhas. Basta usar **três aspas simples ou duplas** para instanciar uma String.

No exemplo de String de múltiplas linhas a seguir, é criada uma instrução SQL com 3 linhas.

```
def consultaTimesSemTitulo = '''
select * from tb_time
where num_titulos = 0
order by dt_cricao desc
'''
```

Caso você precise criar uma **String** de múltiplas linhas com concatenação de métodos e/ou operações matemáticas, basta usar **três aspas duplas** nos limites da **String**. O exemplo a seguir monta uma instrução SQL de múltiplas linhas concatenando seu conteúdo com **uf**.

```
def uf = 'AM'
def consultaTimesAmazonas = """
select * from tb_time
where estado = ${uf}
"""
```

Interoperabilidade com classes Java

A interoperabilidade de Groovy com classes Java é total, contanto que as versões de Java usadas na compilação de ambos sejam compatíveis. Trata-se da mesma restrição entre classes de arquivos escritos em Java.

Para exemplificar essa interoperabilidade, consideremos a classe Java **aploko.pjava.Futebolista**, que contém três atributos simples, um construtor que altera todos eles e seus *getters* e *setters*:

```
package aploko.pjava

// imports

public class Futebolista {
    private String nome;
    private double habilidade;
    private double velocidade;

    public Futebolista(String nome, double habilidade, double velocidade) {
        // atribuições simples
    }

    // getters e setters públicos de todos os atributos
}
```

A classe Groovy **FutebolistasLokos** a seguir usa a classe Java **aploko.pjava.Futebolista** de forma transparente: uma instância é criada usando seu construtor e um de seus métodos, o **setNome()**, é invocado.

```
import apploko.pjava.Futebolista
// demais imports

class FutebolistasLokos {
    Futebolista createHabilidoso() {
        def habilidoso = new Futebolista("Romário", 95, 90)
        habilidoso.setNome("Romário 11")
        return habilidoso
    }
}
```

Se o projeto utilizar Maven ou Gradle, qualquer classe das dependências também será acessível de forma transparente pelas classes Groovy.

NullPointerException é muito fácil de ser evitado

Se você já trabalha com Java há algum tempo já deve ter visto o **java.lang.NullPointerException** (vulgo **NPE**) várias vezes quando executava seu projeto. Isso ocorre quando tentamos invocar um método de um objeto nulo (**null**), o que acontece com uma certa frequência porque é muito trabalhoso (e fácil de esquecer) verificar a cada **getXXX()** se não temos um **null**. Em Groovy esse problema é simples de resolver, pois existe o chamado **operador de navegação segura** (*Safe navigation operator*), que previne o NPE apenas com o uso de **?** antes do **..**. Vide o próximo código.

```
def jogador = // instanciado em tempo de execução a partir do banco de dados, por exemplo
def pais = jogador?.getTime()?.getPais()?.getNome();
```

No último código, caso **jogador** ou **getTime()** ou **getPais()** seja **null**, o objeto **pais** simplesmente receberá **null**, **sem NPE**. Caso o operador **?** não seja usado, o risco de NPE é o mesmo de uma classe Java.

Manipulação de coleções é muito simples

A manipulação de coleções em Java é considerada *verbosa* demais por desenvolvedores acostumados com linguagens como PHP, Python, Ruby e JavaScript. Em Groovy, a manipulação de coleções é bem simples, se comparada com Java, conforme apresentado a seguir.

Criando uma coleção com valores

Em Java, para criar uma coleção com valores deve-se recorrer a classes utilitárias, sejam elas do Java ou de bibliotecas de terceiros. Em Groovy, esse tipo de operação é muito simples, como nos exemplos a seguir. No próximo código, há um exemplo da criação de uma **List** em Groovy. Ele cria uma instância de **ArrayList** com 4 (quatro) elementos.

```
def posicoes = ['goleiro', 'zagueiro', 'meia', 'atacante']
```

O interessante no código anterior é que as IDEs Eclipse e IntelliJ **inferem** o tipo de objeto da coleção quando se cria uma usando todos os elementos do mesmo tipo. Assim, quando tentar a recuperação de elementos dessa coleção, a IDE exibirá que o retorno seria do tipo **String**. Caso não consigam fazer a inferência, a coleção será tratada como de **Object**.

No próximo código, há um exemplo da criação de um **Map** em Groovy. Ele cria uma instância de **HashMap** com 3 (três) elementos, com as chaves **derrotas**, **empates** e **vitorias** cujos valores são **0**, **1** e **3**, respectivamente.

```
def campanha = ['derrotas': 0, 'empates': 1, 'vitorias': 3]
```

Um detalhe bem útil em Groovy é a possibilidade de ser opcional o uso de aspas quando as chaves do mapa são String. Portanto, no código todas as três chaves poderiam estar sem aspas, como no código a seguir.

```
def campanha = [derrotas: 0, empates: 1, vitorias: 3]
```

Criando coleções vazias

Em Java, para criar uma coleção vazia, devemos usar construtores como faríamos para qual outro tipo de classe. Em Groovy, é possível usar os mesmos construtores, porém as principais coleções podem ser criadas com o uso de operadores, como nos exemplos a seguir.

No código Groovy adiante há um exemplo de criação de **List vazia**, implementada como **ArrayList**.

```
def posicoes = []
```

No código a seguir há um exemplo de criação de **Map vazia** implementada como **HashMap** em Groovy.

```
def pontuacoes = [:]
```

Adicionando um elemento a uma coleção

Em Java, para adicionar elementos a uma coleção devemos recorrer a métodos, como **add()** ou **put()**. Em Groovy é possível usar exatamente os mesmos métodos, porém também é possível usar **operadores**, como nos exemplos a seguir.

No próximo código, há um exemplo da criação de uma **List** vazia e posterior inclusão de elementos nela.


```
def posicoes = []
posicoes += 'goleiro'
posicoes += 'zagueiro'
```

Há três maneiras de incluir um elemento num Map em Groovy, todas exemplificadas no código a seguir.

```
def campanha = [:]
campanha.derrotas = 0 // chave: 'derrota' / valor: 0
campanha['empates'] = 1 // chave: 'empate' / valor: 1
campanha += [vitorias: 3] // chave: 'vitoria' / valor: 3
```

Recuperação de itens de uma coleção

Em Java, para recuperar um item de uma coleção devemos recorrer a métodos, como o **get()**. Em Groovy é possível usar exatamente os mesmos métodos, porém também é possível usar **operadores**, como nos exemplos a seguir.

No próximo código, há exemplos da recuperação de um elemento de uma List.

```
def posicoes = ['goleiro', 'zagueiro', 'meia', 'atacante']
println(posicoes[0]) // imprimiria 'goleiro'
println(posicoes[2]) // imprimiria 'meia'
```

Há duas maneiras de recuperar um elemento de um Map em Groovy, exemplificadas no código a seguir.

```
def campanha = [derrota: 0, empate: 1, vitoria: 3]
campanha.derrota // 0
campanha['vitoria'] // 3
```

Recuperando o último elemento de uma lista ou vetor

Imagine que você precise do último elemento em uma lista mas não sabe quantos elementos ela conterá no momento em que precisar dessa informação. Em Java é preciso verificar o tamanho da lista e usar seu valor menos 1 para recuperar o último elemento de uma lista. Em Groovy, há um método chamado **last()** que abstrai isso e recupera o último elemento de uma lista, como no exemplo a seguir.

```
def listaDinamica = // recuperada de forma dinâmica (ex: de um banco de dados)
def ultimo = listaDinamica.last()
```

O método **last()** **não existe em Mapas**, estando disponível apenas para listas e vetores.

Técnicas de iteração simples e embarcadas para números, coleções e Strings

É uma tarefa muito comum realizar iterações (repetições) para resolver problemas computacionais. Essas iterações são comumente baseadas em valores numéricos, nos elementos de uma coleção ou nas linhas de um texto. A seguir, veremos como essas iterações mais comuns podem ser feitas em Groovy.

Iteração a partir de números

Na necessidade da repetição de um trecho de código determinada por um número **N** em Java, é necessário criar estruturas de repetição como **for**, **do-while** ou **while**. Em Groovy, os números inteiros possuem um recurso embarcado que atende facilmente essa necessidade. É a closure **times{}**. No exemplo a seguir, está programada uma repetição de um trecho de código por 11 (onze) vezes.

```
11.times{
    // o código aqui repetirá 11 vezes
}
```

Em vez do número 11, poderíamos ter aplicado o mesmo recurso sobre uma variável do tipo Integer. Caso seja necessário saber em que passo da iteração estamos, basta usar a variável **it**, que é **injetada automaticamente** na closure **times{}**. Ela vai **de 0 a N-1**, onde **N** é o número de iterações. O código do exemplo a seguir demonstra como poderíamos lançar mão desse recurso.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.size().times{
    println(jogadores[it].getNome())
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome da variável do passo da iteração. No exemplo a seguir, usamos o nome **j**.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.size().times{ j ->
    println(jogadores[j].getNome())
}
```

Iteração a partir de List e Set

Quando há repetição de um trecho de código para cada elemento de uma coleção em Java, é necessário criar estruturas de repetição como **for**, **do-while**, **while** ou usar os streams ou método **forEach()** do **Java 8**. Em Groovy, as coleções possuem um recurso embarcado que atende facilmente essa necessidade. São as closures **each{}** e **eachWithIndex{}**, apresentadas e exemplificadas a seguir. No exemplo a seguir, está

programada uma repetição para os itens de uma **List**, mas que funcionaria da mesma forma para **Set**. O **it** dentro da closure é o nome que cada elemento da lista recebe na iteração.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.each{
    println("Nome do Jogador: ${it.getNome()}")
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome do elemento na iteração. No exemplo a seguir, usamos o nome **jogador**.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.each { jogador ->
    println("Nome do Jogador: ${jogador.getNome()}")
}
```

Caso seja necessário saber em que passo da iteração estamos, podemos usar a closure **eachWithIndex()**. Nesse caso, **é obrigatório** indicar os nomes do elemento e da variável que conterá o índice (passo) da iteração (que começa em 0), como demonstrado no código do exemplo a seguir, em que chamamos o elemento de jogador e o índice de **i**.

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.eachWithIndex{ jogador, i ->
    println("Nome do ${i+1} Jogador: ${jogador.getNome()}")
}
```

Iteração a partir de Map

Quando há repetição de um trecho de código para cada elemento de um mapa (**Map**) em Java, é necessário criar estruturas de repetição como **for**, **do-while**, **while** ou usar os streams ou método **forEach()** do Java 8. Em Groovy, as coleções possuem um recurso embarcado que atende facilmente essa necessidade. É a closure **each()**, apresentada e exemplificada a seguir.

No exemplo a seguir, está programada uma repetição para os itens de um **Map**. O **it** dentro da closure é o nome que cada elemento do **Map** recebe na iteração. O atributo **key**, é o **chave** e o value, o **valor** do elemento.

```
def campanha = [derrotas: 0, empates: 2, vitórias: 5]
println("Campanha:")
campanha.each{
    println("${it.key}: ${it.value}") // ex de saída: "vitórias: 5"
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome do elemento na iteração. No exemplo a seguir, usamos o nome **campanha**.

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{ campanha ->
    println("${campanha.key}: ${campanha.value}") // ex de saída: "vitorias: 5"
}
```

É possível ainda definir diretamente um nome para a chave e outro para o valor, como no próximo código, em que os chamamos de **resultado** e **quantidade**, respectivamente.

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{ resultado, quantidade ->
    println("${resultado}: ${quantidade}") // ex de saída: "vitorias: 5"
}
```

Verificação de "verdadeiro/falso" expandida, porém facilitada

A verificação de **verdadeiro/falso** (*true/false*), provavelmente, a mais comum em testes automatizados. É possível fazer esse tipo de verificação de forma simplificada em Groovy conforme as situações a seguir.

null é false e não nulo é true

Qualquer objeto que seja **null**, quando testado em um **assert** ou **if** ou **operador ternário** ou atribuído a uma variável **boolean**, será interpretado como **false**. Vide o código Groovy de exemplo a seguir, cujo **assert** **falharia**.

```
def jogador = null
assert jogador
```

Nas mesmas situações recém citadas, qualquer objeto **não nulo**, **a princípio** é **true**. No código do exemplo a seguir, o **assert** **passaria**.

```
def jogador = new Futebolista()
assert jogador
```

Observe: foi dito que **a princípio**, o resultado é **true**. Existem casos em que o objeto não é nulo, mas sua verificação booleana pode ser **false**. Essas exceções serão apresentadas a seguir.

String vazia é false e não vazia é true

Qualquer **String** que seja **vazia** ("" ou ""), quando testado em um **assert** ou **if** ou **operador ternário** ou atribuído a uma variável **boolean**, será interpretado como **false**. Vide o código Groovy de exemplo a seguir, cujo assert **falharia**.

```
def nomeNogador = ""  
assert nomeJogador
```

Nas mesmas situações recém-citadas, qualquer **String não vazia**, será considerado true. No código do exemplo a seguir, o assert **passaria**.

```
def nomeNogador = "Zé Loko"  
assert nomeJogador
```

Número 0 (zero) é false. Qualquer outro é true

Qualquer número, qualquer tipo numérico (inclusive **BigDecimal**) que seja **0** (zero), quando testado em um assert ou if ou **operador ternário** ou atribuído a uma variável boolean, será interpretado como false. Vide o código Groovy de exemplo a seguir, cujos asserts **falhariam**.

```
def numeroLoko1 = 0  
def numeroLoko2 = 0.0  
def numeroLoko3 = new BigDecimal(0)  
assert numeroLoko1  
assert numeroLoko2  
assert numeroLoko3
```

Nas mesmas situações recém-citadas, qualquer número **diferente de zero, mesmo negativo**, será considerado true.

Coleções vazias são false e não vazias são true

Qualquer coleção (**List**, **Set** ou **Map**) que seja **vazia** (sem elementos), quando testado em um assert ou if ou **operador ternário** ou atribuído a uma variável **boolean**, será interpretado como false. Vide o código Groovy de exemplo a seguir, cujo assert **falharia**.

```
def jogadores = []  
assert jogadores
```

Nas mesmas situações recém-citadas, qualquer coleção **não vazia**, será considerado true. No código do exemplo a seguir, o assert **passaria**.

```
def campanha = [derrotas:0, vitorias:7]  
assert campanha
```

Métodos embarcados para as conversões mais comuns

A conversão de tipos é uma operação muito comum nos problemas computacionais atuais, principalmente devido à grande quantidade de integração de sistemas que ocorre hoje. Em Java, as conversões são um tanto verbosas. Em Groovy há métodos de conversão embarcados para as conversões mais comuns do cotidiano, como exemplificado a seguir.

Conversão de String para qualquer tipo numérico

O tipo **String** em Groovy já possui métodos embarcados para a conversão para todos os tipos numéricos da plataforma Java. No código de exemplo a seguir, variáveis **String** são convertidas com sucesso para **Integer**, **Long**, **BigInteger**, **Float**, **Double** e **BigDecimal**.

```
def txtNumericoInteiro = '8'
Integer inteiro = txtNumericoInteiro.toInteger()
Long longo = txtNumericoInteiro.toLong()
BigInteger bigInteger = txtNumericoInteiro.toBigInteger()

def txtNumericoReal = '8.5'
Float flutuante = txtNumericoReal.toFloat()
Double duplo = txtNumericoReal.toDouble()
BigDecimal bigDecimal = txtNumericoReal.toBigDecimal()
```

Vale destacar que, caso o valor da **String** não contenha um número válido para a conversão solicitada, uma exceção será lançada.

Conversão de String para Boolean

O tipo **String** em Groovy já possui métodos embarcados para a conversão para **Boolean**. Os valores "true", "y" e "1" são convertidos para **true**. Qualquer outro valor é convertido para **false**. No código do exemplo de conversão a seguir, todas as conversões resultariam em true, portanto, todos os assert **passariam**.

```
def textoBoleano = 'true'
assert textoBoleano.toBoolean()
textoBoleano = 'y'
assert textoBoleano.toBoolean()
textoBoleano = '1'
assert textoBoleano.toBoolean()
```

No código do exemplo de conversão a seguir, todas as conversões resultariam em false, portanto, todos os assert **falhariam**.

```
def textoBoleano = 'false'
assert textoBoleano.toBoolean()
textoBoleano = 'oi'
assert textoBoleano.toBoolean()
textoBoleano = ''
assert textoBoleano.toBoolean()
```

Manipulação de datas é muito simples

A manipulação de datas é uma operação muito comum nos problemas computacionais atuais. Em Java, as conversões `Date` para `String` e vice-versa, bem como adicionar ou subtrair dias a datas são um tanto verbosas, mesmo na versão 8. Em Groovy há métodos de conversão embarcados que facilitam as operações mais comuns com datas, como exemplificado a seguir.

Conversão de String para Date

Para converter uma **String** para **Date**, basta usar o método embarcado **parse()**, informando o formato (*pattern*) e o texto a ser convertido. O código a seguir demonstra como converter um texto em data no formato "dd/MM/yyyy".

```
def txtDataLoka = '01/01/1980'
def dataLoka = Date.parse('dd/MM/yyyy', txtDataLoka)
```

Conversão de Date para String

Para converter uma **Date** para **String**, basta usar o método embarcado **format()**, informando o formato (*pattern*). O código a seguir demonstra como converter uma data em texto no formato "dd/MM/yyyy".

```
def agora = new Date()
def txtAgora = agora.format('dd/MM/yyyy')
```

Adicionando ou subtraindo dias de Date

Em Groovy, para adicionar ou subtrair dias de uma data pode-se simplesmente usar os operadores `+` e `-`, literalmente, como é exemplificado no código a seguir, que **subtrai 1 dia** a uma data e depois **adiciona 2 dias** a outra.

```
def hoje = new Date() // por exemplo: 02/01/2000
def ontem = hoje-1    // 01/01/2000
def amanha = ontem+2  // 03/01/2000
```

No código anterior, o objeto `hoje` **não foi alterado**. Para que a data seja realmente alterada, é preciso, literalmente, substituir seu valor, como no exemplo a seguir.

```
def dataLoka = new Date() // por exemplo: 02/01/2000
dataLoka = dataLoka-1    // 01/01/2000
dataLoka = dataLoka+2    // 03/01/2000
dataLoka++               // 04/01/2000
```

Caso o operador - seja aplicado entre datas, o resultado será **a diferença de dias entre a primeira e a segunda**. O código a seguir demonstra isso ao comparar a data de hoje com a de amanhã.

```
def hoje = new Date() // por exemplo: 01/01/2000
def amanha = hoje+1   // 02/01/2000
println(amanha-hoje)  // resultado: 1
println(hoje-amanha)  // resultado: -1
```

Indo para a zero hora de uma Date

Em Groovy, é possível ir para a *zero hora* de uma data usando o método embarcado **clearTime()**. **Importante:** este método **altera** a data no qual é invocado, não retorna uma versão ajustada para *zero hora*. O exemplo a seguir ajustaria a data na qual foi invocado para a zero hora de seu dia.

```
def hoje = new Date() // por exemplo: 01/01/2000 15:30:00
hoje.clearTime()      // 01/01/2000 00:00:00
```

Acesso direto a métodos privados

Em Java, um método privado (private) não pode ser invocado, a não ser com uso da Reflection API, o que não é tarefa simples. Em Groovy, o acesso a métodos privados é transparente, sendo possível invocá-los como se fossem públicos. O código do exemplo a seguir é de uma classe Java que possui o método privado **atualizarClassificacao()**, que é invocado por seus outros dois métodos que são públicos, o **gol()** e o **finalizar()**.

```
public class Partida {
    public void gol(Jogador autor) {
        // ações a cada gol...
        this.atualizarClassificacao();
    }

    public void finalizar() {
        // ações ao finalizar partida
        this.atualizarClassificacao();
    }

    private void atualizarClassificacao() {
        // ações da atualização da classificação
    }
}
```


Para invocar o método **atualizarClassificacao()** de uma instância de Partida a partir de uma classe Groovy, basta invocá-lo diretamente, como se fosse público. O trecho de código Groovy a seguir instancia uma Partida e invoca seu método privado. Inclusive as IDEs Eclipse, IntelliJ e NetBeans sugerem e aceitam a invocação de métodos privados durante a edição de código Groovy.

```
def partidaLoka = new Partida()
partidaLoka.atualizarClassificacao()
```

Esse recurso é muito útil em casos como o do último exemplo, em que métodos privados possuem muita relevância, pois permite a criação de vários cenários de teste diretamente para eles.

Alteração do comportamento de métodos em classes e objetos em tempo de execução

Em algumas situações, métodos podem fazer ações muito complexas e/ou que dependem de elementos externos. Como exemplos temos: acesso a bancos de dados, consumo de APIs e uso de bibliotecas de terceiros. Para casos como esse, pode-se **sobrescrever** o comportamento do método em tempo de execução em Groovy, sem a necessidade de uma subclasse anônima nem de uma instância *Mock*.

Tomemos como base a classe a seguir como exemplo, a **ClienteRestTimesFutebol** e seu método **getToken()**. Esse método solicitaria um *token* de autenticação junto a uma API a partir de um **usuario** e uma senha.

```
public class ClienteRestTimesFutebol {
    public String getToken(String usuario, String senha) {
        // chamada à API para obter 'token'
    }
}
```

No código a seguir, definimos que **uma determinada instância** de **ClienteRestTimesFutebol**, a que chamamos de **clienteLoko** passa a ter um comportamento **diferente** para o método **getToken()**, enquanto a instância **clienteNormal** continua com seu método original:

```
def clienteLoko = new ClienteRestTimesFutebol()
cliente1.metaClass.getToken = { String u, String s ->
    return "fake-token"
}

def clienteNormal = new ClienteRestTimesFutebol()

def token1 = clienteLoko.getToken(null, null) // apenas retorna "fake-token"
def token2 = clienteNormal.getToken(null, null) // invocaria a versão original do "getToken()"
```

Caso seja necessário que todos os objetos de uma classe tenham o comportamento de um método alterado, é possível fazer esse ajuste **na classe**, consequentemente, **em todas as suas instâncias**. Como no exemplo a seguir, que mudaria o **getToken()** para qualquer instância de **ClienteRestTimesFutebol**:

```
ClienteRestTimesFutebol.metaClass.getToken = { String u, String s ->
    return "fake-token"
}
def cliente1 = new ClienteRestTimesFutebol()
def cliente2 = new ClienteRestTimesFutebol()

def token1 = cliente1.getToken(null, null) // apenas retorna "fake-token"
def token2 = cliente2.getToken(null, null) // apenas retorna "fake-token"
```

getters e setters podem invocados apenas com o nome do atributo

Os métodos getters e setters fazem parte do dia a dia dos desenvolvedores da plataforma Java. Fazem parte do padrão **JavaBeans** (<https://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html>) e são usados pelos principais *frameworks* Java. Seu uso é tão comum que as principais IDEs Java possuem assistentes para a rápida criação desses métodos. Em Groovy, existe a possibilidade invocar esses métodos apenas usando o nome do atributo **valorPasse** privado, acessível com um **get** e um **set** escritos de maneira correta:

```
public class Futebolista {
    private double valorPasse;
    public double getValorPasse() {
        return this.valorPasse;
    }
    public void setValorPasse(double valorPasse) {
        this.valorPasse = valorPasse;
    }
}
```

Para acessar os **setValorPasse()** e **getValorPasse()** em uma classe Groovy, poderíamos escrever um código como o do exemplo a seguir:

```
Futebolista jogadorLoko = // obtendo o Futebolista
jogadorLoko.valorPasse = 50000000
def passeDoLoko = jogadorLoko.valorPasse
```

Apesar de parecer que houve um acesso direto ao atributo, **não houve**. O compilador Groovy verifica se existe um **getValorPasse()** na segunda linha e se existe um **setValorPasse(double parametro0)** na terceira. Assim, a versão compilada do último código, para a JVM, seria como o código a seguir:

```
Futebolista jogadorLoko = // obtendo o Futebolista
jogadorLoko.setValorPasse(50000000)
double passeDoLoko = jogadorLoko.getValorPasse()
```

Caso o desenvolvedor ache melhor usar os *getters* e *setters* de forma explícita em suas classes Groovy, pode fazê-lo sem problema algum.

Referências

ANICHE, Maurício. *Testes automatizados de software: Um guia prático*. São Paulo: Casa do Código, 2015.

BAELDUNG. *The State of Java in 2018*. Disponível em <http://www.baeldung.com/java-in-2018>. Acesso em: 25 de maio de 2018.

BECK, Kent. *Test-Driven Development: By Example*. Boston(EUA): Addison-Wesley, 2003.

DEVELOPERS, Android. *Build instrumented unit tests*. Disponível em <https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests>. Acesso em: 10 de maio de 2018.

FOWLER, Martin. *Mocks Aren't Stubs*. Disponível em: <https://martinfowler.com/articles/mocksArentStubs.html>. Acesso em: 06 de janeiro de 2018.

GUTIERREZ, Felipe. *Pro Spring Boot*. New York (EUA): Apress, 2016.

JAVALIBS. *Spock Framework - Core Module*. Disponível em: <https://javalibs.com/artifact/org.spockframework/spock-core>. Acesso em: 18 de maio de 2018.

JAVALIBS. *Spring Boot*. Disponível em: <https://javalibs.com/artifact/org.springframework.boot/spring-boot>. Acesso em: 18 de maio de 2018.

KAPELONIS, Konstantinos. *Java Testing with Spock*. Shelter Island (EUA): Manning Publications, 2015.

KEITH, Mike; SCHINCARIOL, Merrick; KEITH, Jeremy;. *Pro JPA 2 - Mastering the JavaPersistence API*. New York (EUA): Apress, 2010.

MESZAROS, Gerard. *Four Phase Test*. Disponível em: <http://xunitpatterns.com/Four Phase Test.html>. Acesso em: 25 de novembro de 2017.

MESZAROS, Gerard. *Mocks, Fakes, Stubs and Dummies*. Disponível em: <http://xunitpatterns.com/Mocks, Fakes, Stubs and Dummies.html>. Acesso em: 05 de janeiro de 2018.

MESZAROS, Gerard. *XUnit Test Patterns: Refactoring Test Code*. New York (EUA): Pearson Education, 2007.

NORTH, Dan. MOORE, Ivan. *Introducing BDD*. Disponível em: <https://dannorth.net/introducing-bdd/>. Acesso em: 25 de novembro de 2017.

PEREIRA, Caio Ribeiro. *Construindo APIs REST com Node.js*. São Paulo: Casa do Código, 2016.

SILVEIRA, Guilherme; AMARAL, Mário. *Java SE 8 Programmer I: O guia para sua certificação Oracle Certified Associate*. São Paulo: Casa do Código, 2015.

SPOCK FRAMEWORK. *Spock Web Console*. Disponível em: <http://meetspock.appspot.com>. Acesso em: 15 de setembro de 2017.

SPRING.IO. *Testing*. Disponível em: <https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html>. Acesso em: 28 de maio de 2018.