6

Zombando - Removendo Externo Dependências

"Falar é barato. Mostre-me o código."

- Linus Torvalds

TDD é sobre velocidade. Queremos demonstrar rapidamente se uma ideia, conceito ou implementação é válida ou não. Mais adiante, queremos executar todos os testes rapidamente. Um grande gargalo para essa velocidade são as dependências externas. A configuração dos dados de banco de dados exigidos pelos testes pode ser demorada. A execução de testes que verificam código que usa APIs de terceiros pode ser lenta. Mais importante, escrever testes que satisfaçam todas as dependências externas pode se tornar muito complicado para valer a pena. Zombar de dependências externas e internas nos ajuda a resolver esses problemas.

Vamos desenvolver o que fizemos no Capítulo 3, *Red-Green-Refactor – Do Fracasso ao Sucesso até a Perfeição.* Vamos estender o Tic-Tac-Toe para usar o MongoDB como armazenamento de dados. Nenhum de nossos testes de unidade realmente usará o MongoDB, pois todas as comunicações serão simuladas. Ao final, criaremos um teste de integração que verificará se nosso código e o MongoDB estão realmente integrados.

Os seguintes tópicos serão abordados neste capítulo:

- Zombando
- Mockito
- Requisitos do jogo da velha v2
- Desenvolvendo Tic-Tac-Toe v2
- Testes de integração

Zombando

Todo mundo que já fez alguma das aplicações mais complicadas que *o Hello World* sabe que o código Java está cheio de dependências. Pode haver classes e métodos escritos por outros membros da equipe, bibliotecas de terceiros ou sistemas externos com os quais nos comunicamos.
Mesmo as bibliotecas encontradas dentro do JDK são dependências. Podemos ter uma camada de negócios que se comunica com a camada de acesso a dados que, por sua vez, usa drivers de banco de dados para buscar dados.
Ao trabalhar com testes de unidade, levamos as dependências ainda mais longe e geralmente consideramos todos os métodos públicos e protegidos (mesmo aqueles dentro da classe em que estamos trabalhando) como dependências que devem ser isoladas.

Ao fazer TDD no nível de testes unitários, criar especificações que contemplem todas essas dependências pode ser tão complexo que os próprios testes se tornariam gargalos. Seu tempo de desenvolvimento pode aumentar tanto que os benefícios obtidos com o TDD rapidamente se tornam ofuscados pelo custo cada vez maior. Mais importante, essas mesmas dependências tendem a criar testes tão complexos que contêm mais bugs do que a própria implementação.

A ideia do teste de unidade (especialmente quando vinculado ao TDD) é escrever especificações que validem se o código de uma única unidade funciona independentemente das dependências. Quando as dependências são internas, elas já são testadas e sabemos que elas fazem o que esperamos que façam.
Por outro lado, dependências externas requerem confiança. Devemos acreditar que eles funcionam corretamente. Mesmo se não o fizermos, a tarefa de realizar testes profundos, digamos, das classes JDK java.nio é muito grande para a maioria de nós. Além disso, esses problemas potenciais surgirão quando executarmos testes funcionais e de integração.

Enquanto estiver focado em unidades, devemos tentar remover todas as dependências que uma unidade pode usar.
A remoção dessas dependências é realizada por meio de uma combinação de design e simulação.

Os benefícios do uso de mocks incluem dependência de código reduzida e execução de texto mais rápida.

Mocks são pré-requisitos para a execução rápida de testes e a capacidade de se concentrar em uma única unidade de funcionalidade. Ao simular dependências externas ao método que está sendo testado, o desenvolvedor pode se concentrar na tarefa em questão sem perder tempo configurando-as. Em um caso de equipes maiores ou múltiplas trabalhando juntas, essas dependências podem nem ser desenvolvidas. Além disso, a execução de testes sem mocks tende a ser lenta. Bons candidatos para mocks são bancos de dados, outros produtos, serviços e assim por diante.

Antes de nos aprofundarmos nos mocks, vamos analisar as razões pelas quais alguém os empregaria em primeiro lugar.

Por que simula?

A lista a seguir representa alguns dos motivos pelos quais empregamos objetos simulados:

- O objeto gera resultados não determinísticos. Por exemplo, java.util.Date() fornece um resultado diferente toda vez que o instanciamos. Não podemos testar se seu resultado é o esperado:

```
data java.util.Date = new java.util.Date(); data.getTime(); // Qual
é o resultado que este método retorna?
```

- O objeto ainda não existe. Por exemplo, podemos criar uma interface e testá-la. O objeto que implementa essa interface pode não ter sido escrito no momento em que testamos o código que usa essa interface.
- O objeto é lento e requer tempo para processar. O exemplo mais comum seria bancos de dados. Podemos ter um código que recupera todos os registros e gera um relatório. Essa operação pode durar minutos, horas ou, em alguns casos, até dias.

Os motivos anteriores no suporte de objetos simulados se aplicam a qualquer tipo de teste.
Porém, no caso dos testes unitários e, principalmente, no contexto do TDD, há mais um motivo, talvez mais importante que outros. A simulação nos permite isolar todas as dependências usadas pelo método em que estamos trabalhando atualmente. Isso nos permite concentrar em uma única unidade e ignorar o funcionamento interno do código que a unidade invoca.

Terminologia

A terminologia pode ser um pouco confusa, especialmente porque pessoas diferentes usam nomes diferentes para a mesma coisa. Para tornar as coisas ainda mais complicadas, os frameworks mocking tendem a não ser consistentes ao nomear seus métodos.

Antes de prosseguirmos, vamos passar brevemente pela terminologia.

Test doubles é um nome genérico para todos os seguintes tipos:

- O objetivo do objeto fictício é atuar como um substituto para um argumento de método real
- O stub de teste pode ser usado para substituir um objeto real por um objeto específico de teste que alimenta as entradas indiretas desejadas no sistema em teste
- **Test Spy** captura as chamadas de saída indiretas feitas para outro componente pelo **System Under Test (SUT)** para verificação posterior pelo teste
- O objeto simulado substitui um objeto do qual o SUT depende, por um objeto específico de teste que verifica se está sendo usado corretamente pelo SUT
- Objeto falso substitui um componente do qual o SUT depende com uma implementação muito mais leve

Se você está confuso, pode ajudá-lo saber que você não é o único. As coisas são ainda mais complicadas do que isso, pois não há um acordo claro, nem um padrão de nomenclatura, entre frameworks ou autores. A terminologia é confusa e inconsistente, e os termos mencionados anteriormente não são aceitos por todos.

Para simplificar as coisas, ao longo deste livro usaremos a mesma nomenclatura usada por Mockito (nossa estrutura de escolha). Dessa forma, os métodos que você usará corresponderão à terminologia que você lerá mais adiante. Continuaremos usando mocking como um termo geral para o que outros podem chamar **de test doubles.** Além disso, usaremos um termo simulado ou espião para nos referirmos aos métodos Mockito .

Objetos simulados

Objetos simulados simulam o comportamento de objetos reais (muitas vezes complexos). Eles nos permitem criar um objeto que substituirá o real usado no código de implementação. Um objeto simulado espera que um método definido com argumentos definidos retorne o resultado esperado. Ele sabe de antemão o que deve acontecer e como esperamos que ele reaja.

Vejamos um exemplo simples:

```
coleção TicTacToeCollection = mock(TicTacToeCollection.class); assertThat(collection.drop()).isFalse();
doReturn(true).when(collection).drop();


assertThat(collection.drop()).isTrue();
```

Primeiro, definimos a coleção como uma simulação de TicTacToeCollection. Neste momento, todos os métodos deste objeto simulado são falsos e, no caso do Mockito, retornam valores padrão.
Isso é confirmado na segunda linha, onde afirmamos que o método drop retorna false. Mais adiante, especificamos que nossa coleção de objetos simulados deve retornar true quando o método drop for invocado. Finalmente, afirmamos que o método drop retorna true.

Criamos um objeto mock que retorna valores padrão e, para um de seus métodos, definimos qual deve ser o valor de retorno. Em nenhum momento foi usado um objeto real.

Mais tarde, trabalharemos com espiões que têm essa lógica invertida; um objeto usa métodos reais, a menos que especificado de outra forma. Veremos e aprenderemos mais sobre mocking em breve quando começarmos a estender nosso aplicativo Tic-Tac-Toe. Agora, vamos dar uma olhada em uma das estruturas de simulação Java chamada Mockito.

Mockito

Mockito é um framework de simulação com uma API limpa e simples. Os testes produzidos com o Mockito são legíveis, fáceis de escrever e intuitivos. Ele contém três métodos estáticos principais:

- mock(): Isso é usado para criar mocks. Opcionalmente, podemos especificar como esses mocks se comportam com when() e given(). spy(): Isso pode ser usado para simulação parcial. Objetos
- espionados invocam métodos reais, a menos que especifiquemos o contrário. Assim como mock(), o comportamento pode ser definido para cada método público ou protegido (excluindo estático). A principal diferença é que mock() cria uma falsificação de todo o objeto, enquanto spy() usa o objeto real. verifique(): Isso é usado para verificar se os métodos foram chamados com os argumentos
- fornecidos. É uma forma de afirmação.

Iremos aprofundar o Mockito assim que começarmos a codificar nosso aplicativo Tic-Tac-Toe v2. Primeiro, porém, vamos passar rapidamente por um novo conjunto de requisitos.

Requisitos do jogo da velha v2

Os requisitos do nosso aplicativo Tic-Tac-Toe v2 são simples. Devemos adicionar um armazenamento persistente para que os jogadores possam continuar jogando o jogo mais tarde. Usaremos o MongoDB para essa finalidade.

Adicione o armazenamento persistente do MongoDB ao aplicativo.

Desenvolvendo Tic-Tac-Toe v2

Continuaremos de onde paramos com Tic-Tac-Toe no Capítulo 3, *Red-Green-Refactor – Do fracasso ao sucesso até a perfeição.* O código fonte completo do aplicativo desenvolvido até agora pode ser encontrado em https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo.git. Use a opção **VCS|Checkout from Version Control|Git** do **IntelliJ IDEA** para clonar o código.

Como em qualquer outro projeto, a primeira coisa que precisamos fazer é adicionar as dependências ao build.gradle:

```
dependências {
        compile 'org.jongo:jongo:1.1' compile
        'org.mongodb:mongo-java-driver:2.+' testCompile 'junit:junit:4.12'
        testCompile 'org.mockito:mockito-all:1.+'

}
```

A importação do driver MongoDB deve ser autoexplicativa. Jongo é um conjunto muito útil de métodos utilitários que tornam o trabalho com código Java muito mais semelhante à linguagem de consulta Mongo. Para a parte de teste, continuaremos usando o JUnit com a adição de mocks, espiões e validações do Mockito.

Você notará que não instalaremos o MongoDB até o final. Com o Mockito, não precisaremos dele, pois todas as nossas dependências do Mongo serão zombadas.

Depois que as dependências forem especificadas, lembre-se de atualizá-las na caixa de diálogo **Projetos** do IDEA Gradle.

O código-fonte pode ser encontrado na ramificação 00-prerequisites do repositório Git tdd-java-ch06-tic-tac-toe-mongo (https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac- toe-mongo/branch/00- prereq uisites).

Agora que temos os pré-requisitos definidos, vamos começar a trabalhar no primeiro requisito.

Requisito 1 – movimentos de loja

Devemos ser capazes de salvar cada movimento no banco de dados. Como já temos toda a lógica do jogo implementada, isso deve ser trivial de se fazer. No entanto, este será um exemplo muito bom de uso simulado.

Implemente uma opção para salvar um único movimento com o número do turno, as *posições* dos eixos xey e o jogador *(X* ou O).

Devemos começar definindo o Java bean que representará nosso esquema de armazenamento de dados. Não há nada de especial nisso, então vamos pular esta parte com apenas uma nota.

Não gaste muito tempo definindo especificações para código clichê Java. Nossa implementação do bean contém equals e hashCode sobrescritos. Ambos são gerados automaticamente pelo IDEA e não fornecem um valor real, exceto para satisfazer a necessidade de comparar dois objetos do mesmo tipo (usaremos essa comparação mais adiante nas especificações). O TDD deve nos ajudar a projetar melhor e escrever código melhor. Escrever especificações 15-20 para definir código clichê que poderia ser escrito automaticamente pelo IDE (como é o caso do método equals) não nos ajuda a atingir esses objetivos. Dominar o TDD significa não apenas aprender a escrever especificações, mas também saber quando não vale a pena.

Dito isso, consulte o código-fonte para ver a especificação e implementação do bean em sua totalidade.

O código-fonte pode ser encontrado na ramificação 01-bean do repositório Git tdd-java-ch06-tic-tac toe-mongo (https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe -mongo/ramo/01-feijão).

As classes específicas são TicTacToeBeanSpec e TicTacToeBean.

Agora, vamos para uma parte mais interessante (mas ainda sem mocks, espiões e validações). Vamos escrever especificações relacionadas a salvar dados no MongoDB.

Para este requisito, criaremos duas novas classes dentro do pacote com.packtpublishing.tddjava.ch03tictactoe.mongo :

- TicTacToeCollectionSpec (dentro de src/test/java)
- TicTacToeCollection (dentro de src/main/java)

Especificação – nome do banco de dados

Devemos especificar qual será o nome do banco de dados que usaremos:

```
@Teste
public void whenInstantiatedThenMongoHasDbNameTicTacToe() {
    coleção TicTacToeCollection = new TicTacToeCollection(); assertEquals("tic-tac-toe",


coleção.getMongoCollection().getDBCollection().getDB().getName() ); }
```

Estamos instanciando uma nova classe TicTacToeCollection e verificando se o nome do banco de dados é o que esperamos.

Implementação

A implementação é muito simples, como segue:

```
private MongoCollection mongoCollection; protectedMongoCollection
getMongoCollection() { return mongoCollection;


} public TicTacToeCollection() lança UnknownHostException { DB db = new MongoClient().getDB("tic-
    tac-toe"); mongoCollection = new Jongo(db).getCollection("bla");

}
```

Ao instanciar a classe TicTacToeCollection , estamos criando um novo MongoCollection com o nome do banco de dados especificado (tic-tac-toe) e atribuindo-o à variável local.

Conto conosco. Falta apenas mais uma especificação até chegarmos à parte interessante onde usaremos mocks e spies.

Especificação – um nome para a coleção do Mongo Na implementação anterior, usamos bla

como o nome da coleção porque o Jongo nos obrigou a colocar alguma string. Vamos criar uma especificação que definirá o nome da coleção do Mongo que usaremos:

```
@Test
public void whenInstantiatedThenMongoCollectionHasNameGame() { coleção TicTacToeCollection = new
    TicTacToeCollection(); assertEquals( "jogo", coleção.getMongoCollection().getName());



}
```

Esta especificação é quase idêntica à anterior e provavelmente auto-explicativa.

Implementação

Tudo o que precisamos fazer para implementar essa especificação é alterar a string que usamos para definir o nome da coleção:

```
public TicTacToeCollection() lança UnknownHostException {
    DB db = new MongoClient().getDB("tic-tac-toe"); mongoCollection = new
    Jongo(db).getCollection("jogo");
}
```

Refatoração Você

pode ter a impressão de que a refatoração é reservada apenas para o código de implementação. No entanto, quando observamos os objetivos por trás da refatoração (código mais legível, ideal e mais rápido), eles se aplicam tanto às especificações quanto ao código de implementação.

As duas últimas especificações têm a instanciação da classe TicTacToeCollection repetida. Podemos movê-lo para um método anotado com @Before. O efeito será o mesmo (a classe será instanciada antes que cada método anotado com @Test seja executado) e removeremos o código duplicado. Como a mesma instanciação será necessária em outras especificações, remover a duplicação agora fornecerá ainda mais benefícios posteriormente. Ao mesmo tempo, evitaremos lançar UnknownHostException repetidamente:

```
coleção TicTacToeCollection;

@Before
public void before() lança UnknownHostException {
```

```
    coleção = new TicTacToeCollection();
}
@Teste
public void whenInstantiatedThenMongoHasDbNameTicTacToe() { // lança UnknownHostException
{
// Coleção TicTacToeCollection = new TicTacToeCollection();
    assertEquals("tic-tac-
        toe",
coleção.getMongoCollection().getDBCollection().getDB().getName() );

}

@Teste
public void whenInstantiatedThenMongoHasNameGame() { // lança UnknownHostException
{
// Coleção TicTacToeCollection = new TicTacToeCollection();
    assertEquals( "jogo",

        coleção.getMongoCollection().getName());
}
```

Use métodos de configuração e desmontagem. Os benefícios disso permitem que a preparação ou configuração e descarte ou desmontagem do código sejam executados antes e depois da aula ou de cada método de teste.

Em muitos casos, algum código precisa ser executado antes da classe de teste ou de cada método em uma classe. Para isso, o JUnit possui as anotações @BeforeClass e @Before que devem ser utilizadas na fase de configuração. O @BeforeClass executa o método associado antes que a classe seja carregada (antes que o primeiro método de teste seja executado). @Before executa o método associado antes de cada teste ser executado. Ambos devem ser usados quando houver certas pré-condições exigidas pelos testes. O exemplo mais comum é configurar dados de teste no banco de dados (espero na memória). Na extremidade oposta estão as anotações @After e @AfterClass , que devem ser usadas como a fase de desmontagem. Seu principal objetivo é destruir os dados ou estados criados durante a fase de configuração ou pelos próprios testes. Cada teste deve ser independente dos outros. Além disso, nenhum teste deve ser afetado pelos outros. A fase de desmontagem ajuda a manter o sistema como se nenhum teste tivesse sido executado anteriormente.

Agora vamos zombar, espionar e verificar!

Especificação – adicionando itens à coleção do Mongo Devemos criar um método que salve os dados no MongoDB. Depois de estudar a documentação do Jongo, descobrimos que existe o método MongoCollection.save , que faz exatamente isso. Ele aceita qualquer objeto como argumento de método e o transforma (usando Jackson) em JSON, que é usado nativamente no MongoDB. A questão é que depois de brincar com o Jongo, decidimos usar e, mais importante, confiar nessa biblioteca.

Podemos escrever as especificações do Mongo de duas maneiras. Mais um tradicional e apropriado para **End2End (E2E)** ou testes de integração seria trazer uma instância do MongoDB, invocar o método save do Jongo, consultar o banco de dados e confirmar que os dados foram realmente salvos. Não termina aqui, pois precisaríamos limpar o banco de dados antes de cada teste para garantir sempre que o mesmo estado não seja poluído pela execução de testes anteriores. Finalmente, quando todos os testes terminarem de ser executados, talvez queiramos parar a instância do MongoDB e liberar recursos do servidor para algumas outras tarefas.

Como você deve ter adivinhado, há muito trabalho envolvido para um único teste escrito dessa maneira. Além disso, não se trata apenas de trabalho que precisa ser investido na escrita de tais testes.
O tempo de execução aumentaria bastante. A execução de um teste que se comunica com um banco de dados não demora muito. Executar dez testes geralmente ainda é rápido. Correr centenas ou milhares pode levar muito tempo. O que acontece quando leva muito tempo para executar todos os testes de unidade? As pessoas perdem a paciência e começam a dividi-los em grupos ou desistem do TDD todos juntos. Dividir os testes em grupos significa que perdemos a confiança no fato de que nada foi quebrado, pois estamos testando continuamente apenas partes dele. Desistir do TDD... Bem, esse não é o objetivo que estamos tentando alcançar. No entanto, se levar muito tempo para executar testes, é razoável esperar que os desenvolvedores não queiram esperar até que terminem de executar antes de passar para a próxima especificação, e esse é o ponto em que paramos de fazer TDD.
Qual é uma quantidade razoável de tempo para permitir que nossos testes de unidade sejam executados?
Não existe uma regra universal que defina isso; no entanto, como regra geral, se o tempo for superior a 10-15 segundos, devemos começar a nos preocupar e dedicar tempo para otimizá-los.

> Os testes devem ser executados rapidamente. Os benefícios são que os testes são usados com frequência.

Se levar muito tempo para executar os testes, os desenvolvedores deixarão de usá-los ou executarão apenas um pequeno subconjunto relacionado às alterações que estão fazendo. Um benefício dos testes rápidos, além de favorecer seu uso, é o feedback rápido. Quanto mais cedo o problema for detectado, mais fácil será corrigi-lo. O conhecimento sobre o código que produziu o problema ainda é recente. Se um desenvolvedor já começou a trabalhar no próximo recurso enquanto aguarda a conclusão da execução dos testes, ele pode decidir adiar a correção do problema até que o novo recurso seja desenvolvido. Por outro lado, se eles abandonarem seu trabalho atual para corrigir o bug, perderão tempo na troca de contexto.

Se usar o banco de dados ao vivo para executar testes de unidade não for uma boa opção, qual é a alternativa? Zombando e espionando! Em nosso exemplo, sabemos qual método de uma biblioteca de terceiros deve ser invocado. Também investimos tempo suficiente para confiar nesta biblioteca (além dos testes de integração que serão realizados posteriormente). Uma vez que saibamos como usar a biblioteca, podemos limitar nosso trabalho a verificar se as invocações corretas dessa biblioteca foram feitas.

Vamos tentar.

Primeiro, devemos modificar nosso código existente e converter nossa instanciação do TicTacToeCollection em um espião:

```
import estático org.mockito.Mockito.*;
 ...
@Before
public void before() lança UnknownHostException { collection = spy(new
    TicTacToeCollection());
}
```

Espionar uma classe é chamado de zombaria **parcial .** Quando aplicada, a classe se comportará exatamente como se fosse instanciada normalmente. A principal diferença é que podemos aplicar mocks parciais e substituir um ou mais métodos por mocks. Como regra geral, tendemos a usar espiões principalmente nas classes em que estamos trabalhando. Queremos manter toda a funcionalidade de uma classe para a qual estamos escrevendo especificações, mas com uma opção adicional para, quando necessário, simular uma parte dela.

Agora vamos escrever a especificação em si. Pode ser o seguinte:

```
@Teste
public void whenSaveMoveThenInvokeMongoCollectionSave() { bean TicTacToeBean = new
    TicTacToeBean(3, 2, 1, 'Y'); MongoCollection mongoCollection = mock(MongoCollection.class);
    doReturn(mongoCollection).when(collection).getMongoCollection(); coleção.saveMove(bean);



    verifique(mongoCollection, times(1)).save(bean);
}
```

Métodos estáticos, como mock, doReturn e Verify , são todos da classe org.mockito.Mockito .

Primeiro, estamos criando um novo TicTacToeBean. Não há nada de especial lá. Em seguida, estamos criando um objeto simulado do MongoCollection. Como já estabelecemos que, ao trabalhar em nível de unidade, queremos evitar a comunicação direta com o banco de dados, zombar dessa dependência fornecerá isso para nós. Ele converterá uma classe real em uma classe simulada. Para a classe usando mongoCollection, parecerá real; no entanto, nos bastidores, todos os seus métodos são superficiais e não fazem nada. É como sobrescrever essa classe e substituir todos os métodos por outros vazios:

```
MongoCollection mongoCollection = mock(MongoCollection.class);
```

Em seguida, estamos dizendo que um mongoCollection simulado deve ser retornado sempre que chamarmos o método getMongoCollection da classe espiada de coleção. Em outras palavras, estamos dizendo à nossa classe para usar uma coleção falsa em vez da real:

```
doReturn(mongoCollection).when(collection).getMongoCollection();
```

Então, estamos chamando o método em que estamos trabalhando:

```
coleção.saveMove(bean);
```

Por fim, devemos verificar se a invocação correta da biblioteca Jongo é realizada uma vez:

```
verifique(mongoCollection, times(1)).save(bean);
```

Vamos tentar implementar esta especificação.

Implementação Para entender melhor a especificação que acabamos de escrever, vamos fazer apenas uma implementação parcial. Vamos criar um método vazio, saveMove. Isso permitirá que nosso código compile sem implementar a especificação ainda:

```
public void saveMove(TicTacToeBean bean) { }
```

Quando executamos nossas especificações (teste gradle), o resultado é o seguinte:

```
Procurado, mas não invocado:
mongoCollection.save(Turn: 3; X: 2; Y: 1; Jogador: Y);
```

Mockito nos diz que, de acordo com nossa especificação, esperamos que o método mongoCollection.save seja invocado e que a expectativa não foi cumprida.
Como o teste ainda está falhando, precisamos voltar e terminar a implementação. Um dos maiores pecados do TDD é ter um teste reprovado e passar para outra coisa.

> Todos os testes devem passar antes que um novo teste seja escrito. Os benefícios disso são que o foco é mantido em uma pequena unidade de trabalho e o código de implementação está (quase) sempre em condições de funcionamento.

> Às vezes, é tentador escrever vários testes antes da implementação real. Em outros casos, os desenvolvedores ignoram os problemas detectados pelos testes existentes e avançam para novos recursos. Isso deve ser evitado sempre que possível. Na maioria dos casos, a quebra dessa regra só introduzirá dívida técnica que precisará ser paga com juros. Um dos objetivos do TDD é garantir que o código de implementação esteja (quase) sempre funcionando conforme o esperado. Alguns projetos, por pressões para atingir a data de entrega ou manter o orçamento, quebram essa regra e dedicam tempo a novas funcionalidades, deixando para depois a correção do código associado a testes reprovados. Esses projetos geralmente acabam adiando o inevitável.

Vamos modificar a implementação também, por exemplo, o seguinte:

```
public void saveMove(TicTacToeBean bean)
    { getMongoCollection().save(null);
}
```

Se executarmos nossas especificações novamente, o resultado será o seguinte:

```
Os argumentos são diferentes! Procura-se:
mongoCollection.save(Turn: 3; X: 2; Y: 1; Jogador: Y);
```

Desta vez estamos invocando o método esperado, mas os argumentos que estamos passando para ele não são o que esperávamos. Na especificação, definimos a expectativa para um bean (new TicTacToeBean(3, 2, 1, 'Y')) e na implementação, passamos null. Além disso, as verificações do Mockito podem nos dizer se um método correto foi invocado e também se os argumentos passados para esse método estão corretos.

A implementação correta da especificação é a seguinte:

```
public void saveMove(TicTacToeBean bean)
    { getMongoCollection().save(bean);
}
```

Desta vez todas as especificações devem passar, e podemos, felizmente, prosseguir para a próxima.

Especificação - adicionando feedback de operação

Vamos alterar o tipo de retorno do nosso método saveMove para booleano:

```
@Test
public void whenSaveMoveThenReturnTrue() { bean TicTacToeBean
    = new TicTacToeBean(3, 2, 1, 'Y'); MongoCollection mongoCollection =
    mock(MongoCollection.class); doReturn(mongoCollection).when(collection).getMongoCollection();
    assertTrue(coleção.saveMove(bean));

}
```

Implementação Esta

implementação é muito simples. Devemos alterar o tipo de retorno do método. Lembre-se que uma das regras do TDD é usar a solução mais simples possível. A solução mais simples é retornar true como no exemplo a seguir:

```
public boolean saveMove(TicTacToeBean bean)
    { getMongoCollection().save(bean); retorne verdadeiro;

}
```

Refatorando Você

provavelmente notou que as duas últimas especificações têm as duas primeiras linhas duplicadas. Podemos refatorar o código de especificações movendo-os para o método anotado com @Before:

```
coleção TicTacToeCollection;
feijão TicTacToeBean;
MongoCollection mongoCollection;

@Antes da
public void before() lança UnknownHostException { collection = spy(new
    TicTacToeCollection()); bean = new TicTacToeBean(3, 2, 1, 'Y'); mongoCollection
    = mock(MongoCollection.class);

}
...
@Test
public void whenSaveMoveThenInvokeMongoCollectionSave() { // TicTacToeBean bean = new
TicTacToeBean(3, 2, 1, 'Y'); // MongoCollection mongoCollection = mock(MongoCollection.class);
```

```
    doReturn(mongoCollection).when(collection).getMongoCollection(); coleção.saveMove(bean);
    verifique(mongoCollection, times(1)).save(bean);

}

@Test
public void whenSaveMoveThenReturnTrue() { // Bean TicTacToeBean
= new TicTacToeBean(3, 2, 1, 'Y'); // MongoCollection mongoCollection =
mock(MongoCollection.class);
      doReturn(mongoCollection).when(collection).getMongoCollection(); assertTrue(coleção.saveMove(bean));

}
```

Especificação - tratamento de erros

Agora vamos contemplar a opção de que algo pode dar errado ao usar o MongoDB. Quando, por exemplo, uma exceção é lançada, podemos querer retornar false do nosso método saveMove :

```
@Test
public void dadoExceptionWhenSaveMoveThenReturnFalse() {
    doThrow(new MongoException("Bla"))
        .when(mongoCollection).save(any(TicTacToeBean.class));
    doReturn(mongoCollection).when(collection).getMongoCollection(); assertFalse(coleção.saveMove(bean));

}
```

Aqui, apresentamos outro método Mockito: doThrow. Ele age de maneira semelhante ao doReturn e lança uma exceção quando as condições definidas são atendidas. A especificação lançará a MongoException quando o método save dentro da classe mongoCollection for invocado. Isso nos permite afirmar que nosso método saveMove retorna false quando uma exceção é lançada.

Implementação

A implementação pode ser tão simples quanto adicionar um bloco try/catch :

```
public boolean saveMove(TicTacToeBean bean) {
    tente
        { getMongoCollection().save(bean); retorne verdadeiro;

    } catch (Exception e) { return false;

    }
}
```

Especificação – estado claro entre jogos Esta é uma aplicação muito

simples que, pelo menos neste momento, pode armazenar apenas uma sessão de jogo. Sempre que uma nova instância é criada, devemos recomeçar e remover todos os dados armazenados no banco de dados. A maneira mais fácil de fazer isso é simplesmente descartar a coleção do MongoDB. O Jongo tem o método MongoCollection.drop() que pode ser usado para isso. Criaremos um novo método, drop, que atuará de forma semelhante ao saveMove.

Se você não trabalhou com Mockito, MongoDB e/ou Jongo, é provável que você não tenha conseguido fazer os exercícios deste capítulo sozinho e decidiu seguir as soluções que fornecemos. Se for esse o caso, este é o momento em que você pode querer mudar de marcha e tentar escrever as especificações e a implementação por conta própria.

Devemos verificar se MongoCollection.drop() é invocado a partir de nosso próprio método drop() dentro da classe TicTacToeCollection . Experimente você mesmo antes de olhar para o código a seguir. Deve ser quase o mesmo que fizemos com o método save :

```
@Teste
public void whenDropThenInvokeMongoCollectionDrop() {
    doReturn(mongoCollection).when(collection).getMongoCollection(); coleção.drop();
    verifique(mongoCollection).drop();

}
```

Implementação

Como este é um método wrapper, implementar esta especificação deve ser bastante fácil:

```
public void drop() {
    getMongoCollection().drop();
}
```

Especificação – feedback da operação de descarte

Estamos quase terminando esta aula. Restam apenas duas especificações.

Vamos nos certificar de que, em circunstâncias normais, retornamos true:

```
@Teste
public void whenDropThenReturnTrue() {
    doReturn(mongoCollection).when(collection).getMongoCollection(); assertTrue(coleção.drop());

}
```

Implementação Se as coisas parecem muito fáceis com o TDD, então isso é proposital. Estamos dividindo as tarefas em entidades tão pequenas que, na maioria dos casos, implementar uma especificação é muito fácil. Este não é exceção:

```
public boolean drop()
    { getMongoCollection().drop();
    retorne verdadeiro;
}
```

Especificação - tratamento de erros

Por fim, vamos garantir que o método drop retorne false em caso de Exception:

```
@Test
public void dadoExceptionWhenDropThenReturnFalse() {
    doThrow(new MongoException("Bla")).when(mongoCollection).drop();
    doReturn(mongoCollection).when(collection).getMongoCollection(); assertFalse(coleção.drop());

}
```

Implementação

Vamos apenas adicionar um bloco try/catch :

```
public boolean drop() { try
    { getMongoCollection().drop();
        retorne verdadeiro;

    } catch (Exception e) { return false;

    }
}
```

Com esta implementação, terminamos com a classe TicTacToeCollection que atua como uma camada entre nossa classe principal e o MongoDB.

O código-fonte pode ser encontrado na ramificação 02-save-move do repositório Git tdd-java-ch06-tic tac-toe-mongo (https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac -toe-mongo/branch/02-save-m ove). As classes em particular são TicTacToeCollectionSpec e

Coleção TicTacToe.

Requisito 2 – armazene a cada turno

Vamos empregar os métodos TicTacToeCollection dentro de nossa classe principal TicTacToe.

Sempre que um jogador joga um turno com sucesso, devemos salvá-lo no BD. Além disso, devemos descartar a coleção sempre que uma nova classe for instanciada para que um novo jogo não se sobreponha ao antigo. Poderíamos torná-lo muito mais elaborado do que isso; no entanto, para os propósitos deste capítulo e aprender como usar mocking, este requisito deve servir para
agora.

Salve cada turno no banco de dados e certifique-se de que uma nova sessão limpe os dados antigos.

Vamos fazer alguma configuração primeiro.

Especificação – criando uma nova coleção Como todos os nossos métodos que serão usados para se comunicar com o MongoDB estão na classe TicTacToeCollection , devemos nos certificar de que ela seja instanciada. A especificação pode ser a seguinte:

```
@Test
public void whenInstantiatedThenSetCollection() {
    assertNotNull(ticTacToe.getTicTacToeCollection());
}
```

A instanciação do TicTacToe já é feita no método anotado com @Before.
Com essa especificação, garantimos que a coleção também seja instanciada.

Implementação Não há nada de especial nesta implementação. Devemos simplesmente sobrescrever o construtor padrão e atribuir uma nova instância à variável ticTacToeCollection .

Para começar, devemos adicionar uma variável local e um getter para TicTacToeCollection:

```
private TicTacToeCollection ticTacToeCollection;

protegido TicTacToeCollection getTicTacToeCollection() {
    return ticTacToeCollection;
}
```

Agora tudo o que resta é instanciar uma nova coleção e atribuí-la à variável quando a classe principal for instanciada:

```
public TicTacToe() lança UnknownHostException {
    this(new TicTacToeCollection());

} protected TicTacToe(coleçãoTicTacToeCollection) { ticTacToeCollection = coleção;

}
```

Também criamos outra forma de instanciar a classe passando TicTacToeCollection como argumento. Isso será útil dentro das especificações como uma maneira fácil de passar uma coleção simulada.

Agora vamos voltar para a classe de especificações e fazer uso deste novo construtor.

Refatoração de especificações

Para utilizar um construtor TicTacToe recém-criado , podemos fazer algo como o seguinte:

```
coleção privada TicTacToeCollection;

@Antes da
public final void before() lança UnknownHostException {
    coleção = mock(TicTacToeCollection.class);
// ticTacToe = new TicTacToe(); ticTacToe = new
    TicTacToe(coleção);
}
```

Agora todas as nossas especificações usarão uma versão simulada do TicTacToeCollection. Existem outras maneiras de injetar dependências simuladas (por exemplo, com Spring); no entanto, quando possível, sentimos que a simplicidade supera as estruturas complicadas.

Especificação – armazenando o movimento atual

Sempre que jogamos um turno, ele deve ser salvo no BD. A especificação pode ser a seguinte:

```
@Teste
public void whenPlayThenSaveMoveIsInvoked() {
    TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X'); ticTacToe.play(move.getX(),
    move.getY()); verificar(coleção).saveMove(mover);

}
```

Até agora, você já deve estar familiarizado com o Mockito, mas vamos passar pelo código para relembrar:

1. Primeiro, estamos instanciando um TicTacToeBean , pois ele contém os dados que nossas coleções esperam:

```
TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X');
```

2. Em seguida, é hora de jogar um turno real:

```
ticTacToe.play(move.getX(), move.getY());
```

3. Finalmente, precisamos verificar se o método saveMove é realmente invocado:

```
verifique(coleção, times(1)).saveMove(move);
```

Como fizemos ao longo deste capítulo, isolamos todas as invocações externas e focamos apenas na unidade (jogo) em que estamos trabalhando. Tenha em mente que esse isolamento é limitado apenas aos métodos públicos e protegidos. Quando se trata da implementação real, podemos optar por adicionar a invocação saveMove ao método público play ou um dos métodos privados que escrevemos como resultado da refatoração que fizemos anteriormente.

Implementação Esta

especificação apresenta alguns desafios. Primeiro, onde devemos colocar a invocação do método saveMove ? O método privado setBox parece um bom lugar. É aí que estamos fazendo as validações de se o turno é válido, e se for, podemos chamar o método saveMove . No entanto, esse método espera um bean em vez das variáveis x, y e lastPlayer que estão sendo usadas agora, então podemos querer alterar a assinatura do método setBox .

É assim que o método se parece agora:

```
private void setBox(int x, int y, char lastPlayer) {
    if (board[x - 1][y - 1] != '\0') { throw new
        RuntimeException("Caixa está ocupada"); } else { board[x - 1][y - 1] =
    lastPlayer;

    }
}
```

É assim que fica depois que as alterações necessárias são aplicadas:

```
private void setBox(TicTacToeBean bean) {
    if (board[bean.getX() - 1][bean.getY() - 1] != '\0') { throw new RuntimeException("Caixa está
        ocupada"); } senão {

        board[bean.getX() - 1][bean.getY() - 1] = lastPlayer; getTicTacToeCollection().saveMove(bean);

    }
}
```

A alteração da assinatura setBox desencadeia algumas outras alterações. Como ele é invocado a partir do método play , precisaremos instanciar o bean lá:

```
public String play(int x, int y) { checkAxis(x); checkAxis(y);
    últimoJogador = próximoJogador(); // setBox(x, y,
    lastPlayer);
```

```
        setBox(new TicTacToeBean(1, x, y, lastPlayer)); if (éVencer(x, y)) {

            return lastPlayer + " é o vencedor"; } else if (isDraw()) { return
        RESULT_DRAW;

        } senão {
            return NO_WINNER;
        }
    }
```

Você deve ter notado que usamos um valor constante 1 como um turno. Ainda não há especificação que diga o contrário, então pegamos um atalho. Nós vamos lidar com isso mais tarde.

Todas essas mudanças ainda eram muito simples e levou um período de tempo razoavelmente curto para implementá-las. Se as mudanças fossem maiores, poderíamos ter escolhido um caminho diferente; e fez uma alteração mais simples para chegar à solução final por meio da refatoração. Lembre-se que a velocidade é a chave. Você não quer ficar preso a uma implementação que não passa nos testes por muito tempo.

Especificação – tratamento de erros O que acontece se

um movimento não puder ser salvo? Nosso método auxiliar saveMove retorna true ou false dependendo do resultado da operação do MongoDB. Podemos querer lançar uma exceção quando ela retornar false.

Antes de mais nada: devemos alterar a implementação do método before e garantir que, por padrão, saveMove retorne true:

```
@Before
public final void before() lança UnknownHostException { collection =
    mock(TicTacToeCollection.class);
doReturn(true).when(collection).saveMove(any(TicTacToeBean.class));
    ticTacToe = new TicTacToe(coleção);
}
```

Agora que acabamos com a coleção simulada com o que achamos ser o comportamento padrão (retorna true quando saveMove é invocado), podemos prosseguir e escrever a especificação:

```
@Test
public void whenPlayAndSaveReturnsFalseThenThrowException()
{ doReturn(false).when(collection).saveMove(any(TicTacToeBean.class))
;
    TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X');
```

```
    exceção.expect(RuntimeException.class); ticTacToe.play(move.getX(),
    move.getY());
}
```

Estamos usando o Mockito para retornar false quando saveMove é invocado. Como, neste caso, não nos importamos com uma invocação específica de saveMove, usamos any(TicTacToeBean.class) como argumento do método. Este é outro dos métodos estáticos do Mockito.

Depois que tudo estiver definido, usamos uma expectativa JUnit da mesma forma que fizemos antes no Capítulo 3, *Red-Green-Refactor – Do fracasso ao sucesso até a perfeição.*

Implementação

Vamos fazer um if simples e lançar um RuntimeException quando o resultado não for esperado:

```
private void setBox(TicTacToeBean bean) {
    if (board[bean.getX() - 1][bean.getY() - 1] != '\0') { throw new RuntimeException("Caixa está
        ocupada"); } else { board[bean.getX() - 1][bean.getY() - 1] = lastPlayer; //
    getTicTacToeCollection().saveMove(bean);


        if (!getTicTacToeCollection().saveMove(bean)) {
            throw new RuntimeException("Falha ao salvar no banco de dados");
        }
    }
}
```

Especificação – jogadores alternativos Você se lembra

do turno que codificamos para ser sempre 1? Vamos corrigir esse comportamento.

Podemos invocar o método play duas vezes e verificar se o turno muda de 1 para 2:

```
@Teste
public void whenPlayInvokedMultipleTimesThenTurnIncreases() { TicTacToeBean move1 = new
    TicTacToeBean(1, 1, 1, 'X'); ticTacToe.play(move1.getX(), move1.getY()); verifique(coleção,
    times(1)).saveMove(move1); TicTacToeBean move2 = new TicTacToeBean(2, 1, 2, 'O');
    ticTacToe.play(move2.getX(), move2.getY()); verifique(coleção, times(1)).saveMove(move2);



}
```

Implementação

Como em quase tudo feito no estilo TDD, a implementação é bastante fácil:

```
private int turno = 0;
 ...
public String play(int x, int y) { checkAxis(x);
    checkAxis(y); últimoJogador = próximoJogador();
    setBox(new TicTacToeBean(++turno, x, y,
    lastPlayer)); if (éVencer(x, y)) {


        lastPlayer + } else if (isDraw()) { return RESULT_DRAW; }
    else { return NO_WINNER;




    }
}
```

Exercícios

Ainda faltam mais algumas especificações e suas implementações. Devemos invocar o método drop() sempre que nossa classe TicTacToe for instanciada. Também devemos garantir que RuntimeException seja lançada quando drop() retornar false. Deixaremos essas especificações e suas implementações como exercício para você.

O código-fonte pode ser encontrado na ramificação 03-mongo do repositório Git tdd-java-ch06-tic-tac toe-mongo (https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe -mongo/filial/03-mongo).

As classes em particular são TicTacToeSpec e TicTacToe.

Testes de integração

Fizemos muitos testes unitários. Confiamos muito na confiança. Unidade após unidade foi especificada e implementada. Enquanto trabalhávamos nas especificações, isolamos tudo, menos as unidades em que estávamos trabalhando, e verificamos que uma invocava a outra corretamente. No entanto, chegou a hora de validar que todas essas unidades são realmente capazes de se comunicar com o MongoDB. Podemos ter cometido um erro ou, mais importante, podemos não ter o MongoDB funcionando. Seria um desastre descobrir que, por exemplo, implantamos nosso aplicativo, mas esquecemos de abrir o banco de dados ou que a configuração (IP, porta e assim por diante) não está definida corretamente.

O objetivo do teste de integração é validar, como você deve ter adivinhado, a integração de componentes, aplicativos, sistemas separados etc. Se você se lembrar da pirâmide de testes, ela afirma que os testes de unidade são os mais fáceis de escrever e mais rápidos de executar, portanto, devemos manter outros tipos de testes limitados a coisas que os UTs não cobrem.

Devemos isolar nossos testes de integração de forma que possam ser executados ocasionalmente (antes de enviarmos nosso código para o repositório ou como parte de nosso processo de **integração contínua (CI))** e manter o teste de unidade como um loop de feedback contínuo.

Separação de testes

Se seguirmos algum tipo de convenção, é bastante fácil separar os testes no Gradle. Podemos ter nossos testes em diferentes diretórios e pacotes distintos ou, por exemplo, com diferentes sufixos de arquivo. Neste caso, escolhemos o último. Todas as nossas classes de especificação são nomeadas com o sufixo Spec (ou seja, TicTacToeSpec). Podemos fazer uma regra que todos os testes de integração tenham o sufixo Integ .

Com isso em mente, vamos modificar nosso arquivo build.gradle .

Primeiro, informaremos ao Gradle que apenas as classes que terminam com Spec devem ser usadas pela tarefa de teste :

```
test
    { include '**/*Spec.class'
}
```

Em seguida, podemos criar uma nova tarefa, testInteg:

```
task testInteg(type: Test) {
        include '**/*Integ.class'
}
```

Com essas duas adições ao build.gradle, continuamos tendo as tarefas de teste que usamos muito ao longo do livro; no entanto, desta vez, estão limitados apenas às especificações (testes unitários). Além disso, todos os testes de integração podem ser executados clicando na tarefa testInteg na janela Gradle projects IDEA ou executando o seguinte comando no prompt de comando:

```
gradle testInteg
```

Vamos escrever um teste de integração simples.

O teste de integração

Criaremos uma classe TicTacToeInteg dentro do pacote com.packtpublishing.tddjava.ch03tictactoe no diretório src/test/java . Como sabemos que o Jongo lança uma exceção se não puder se conectar ao banco de dados, uma classe de teste pode ser tão simples quanto o seguinte:

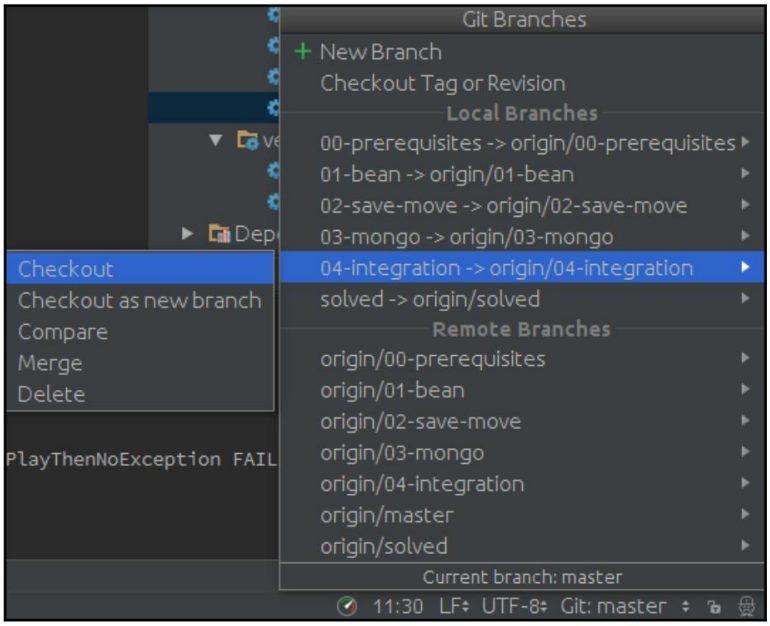
```
importar org.junit.Test; importar
java.net.UnknownHostException; importar estático
org.junit.Assert.*;

classe pública TicTacToeInteg {

    @Test
    public void dadoMongoDbIsRunningWhenPlayThenNoException()
            lança UnknownHostException { TicTacToe
        ticTacToe = new TicTacToe(); assertEquals(TicTacToe.NO_WINNER,
        ticTacToe.play(1, 1));
    }
}
```

A invocação de assertEquals é apenas uma precaução. O objetivo real deste teste é garantir que nenhuma exceção seja lançada. Como não iniciamos o MongoDB (a menos que você seja muito proativo e tenha feito isso sozinho, nesse caso você deve interrompê-lo), o teste deve falhar:

```
vfarcic@viktor: ~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo

vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$ gradle testInteg
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:testInteg

com.packtpublishing.tddjava.ch03tictactoe.TicTacToeInteg > givenMongoDbIsRunning
WhenPlayThenNoException                                                      F
AILED
    java.lang.RuntimeException at TicTacToeInteg.java:12

1 test completed, 1 failed
:testInteg FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':testInteg'.
> There were failing tests. See the report at: file:///home/vfarcic/IdeaProjects
/tdd-java-ch06-tic-tac-toe-mongo/build/reports/tests/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug
option to get more log output.

BUILD FAILED

Total time: 14.6 secs
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$
```

Agora que sabemos que o teste de integração funciona, ou em outras palavras, que ele realmente falha quando o MongoDB não está funcionando, vamos tentar novamente com o banco de dados iniciado. Para abrir o MongoDB, usaremos o Vagrant para criar uma máquina virtual com o Ubuntu OS. O MongoDB será executado como um Docker.

Certifique-se de que a ramificação **de integração 04** está com check-out:



No prompt de comando, execute o seguinte comando:

$ vagando para cima

Seja paciente até que a VM esteja funcionando (pode demorar um pouco quando executado pela primeira vez, especialmente em uma largura de banda mais lenta). Quando terminar, execute novamente os testes de integração:

```
vfarcic@viktor: ~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$ gradle testInteg
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:testInteg

BUILD SUCCESSFUL

Total time: 4.46 secs
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$
```

Funcionou e agora estamos confiantes de que estamos realmente integrados ao MongoDB.

Este foi um teste de integração muito simplista e, no mundo real, faríamos um pouco mais do que este único teste. Poderíamos, por exemplo, consultar o banco de dados e confirmar que os dados foram armazenados corretamente. No entanto, o objetivo deste capítulo foi aprender tanto como zombar e que não devemos depender apenas de testes de unidade. O próximo capítulo explorará a integração e os testes funcionais com mais profundidade.

O código-fonte pode ser encontrado na ramificação 04-integration do repositório Git tdd-java-ch06-tic tac-toe-mongo (https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe -mongo/branch/04-integração).

Resumo

Técnicas de zombaria e espionagem são usadas para isolar diferentes partes do código ou bibliotecas de terceiros. Eles são essenciais se quisermos prosseguir com grande velocidade, não apenas durante a codificação, mas também durante a execução de testes. Testes sem mocks costumam ser muito complexos para serem escritos e podem ser tão lentos que, com o tempo, o TDD tende a se tornar quase impossível. Testes lentos significam que não poderemos executá-los toda vez que escrevermos uma nova especificação. Isso por si só leva à deterioração da confiança que temos em nossos testes, já que apenas uma parte deles é executada.

A simulação não é útil apenas como forma de isolar dependências externas, mas também como forma de isolar nosso próprio código de uma unidade em que estamos trabalhando.

Neste capítulo, apresentamos o Mockito como, em nossa opinião, o framework com melhor equilíbrio entre funcionalidade e facilidade de uso. Convidamos você a investigar sua documentação com mais detalhes (http://mockito.org/), assim como outros frameworks Java dedicados ao mocking. EasyMock (http://easymock.org/), JMock (http://www.jmock.org/), e PowerMock (https://code.google.com/p/powermock/) são alguns dos mais populares.

No próximo capítulo vamos colocar alguns conceitos de programação funcional, bem como alguns conceitos de TDD aplicados a eles. Para isso, parte da API funcional do Java será apresentada.