Machine Translated by Google

3

Red-Green-Refactor - De Fracasso Através do Sucesso até Perfeição

"Saber não é suficiente; devemos aplicar. Desejar não é suficiente; devemos fazer."

– Bruce Lee

A técnica **Red-Green-Refactor** é a base do **desenvolvimento orientado a testes (TDD).** É um jogo de pingue-pongue no qual estamos alternando entre testes e código de implementação em grande velocidade. Falharemos, depois teremos sucesso e, finalmente, melhoraremos.

Vamos desenvolver um jogo Tic-Tac-Toe passando por cada requisito, um de cada vez. Vamos escrever um teste e ver se ele falha. Em seguida, escreveremos o código que implementa esse teste, executaremos todos os testes e os veremos com sucesso. Por fim, refatoramos o código e tentamos melhorá-lo. Este processo será repetido muitas vezes até que todos os requisitos sejam implementados com sucesso.

Começaremos configurando o ambiente com Gradle e JUnit. Então, iremos um pouco mais fundo no processo Red-Green-Refactor. Quando estivermos prontos com a configuração e a teoria, passaremos pelos requisitos de alto nível do aplicativo.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Com tudo definido, vamos mergulhar direto no código — um requisito por vez. Depois que tudo estiver pronto, daremos uma olhada na cobertura do código e decidiremos se é aceitável ou se mais testes precisam ser adicionados.

Os seguintes tópicos serão abordados neste capítulo:

- Configurando o ambiente com Gradle e JUnit
- O processo Red-Green-Refactor
- Requisitos do jogo da velha
- Desenvolvendo o jogo da velha
- Cobertura de código
- Mais exercícios

Configurando o ambiente com Gradle e JUnit

Você provavelmente está familiarizado com a configuração de projetos Java. No entanto, você pode não ter trabalhado com o IntelliJ IDEA antes ou pode ter usado o Maven em vez do Gradle. Para garantir que você possa seguir o exercício, passaremos rapidamente pela configuração.

Configurando o projeto Gradle/Java no IntelliJ IDEA

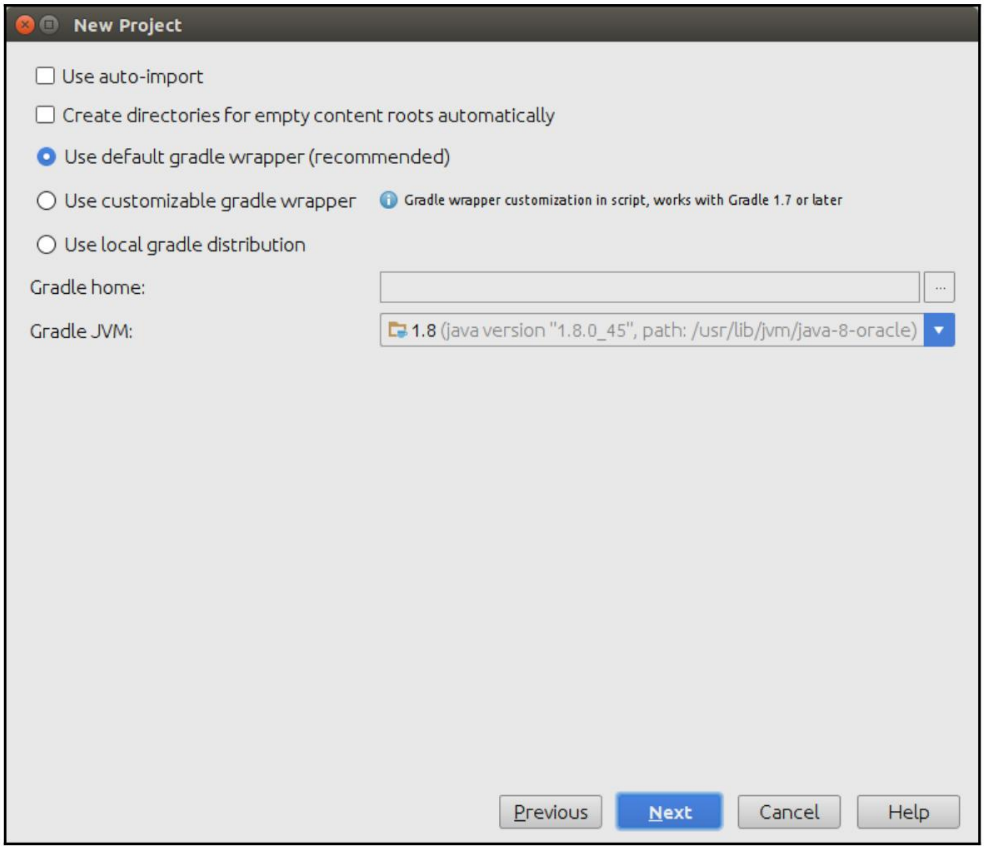
O objetivo principal deste livro é ensinar TDD, então não entraremos em detalhes sobre Gradle e IntelliJ IDEA. Ambos são usados como exemplo. Todos os exercícios deste livro podem ser feitos com diferentes opções de IDE e ferramentas de construção. Você pode, por exemplo, usar Maven e Eclipse. Para a maioria, pode ser mais fácil seguir as mesmas diretrizes apresentadas ao longo do livro, mas a escolha é sua.

As etapas a seguir criarão um novo projeto Gradle no IntelliJ IDEA:

1. Abra o **IntelliJ IDEA.** Clique em **Criar novo projeto** e selecione **Gradle** no menu do lado esquerdo. Em seguida, clique em **Avançar.**
2. Se você estiver usando o IDEA 14 e superior, será solicitado um **ID de artefato.** Modelo tdd-java-ch03-tic-tac-toe e clique em **Next** duas vezes. Digite tdd-java-ch03-tic-tac-toe como o nome do projeto. Em seguida, clique no botão **Concluir :**

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google



Na caixa de diálogo **Novo Projeto ,** podemos observar que o IDEA já criou o arquivo build.gradle . Abra-o e você verá que ele já contém a dependência JUnit.
Como esta é nossa estrutura de escolha neste capítulo, não há configuração adicional que devamos fazer. Por padrão, build.gradle é configurado para usar Java 1.5 como uma configuração de compatibilidade de origem. Você pode alterá-lo para qualquer versão que preferir. Os exemplos neste capítulo não usarão nenhum dos recursos Java que vieram após a Versão 5, mas isso não significa que você não possa resolver o exercício usando, por exemplo, o JDK 8.

Nosso arquivo build.gradle deve se parecer com o seguinte:

```
aplicar plugin: 'java'

versão = '1.0'

repositórios {
    mavenCentral()
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

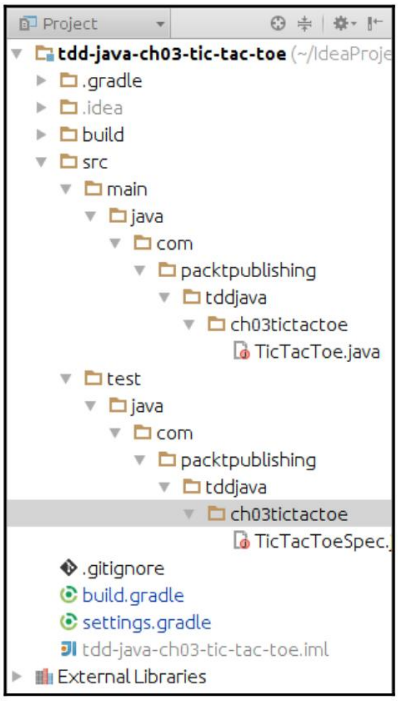

```
}

dependências {
    testCompile group: 'junit', nome: 'junit', versão: '4.11'
}
```

Agora, tudo o que resta a fazer é criar pacotes que usaremos para testes e implementação. Na caixa de diálogo **Projeto ,** clique com o botão direito do mouse para abrir o menu **Contexto** e selecione **Novo|Diretório.** Digite src/test/java/com/packtpublishing/tddjava/ch03tictactoe e clique no botão **OK** para criar o pacote de testes. Repita as mesmas etapas com o diretório src/main/java/com/packtpublishing/tddjava/ch03tictactoe para criar o pacote de implementação.

Finalmente, precisamos fazer as classes de teste e implementação. Crie a classe TicTacToeSpec dentro do pacote com.packtpublishing.tddjava.ch03tictactoe no diretório src/test/java . Esta classe conterá todos os nossos testes. Repita o mesmo para a classe TicTacToe no diretório src/main/java .

A estrutura do seu **projeto** deve ser semelhante à apresentada na captura de tela a seguir:



Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

O código-fonte pode ser encontrado na ramificação 00-setup do repositório Git tdd-java-ch03-tic-tac toe em https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch /00-configuração.

Sempre separe os testes do código de implementação.

Os benefícios são os seguintes: isso evita o empacotamento acidental de testes junto com os binários de produção; muitas ferramentas de compilação esperam que os testes estejam em um determinado diretório de origem.

Uma prática comum é ter pelo menos dois diretórios de origem. O código de implementação deve estar localizado em src/main/java e o código de teste em src/test/java. Em projetos maiores, o número de diretórios de origem pode aumentar, mas a separação entre implementação e testes deve permanecer.

Ferramentas de compilação, como Maven e Gradle, esperam diretórios de origem, separação e convenções de nomenclatura.

É isso. Estamos prontos para começar a trabalhar em nosso aplicativo Tic-Tac-Toe usando JUnit como estrutura de teste de escolha e Gradle para compilação, dependências, testes e outras tarefas. No Capítulo 1, *Por que devo me importar com o desenvolvimento orientado a testes?,* você encontrou pela primeira vez o procedimento Red Green-Refactor. Como é a pedra angular do TDD e o objetivo principal do exercício deste capítulo, pode ser uma boa ideia entrar em mais detalhes antes de iniciarmos o desenvolvimento.

O processo Red-Green-Refactor

O processo Red-Green-Refactor é a parte mais importante do TDD. É o pilar principal, sem o qual nenhum outro aspecto do TDD funcionará.

O nome vem dos estados em que nosso código está dentro do ciclo. Quando no estado vermelho, o código não funciona; quando no estado verde, tudo está funcionando como esperado, mas não necessariamente da melhor maneira possível. Refatorar é a fase em que sabemos que os recursos estão bem cobertos com testes e, portanto, nos dá confiança para alterá-lo e torná-lo melhor.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Escrevendo um teste

Cada novo recurso começa com um teste. O principal objetivo deste teste é focar nos requisitos e design de código antes de escrever o código. Um teste é uma forma de documentação executável e pode ser usado posteriormente para entender o que o código faz ou quais são as intenções por trás dele.

Neste ponto, estamos no estado vermelho, pois a execução dos testes falha. Há uma discrepância entre o que os testes esperam do código e o que o código de implementação realmente faz.
Para ser mais específico, não há código que atenda à expectativa do último teste; ainda não escrevemos. É possível que nesta fase todos os testes estejam realmente passando, mas isso é sinal de um problema.

Executando todos os testes e confirmando que o último está falhando

Confirmar que o último teste está falhando, confirma que o teste não passaria, por engano, sem a introdução de um novo código. Se o teste for aprovado, o recurso já existe ou o teste está produzindo um falso positivo. Se for esse o caso e o teste realmente sempre passar independentemente da implementação, ele é, por si só, inútil e deve ser removido.

Um teste não deve apenas falhar, mas deve falhar pelo motivo esperado. Nesta fase, ainda estamos na fase vermelha. Os testes foram executados e o último falhou.

Escrevendo o código de implementação

O objetivo desta fase é escrever o código que fará o último teste passar. Não tente torná-lo perfeito, nem tente gastar muito tempo com ele. Se não estiver bem escrito ou não for o ideal, tudo bem. Vai melhorar mais tarde. O que estamos realmente tentando fazer é criar uma rede de segurança na forma de testes que sejam aprovados. Não tente introduzir nenhuma funcionalidade que não tenha sido descrita no último teste. Para fazer isso, somos obrigados a voltar ao primeiro passo e começar com um novo teste. No entanto, não devemos escrever novos testes até que todos os existentes sejam aprovados.

Nesta fase, ainda estamos na fase vermelha. Embora o código que foi escrito provavelmente passaria em todos os testes, essa suposição ainda não foi confirmada.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Executando todos os testes

É muito importante que todos os testes sejam executados e não apenas o último teste que foi escrito. O código que acabamos de escrever pode ter passado no último teste enquanto quebrava outra coisa.
A execução de todos os testes confirma não apenas que a implementação do último teste está correta, mas também que não quebrou a integridade do aplicativo como um todo. Essa execução lenta de todo o conjunto de testes é um sinal de testes mal escritos ou de muito acoplamento no código. O acoplamento evita o fácil isolamento de dependências externas, aumentando assim o tempo necessário para a execução dos testes.

Nesta fase, estamos no estado verde. Todos os testes estão passando e o aplicativo se comporta como esperamos que ele se comporte.

Reestruturação

Embora todas as etapas anteriores sejam obrigatórias, esta é opcional. Embora a refatoração raramente seja feita no final de cada ciclo, mais cedo ou mais tarde ela será desejada, se não obrigatória. Nem toda implementação de um teste requer refatoração. Não existe uma regra que diga quando refatorar e quando não. O melhor momento é assim que se tem a sensação de que o código pode ser reescrito de uma maneira melhor ou mais otimizada.

O que constitui um candidato para refatoração? Esta é uma pergunta difícil de responder, pois pode ter muitas respostas - é difícil entender o código, a localização ilógica de um pedaço de código, duplicação, nomes que não declaram claramente um propósito, métodos longos, classes que fazem muitas coisas, e assim por diante. A lista pode continuar e continuar. Não importa quais sejam os motivos, a regra mais importante é que a refatoração não pode alterar nenhuma funcionalidade existente.

recorrente

Uma vez que todas as etapas (com a refatoração sendo opcional) são concluídas, nós as repetimos.
À primeira vista, todo o processo pode parecer muito longo ou muito complicado, mas não é. Praticantes experientes de TDD escrevem de uma a dez linhas de código antes de passar para a próxima etapa. Todo o ciclo deve durar entre alguns segundos e não mais do que alguns minutos. Se for mais do que isso, o escopo de um teste é muito grande e deve ser dividido em partes menores.
Seja rápido, falhe rápido, corrija e repita.

Com esse conhecimento em mente, vamos passar pelos requisitos do aplicativo que estamos prestes a desenvolver usando o processo Red-Green-Refactor.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Requisitos do jogo Tic-Tac-Toe

Tic-Tac-Toe é mais frequentemente jogado por crianças pequenas. As regras do jogo são bastante simples.

Tic-Tac-Toe é um jogo de papel e lápis para dois jogadores, *X* e O, que se revezam marcando os espaços em uma grade 3×3. O jogador que conseguir colocar três marcas respectivas em uma linha horizontal, vertical ou diagonal, ganha o jogo.

Para mais informações sobre o jogo, visite a Wikipedia (http://en.wikipedia.org/wiki/Tic-tac-toe).

Requisitos mais detalhados serão apresentados posteriormente.

O exercício consiste na criação de um único teste que corresponda a um dos requisitos. O teste é seguido pelo código que atende às expectativas desse teste.
Finalmente, se necessário, o código é refatorado. O mesmo procedimento deve ser repetido com mais testes relacionados ao mesmo requisito. Quando estivermos satisfeitos com os testes e a implementação desse requisito, passaremos para o próximo até que todos estejam prontos.

Em situações do mundo real, você não obteria requisitos tão detalhados, mas mergulharia direto em testes que atuariam como requisitos e validação. No entanto, até você se sentir confortável com o TDD, teremos que definir os requisitos separadamente dos testes.

Mesmo que todos os testes e a implementação sejam fornecidos, tente ler apenas um requisito por vez e escreva você mesmo os testes e o código de implementação. Uma vez feito, compare sua solução com a deste livro e passe para o próximo requisito.
Não há uma e apenas uma solução; o seu pode ser melhor do que os apresentados aqui.

Desenvolvendo o jogo da velha

Você está pronto para codificar? Vamos começar com o primeiro requisito.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Requisito 1 – colocação de peças

Devemos começar definindo os limites e o que constitui uma colocação inválida de uma peça.

Uma peça pode ser colocada em qualquer espaço vazio de um tabuleiro 3×3.

Podemos dividir esse requisito em três testes:

- Quando uma peça é colocada em qualquer lugar fora do eixo x, RuntimeException é lançada
- Quando uma peça é colocada em qualquer lugar fora do eixo y, RuntimeException é lançada
- Quando uma peça é colocada em um espaço ocupado, RuntimeException é lançada

Como você pode ver, os testes relacionados a esse primeiro requisito são sobre validações do argumento de entrada. Não há nada nos requisitos que diga o que deve ser feito com essas peças.

Antes de prosseguirmos com o primeiro teste, é necessária uma breve explicação de como testar exceções com JUnit.

A partir do Release 4.7, o JUnit introduziu um recurso chamado Rule. Ele pode ser usado para fazer muitas coisas diferentes (mais informações podem ser encontradas em https://github.com/junit-team/junit/wiki/Rules), mas no nosso caso estamos interessados na regra ExpectedException :

```
public class FootTest { @Rule public
    ExpectedException exception =
    ExpectedException.none(); @Teste

    public void whenDoFooThenThrowRuntimeException() { Foo foo = new Foo();
        exceção.expect(RuntimeException.class); foo.doFoo();


    }
}
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Neste exemplo, definimos que a ExpectedException é uma regra. Mais tarde, no teste doFooThrowsRuntimeException , especificamos que esperamos que a RuntimeException seja lançada depois que a classe Foo for instanciada. Se for lançado antes, o teste falhará. Se a exceção for lançada depois, o teste será bem-sucedido.

@Before pode ser usado para anotar um método que deve ser executado antes de cada teste. É um recurso muito útil com o qual podemos, por exemplo, instanciar uma classe usada em testes ou realizar outros tipos de ações que devem ser executadas antes de cada teste:

```
privado Foo foo;

@Antes da
public final void antes() { foo = new Foo();

}
```

Neste exemplo, a classe Foo será instanciada antes de cada teste. Dessa forma, podemos evitar ter código repetitivo que instanciaria Foo dentro de cada método de teste.

Cada teste deve ser anotado com @Test. Isso informa ao JunitRunner quais métodos constituem testes. Cada um deles será executado em ordem aleatória, portanto, certifique-se de que cada teste seja autossuficiente e não dependa do estado que possa ser criado por outros testes:

```
@Teste
public void whenSomethingThenResultIsSomethingElse() {
    // Este é um método de teste
}
```

Com esse conhecimento, você deve ser capaz de escrever seu primeiro teste e segui-lo com a implementação. Uma vez feito, compare-o com a solução fornecida.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Use nomes descritivos para métodos de teste.

Um dos benefícios é que ajuda a entender o objetivo dos testes.

O uso de nomes de métodos que descrevem testes é benéfico ao tentar descobrir por que alguns testes falharam ou quando a cobertura deve ser aumentada com mais testes. Deve ficar claro quais condições são estabelecidas antes do teste, quais ações são executadas e qual é o resultado esperado.

Há muitas maneiras diferentes de nomear métodos de teste. Meu método preferido é nomeá-los usando a sintaxe dada/quando/então usada em cenários de BDD. Dado descreve (pré) condições, Quando descreve ações e Então descreve o resultado esperado. Se um teste não tiver pré-condições (geralmente definidas usando as anotações @Before e @BeforeClass), Given pode ser ignorado.

Não confie apenas nos comentários para fornecer informações sobre os objetivos do teste. Os comentários não aparecem quando os testes são executados a partir do seu IDE favorito, nem aparecem nos relatórios gerados pelo CI ou pelas ferramentas de construção.

Além de escrever testes, você também precisará executá-los. Como estamos usando o Gradle, eles podem ser executados no prompt de comando:

```
$ teste gradle
```

O IntelliJ IDEA fornece um modelo de tarefas Gradle muito bom que pode ser acessado clicando em View|Tool Windows|Gradle. Ele lista todas as tarefas que podem ser executadas com Gradle (test sendo uma delas).

A escolha é sua - você pode executar testes da maneira que achar melhor, desde que execute todos eles.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Teste - limites da placa I

Devemos começar verificando se uma peça é colocada dentro dos limites do tabuleiro 3x3:

```
pacote com.packtpublishing.tddjava.ch03tictactoe;

import org.junit.Before; import
org.junit.Rule; importar org.junit.Test;
importar org.junit.rules.ExpectedException;


classe pública TicTacToeSpec {
    @Regra
    exceção pública ExpectedException = ExpectedException.none(); particular TicTacToe ticTacToe;


    @Antes da
    public final void antes() {
        ticTacToe = new TicTacToe();
    }
    @Teste
    public void whenXOutsideBoardThenRuntimeException() {
        exceção.expect(RuntimeException.class); ticTacToe.play(5, 2);

    }
}
```

Quando uma peça é colocada em qualquer lugar fora do eixo x, RuntimeException é lançada.

Neste teste, estamos definindo que RuntimeException é esperado quando o método ticTacToe.play(5, 2) é invocado. É um teste muito curto e fácil, e fazê-lo passar deve ser fácil também. Tudo o que precisamos fazer é criar o método play e garantir que ele lance RuntimeException quando o argumento x for menor que 1 ou maior que 3 (o tabuleiro é 3x3). Você deve executar este teste três vezes. Na primeira vez, deve falhar porque o método play não existe. Depois de adicionado, ele deve falhar porque RuntimeException não é lançado. Na terceira vez, deve ser bem-sucedido porque o código que corresponde a este teste está totalmente implementado.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação

Agora que temos uma definição clara de quando uma exceção deve ser lançada, a implementação deve ser direta:

```
pacote com.packtpublishing.tddjava.ch03tictactoe;

public class TicTacToe { public void
    play(int x, int y) { if (x < 1 || x > 3) {

            throw new RuntimeException("X está fora da placa");
        }
    }
}
```

Como você pode ver, esse código não contém mais nada, mas o mínimo necessário para que o teste seja aprovado.

Alguns praticantes de TDD tendem a tomar mínimo como um significado literal. Eles teriam o método play com apenas o throw new RuntimeException(); linha. Eu tendo a traduzir o mínimo para o mínimo possível dentro da razão.

Não estamos somando números, nem retornando nada. Trata-se de fazer pequenas mudanças muito rapidamente. (Lembra-se do jogo de pingue-pongue?) Por enquanto, estamos fazendo passos vermelho-verde. Não há muito que possamos fazer para melhorar esse código, então estamos pulando a refatoração.

Vamos para o próximo teste.

Teste - limites da placa II

Este teste é quase o mesmo que o anterior. Desta vez, devemos validar o eixo y:

```
@Teste
public void whenYOutsideBoardThenRuntimeException() {
    exceção.expect(RuntimeException.class); ticTacToe.play(2, 5);

}
```

Quando uma peça é colocada em qualquer lugar fora do eixo y, RuntimeException é lançada.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação A

implementação desta especificação é quase a mesma da anterior. Tudo o que precisamos fazer é lançar uma exceção se y não estiver dentro do intervalo definido:

```
public void play(int x, int y) { if (x < 1 || x > 3) {

        throw new RuntimeException("X está fora da placa"); } else if (y < 1 || y > 3) { throw
    new RuntimeException("Y está fora da placa");

    }
}
```

Para que o último teste passasse, tivemos que adicionar a cláusula else que verifica se Y está dentro da placa.

Vamos fazer o último teste para este requisito.

Teste - local ocupado

Agora que sabemos que as peças são colocadas dentro dos limites do tabuleiro, devemos nos certificar de que elas podem ser colocadas apenas em espaços desocupados:

```
@Teste
public void whenOccupiedThenRuntimeException() { ticTacToe.play(2, 1);
    exceção.expect(RuntimeException.class); ticTacToe.play(2, 1);


}
```

Quando uma peça é colocada em um espaço ocupado, RuntimeException é lançada.

É isso; este foi o nosso último teste. Uma vez finalizada a implementação, podemos considerar o primeiro requisito como concluído.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação Para

implementar o último teste, devemos armazenar a localização das peças colocadas em um array. Cada vez que uma nova peça é colocada, devemos verificar se o lugar não está ocupado, ou então lançar uma exceção:

```
Caractere privado[][] board = { {'\0', '\0', '\0'}, {'\0',
    '\0', '\0'}, {'\0', ' \0', '\0'}


};

public void play(int x, int y) { if (x < 1 || x > 3) {

        throw new RuntimeException("X está fora da placa"); } else if (y < 1 || y > 3)
    { throw new RuntimeException("Y está fora da placa");


    } if (board[x - 1][y - 1] != '\0') { throw new
        RuntimeException("Caixa está ocupada"); } else { placa[x - 1][y - 1] = 'X';


    }
}
```

Estamos verificando se um lugar que foi reproduzido está ocupado e, se não estiver, estamos alterando o valor da entrada do array de vazio (\0) para ocupado (X). Tenha em mente que ainda não estamos armazenando quem jogou (X ou O).

Reestruturação

Embora o código que fizemos até agora cumpra os requisitos definidos pelos testes, parece um pouco confuso. Se alguém o lesse, não ficaria claro o que o método de jogo faz.
Devemos refatorá-lo movendo o código em métodos separados. O código refatorado terá a seguinte aparência:

```
public void play(int x, int y) { checkAxis(x);
    checkAxis(y); setBox(x,y);


}

private void checkAxis(int axis) { if (axis < 1 || axis > 3)
    { throw new RuntimeException("X está fora da
        placa");
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google


```
        }
    }

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


        }
    }
```

Com essa refatoração, não alteramos a funcionalidade do método play . Ele se comporta exatamente da mesma forma que antes, mas o novo código é um pouco mais legível. Como tínhamos testes que cobriam todas as funcionalidades existentes, não havia medo de que pudéssemos fazer algo errado. Contanto que todos os testes estejam passando o tempo todo e a refatoração não tenha introduzido nenhum novo comportamento, é seguro fazer alterações no código.

O código-fonte pode ser encontrado na ramificação 01-exceptions do repositório Git tdd-java-ch03-tic tac-toe em https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch /01-exceções.

Requisito 2 – adicionar suporte para dois jogadores

Agora é hora de trabalhar na especificação de qual jogador está prestes a jogar sua vez.

Deve haver uma maneira de descobrir qual jogador deve jogar em seguida.

Podemos dividir esse requisito em três testes:

- O primeiro turno deve ser jogado pelo jogador X
- Se o último turno foi jogado por X, então o próximo turno deve ser jogado por O
- Se o último turno foi jogado por O, então o próximo turno deve ser jogado por X

Até este momento, não usamos nenhuma das assertivas do JUnit. Para usá-los, precisamos importar os métodos estáticos da classe org.junit.Assert :

```
import static org.junit.Assert.*;
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Em sua essência, os métodos dentro da classe Assert são muito simples. A maioria deles começa com assert. Por exemplo, assertEquals compara dois objetos — assertNotEquals verifica se dois objetos não são iguais e assertArrayEquals verifica se duas matrizes são iguais. Cada uma dessas asserções tem muitas variações sobrecarregadas para que quase qualquer tipo de objeto Java possa ser usado.

No nosso caso, precisaremos comparar dois caracteres. O primeiro é o que estamos esperando e o segundo é o personagem real recuperado do método nextPlayer .

Agora é hora de escrever esses testes e a implementação.

Escreva o teste antes de escrever o código de implementação.

Os benefícios de fazer isso são os seguintes: garante que o código testável seja escrito e garante que cada linha de código receba testes escritos para ele.

Ao escrever ou modificar o teste primeiro, o desenvolvedor se concentra nos requisitos antes de começar a trabalhar em um código. Esta é a principal diferença quando comparado à escrita de testes após a implementação ser feita. Um benefício adicional é que, com testes em primeiro lugar, evitamos o perigo de que os testes funcionem como verificação de qualidade em vez de garantia de qualidade.

Teste – X joga primeiro

O jogador X tem o primeiro turno:

```
@Teste
public void dadoFirstTurnWhenNextPlayerThenX() { assertEquals('X',
    ticTacToe.nextPlayer());
}
```

O primeiro turno deve ser jogado pelo Jogador X.

Este teste deve ser autoexplicativo. Esperamos que o método nextPlayer retorne X.
Se você tentar executar isso, verá que o código nem compila. Isso porque o método nextPlayer nem existe.
Nosso trabalho é escrever o método nextPlayer e garantir que ele retorne o valor correto.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação Não há

necessidade real de verificar se é realmente o primeiro turno do jogador ou não. Tal como está, este teste pode ser realizado sempre retornando X. Testes posteriores nos forçarão a refinar este código:

```
public char nextPlayer() { return 'X';

}
```

Teste – O joga logo após o X Agora, devemos ter

certeza de que os jogadores estão mudando. Depois que X terminar, deve ser a vez de O, depois novamente X, e assim por diante:

```
@Teste
public void dadoLastTurnWasXWhenNextPlayerThenO() { ticTacToe.play(1, 1);
    assertEquals('O', ticTacToe.nextPlayer());

}
```

Se o último turno foi jogado por X, então o próximo turno deve ser jogado por O.

Implementação

Para rastrear quem deve jogar em seguida, precisamos armazenar quem jogou por último:

```
caractere privado lastPlayer = '\0';

public void play(int x, int y) { checkAxis(x); checkAxis(y);
    setBox(x,y); últimoJogador = próximoJogador();



}

public char nextPlayer() { if (lastPlayer ==
    'X') {
        retornar 'O';

    } return 'X';
}
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Você provavelmente está começando a pegar o jeito. Os testes são pequenos e fáceis de escrever. Com experiência suficiente, deve levar um minuto, se não segundos, para escrever um teste e tanto tempo ou menos para escrever a implementação.

Teste – X joga logo após O Finalmente, podemos

verificar se a vez de X vem depois de O jogar.

Se o último turno foi jogado por O, então o próximo turno deve ser jogado por X.

Não há nada a fazer para cumprir este teste e, portanto, o teste é inútil e deve ser descartado. Se você escrever este teste, descobrirá que é um falso positivo. Passaria sem alterar a implementação; Experimente. Escreva este teste e se for bem sucedido sem escrever nenhum código de implementação, descarte-o.

O código-fonte pode ser encontrado na ramificação 02-next-player do repositório Git tdd-java-ch03-tic tac-toe em https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe /branch/02-next-player.

Requisito 3 – adicionar condições vencedoras

É hora de trabalhar para vencer de acordo com as regras do jogo. Esta é a parte onde, quando comparado com o código anterior, o trabalho se torna um pouco mais tedioso. Devemos verificar todas as combinações vencedoras possíveis e, se uma delas for cumprida, declarar um vencedor.

Um jogador ganha sendo o primeiro a conectar uma linha de peças amigas de um lado ou canto do tabuleiro ao outro.

Para verificar se uma linha de peças amigas está conectada, devemos verificar as linhas horizontais, verticais e diagonais.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Teste - por padrão, não há vencedor

Vamos começar definindo a resposta padrão do método play :

```
@Test
public void whenPlayThenNoWinner() {
    String real = ticTacToe.play(1,1); assertEquals("Nenhum
    vencedor", real);
}
```

Se nenhuma condição de vitória for cumprida, então não há vencedor.

Implementação

Os valores de retorno padrão são sempre mais fáceis de implementar e este não é exceção:

```
public String play(int x, int y) { checkAxis(x); checkAxis(y);
    setBox(x,y); últimoJogador = próximoJogador(); return
    "Nenhum vencedor";



}
```

Teste – condição vencedora I

Agora que declaramos qual é a resposta padrão (Sem vencedor), é hora de começar a trabalhar em diferentes condições de vitória:

```
@Teste
public void whenPlayAndWholeHorizontalLineThenWinner() { ticTacToe.play(1, 1); // X
    ticTacToe.play(1, 2); // O ticTacToe.play(2, 1); // X ticTacToe.play(2, 2); // O String real =
    ticTacToe.play(3, 1); // X assertEquals("X é o vencedor", real);



}
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

O jogador ganha quando toda a linha horizontal estiver ocupada pelas suas peças.

Implementação Para cumprir

este teste, precisamos verificar se alguma linha horizontal é preenchida pela mesma marca que o jogador atual. Até este momento, não nos importamos com o que foi colocado no array da placa. Agora, precisamos apresentar não apenas quais caixas de tabuleiro estão vazias, mas também qual jogador as jogou:

```
public String play(int x, int y) { checkAxis(x); checkAxis(y);
    últimoJogador = próximoJogador(); setBox(x, y,
    últimoJogador); for (int index = 0; index < 3; index++)
    { if (board[0][index] == lastPlayer



                && board[1][index] == lastPlayer && board[2][index]
                == lastPlayer) { return lastPlayer + " é o vencedor";

        }
    }
    return "Nenhum vencedor";

} 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;


    }
}
```

Refatoração O

código anterior satisfaz os testes, mas não é necessariamente a versão final. Ele serviu ao propósito de obter cobertura de código o mais rápido possível. Agora, como temos testes que garantem a integridade do comportamento esperado, podemos refatorar o código:

```
private static final int SIZE = 3;

public String play(int x, int y) { checkAxis(x);
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google


```
        checkAxis(y);
        últimoJogador = próximoJogador(); setBox(x,
        y, últimoJogador); if (isWin()) { return
        lastPlayer +
                                                "é o vencedor";

        } return "Nenhum vencedor";
    }

    private boolean isWin() { for (int i = 0; i <
        SIZE; i++) { if (board[0][i] + board[1][i] + board[2][i] ==
            (lastPlayer *
    TAMANHO))
                { return true;
            }

        } retorna falso;
    }
```

Esta solução refatorada parece melhor. O método de jogo continua sendo curto e fácil de entender. A lógica vencedora é movida para um método separado. Não apenas mantivemos claro o propósito do método de jogo , mas essa separação também nos permite aumentar o código da condição vencedora em separação do resto.

Teste – condição vencedora II

Também devemos verificar se há uma vitória preenchendo a linha vertical:

```
@Test
public void whenPlayAndWholeVerticalLineThenWinner() {
    ticTacToe.play(2, 1); // X ticTacToe.play(1,
    1); // O ticTacToe.play(3, 1); // X
    ticTacToe.play(1, 2); // O ticTacToe.play(2,
    2); // X String real = ticTacToe.play(1, 3); // O
    assertEquals("O é o vencedor", real);


}
```

O jogador ganha quando toda a linha vertical estiver ocupada pelas suas peças.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação Esta implementação deve ser semelhante à anterior. Já temos a verificação horizontal e agora precisamos fazer o mesmo na vertical:

```
private boolean isWin() { int playerTotal =
    lastPlayer * 3; for (int i = 0; i < TAMANHO; i++) {

        if (tabuleiro[0][i] + tabuleiro[1][i] + tabuleiro[2][i] == jogadorTotal) {
            retorne verdadeiro; }
        else if (placa[i][0] + placa[i][1] + placa[i][2] ==
jogadorTotal) {
            retorne verdadeiro;
        }

    } retorna falso;
}
```

Teste – condição vencedora III

Agora que as linhas horizontais e verticais estão cobertas, devemos nos concentrar nas combinações diagonais:

```
@Teste
public void whenPlayAndTopBottomDiagonalLineThenWinner() { ticTacToe.play(1, 1); // X
    ticTacToe.play(1, 2); // O ticTacToe.play(2, 2); // X ticTacToe.play(1, 3); // O String real =
    ticTacToe.play(3, 3); // X assertEquals("X é o vencedor", real);




}
```

O jogador vence quando toda a linha diagonal do canto superior esquerdo ao canto inferior direito estiver ocupada por suas peças.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação

Como há apenas uma linha que pode constituir com o requisito, podemos verificá-la diretamente sem nenhum loop:

```
private boolean isWin() { int playerTotal =
    lastPlayer * 3; for (int i = 0; i < TAMANHO; i++) {

        if (tabuleiro[0][i] + tabuleiro[1][i] + tabuleiro[2][i] == jogadorTotal) {
            retorne verdadeiro; }
        else if (placa[i][0] + placa[i][1] + placa[i][2] ==
jogadorTotal) {
            retorne verdadeiro;
        }

    } if (tabuleiro[0][0] + tabuleiro[1][1] + tabuleiro[2][2] == jogadorTotal) {
        retorne verdadeiro;

    } retorna falso;
}
```

Teste - condição vencedora IV

Finalmente, há a última condição de vitória possível para resolver:

```
@Teste
public void whenPlayAndBottomTopDiagonalLineThenWinner() { ticTacToe.play(1, 3); // X
    ticTacToe.play(1, 1); // O ticTacToe.play(2, 2); // X ticTacToe.play(1, 2); // O String real =
    ticTacToe.play(3, 1); // X assertEquals("X é o vencedor", real);




}
```

O jogador ganha quando toda a linha diagonal do canto inferior esquerdo ao canto superior direito é ocupada por suas peças.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação

A implementação deste teste deve ser quase a mesma do anterior:

```
private boolean isWin() { int playerTotal =
    lastPlayer * 3; for (int i = 0; i < SIZE; i++) { if (tabuleiro[0]
    [i] + tabuleiro[1][i] + tabuleiro[2][i] == jogadorTotal) {

            retorne verdadeiro; }
        else if (placa[i][0] + placa[i][1] + placa[i][2] ==
jogadorTotal) {
            retorne verdadeiro;
        }

    } if (tabuleiro[0][0] + tabuleiro[1][1] + tabuleiro[2][2] == jogadorTotal) {
        retorne verdadeiro; }
     else if (placa[0][2] + placa[1][1] + placa[2][0] ==
jogadorTotal) { return
        true;

    } retorna falso;
}
```

Refatorando Da

forma como estamos lidando com possíveis vitórias diagonais, o cálculo não parece correto. Talvez a reutilização do loop existente fizesse mais sentido:

```
private boolean isWin() { int playerTotal =
    lastPlayer * 3; char diagonal1 = '\0'; char diagonal2 =
    '\0'; for (int i = 0; i < SIZE; i++) { diagonal1 += board[i][i];
    diagonal2 += tabuleiro[i][SIZE - i - 1]; if (tabuleiro[0][i] +
    tabuleiro[1][i] + tabuleiro[2][i]) == jogadorTotal) {



            retorne verdadeiro; }
        else if (placa[i][0] + placa[i][1] + placa[i][2] ==
jogadorTotal) {
            retorne verdadeiro;
        }

    } if (diagonal1 == jogadorTotal || diagonal2 == jogadorTotal) {
        retorne verdadeiro;

    } retorna falso;
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google


```
}
```

O código-fonte pode ser encontrado na ramificação 03-wins do repositório Git tdd-java-ch03-tic-tac-toe em https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/ filial/03-ganhas.

Agora, vamos passar pelo último requisito.

Requisito 4 - condições de empate

A única coisa que falta é como lidar com o resultado do empate.

O resultado é um empate quando todas as caixas estiverem preenchidas.

Teste – lidando com uma situação de empate

Podemos testar o resultado do sorteio preenchendo todas as caixas do tabuleiro:

```
@Teste
public void whenAllBoxesAreFilledThenDraw() { ticTacToe.play(1, 1);
    ticTacToe.play(1, 2); ticTacToe.play(1, 3); ticTacToe.play(2, 1);
    ticTacToe.play(2, 3); ticTacToe.play(2, 2); ticTacToe.play(3, 1);
    ticTacToe.play(3, 3); String real = ticTacToe.play(3, 2); assertEquals("O
    resultado é empate", real);
```

```
}
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Implementação Verificar

se é um empate é bastante simples. Tudo o que temos a fazer é verificar se todas as caixas do tabuleiro estão preenchidas. Podemos fazer isso iterando pelo array da placa:

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

        lastPlayer + } else if (isDraw()) { return "O resultado é "O vencedor é"; return
     empate"; } else { return "Nenhum vencedor";

    }
}

private boolean isDraw() { for (int x = 0; x <
    SIZE; x++) {
        for (int y = 0; y < SIZE; y++) { if (board[x][y] == '\0') {
                retorna falso;
            }
        }
    }
    retorne verdadeiro;
}
```

Reestruturação

Mesmo que o método isWin não seja o escopo do último teste, ele ainda pode ser refatorado ainda mais. Por uma vez, não precisamos verificar todas as combinações, mas apenas aquelas relacionadas à posição da última peça tocada. A versão final pode ter a seguinte aparência:

```
private boolean isWin(int x, int y) { int playerTotal = lastPlayer
    * 3; char horizontal, vertical, diagonal1, diagonal2; horizontal
    = vertical = diagonal1 = diagonal2 = '\0'; for (int i = 0; i < SIZE; i++) { horizontal
    += placa[i][y - 1]; vertical += placa[x - 1][i]; diagonal1 += tabuleiro[i][i]; diagonal2 +=
    tabuleiro[i][SIZE - i - 1];

    }
```


Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google


```
        if (horizontal == jogadorTotal
                || vertical == jogadorTotal || diagonal1 ==
                jogadorTotal || diagonal2 == jogadorTotal) {

            retorne verdadeiro;

        } retorna falso;
    }
```

A refatoração pode ser feita em qualquer parte do código a qualquer momento, desde que todos os testes sejam bem-sucedidos. Embora muitas vezes seja mais fácil e rápido refatorar o código que acabou de ser escrito, voltar para algo que foi escrito no outro dia, mês anterior ou mesmo anos atrás é mais do que bem-vindo. O melhor momento para refatorar algo é quando alguém vê uma oportunidade de torná-lo melhor. Não importa quem escreveu ou quando; melhorar o código é sempre uma boa coisa a se fazer.

O código-fonte pode ser encontrado na ramificação 04-draw do repositório Git tdd-java-ch03-tic-tac-toe em https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/ filial/04-sorteio.

Cobertura de código

Não usamos ferramentas de cobertura de código ao longo deste exercício. A razão é que queríamos que você se concentrasse no modelo Red-Green-Refactor. Você escreveu um teste, viu ele falhar, escreveu o código de implementação, viu que todos os testes foram executados com sucesso, refatorou o código sempre que viu uma oportunidade de torná-lo melhor e então repetiu o processo. Nossos testes cobriram todos os casos? Isso é algo que ferramentas de cobertura de código como JaCoCo podem responder. Você deve usar essas ferramentas? Provavelmente, apenas no início. Deixe-me esclarecer isso.
Quando você está começando com o TDD, provavelmente vai perder alguns testes ou implementar mais do que os testes definiram. Nesses casos, usar a cobertura de código é uma boa maneira de aprender com seus próprios erros. Mais tarde, quanto mais experiente você se tornar com o TDD, menor será a necessidade de tais ferramentas. Você escreverá testes e apenas o suficiente do código para fazê-los passar. Sua cobertura será alta com ou sem ferramentas como JaCoCo. Haverá uma pequena quantidade de código não coberta por testes porque você tomará uma decisão consciente sobre o que não vale a pena testar.

Ferramentas como JaCoCo foram projetadas principalmente como forma de verificar se os testes escritos após o código de implementação estão fornecendo cobertura suficiente. Com o TDD, estamos adotando uma abordagem diferente com a ordem invertida (testes antes da implementação).

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Ainda assim, sugerimos que você use o JaCoCo como uma ferramenta de aprendizado e decida por si mesmo se deve usá-lo no futuro.

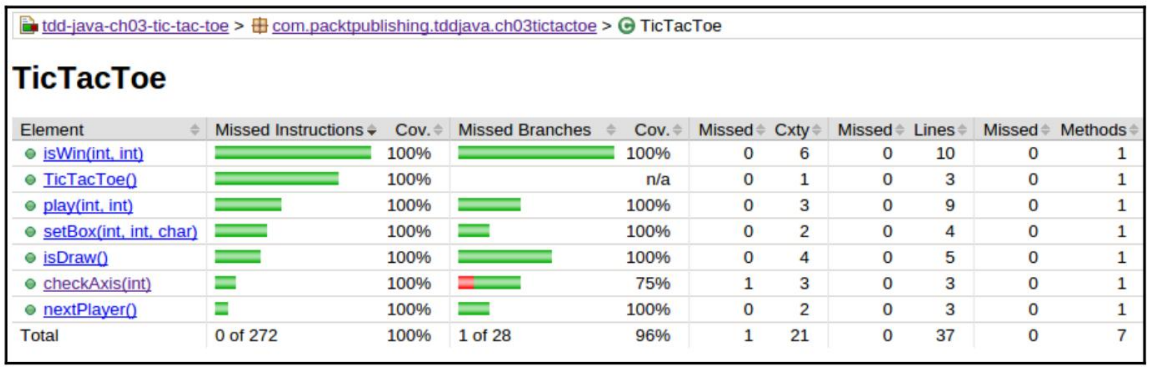
Para habilitar o JaCoCo no Gradle, adicione o seguinte ao build.gradle:

```
aplicar plugin: 'jacoco'
```

A partir de agora, o Gradle coletará as métricas do JaCoCo toda vez que executarmos os testes. Essas métricas podem ser transformadas em um bom relatório usando o destino jacocoTestReport Gradle. Vamos executar nossos testes novamente e ver qual é a cobertura do código:

```
$ gradle clean test jacocoTestReport
```

O resultado final é o relatório localizado no diretório build/reports/jacoco/test/html .
Os resultados irão variar dependendo da solução que você fez para este exercício. Meus resultados dizem que há 100% de cobertura de instruções e 96% de cobertura de agências; 4% está faltando porque não houve caso de teste onde o jogador jogou em uma caixa 0 ou negativa. A implementação desse caso está lá, mas não há nenhum teste específico que cubra isso. No geral, esta é uma cobertura muito boa:



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isWin(int, int)		100%		100%	0	6	0	10	0	1
TicTacToe()		100%		n/a	0	1	0	3	0	1
play(int, int)		100%		100%	0	3	0	9	0	1
setBox(int, int, char)		100%		100%	0	2	0	4	0	1
isDraw()		100%		100%	0	4	0	5	0	1
checkAxis(int)		100%		75%	1	3	0	3	0	1
nextPlayer()		100%		100%	0	2	0	3	0	1
Total	0 of 272	100%	1 of 28	96%	1	21	0	37	0	7

JaCoCo será adicionado no código fonte. Isso é encontrado na ramificação 05-jacoco do repositório Git tdd-java-ch03-tic-tac-toe em https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/05 -jacoco.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.

Machine Translated by Google

Mais exercícios

Acabamos de desenvolver uma variação (mais comumente usada) do jogo Tic-Tac-Toe. Como exercício adicional, escolha uma ou mais variações da Wikipedia (http://en.wikipedia.org/wiki/Tic-tac-toe) e implementá-lo usando o procedimento Red-Green Refactor. Quando terminar, implemente uma espécie de IA que jogaria os turnos de O. Como o Tic-Tac-Toe geralmente leva a um empate, a IA pode ser considerada concluída quando atinge com sucesso um empate para qualquer combinação de movimentos de X.

Enquanto estiver trabalhando nesses exercícios, lembre-se de ser rápido e jogar pingue-pongue. Além disso, acima de tudo, lembre-se de usar o procedimento Red-Green-Refactor.

Resumo

Conseguimos terminar nosso jogo Tic-Tac-Toe usando o processo Red-Green-Refactor. Os exemplos em si eram simples e você provavelmente não teve problemas em segui-los.

O objetivo deste capítulo não foi mergulhar em algo complicado (que vem depois), mas adquirir o hábito de usar o ciclo curto e repetitivo chamado Red-Green-Refactor.

Aprendemos que a maneira mais fácil de desenvolver algo é dividindo-o em pedaços muito pequenos. O design estava surgindo de testes em vez de usar uma grande abordagem inicial. Nenhuma linha do código de implementação foi escrita sem antes escrever um teste e vê-lo falhar. Ao confirmar que o último teste falhou, estamos confirmando que ele é válido (é fácil cometer um erro e escrever um teste que sempre dá certo) e o recurso que estamos prestes a implementar não existe. Depois que o teste falhou, escrevemos a implementação desse teste.
Enquanto escrevíamos a implementação, tentamos torná-la mínima com o objetivo de fazer o teste passar, não de tornar a solução final. Repetimos esse processo até sentirmos que havia a necessidade de refatorar o código. A refatoração não introduziu nenhuma nova funcionalidade (não alteramos o que o aplicativo faz), mas tornou o código mais otimizado e mais fácil de ler e manter.

No próximo capítulo, detalharemos mais sobre o que constitui uma unidade no contexto do TDD e como abordar a criação de testes baseados nessas unidades.

Farcic, V., & Garcia, A. (2018). Desenvolvimento java orientado a testes, segunda edição: Invoque os princípios tdd para desenvolvimento de aplicativos de ponta a ponta, 2ª edição. Packt Publishing, Limited.
Criado a partir de univbrasilia-ebooks em 2022-08-11 00:50:11.