# Solução Kata – TDD : Boliche

## Esclarecimento

Esse documento mostra uma possível solução para o Kata. Não significando que é a única solução. Times e experiências diferentes podem produzir soluções funcionais diferentes.

## O que você irá precisar

É necessário ter instalado, no seu Visual Studio 2013, o plugin **NUnit Test Adapter**. (Menu **Tools** / **Extensions and Updates** / Opção **Online** / Campo de busca: **NUnit Test Adapter**) Após instalar, reinicie o Visual Studio 2013 para que a instalação do plugin seja efetivada.

## Iniciando

* Crie um projeto chamado **JogoBoliche**
* Crie um projeto, para os testes unitários, chamado **JogoBolicheUnitTest**
* No projeto **JogoBolicheUnitTest**, instale, utilizando **NuGet**, o pacote **NUnit**.

## Testes, Testes, Testes

Inicialmente, vamos escrever o primeiro teste da seguinte forma:

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

}

}

Compile o projeto e observe que na janela **Test Explorer** não existem testes para serem executados. Isso é normal, afinal, ainda não criamos nenhum testes, apenas definimos a classe que irá conter nossos testes.

## Primeiro Teste: Canaleta

O primeiro teste que iremos escrever nos dirá que se todas as bolas forem jogadas na canaleta, não pode ser marcado nenhum ponto.

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

var jogo = new Jogo();

}

}

}

Se compilarmos esse trecho de código, receberemos um erro dizendo que a classe Jogo não foi definida. Então, criamos a classe Jogo no projeto **JogoBoliche**.

namespace JogoBoliche.Core {

public class Jogo {

}

}

E referenciamos o projeto **JogoBoliche** no projeto **JogoBolicheUnitTest**. Compile novamente e veja que não existe mais o erro.

Continuando, para que um jogo seja todo na canaleta, temos que arremessar a bola 20 vezes (duas por Frame) e a pontuação deve ser 0 (zero).

using JogoBoliche.Core;

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

var jogo = new Jogo();

// act

for (var arremesso = 0; arremesso < 20; arremesso++) {

jogo.Arremesso(0);

}

}

}

}

Compilando, novamente vemos o erro que o método “Arremesso” não está definido, vamos defini-lo.

namespace JogoBoliche.Core {

public class Jogo {

public void Arremesso(int pinos) {

}

}

}

Se compilar o teste novamente, o erro irá desaparecer. Vamos continuar e solicitar a pontuação, ela deve ser 0 (zero).

using JogoBoliche.Core;

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

var jogo = new Jogo();

// act

for (var arremesso = 0; arremesso < 20; arremesso++) {

jogo.Arremesso(0);

}

// assert

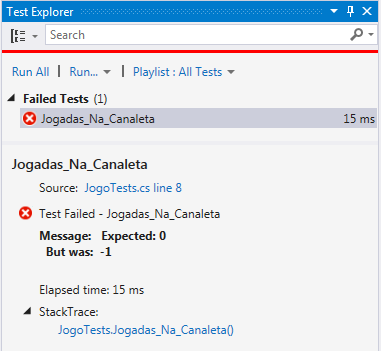
Assert.AreEqual(0, jogo.Pontuacao());

}

}

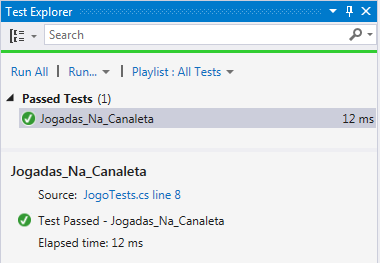
}

Defina o método “Pontuacao” na classe “Jogo”, faça com que ele retorne o valor -1 e compile o teste, tudo irá compilar corretamente. Agora, a janela Test Explorer, observe que existe uma nova entrada, que é “Jogadas\_Na\_Canaleta”. Clique em **Run All**, para rodar todos os testes e observe o resultado.



Veja que a janela Test Explorer mostra quais o testes que foram executados e, quando um teste dá errado, ele marca o mesmo com um “x” vermelho. Também, quando você seleciona o teste que deu erro, ele irá mostrar o motivo. Aqui podemos ver que era esperado que o resultado da execução do método retornasse 0 (zero), porém o resultado foi -1.

Altere o retorno do método “Pontuacao” para zero e execute o teste novamente.



## Segundo Teste: Todos 1’s

Esse teste tem como objetivo verificar se todos os arremessos forem executados, derrubando apenas 1 pino, o resultado final deve ser 20. Extrapolando esse teste, poderíamos dizer que se em todas as jogadas 2 pinos fossem derrubados, então o resultado final seria 40.

[Test]

public void Todos\_Arremessos\_Apenas\_Um\_Pino() {

// arrange

var jogo = new Jogo();

// act

for (var arremesso = 0; arremesso < 20; arremesso++) {

jogo.Arremesso(1);

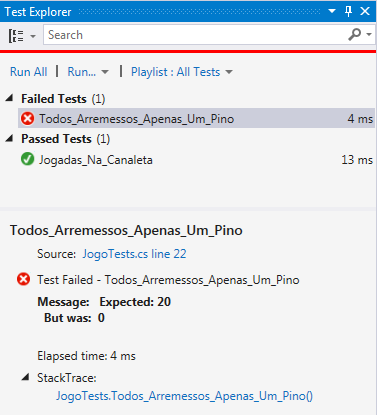
}

// assert

Assert.AreEqual(20, jogo.Pontuacao());

}

Rodar esse teste resultará em erro.



Isso não é surpresa, afinal, a implementação do método “Pontuacao” sempre irá retornar 0 (zero). Então, agora vamos implementá-lo de forma que a pontuação se acumule, dado os arremessos.

namespace JogoBoliche.Core {

public class Jogo {

private int \_pontuacao;

public void Arremesso(int pinos) {

\_pontuacao += pinos;

}

public int Pontuacao() {

return \_pontuacao;

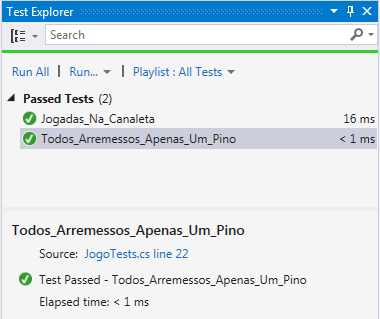
}

}

}

Observe que implementamos somente o necessário para que o teste seja executado corretamente, esse é o espírito do TDD.

A execução do teste:



## Considerações Sobre os Testes

Antes de continuarmos, vamos ver algumas coisas que escrevemos até o momento em nosso teste.

using JogoBoliche.Core;

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

var jogo = new Jogo();

// act

for (var arremesso = 0; arremesso < 20; arremesso++) {

jogo.Arremesso(0);

}

// assert

Assert.AreEqual(0, jogo.Pontuacao());

}

[Test]

public void Todos\_Arremessos\_Apenas\_Um\_Pino() {

// arrange

var jogo = new Jogo();

// act

for (var arremesso = 0; arremesso < 20; arremesso++) {

jogo.Arremesso(1);

}

// assert

Assert.AreEqual(20, jogo.Pontuacao());

}

}

}

Podemos observar, primeiramente, que existe uma duplicação na declaração da instância da classe “Jogo”. Uma no teste “Jogadas\_Na\_Canaleta” e outra no teste “Todos\_Arremessos\_Apenas\_Um\_Pino”. Então, para remover essa duplicação, faremos um refactor no teste. Mas, vamos notar também, que existe duplicação no “for” que executa os arremessos. Então, vamos fazer algumas alterações.

### SetUpAttribute e TearDownAttribute

No framework de teste que estamos utilizando (**NUnit**) existem dois atributos que, adicionados a um método, executam ações antes e depois de um método de teste. Por exemplo, imagine que antes de executar um teste você deseja exibir, no console, a mensagem “Iniciando Teste” e, após a finalização do teste, exiba a mensagem “Teste Finalizado”. Isso pode ser alcançado utilizando os atributos mencionados.

Também existem, também, dois outros atributos, **SetUpFixtureAttribute** e **TearDownFixtureAttribute**, que sua ideia é análoga, porém, a execução é antes de todos os testes serem iniciados e após todos os testes serem finalizados.

### Refactor do Teste

using JogoBoliche.Core;

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

private Jogo \_jogo;

[SetUp]

public void SetUp() {

\_jogo = new Jogo();

}

[TearDown]

public void TearDown() {

\_jogo = null;

}

private void ExecutarArremessos(int vezes, int pinos) {

for (int arremesso = 0; arremesso < vezes; arremesso++) {

\_jogo.Arremesso(pinos);

}

}

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

// Feito pelo método SetUp

// act

ExecutarArremessos(20, 0);

// assert

Assert.AreEqual(0, \_jogo.Pontuacao());

}

[Test]

public void Todos\_Arremessos\_Apenas\_Um\_Pino() {

// arrange

// Feito pelo método SetUp

// act

ExecutarArremessos(20, 1);

// assert

Assert.AreEqual(20, \_jogo.Pontuacao());

}

}

}

Execute todos os testes novamente e verifique que ainda todos estão OK.

## Terceiro Teste: Arremesso “Spare”

Agora, iremos fazer o teste que indicará que o jogador executou um “Spare”, ou seja, seu primeiro arremesso não terminou em “Strike” (derrubou todos os pinos), porém o segundo arremesso derrubou todos os pinos que ficaram em pé. O “Spare” tem uma pontuação diferenciada. O jogador que alcança um “Spare” tem sua pontuação calculada da seguinte forma: 10 pontos + um bônus que é dado pelo número de pontos da próxima jogada. Ou seja, se ele acertar um “Spare” em um “Frame”, e no próximo “Frame”, no primeiro arremesso ele acertar 4 pinos, sua pontuação de “Spare” será 10 + 4 = 14.

Vamos tentar escrever o teste primeiro:

[Test]

public void Alcancando\_Um\_Spare() {

// arrange

// Feito pelo método SetUp

// act

\_jogo.Arremesso(5);

\_jogo.Arremesso(5);

\_jogo.Arremesso(3);

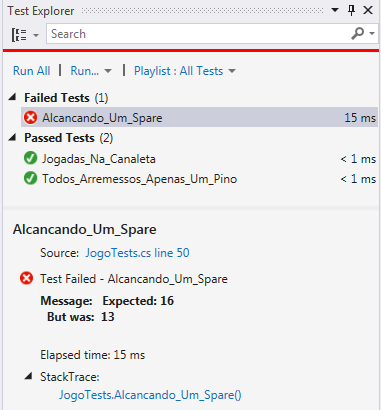
ExecutarArremessos(17, 0); // Todos os próximos arremessos foram na canaleta. =(

// assert

Assert.AreEqual(16, \_jogo.Pontuacao());

}

Esse teste resultará em erro:



Vejamos nosso código:

namespace JogoBoliche.Core {

Arremesso está calculando a pontuação, porém seu nome não implica isso.

public class Jogo {

private int \_pontuacao;

public void Arremesso(int pinos) {

\_pontuacao += pinos;

Pontuacao não está calculando a pontuação, porém seu nome implica isso.

}

public int Pontuacao() {

return \_pontuacao;

}

}

}

A tentativa de utilizar a variável global “\_pontuacao” parece não ter surtido o efeito desejado. O motivo é que não estamos verificando nada, apenas adicionado a quantidade de pinos derrubados. Precisamos adicionar uma lógica para se calcular o “Spare”. Também precisamos corrigir nosso design, que não parece estar fazendo o que deveria. As responsabilidades estão trocadas nos métodos que definimos.

Comente o código do teste “Alcancando\_Um\_Spare” e refatore a classe “Jogo” para que ela comporte o design especificado.

namespace JogoBoliche.Core {

public class Jogo {

private int \_pontuacao;

private int[] \_arremessos = new int[21]; // Todos os arremessos, e um possível 3º arremesso no 10º frame

private int \_arremessoAtual = 0;

public void Arremesso(int pinos) {

\_arremessos[\_arremessoAtual++] = pinos;

}

public int Pontuacao() {

var pontuacao = 0;

for (var idxArremesso = 0; idxArremesso < \_arremessos.Length; idxArremesso++) {

pontuacao += \_arremessos[idxArremesso];

}

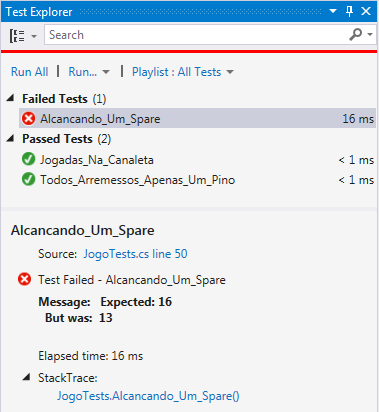
return pontuacao;

}

}

}

Rode todos os testes e verifique que todos passam. Se “descomentarmos” o trecho do teste relacionado ao teste do “Spare”, vamos verificar que isso ainda não será o suficiente para tudo sair como esperado. Verifique.



O que precisamos fazer para que esse teste (e a implementação) funcione é primeiro ter certeza que estamos nos referindo ao arremesso da primeira bola do “Frame” e, também, devemos “andar” pelo array de “Frames” de dois em dois (afinal, são dois arremessos por “Frame”).

namespace JogoBoliche.Core {

public class Jogo {

private const int PontuacaoSpare = 10;

private int[] \_arremessos = new int[21]; // Todos os arremessos, e um possível 3º arremesso no 10º frame

private int \_arremessoAtual = 0;

public void Arremesso(int pinos) {

\_arremessos[\_arremessoAtual++] = pinos;

}

public int Pontuacao() {

const int totalFrames = 10;

var pontuacao = 0;

var frameIndex = 0;

for (var frame = 0; frame < totalFrames; frame++) {

if (Spare(frameIndex)) {

pontuacao += PontuacaoSpare + \_arremessos[frameIndex + 2];

frameIndex += 2;

} else {

pontuacao += \_arremessos[frameIndex] + \_arremessos[frameIndex + 1];

frameIndex += 2;

}

}

return pontuacao;

}

private bool Spare(int frameIndex) {

return ((\_arremessos[frameIndex] + \_arremessos[frameIndex + 1]) == 10);

}

}

}

O teste também sofreu um pequeno refactor.

using JogoBoliche.Core;

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

private Jogo \_jogo;

[SetUp]

public void SetUp() {

\_jogo = new Jogo();

}

[TearDown]

public void TearDown() {

\_jogo = null;

}

private void ExecutarArremessos(int vezes, int pinos) {

for (int arremesso = 0; arremesso < vezes; arremesso++) {

\_jogo.Arremesso(pinos);

}

}

private void ArremessarSpare() {

\_jogo.Arremesso(5);

\_jogo.Arremesso(5);

}

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

// Feito pelo método SetUp

// act

ExecutarArremessos(20, 0);

// assert

Assert.AreEqual(0, \_jogo.Pontuacao());

}

[Test]

public void Todos\_Arremessos\_Apenas\_Um\_Pino() {

// arrange

// Feito pelo método SetUp

// act

ExecutarArremessos(20, 1);

// assert

Assert.AreEqual(20, \_jogo.Pontuacao());

}

[Test]

public void Alcancando\_Um\_Spare() {

// arrange

// Feito pelo método SetUp

// act

ArremessarSpare();

\_jogo.Arremesso(3);

ExecutarArremessos(17, 0); // Todos os próximos arremessos foram na canaleta. =(

// assert

Assert.AreEqual(16, \_jogo.Pontuacao());

}  
 }  
}

## Quarto Teste: Strike!

Agora precisamos verificar o “Strike”. O “Strike” ocorre quando são derrubados todos os pinos em um único arremesso (“Frame”). Também, tem sua contagem de pontos um pouco diferente do convencional. O “Strike” vale 10 pontos por “Frame”, sendo que um “Strike” consecutivo acumula a pontuação do “Strike” subsequente, até um máximo de 30 pontos. Caso o próximo arremesso não cause um “Strike”, somente os pontos do “Frame” serão adicionados como bônus. Exemplo:

Frame 1: Strike (10 pontos) : Frame 1 + Frame 2 + Frame 3 = 30

Frame 2: Strike (10 pontos) : Frame 2 + Frame 3 + Frame 4(1) = 20

Frame 3: Strike (10 pontos) : Frame 3 + Frame 4 = 19

Frame 4: 0 / 9 (9 pontos): Frame 4 = 9

Pontuação total até o momento: 78

Nosso teste:

[Test]

public void Alcancando\_Um\_Strike() {

// arrange

// Feito pelo método SetUp

// act

\_jogo.Arremesso(10);

\_jogo.Arremesso(3);

\_jogo.Arremesso(4);

ExecutarArremessos(16, 0); // Todos os próximos arremessos foram na canaleta. =(

// Frame 1 (Strike): 10 + (Frame 2:1) + (Frame 2:2) = 17

// Frame 2:1 (3): 3

// Frame 2:2 (4): 4

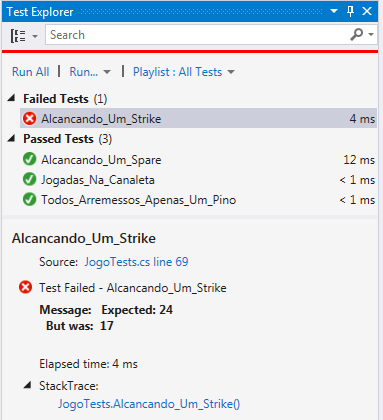
// Total: 24

// assert

Assert.AreEqual(24, \_jogo.Pontuacao());

}

Se rodarmos, veremos um erro.



Vamos refatorar nosso código para que ele funcione como esperado:

namespace JogoBoliche.Core {

public class Jogo {

private const int PontuacaoSpare = 10;

private const int PontuacaoStrike = 10;

private int[] \_arremessos = new int[21]; // Todos os arremessos, e um possível 3º arremesso no 10º frame

private int \_arremessoAtual = 0;

public void Arremesso(int pinos) {

\_arremessos[\_arremessoAtual++] = pinos;

}

public int Pontuacao() {

const int totalFrames = 10;

var pontuacao = 0;

var frameIndex = 0;

for (var frame = 0; frame < totalFrames; frame++) {

if (Strike(frameIndex)) {

pontuacao += PontuacaoStrike + PontuacaoBonusStrike(frameIndex);

frameIndex++;

} else if (Spare(frameIndex)) {

pontuacao += PontuacaoSpare + PontuacaoBonusSpare(frameIndex);

frameIndex += 2;

} else {

pontuacao += SomarTotalPontuacaoFrame(frameIndex);

frameIndex += 2;

}

}

return pontuacao;

}

private bool Strike(int frameIndex) {

return \_arremessos[frameIndex] == 10;

}

private int PontuacaoBonusStrike(int frameIndex) {

return \_arremessos[frameIndex + 1] + \_arremessos[frameIndex + 2];

}

private bool Spare(int frameIndex) {

return ((\_arremessos[frameIndex] + \_arremessos[frameIndex + 1]) == 10);

}

private int PontuacaoBonusSpare(int frameIndex) {

return \_arremessos[frameIndex + 2];

}

private int SomarTotalPontuacaoFrame(int frameIndex) {

return \_arremessos[frameIndex] + \_arremessos[frameIndex + 1];

}

}

}

Nosso teste ficou assim:

using JogoBoliche.Core;

using NUnit.Framework;

namespace JogoBoliche.UnitTest {

[TestFixture]

public class JogoTests {

private Jogo \_jogo;

[SetUp]

public void SetUp() {

\_jogo = new Jogo();

}

[TearDown]

public void TearDown() {

\_jogo = null;

}

private void ExecutarArremessos(int vezes, int pinos) {

for (int arremesso = 0; arremesso < vezes; arremesso++) {

\_jogo.Arremesso(pinos);

}

}

private void ArremessarSpare() {

\_jogo.Arremesso(5);

\_jogo.Arremesso(5);

}

[Test]

public void Jogadas\_Na\_Canaleta() {

// arrange

// Feito pelo método SetUp

// act

ExecutarArremessos(20, 0);

// assert

Assert.AreEqual(0, \_jogo.Pontuacao());

}

[Test]

public void Todos\_Arremessos\_Apenas\_Um\_Pino() {

// arrange

// Feito pelo método SetUp

// act

ExecutarArremessos(20, 1);

// assert

Assert.AreEqual(20, \_jogo.Pontuacao());

}

[Test]

public void Alcancando\_Um\_Spare() {

// arrange

// Feito pelo método SetUp

// act

ArremessarSpare();

\_jogo.Arremesso(3);

ExecutarArremessos(17, 0); // Todos os próximos arremessos foram na canaleta. =(

// assert

Assert.AreEqual(16, \_jogo.Pontuacao());

}

[Test]

public void Alcancando\_Um\_Strike() {

// arrange

// Feito pelo método SetUp

// act

\_jogo.Arremesso(10);

\_jogo.Arremesso(3);

\_jogo.Arremesso(4);

ExecutarArremessos(16, 0); // Todos os próximos arremessos foram na canaleta. =(

// Frame 1 (Strike): 10 + 3 + 4 (17)

// Frame 2:1 (3): 3

// Frame 2:2 (4): 4

// Total: 24

// assert

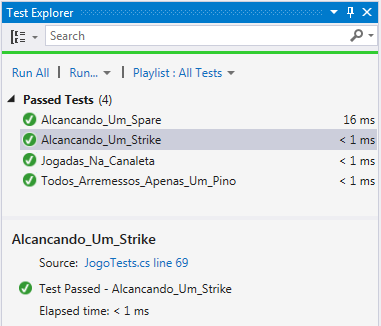
Assert.AreEqual(24, \_jogo.Pontuacao());

}

}

}

Resultado final:



Fim!