Prática 11 - Test-Driven Development (TDD)

Breno Farias da Silva

2025-03-07

Introdução

Este relatório descreve a aplicação da técnica de *Test-Driven Development* (TDD) na resolução de um problema clássico: o cálculo da pontuação de um jogo de boliche. A prática foi realizada no contexto da disciplina **PPGCC12 - Teste de Software**, ministrada pelo professor Reginaldo Ré.

O desenvolvimento seguiu os ciclos do TDD: **vermelho-verde-refatoração**, priorizando a escrita de testes antes da implementação da lógica de produção.

Descrição do Problema

O método `calcular_pontuacao_boliche` recebe uma lista de inteiros que representam os pinos derrubados em cada jogada, e retorna a pontuação total do jogo, respeitando as regras oficiais de boliche:

- Um jogo é composto por 10 frames.
- Um *strike* (10 pinos na primeira jogada do frame) concede bônus com os pontos das duas jogadas seguintes.
- Um *spare* (10 pinos em duas jogadas) concede bônus com os pontos da próxima jogada.
- O décimo frame pode conter até duas jogadas bônus, caso haja strike ou spare.

Ciclos TDD: Teste, Implementação e Refatoração

1º Ciclo: Jogo com todas jogadas zeradas

Teste escrito:

```
@Test
void deveRetornarZeroParaTodasAsJogadasComZero() {
   int[] jogadas = new int[20];
   assertEquals(0, Boliche.calcular_pontuacao_boliche(jogadas));
}
```

Código de produção:

O código em produção que contém o método *calcular_pontuacao_boliche* está localizado em *src/main/java/edu/utfpr/Boliche.java*

```java
public static int calcular_pontuacao_boliche(int[] jogadas) {
   return 0;
}
```

Refatoração:

Código reescrito para lógica parcial com loop básico e soma, iniciando o esqueleto da função.

2º Ciclo: Jogo simples sem strikes ou spares

Teste escrito:

```java
@Test
void devecalcular_pontuacao_bolicheSimples() {
   int[] jogadas = new int[20];
   Arrays.fill(jogadas, 3);
   assertEquals(60, Boliche.calcular_pontuacao_boliche(jogadas));
}
```

Código de produção:

```java
int score = 0;
int index = 0;

for (int frame = 0; frame < 10; frame++) {
   score += jogadas[index] + jogadas[index + 1];
   index += 2;
}
return score;
```

Refatoração:

Nenhuma refatoração necessária neste ponto.

3º Ciclo: Jogo com spare

Teste escrito:

```java
@Test
void deveCalcularSpare() {
   int[] jogadas = {3, 7, 4, 2, 0, 0, ...};
   assertEquals(20, Boliche.calcular_pontuacao_boliche(jogadas));
}
```

Código atualizado:

```java
if (jogadas[index] + jogadas[index + 1] == 10) { // Spare
   score += 10 + jogadas[index + 2];
```

```
    index += 2;
}
```

Refatoração:

Extração da lógica do strike e spare como condição dentro do loop for.

4º Ciclo: Jogo com strike

Teste escrito:

```
@Test
void deveCalcularStrike() {
    int[] jogadas = {10, 5, 3, 0, 0, ...};
    assertEquals(26, Boliche.calcular_pontuacao_boliche(jogadas));
}
```

Código atualizado:

```
if (jogadas[index] == 10) { // Strike
    score += 10 + jogadas[index + 1] + jogadas[index + 2];
    index++;
}
```

Refatoração:

Reordenação das condições para priorizar strike antes de avaliar spare.

5º Ciclo: Jogo perfeito

Teste escrito:

```
@Test
void deveCalcularJogoPerfeito() {
    int[] jogadas = {10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10};
    assertEquals(300, Boliche.calcular_pontuacao_boliche(jogadas));
}
```

Código final (produção):

```
public static int calcular_pontuacao_boliche(int[] jogadas) {
    int score = 0;
    int index = 0;

    for (int frame = 0; frame < 10; frame++) {
        if (jogadas[index] == 10) { // Strike
            score += 10 + jogadas[index + 1] + jogadas[index + 2];
            index++;
```

```
        } else if (jogadas[index] + jogadas[index + 1] == 10) { // Spare
            score += 10 + jogadas[index + 2];
            index += 2;
        } else {
            score += jogadas[index] + jogadas[index + 1];
            index += 2;
        }
    }
    return score;
}
```

Implementação dos Testes

Os testes foram escritos utilizando JUnit 5, sem uso de mocks ou dublês, pois a função é puramente algorítmica e determinística.

Todos os testes unitários foram definidos em:

`src/test/java/edu/utfpr/BolicheTest.java`

Execução dos Testes

Os testes foram executados utilizando o comando padrão do Maven:

```
mvn clean test
```

Resultados:

Teste	Status
Jogo com zeros	Passou
Jogo simples	Passou
Jogo com spare	Passou
Jogo com strike	Passou
Jogo perfeito (12 strikes)	Passou

Relatórios Gerados

- Surefire Reports: `target/surefire-reports/`
- Cobertura de código (Jacoco): `target/site/jacoco/index.html`
- Cobertura estimada: **100% das linhas cobertas** (todas as regras testadas explicitamente).

Conclusão

A prática foi essencial para consolidar os conceitos de TDD. Cada nova funcionalidade foi guiada por um teste falho inicial, seguido por uma implementação mínima e posterior refatoração. Esse processo resultou em um código limpo, testável e totalmente coberto por testes.

Além disso, os testes garantem que o método `calcular_pontuacao_boliche` lide corretamente com todos os casos relevantes: jogadas comuns, spares, strikes, bônus no décimo frame e jogos perfeitos.

Referências

- Livro: Effective Software Testing — Maurício Aniche, 2022.
- Documentação JUnit 5: https://junit.org/junit5/
- Documentação Jacoco: https://www.jacoco.org/