

Teste Caixa-Branca

Prof. André Takeshi Endo

Até o momento, vimos uma abordagem de **teste caixa preta**. Basicamente, apenas os requisitos e as funcionalidades do software são utilizadas para projetar e implementar os casos de teste.

Em alguns casos, é preciso que o teste parta de um código já implementado. Nesse caso, diz-se que o teste realizado é **caixa branca**. Nesse tipo de teste, a estrutura interna do software (ou seja, o código fonte dos programas) pode ser utilizada para guiar o projeto e a implementação dos casos de teste.

No teste caixa-branca, a qualidade dos testes gerados podem ser medidos pela porcentagem de cobertura de um dado critério. A seguir, serão apresentados dois **critérios baseados no fluxo de controle**.

Line Coverage: o primeiro critério que será abordado é a “cobertura de linhas” (em Inglês, *Line Coverage*). Esse critério é bem simples e objetiva cobrir as linhas de código em programa. Por exemplo, considere a classe em Java na Figura 1.

```
7 public class MathOps
8 {
9     public static int safe_add(int left, int right)
10         throws Exception
11     {
12         if(right > 0) {
13             if(left > Integer.MAX_VALUE - right)
14                 throw new Exception("Overflow");
15         }
16         else {
17             if(left < Integer.MIN_VALUE - right)
18                 throw new Exception("Underflow");
19         }
20
21         return left + right;
22     }
23 }
```

Figura 1: método que faz uma adição segura de inteiros.

Ao testar o método `safe_add`, para conseguir 100% de cobertura do critério *Line Coverage*, casos de teste deveriam ser implementados de forma que, ao executá-los, o fluxo de execução passe ao menos uma vez pelas

Linhas 9 a 22. A porcentagem de cobertura do *Line Coverage* é dado pelo total de linhas executadas pelos testes dividido pelo total de linhas do método.

- **Nota:** quando medir a porcentagem de cobertura, desconsidere as linhas em branco e as linhas que contém apenas o símbolos “{” e “}”.

Exercício 1: para cada um dos casos de teste listados na Figura 2, diga quais são as linhas de código executadas no método da Figura 1. Conseguiu-se 100% de cobertura para o critério *Line Coverage*?

```
@Test
void testSomaNormal() throws Exception {
    int res = MathOps.safe_add(10, 20);
    assertEquals(30, res);
}

@Test
void testOverflow() {
    Exception exception = assertThrows(Exception.class, () -> {
        MathOps.safe_add(Integer.MAX_VALUE, 20);
    });

    assertEquals("Overflow", exception.getMessage());
}

@Test
void testUnderflow() {
    Exception exception = assertThrows(Exception.class, () -> {
        MathOps.safe_add(Integer.MIN_VALUE, -2);
    });

    assertEquals("Underflow", exception.getMessage());
}
```

Figura 2: Casos de teste em JUnit.

Exercício 2: implemente um projeto Java (usando uma IDE, Netbeans ou Eclipse) com o código apresentado nas Figuras 1 e 2.

Branch Coverage: este critério é referente a “cobertura de desvios” (em Inglês, *Branch Coverage*). Este critério objetiva cobrir um valor verdadeiro e um valor falso para cada condição existente em um programa. As diferentes direções que são tomadas nessas condições (verdadeiro ou falso) são

chamados de desvios. Em Java, essas condições existem nas instruções if, while, for, do-while, switch e expressões ternárias ("? : ").

No código apresentado na Figura 1, nós temos exatamente três condições, nas Linhas 12, 13 e 17; isso implica no total de seis desvios. A porcentagem de cobertura do *Branch Coverage* é dado pelo total de desvios executados pelos testes dividido pelo total de desvios do método.

Exercício 3: usando os casos de teste na Figura 2, qual a porcentagem de cobertura para o critério *Branch Coverage*? Caso a cobertura seja menor que 100%, adicione os casos de teste necessários para cobrir os desvios (*branches*) faltantes.

Exercício 4 (Vincenzi e de Deus): implemente uma classe Java e os testes em JUnit para cobrir 100% dos critérios *Line Coverage* e *Branch Coverage*, considerando o método a seguir.

```
2      public void bolha(int[] a, int size) {
3          int i, j, aux;
4          for (i = 0; i < size; i++) {
5              for (j = size - 1; j > i; j--) {
6                  if (a[j - 1] > a[j]) {
7                      aux = a[j - 1];
8                      a[j - 1] = a[j];
9                      a[j] = aux;
10             }
11         }
12     }
13 }
```

Exercício 5 (Vincenzi e de Deus): implemente uma classe Java e os testes em JUnit para cobrir 100% dos critérios *Line Coverage* e *Branch Coverage*, considerando o método a seguir.

```
15      void insercao(int a[], int size) {
16          int i, j, aux;
17          for (i = 1; i < size; i++) {
18              aux = a[i];
19              j = i - 1;
20              while (j >= 0 && a[j] >= aux) {
21                  a[j + 1] = a[j];
22                  j--;
23              }
24              a[j + 1] = aux;
25          }
26      }
```

Exercício 6: implemente uma classe Java e os testes em JUnit para cobrir 100% dos critérios *Line Coverage* e *Branch Coverage*, considerando os métodos a seguir.


```
public class MyRandomNumber {  
    Random random = new Random();  
    int anterior = -1;  
  
    public int nextRandomNumber(int begin, int end) throws IntervaloInvalidoException {  
        verificarExcecoes(begin, end);  
        int atual = gerarEntre(end, begin);  
        if(atual == anterior) {  
            atual++;  
            if(atual > end)  
                atual = begin;  
        }  
        anterior = atual;  
        return atual;  
    }  
  
    private int gerarEntre(int end, int begin) {  
        int diff = end - begin + 1;  
        int ret = begin + random.nextInt(diff);  
        return ret;  
    }  
  
    private void verificarExcecoes(int begin, int end) throws IntervaloInvalidoException {  
        if (begin < 0 || end < 0) {  
            throw new IntervaloInvalidoException("begin eh menor que zero");  
        }  
  
        if (begin >= end) {  
            throw new IntervaloInvalidoException("begin eh maior que zero");  
        }  
    }  
}
```

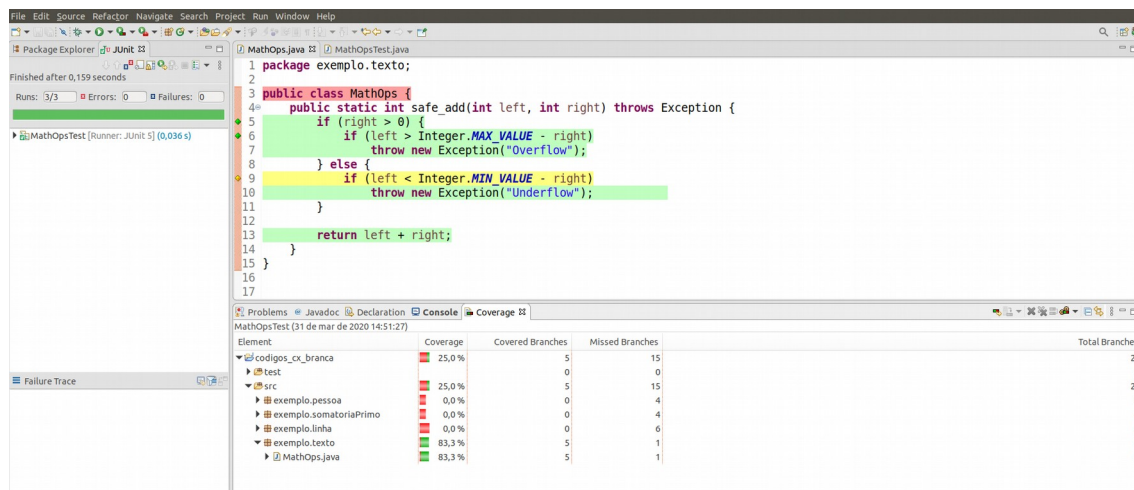
Veja exemplos de códigos em:

https://github.com/andreendo/software-testing-undergrad-course/tree/master/codigos_cx_branca

Visualizar cobertura de código no Eclipse (Recomendado)

No Eclipse (versão 2019-12), a coleta e visualização da cobertura de código para testes usando JUnit já vêm embutida na IDE.

1. Escolha a classe de teste JUnit que você gostaria de visualizar a cobertura
2. Clique com o botão direito, escolha “Coverage As” → “JUnit Test”
3. É possível visualizar os trechos de código cobertos e o relatório de cobertura.
4. Para remover informações de cobertura, use o botão “Remove All Sessions” 



The screenshot shows the Eclipse IDE interface. The main editor displays the source code of `MathOps.java` from the `exemplo.texto` package. The code includes a `safe_add` method that throws exceptions for overflow and underflow. The code is color-coded to show coverage: green for covered lines and yellow for missed branches.

The Coverage tab at the bottom provides a detailed report for the test run `MathOpsTest [31 de mar de 2020 14:51:27]`. The report includes a table with the following data:

Element	Coverage	Covered Branches	Missed Branches	Total Branches
codigos_cx_branca	25,0 %	5	15	20
test	0,0 %	0	0	0
src	25,0 %	5	15	20
exemplo.pessoa	0,0 %	0	4	4
exemplo.somatoriaPrimo	0,0 %	0	4	4
exemplo.linha	0,0 %	0	6	6
exemplo.texto	83,3 %	5	1	6
MathOps.java	83,3 %	5	1	6

Visualizar cobertura de código no NetBeans (apenas <8.2)

1. No Netbeans, escolha a opção de menu “Ferramentas” → Plugins
2. Na aba plugins disponíveis, pesquisar por “Jacocoverage”.
3. Selecione o plugin TikiOne JaCoCoverage e instalar.

Caso tenha problemas via IDE, você pode baixar o plugin diretamente de:

<http://plugins.netbeans.org/plugin/48570/tikione-jacocoverage>

Explore a ferramenta.

- Botão direito no projeto → “Test with JaCoCoverage”
- Verifique o relatório.
- O que é verde, vermelho e amarelo?
- Para limpar informações de cobertura, botão direito → “reset coverage data”