Java Spring Expert Capítulo 5

Cobertura de código com Jacoco

## Competências

* Parte 1: Avançando nos testes unitários
  + Abordagens de teste
    - Caixa branca
    - Caixa preta
  + Principais anotações Mockito
    - @Mock
    - @Spy
    - @InjectMocks
    - Exemplo utilização @Mock, @Spy e @InjectMocks
  + Introdução à cobertura de código
* Parte 2: Cobertura de código com Jacoco
  + Tipos básicos de cobertura de código/testes
    - Statement Coverage (Line Coverage)
    - Branch Coverage
    - Function Coverage
  + Discussão
  + Ferramentas para cobertura
  + JaCoCo
    - Dependência
  + Recursos importantes
* Exercícios

# Parte 1: Avançando nos testes unitários

Objetivo: apresentar tópicos adicionais sobre testes unitários para complementar o conteúdo aprendido nos [capítulos anteriores](https://drive.google.com/drive/folders/1Bg8pH4xbc07jSZsO3WImQPnw44aP04WN?usp=drive_link).

## Abordagens de teste

* Uma das partes principais deste processo é o **teste de software**, que tem como objetivo descobrir sistematicamente diferentes classes de erros com uma quantidade de tempo e esforço mínimos.
* Os testes podem ser divididos em 2 grupos, sendo o teste **caixa branca** e o **teste caixa preta**:
  + **Caixa branca**:
    - O teste de caixa branca possui esse nome porque o testador tem acesso à estrutura interna da aplicação. Logo, seu foco é garantir que os componentes do software estejam concisos.
    - O teste é realizado diretamente no código fonte, ou seja, o teste analisa a estrutura interna dos componentes do sistema.
    - Nesta técnica são analisados os caminhos básicos do software e a ideia é que esses caminhos sejam testados.
    - Um dos principais testes de caixa branca são os **testes unitários**;
  + **Caixa preta**:
    - Baseia-se nos requisitos básicos do software, sendo o foco nos requisitos da aplicação, ou seja, nas ações que deve desempenhar.
    - Diferente do teste de caixa branca, ele possui esse nome porque o código-fonte é ignorado no teste. Assim, ao se utilizar dessa técnica, o *tester* não está preocupado com os elementos constitutivos do software, mas em como ele funciona.
    - Os principais testes caixa preta são os testes de integração e de API;

|  |
| --- |
| 1. À direita uma representação do teste de **caixa branca** que tem acesso direto ao código fonte; (b) À esquerda o teste de **caixa preta** onde o que são analisados são os requisitos. |

* Uma das principais vantagens ao implementar os testes unitários, por exemplo, é proteger os recursos já implementados de serem quebrados à medida que o código muda. Além de proporcionar ao desenvolvedor um senso de proteção da aplicação contra bugs.

## Principais anotações Mockito

### @Mock

Um mock em testes unitários é um objeto que implementa o comportamento de algum componente do sistema. Em outras palavras, substitui as dependências. É muito usado para incluir alguma dependência (ex: Repository ou Service).

### @Spy

A ideia do spy é permitir encapsular a instância de algum objeto existente. É como se ele espionasse um objeto real. Por padrão, o spy irá delegar as chamadas de métodos para o objeto real e rastrear as chamadas e parâmetros.

É usado em circunstâncias mais específicas quando comparado com o @Mock, sendo elas:

* Simular o comportamento de um método da mesma classe que está sendo testada.
* Testes Unitários em sistemas legados.

### @InjectMocks

Usado para instanciar o objeto testado automaticamente e injetar todas as dependências anotadas @Mock e @Spy.

## Exemplo utilizando @Mock, @Spy e @InjectMocks

Considere um serviço **ProductService** que implementa os métodos insert e update de produtos. Cada produto é composto pelos campos id, name e price. Ambos os métodos utilizam uma função **validateData()**, responsável por validar se o nome não é vazio e se o preço é positivo.

Implemente os testes unitários da camada de serviços para os métodos insert e update considerando os cenários de teste.

*Observação*: Assuma que não estamos usando lib de validação

**Repositório**: <https://github.com/devsuperior/poc-example-mock-spy>

# Parte 2: Cobertura de código com Jacoco

## Introdução à cobertura de código

* No processo de desenvolvimento de software um dos principais objetivos é criar aplicações de alta qualidade e livres de falhas, atendendo aos requisitos funcionais e não funcionais.
* Uma das partes principais deste processo é o **teste de software**, que tem como objetivo descobrir sistematicamente diferentes classes de erros com uma quantidade de tempo e esforço mínimos.
* Uma das principais vantagens ao implementar os testes unitários, por exemplo, é proteger os recursos já implementados de serem quebrados à medida que o código muda. Além de proporcionar ao desenvolvedor um senso de proteção da aplicação contra bugs.
* No entanto, alguns autores defendem que somente implementar os testes unitários não é o suficiente. Neste caso, muitos recomendam abordagens de cobertura de código.
* **Cobertura de código** é uma métrica que indica a porcentagem de código que está coberta por ao menos um teste automatizado.
  + **Exemplo:** Uma cobertura de 90% indica que 10% do código não está coberto por nenhum teste automatizado.
* A cobertura de testes é muito recomendada em alguns contextos desde o princípio do desenvolvimento do software por eliminar possíveis bugs ou permitir que sejam descobertos no estágio inicial do desenvolvimento.
* Podemos dizer que a cobertura de código é uma parte que compõe a **cobertura de testes (test coverage)**, que é definido como métrica de teste de software que mede a quantidade de testes executados, dado um conjunto de casos de testes.
* Enquanto a cobertura de código é uma medida quantitativa (número de linhas de código que foram executadas pelos testes), a cobertura de teste é uma medida qualitativa, permitindo validar a implementação dos requisitos do produto.
* Para realizar a cobertura de código de maneira adequada é necessário ter acesso aos componentes internos (classes e funções) da aplicação.

## Tipos básicos de cobertura de código

### Statement Coverage (Line coverage)

* Usado para verificar quantas instruções ou comandos do código são executadas;
* Também é chamado de *line coverage* por alguns autores;
* O cálculo do percentual de *statement coverage* pode ser calculado da seguinte forma:
  + ***Statement coverage*** = Número de statements executados / Número total de statements \* 100
* Exemplo:

| **Algoritmo 1**: *Soma(a, b)* | |
| --- | --- |
| **1** | result = a + b |
| **2** | **if** result > 0 **then** |
| **3** | print(“Greater than zero”); |
| **4** | **else if** result < 0 |
| **5** | print(“Less than zero”); |
| **6** | **else** |
| **7** | print(“Zero”); |
| **8** | **end** |

Retirado de <https://www.baeldung.com/cs/code-coverage>

* Considerando 3 cenários, temos:
  + Para *a = 3, b = 5*, serão executadas as linhas 1, 2, 3 e 8. Desta forma temos 4 linhas de 8 o que significa que temos 4/8 ou 50% de cobertura.
  + Para *a = 3, b = -5*, serão executadas as linhas 1, 2, 4, 5 e 8, ou seja 5/8 o que equivale a 63% de cobertura.
  + Para *a = 10, b = -10*, serão executadas as linhas 1, 2, 4, 6, 7 e 8, ou seja 75%
  + Neste caso, para termos uma cobertura de 100% no método soma, todos os 3 cenários devem ser considerados.
* A vantagem desta abordagem está em permitir verificar diferentes caminhos e quais deles não estão cobertos;

### Branch Coverage

* Verifica se cada ramificação de cada estrutura de controle (incluindo if/else, switch case, for, while) é executada;
* O cálculo do percentual de *branch coverage* pode ser calculado da seguinte forma:
  + *Branch coverage* = Número de branchs executadas / Número total de branchs \* 100
* Exemplo:

| **Algoritmo 2**: *Even(a)* | |
| --- | --- |
| **1** | **if** *a is even* **then** |
| **2** | print(“Even”); |
| **3** | print(a); |

Retirado de <https://www.baeldung.com/cs/code-coverage>



* Considerando os 2 cenários:
  + Para *a = 1*, será executada as linhas 1 e 3, ou seja 2/3 equivalente a 75%;
  + Para *a = 4*, serão executadas as linhas 1-3 ou seja 100%
  + Neste caso, ambos os cenários oferecem uma cobertura de 100%.
* Algumas vantagens desta abordagem:
  + Permite identificar comportamentos não previstos
  + Permite mapear áreas do código que fonte que outras abordagens não mapeiam.

### Function Coverage

* A cobertura de função verifica se cada função de um programa está sendo chamada pelo menos uma vez.
* **Exemplo**: No caso de uma aplicação composta por uma única função ou método, a implementação de um único teste de unidade para este método resultará em uma cobertura de 100%.

## Discussão

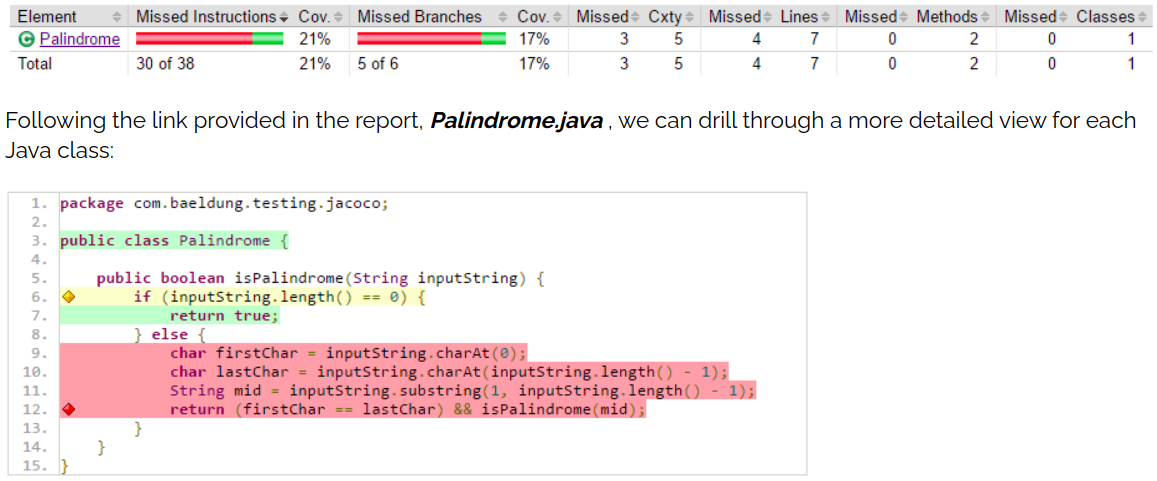
* Qual o percentual de cobertura a ser perseguido?
* Apesar de a ideia parecer ótima, alcançar os 100% de cobertura de código não deveria ser uma meta absoluta, pois existem trechos que não precisam diretamente de serem testados.
  + **Exemplo**: Códigos que podemos gerar automaticamente com a própria IDE, como Getters, Setters.
* É uma decisão difícil escolher qual trecho de código não precisa ser testado. O fato é que se você precisar priorizar, teste aqueles métodos que são complicados e/ou importantes. Use o número de cobertura para ajudá-lo a identificar trechos que não estão testados.
* Alcançar os 100% de cobertura é desejável, mas não é uma garantia que o seu sistema seja a prova de defeitos.

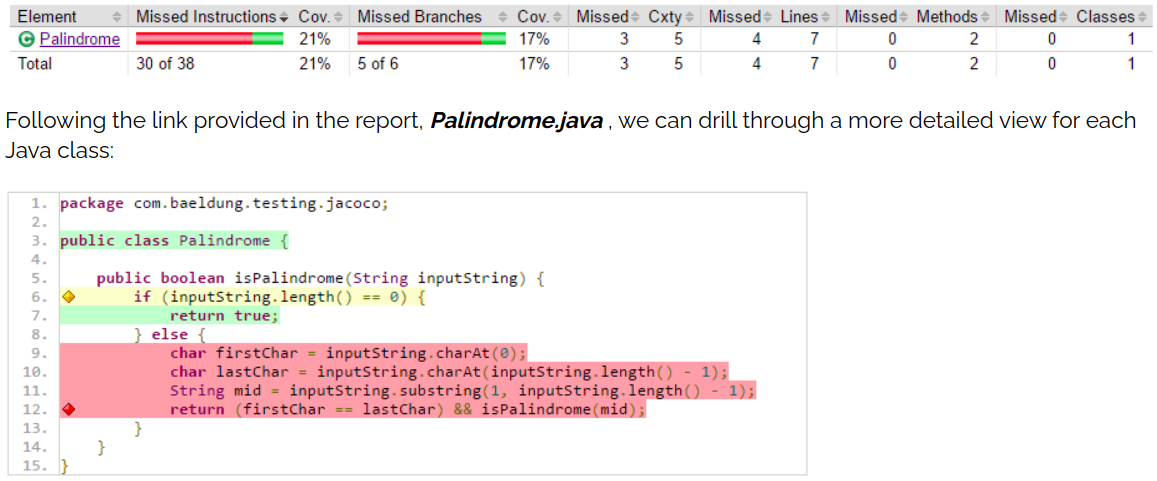
## Ferramentas para cobertura

* Algumas das principais ferramentas de cobertura de testes são:
  + JaCoCo no contexto do Java;
  + Istambul no contexto do Javascript;
  + Coverage.py no contexto do Python;
  + NCover no contexto do .NET.
* Vamos focar na utilização do **JaCoCo**;

## JaCoCo

* JaCoCo é uma ferramenta de código aberto (*open-source*) usada para **mensurar a cobertura de código** em aplicações;
* A partir de relatórios visuais é possível identificar as partes do código que estão cobertas e que ainda faltam cobertura;
* O JaCoCo implementa 3 métricas principais para cobertura, sendo:
  + *Line Coverage/Statement*;
  + *Branch Coverage*;
  + *Cyclomatic complexity*: A partir de uma combinação linear apresenta o números de caminhos que necessitam cobertura;
* O JaCoCo apresenta auxilia o usuário na visualização e análise da cobertura usando diamantes coloridos, conforme a imagem abaixo:



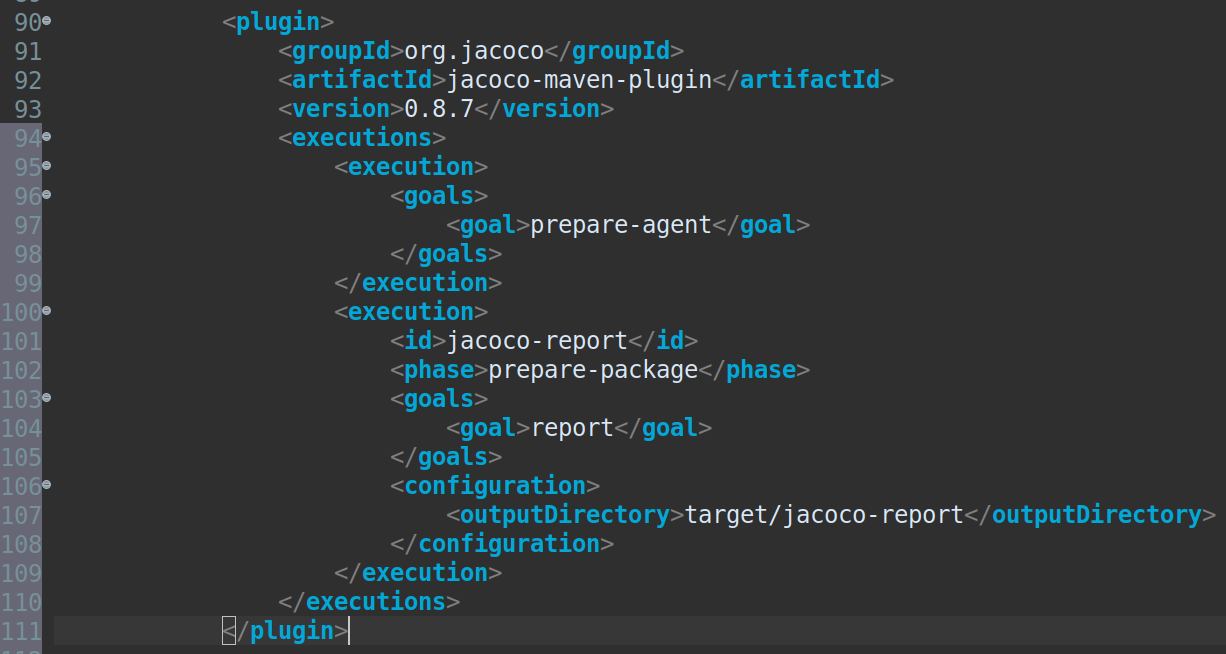


Retirado de <https://www.baeldung.com/jacoco>

* Diamante vermelho: Indica que nenhum teste está cobrindo o branch;
* Diamante amarelo: Indica que o código está parcialmente coberto;
* Diamante verde: Indica que todo o branch foi testado e coberto;

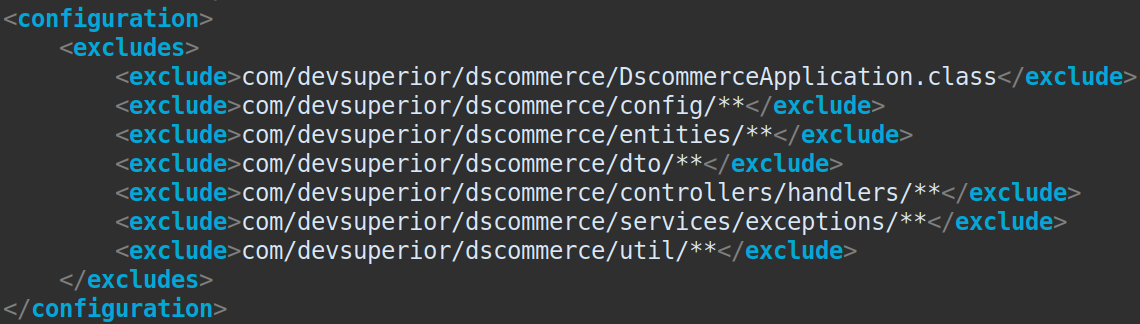
## Dependência

* Para incluir o JaCoCo no projeto é necessário incluir a seguinte dependência:



Retirado de <https://www.eclemma.org/jacoco/trunk/doc/maven.html>

* Incluir classes a serem excluídas da cobertura de testes



Retirado de <https://www.eclemma.org/jacoco/trunk/doc/maven.html>

* Link dependência JaCoCo

<https://gist.github.com/oliveiralex/fd320a363a4294860b43c8e9bf63ebfc>

## Recursos importantes

* Classe **TokenUtil**
  + Responsável por obter token de acesso;
  + <https://gist.github.com/oliveiralex/faeba65e214f7e6d738c01516ac7d6d2>

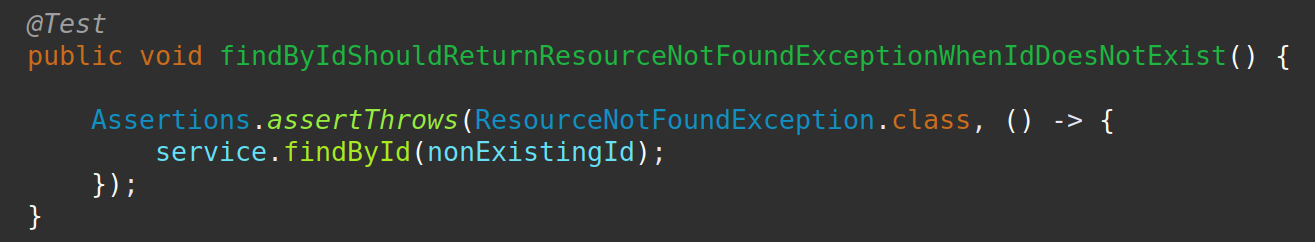
# Exercícios de fixação: test coverage e testes unitários na camada service

## Baixar projeto DSCommerce

<https://drive.google.com/drive/folders/1Xk_3gwt8jKVgHKg49hYwpA6dRWU5OL-4>

## **Problema 1:** Consultar produto por id

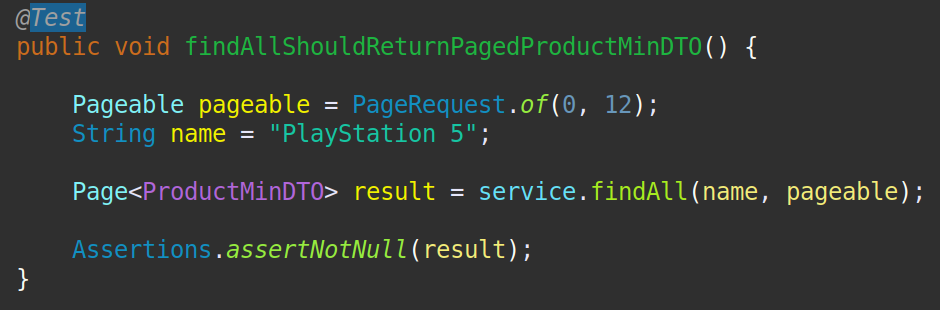
Implemente o *mock* para simular o comportamento do método **findById** do *ProductRepository*, de forma que ao chamar ProductRepository.findById passando como argumento um id de produto não existente, deve retornar uma instância do Optional vazia (empty). Na figura 1 (abaixo), é apresentado o teste unitário do serviço ProductService para o findById que retorna uma exceção ResourceNotFoundException quando o id do produto não existir.



**Figura 1:** Exemplo do método findById que retorna resource not found quando id não existir

## **Problema 2:** Consultar produto por nome

Implemente o *mock* para o método **searchByName** do ProductRepository. O método pode receber com argumento qualquer cadeia de caracteres representando um nome e um Pageable e deve retornar um page de Products. Na figura 2 (abaixo), é apresentado o teste unitário findAll, que deve retornar um page de ProductMinDTO no ProductService.



**Figura 2:** Método findAll deve retornar page de ProductMinDTO

## **Problema 3:** Atualizar produto

Implemente os testes unitários do método ProductService.update, cobrindo os seguintes cenários. Lembre-se de mockar os métodos do ProductRepository que são usados pelo ProductService.update.

1. Atualização de produto atualiza produto para id existente
2. Atualização de produto lança exceção *ResourceNotFoundException* para produto inexistente

## **Problema 4:** Deletar produto

Implemente os testes unitários para o método ProductSevice.delete, cobrindo os seguintes cenários. Lembre-se de mockar os métodos do ProductRepository que são usados pelo ProductService.delete.

1. Deleção de produto deleta produto para id existente
2. Deleção de produto lança exceção *ResourceNotFoundException* para id inexistente
3. Deleção de produto lança exceção *DatabaseException* para id dependente (quando o produto participa de um pedido).

## **Problema 5**: Carregar usuário

Implemente os testes unitários do método UserService.loadUserByUsername, cobrindo todos os cenários importantes. Você deve identificar os casos de testes necessários para cobertura completa do método loadUserByUsername.

## **Problema 6**: Consultar usuário logado

Implemente os testes unitários do método UserService.getMe, cobrindo todos os cenários importantes. Você deve identificar os casos de testes necessários para cobertura completa do método getMe.