WEB ACADEMY

Testes

Daniel Augusto Nunes da Silva

Apresentação

Ementa

- Visão geral. Taxonomia (testes de unidade, integração e sistema). **Testes Automatizados**. Teste de **back-end** (teste de API REST) e de **front-end** (testes de UI, E2E). **Frameworks de teste** (back-end e front-end). **Cobertura de código**.

Objetivos

- **Geral:** Capacitar o aluno a compreender e aplicar **técnicas e ferramentas** modernas para a realização de **testes em aplicações web back-end e front-end**, enfatizando a importância dos testes no ciclo de desenvolvimento de software.

- **Específicos:**
  - Compreender o papel dos testes no processo de desenvolvimento de software e distinguir os diferentes níveis de testes.
  - Aplicar técnicas de testes no back-end, abordando testes unitários e de integração em aplicações Spring Boot.
  - Desenvolver habilidades para conduzir testes no front-end, com foco em aplicações Angular e testes end-to-end.
  - Explorar técnicas para mensurar a cobertura de testes/código, bem como as ferramentas associadas.

Conteúdo programático

Introdução

- O processo de Verificação e Validação;
- O que é um teste e por que testar?;
- Limites dos testes;
- Classificação de testes por nível;
- Automatização de testes;
- TDD;
- Frameworks.

Teste no Back-end

- Testes em aplicações Spring Boot;
- Teste de API REST;
- Testes de unidade (camadas de modelo, serviço e controle);
- Mocks;
- Testes de integração em controladores e repositórios de dados.

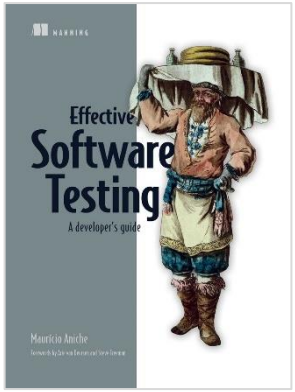
Teste no Front-end

- Testes em aplicações Angular;
- Testes de componentes;
- Testes de Sistema (end-to-end).

Cobertura de testes/código

- Definição;
- Tipos de cobertura;
- Cobertura de testes e cobertura de código;
- Nível de cobertura ideal;
- Ferramentas: back-end e front-end.

Bibliografia



Effective Software Testing

Maurício Aniche
1ª Edição – 2022
Editora Manning
ISBN 9781633439931

Engenharia de Software Moderna

Marco Tulio Valente
https://engsoftmoderna.info/

Sites de referência

- Angular Developer Guides - Testing:
  - https://angular.io/guide/
- Testing Angular - A Guide to Robust Angular Applications
  - https://testing-angular.com/
- Spring Boot Reference - Testing:
  - https://docs.spring.io/spring-boot/docs/3.0.11/reference/html/features.html#features.testing
- JUnit 5 User Guide:
  - https://junit.org/junit5/docs/current/user-guide/
- Testing Java with Visual Studio Code:
  - https://code.visualstudio.com/docs/java/java-testing

Contato

https://linkme.bio/danielnsilva/

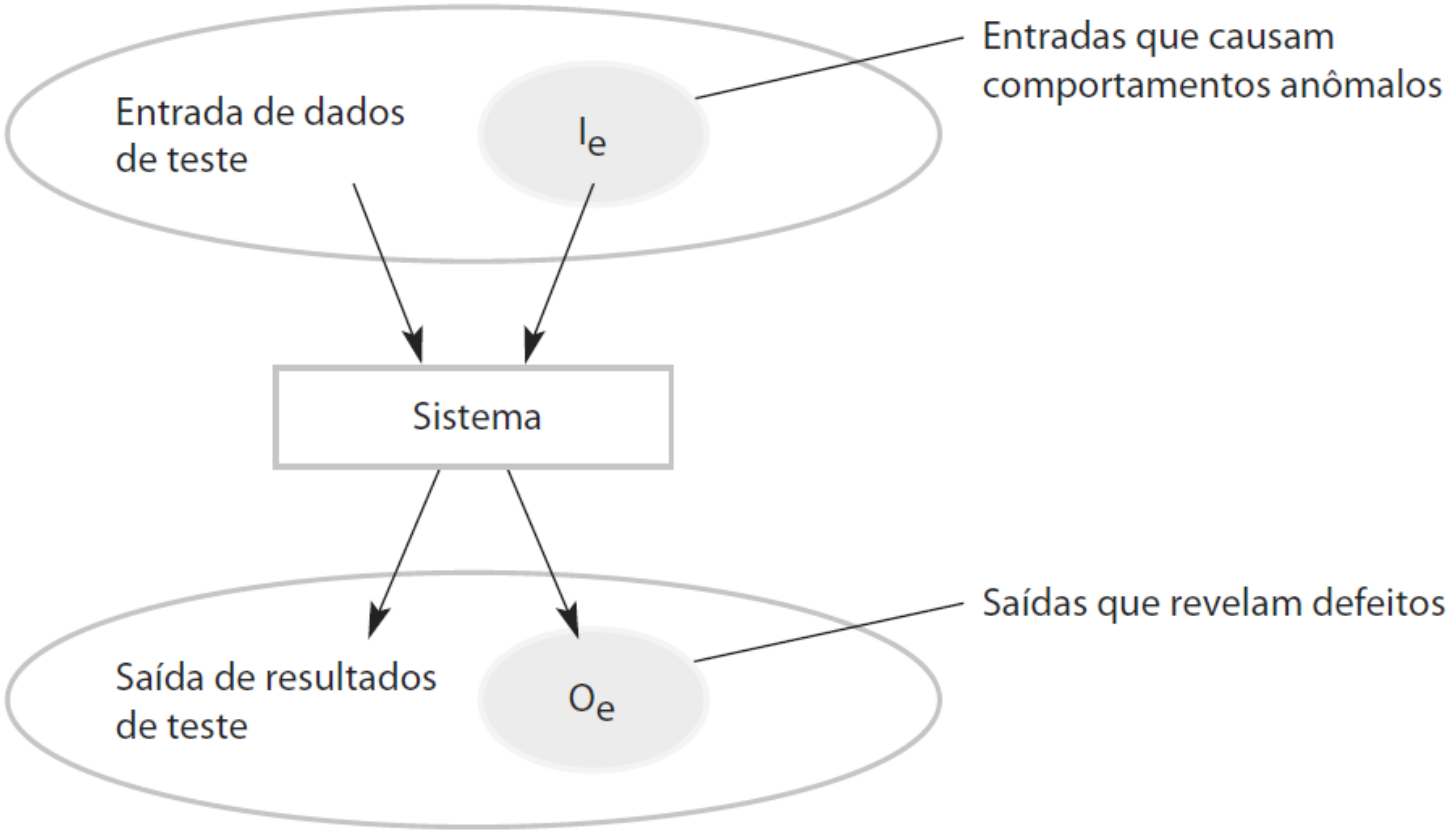
Introdução

Verificação e Validação (V&V)

- Uma das atividades básicas dos processos de densevolvimento de software.
- **Verificação:** assegurar que o software atenda aos requisitos.
- **Validação:** assegurar que o software atenda às necessidades e expectativas do cliente.
- **Teste** é uma das principais técnicas aplicadas ao processo de V&V.

O que é um teste?

- **Teste** é um processo que tem por objetivo mostrar que um **software faz o que é proposto a fazer**, além de **descobrir os defeitos** do software antes do uso.



Fonte: SOMMERVILLE, 2011.

Por que testar?

1. **Construir um produto de qualidade**: Testar software assegura que ele atende às especificações e expectativas, entregando um produto confiável ao usuário.

2. **Redução de custos**: Apesar de inicialmente exigir mais tempo e recursos, os testes podem reduzir gastos futuros, minimizando falhas após a implementação, o que poderia demandar mais recursos para corrigir os problemas.

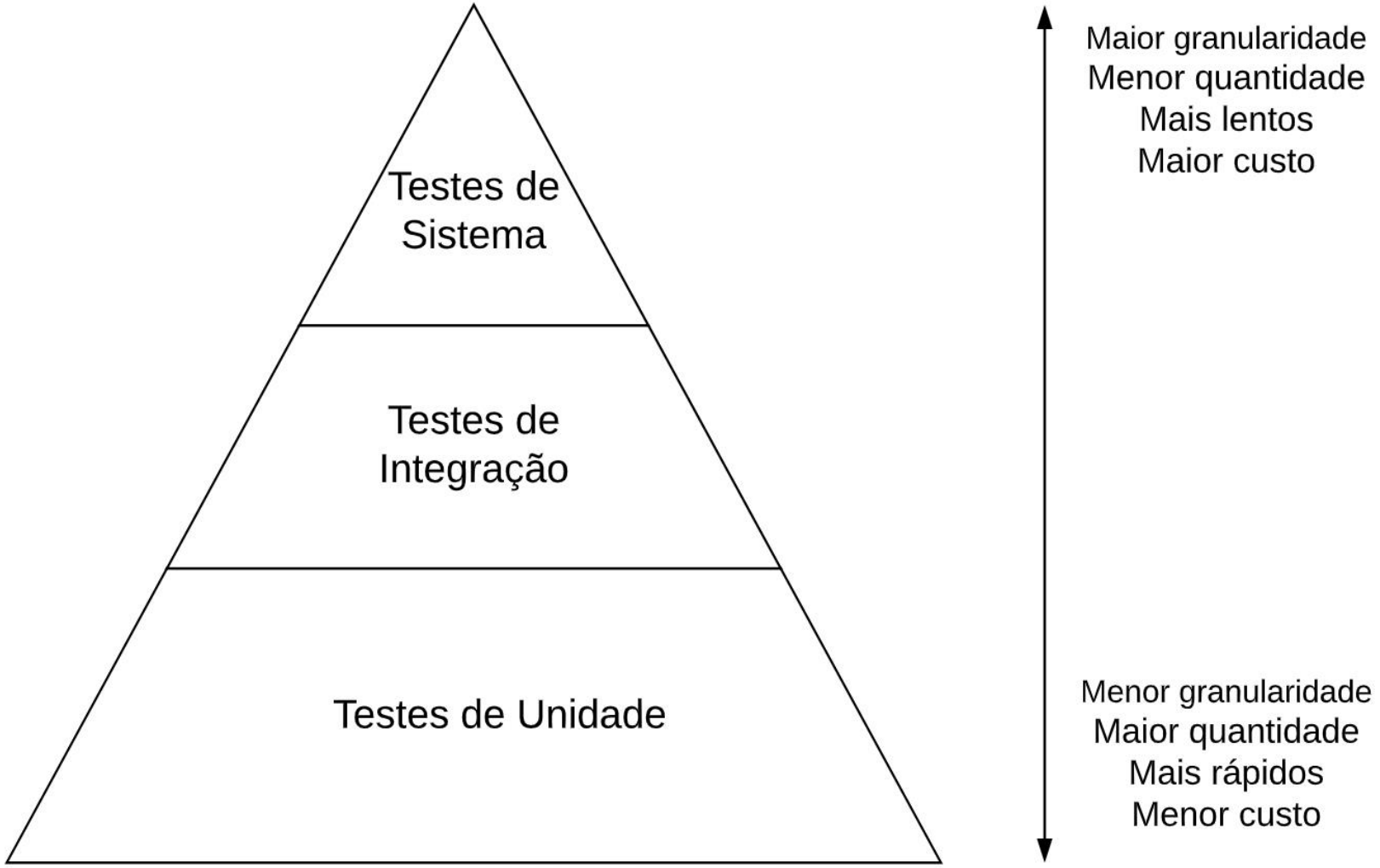
3. **Eficiência no processo de desenvolvimento**: Testes facilitam a identificação e correção de erros antecipadamente, acelerando o ciclo de desenvolvimento e lançamentos.

Limites dos testes

- Os testes não podem demonstrar se o software é livre de defeitos.
- É sempre possível que um teste que você tenha esquecido seja aquele que poderia descobrir mais problemas no sistema.

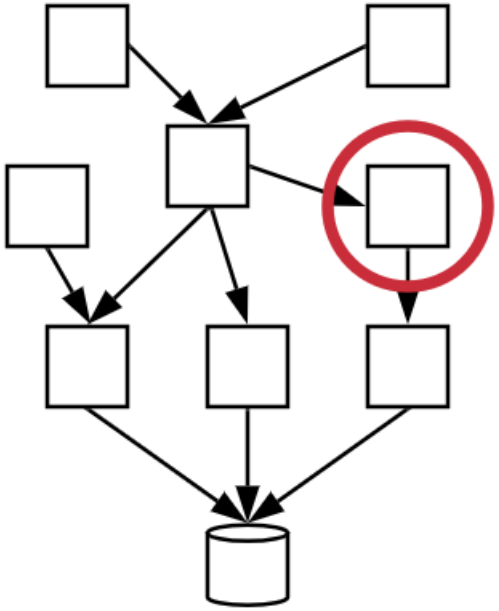
"Os testes podem mostrar apenas a presença de erros, e não sua ausência". ***(Edsger Dijkstra)***

Classificação de testes por nível



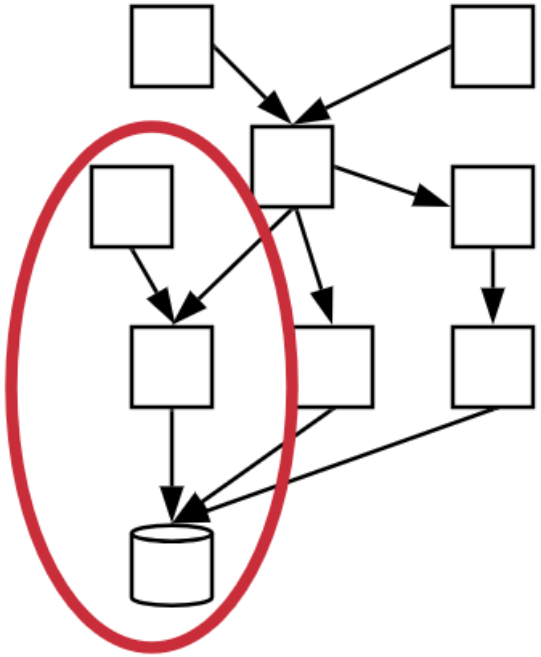
Fonte: VALENTE, 2020.

Classificação de testes por nível



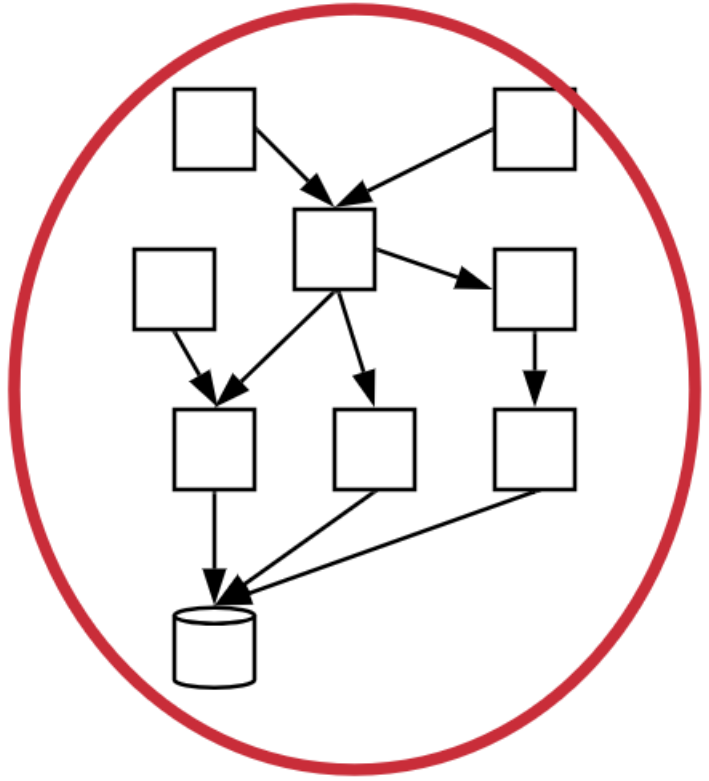
Teste de Unidade

Verificação de componentes de forma independente (pequenas partes do código, como uma classe ou função)



Teste de Integração

Verificação das interações entre os componentes (internos ou externos)



Teste de Sistema (*end-to-end*)

Simula uma sessão de uso do sistema por um usuário real.

Automatização de testes

- Processo de teste geralmente envolve uma mistura de testes manuais e automatizados.
  - **Teste manual:** testador executa o programa com alguns dados de teste e compara os resultados.
  - **Teste automatizado:** testes são codificados em um programa que é executado cada vez que o sistema em desenvolvimento é testado.

Test Driven Development – TDD

- **TDD** é uma técnica de desenvolvimento de software que enfatiza a **escrita de testes antes da implementação do código** (*test-first*).
- Uma das práticas de programação propostas pela metodologia ágil XP.

Frameworks

Teste no Back-end

Testes em aplicações Spring Boot

- O **Spring Boot** oferece vários recursos (métodos, anotações, etc.) para auxiliar no **teste de webapps**.
- O suporte a testes é fornecido por dois módulos: o **spring-boot-test** contém recursos principais, e o **spring-boot-test-autoconfigure** fornece autoconfiguração para testes.
- O **spring-boot-starter-test** importa ambos os módulos de teste do Spring Boot, bem como **JUnit**, **Mockito**, e várias outras bibliotecas úteis.

Teste de API REST

- Uma vez que não tem UI, os **testes de API** focam na lógica de negócios, garantindo que a **comunicação e os dados estejam corretos**, sem se preocupar com a apresentação visual.
- **Teste de endpoints**:
  - Cada endpoint de uma API REST precisa ser testado para garantir que está retornando os dados corretos e respondendo adequadamente a diferentes tipos de entradas.
- O foco são **testes de unidade e de integração**, mas testes de sistema também podem ser feitos dependendo do contexto.

Testes de unidade

- No contexto de uma API, testes de unidade garantem que cada camada da aplicação funcione corretamente de forma isolada.
  - Camadas: **modelo**, **serviço** e **controle**.
  - Apesar de ser possível testes de unidade na **camada de repositório de dados**, faz mais sentido que sejam **testes de integração**.
  - É necessário testar classes simples da camada de modelo (*getters* e *setters*)?
- Para testar componentes que dependem de outros é necessário simular o funcionamento das dependências com uso de **mocks**.

Mocks

- Mocks são **objetos simulados que replicam o comportamento de objetos reais em um sistema**, sendo utilizados para isolar e testar partes específicas de um software.
- Vantagens:
  - **Isolamento de componentes**: testar um componente de forma isolada implica em que, se o teste falhar, o problema está definitivamente no componente em teste e não em uma dependência externa.
  - **Eficiência e velocidade**: não é necessário interagir com dependências externas, como bancos de dados ou serviços web, o que reduz o tempo de execução dos testes.

Testes de integração

- No contexto de uma API, os testes de integração asseguram que os **serviços fornecidos por cada camada estão corretamente integrados** aos demais componentes do sistema.
  - Exemplo: teste que verifica o acesso a um endpoint passando por todas as camadas até o banco de dados (sem mock).
- **Mocks** ainda podem ser úteis em testes de integração quando se deseja isolar seu funcionamento em relação a um determinado componente.
  - Exemplo: teste que verifica integração entre camada de controle e serviço, utilizando mocks para simular o acesso ao banco de dados.

Teste no Front-end

Testes em aplicações Angular

- No Angular, os testes são automatizados usando principalmente o **Jasmine para definição dos testes** (entradas e saídas), e o **Karma como executor**, simulando um navegador web para rodar esses testes.
- Arquivos de teste possuem o sufixo **.spec.ts**
- **TestBed**: fornece um ambiente isolado para testar partes específicas da aplicação.
- Execução dos testes: **ng test**

```
// Agrupa testes relacionados
describe('AtendimentoService', () => {
  let service: AtendimentoService;
  // Configurações iniciais do teste
  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(AtendimentoService);
  });
  // Código do teste e expectativas
  it('should be created', () => {
    expect(service).toBeTruthy();
  });
  // ... mais testes (it) se necessário ...
});
```

Testes de componentes

- Um componente é uma combinação de template HTML e classe TypeScript, e portanto **a interação entre template e classe deve fazer parte do conjunto de testes**.
- Um componente também interage com o **DOM** e outros componentes.
- **TestBed** facilita o teste do componente no DOM, e pode testar a classe do componente isoladamente ou com interação com o DOM.

```
describe('AppComponent', () => {
  beforeEach(() => TestBed.configureTestingModule({
    imports: [RouterTestingModule],
    declarations: [AppComponent]
  }));
  it('should create the app', () => {
    const fixture = TestBed.createComponent(
        AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });
});
```

Testes de Sistema (end-to-end)

- Testes ***end-to-end*** (E2E) visam **simular a interação real do usuário com a aplicação**.
  - Testes manuais com usuários podem ser úteis, mas os testes E2E automatizados asseguram eficiência e consistência no processo de validação.
- Diferente dos testes com Karma e Jasmine, que focam apenas em partes isoladas, os testes E2E analisam a aplicação sob uma perspectiva mais ampla.
- Um exemplo de framework que auxilia na construção e execução de testes E2E é o **Cypress**.

Cobertura de testes/código

Definição

Métrica que ajuda a definir o número de testes necessários.

$$cobertura\ de\ testes = \frac{número\ de\ comandos\ executados\ pelos\ testes}{total\ de\ comandos\ do\ programa}$$

Tipos de cobertura

- A definição de **cobertura**, apresentada anteriormente, é **baseada em comandos (ou linhas)**, sendo a definição mais comum.

- Existem definições alternativas:
  - **Cobertura de funções:** percentual de funções que são executadas por um teste;
  - **Cobertura de *branches*:** percentual de *branches* de um programa que são executados por testes; um comando *if* sempre gera dois *branches* – quando a condição é verdadeira e quando ela é falsa;
  - **Cobertura de condições:** verifica se cada sub-expressão booleana são avaliadas.

Tipos de cobertura

Comandos

```
public int coverage(int x, int y) {
    z = 0;
    if ((x > 0) && (y > 0)) {
        z = x;
    }
    return z;
}

coverage(1, 0);
```

Branches

```
public int coverage(int x, int y) {
    z = 0;
    if ((x > 0) && (y > 0)) {
        z = x;
    }
    return z;
}

coverage(1, 0);
```

Tipos de cobertura

Funções

```
public int coverage(int x, int y) {
    z = 0;
    if ((x > 0) && (y > 0)) {
        z = x;
    }
    return z;
}

coverage(1, 0);
```

Condições

```
public int coverage(int x, int y) {
    z = 0;
    if ((x > 0) && (y > 0)) {
        z = x;
    }
    return z;
}

coverage(1, 0);
```

Cobertura de testes e cobertura de código

- A cobertura de código não é a mesma coisa que cobertura de testes.
- **Cobertura de código: métrica quantitativa** que visa medir quanto (%) do software é coberto/exercitado ao executar um determinado conjunto de casos de testes.
- **Cobertura de testes: métrica qualitativa** que visa medir a eficácia dos testes perante os requisitos testados, determinando se os casos de testes existentes cobrem os requisitos que estão sendo testados.

Existe cobertura ideal?

- Não existe um número mágico e absoluto para cobertura de testes/código.
- Depende do projeto.
- Mesmo quando se usa TDD, a cobertura de testes/código costuma não chegar a 100%, embora a tendência é que seja alta.
- **100%** de cobertura **não significa código livre de defeitos**.
- A **métrica** avaliada de forma isolada **não implica, necessariamente, em qualidade**.

Ferramentas

- Serviços como **Codecov** (https://codecov.io/) e **Coveralls** (https://coveralls.io/) integram-se ao GitHub (ou similares) para gerar relatórios de cobertura de código.
- **JaCoCo** - Java Code Coverage (https://www.jacoco.org/jacoco/)
  - Gerar relatório (após configuração do pom.xml): **mvn test**
- **Istanbul** / Angular (https://istanbul.js.org/)
  - Gerar relatório: **ng test --no-watch --code-coverage**

Fim!

Referências

- ANICHE, Maurício. **Effective Software Testing**. 1. ed. Shelter Island: Manning Publications, 2022. 328 p.
- MARCO TULIO VALENTE. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**, 2020. Disponível em: https://engsoftmoderna.info/
- MATHIAS SCHÄFER. **Testing Angular - A Guide to Robust Angular Applications**. [S. l.], 2022. Disponível em: https://testing-angular.com.
- SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo: Pearson Addison-Wesley, 2011.