# Web Academy

## Testes

Ufac

Manoel Limeira de Lima Júnior

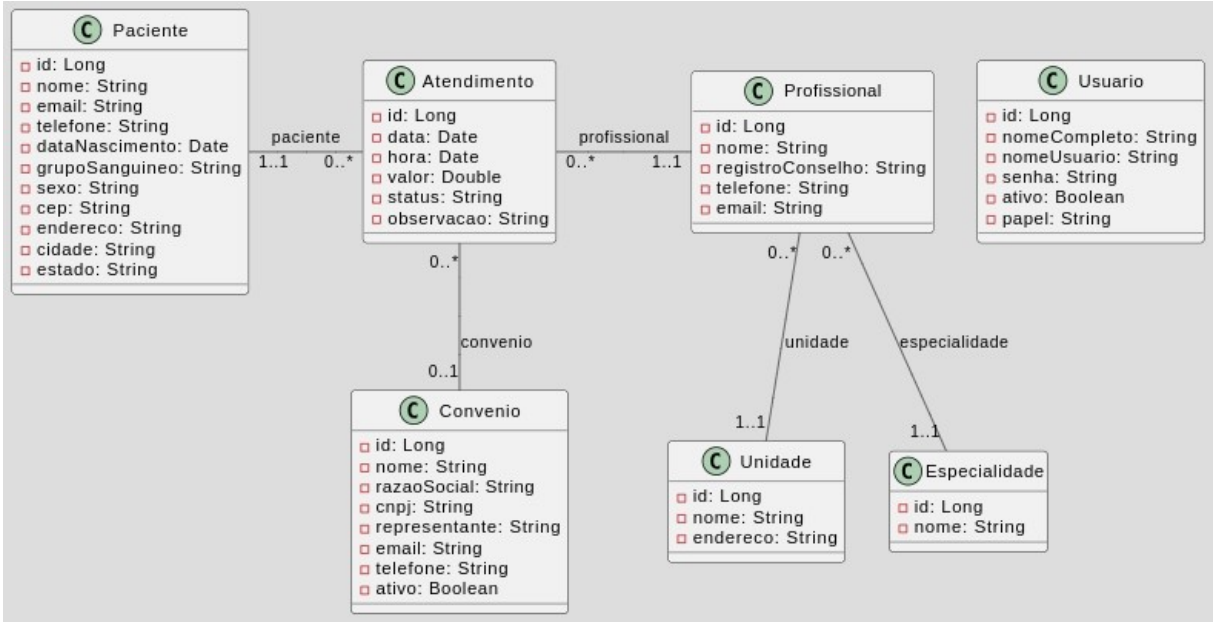

Web Academy

Ufac

# Apresentação

# SGCM – Sistema de Gerenciamento de Consultas Médicas

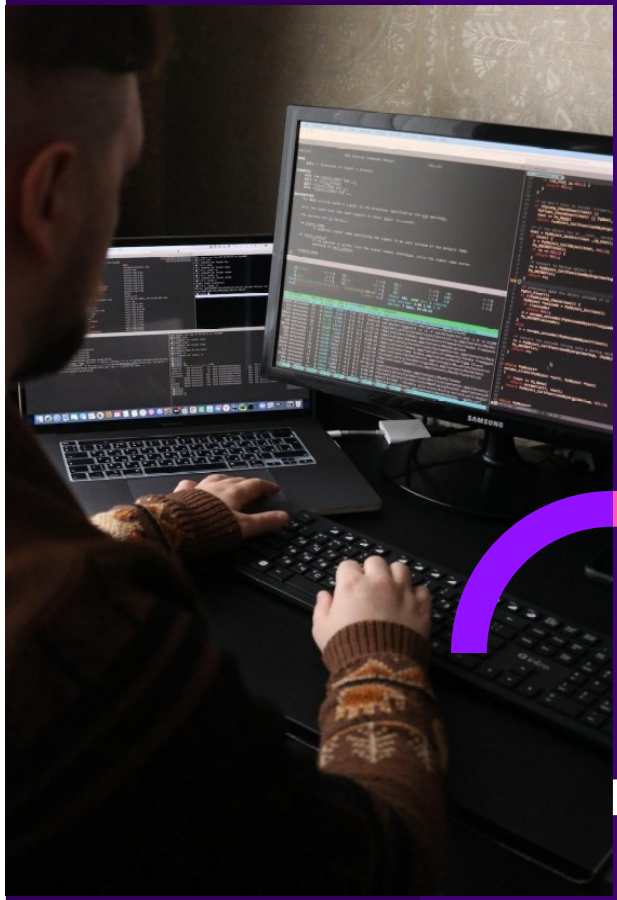
- Documentação: https://github.com/webacademyufac/sgcmdocs
  - Diagrama de classes

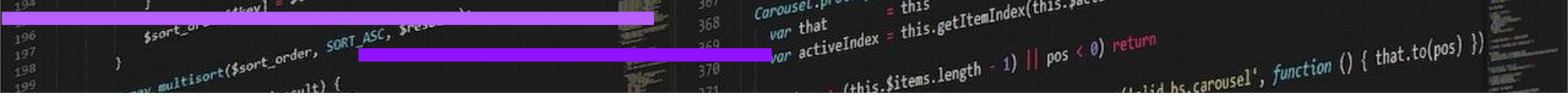


Web Academy

# Ementa

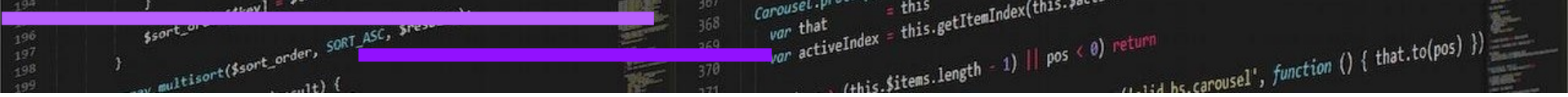
1. Visão geral.
2. Taxonomia (testes de unidade, integração e sistema).
3. Testes Automatizados.
4. Teste de back-end (teste de API REST) e de front-end (testes de UI, E2E).
5. Frameworks de teste (back-end e front-end).
6. Cobertura de código.





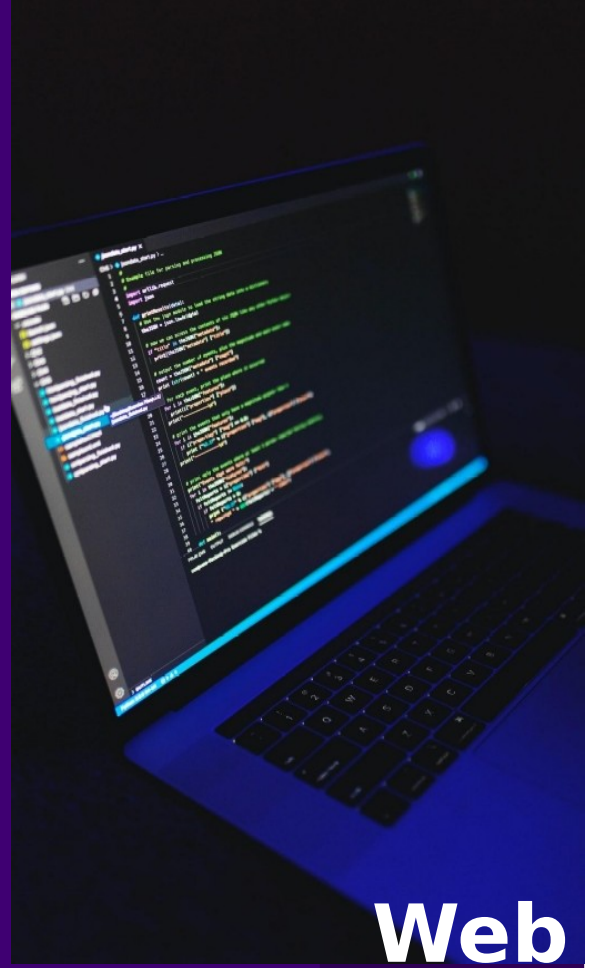
# Objetivos

- **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 de unidade 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

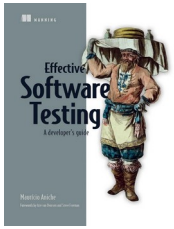
| Introdução | Testes Back-end | Testes Front-end | Cobertura |
|---|---|---|---|
| O processo de Verificação, Validação e Testes (VV&T);<br>Termos e conceitos;<br>O que é um teste e por que testar?;<br>Limites dos testes;<br>Classificação de testes por nível (Taxonomia);<br>Automatização;<br>TDD;<br>Frameworks. | Testes em aplicações Spring Boot;<br>Teste de API REST;<br>Testes de unidade (camadas de modelo, serviço e controle);<br>Mocks;<br>Testes de integração em controladores e repositórios de dados. | Testes em aplicações Angular;<br>Testes de componentes;<br>Testes de Sistema (end-to-end). | Definição;<br>Tipos de cobertura;<br>Cobertura de testes e cobertura de código;<br>Nível de cobertura ideal;<br>Ferramentas: back-end e front-end. |



Web

Academy

# Bibliografia


**Effective Software Testing**
1ª Edição – 2022
Maurício Aniche
Editora Manning


**Engenharia de Software Moderna**
Marco Tulio Valente
https://engsoftmoderna.info/


**Introdução ao Teste de Software.**
Marcio Delamaro. 2ª Edição.
Rio de Janeiro: GEN LTC, 2016.

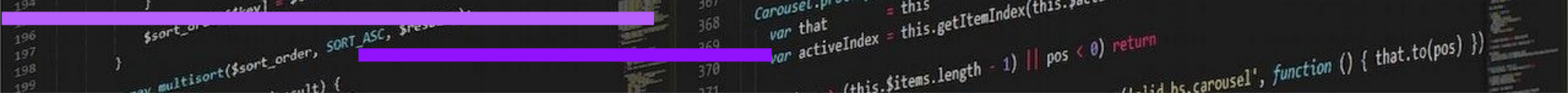
# Sites de referência

- 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


Web Academy

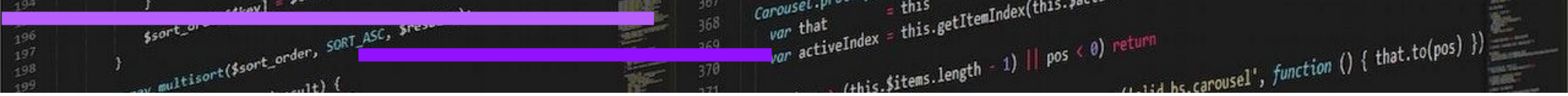
Ufac

# Introdução



# Introdução ao Teste de Software

- O desenvolvimento de software pode se tornar uma **tarefa bastante complexa**.
- Está sujeito a **diversos tipos de problemas** que acabam resultando na obtenção de um produto diferente daquele que se esperava.
- A maioria dos problemas tem **uma origem: o erro humano**.
- O software depende da habilidade, da interpretação e da execução das pessoas que o constroem; por isso, **erros acabam surgindo**, mesmo com a utilização de métodos e ferramentas de engenharia de software.
- **Para que os erros sejam descobertos** antes de o software ser liberado para utilização, **existe uma série de atividades, chamadas de "Verificação, Validação e Teste",** ou simplesmente **"VV&T".**



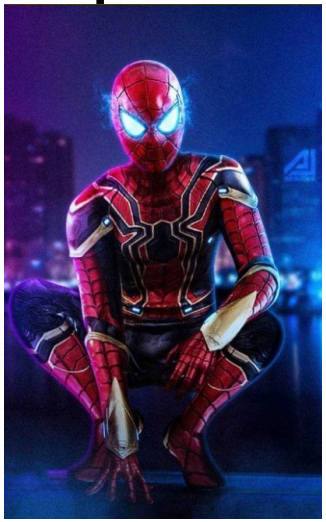
# Verificação, Validação e Teste de software (VV&T)

- As atividades de **VV&T têm a finalidade de garantir** que **tanto o modo** pelo qual o software está sendo construído **quanto o produto** em si estejam em conformidade com o especificado.
- As atividades de VV&T não se restringem ao produto final. Podem e devem ser **conduzidas durante todo o processo de desenvolvimento do software**, desde a sua concepção, e englobam diferentes técnicas.
- Em geral, dividem-se as atividades de VV&T em:
  - **Estáticas** que não requerem a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas.
  - **Dinâmicas** que se baseiam na execução de um programa ou de um modelo.

# Diferença entre Verificação e Validação

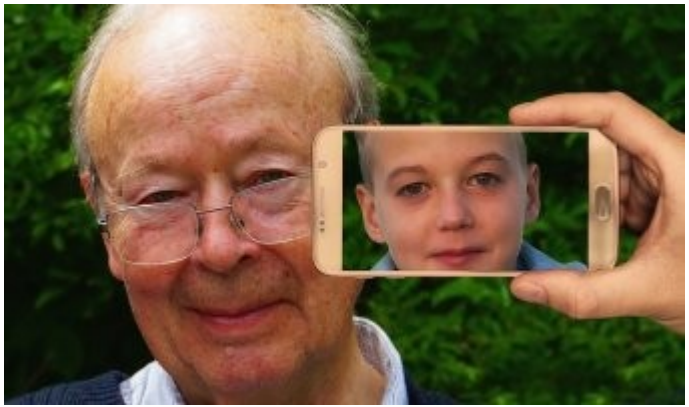
- Barry Boehm, expressou sucintamente a diferença entre validação e verificação (BOEHM, 1979).
  - **Verificação**: estamos construindo o produto da maneira certa?
  - **Validação**: estamos construindo o produto certo?

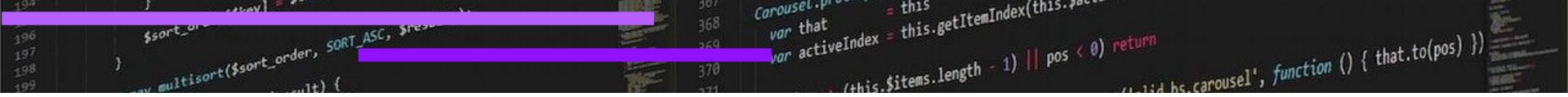
**Expectativa e Realidade**



# Engenharia de Software: Verificação e Validação e Testes (VV&T)

- Conjunto de atividades dos processos de desenvolvimento de software.
  - **Verificação**: assegurar que o software atenda aos requisitos (funcionais e não funcionais).
  - **Validação**: assegurar que o software atenda às necessidades e expectativas do cliente.
    - **Testes** executam o código-fonte para encontrar erros, defeitos ou falhas.
    - **Inspeções** ou **revisões** (técnicas estáticas) no código ou qualquer artefato (requisitos, modelos, diagramas) sem execução para encontrar defeitos.

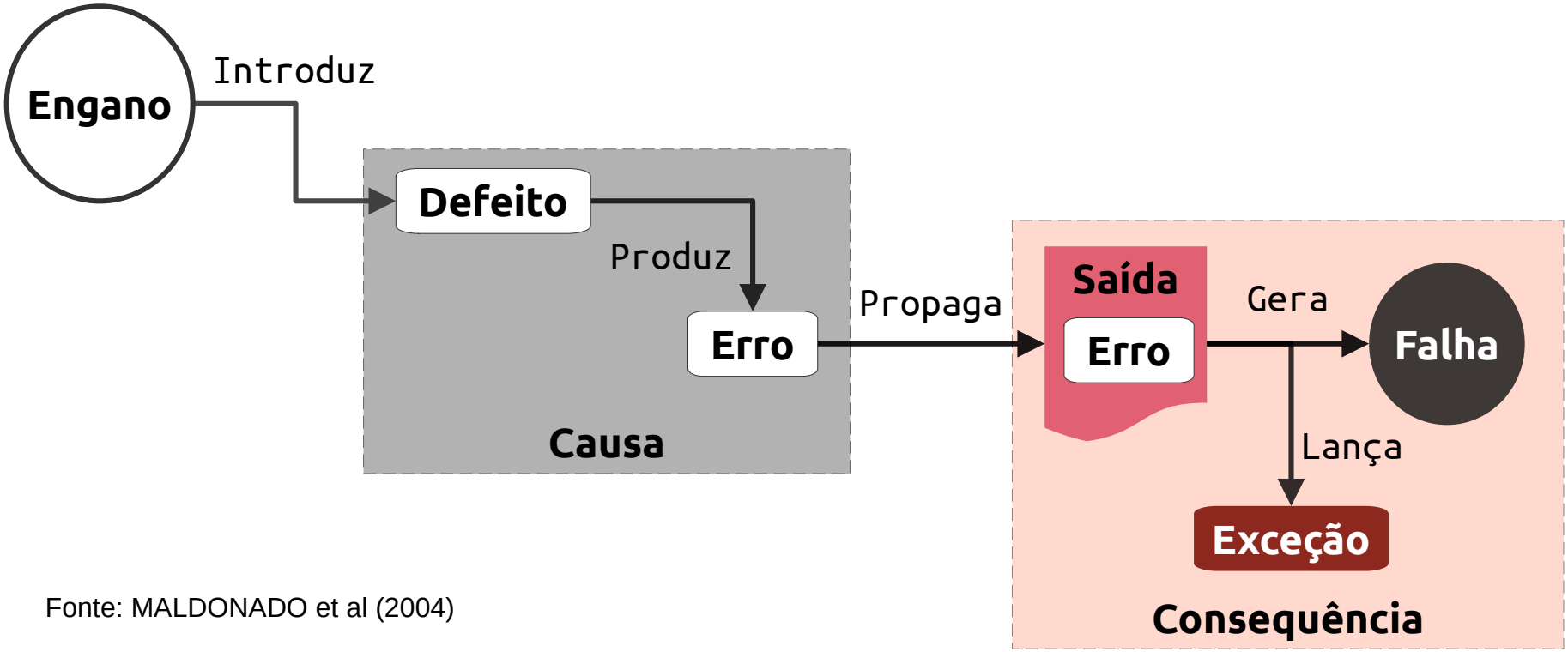




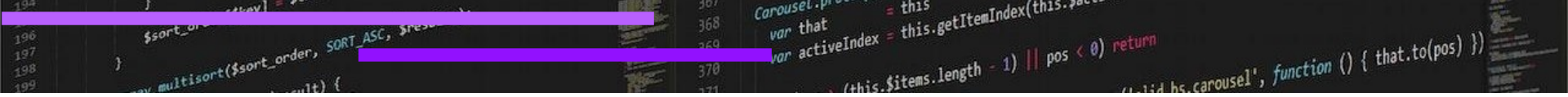
# Definições de termos

- **Engano** (*mistake*): é uma ação humana que **introduz um erro**.
- **Erro** (*error*): é a **diferença entre um resultado observado** e o **resultado verdadeiro**.
- **Defeito** (*fault*)**:** é uma **manifestação de um erro**, uma imperfeição em um artefato de software que não atende a seus requisitos, precisa ser consertado e pode acarretar em uma falha.
- **Falha** (*failure*): é quando um software ou componente **executa uma função fora dos limites** especificados. Um resultado incorreto.
- **Exceção** (*exception*): é um evento que causa a **suspensão da execução normal** do software.

# Engano, erro, defeito, falha e exceção

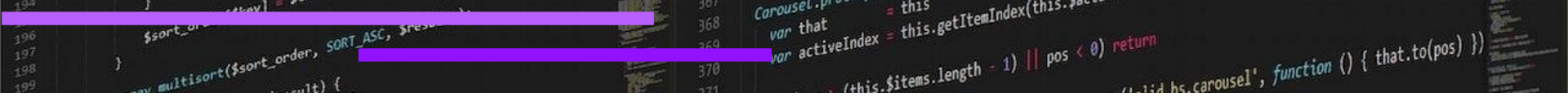



Fonte: MALDONADO et al (2004)



# Por que existem falhas no software?

- Pessoas cometem erros – isso é natural/normal.
- Erros são cometidos por diversos motivos:
  - Pressão do tempo;
  - Falha humana;
  - Participantes do projeto inexperientes ou insuficientemente qualificados;
  - Falta de comunicação entre os participantes do projeto, incluindo falta de comunicação sobre os requisitos e a modelagem;
  - Complexidade do código, modelagem, arquitetura, o problema a ser resolvido ou as tecnologias utilizadas;
  - Mal-entendidos;
  - Tecnologias novas ou desconhecidas.

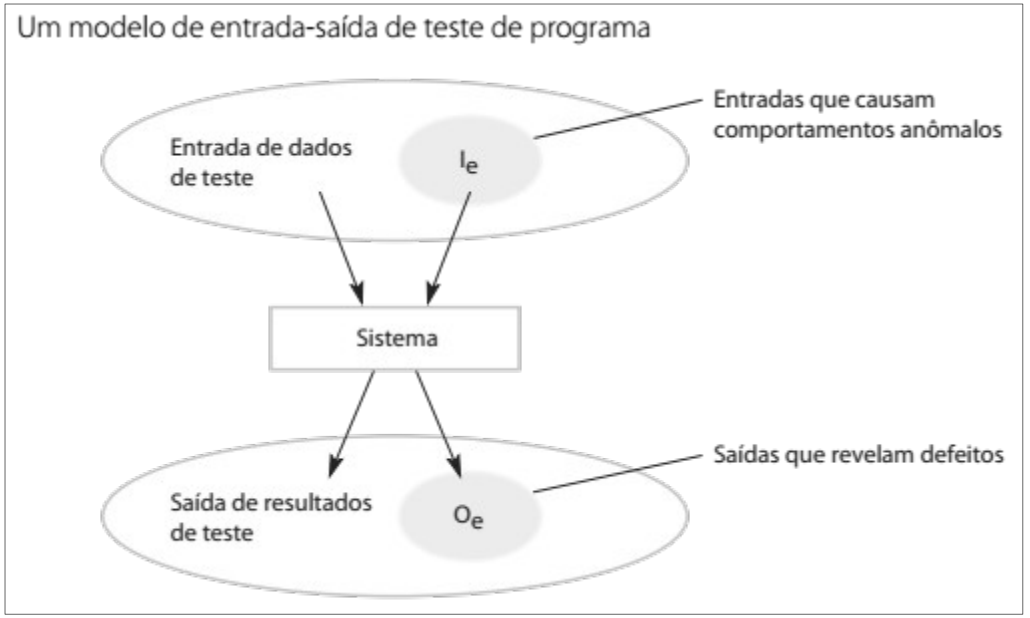


# Tipos e exemplos de falhas

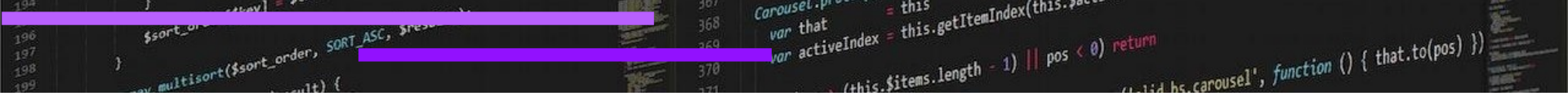
- **Visuais**
  - Problemas de alinhamento de componentes
  - Sobreposição de componentes
  - Texto impossível de ler
- **Funcionais**
  - O usuário não consegue fazer login
  - O usuário não consegue fazer pagamento com dois cartões
  - Não é possível atualizar o número de itens no carrinho de compras
  - O usuário não consegue escolher outro endereço de entrega
- **Não-funcionais**
  - Desempenho: a página demora 15 segundos para carregar
  - Segurança: a senha digitada não aparece ofuscada/oculta

# O que é um teste de software?

- É um processo com objetivo mostrar que **um software faz o que é proposto a fazer**, além de descobrir os defeitos do software antes do uso.
- O teste de software consiste em executar um software com o objetivo de **revelar uma falha** (MYERS, 1979).
  - **Depuração** (*debug*) é o processo de encontrar um **defeito dado uma falha na execução do software**. Resultado de uma atividade de teste bem sucedida.

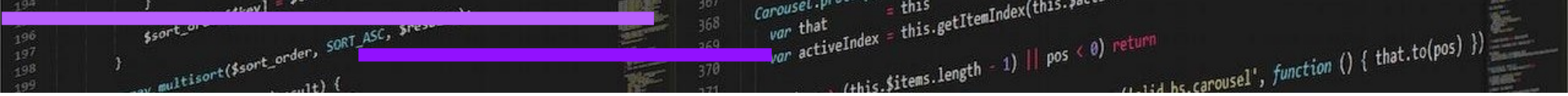



Fonte: SOMMERVILLE, 2011.



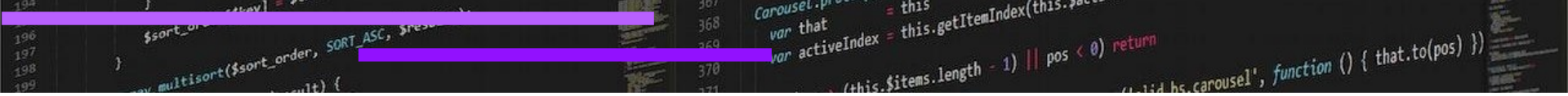
# Por que testar software?

1. **Construir um produto de qualidade**: assegura que ele atende às especificações e expectativas, entregando um produto confiável ao usuário.
2. **Redução de custos**: inicialmente exige tempo e recurso, mas 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**: facilita a identificação e correção de erros antecipadamente, acelerando o ciclo de desenvolvimento e lançamentos.



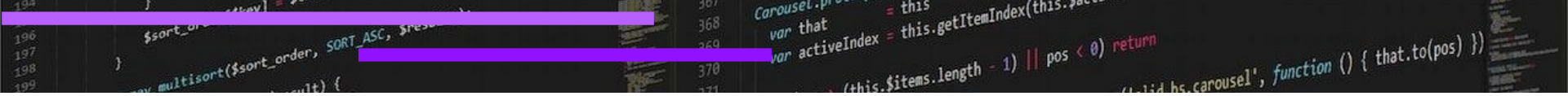
# Por que testar software?

4. **Documentação**: alguém que está tentando entender um pedaço de código pode olhar para os testes para entender o que o código deve fazer.
5. **Melhorar a colaboração**: facilita a colaboração e a revisão de código entre os desenvolvedores. Ajuda a entender a intenção do código e garantir que as alterações não quebrem a funcionalidade.
6. ***Feedback* rápido**: fornece *feedback* rápido sobre a saúde do software. Isso permite que os desenvolvedores corrijam erros e *bugs* mais cedo no ciclo de desenvolvimento, o que pode economizar tempo e recursos.



# Por que testar software?

7. **Satisfação do cliente**: menos *bugs* e problemas para os usuários finais resultam em uma maior satisfação do cliente.
8. **Integração contínua/Entrega contínua (CI/CD)**: permite que as alterações sejam verificadas automaticamente, facilitando a integração e entrega de alterações de código de maneira eficiente e confiável.
9. **TDD – *Test Driven Development***: metodologia que coloca os testes no centro do processo de desenvolvimento. Antes de escrever o código, os desenvolvedores primeiro escrevem um teste, então eles escrevem o código para passar o teste, e finalmente refatoram o código para padrões aceitáveis.



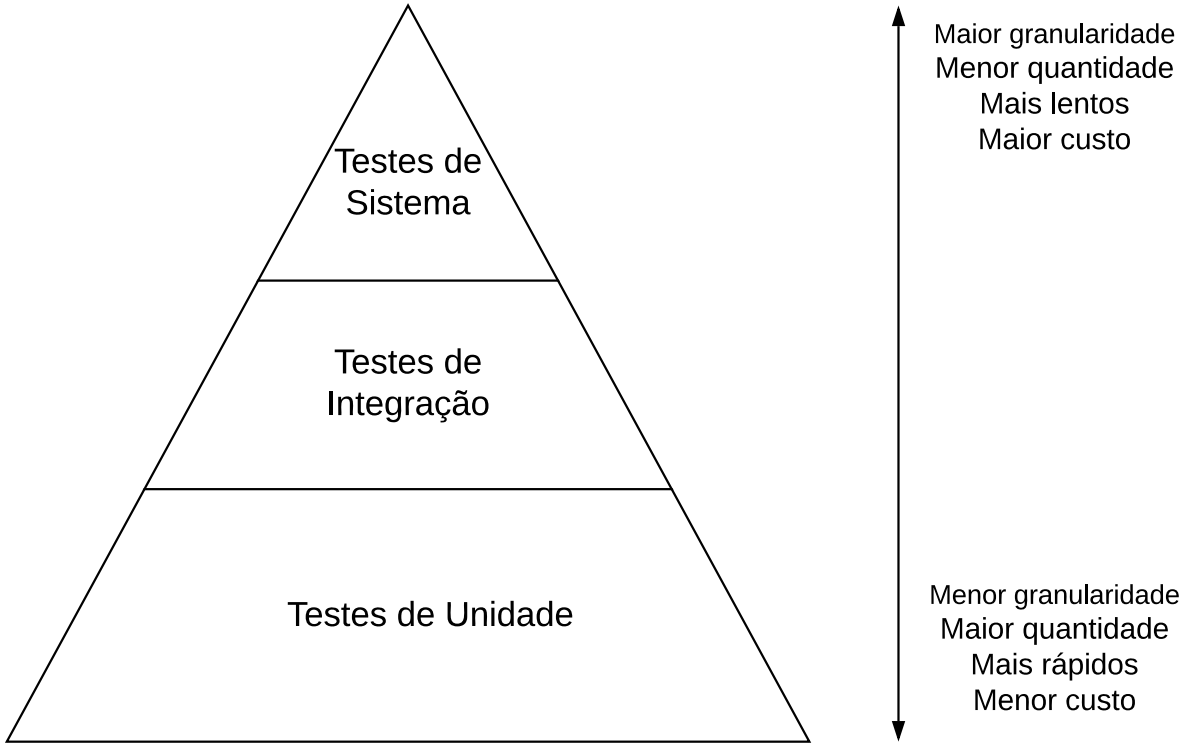
# Limites dos testes

- 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 software.

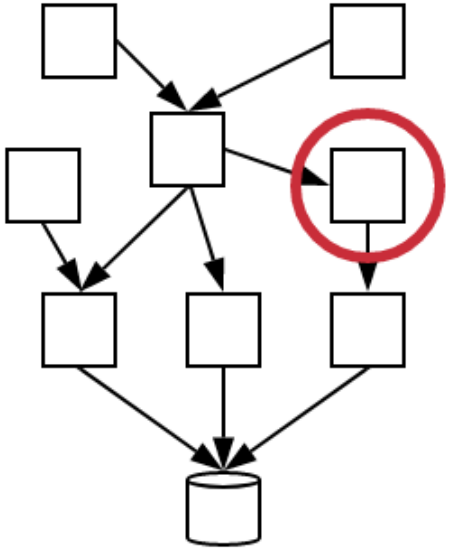
***“Os testes podem mostrar apenas a presença de erros, e não sua ausência”. (Edsger Dijkstra)***

# Taxonomia

- Classificação de testes por nível de acordo com sua granularidade

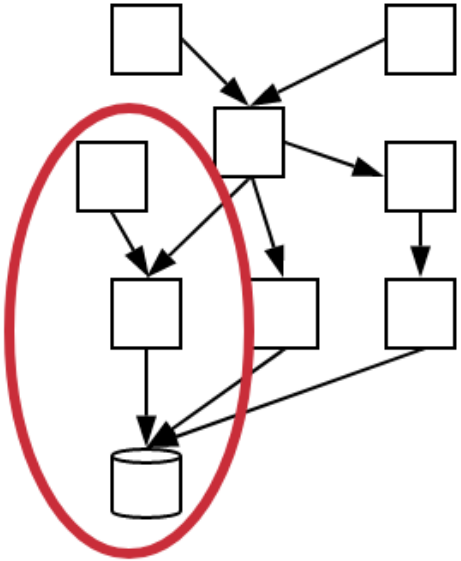


# Classificação de testes por nível



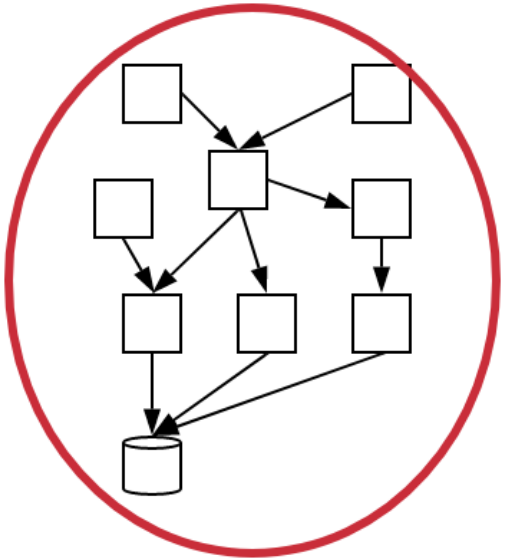
**Teste de Unidade**

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



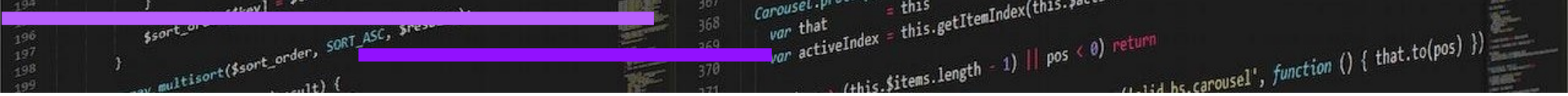
**Teste de Integração**

- Verificação das interações entre os componentes (internos e externos).



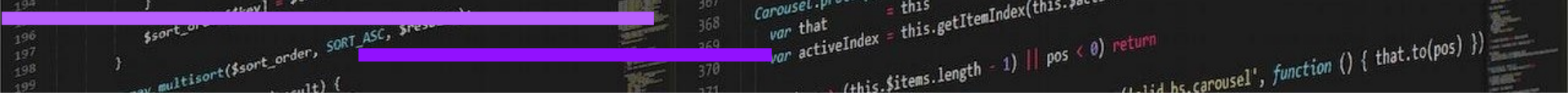
**Teste de Sistema (*end-to-end*)**

- Simula uma seção de uso do sistema por um usuário real



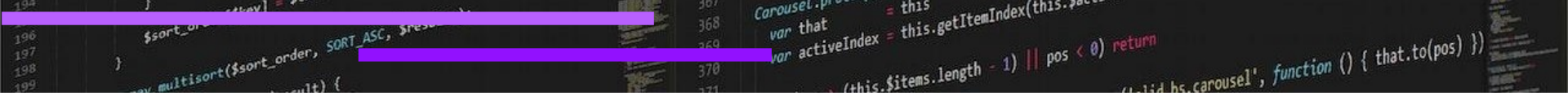
# Automatização de testes

- 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 software em desenvolvimento é testado.



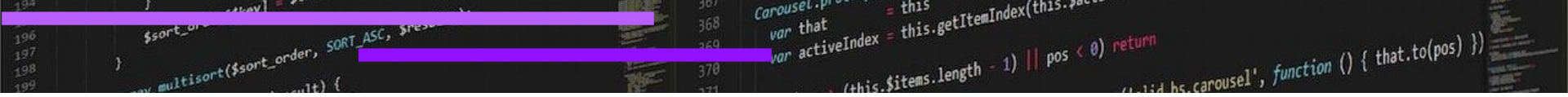
# Testes de Unidade

- São **testes automatizados** de pequenas unidades de código, normalmente **classes**, as quais **são testadas de forma isolada** do restante do sistema.
- Resumidamente é um programa que **chama métodos de uma classe e verifica se eles retornam os resultados esperados**.
- Assim, quando se usa testes de unidades, o código de um sistema pode ser dividido em dois grupos: um conjunto de classes — que implementam os requisitos do sistema — e um conjunto de testes.



# Testes de Integração

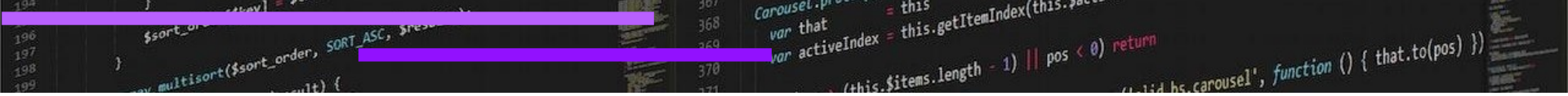
- São testes automatizados que envolvem a execução de métodos que exigem a **integração entre mais de uma unidade/camada/classe do software**.
- Na prática, o teste de integração vai ser aquele que vai testar funcionalidades que precisam **acessar o banco de dados, escrever/ler dados para/de um arquivo, invocar um serviço na rede**, etc.
- Testes de integração são **mais lentos** que testes de unidade.

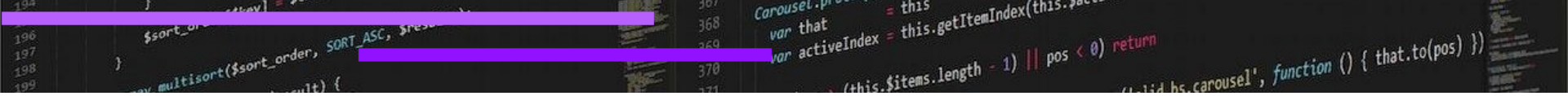


# Testes de Sistema

- São testes automatizados que envolvem o teste do **software como um todo** (considerando suas operações globais, geralmente acessadas via interface gráfica ou API)
- Podem ser chamados também de **testes *end-to-end***
- Podem envolver o teste de propriedades como:
  - Desempenho
  - Segurança
  - Disponibilidade

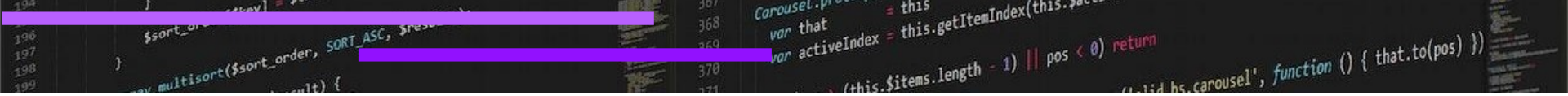
# Frameworks





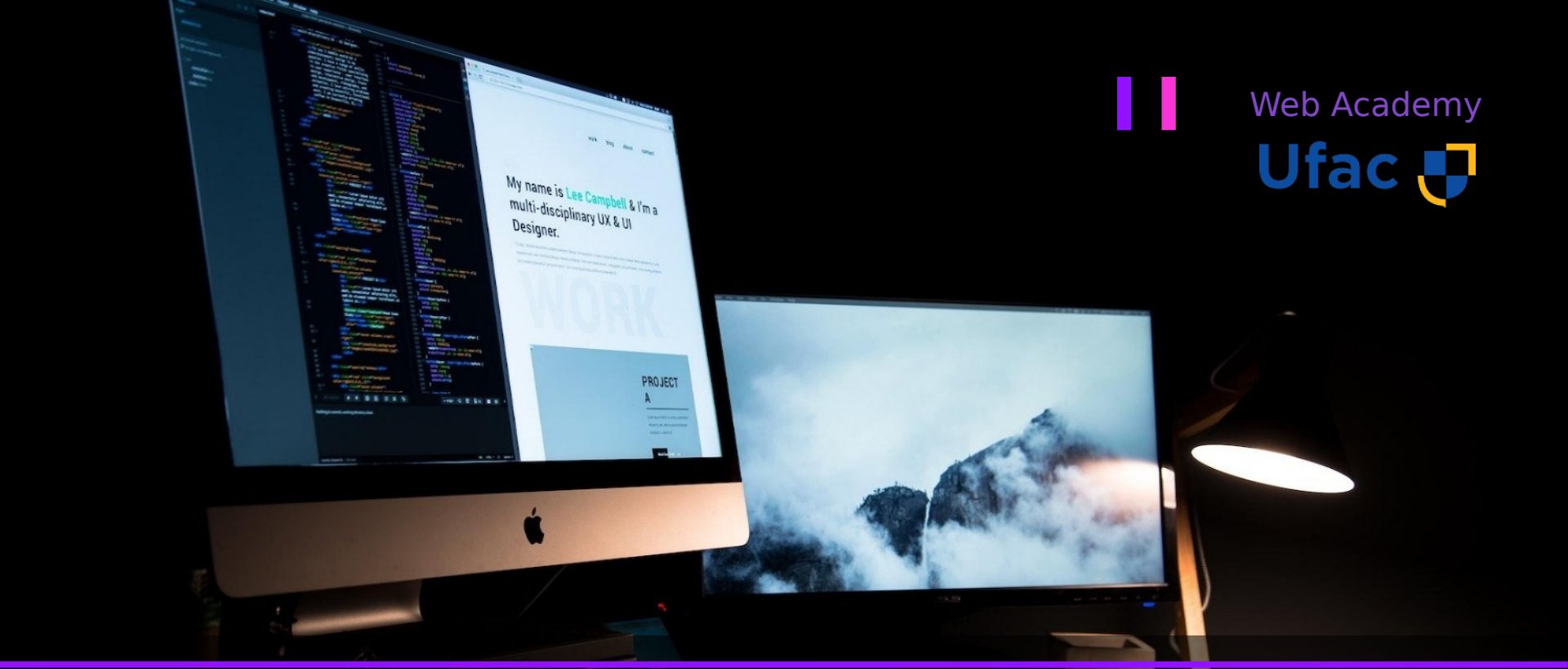
# Qual o defeito no método lastZero?

```
public class Example {
    // Se a == null, throw NullPointerException
    // Senão retorna o índice do último ZERO
    // Retorna -1 se não houver ocorrências de ZERO
    public static int lastZero(int[] a) {
        for(int i = 0; i < a.length; i++)
            if (a[i] == 0)
                return i;
        return -1;
    }
}
//test: a=[0,1,0] expected=2
```



# Qual o defeito no método lastZero?

```
public class Example {
    // Se a == null, throw NullPointerException
    // Senão retorna o índice do último ZERO
    // Retorna -1 se não houver ocorrências de ZERO
    public static int lastZero(int[] a) {
        int x = -1;
        for(int i = 0; i < a.length; i++)
            if (a[i] == 0)
                x = i;
        return x;
    }
}
//test: a=[0,1,0] expected=2
```



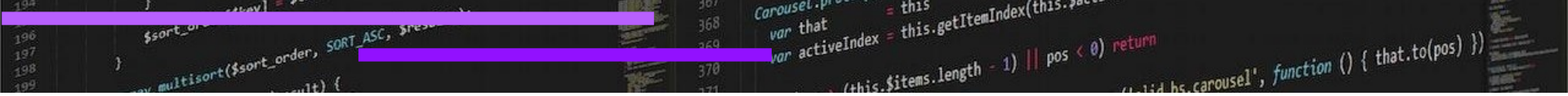

Web Academy

Ufac

# Testes Automatizados com JUnit

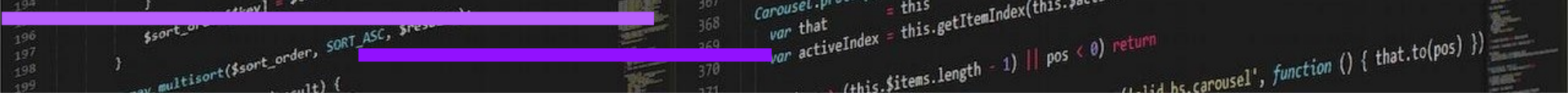
# Test Driven Development – TDD

- **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 pelas metodologias ágeis como Scrum e E**X**treme **P**rogramming (XP).



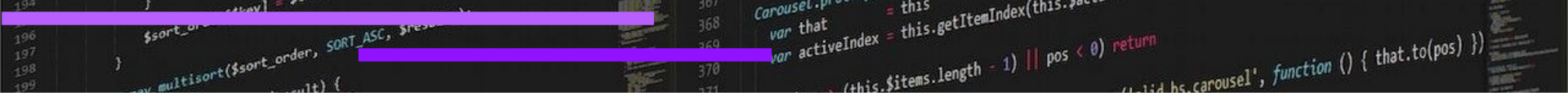
# Quando Escrever Testes de Unidade?

- Após implementar **uma pequena funcionalidade**.
- Pode-se escrever **antes de qualquer código**. Os testes vão falhar, em seguida implementa-se o código e testa-se novamente (*Test-Driven Development* – TDD).
  - **Quando um usuário reportar um bug**, escrevemos um teste que reproduz o *bug* e que, portanto, vai falhar. No passo seguinte, corrigimos o *bug*.
  - **Quando estiver depurando um código**, evite escrever um *println* para testar o resultado de um método. O teste tem a vantagem de contribuir para a suíte de testes.
- Não é recomendável:
  - Implementar **todos os testes após o sistema ficar pronto**. Isso pode produzir testes com baixa qualidade ou nem serem implementados.
  - **Outro time ou empresa** de desenvolvimento implementar os testes.



# *Test Driven Development* – TDD

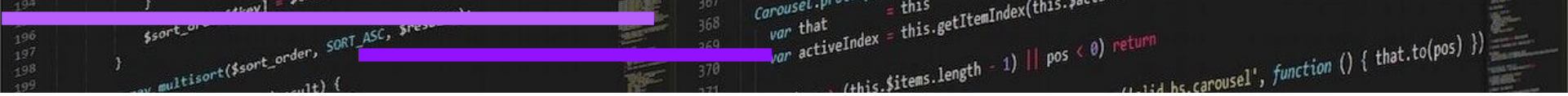
- TDD **evita que os desenvolvedores esqueçam de escrever testes**.
  - Para isso, TDD promove testes à primeira atividade de qualquer tarefa de programação, seja ela corrigir um *bug* ou implementar uma nova funcionalidade.
- TDD **favorece a escrita de código com alta testabilidade** (acima de 90%).
  - É uma consequência da inversão do fluxo de trabalho: o desenvolvedor sabe que terá que escrever o teste T e depois a classe C, é natural que planeje C de forma a facilitar a escrita de T.
- TDD **é uma prática relacionada com a melhoria do *design* do software**.
  - Ao começar pela escrita de um teste T, o desenvolvedor coloca-se na posição de um usuário da classe C. O primeiro usuário da classe é seu próprio desenvolvedor — lembre que T é um cliente de C, pois ele chama métodos de C.



# JUnit

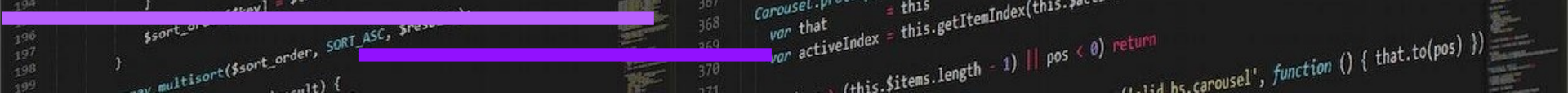
- É um *framework open-source* para **construção e execução de testes unitários automatizados em Java**, que verifica as funcionalidades de classes e seus métodos.
- O JUnit **funciona com base em anotações** (*Annotations*) que indicam se um método é de teste ou não e **asserções** (*Asserts*) que verificam o resultado atual de um método com o resultado esperado.
- O JUnit 5 é composto por 3 módulos distintos:
  - **JUnit Platform:** atua como a base para a execução dos testes, oferecendo suporte para diferentes ambientes de execução;
  - **JUnit Jupiter:** traz novas funcionalidades, como anotações poderosas e recursos de parametrização; e
  - **JUnit Vintage:** possibilita a execução de testes legados.

JUnit 5



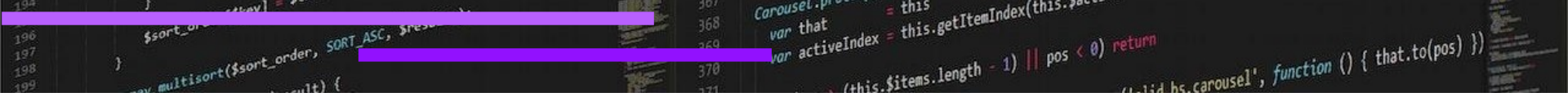
# JUnit – Anotações

- **@Test**
  - Usada para anotar os métodos para serem executados como um teste;
- **@BeforeAll**
  - Método com essa anotação:
    - Deve ser um método estático; e
    - Será executado uma vez antes de qualquer teste ser executado;
- **@BeforeEach**
  - O método será executado uma vez antes de cada método de teste anotado com @Test;
- **@AfterAll**
  - Semelhante ao @BeforeAll, mas será executado após a execução de todos os testes;
- **@AfterEach**
  - Semelhante ao @BeforeEach, mas será executado uma vez após a execução de cada teste.



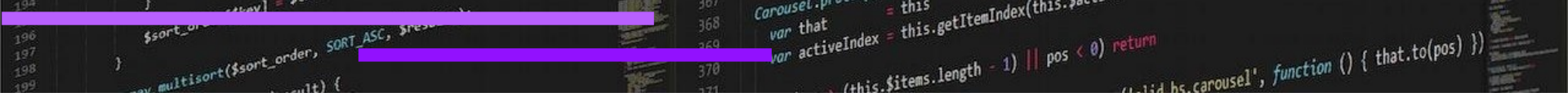
# JUnit – Asserções

- **`assertTrue`**`(boolean condition)`
  - Verifica se a condição é verdadeira.
- **`assertFalse`**`(boolean condition)`
  - Verifica se a condição é falsa.
- **`assertEquals`**`(expected, actual)`
  - Testa se dois valores (esperado e atual) são os mesmos. No caso de arrays, a verificação é em relação à referência e não ao conteúdo.
- **`assertNull`**`(object)`
  - Verifica se o objeto é nulo.
- **`assertNotNull`**`(object)`
  - Verifica se o objeto não é nulo.
- **`assertSame`**`(expected, actual)`
  - Verifica se ambas as variáveis se referem ao mesmo objeto.
- **`assertNotSame`**`(expected, actual)`
  - Verifica se ambas as variáveis se referem a objetos diferentes.
- **`assertThrows`**`(Class<T> expectedType, Executable executable)`
  - Verifica se a execução retorna a exceção esperada.



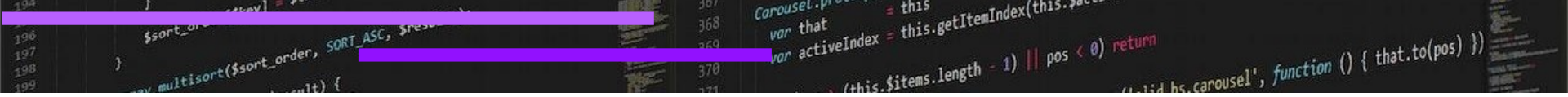
# Testes de Unidade com JUnit

- Por convenção, **classes de teste têm o mesmo nome** das classes testadas, mas **com um sufixo Test**, por exemplo, CalculadoraTest.
- Os **métodos de teste começam com o prefixo test,** por exemplo, public void testSoma(), e devem, obrigatoriamente, atender às seguintes condições:
  1) Serem públicos, pois eles serão chamados pelo JUnit;
  2) Não possuírem parâmetros;
  3) Possuírem a anotação @Test, a qual identifica métodos que deverão ser executados durante um teste.



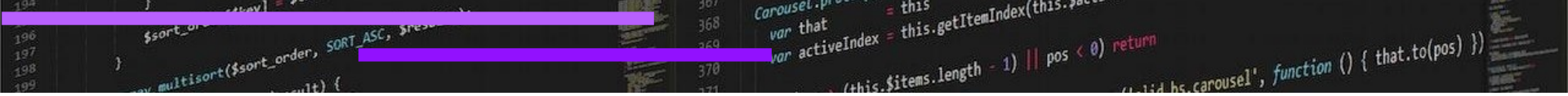
# Exemplo 1 de Testes de Unidade com JUnit

- Implementar uma classe com 4 operações (métodos) para uma calculadora simples:
  - Soma
  - Subtração
  - Multiplicação
  - Divisão
- Escrever uma classe de Testes para verificar os métodos implementados



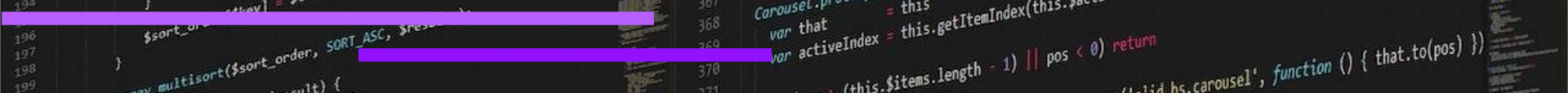
# Exemplo 1 de Testes de Unidade com JUnit

```
public class Calculadora {
  public double soma(double a, double b) {
    return a + b;
  }
  public double somaDecimal(double a, double b) {
    return BigDecimal.valueOf(a).add(BigDecimal.valueOf(a)).doubleValue();
  }
  public double sub(double a, double b) {
    return a - b;
  }
  public double mul(double a, double b) {
    return a * b;
  }
  public double div(double a, double b) {
    if (b == 0)
      throw new ArithmeticException();
    return a / b;
  }
}
```



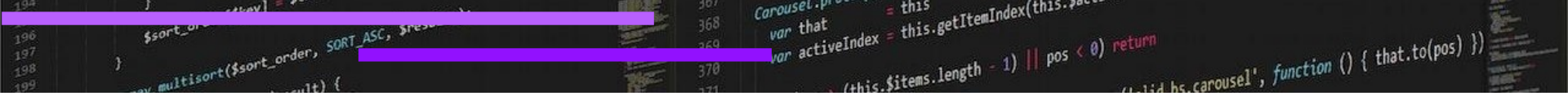
# Exemplo 1 de Testes de Unidade com JUnit

```java
public class CalculadoraTest {
  Calculadora c;
  @Before // Criar o contexto (fixture)
  public void setUp() {
     c = new Calculadora();
  }
  @Test // Testa o método soma
  public void testSoma() {
     // Chama o método e armazena o resultado
     double soma = c.soma(0.8, 0.02);
     // Testa o resultado
     assertEquals(0.82, soma, 0.000000000000001);
  }
  @Test // Testa o método somaDecimal
  public void testSomaDecimal() {
     // Chama o método e armazena o resultado
     double somaDecimal = c.somaDecimal(0.8, 0.02);
     // Testa o resultado
     assertEquals(0.82, somaDecimal, 0.000000000000001);
  }
  @Test // Testa o método sub
  public void testSub() {
     // Chama o método e armazena o resultado
     double sub = c.sub(3, 2);
     // Testa o resultado
     assertEquals(0.82, sub, 0.01);
  }
  @Test // Testa o método div
  public void testDiv() {
     // Chama o método e armazena o resultado
     double div = c.div(5, 2);
     // Testa o resultado
     assertEquals(2.5, div, 0.01);
  }
  // Testa a exceção do método div
  public void testDivZero() {
     // Testa o resultado
     assertThrows(ArithmeticException.class, ()-> c.div(5, 0));
  }

}
```



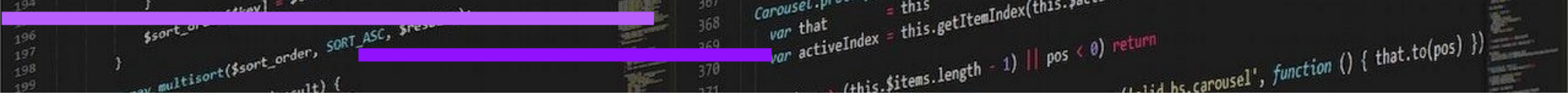
# Exemplo 2 de Testes de Unidade com JUnit

- Implementar uma classe com 4 operações (métodos) para uma estrutura de dados do tipo **Pilha** (FILO – *First-In*, *Last-Out*):
  - Retornar o tamanho da pilha
  - Verificar se a pilha está vazia
  - Empilhar um elemento de qualquer tipo
  - Desempilhar um elemento de qualquer tipo
- Escrever uma classe de Testes para verificar os métodos implementados



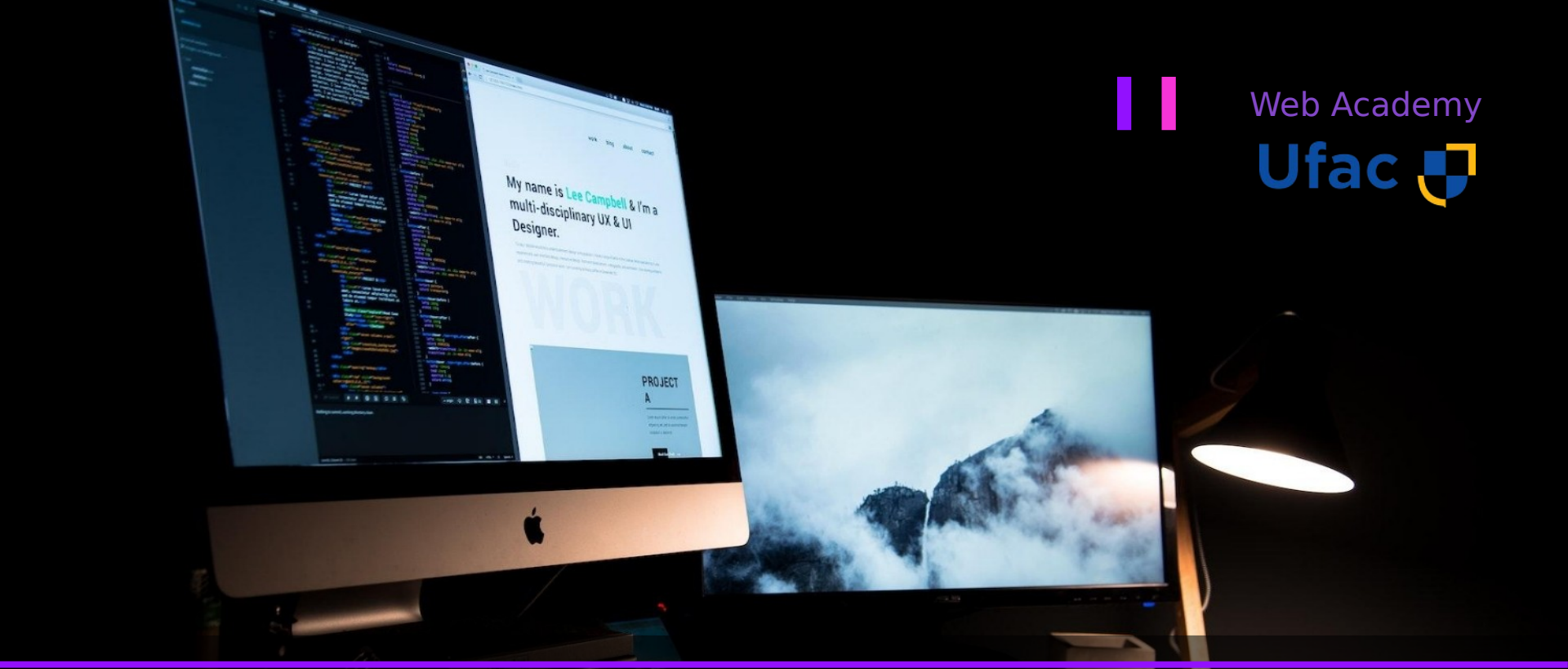
# Exemplo de Testes de Unidade com JUnit

```
public class Pilha<T> {
  private List<T> elementos = new ArrayList<T>();
  private int tam = 0;
  public boolean isEmpty() {
    return (tam == 0);
  }
  public int size() {
    return tam;
  }
  public void push(T item) {
    elementos.add(item);
    tam++;
  }
  public T pop() throws EmptyStackException {
    if (isEmpty())
      throw new EmptyStackException();
    T item = elementos.remove(tam - 1);
    tam--;
    return item;
  }
}
```



# Exemplo de Testes de Unidade com JUnit

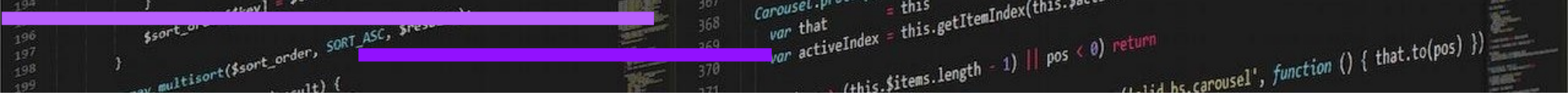
```java
public class PilhaTest {
  Pilha<Integer> pilha;
  @BeforeEach // Criar o contexto (fixture)
  public void setUp() {
    pilha = new Pilha<>();
  }
  @Test // Testa se a pilha está vazia
  public void testEmptyPilha() {
    // Chama o método e armazena o resultado
    boolean status = pilha.isEmpty();
    // Testa o resultado
    assertTrue(status);
  }
  @Test // Testa se a pilha não está vazia
  public void testNotEmptyPilha() {
    pilha.push(100);
    boolean status = pilha.isEmpty();
    assertFalse(status);
  }
  @Test // Testa o empilhamento e o desempilhamento
  public void testPushPopPilha() {
    pilha.push(100);
    pilha.push(200);
    int result = pilha.pop();
    assertEquals(200, result);
  }
  @Test // Testa o tamanho da pilha
  public void testSizePilha() {
    pilha.push(100);
    pilha.push(200);
    int result = pilha.size();
    assertEquals(2, result);
  }
  // Testa a exceção de pilha vazia
  @Test
  public void testEmptyPilhaException() {
    pilha.push(100);
    int result = pilha.pop();
    assertThrows(EmptyStackException.class, pilha::pop);
  }
}
```




Web Academy

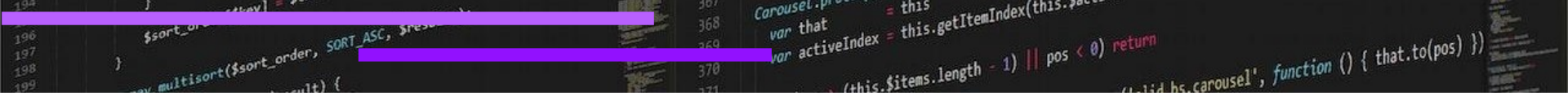
Ufac

# Testes Back-end



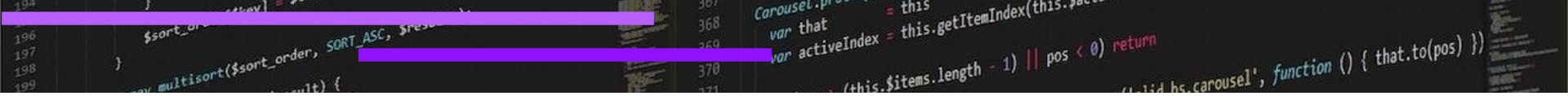
# Testes em aplicações Spring Boot

- 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 os frameworks **JUnit**, **Mockito** e várias outras bibliotecas úteis.



# Testes de Unidade

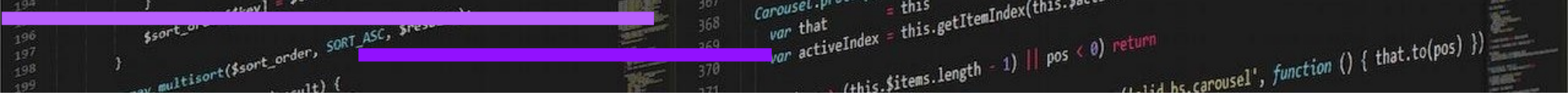
- No contexto de uma API, testes de unidade garantem que cada camada da aplicação funcione corretamente de forma isolada.
  - Camadas: **modelo**, **serviço** e **controle**.
  - Apesar de ser possível testes de unidade na **camada de repositório de dados**, faz mais sentido que sejam testes de integração.
  - 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**.



# Exemplo de testes na camada de modelo

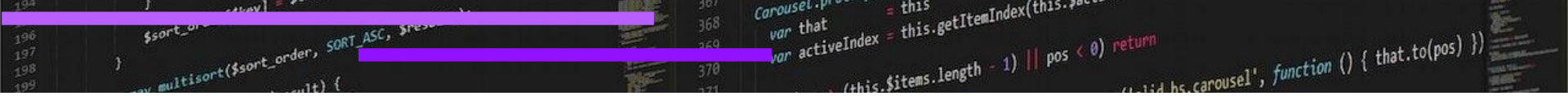
```java
public class AtendimentoTest {
    private Atendimento atendimento;
    @BeforeEach
    public void setUp() {
        atendimento = new Atendimento();
    }

    @Test
    public void testAtendimentoId() {
        Long id = 1L;
        atendimento.setId(id);
        assertEquals(1L, atendimento.getId());
    }
}
```

```java
public class EStatusTest {
    @Test
    void testProximo() {
        EStatus cancelado = EStatus.CANCELADO;
        EStatus agendado = EStatus.AGENDADO;
        EStatus confirmado = EStatus.CONFIRMADO;
        EStatus chegada = EStatus.CHEGADA;
        EStatus atendimento = EStatus.ATENDIMENTO;
        EStatus encerrado = EStatus.ENCERRADO;
        assertEquals(cancelado, cancelado.proximo());
        assertEquals(confirmado, agendado.proximo());
        assertEquals(chegada, confirmado.proximo());
        assertEquals(atendimento, chegada.proximo());
        assertEquals(encerrado, atendimento.proximo());
        assertEquals(encerrado, encerrado.proximo());
    }
}
```



# Testes 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.

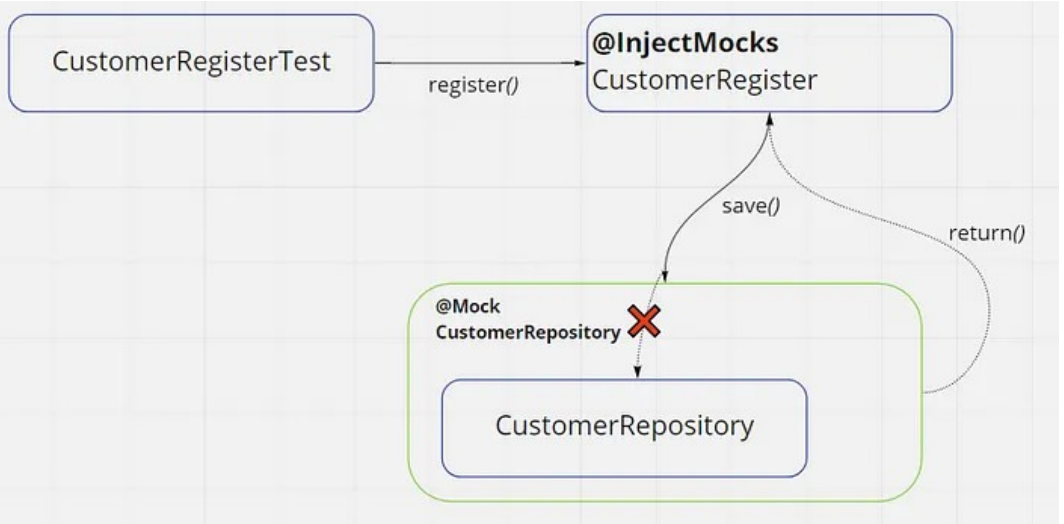


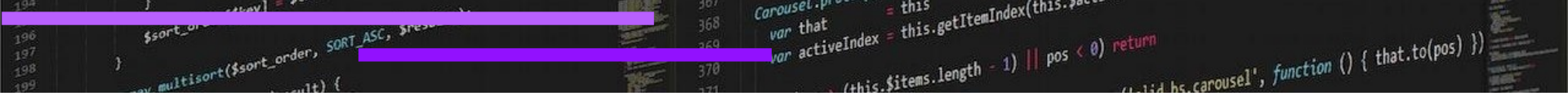
# Mocks

- 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.

# Mocks e Mockito

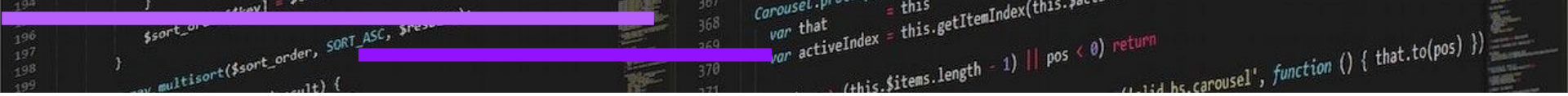
- O Mockito é um framework de testes unitários para Java e o seu principal objetivo é **simular instâncias de classes e comportamentos de métodos**.
- Isso é chamado de mock, na tradução livre quer dizer “zombar”. Ao mockar a dependência de uma classe, a classe testada pensa estar invocando o método realmente, mas de fato não está.





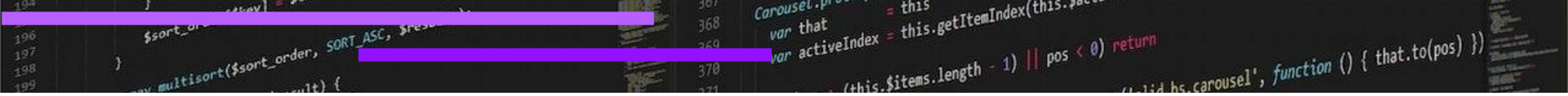
# Exemplo de testes na camada de serviço

```java
@ExtendWith(MockitoExtension.class)
public class AtendimentoServiceTest {
    @Mock
    private AtendimentoRepository repo;
    @InjectMocks
    private AtendimentoService servico;
    private Atendimento atendimento;
    private List<Atendimento> atendimentos;
    @BeforeEach
    public void setUp() {
        Atendimento atendimento1 = new Atendimento();
        atendimento1.setId(1L);
        atendimentos = new ArrayList<>();
        atendimentos.add(atendimento1);
    }
    @Test
    public void testAtendimentoGetAll() {
        Mockito.when(repo.findAll()).thenReturn(atendimentos);
        List<Atendimento> result = servico.get();
        assertEquals(2, result.size());
        assertEquals(1L, result.get(0).getId());
    }
    @Test
    public void testAtendimentoUpdateStatus() {
     Mockito.when(repo.findById(1L)).thenReturn(Optional.of(atendimento));
     Atendimento result = servico.updateStatus(1L);
     assertNotNull(result);
     assertEquals(EStatus.CONFIRMADO, result.getStatus());
    }
}
```



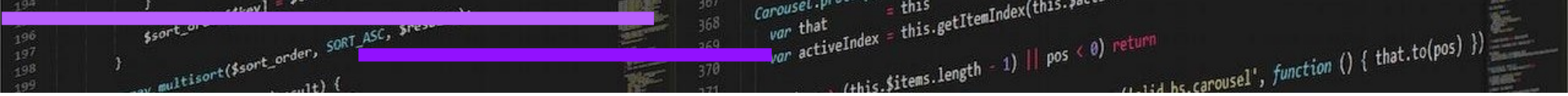
# Exemplo de testes na camada de controle

```
@WebMvcTest(AtendimentoController.class)
public class AtendimentoControllerTest {
   @MockBean
   private AtendimentoService servico;
   @Autowired
   private MockMvc mock;
   private Atendimento atendimento;
   private String jsonContent;
   @BeforeEach
   private void setUp() throws JsonProcessingException {
       atendimento = new Atendimento();
       atendimento.setId(1L);
       jsonContent = new ObjectMapper().writeValueAsString(atendimento);
   }
   @Test
   void testInsert() throws Exception {
       Mockito.when(servico.save(any(Atendimento.class))).thenReturn(atendimento);
       mock.perform(MockMvcRequestBuilders.post("/atendimento/").contentType(MediaType.APPLICATION_JSON)
               .content(jsonContent)).andExpect(MockMvcResultMatchers.status().isCreated())
               .andExpect(MockMvcResultMatchers.jsonPath("$.id", Matchers.is(1)));
   }
}
```



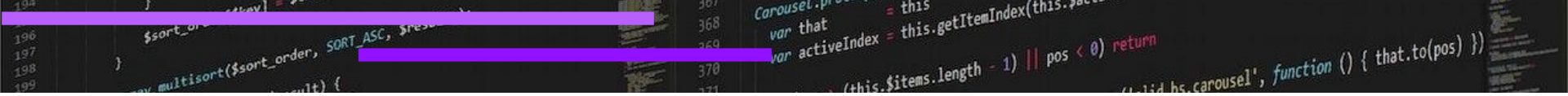
# Testes de integração

- 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 de integração na camada Repository

```
@DataJpaTest
public class AtendimentoRepositoryIntegrationTest {
 @Autowired
 private AtendimentoRepository repo;
 @Test
 public void testAtendimentoGetProfissional() {
   String termo = "Maria";
   List<Atendimento> atendimentos = repo.busca(termo);
   assertEquals(2, atendimentos.size());
   assertTrue(atendimentos.get(0).getProfissional().getNome().startsWith(termo));
 }
}
```



# Testes de integração na camada Controller

```
@SpringBootTest
@AutoConfigureTestDatabase
@AutoConfigureMockMvc
public class AtendimentoControllerIntegrationTest {
  @Autowired
  private MockMvc mockMvc;
  @Test
  public void testAtendimentoGetAll() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get("/atendimento/"))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$", Matchers.hasSize(5)))
        .andExpect(MockMvcResultMatchers.jsonPath("$[0].convenio.ativo", Matchers.is(true)));
  }
}
```