

**MBAUSP**  
**ESALQ**

# Gerenciamento de Testes (TDD)

Prof. Helder Prado Santos

# MBAUSP ESALA

A responsabilidade pela idoneidade, originalidade e licitude dos conteúdos didáticos apresentados é do professor.

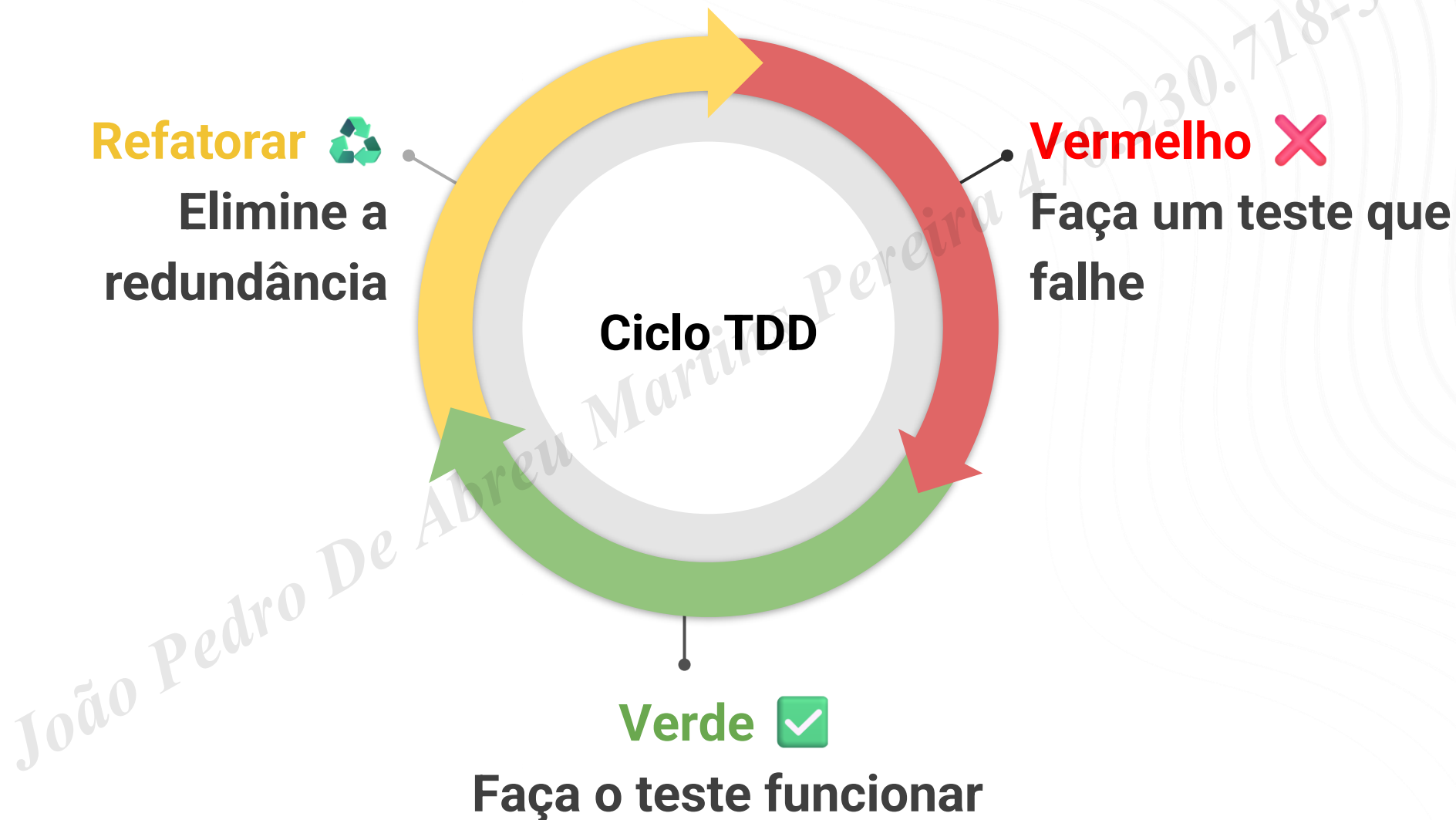
**Proibida a reprodução**, total ou parcial, sem autorização.

Lei nº 9610/98

# Desenvolvimento orientado a testes (TDD)



# Ciclo de vida do TDD



# Objetivos e Impactos

## ❑ Objetivos do TDD:

- Reduzir a incerteza e a ansiedade no desenvolvimento de software.
- Aumentar a qualidade e a confiabilidade do código desde as etapas iniciais.
- Promover um desenvolvimento mais ágil e seguro.

## ❑ Impacto do Medo no Trabalho:

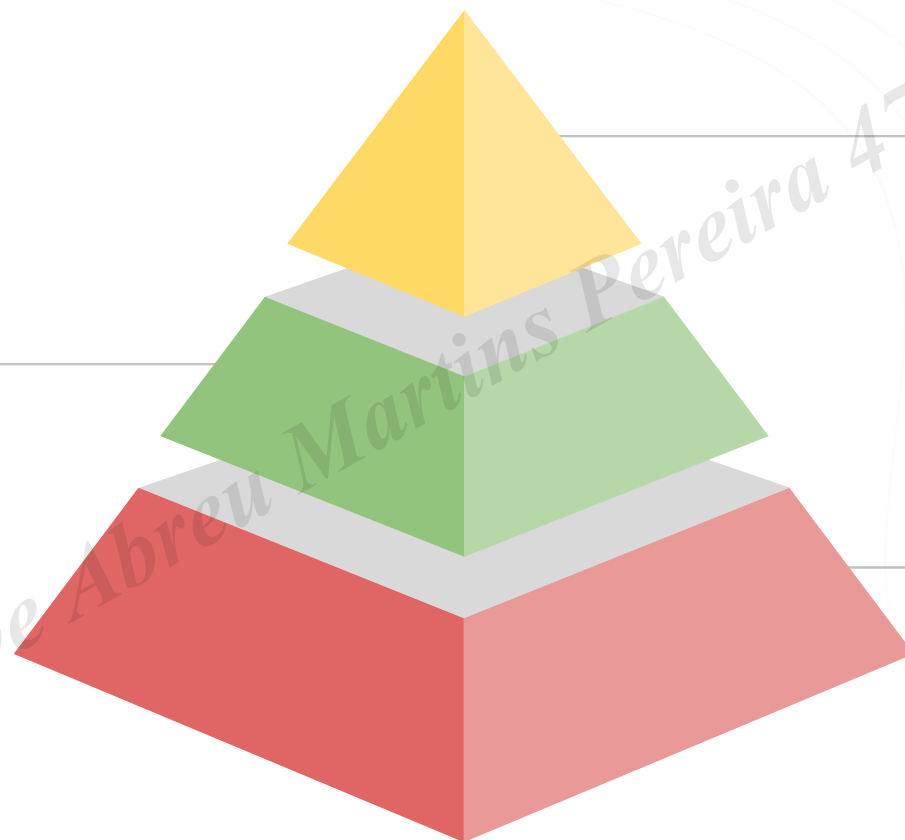
- O medo em excesso leva a programadores hesitantes, com menor proatividade e menos abertura para a colaboração.
- Pode afetar a motivação, causando frustração e bloqueando o aprendizado e a criatividade.

# Pirâmide de testes



**Integração**  
Verificam se diferentes módulos ou componentes do sistema funcionam corretamente quando integrados entre si.

2



1

## End to End

Simulam o comportamento do usuário em um fluxo completo do sistema, desde o início até o final de uma operação.

## Unitário

Verificam o funcionamento de componentes individuais do código, como funções, métodos ou classes, de forma isolada.

3



Quantidade

# Testes Unitários

- ❑ **O que são:** Os testes unitários verificam o funcionamento de componentes individuais do código, como funções, métodos ou classes, de forma isolada.
- ❑ **Objetivo:** Garantir que cada unidade de código funcione corretamente de maneira independente.
- ❑ **Foco:** Testar uma única parte do código, sem dependência de outras partes ou de sistemas externos, como bancos de dados.
- ❑ **Exemplo:** Testar os métodos de uma classe, como a validação dos seus atributos

# Testes Unitários

## ❑ Vantagens:

- Rapidez na execução, pois testam apenas pequenas partes do código.
- Facilidade de identificação de onde está o problema quando um teste falha, já que cada teste é focado em um único componente.

## ❑ Ferramentas Comuns:

- JUnit (Java)
- NUnit (.NET)
- Jest (JavaScript)
- PyTest (Python)



# Testes de Integração

- ❑ **O que são:** Os testes de integração verificam se diferentes módulos ou componentes do sistema funcionam corretamente quando integrados entre si.
- ❑ **Objetivo:** Garantir que a interação entre módulos de código (por exemplo, interação entre entidades diferentes, ou entre uma aplicação e seu banco de dados) ocorre como esperado.
- ❑ **Foco:** Testar o funcionamento do sistema em conjunto, geralmente incluindo as dependências entre componentes.
- ❑ **Exemplo:** Testar a funcionalidade onde duas entidades interagem.

# Testes de Integração

## ❑ Vantagens:

- Identificação de problemas de integração que testes unitários não conseguem cobrir, como erros de comunicação entre módulos.
- Útil para verificar a compatibilidade entre diferentes partes do sistema que precisam operar juntas.

## ❑ Ferramentas Comuns:

- Spring Test (Java)
- DBUnit (para testar banco de dados)
- Mockito (para simulação de dependências).

# Testes de end-to-end (E2E)

- ❑ **O que são:** Testes end-to-end (E2E) verificam o funcionamento de uma aplicação em um fluxo completo, como um todo, para assegurar que os componentes integrados fornecem a resposta esperada ao cliente.
- ❑ **Objetivo:** Garantir que o sistema funcione como um todo, validando o comportamento do software do ponto de vista do usuário final.
- ❑ **Foco:** Testar a aplicação inteira, simulando como o sistema responde a operações reais.
- ❑ **Exemplo:** Testar um endpoint de login, enviando uma requisição de autenticação de usuário e verificando se o sistema responde com sucesso.

# Testes End-to-End (E2E)

## ❑ Vantagens:

- Confiança de que o sistema está pronto para o usuário final, pois cobre o fluxo completo.
- Capacidade de identificar problemas de integração entre sistemas e fluxos complexos que dependem de múltiplos serviços e componentes.

## ❑ Ferramentas Comuns:

- Postman
- Cypress
- Swagger

# Ciclo de Vida de um Teste

- Planejamento e especificação dos testes;
- Desenvolvimento e implementação dos testes;
- Execução e registro dos resultados;
- Análise e acompanhamento de falhas;

João Pedro De Abreu Martins Pereira 470.230.718-57

# Planejamento e Estratégia de Testes

- Definição dos critérios de aceitação e cobertura de testes;
- Planejamento de casos de teste e escopo dos testes;
- Riscos e critérios de priorização de testes;
- Alocação de recursos e estimativa de esforço.

João Pedro De Abreu Martins Pereira 470.230.718-57

# Boas Práticas no TDD

- Manter testes independentes e atômicos;
- Nomeação clara e descritiva dos testes;
- Evitar dependências e acoplamento excessivo nos testes;
- Refatorar frequentemente para evitar "test code smells";

João Pedro De Abreu Martins Pereira 470.230.718-57

# Cobertura de Testes e Métricas

## ❏ O Que é Cobertura de Testes?

- Mede o quanto do código é executado durante a execução dos testes.
- Indicador essencial para avaliar a eficácia e a amplitude dos testes.
- Ajuda a identificar áreas não cobertas que podem conter defeitos.

João Pedro De Abreu Martins

170.230.718-57



# Métricas de Qualidade para Testes e Análise de Resultados

- ❑ **Taxa de Cobertura:** Percentual de cobertura considerado adequado para a aplicação.
- ❑ **Métricas de Defeitos:** Número e tipo de defeitos encontrados em áreas cobertas e não cobertas.
- ❑ **Indicadores de Efetividade:** Relação entre cobertura e defeitos, mostrando a precisão dos testes em identificar erros.
- ❑ **Tendência ao Longo do Tempo:** Análise da evolução da cobertura e qualidade dos testes em cada release.

# Ferramentas para Análise de Cobertura de Código

- ❑ **Jacoco** (Java), **Istanbul** (JavaScript), **Coverage.py** (Python);
- ❑ **SonarQube:** Para análise contínua da qualidade do código e integração com ferramentas de CI/CD;
- ❑ **Codecov e Coveralls:** Para visualização e monitoramento de cobertura de código em projetos de código aberto e CI/CD.

# Gerenciamento de Defeitos

- Processo de identificação e categorização de defeitos;
- Ciclo de vida de um defeito: Detecção, análise, correção e verificação;
- Ferramentas de gerenciamento de defeitos (ex.: Jira, Bugzilla);
- Relacionamento entre defeitos e a cobertura de testes;

João Pedro De Abreu Martins Pereira 470.230.718-57

# Testes como Documentação do Sistema


## ❑ Testes como Fonte de Documentação Viva

- Testes, especialmente os automatizados, oferecem uma visão clara de como o sistema deve se comportar em diferentes cenários.
- Servem como documentação atualizada automaticamente: sempre que o código é alterado, os testes refletem essas mudanças.

# Vantagens dos Testes como Documentação

- ❑ **Especificação Executável:** Os testes mostram como cada funcionalidade deve operar, substituindo descrições manuais.
- ❑ **Atualização Contínua:** Cada execução de teste verifica e mantém a documentação funcional do sistema.
- ❑ **Entendimento do Comportamento do Sistema:** Testes unitários, de integração e end-to-end explicam a lógica e interações dos componentes em situações reais.

# Exemplo do ciclo TDD




```
1  import pytest
2  from uuid import uuid4
3
4  class TestUser:
5
6      # Teste para inicialização do usuário
7      def test_user_initialization(self):
8          user_id = uuid4()
9          user_name = "Test User"
10         user = User(id=user_id, name=user_name)
11
12         assert user.id == user_id
13         assert user.name == user_name
14
```

# Exemplo do ciclo TDD

```
1  from uuid import UUID
2
3  class User:
4
5      id: UUID
6      name: str
7
8      def __init__(self, id: UUID, name: str):
9          self.id = id
10         self.name = name
```

# Exemplo do ciclo TDD

```
1  import pytest
2  from uuid import uuid4
3  from domain.user.user_entity import User
4
5
6  class TestUser:
7
8      # Teste para inicialização do usuário
9      def test_user_initialization(self):
10         user_id = uuid4()
11         user_name = "Test User"
12         user = User(id=user_id, name=user_name)
13
14         assert user.id == user_id
15         assert user.name == user_name
16
```





# Exemplo do ciclo TDD

```
1  import pytest
2  from uuid import uuid4
3  from domain.user.user_entity import User
4
5  class TestUser:
6
7      # Teste para inicialização do usuário
8      def test_user_initialization(self):
9          user_id = uuid4()
10         user_name = "Test User"
11         user = User(id=user_id, name=user_name)
12
13         assert user.id == user_id
14         assert user.name == user_name
15
16     # Teste para validação do ID do usuário
17     def test_user_id_validation(self):
18         with pytest.raises(Exception, match="id must be an UUID"):
19             User(id="invalid_id", name="Test User")
20
21     # Teste para validação do nome do usuário
22     def test_user_name_validation(self):
23         user_id = uuid4()
24         with pytest.raises(Exception, match="name is required"):
25             User(id=user_id, name="")
```

# Exemplo do ciclo TDD

```
1  from uuid import UUID
2
3  class User:
4
5      id: UUID
6      name: str
7
8      def __init__(self, id: UUID, name: str):
9          self.id = id
10         self.name = name
11         self.validate()
12
13     def validate(self):
14         if not isinstance(self.id, UUID):
15             raise Exception("id must be an UUID")
16
17         if not isinstance(self.name, str) or len(self.name) == 0:
18             raise Exception("name is required")
```

# Exemplo do ciclo TDD

```
1  import pytest
2  from uuid import uuid4
3  from domain.user.user_entity import User
4
5  class TestUser:
6
7      # Teste para inicialização do usuário
8      def test_user_initialization(self):
9          user_id = uuid4()
10         user_name = "Test User"
11         user = User(id=user_id, name=user_name)
12
13         assert user.id == user_id
14         assert user.name == user_name
15
16     # Teste para validação do ID do usuário
17     def test_user_id_validation(self):
18         with pytest.raises(Exception, match="id must be an UUID"):
19             User(id="invalid_id", name="Test User")
20
21     # Teste para validação do nome do usuário
22     def test_user_name_validation(self):
23         user_id = uuid4()
24         with pytest.raises(Exception, match="name is required"):
25             User(id=user_id, name="")
```



# MBAUSP ESALQ

## Obrigado(a)!

[Helder Prado | linkedin.com/in/helderprado](https://www.linkedin.com/in/helderprado)