Aula 5

Metodologias Ágeis

Prof. Manoel Flavio Leal

1

Conversa Inicial

2

Estudaremos nesta aula

- **TDD – *Test Driven Development***
- **Principais conceitos e técnicas**
  - **Criação de testes antes da implementação**
  - **Complexidade incremental no *design* de *software***
  - **Abordagem *red-green-refactoring***

Profit_Image/Shutterstock

3

Objetivos

- **Compreender os fundamentos e a relevância do TDD**
- **Dominar a criação de testes antes da implementação**
- **Saber aplicar *red-green-factory***

4

Por que criar um teste antes da implementação do código

5

Relembrando

- **TDD – *Test Driven Development***
- **Melhorar a qualidade do código e reduzir *bugs* no processo de desenvolvimento de *software***
- **Escrever testes antes do código para identificar erros desde o início e garantir a correção do código**

6

Benefícios do TDD

- **Maior qualidade do produto**
- **Maior segurança durante mudanças no *software***
- **Testes automatizados executados repetidamente, proporcionando *feedback* rápido**
- **Detecção antecipada de problemas no código em desenvolvimento**

Paul Craft/Shutterstock

7

TDD x Abordagem Tradicional

Aspectos	TDD	Abordagem Tradicional
Sequência de desenvolvimento	Testes escritos antes da implementação do código	Código implementado primeiro; testes escritos posteriormente
Foco no resultado	Testes representam requisitos e comportamentos esperados	Maior enfoque em correção de erros específicos
Ciclo de feedback	Feedback instantâneo por meio de testes automatizados	Feedback obtido após a implementação
Orientação ao usuário	Testes representam a perspectiva do usuário final	Perspectiva do usuário abordada posteriormente
Colaboração e comunicação	Promove a colaboração e a comunicação na equipe de desenvolvimento	Comunicação pode ser menos intensa durante o processo de desenvolvimento dos testes

8

Pesquisas comprovam

- **Sistemas criados com TDD geram código de melhor qualidade e com menos falhas**
- **TDD é uma estratégia poderosa para desenvolver *softwares* confiáveis e orientados às necessidades dos usuários**

Alexdndz/Shutterstock

9

- **Criar um teste antes da implementação do código traz benefícios significativos para o processo de desenvolvimento de *software***
- **TDD aprimora a qualidade, a segurança e a colaboração na equipe de desenvolvimento**
- **Prática essencial para criar *software* de alta qualidade e confiável**

10

Ciclos de refinamento contínuo

11

Ciclo de refinamento contínuo no TDD

- **Tem como objetivo melhorar a qualidade do código e do *software* de forma iterativa**
- **Composto por etapas sequenciais e repetitivas**
- **Cada ciclo consiste na criação de um teste automatizado antes da implementação do código**

12

Ciclo de TDD – passos

- Escreva um teste "falhando"
- Escreva o código mínimo para fazer o teste passar
- Refatore o código para torná-lo mais legível e fácil de manter
- Repita os passos até que todos os testes passem

13

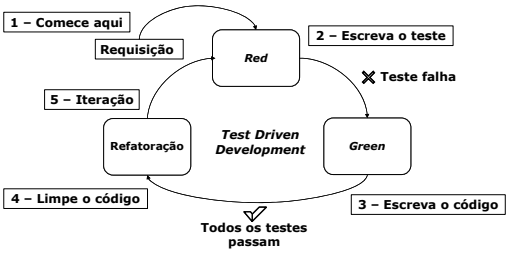
Red-Green-Refactoring

- Técnica utilizada para garantir que o código seja funcional e bem projetado
- Fase Vermelha (*Red*)
  - Escrever um caso de teste que falhe
- Fase Verde (*Green*)
  - Escrever a quantidade mínima de código necessária para fazer o teste passar
- Fase de Refatoração (*Refactoring*)
  - Melhorar o *design* do código, garantindo que todos os testes continuem passando

Oakview Studios/Shutterstock

14

Red-Green-Refactoring em ação



15

A importância da iteração rápida e contínua

- *Feedback* contínuo sobre o código produzido
- Identificação precoce de erros e falhas, permitindo correções rápidas e eficientes
- Código mais legível e bem estruturado com criação de testes antes da implementação

16

A importância da iteração rápida e contínua

- Flexibilidade para responder a mudanças nos requisitos ou nas necessidades do cliente
- Demonstração de progresso e qualidade do trabalho, promovendo confiança e transparência na equipe e nos *stakeholders*

Marish/Shutterstock

17

- Ciclos de refinamento contínuo do TDD aprimoram a qualidade do código
- Criação de testes antes da implementação garante integridade e valida funcionalidades
- Implementação orientada pelos testes resulta em código seguro e confiável
- Refatoração contínua melhora legibilidade e manutenibilidade, eliminando duplicações

18

Como desenvolver um teste

19

Visão geral

- **Desenvolver um teste em TDD envolve escrever os testes antes da implementação do código**
- **O objetivo é criar funcionalidades de forma assertiva e com qualidade, melhorando a manutenibilidade e a confiabilidade do sistema**

20

Modelo F.I.R.S.T

- **F - *Fast* (Rápido): rápido para permitir execuções frequentes e identificação ágil de problemas**
- **I - *Isolated* (Isolado): independente, sem depender de outros para evitar inconsistências**
- **R - *Repeatable* (Repetível): produção dos mesmos resultados sempre que executados**
- **S - *Self-validating* (Autovalidação): verifica automaticamente se o resultado é o esperado**
- **T - *Timely* (Oportuno): escrito antes do código de produção, garantindo uma base clara e orientada para o desenvolvimento**

21

Definição de casos de teste e cenários

- **Essencial no TDD, envolvendo identificar cenários e comportamentos do sistema**
- **Cada caso de teste é específico, descrevendo a ação, condições iniciais e resultado esperado**
- **A técnica GWT (*given-when-then*) pode ser utilizada para estruturar e comunicar claramente o cenário de teste**
- **Casos de teste são a base para criação de testes automatizados, refletindo o comportamento esperado do código**

22

Técnica GWT

- ***Given* (Dado): descreve condições iniciais e prepara o contexto do teste**
- ***When* (Quando): representa a ação ou evento específico a ser testado**
- ***Then* (Então): especifica o resultado esperado após a ação, verificando se o sistema se comportou como esperado**

23

Exemplo usando a técnica GWT

CASO DE TESTE
Dado que usuário está logado no sistema
Quando clica no botão "Enviar"
Então o sistema deve exibir uma mensagem de confirmação
E o sistema deve salvar os dados enviados no banco de dados

- ***Given*: usuário está logado no sistema**
- ***When*: ação de clicar no botão "Enviar"**
- ***Then*: a exibição da mensagem de confirmação e a persistência dos dados**

24

Escrevendo testes automatizados

- **Testes automatizados seguem uma estrutura com etapas de configuração, preparação, execução, verificação e limpeza**
- **Recomenda-se uma cobertura abrangente para testar diferentes cenários e comportamentos do sistema**

25

Escrevendo testes automatizados

- ***Frameworks* populares, como JUnit, NUnit, pytest e Jasmine, facilitam a escrita e a execução de testes automatizados**
- **Testes eficazes em TDD garantem a qualidade do *software* e promovem um desenvolvimento ágil e confiável**

26

- **O TDD oferece uma abordagem assertiva e orientada à qualidade no desenvolvimento de *software***
- **Desenvolver testes antes da implementação ajuda a identificar erros precocemente e melhora a manutenibilidade do código**
- **O modelo F.I.R.S.T. garante testes rápidos, isolados, repetíveis, autovalidáveis e oportunos**
- **Testes automatizados são fundamentais para garantir a integridade e a confiabilidade do sistema, resultando em um *software* de alta qualidade**

27

Teste legível, isolado, minucioso e explícito

28

Teste legível

- **Facilmente compreensível por qualquer pessoa**
- **Nomes de métodos e variáveis descritos e claros**
- **Código organizado de maneira lógica e fácil de seguir**
- **Importante para identificar rapidamente problemas ou falhas**

29

Exemplo de teste legível

```
def test_calculate_total_amount():

    # Configuração
    cart = ShoppingCart()
    cart.add_item(Item("Product 1", 10.99))
    cart.add_item(Item("Product 2", 15.99))

    # Execução
    total_amount = cart.calculate_total_amount()

    # Verificação
    assert total_amount == 26.98
```

Python

30

Teste isolado

- Não depende de outras partes do sistema
- Evita dependências externas, como bancos de dados e chamadas de API
- Mais rápido, confiável e fácil de depurar
- Foco na unidade testada, sem interações complexas

31

Exemplo de teste isolado

```
@Test
public void testCalculateDiscountedPrice() {
    // Configuração
    Product product = new Product("ABC123", "Product 1", 50.0);
    DiscountCalculator calculator = new DiscountCalculator();

    // Execução
    double discountedPrice = calculator.calculateDiscountedPrice(product, 0.1);

    // Verificação
    assertEquals(45.0, discountedPrice, 0.001);
}
```

Fatmawati Achmad Zaenuri/Shutterstock

32

Teste minucioso

- Todos os comportamentos e funcionalidades relevantes são testados
- Garantia de que cada parte do código é executada e validada
- Teste de entradas válidas, inválidas e limites de valores
- Verificação de comportamento em situações de erro

33

Exemplo de teste minucioso

```
def test_is_valid_email():
    # Caso de sucesso
    assert is_valid_email("example@email.com") == True

    # Casos de falha
    assert is_valid_email("exampleemail.com") == False
    assert is_valid_email("example@.com") == False
    assert is_valid_email("@email.com") == False
    assert is_valid_email("example@") == False
```

Fatmawati Achmad Zaenuri/Shutterstock

34

Teste explícito

- Define o que está sendo testado e o resultado esperado
- Estabelece as condições iniciais necessárias para o teste
- Realiza a ação ou operação a ser testada
- Compara o resultado real com o esperado
- Fornece informações claras em caso de falha

35

Exemplo de teste explícito

```
test("calculateTax should correctly calculate the tax amount", () => {
    // Configuração
    const price = 100;
    const taxRate = 0.1;

    // Execução
    const taxAmount = calculateTax(price, taxRate);

    // Verificação
    expect(taxAmount).toEqual(10);
});
```

Fatmawati Achmad Zaenuri/Shutterstock

36

- A combinação de testes é essencial para qualidade e conformidade
- Integração de testes é prioridade no desenvolvimento de *software* de alta qualidade
- Prática de testes é garantia de sucesso e melhoria contínua

37

Complexidade incremental e *design* incremental

38

Complexidade incremental

- É a prática de adicionar novos recursos e funcionalidades de forma gradual e controlada ao longo do processo de desenvolvimento

39

Complexidade incremental no TDD

- Escrita de testes como requisitos

```
def test_soma():
    calculadora = Calculadora()
    resultado = calculadora.soma(2, 3)
    assert resultado == 5
```

- O teste "test_soma" representa o requisito de que a função soma() da calculadora deve retornar corretamente a soma de dois números

40

Complexidade incremental no TDD

- Implementação mínima

```
class Calculadora:
    def soma(self, a, b):
        return a + b
```

- Apenas realiza a soma dos dois números passados como argumento

41

Complexidade incremental no TDD

- Teste e validação contínuos

```
def test_soma():
    calculadora = Calculadora()
    resultado = calculadora.soma(2, 3)
    assert resultado == 5
```

- Se o teste passar, significa que a implementação está correta. Em caso contrário, o teste falhará, indicando que algo precisa ser corrigido

42

Complexidade incremental no TDD

- **Refatoração**

```
class Calculadora:
    def soma(self, a, b):
        resultado = a + b
        return resultado
```

- **Nesse exemplo, a refatoração consiste em armazenar o resultado da soma em uma variável antes de retorná-lo**

43

Complexidade incremental no TDD

- **Repetição do ciclo**

```
def test_subtracao():
    calculadora = Calculadora()
    resultado = calculadora.subtracao(5, 2)
    assert resultado == 3
```

- **Você implementa novos requisitos. A função subtracao() com a implementação mínima, executa os testes, faz a validação contínua e, se necessário, realiza a refatoração**

44

Design incremental

- **É uma prática que permite que o *design* do sistema evolua gradualmente, em resposta aos requisitos em constante mudança, enquanto mantém um foco contínuo na validação e teste do *software***

45

Aplicação do *design* incremental

- **Pequenos incrementos**
- **Testes como requisitos**
- **Refatoração contínua**
- **Validação contínua**
- ***Feedback* e aprendizado**

46

Benefícios do *design* incremental

- **Código mais limpo e modular**
- **Melhor compreensão dos requisitos**
- ***Feedback* rápido**
- **Maior confiança na qualidade do código**
- **Flexibilidade para mudanças**

Vectorhot/Shutterstock

47

- **Código que atende aos requisitos desde o início, evitando erros futuros**
- **Melhoria gradual do código por meio de ciclos iterativos**
- **Testes eficazes: legíveis, isolados, minuciosos e explícitos para maior compreensão e abrangência**
- ***Design* gradual para evitar complexidade excessiva e facilitar a manutenção**

48