




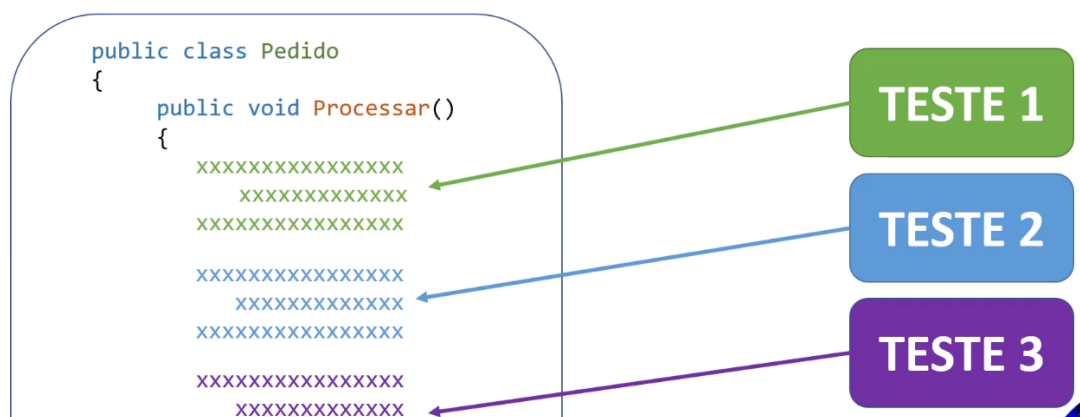


Notas sobre Testes de Unidade

Type	Info
Topics	 <u>Studies</u>
Notebooks	 <u>Programação Profissional com Clean Code</u>
Related Notes	 <u>Ementa do Curso</u>

Princípios do TDD

- Testes de unidade garante que cada parte do sistema funcione corretamente de forma isolada; ajudam a identificar erros precocemente durante o desenvolvimento; facilitam a refatoração do código
- Test Driven Development (TDD)
 - Escrever um teste e assistir ele falhar
 - Escrever o mínimo de código para passar no teste
 - Refatorar, melhorar e testar novamente



- 3 leis do TDD
 - Escrever código de produção até escrever um teste que falhe

- Guiar o desenvolvimento; define a funcionalidade; cada pedaço de código relacionado a um requisito funcional
- Escrever um teste que falhe por vez
 - Reduz a complexidade e escopo de cada ciclo de desenvolvimento
- Não escrever mais código de produção do que o necessário para passar no teste
 - Evita código que não esteja relacionado à funcionalidade

```
public class TestesTDD
{
    // Primeira Lei: Não escreva código de produção até
    // que você tenha escrito um teste que falhe.
    // Tenha claro o conceito do que é "assistir" o test
    // e falhar!
    [Fact]
    public void TesteFalha_Inicial()
    {
        var resultado = Calculadora.Somar(2, 2);
        Assert.Equal(4, resultado); // Este teste deve f
        alhar inicialmente, pois o código não existe
    }

    // Segunda Lei: Escreva apenas um teste de unidade q
    // ue falhe por vez.
    [Fact]
    public void TesteFalha_Suficiente()
    {
        var resultado = Calculadora.Somar(2, 2);
        Assert.Equal(4, resultado); // Este teste deve p
        assar após correção
    }

    // Terceira Lei: Não escreva mais código de produção
    // do que o necessário para passar no teste atual.
    public class Calculadora
    {

```

```

        public static int Somar(int a, int b)
        {
            return a + b; // Apenas o necessário para pa
ssar o teste
        }
    }
}

```

Práticas de Testes Limpos

- Exemplos
 - Manter testes limpos

```

public class TestesLimpos
{
    private Calculadora calculadora = new Calculadora
();

    // Ruim: Teste desorganizado e sem clareza
    [Fact]
    public void Teste_Calculadora()
    {
        var resultado = calculadora.Somar(2, 2);
        Assert.Equal(4, resultado);

        resultado = calculadora.Subtrair(2, 2);
        Assert.Equal(0, resultado);
    }

    // Bom: Testes claros e organizados
    [Fact]
    public void Teste_Soma()
    {
        var resultado = calculadora.Somar(2, 2);
        Assert.Equal(4, resultado);
    }

    [Fact]

```

```

public void Teste_Subtracao()
{
    var resultado = calculadora.Subtrair(2, 2);
    Assert.Equal(0, resultado);
}
}

```

- Utilizar termos do domínio da aplicação

```

public class TestesDominio
{
    // Ruim: Termos genéricos e sem significado de do
    // mínimo
    [Fact]
    public void Teste_SaldoConta()
    {
        var conta = new Conta();
        conta.Depositar(100);
        Assert.Equal(100, conta.Saldo);
    }

    // Bom: Termos específicos do domínio bancário
    [Fact]
    public void Teste_DepositoConta()
    {
        var conta = new Conta();
        conta.Depositar(100);
        Assert.Equal(100, conta.ObterSaldo());
    }

    public class Conta
    {
        public decimal Saldo { get; private set; }

        public void Depositar(decimal valor)
        {
            Saldo += valor;
        }
    }
}

```

```

        public decimal ObterSaldo()
        {
            return Saldo;
        }
    }
}

```

- Evitar duplicação entre os testes e código

```

public class TestesPadraoDuplo
{
    private Calculadora calculadora = new Calculadora
    ();

    // Ruim: Duplicação de lógica de produção no test
    e
    [Fact]
    public void Teste_Duplicacao()
    {
        var resultado = Somar(2, 2);
        Assert.Equal(4, resultado); // Lógica duplica
da do método de produção

        int Somar(int a, int b)
        {
            return a + b; // Duplicação
        }
    }

    // Bom: Teste simples e sem duplicação
    [Fact]
    public void Teste_SemDuplicacao()
    {
        var resultado = calculadora.Somar(2, 2);
        Assert.Equal(4, resultado);
    }
}

```

- Verificar uma única condição para facilitar a identificação de falhas

```
public class TestesAssertUnico
{
    // Ruim: Múltiplos asserts em um único teste
    [Fact]
    public void Teste_MultiplosAsserts()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
        Assert.Equal(0, calculadora.Subtrair(2, 2));
    }

    // Bom: Um assert por teste
    [Fact]
    public void Teste_Soma_AssertUnico()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }

    [Fact]
    public void Teste_Subtracao_AssertUnico()
    {
        var calculadora = new Calculadora();
        Assert.Equal(0, calculadora.Subtrair(2, 2));
    }
}
```

- Focar em um único conceito a cada teste

```
public class TestesConceitoUnico
{
    // Ruim: Múltiplos conceitos em um único teste
    [Fact]
    public void Teste_MultiplosConceitos()
    {
        var calculadora = new Calculadora();
```

```

        Assert.Equal(4, calculadora.Somar(2, 2));
        Assert.Equal(0, calculadora.Subtrair(2, 2));
        Assert.Equal(6, calculadora.Multiplicar(2,
3));
    }

    // Bom: Um conceito por teste
    [Fact]
    public void Teste_Soma()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }

    [Fact]
    public void Teste_Subtracao()
    {
        var calculadora = new Calculadora();
        Assert.Equal(0, calculadora.Subtrair(2, 2));
    }

    [Fact]
    public void Teste_Multiplicacao()
    {
        var calculadora = new Calculadora();
        Assert.Equal(6, calculadora.Multiplicar(2,
3));
    }
}

```

- Implementar testes Fast, Independent, Repeatable, Self-validating e Timely (FIRST)
 - Fast: executados rapidamente para que rodem frequentemente
 - Independent: os resultados ou estado de cada teste não influenciam outros
 - Repeatable: produzir os mesmos resultados independente do ambiente

- Self-validating: devem ter indicação de sucesso ou falha; compara resultados esperados com reais
- Timely: devem ser escritos antes do código

```
public class TestesFIRST
{
    // Fast (Rápidos)
    // Testes de unidade devem ser executados rapidamente para que possam ser rodados
    // frequentemente, sem atrapalhar o fluxo de trabalho do desenvolvedor.
    [Fact]
    public void Teste_Rapido()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }

    // Independent (Independentes)
    // Cada teste de unidade deve ser independente de outros testes. O resultado de um
    // teste não deve depender do resultado ou do estado gerado por outro teste.
    [Fact]
    public void Teste_Independente()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }

    // Repeatable (Repetíveis)
    // Testes de unidade devem produzir os mesmos resultados toda vez que são executados,
    // independentemente do ambiente em que são rodados.
    [Fact]
    public void Teste_Repetivel()
    {

```



```

        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }

    // Self-validating (Auto-validados)
    // Testes de unidade devem ter uma clara indicação de sucesso ou falha. Isso geralmente
    // é feito através de asserts que verificam se os resultados esperados correspondem aos resultados reais.
    [Fact]
    public void Teste_AutoValidado()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }

    // Timely (Oportunos)
    // Testes de unidade devem ser escritos no momento certo, preferencialmente antes do código de produção
    // correspondente. Isso está alinhado com a prática de Test-Driven Development (TDD).
    [Fact]
    public void Teste_Oportuno()
    {
        var calculadora = new Calculadora();
        Assert.Equal(4, calculadora.Somar(2, 2));
    }
}

```