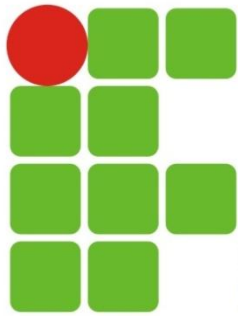


INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
RIO GRANDE DO NORTE

Análise e Projeto Orientados a Objetos

Testes de unidade

Diretoria Acadêmica de Gestão e Tecnologia da Informação

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Introdução

- Por que testar?
  - Para verificar o correto funcionamento do código.
  - Para garantir a qualidade do software.
- Entretanto muitos programadores não testam seus programas como deveriam pois:
  - A execução do teste é tediosa (repetitiva).
  - A elaboração do teste nem sempre é fácil.
  - A interface com o usuário do programa nem sempre colabora para a execução de testes.

Testes Automatizados

- Facilitam a adoção da prática de testes no ambiente de desenvolvimento.
- Dão segurança para alterações de código.
- Essenciais para garantir a qualidade do software.
- Propiciam agilidade na execução dos testes.
- Livra os desenvolvedores do trabalho repetitivo.
- Aumentam a produtividade.

Testes de unidade

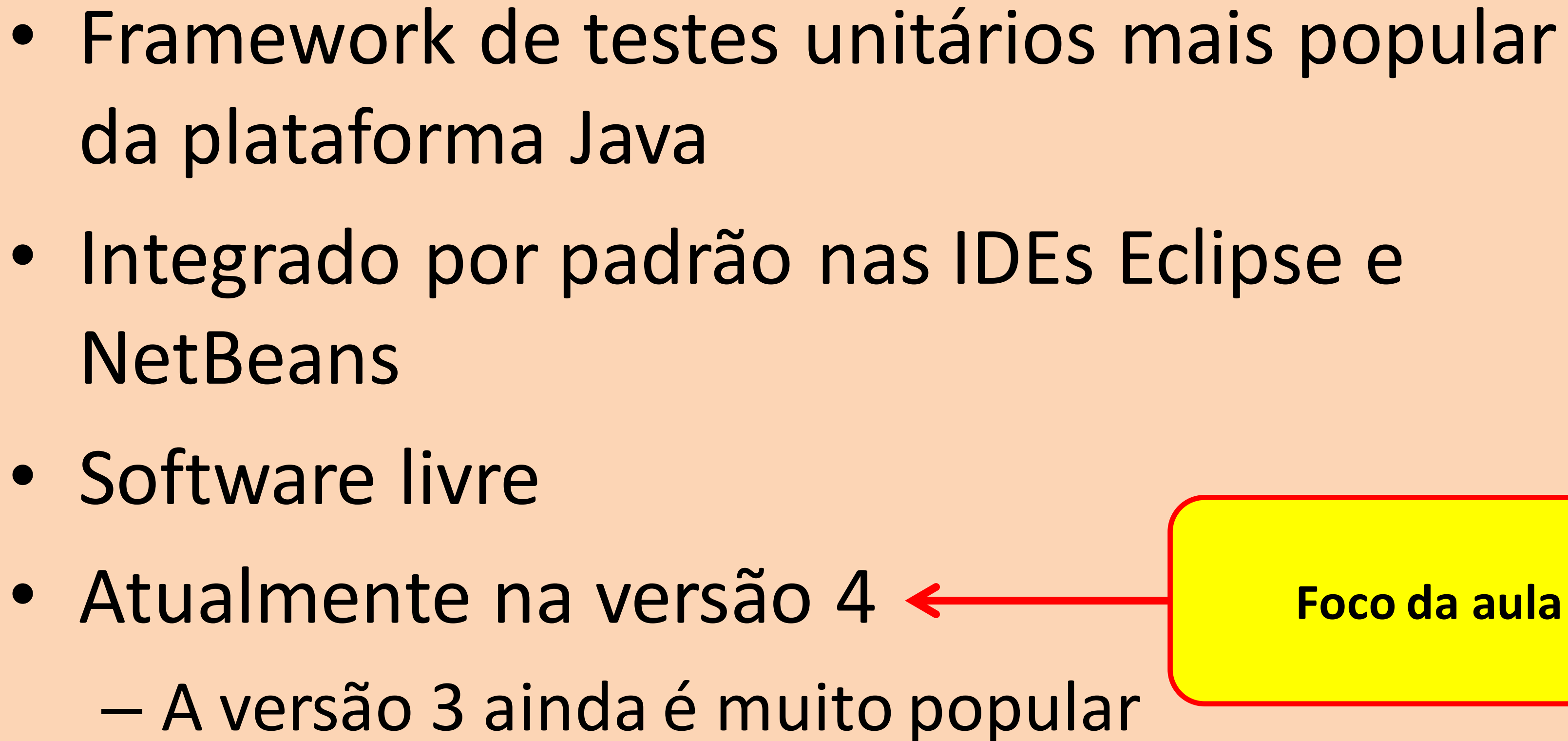
- Têm como objetivo testar, de forma isolada, as menores unidades de software.
  - Em programas OO: classes e métodos
- São escritos pelo programadores.
- Facilitam o processo de depuração.
- Documentam o comportamento esperado.
- Levam à criação de classes mais coesas.
- Geralmente utiliza-se uma ferramenta de apoio para a escrita e execução de testes unitários.

JUnit

- Framework de testes unitários mais popular da plataforma Java.
- Integrado por padrão nas IDEs Eclipse e NetBeans.
- Software livre.
- Atualmente na versão 4
  – A versão 3 ainda é muito popular

JUnit

- Framework de testes unitários mais popular da plataforma Java
- Integrado por padrão nas IDEs Eclipse e NetBeans
- Software livre
- Atualmente na versão 4
  - A versão 3 ainda é muito popular



Exemplo – classe Fração

- Vamos escrever e testar uma classe para representar e operar frações matemáticas.
- Abra o Eclipse e crie um novo projeto do tipo Java Project
  - File > New > Other > Java Project

Adicionando JUnit ao projeto

- Nas propriedades do projeto, escolha a opção **Java Build Path**
- Na guia **Libraries**, pressione o botão **Add Library**
- Na janela **Library**, selecione a opção **JUnit** e então pressione o botão **Next**
- Na janela seguinte, escolha a versão desejada do JUnit e clique em **Finish**
- De volta a janela **Java Build Path**, pressione **OK**

Classe Fração

- Pacote: **fracao**.
- Atributos inteiros, privados e finais (instâncias imutáveis): **numerador** e **denominador**.
- Construtor para inicializar os atributos.
- Métodos de acesso (get) para os atributos
- Restrição: denominador não pode ser zero.
  - Construtor deve lançar **IllegalArgumentException**.

Classe Fração

```
public Fracao(int n, int d){
  if(denominador == 0){
    throw new IllegalArgumentException(
      "Denominador não pode ser igual a zero");
  }
  this.numerador = n;
  this.denominador = denominador;
}
```

Escrevendo testes

```java
package fracao;

import org.junit.Test;

public class FracaoTest {

    @Test
    public void testFracao_1(){
        new Fracao(3, 5);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testFracao_2(){
        new Fracao(5, 0);
    }

}
```

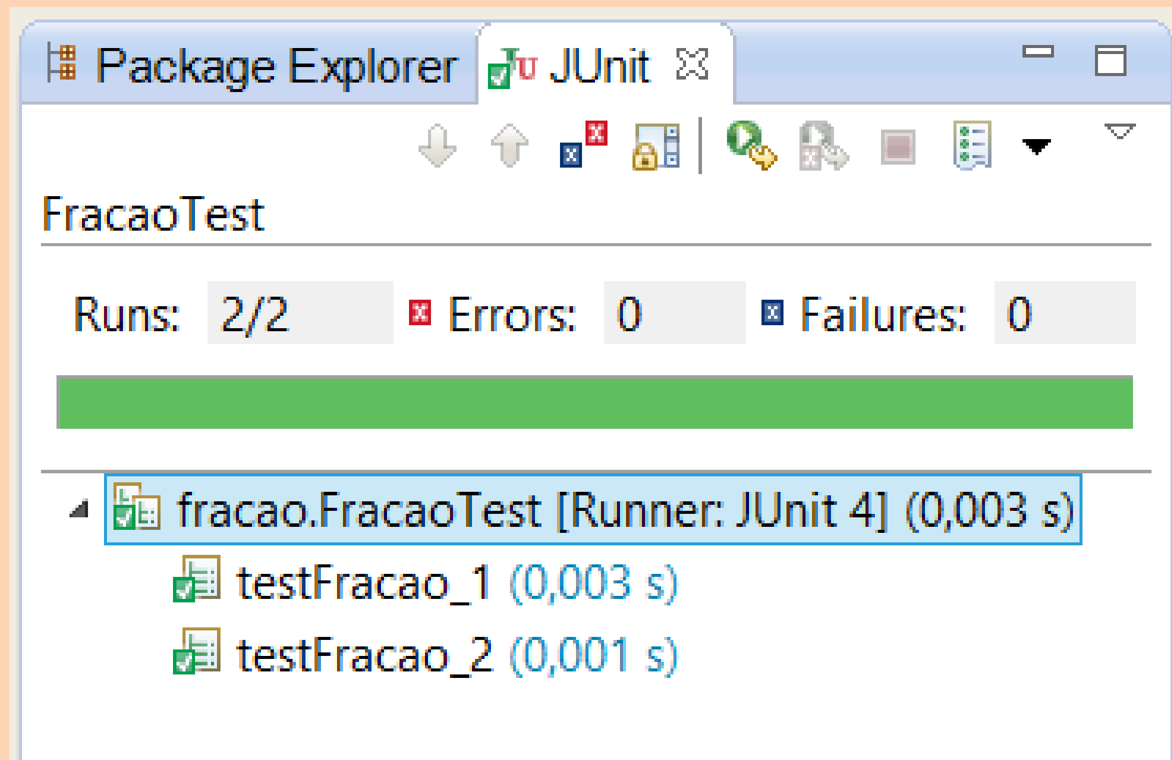
Escrevendo testes

A anotação **@Test** indica ao JUnit de que o método representa um caso de teste.
O atributo **expected** indica que durante a execução do teste, aquela exceção deve ser gerada para que o teste seja bem sucedido.

```
public class FracaoTest {

    @Test
    public void testFracao_1(){
        new Fracao(3, 5);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testFracao_2(){
        new Fracao(5, 0);
    }

}
```

Executando os testes

- Clique com o botão direito sobre a classe **FracaoTest** e selecione a opção **Run As > JUnit Test**



Assertivas

- São métodos auxiliares utilizados para verificar se uma dada condição é verdadeira. Em caso negativo, geram um erro indicando que o teste falhou.
- Devem ser importadas para que possam ser utilizadas.

```
package fracao;

import static org.junit.Assert.*;

import org.junit.Test;

public class FracaoTest {
```

Assertivas

Assertiva/método	Descrição
assertEquals(int esperado, int atual) assertEquals(long esperado, long atual)	Verifica se **esperado** e **atual** são valores iguais.
assertEquals(double esperado, double atual, double delta)	Verifica se a diferença entre **esperado** e **atual** está dentro de **delta**.
assertEquals(Object esperado, Object atual)	Verifica se **esperado** e **atual** são objetos iguais segundo o método **equals**.
assertFalse(boolean condição)	Verifica se **condição** é falsa.
assertTrue(boolean condição)	Verifica se **condição** é verdadeira.

Assertivas

Assertiva/método	Descrição
assertNotNull(Object objeto)	Verifica se **objeto** não é nulo
assertNull(Object objeto)	Verifica se **objeto** é nulo
assertNotSame(Object esperado, Object atual)	Verifica se **esperado** e **atual** não são referências para objetos distintos.
assertSame(Object esperado, Object atual)	Verifica se **esperado** e **atual** são referências para o mesmo objeto.
fail(String mensagem)	Faz com que o teste falhe utilizando uma mensagem informativa.

Método multiplicar

- Deve multiplicar a fração em questão com uma outra fração e retornar o resultado como uma nova instância.

```
public Fracao multiplicar(Fracao f){
  int n = numerador * f.getNumerador();
  int d = denominador * f.getDenominador();
  return new Fracao(n, d);
}
```

Método multiplicar - teste

```java
@Test
public void testMultiplicar(){
    Fracao f1 = new Fracao(1, 2);
    Fracao f2 = new Fracao(3, 5);
    Fracao resultado = f1.multiplicar(f2);
    //testa resultado
    assertEquals(3, resultado.getNumerador());
    assertEquals(10, resultado.getDenominador());
}
```

Facilitando as comparações

- Implementar método **equals** em **Fracao**:

```
@Override
public boolean equals(Object o){
    if(!(o instanceof Fracao))
        return false;
    Fracao f = (Fracao)o;
    return (numerador == f.getNumerador() && denominador == f.getDenominador());
}
```

- Testar método **equals**

```
@Test
public void testEquals(){
    Fracao f1 = new Fracao(1, 2);
    Fracao f2 = new Fracao(1, 2);
    Fracao f3 = new Fracao(3, 5);
    assertEquals(f1, f1);
    assertEquals(f1, f2);
    assertFalse(f1.equals(f3));
}
```

Facilitando as comparações

- Atualizar método **testMultiplicar**

```
@Test
public void testMultiplicar(){
    Fracao f1 = new Fracao(1, 2);
    Fracao f2 = new Fracao(3, 5);
    Fracao resultado = f1.multiplicar(f2);
    //testa resultado
    assertEquals(new Fracao(3, 10), resultado);
}
```

Observações sobre métodos de teste

- Devem ser públicos, sem retorno e sem parâmetros.
- São independentes: JUnit cria uma nova instância da classe de testes antes da execução de cada método.
- Não há como determinar a ordem de execução dos métodos.

Anotações de apoio

- **@Before**: marca um método a ser executado antes de cada método de teste.
- **@After**: marca um método a ser executado após cada método de teste.
- **@BeforeClass**: marca um método a ser executado antes do início de todos os métodos de teste.
- **@AfterClass**: marca um método a ser executado após o término de todos os métodos de teste.

Exercícios

- Implementar e testar métodos para as operações de divisão, adição e subtração de frações.

Referências

- Tahchiev, Petar et al. **JUnit in Action**, 2ª ed, Manning: 2011.
- Richardson, Chris. **POJOS em ação**, Ciência Moderna, Rio de Janeiro: 2007.