


Laboratório de Engenharia de Software

INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho
ivan@inf.puc-rio.br

Programa – Capítulo 18



Laboratório de Engenharia de Software

- Testes unitários com JUnit

© LES/PUC-Rio

Programa – Capítulo 18




Laboratório de Engenharia de Software

- Testes unitários com JUnit

© LES/PUC-Rio

Anotações (introduzidas na versão 1.5)




Laboratório de Engenharia de Software

- As anotações são uma forma de metadados
 - fornecem dados sobre um programa, mas não fazem parte do próprio programa
 - não têm efeito direto nas operações anotadas
- As anotações têm vários usos, entre eles estão:
 - **Informações para o compilador** – as anotações podem ser usadas pelo compilador para detectar erros ou suprimir avisos
 - **Processamento de tempo de compilação e tempo de implantação** – as ferramentas de software podem processar informações de anotação para gerar código, arquivos XML e assim por diante
 - **Processamento em tempo de execução** – algumas anotações estão disponíveis para serem examinadas em tempo de execução

© LES/PUC-Rio

Laboratório de Engenharia de Software

Exemplos de anotações



- Em sua forma mais simples, uma anotação se parece com o seguinte: `@Entity`
- O caractere de arroba (@) indica ao compilador que o que se segue é uma anotação. No exemplo a seguir, o nome da anotação é `Override`

```
@Override
void mySuperMethod() { ... }
```


- Anotações podem incluir elementos, que podem ser nomeados e possuir valores

```
@Author(
    name = "Benjamin Franklin",
    date = "3/27/2003"
)
class MyClass() { ... }
```

© LES/PUC-Rio

Laboratório de Engenharia de Software

Teste unitário (1)



- Teste unitário**
 - Procura erros em um subsistema isolado
 - Em geral, subsistema significa uma classe ou objeto particular
 - A biblioteca Java **JUnit** nos ajuda a realizar facilmente testes de unidade
- A ideia básica**
 - Para uma determinada classe **Xpto**, crie outra classe **XptoTest** para testá-la, contendo vários métodos de "caso de teste" a serem executados
 - Cada método procura resultados específicos (passa ou falha)

© LES/PUC-Rio

Teste unitário (2)



- **JUnit fornece comandos assert para a escrita de testes**
 - **A ideia:** insira chamadas de asserções em seus métodos de teste para verificar coisas que você espera que sejam verdadeiras. Se não forem, o teste falhará

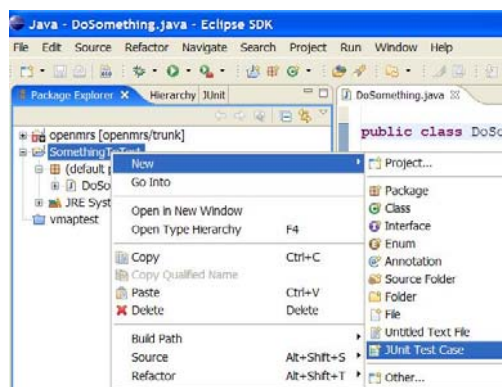
© LES/PUC-Rio

JUnit e Eclipse



- **Para adicionar JUnit a um projeto Eclipse, clique em:**
 - Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish

- Para criar um caso de teste:
 - clique com o botão direito em um arquivo e escolha **New → JUnit Test Case**
 - Ou clique **File → New → JUnit Test Case**
 - O Eclipse pode criar stubs de métodos de teste para você



© LES/PUC-Rio

Uma classe de teste JUnit



```
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() { // um método de caso de
        ...             // teste
    }
}
```

- Um método com **@Test** é um caso de teste JUnit
 - Todos os métodos **@Test** são executados quando JUnit executa uma classe de teste

© LES/PUC-Rio

Métodos de asserção JUnit



<code>assertTrue(test)</code>	fails if the boolean test is false
<code>assertFalse(test)</code>	fails if the boolean test is true
<code>assertEquals(expected, actual)</code>	fails if the values are not equal
<code>assertSame(expected, actual)</code>	fails if the values are not the same (by ==)
<code>assertNotSame(expected, actual)</code>	fails if the values are the same (by ==)
<code>assertNull(value)</code>	fails if the given value is <i>not</i> null
<code>assertNotNull(value)</code>	fails if the given value is null
<code>fail()</code>	causes current test to immediately fail

- Cada método também pode receber uma string para ser exibida em caso de falha
 - por exemplo, `assertEquals("mensagem", esperado, real)`

© LES/PUC-Rio

Teste JUnit com ArrayList

LES

```

import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayList {
    @Test
    public void testAddGet1() {
        ArrayList list = new ArrayList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayList list = new ArrayList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
}


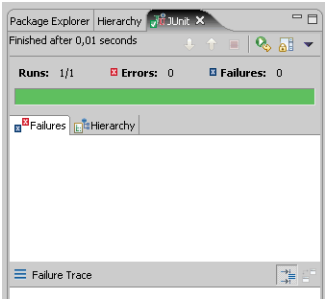
```

© LES/PUC-Rio

Executando um teste

LES

- Clique com o botão direito no Eclipse Package Explorer e escolha
 - **Run As → JUnit Test**
- A barra JUnit ficará **verde** se todos os testes forem aprovados e **vermelha** se houver falha
- O rastreamento de falha mostra quais testes falharam, se houver, e por quê

© LES/PUC-Rio

Exercício



- Seja uma classe `Date` com os seguintes métodos:
 - `public static Date getDate(int year, int month, int day)`
 - `private Date()` `// hoje`
 - `public int getDay(), getMonth(), getYear()`
 - `public void addDays(int days)` `// avança alguns dias`
 - `public int daysInMonth()`
 - `public String dayOfWeek()` `// ex: "domingo"`
 - `public boolean equals(Object o)`
 - `public boolean isLeapYear()`
 - `public void nextDay()` `// avança um dia`
 - `public String toString()`
- Faça testes unitários para verificar o seguinte:
 - nenhum objeto **Date** pode entrar em um estado inválido
 - o método **addDays** funciona corretamente

© LES/PUC-Rio

O que há de errado nisso?



```
public class DateTest {

    @Test
    public void test1() {
        Date d = Date.getDate(2050, 2, 15);
        assertNotNull(d);
    }

    @Test
    public void test2() {
        Date d = Date.getDate(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }
}
```

© LES/PUC-Rio

Assertivas bem estruturadas



```
public class DateTest {

    @Test
    public void test1() {
        Date d = Date.getDate(2050, 2, 15);
        assertNotNull("Data inválida", d);

        // casos de teste devem, usualmente, ter mensagens
        // explicativas sobre o que está sendo testado
    }

    @Test
    public void test2() {
        Date d = Date.getDate(2050, 2, 15);
        d.addDays(14);
        assertEquals("Ano inválido", 2050, d.getYear());
        assertEquals("Mês inválido", 2, d.getMonth());
        assertEquals("Dia inválido", 19, d.getDay());

        // os valores esperados devem ser os da esquerda
    }
}
```

Objetos de respostas esperadas



```
public class DateTest {

    @Test
    public void test1() {
        Date d = Date.getDate(2050, 2, 15);
        d.addDays(4);
        Date expected = Date.getDate(2050, 2, 19);
        assertEquals(expected, d);
    }

    // Use um objeto de resposta esperada para minimizar os testes
    // Date deve implementar os métodos toString e equals
}
}
```


Nomeando casos de teste



```
public class DateTest {

    @Test
    public void testaSomaDias() {
        Date actual = Date.getDate(2050, 2, 15);
        actual.addDays(4);
        Date expected = Date.getDate(2050, 2, 19);
        assertEquals(expected, actual);
    }

    // Dê aos métodos de caso de teste nomes descritivos
    // realmente longos

    // Dê nomes descritivos aos valores esperados / reais
}
```

© LES/PUC-Rio

Testes com limite de tempo



```
@Test(timeout = 5000)
public void name() { ... }
```

- O método acima será considerado uma falha se não terminar a execução em 5000 ms

```
private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Tempo esgotado / falha após 2000 ms

© LES/PUC-Rio

Limites de tempo gerais



```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void testaDataValida() {
        Date d = Date.getDate(2050, 2, 15);
        assertNotNull("Data inválida", d);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void testaSomaDias() {
        Date actual = Date.getDate(2050, 2, 15);
        actual.addDays(14);
        assertEquals("Ano inválido", 2050, actual.getYear());
        assertEquals("Mês inválido", 2, actual.getMonth());
        assertEquals("Dia inválido", 19, actual.getDay());
    }
    // quase todos os testes devem ter um tempo limite
    // isso evita a ocorrência de loops infinitos

    private static final int DEFAULT_TIMEOUT = 2000;
}
```

© LES/PUC-Rio

Teste de exceções




```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Passará se lançar a exceção fornecida em @Test
 - Se a exceção não for lançada, o teste falhará
 - Use isso para testar os erros esperados

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayList list = new ArrayList();
    list.get(4); // deveria falhar
}
```

© LES/PUC-Rio

Pré e pós-condições



Laboratório de Engenharia de Software

```

@Before
public void name() { ... }
@After
public void name() { ... }

• Métodos a serem executados antes / depois de cada método de caso de teste que for chamado


@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }

• Métodos a serem executados uma única vez antes / depois de todos métodos de teste definidos na classe

```

© LES/PUC-Rio

Dicas para testes



Laboratório de Engenharia de Software

- Não se pode testar todas as entradas possíveis, valores de parâmetros e etc
 - portanto, deve-se pensar em um conjunto limitado de testes com probabilidade de expor bugs
- Pense em casos limites
 - positivo; zero; números negativos
 - bem nos limites de uma matriz ou tamanho de uma coleção
- Pense em casos vazios e casos de erro
 - 0, -1, nulo; uma lista ou array vazio
- Teste combinações de comportamentos
 - talvez o método **add()** geralmente funcione, mas falhe depois que **remove()** for chamado
 - faça várias chamadas; talvez o método **size()** falhe apenas na segunda vez

© LES/PUC-Rio

Testes confiáveis



- Teste uma coisa de cada vez em cada método de teste
 - 10 testes pequenos são muito melhores do que um único teste que seja 10 vezes maior
- Cada método de teste deve ter poucas (provavelmente 1) chamadas à assertiva
 - ao se testar muitas assertivas, a primeira que falhar irá interromper o teste
 - assim não se saberá se uma assertiva posterior teria falhado
- Os testes devem ter poucas estruturas de controle de fluxo
 - deve-se minimizar a ocorrência de if / else, loops, switch e etc
 - try / catch deve ser evitado – se for para lançar uma exceção, use **expected** = ... se não, deixe o JUnit pegá-la
- Testes de estresse são aceitáveis, mas apenas em adição aos testes simples

© LES/PUC-Rio

Domine a redundância



```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_withinSameMonth_1() {
        addHelper(2050, 2, 15, +4, 2050, 2, 19);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_wrapToNextMonth_2() {
        addHelper(2050, 2, 15, +14, 2050, 3, 1);
    }

    //use muitos "helpers" para tornar os testes mais curtos
    private void addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date act = new Date(y1, m1, d1);
        act.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", exp, act);
    }

    //pode-se usar, também, "testes parametrizados"
}
```

© LES/PUC-Rio

“Helpers” (métodos auxiliares) flexíveis



```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
        addHelper(d, +32, 2050, 4, 2);
        addHelper(d, +98, 2050, 7, 9);
    }
    // criar variações pode tornar os helpers mais
    // flexíveis
    private Date addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date date = new Date(y1, m1, d1);
        addHelper(date, add, y2, m2, d2);
        return d;
    }
    private void addHelper(Date d, int add,
                           int y2, int m2, int d2) {
        d.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("date after + " + add + " days", exp, d);
    }
}
```

© LES/PUC-Rio

Teste de regressão (1)




- **Regressão** – quando um recurso que funcionava, não funciona mais
 - Provável de acontecer quando o código muda e cresce com o tempo
 - Um novo recurso / correção pode causar um novo bug ou reintroduzir um bug antigo
- **Teste de regressão** – reexecução de testes de unidade anteriores após uma mudança
 - Frequentemente feito por scripts durante o teste automatizado
 - Usado para garantir que bugs antigos corrigidos não estejam mais presentes
 - Fornece ao software um nível mínimo de funcionalidades devidamente funcionando

© LES/PUC-Rio

Laboratório de Engenharia de Software

Teste de regressão (2)




- Os módulos devem ser submetidos a conjunto de testes de verificação obrigatórios antes que possam ser adicionados a um repositório de código-fonte

© LES/PUC-Rio

Laboratório de Engenharia de Software

Desenvolvimento orientado a testes (TDD)



- Testes unitários podem ser escritos depois, durante ou mesmo antes da codificação
 - **desenvolvimento orientado a testes** – escreva os testes e, em seguida, escreva código a ser testado
- Imagine que se queira adicionar um método chamado `subtractWeeks` à classe `Date`, que desloca uma data para trás no tempo pelo número de semanas fornecido
- Escreva o código para testar esse método antes que ele seja escrito
 - Logo após a implementação do método estar concluída vai se saber se ele funciona

© LES/PUC-Rio

O que deve ser evitado na criação de testes?



- Testes devem ser independentes e não devem interferir uns nos outros
- Deve-se evitar na elaboração de testes:
 - Ordem de teste estrita – o teste A deve ser executado antes do teste B (*geralmente uma tentativa equivocada de testar a ordem / fluxo*)
 - Testes que chamam uns aos outros – o método de teste A chama o método do teste B (*entretanto, chamar um auxiliar compartilhado é normal*)
 - Estado compartilhado mutável – os testes A / B usam um objeto compartilhado (*se A "quebra", o que acontece com B?*)

© LES/PUC-Rio

Suite de testes



- **Suite de teste** – uma classe que executa muitos testes JUnit
 - Uma maneira fácil de executar todos os testes de um software / módulo de uma vez

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestClassName.class,
    TestClassName.class,
    ...
    TestClassName.class,
})
public class name {}
```

© LES/PUC-Rio

Laboratório de Engenharia de Software

Exemplo de suite de testes




```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

© LES/PUC-Rio

Laboratório de Engenharia de Software

Resumo JUnit (1)




- Testes precisam de atomicidade de falha (capacidade de saber exatamente o que falhou)
 - Cada teste deve ter um nome claro, longo e descritivo
 - As assertivas devem sempre ter mensagens claras para que se possa saber o que falhou
 - Escreva muitos testes pequenos, em vez de um grande teste
 - Cada teste deve ter, sempre que possível, apenas uma assertiva
- Sempre use um parâmetro de tempo limite (**timeout**) para cada teste
- Teste os erros / exceções esperados
- Escolha um método de **assert** adequado, nem sempre assertTrue é a melhor escolha

© LES/PUC-Rio

Laboratório de Engenharia de Software

Resumo JUnit (2)



- Sempre que possível, evite lógica complexa em métodos de teste
- Use métodos auxiliares e `@Before` para reduzir a redundância entre os testes

© LES/PUC-Rio