

CURSO DE TESTES AUTOMATIZADOS

TESTES UNITÁRIOS

Wagner Costa
wcaquino@gmail.com

O QUE É JUNIT?

- Atualmente é o framework padrão para testes de unidade em Java
 - Desenvolvido por Kent Beck e Erich Gamma
 - Autores do conhecido livro *Design Patterns*
- Design simples e extensível
 - Existem (e continuam a surgir) vários outros frameworks de teste que estendem o JUnit
 - Por exemplo, para testes com bancos de dados, com XML, *JavaScript*, ...
 - Neste caso, integração significa criar uma suíte de testes do JUnit usando casos de testes destes outros frameworks mais especializados
 - Tudo roda no mesmo executor de testes, que é o executor fornecido pelo JUnit. Já integrado com o Eclipse.
- JUnit torna as coisas mais agradáveis, facilitando:
 - A execução automática de testes
 - A interpretação dos resultados

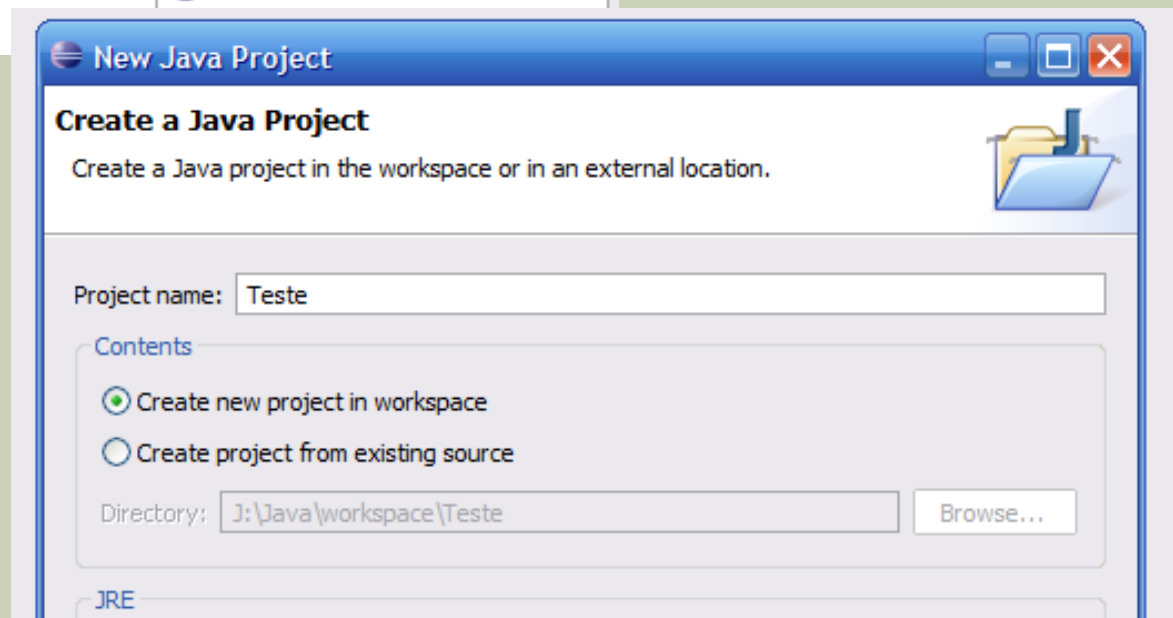
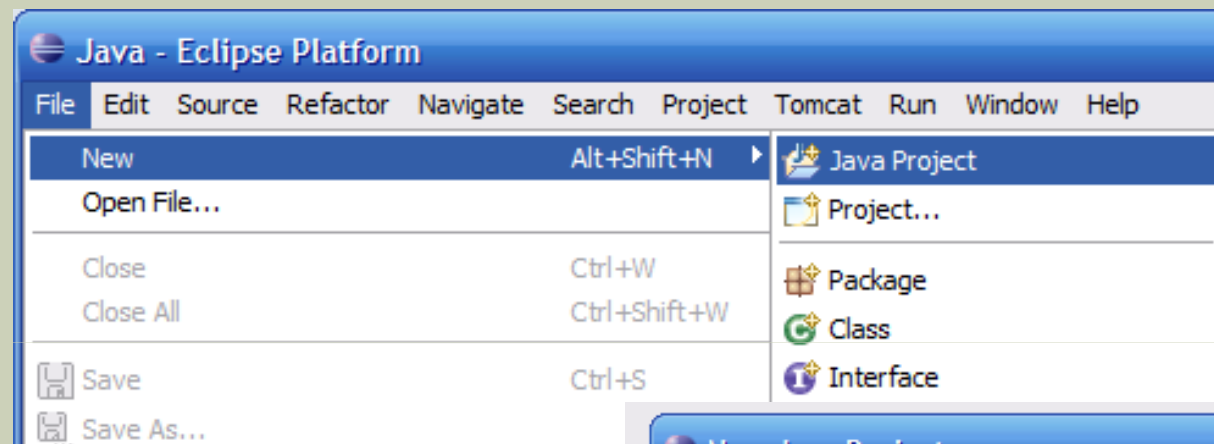
OBTENDO O JUNIT

- Download gratuito em www.junit.org
- Última versão: 4.11

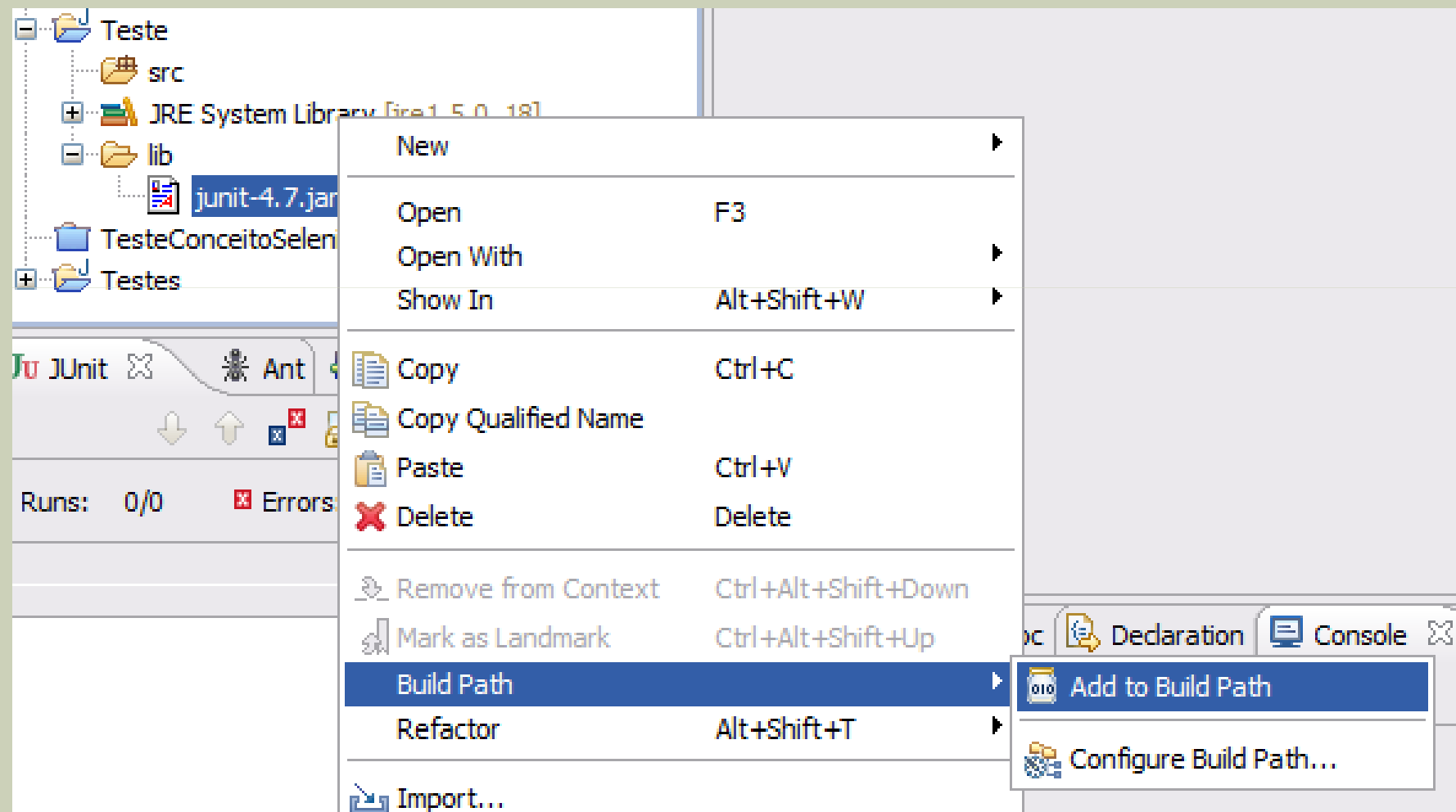
INSTALAÇÃO

- Não precisa instalar nada!
- Basta colocar o jar do JUnit (e suas dependências) no classpath da aplicação que você quer testar
- Pronto!
- E para remover o JUnit?
 - Mesmo processo:
 - Basta apagar as libs inseridas
 - É claro, quaisquer testes que você tenha escrito até então não vão mais compilar

CRIANDO UM PROJETO

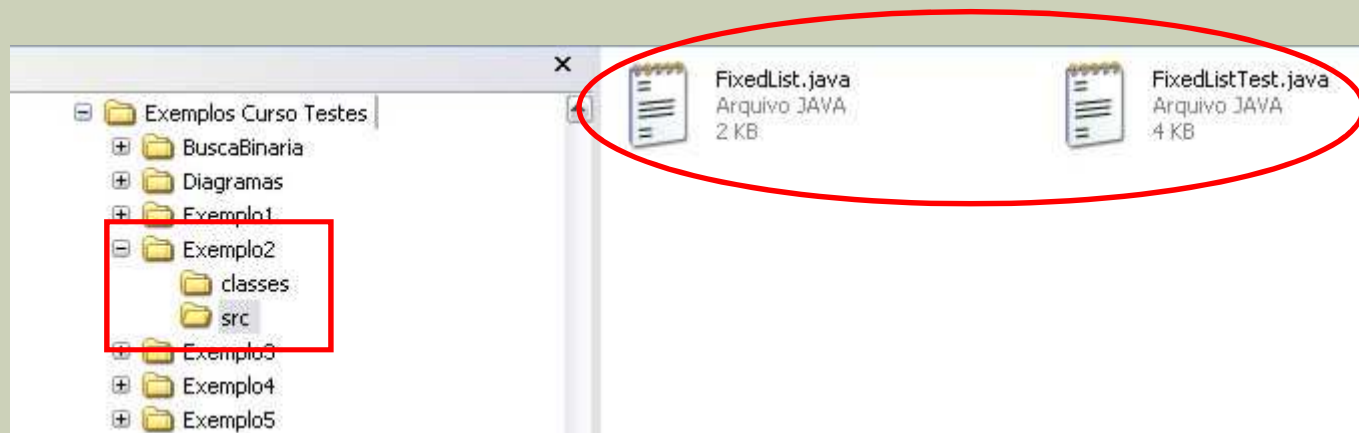


ADICIONAR LIB DO JUNIT NO BUILD PATH



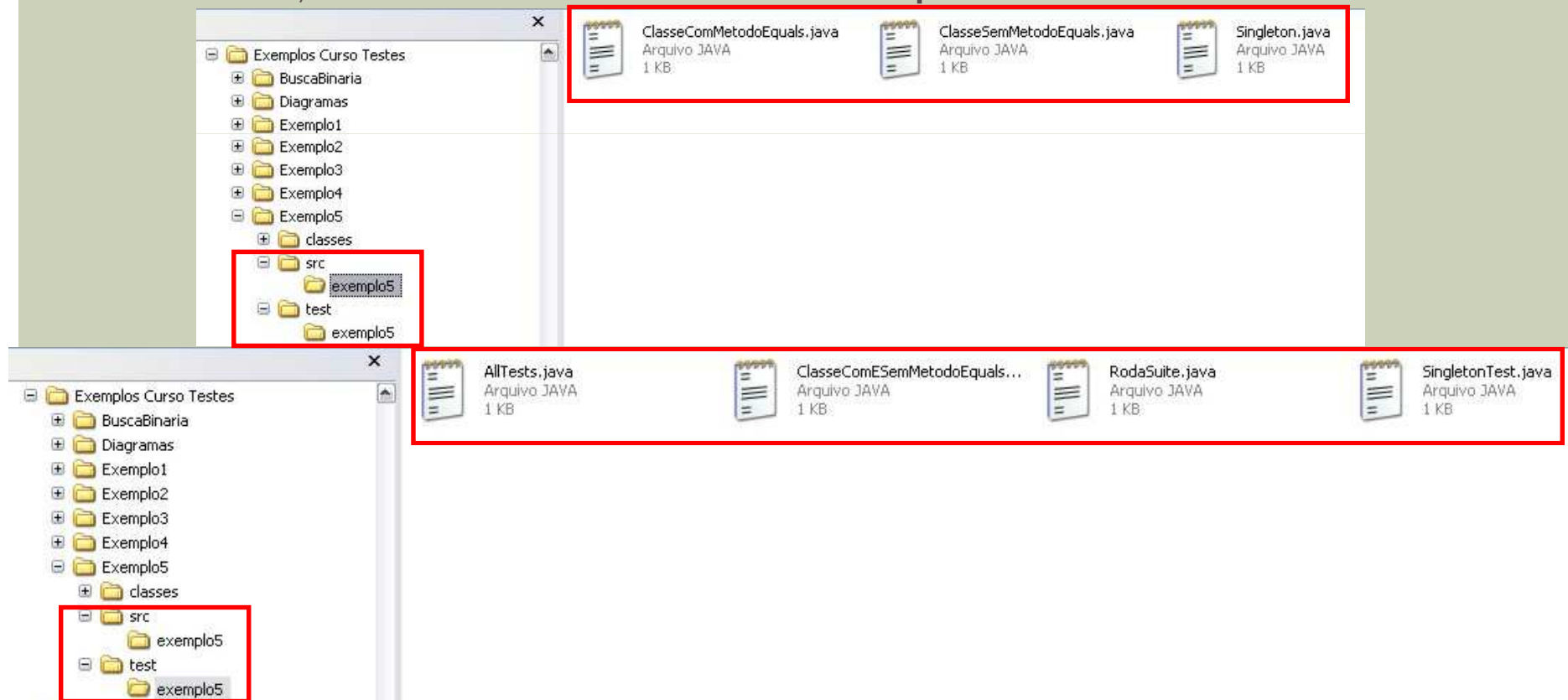
ONDE COLOCAR OS TESTES (1/2)

- Antigamente testes de unidade eram colocados no mesmo diretório que as classes testadas
- Também é convenção usar nos testes de unidade o mesmo nome da classe testada + sufixo “Test”



ONDE COLOCAR OS TESTES (2/2)

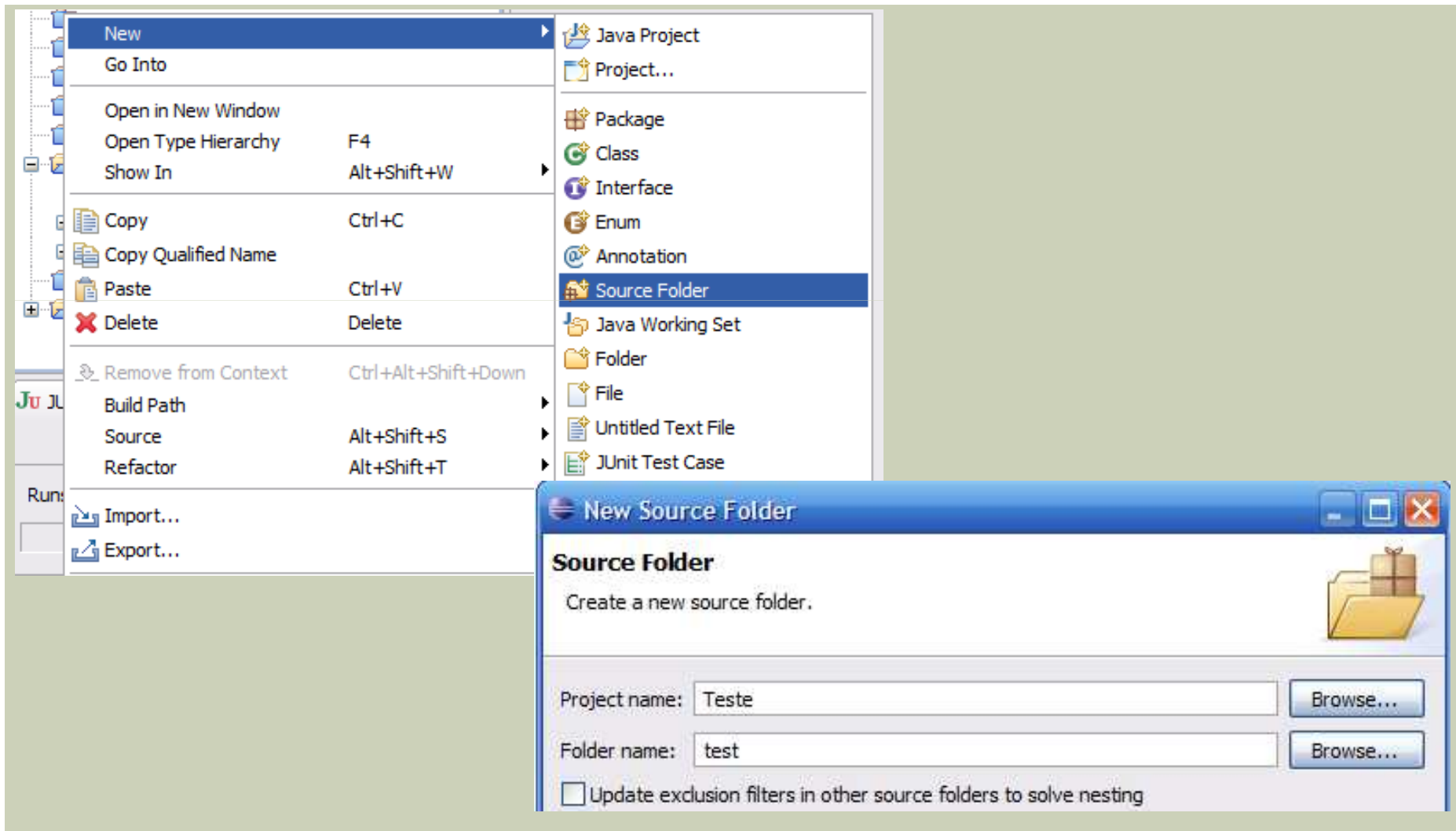
- Padrão mais recente sugere colocar testes numa hierarquia de pastas diferente da hierarquia do código fonte
 - Porém, usando a mesma estrutura de pacotes



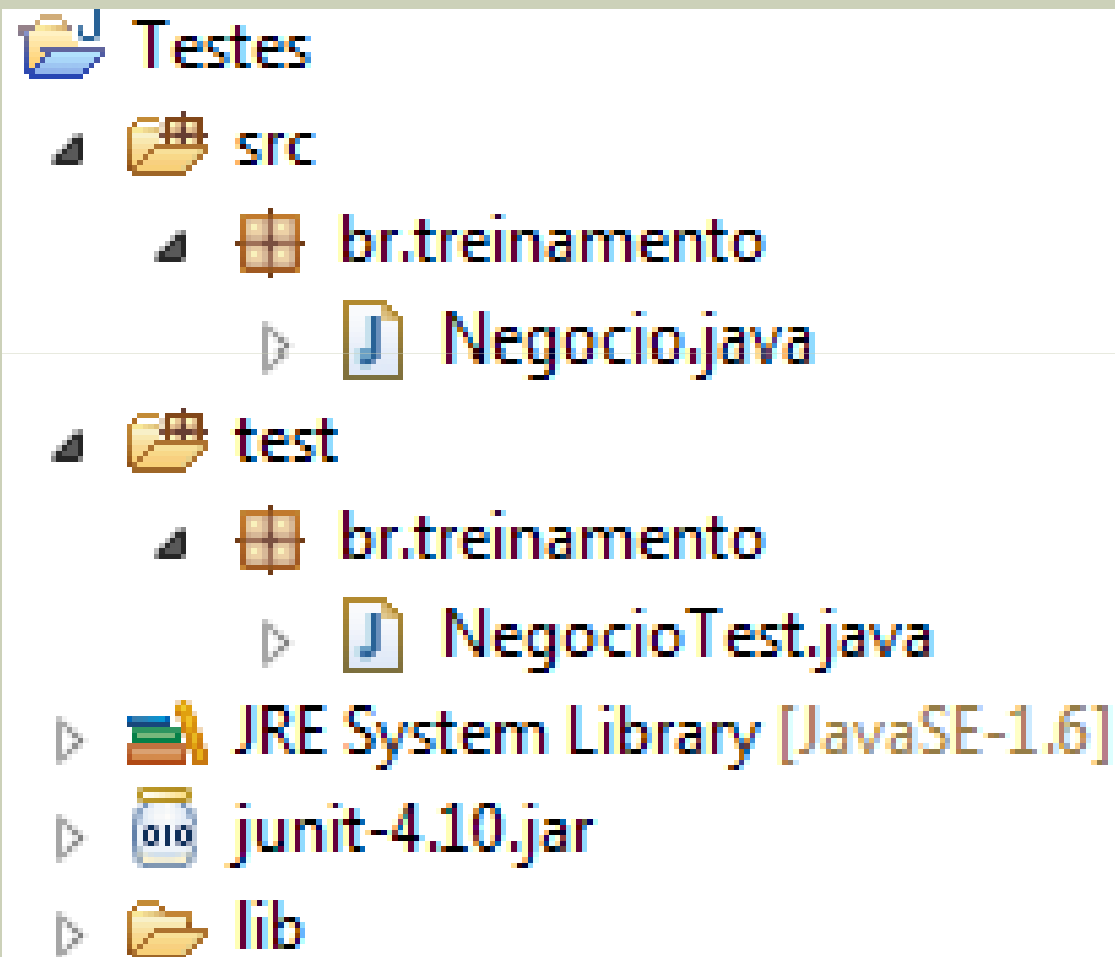
PADRÕES DO NOMENCLATURA

- Existem alguns poucos padrões de nomenclatura recomendados ao escrever testes de unidade com JUnit
 - Em geral, cada classe de teste está associada com exatamente uma classe testada (não é regra nem convenção, é costume...)
 - Neste caso, o nome da classe de teste será <nome classe testada>Test.java
 - Exemplo:
 - Carro.java => CarroTest.java
 - Socket.java => SocketTest.java
- Têm-se como bom costume dar nomes sugestivos aos métodos de teste
 - testLerArquivoValido()
 - testLerArquivoQueNaoExiste()
 - testLerArquivoSemPropriedadeNetworkEnabled()
- Não tenha medo de nomes de método grandes!

CRIANDO UM NOVO SOURCE FOLDER



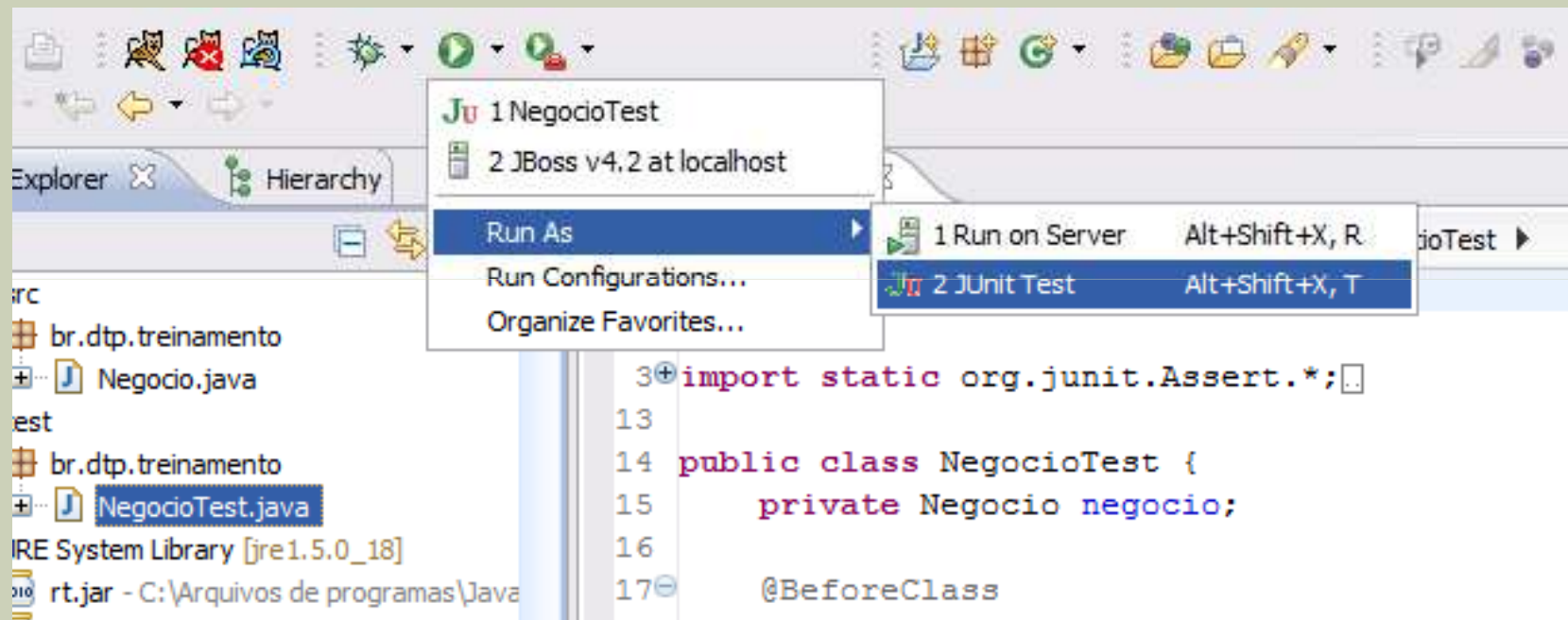
TUDO NO SEU LUGAR



OS EXECUTORES DE TESTE

- Provavelmente a maior virtude do JUnit é sua facilidade para executar os testes escritos usando sua API
- Para executar testes no JUnit
 - Crie um teste, ou uma suíte de testes
 - Mande o executor rodar o(s) teste(s)

EXECUTAR TESTES



ATALHOS

■ Run

- Ctrl + F11
- Alt + Shift + X => T

■ Debug

- F11
- Alt + Shift + D => T

Run Ant Build	Alt+Shift+X, Q	
Run Eclipse Application	Alt+Shift+X, E	
Run JUnit Plug-in Test	Alt+Shift+X, P	
Run JUnit Test	Alt+Shift+X, T	
Run Java Applet	Alt+Shift+X, A	
Run Java Application	Alt+Shift+X, J	
Run OSGi Framework	Alt+Shift+X, O	
Run on Server	Alt+Shift+X, R	

INTERFACE JUNIT

The screenshot displays the JUnit interface within an IDE. The top toolbar includes icons for Markers, Properties, Servers, Data Source Explorer, Snippets, Console, Search, and JUnit. Below the toolbar, a status bar indicates "Finished after 1,088 seconds". The main area shows test results for "br.treinamento.NegocioTest [Runner: JUnit 4] (1,040 s)". The results list includes:

- testSomar (0,000 s) [Success]
- testDividir (0,000 s) [Success]
- testDividirPorZero_robusto (0,000 s) [Success]
- testMDC (0,000 s) [Success]
- testDividirPorZero_elegante (0,000 s) [Success]
- infinito (1,039 s) [Failure]

The "Failure Trace" panel on the right shows the following error:

```
java.lang.Exception: test timed out after 1000 milliseconds  
at br.treinamento.NegocioTest.infinito(NegocioTest.java:79)
```

FALHAS E ERROS

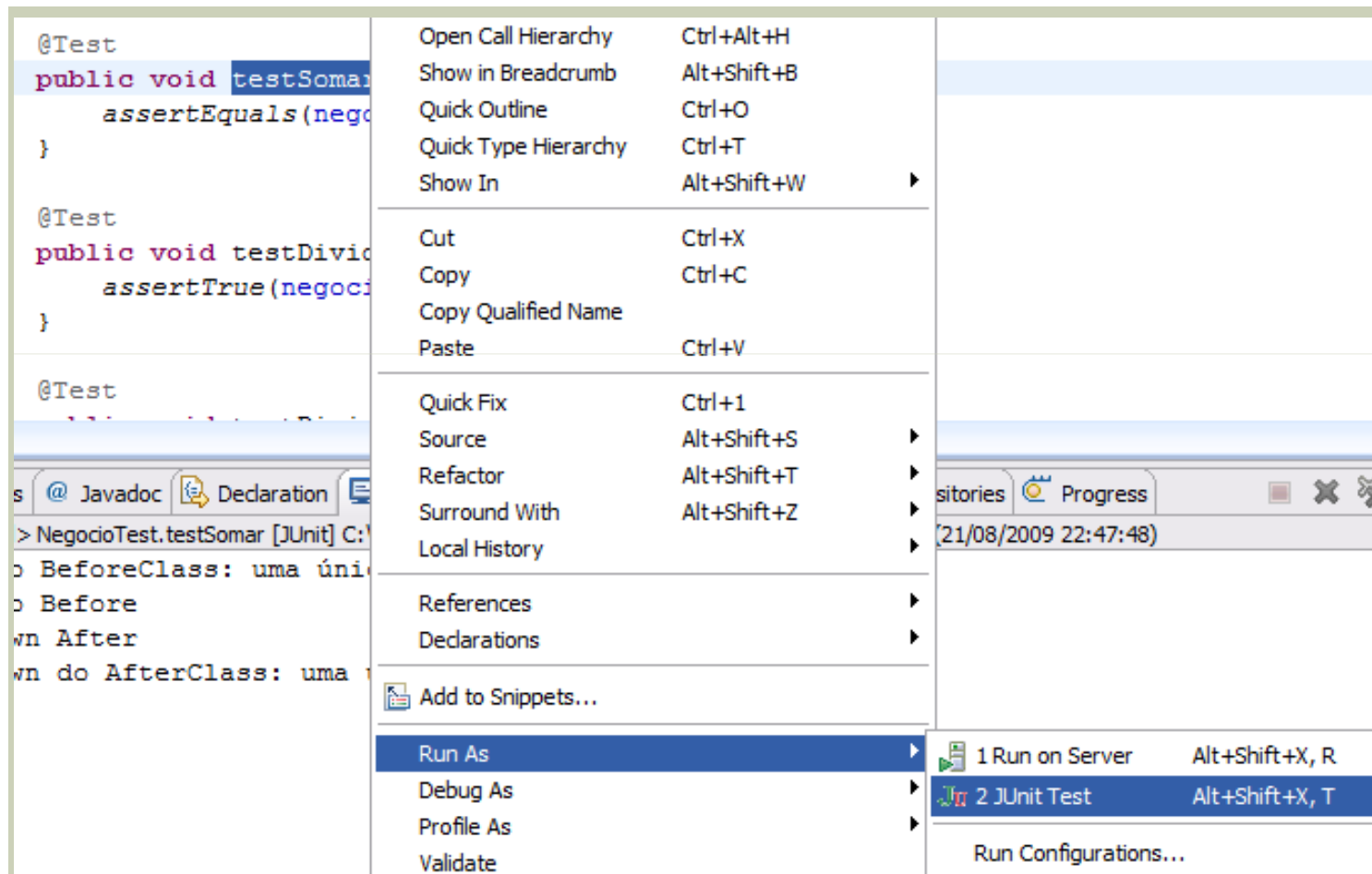


Falha: Alguma condição esperada no teste não foi satisfeita.



Erro: Ocorreu algum problema (inesperado) durante a execução do teste que não permitiu a continuação do mesmo.

RODAR UM ÚNICO MÉTODO DE TESTE



Os atalhos também funcionam.

MÃOS À OBRA

- Criar um projeto com o nome “Testes”
 - Adicionar o JUnit no classpath
 - Adicionar a classe “Basico.java” no source folder principal
 - Adicionar a classe “BasicoTest.java” no source folder de testes.
- Executar TODOS os testes da classe BasicoTest
- Executar apenas um teste da classe BasicoTest
- Reconhecer os elementos da interface do Junit
- Debugar algum método
- Testar os atalhos

ANOTAÇÕES

- Test
- Before e After
- BeforeClass e AfterClass
- Ignore
- Rule

@TEST

Anote seus testes com @Test para indicar ao JUnit que este método deve ser tratado como um método de teste.

```
@Test
public void testSomar() {
    assertEquals(negocio.somar(1, 1), 2);
}
```

```
@Test
public void testDividir() {
    assertTrue(negocio.dividir(10, 2) == 5d);
}
```

@BEFORE E @AFTER

Utilize as *anottations* @Before e @After para para “*setup*” e “*tearDown*” respectivamente. Eles serão executados antes e depois de cada um dos seus casos de testes.

```
@Before
public void setUpBefore() {
    System.out.println("SetUp do Before");
    negocio = new Negocio();
}

@After
public void tearDownAfter() {
    System.out.println("Tear Down After");
    negocio = null;
}
```

@BEFORECLASS E @AFTERCLASS

Utilize as *annotations* `@BeforeClass` e `@AfterClass` para “*setup*” e “*tearDown*” a nível de classe. Pense neles como “*setup*” e “*teardown*” que são executados apenas uma vez, antes e depois dos testes.

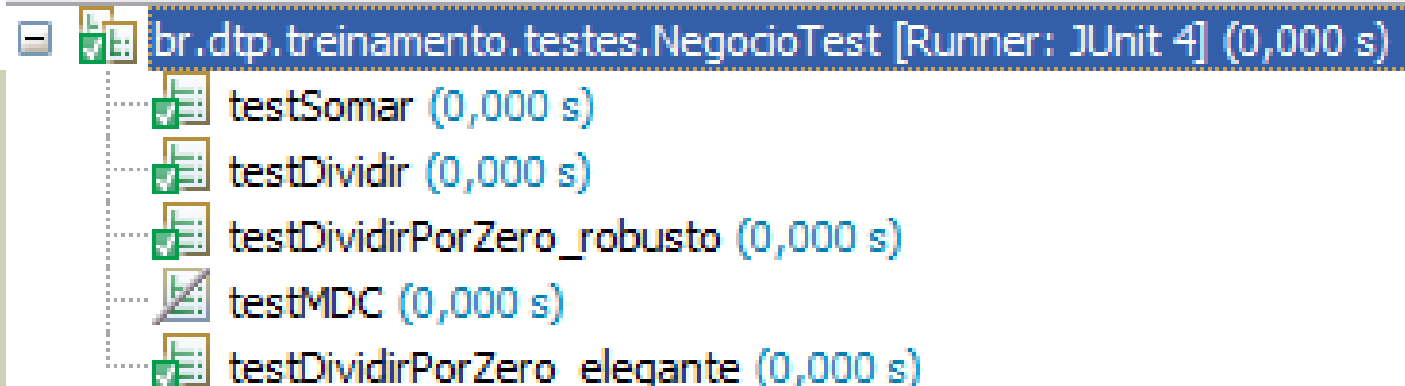
```
@BeforeClass
public static void setUpBeforeClass() {
    System.out.println("SetUp do BeforeClass: uma única vez");
}

@AfterClass
public static void tearDownAfterClass() {
    System.out.println("Tear Down do AfterClass: uma única vez");
}
```

@IGNORE

Use a *annotation* @Ignore para testes que você queira ignorar. Você pode adicionar um parâmetro String que defina o motivo pelo qual você está ignorando o teste.

```
@Ignore("Não lembro como faz isso")
@Test
public void testMDC() {
    assertEquals(negocio.mdc(1, 2, 3), 100);
}
```



TIMEOUT

O parâmetro "*timeout*" define o tempo máximo em milisegundos. O teste falha caso o período seja excedido.

```
@Test(timeout = 1000)
public void infinito() {
    while (true);
}
```


@RULE

- Permite a redefinição de comportamento para os testes

- Redefinição de Timeout

```
@Rule  
public Timeout timeOut = new Timeout(20);
```

- Tratamento de exceções

- Para criar regras novas, basta criar uma classe implementando a interface TestRule

TRATAMENTO DE EXCEÇÕES

- As exceções previstas nos testes podem ser tratadas de três formas:

- Elegante

- Robusta

- Nova

FORMA ELEGANTE

Utilize o parâmetro "*expected*" da *annotation* `@Test` para casos de uso que esperam *exceptions*. Escreva o nome da classe da exceção que deverá ser lançada.

```
@Test(expected = ArithmeticException.class)
public void testDividirPorZero_elegante() {
    assertTrue(negocio.dividir(10, 0) == 0d);
}
```

FORMA ROBUSTA

- Não é das soluções mais belas, mas permite continuar os testes após o lançamento da exceção, inclusive realizar outras assertivas.

```
@Test
public void testDividirPorZero_robusto() {
    try{
        assertTrue(negocio.dividir(10, 0) == 0d);
        fail("Não deveria chegar nesta linha");
    } catch (ArithmeticException e) {
        assertEquals("/ by zero", e.getMessage());
    }
}
```

FORMA NOVA - REGRAS

- Com a chegada das regras, é possível utilizar a `ExpectedException` para definir tanto a exceção desejada quanto a mensagem esperada.

```
@Rule
public ExpectedException thrown= ExpectedException.none();
```

```
@Test
public void testDividirPorZero_regra() {
    thrown.expect(Exception.class);
    thrown.expectMessage(is("/ by zero"));
    assertEquals(negocio.dividir("10", "0"), 10d, 0d);
}
```

MÃOS A OBRA

- Alterar os testes para ver o comportamento dos mesmos
 - Alterar o Assert de algum teste com sucesso para causar uma falha
 - Retirar a exceção esperada do teste elegante para obter um erro
 - Montar um caso onde o tipo de tratamento “Elegante” para as exceções falhe
 - Adicionar um contador no método que possui a anotação @Before e analisem o seu comportamento

A CLASSE DE TESTE

- Não há limite para quantas assertivas são feitas em cada método de teste
- O ideal são testes pequenos, que possam ser rapidamente compreendidos
 - Testes pequenos são também mais fáceis de manter

ASSERTIVAS

- `assertFalse(boolean)`
 - Afirma que o resultado da expressão recebida é falso
- `assertTrue(boolean)`
 - Afirma que o resultado da expressão recebida é verdadeiro
- `fail()`
 - Provoca uma falha no teste
- O `assertEquals()` recebe dois parâmetros e, como o nome sugere, é usado para comparar dois valores: o esperado e o obtido

ASSERT EQUALS

- Existem assertEquals() para todos os tipos nativos + objetos quaisquer

- assertEquals(int, int)
- assertEquals(short, short)
- assertEquals(long, long)
- assertEquals(double, double, double)
- assertEquals(float, float, float)
- assertEquals(char, char)
- assertEquals(byte, byte)
- assertEquals(boolean, boolean)
- assertEquals(Object, Object)

FALHAS MAIS AMIGÁVEIS

- Via de regra, todos os métodos para teste do JUnit possuem também versões onde recebem um string como primeiro parâmetro
 - Essa string é uma mensagem que é exibida pelo JUnit caso a assertiva seja falsa ou caso o teste falhe
- Assim, também estão definidos os métodos
 - `assertFalse(String, boolean)`
 - `assertTrue(String, boolean)`
 - `fail(String)`
 - `assertEquals(String, ..., ...)`

IGUALDADE DE OBJETOS

- Como o JUnit testa a igualdade dos objetos?
- O Junit utiliza o método *equals()* do próprio objeto
 - O critério de igualdade numa classe pode ser apenas uma comparação de 3 dos seus 12 atributos, por exemplo

CUIDADOS COM O EQUALS

- O equals e hashCode devem conter os mesmos atributos
 - O eclipse pode gerar esses códigos, se você pedir...
- Os atributos transientes não são recomendados para utilização em comparações
- O id também pode causar problemas

EQUALS PARA OS TESTES

- Quase sempre que quisermos testar a “igualdade” entre dois objetos usando `assertEquals` precisamos primeiro implementar o método `equals()` do objeto testado.
- O que acontece quando tentamos comparar dois objetos sem implementar seu método `equals()`?

MAIS ASSERTIVAS

- `assertEquals()`
- `assertNotEquals()`
- `assertNull()`
- `assertNotNull()`
- `assertSame()`
- `assertNotSame()`
- `assertArrayEquals()`
- `assertThat()`

ASSERT THAT

- `is()`
- `allOf()`
- `anyOf()`
- `instanceOf()`
- `not()`
- `nullValue()`
- `notNullValue()`
- `sameInstance()`

TESTES PARAMETRIZADOS

- Util para testar vários cenários com entradas e saídas distintas de um mesmo método

```
@RunWith(Parameterized.class)
public class SomaTest {

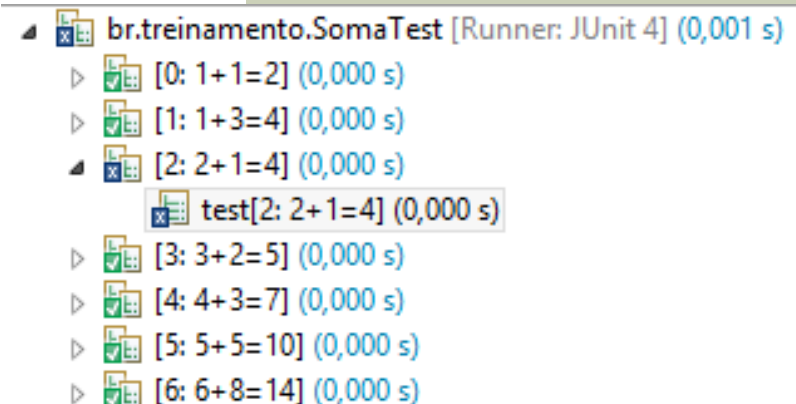
    @Parameters(name = "{index}: {0}+{1}={2}")
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {

            { 1, 1, 2 }, { 1, 3, 4 }, { 2, 1, 4 }, { 3, 2, 5 }

        });
    }

    @Parameter(value=0)
    public int num1;
    @Parameter(value=1)
    public int num2;
    @Parameter(value=2)
    public int resultado;

    @Test
    public void test() {
        Basico basico = new Basico();
        assertEquals(resultado, basico.somar(num1, num2));
    }
}
```



br.treinamento.SomaTest [Runner: JUnit 4] (0,001 s)

- ▷ [0: 1+1=2] (0,000 s)
- ▷ [1: 1+3=4] (0,000 s)
- ▷ [2: 2+1=4] (0,000 s)
 - test[2: 2+1=4] (0,000 s)
- ▷ [3: 3+2=5] (0,000 s)
- ▷ [4: 4+3=7] (0,000 s)
- ▷ [5: 5+5=10] (0,000 s)
- ▷ [6: 6+8=14] (0,000 s)

ASSUMPTIONS

- Executa o teste apenas se uma determinada configuração ocorrer
 - Versões específicas do sistema
 - Algum recurso disponível

```
@Test
public void testApenasVersaoAcima4()    {
    Integer versionNumber = getBuildVersion(); //Coleta no arquivo de configuração
    Assume.assumeTrue(versionNumber >= 4);

    //...
}
```

TESTAR MÉTODOS PRIVADOS

- Através de métodos públicos
- Alterar o modificador para protected ou default (polêmico)
- Powermock

TESTES INDEPENDENTES DE TEMPO

- Evite escrever testes que possuem qualquer tipo de dependência temporal
 - Dependência com relação à hora ou data atuais
 - Consultar a hora do sistema ao invés de definir a hora/data atuais via código no momento do teste
 - Dados que podem expirar durante o teste
 - Testar o tempo que uma operação leva para acontecer
 - Uma resposta pode demorar para chegar por diversos fatores: CPU ocupada, rede congestionada, ...
 - Existem ferramentas específicas para testes de desempenho.

TESTANDO EXCEÇÕES

- Qualquer exceção lançada, e não tratada, durante um teste é capturada pelo JUnit que automaticamente considera o teste como falho.
 - Use esse fato para evitar código desnecessário e código redundante
 - Isso mantém seus testes curtos e mais fáceis de entender por outras pessoas que também sabem como funciona o JUnit

ORDEM DE EXECUÇÃO DOS TESTES

- Nunca assuma a ordem na qual os testes serão executados
- Isso pode variar entre implementações de JVM
 - Depende de como cada uma implementa o mecanismo de reflexão (usado pelo JUnit para carregar os testes)
- A partir da versão 4.11, uma forma de garantir a ordem é usar a anotação: `@FixMethodOrder(...)`
 - `MethodSorters.JVM`
 - Ordem retornada pela JVM
 - `MethodSorters.DEFAULT`
 - determinística mas não previsível
 - `MethodSorters.NAME_ASCENDING`
 - Ordem alfabética

TESTES COM EFEITOS COLATERAIS

- Neste contexto, efeito colateral é qualquer resquício que possa atrapalhar a execução de outros testes ou mesmo do próprio teste
- Efeitos colaterais jogam pela janela o ganho de se ter uma suíte de testes automáticos
 - Ou seja, um humano tem que limpar a sujeira deixada por algum teste antes de tentar rodar novos testes
- Sintoma clássico desse tipo de problema:
 - Um determinado teste da barra verde quando executado sozinho, mas barra vermelha quando executado junto com o restante da suíte

CAMINHOS ABSOLUTOS DE ARQUIVO

- Evitar implementar leitura\escrita de arquivos com valores absolutos

```
File f = new File( "c:\\joaozinho\\testes\\props.dat" );
```

- O que acontece quando o desenvolvedor “Marcos” for implementar os testes na máquina dele?

```
File f = new File( ".." + File.separator + "testes" +  
File.separator + "props.dat" );
```

- Outras alternativas
 - Variáveis de ambiente
 - Arquivos de propriedades

MANTENDO TESTES RÁPIDOS

- Uma suíte de testes rápidos viabiliza a execução constante dos testes durante o desenvolvimento
 - Desenvolvedores tem um feedback rápido sobre eventuais impactos de suas mudanças no código que já existe
- Essa é a tão falada Integração Contínua
 - Ou seja, continuamente verificar se mudanças pequenas se integram bem com o código que já existe
- Entretanto, incorporar a prática de integração contínua ao processo de desenvolvimento pode ser complicado se testes demoram pra executar
 - Imagine rodar 30 vezes por dia uma bateria de testes que demora 3 minutos para terminar

OPERAÇÕES QUE MAIS DEMORAM EM TESTES

- Conexões com bancos de dados
- Comunicação com sistemas externos
- Trocar dados via rede
- Ler/escrever arquivos

- Alguns testes são demorados por natureza
 - Testes com componentes distribuídos
 - Testes de interface gráfica
 - Testes de carga/desempenho

ACELERANDO TESTES

- Reduzir atrasos de comunicação com sistemas externos através de Mocks.
- Uma boa solução é criar várias suítes de testes:
 - Uma suíte para testes rápidos
 - Os codificadores rodam ela continuamente durante desenvolvimento
 - Uma suíte de testes lentos
 - Rodam no repositório quando alguém faz um commit ou durante a madrugada, após um dia de trabalho
 - Uma suíte de testes muito lentos
 - E.g. testes de stress que rodam no repositório uma vez por semana/mês...
- O importante é não perder o benefício de rodar testes constantemente a medida que o trabalho de codificação evolui

COMO CRIAR UMA SUÍTE?

```
import org.junit.runner.RunWith;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestContaIntegrado.class,
    TestSubContaIntegrado.class,
    TestRendimentoIntegrado.class,
    TestTransferenciaSubContaIntegrado.class,
    TestMovimentacaoSubContaIntegrado.class,
    TestRendimentoSubContaIntegrado.class
})
public class SuiteIntegradoGeral {

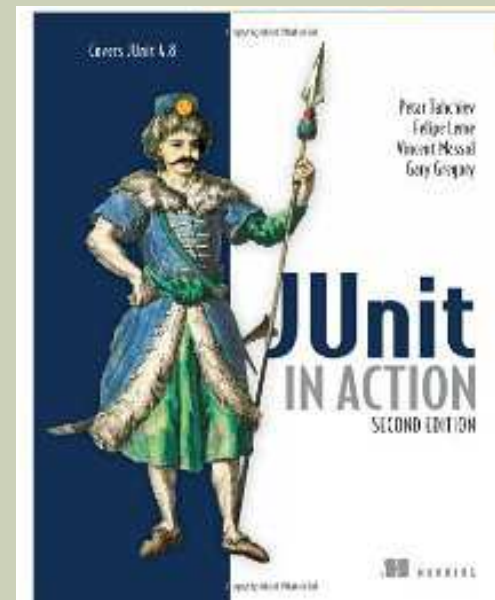
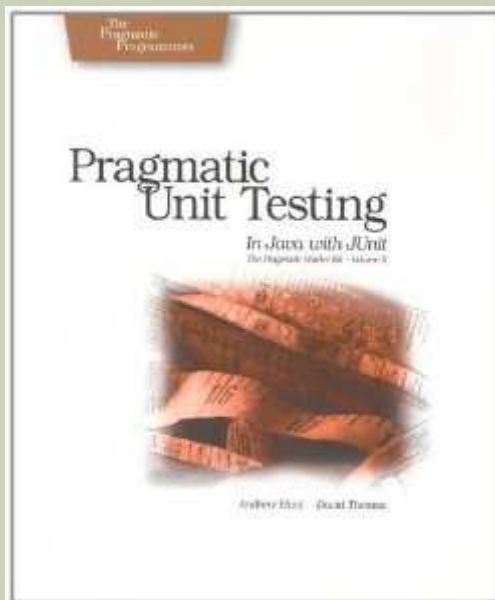
}
```

O QUE O JUNIT NÃO FAZ?

- O JUnit não automatiza a criação de testes
 - Ele automatiza sua execução
- O JUnit não garante que sua aplicação funciona
 - Testes provam que o software está quebrado, mas não podem provar que o software funciona

Testes só garantem que os comportamentos testados funcionam de acordo com o esperado

MAIS CONTEÚDO...



- Pragmatic JUnit Testing in Java with Junit
- Manning JUnit in Action – 2nd Edition

MÃOS À OBRA

- Copiar as classes “Entidade.java” e “ExercicioJUnit.java” para o projeto.
- Criar testes para todos os métodos da classe ExercicioJUnit
 - Metas:
 - Barra Verde!!!
 - 100% de linhas cobertas
 - 100% de branches cobertos