4

J820

JUnit

Helder da Rocha (helder@acm.org)

O que é "Testar código"?

- É a parte mais importante do desenvolvimento
  - Se seu código não funciona, ele não presta!
- Todos testam
  - Você testa um objeto quando escreve uma classe e cria algumas instâncias no método main()
  - Seu cliente testa seu software quando ele o utiliza (ele espera que você o tenha testado antes)
- O que são testes automáticos?
  - Programas que avaliam se outro programa funciona como esperado e retornam resposta tipo "sim" ou "não"
  - Ex: um main() que cria um objeto de uma classe testada, chama seus métodos e avalia os resultados
  - Validam os requisitos de um sistema

argonavis.com.br

Por que testar?

- Por que não?
  - Como saber se o recurso funciona sem testar?
  - Como saber se ainda funciona após alteração do design?
- Testes dão maior segurança: coragem para mudar
  - Que adianta a OO isolar a interface da implementação se programador tem medo de mudar a implementação?
  - Código testado é mais confiável
  - Código testado pode ser alterado sem medo
- Como saber quando o projeto está pronto
  - Testes == requisitos 'executáveis'
  - Testes de unidade devem ser executados o tempo todo
  - Escreva os testes antes. Quando todos rodarem 100%, o projeto está concluído!

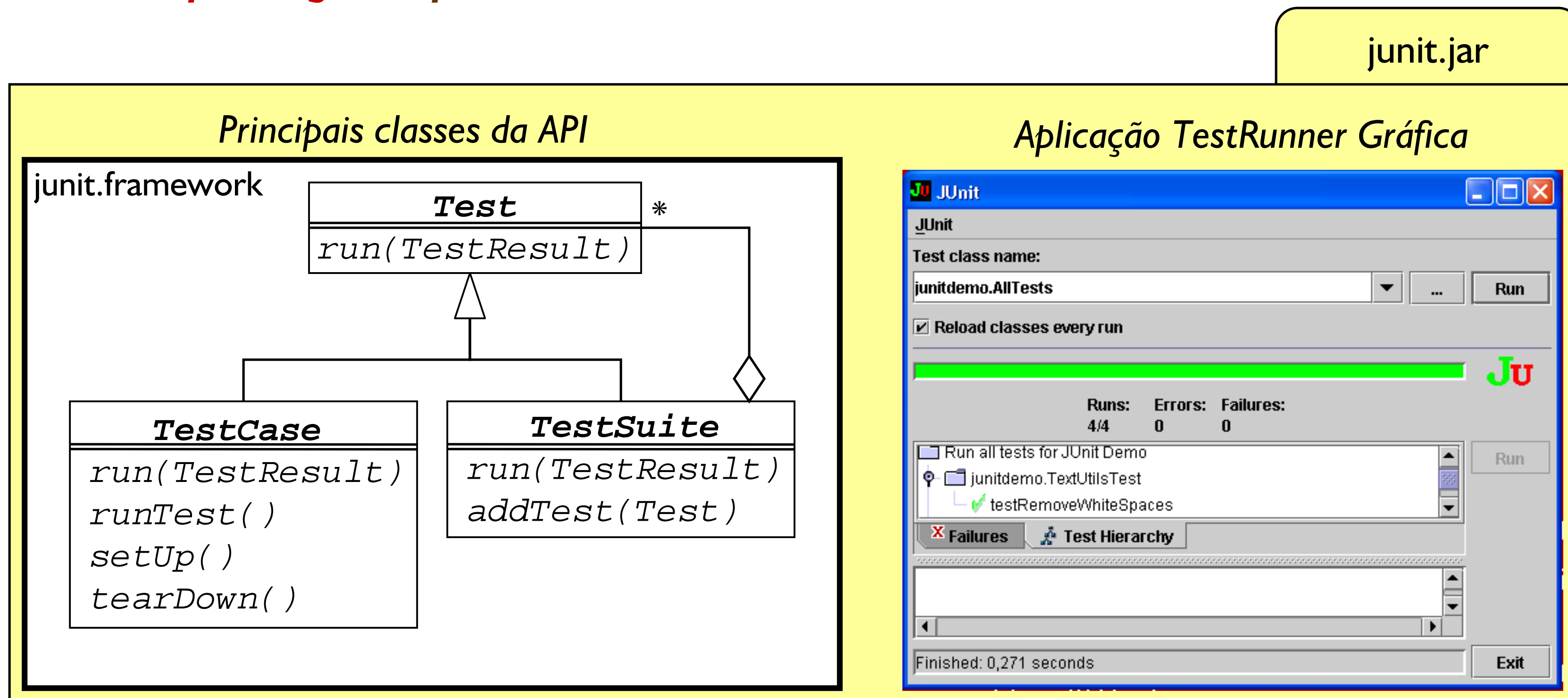
Tipos de testes

- Testes de unidade
  - Testam unidades de lógica. Em linguagens orientadas a objetos, unidades geralmente representam métodos, mas podem também representar um objeto inteiro ou ainda um estado de um método
  - Ignoram condições ou dependências externas. Testes de unidade usam dados suficientes para testar apenas a lógica da unidade em questão
- Testes de integração
  - Testam como uma coleção de unidades interage entre si ou com o ambiente onde executam.
- Testes funcionais ("caixa-preta")
  - Testam casos de uso de uma aplicação. Validam a interface do usuário, operações requisitadas, etc.

argonavis.com.br

O que é JUnit?

- *Um **framework** que facilita o desenvolvimento e execução de **testes de unidade** em código Java*
  - *Uma API para construir os testes: junit.framework.**
  - *Aplicações para executar testes: TestRunner*



Como usar o JUnit?

- Há várias formas de usar o JUnit. Depende da metodologia de testes que está sendo usada
  - Código existente: precisa-se escrever testes para classes que já foram implementadas
  - Desenvolvimento guiado por testes (TDD): código novo só é escrito se houver um teste sem funcionar
- Onde obter o JUnit?
  - www.junit.org
- Como instalar?
  - Incluir o arquivo junit.jar no classpath para compilar e rodar os programas de teste
- Extensões do JUnit
  - Permitem usá-lo para testes funcionais e de integração

JUnit para testar código existente

Exemplo de um roteiro típico

1. *Crie uma classe que estenda junit.framework.TestCase para cada classe a ser testada*

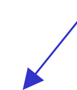
```
import junit.framework.*;
class SuaClasseTest extends TestCase {...}
```

2. *Para cada método xxx(args) a ser testado defina um método public void testXxx()* no test case*

- `SuaClasse`:
  ```
  public boolean equals(Object o) { ... }
  ```
- `SuaClasseTest`:
  ```
  public void testEquals() {...}
  ```

3. *Crie um método estático suite()* no test case*

Usará reflection para descobrir métodos que começam com "test"



```
public static Test suite() {
    return new TestSuite(SuaClasseTest.class);
}
```

* *Esta não é a única maneira de definir um teste no JUnit mas é a forma recomendada*

O que colocar em um teste?

- Cada método testXXX() do seu TestCase é um teste
  - Escreva qualquer código que sirva para verificar o correto funcionamento da unidade de código testada
  - Use asserções do JUnit para causar a falha quando resultados não estiverem corretos
- Asserções são métodos de junit.framework.Assert
  - Afirmam que certas condições são verdadeiras
  - Causam **`AssertionFailedError`** se falharem
  - **`TestCase`** estende **`Assert`**
- Principais asserções
  - **`assertEquals`**`(objetoEsperado, objetoRecebido)`
  - **`assertTrue`**`(valorBooleano)`
  - **`assertNotNull`**`(objeto)`
  - **`assertSame`**`(objetoUm, objetoDois)`
  - **`fail`**`()`

Como implementar e rodar?

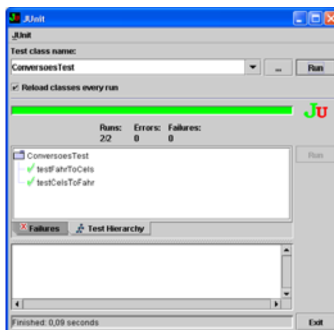
- *Exemplo de test case com **um** teste:*

```
public class CoisaTest extends TestCase {

    public static Test suite() {
        return new TestSuite(CoisaTest.class);
    }

    public void testToString() {
        Coisa coisa = new Coisa("Bit");
        assertEquals("<coisa>Bit</coisa>", coisa.toString());
    }
}
```

TestRunner chama suite() automaticamente e trata como testes (e executa) todos os métodos sem argumentos cujos nomes começarem com "test"

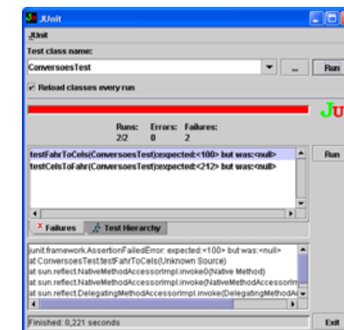
- *Para executar (aplicação gráfica do JUnit)*

```
java -cp junit.jar junit.swingui.TestRunner CoisaTest
```



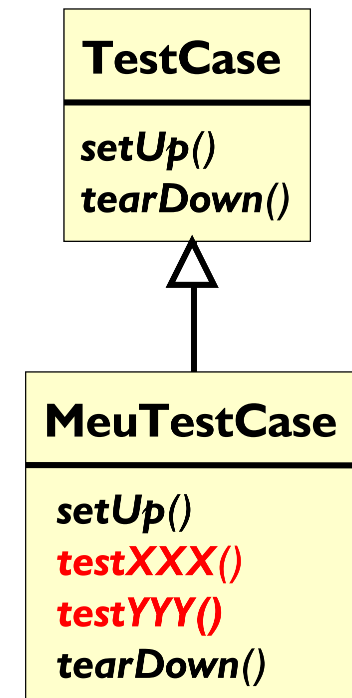
Passou!

Falhou!



Como funciona?

- O TestRunner recebe uma subclasse de junit.framework.TestCase e executa seu método run(Test)
  - Implementação default obtém dados de TestSuite (método suite())
  - TestSuite usa Java Reflection para descobrir métodos de teste
- Para cada método public void testXXX(), TestRunner executa:
  - 1. o método setUp()
  - 2. o próprio método testXXX()
  - 3. o método tearDown()
- O test case é instanciado para executar um método testXXX() de cada vez.
  - As alterações que ele fizer ao estado do objeto não afetarão os demais testes
- Método pode **terminar**, **falhar** ou causar **exceção**
  - Falhar é provocar AssertionFailedError



TestSuite

- *Representa uma composição de testes*
  - *Crie um test suite com new TestSuite("Nome");*
  - *Use addTest(Test) para incluir testes. O construtor do test case deve conter o nome do método a executar*

```
TestSuite suite = new TestSuite("Utilitarios");
suite.addTest(new ConversoesTest("testCelsToFahr"));
suite.addTest(new ConversoesTest("testFahrToCels"));
```

  - *O construtor TestSuite(classe) recebe test case e adiciona todos os métodos cujos nomes começam com "test"*

```
suite.addTest(new TestSuite(ConversoesTest.class));
```

- *Um TestSuite é usado pelo TestRunner para saber quais métodos devem ser executados como testes*
  - *TestRunner procura método static TestSuite suite()*
  - *Boa prática: defina um método suite() em cada test case retornando um TestSuite criado com a classe do test case*

TestCase com TestSuite

```
import junit.framework.*;

public class OperacoesTest extends TestCase {

    Operacoes e = new Operacoes();

    public void testQuadrado() throws IOException {
        int resultado = e.quadrado(6);
        assertEquals(36, resultado);
        assertEquals(9, e.quadrado(3));
    }

    public void testSoma() throws IOException {
        assertEquals(4, e.soma(2, 2));
    }

    public static Test suite() {
        TestSuite suite = new TestSuite("Testar apenas soma");
        suite.addTest(new OperacoesTest("testSoma"));
        return suite;
    }
}
```

JUnit para guiar o desenvolvimento

Cenário de Test-Driven Development (TDD)

1. Defina uma lista de tarefas a implementar
   - Quebre em tarefas mais simples se necessário
2. Escreva uma classe (test case) e implemente um método de teste para uma tarefa da lista.
3. Rode o JUnit e certifique-se que o teste falha
4. Implemente o código mais simples que rode o teste
   - Crie classes, métodos, etc. para que código compile
   - Código pode ser código feio, óbvio, mas deve rodar!
5. Refatore o código para remover a duplicação de dados
6. Escreva mais um teste ou refine o teste existente
7. Repita os passos 2 a 6 até implementar toda a lista

Test-Driven Development (TDD)

- Desenvolvimento guiado pelos testes
  - Só escreva código novo se um teste falhar
  - Refatore (altere o design) até que o teste funcione
  - Alternância: "red/green/refactor" - nunca passe mais de 10 minutos sem que a barra do JUnit fique verde.
- Técnicas
  - "Fake It Til You Make It": faça um teste rodar fazendo método retornar a constante esperada
  - Triangulação: abstraia o código apenas quando houver dois ou mais testes que esperam respostas diferentes
  - Implementação óbvia: se operações são simples, implemente-as e faça que os testes rodem

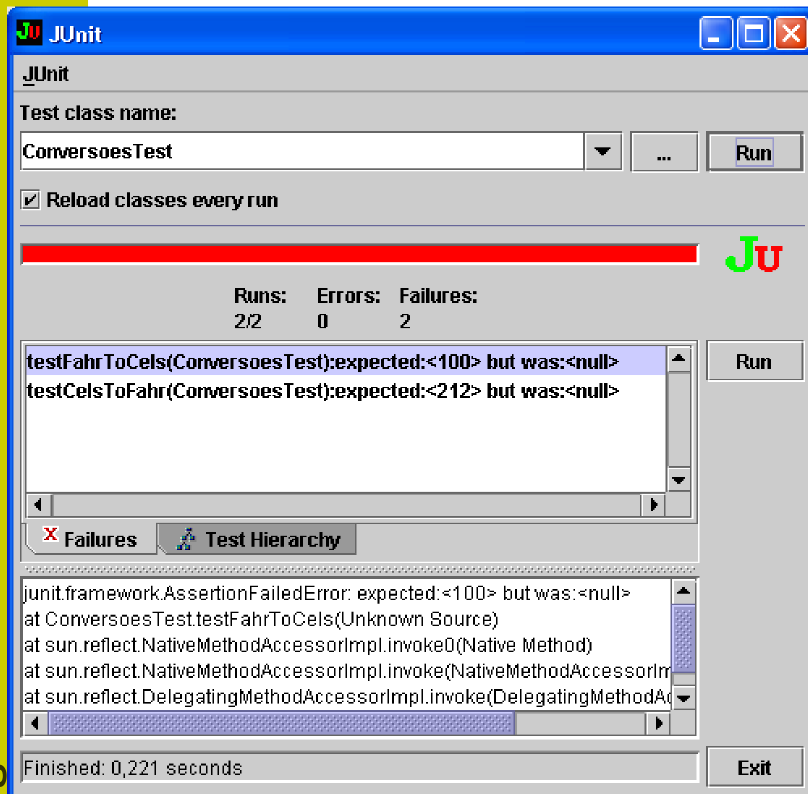
Exemplo de TDD: 1) Escreva os testes

```
import junit.framework.*;
import java.math.BigDecimal;

public class ConversoesTest extends TestCase {

    Conversoes conv = new Conversoes();

    public void testFahrToCels() {
        assertEquals(new BigDecimal(100),
                     conv.fahrToCels(new BigDecimal(212)));
    }

    public void testCelsToFahr() {
        assertEquals(new BigDecimal(212),
                     conv.celsToFahr(new BigDecimal(100)));
    }
}
```

2) Rode o JUnit

```
java -cp junit.jar junit.swingui.TestRunner ConversoesTest
```

- *JUnit não chega a rodar porque testes não compilam!*
  - *É preciso criar a classe Conversoes, contendo os métodos celsToFahr() e fahrToCels()*



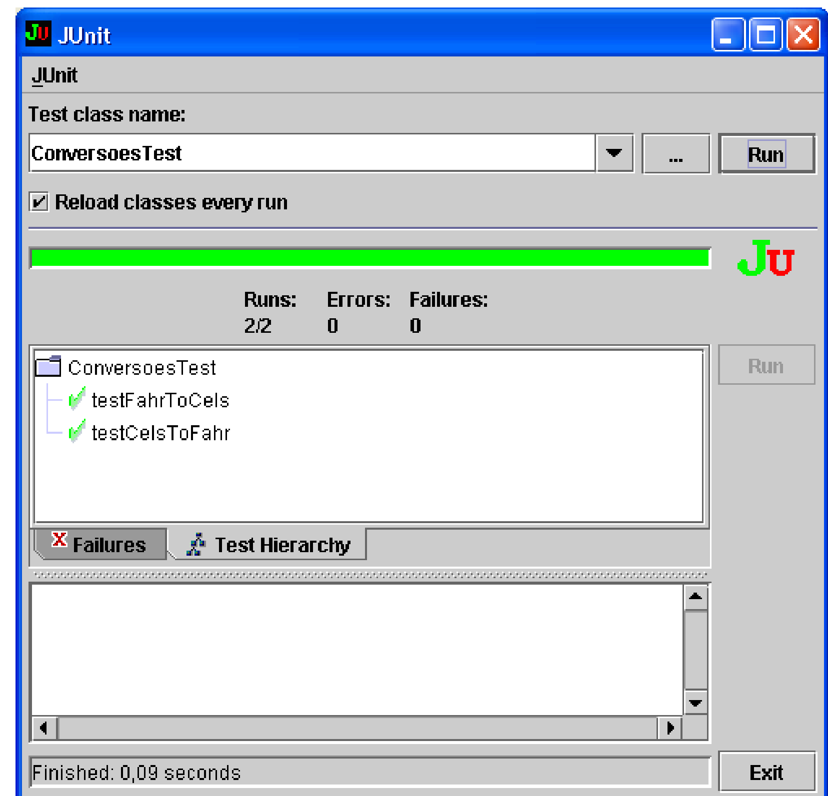

```
import java.math.BigDecimal;
public class Conversoes {

  public BigDecimal
    fahrToCels(BigDecimal fahr) {
      return null;
  }

  public BigDecimal
    celsToFahr(BigDecimal cels) {
      return null;
  }
}
```

Ainda assim, teste falha!

3) Uma classe que faz o teste passar

```
import java.math.BigDecimal;
public class Conversoes {

  public BigDecimal fahrToCels(BigDecimal fahr) {
    return new BigDecimal(100);
  }

  public BigDecimal
    celsToFahr(BigDecimal cels) {
      return new BigDecimal(212);
  }
}
```

- *O teste passa!*
  - *"Fake it till you make it"*
  - *Há duplicação de dados! É preciso eliminá-la!*

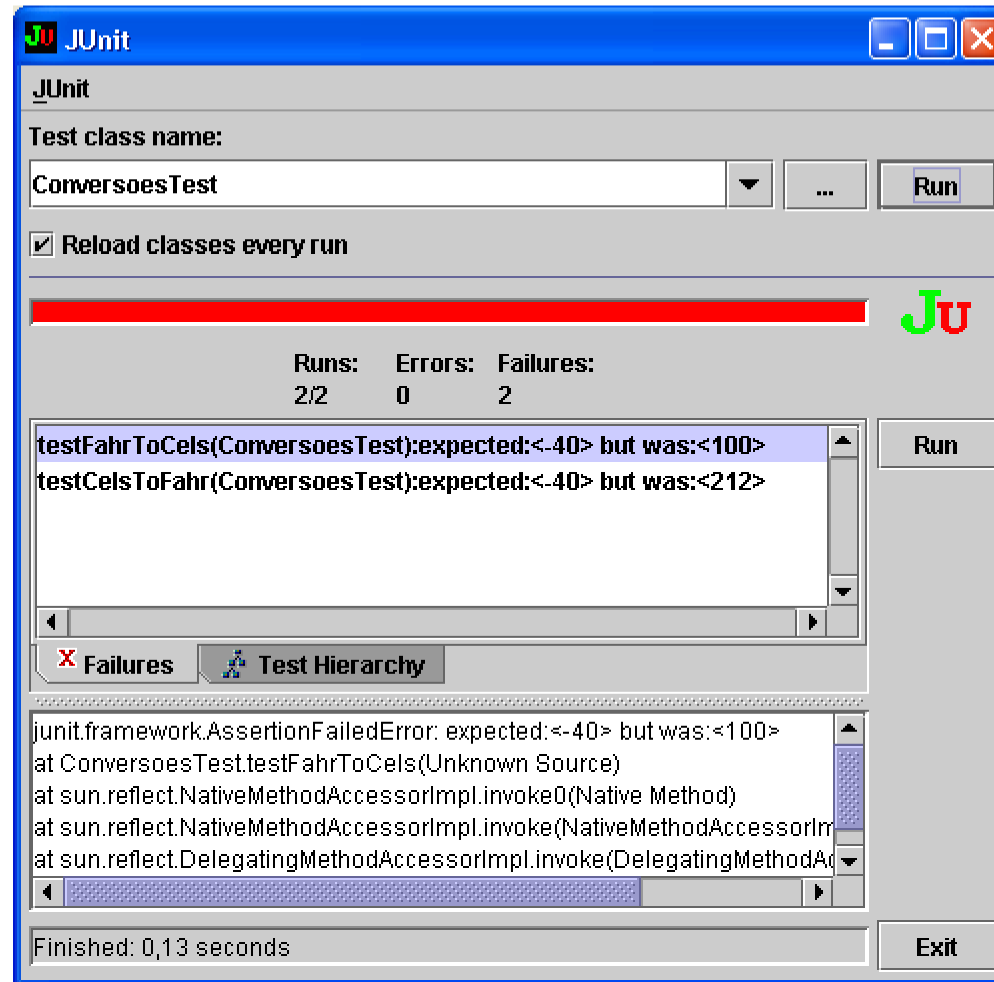


argonavis.com.br

4) Forçando nova falha (Triangulação)

```
import junit.framework.*;
import java.math.BigDecimal;

public class ConversoesTest extends TestCase {

    Conversoes conv = new Conversoes();

    public void testFahrToCels() {
        assertEquals(new BigDecimal(100),
                     conv.fahrToCels(new BigDecimal(212)));
        assertEquals(new BigDecimal(-40),
                     conv.fahrToCels(new BigDecimal(-40)));
    }

    public void testCelsToFahr() {
        assertEquals(new BigDecimal(212),
                     conv.celsToFahr(new BigDecimal(100)));
        assertEquals(new BigDecimal(-40),
                     conv.celsToFahr(new BigDecimal(-40)));
    }
}
```

5) Teste falha!



6) Uma boa implementação

```
import java.math.BigDecimal;

public class Conversoes {

        public BigDecimal fahrToCels(BigDecimal fahr) {
                double fahrenheit = fahr.doubleValue();
                double celsius = (5.0/9.0) * (fahrenheit - 32);
                return new BigDecimal(celsius);
        }

        public BigDecimal celsToFahr(BigDecimal cels) {
                double celsius = cels.doubleValue();
                double fahrenheit = (9.0/5.0) * celsius + 32;
                return new BigDecimal(fahrenheit);
        }
}
```

7) Outra boa implementação

- *Implementação pode ser melhorada sem quebrar o teste!*
  - *Teste garante que nova implementação cumpre os requisitos*

```
import java.math.BigDecimal;
public class Temperatura {
  private double celsius;
  private double fahrenheit;

  public void setCelsius(BigDecimal valor) {
    if (valor != null) {
      celsius = valor.doubleValue();
      fahrenheit = (9.0 * celsius) / 5.0 + 32;
    }
  }
  public void setFahrenheit(BigDecimal valor) {
    if (valor != null) {
      fahrenheit = valor.doubleValue();
      celsius = 5.0/9.0 * (fahrenheit - 32);
    }
  }
  public BigDecimal getCelsius() {
    return new BigDecimal(celsius);
  }
  public BigDecimal getFahrenheit() {
    return new BigDecimal(fahrenheit);
  }
}
```

JavaBean que converte temperaturas

```
import java.math.BigDecimal;
public class Conversoes_2 {
  Temperatura temp = new Temperatura();
  public BigDecimal
    fahrToCels(BigDecimal fahr) {
      temp.setFahrenheit(fahr);
      return temp.getCelsius();
  }

  public BigDecimal
    celsToFahr(BigDecimal cels) {
      temp.setCelsius(cels);
      return temp.getFahrenheit();
  }
}
```

Implementação usa o JavaBean

argonavis.com.br

TestCase Composite

- TestSuite pode ser usada para compor uma coleção de testes de um TestCase ou uma coleção de TestCases
- Composite pattern para TestCases:
  - Crie uma classe AllTests (convenção) em cada pacote
  - Adicione testes individuais, testcases e composições de testes em subpacotes

```
public class AllTests {
    public static Test suite() {
        TestSuite testSuite = new TestSuite("Roda tudo");
        testSuite.addTest(new ConversoesTest("testCelsToFahr"));
        testSuite.addTest(new ConversoesTest("testFahrToCels"));

        testSuite.addTest(OperacoesTest.suite());
        testSuite.addTestSuite(TransformacoesTest.class);

        testSuite.addTest(unidades.AllTests.suite());
        return testSuite;
    }
}
```

Testes individuais

Test cases inteiros

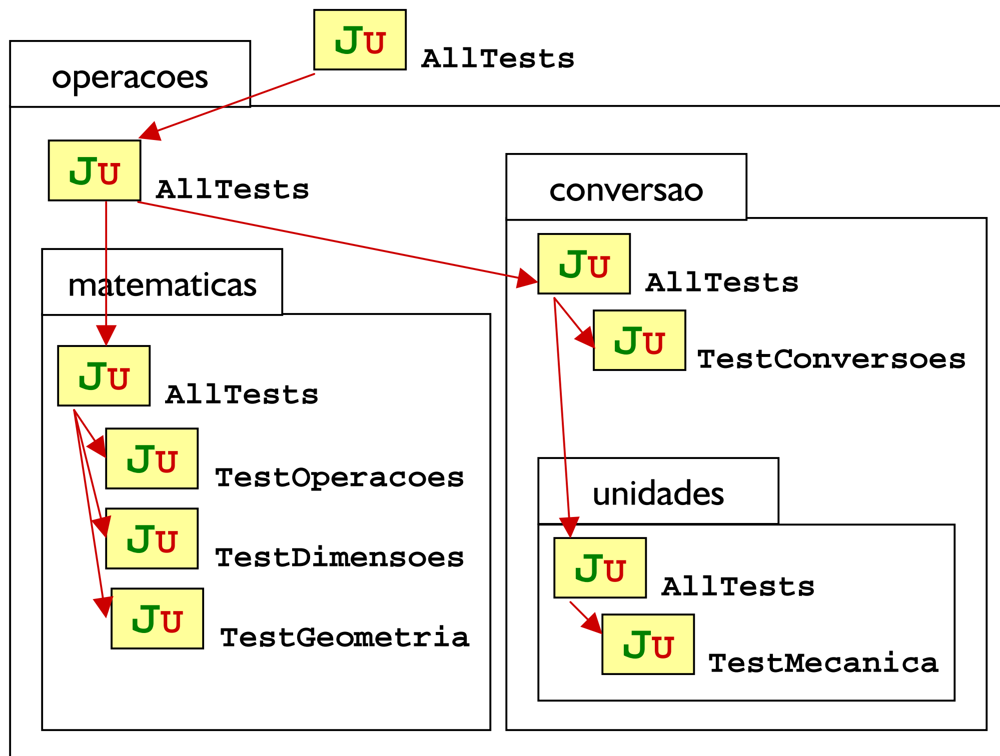
Coleção de test cases de subpacote

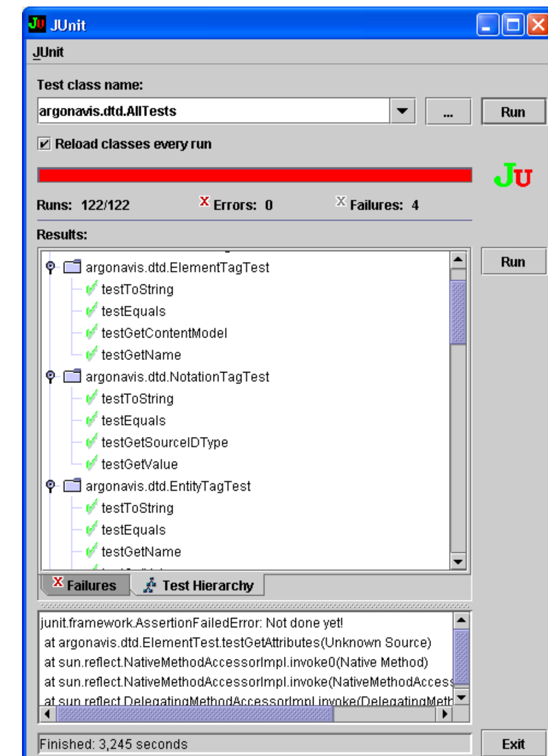
argonavis.com.br

Árvore de testes

- *Usando um Composite de TestCases, pode-se passar para o TestRunner a raiz dos TestCases e todos os seus componentes serão executados*

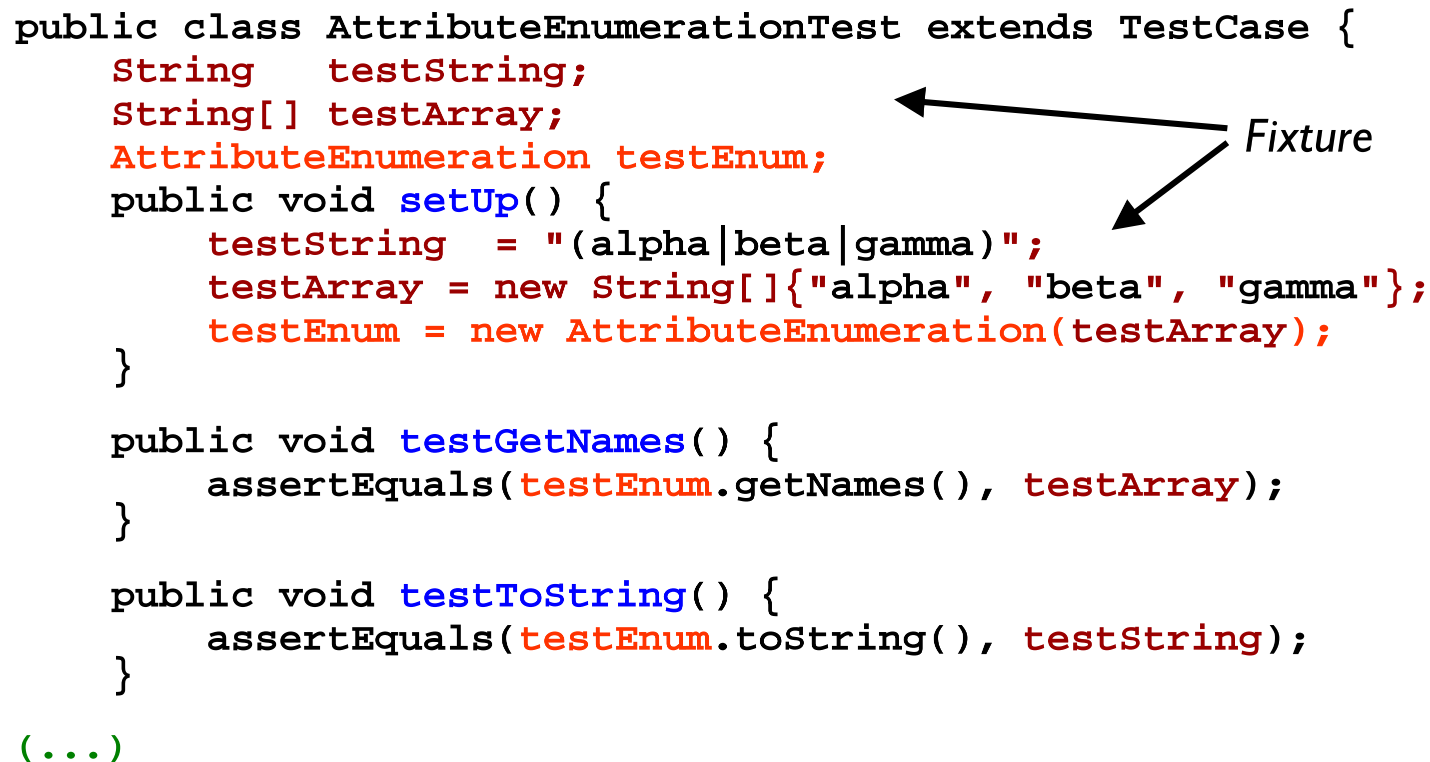
```
java -cp junit.jar junit.swingui.TestRunner AllTests
```





Fixtures

- São os dados reutilizados por vários testes
  - Inicializados no setUp() e destruídos no tearDown() (se for necessário)



```
public class AttributeEnumerationTest extends TestCase {
    String   testString;
    String[] testArray;
    AttributeEnumeration testEnum;
    public void setUp() {
        testString  = "(alpha|beta|gamma)";
        testArray = new String[]{"alpha", "beta", "gamma"};
        testEnum = new AttributeEnumeration(testArray);
    }

    public void testGetNames() {
        assertEquals(testEnum.getNames(), testArray);
    }

    public void testToString() {
        assertEquals(testEnum.toString(), testString);
    }

(...)
```


argonavis.com.br

Tamanho dos fixtures

- Fixtures devem conter apenas dados suficientes
  - Não teste 10 condições se três forem suficientes
  - Às vezes 2 ou 3 valores validam 99% da lógica
- Quando uma maior quantidade de dados puder ajudar a expor falhas, e esses dados estiverem disponíveis, pode-se usá-los no TestCase
  - Carregue-os externamente sempre que possível
- Extensão **JXUnit** (jxunit.sourceforge.net) permite manter dados de teste em arquivo XML (*.jxu)
  - Mais flexibilidade. Permite escrever testes rigorosos, com muitos dados
  - XML pode conter dados lidos de um banco

argonavis.com.br

Teste situações de falha

- *É tão importante testar o cenário de falha do seu codigo quanto o sucesso*
- *Método fail() provoca uma falha*
  - *Use para verificar se exceções ocorrem quando se espera que elas ocorram*
- *Exemplo*

```
public void testEntityNotFoundException() {
   resetEntityTable(); // no entities to resolve!
   try {
      // Following method call must cause exception!
      ParameterEntityTag tag = parser.resolveEntity("bogus");
      fail("Should have caused EntityNotFoundException!");
   } catch (EntityNotFoundException e) {
       // success: exception occurred as expected
   }
}
```

Asserções do J2SDK1.4

- *São expressões booleanas que o programador define para afirmar uma condição que ele acredita ser verdade*
  - *Asserções são usadas para validar código procedural (ter a certeza que um vetor tem determinado tamanho, ter a certeza que o programa não passou por determinado lugar)*
  - *Melhoram a qualidade do código: tipo de teste*
  - *Devem ser usadas durante o desenvolvimento e desligadas na produção (afeta a performance)*
  - *Não devem ser usadas como parte da lógica do código*
- *Asserções estão disponíveis no Java a partir do Java 1.4*
  - *Nova palavra-chave:* `assert`
  - *É preciso compilar usando a opção -source 1.4:*

    ```
    > javac -source 1.4 Classe.java
    ```
  - *Para executar, é preciso habilitar asserções (enable assertions):*

    ```
    > java -ea Classe
    ```

Asserções do JUnit vs. asserções do Java

- Asserções do J2SDK 1.4 são usadas dentro do código
  - Podem incluir testes dentro da lógica procedural de um programa

```
if (i%3 == 0) {
   doThis();
} else if (i%3 == 1) {
   doThat();
} else {
   assert i%3 == 2: "Erro interno!";
}
```

  - Provocam um AssertionError quando falham (que pode ser encapsulado pelas exceções do JUnit)
- Asserções do JUnit são usadas em classe separada (TestCase)
  - Não têm acesso ao interior dos métodos (verificam se a interface dos métodos funciona como esperado)
- Asserções do J2SDK1.4 e JUnit são complementares
  - Asserções do JUnit testam a interface dos métodos
  - `assert` testa trechos de lógica dentro dos métodos

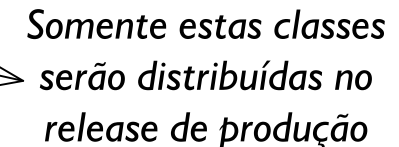
Limitações do JUnit

- Acesso aos dados de métodos sob teste
  - Métodos private e variáveis locais não podem ser testadas com JUnit
  - Dados devem ser pelo menos package-private (friendly)
- Possíveis soluções com alteração do design
  - Isolar em métodos private apenas código inquebrável
  - Transformar métodos private em package-private
    - Desvantagem: redução do encapsulamento
    - Classes de teste devem estar no mesmo pacote que as classes testadas para que JUnit tenha acesso a elas
- Solução usando extensão do JUnit (open-source)
  - **JUnitX**: usa reflection para ter acesso a dados private
  - http://www.extreme-java.de/junitx/index.html

Onde guardar os TestCases

- *Estratégia recomendada é colocá-los nos mesmos diretórios (pacotes) onde estão as fontes testadas*
  - *Podem ser separados facilmente do código de produção durante a distribuição: Ant*
  - *Testes no mesmo pacote terão acesso e poderão testar membros package-private*
- *Exemplo de estrutura de testes*

```
pacote.AllTests
  pacote.subpacote.AllTests
  pacote.subpacote.Primeiro
  pacote.subpacote.PrimeiroTest
  pacote.subpacote.Segundo
  pacote.subpacote.SegundoTest
    pacote.subpacote.sub2.AllTests
    pacote.subpacote.sub2.Um
    pacote.subpacote.sub2.UmTest
```



Como escrever bons testes

- *JUnit facilita bastante a criação e execução de testes, mas elaborar bons testes exige mais*
  - *O que testar? Como saber se testes estão completos?*
- *"Teste tudo o que pode falhar" [2]*
  - *Métodos triviais (get/set) não precisam ser testados.*
  - *E se houver uma rotina de validação no método set?*
- *É melhor ter testes a mais que testes a menos*
  - *Escreva testes curtos (quebre testes maiores)*
  - *Use assertNotNull() (reduz drasticamente erros de NullPointerException difíceis de encontrar)*
  - *Reescreva e altere o design de seu código para que fique mais fácil de testar: promove design melhor!*

Como descobrir testes?

- Listas de tarefas (to-do list)
  - Comece implementando os testes mais simples e deixe os testes "realistas" para o final
  - Requerimentos, use-cases, diagramas UML: rescreva os requerimentos em termos de testes
  - Quebre requisitos complexos em pedaços menores
- Bugs revelam testes
  - Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)!
- Descoberta de testes é atividade de análise e design
  - Sugerem nomes e estrutura de classes da solução
  - Permitem que se decida sobre detalhes de implementação após a elaboração do teste

Testes como documentação

- Testes são documentação executável
  - Execute-os periodicamente para mantê-los atualizados
  - Use nomes significativos
  - Mantenha-os simples!
- Todas as asserções do JUnit possuem um argumento para descrever o que está sendo testado
  - Quando presente é o primeiro argumento
  - A mensagem passada será mostrada em caso de falha
  - Use, sempre que possível!

```
assertEquals("Array não coincide!", esperado, testArray);
assertNotNull("obj é null!", obj);
assertTrue("xyz() deveria retornar true!", a.xyz());
```

Ant + JUnit

- Ant: ferramenta para automatizar processos de construção de aplicações Java
- Pode-se executar todos os testes após a integração com um único comando:
  - `ant roda-testes`
- Com as tarefas <junit> e <junitreport> é possível
  - executar todos os testes
  - gerar um relatório simples ou detalhado, em diversos formatos (XML, HTML, etc.)
  - executar testes de integração
- São tarefas opcionais. É preciso ter no `$ANT_HOME/lib`
  - optional.jar (distribuído com Ant)
  - junit.jar (distribuído com JUnit)

Exemplo: <junit>

```
<target name="test" depends="build">
   <junit printsummary="true" dir="${build.dir}"
          fork="true">
      <formatter type="plain" usefile="false" />
      <classpath path="${build.dir}" /
      <test name="argonavis.dtd.AllTests" />
   </junit>
</target>
```

Formata os dados na tela (plain)
Roda apenas arquivo AllTests

```
<target name="batchtest" depends="build" >
   <junit dir="${build.dir}" fork="true">
      <formatter type="xml" usefile="true" />
      <classpath path="${build.dir}" />
      <batchtest todir="${test.report.dir}">
         <fileset dir="${src.dir}">
             <include name="**/*Test.java" />
             <exclude name="**/AllTests.java" />
         </fileset>
      </batchtest>
   </junit>
</target>
```

Gera arquivo XML
Inclui todos os arquivos que terminam em TEST.java

<junitreport>

- *Gera um relatório detalhado (estilo JavaDoc) de todos os testes, sucessos, falhas, exceções, tempo, ...*

```
<target name="test-report" depends="batchtest" >
  <junitreport todir="${test.report.dir}">
    <fileset dir="${test.report.dir}">
       <include name="TEST-*.xml" />
    </fileset>
    <report todir="${test.report.dir}/html"
            format="frames" />
  </junitreport>
</target>
```

Usa arquivos XML gerados por <formatter>

Unit Test Results. - Microsoft Internet Explorer fornecido por UOL

Packages

argonavis.dtd
argonavis.dtd.par
argonavis.dtd.tag

Classes

AttlistTagParserTe
AttlistTagTest
AttributeEnumera
AttributeNotationE
AttributeParserTe
AttributeTest
AttributeTypeTest
ChildElementTest
CommentTagTest
ConditionalInclusi
ContentModelItem
ContentModelPars

Unit Test Results

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
122	4	0	96.72%	59.100

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)
argonavis.dtd	70	0	4	43.070
argonavis.dtd.parsers	26	0	0	7.910
argonavis.dtd.tagdata	26	0	0	8.120

Exercícios

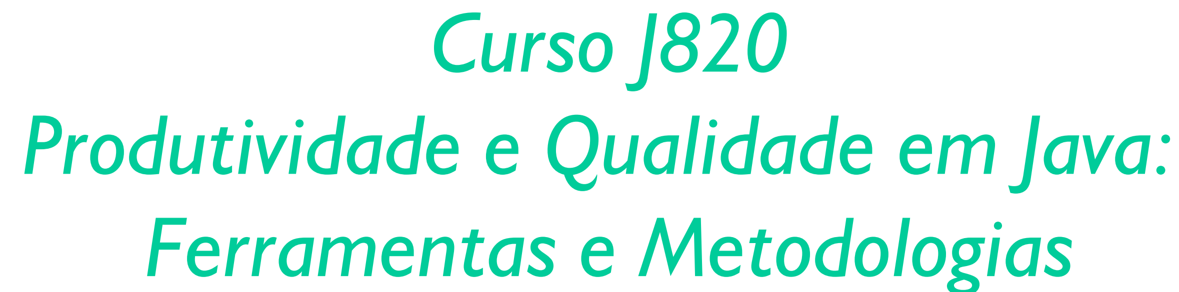
- *1. Escreva testes para os exemplos do capítulo 04. Use a seguinte metodologia:*
  - *Escreva uma classe TestCase para cada classe*
  - *Escreva métodos testXXX() para cada método de cada classe*
  - *Inclua código em cada método testXXX() para chamar o método, passar parâmetros de teste e avaliar seu funcionamento*
- *2. Transforme o seguinte requisito em um teste*
  - *fatorial(0) = 1, fatorial(1) = 1, fatorial(2) = 2, fatorial(3) = 6, fatorial(4) = 24, fatorial(5) = 120*

  Escreva código para fazer o teste passar.

argonavis.com.br

Fontes

[1] *Documentação JUnitPerf*. *junitperf.sourceforge.net*
[2] *Hightower/Lesiecki. Java Tools for eXtreme Programming. Wiley, 2002*
[3] *Eric Burke & Brian Coyner. Java eXtreme Programming Cookbook. O'Reilly, 2003*



Revisão 1.1

© 2002, 2003, Helder da Rocha
(helder@acm.org)