argonavis.com.br

Desenvolvimento guiado por testes com JUnit

Helder da Rocha (helder@acm.org)

Objetivos

Apresentar o JUnit

Propor uma metodologia de desenvolvimento Java guiada por testes utilizando o JUnit

Discutir melhores práticas, padrões e soluções

Assuntos abordados

Introdução ao JUnit e testes de unidade

Test-driven development (TDD) e exemplos

Test patterns: Composite (suites), Fixtures, Fail tests, Mock objects

Asserções

Melhores práticas

Testes de integração em aplicações J2EE

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*

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

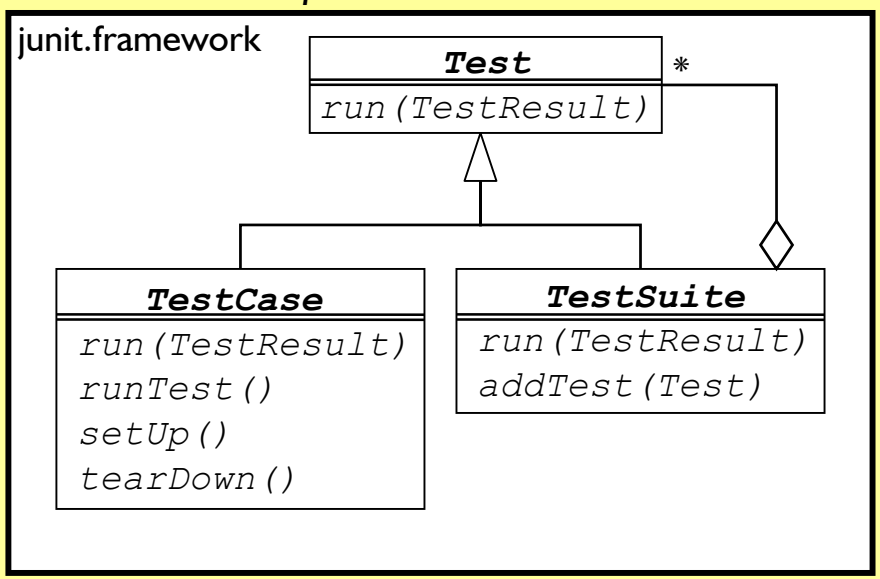
- *Testes de unidade*
  - *Testam unidades de lógica. Em linguagens orientadas a objetos, unidades geralmente representam métodos, mas podem também representar um objeto inteiro ou ainda um estado de um método*
  - *Ignoram condições ou dependências externas. Testes de unidade usam dados suficientes 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.*

O que é JUnit?

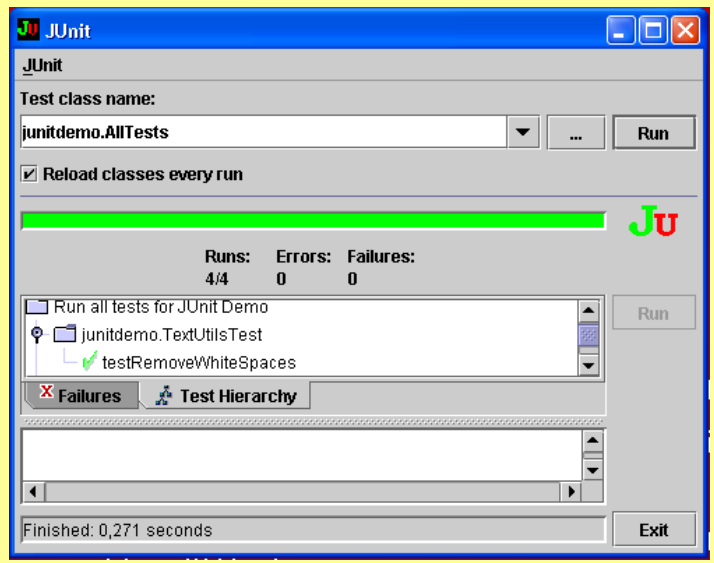
- *Um **framework** que facilita o desenvolvimento e execução de **testes de unidade** em código Java*
  - *Uma API para construir os testes: junit.framework.**
  - *Aplicações para executar testes: TestRunner*

junit.jar

Principais classes da API



Aplicação TestRunner Gráfica



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() {...}
```

Usará reflection para descobrir métodos que começam com "test"

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

```
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?

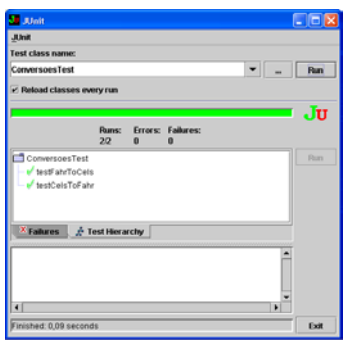
- *Exemplo de test case com **um** teste:*

```
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"*

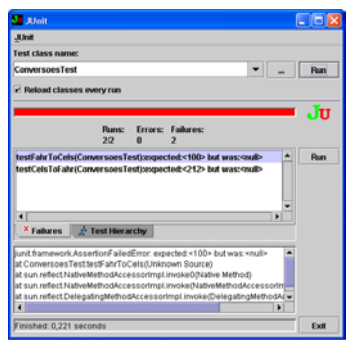
- *Para executar (aplicação gráfica do JUnit)*

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



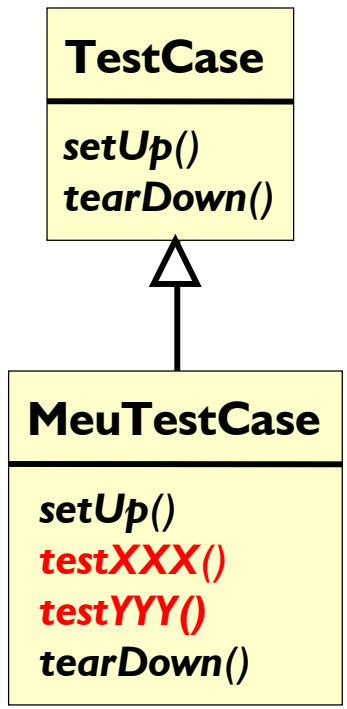
Passou!

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*

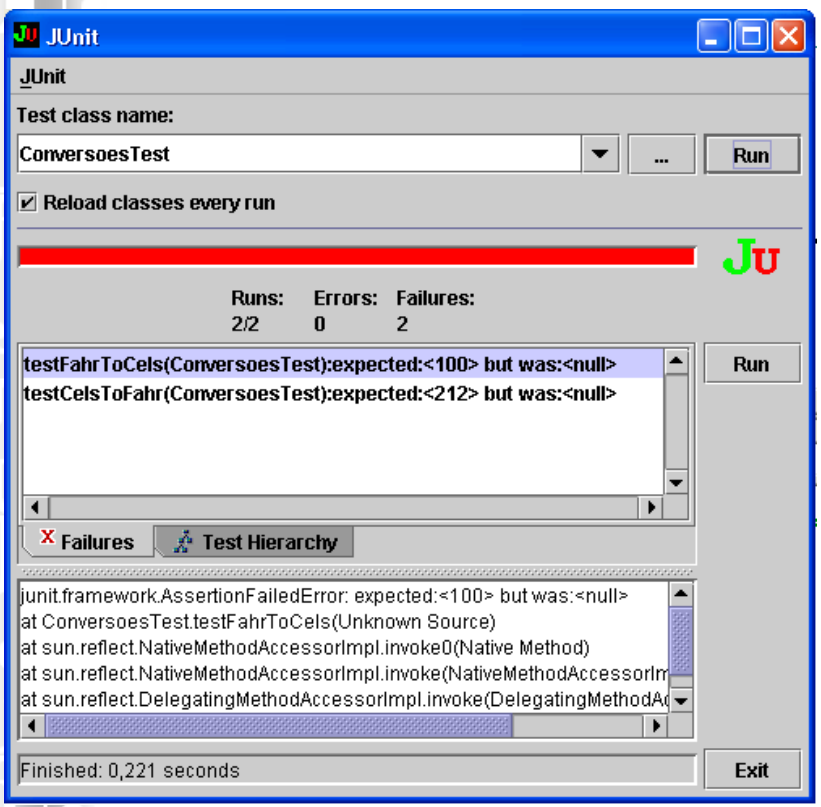
Exemplo de TDD: 1) Escreva os testes

```
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()*

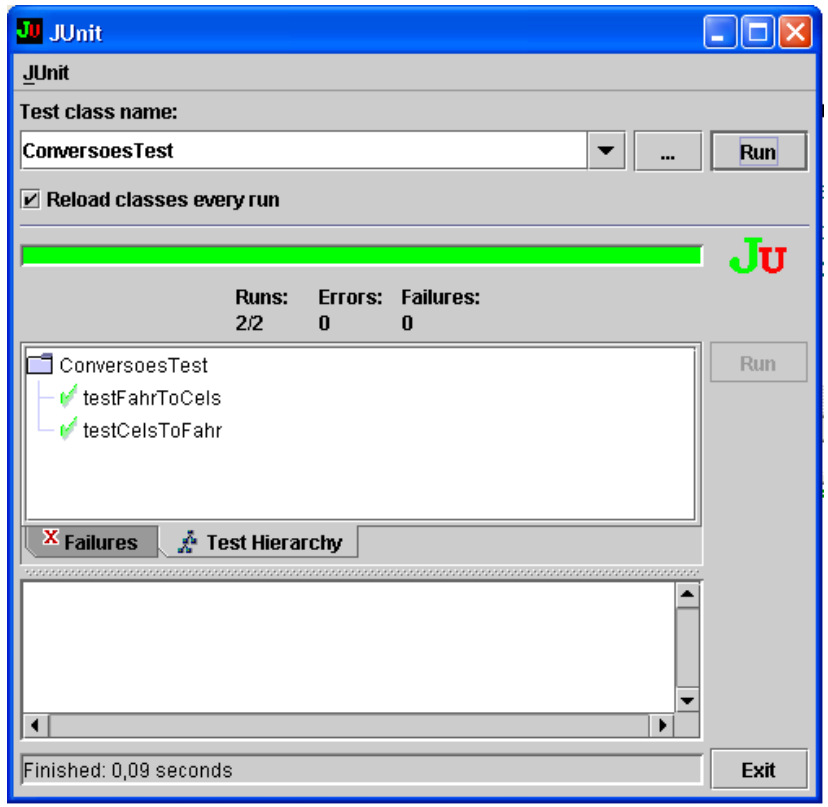



```
import java.math.BigDecimal;
public class Conversoes {

  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);
  }
}
```

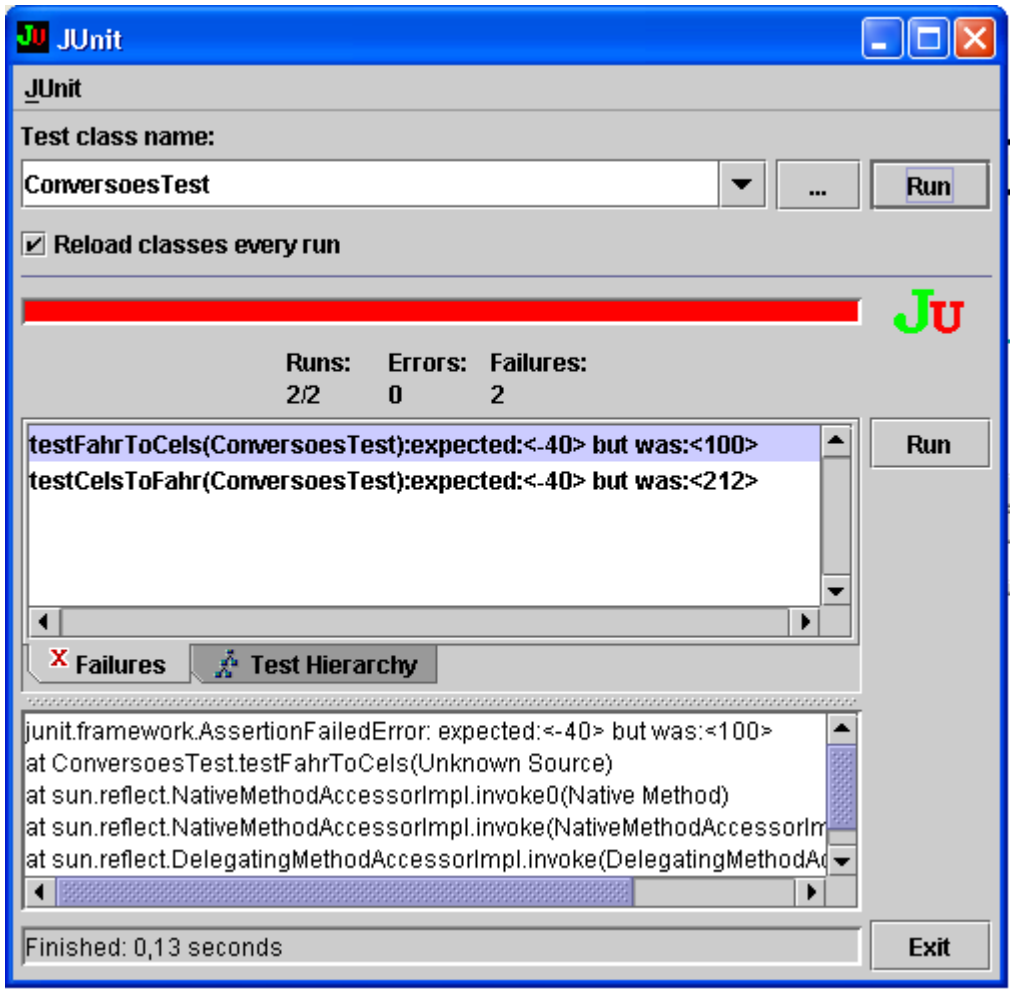


- *O teste passa!*
  - *"Fake it till you make it"*
  - *Há duplicação de dados! É preciso eliminá-la!*

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

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

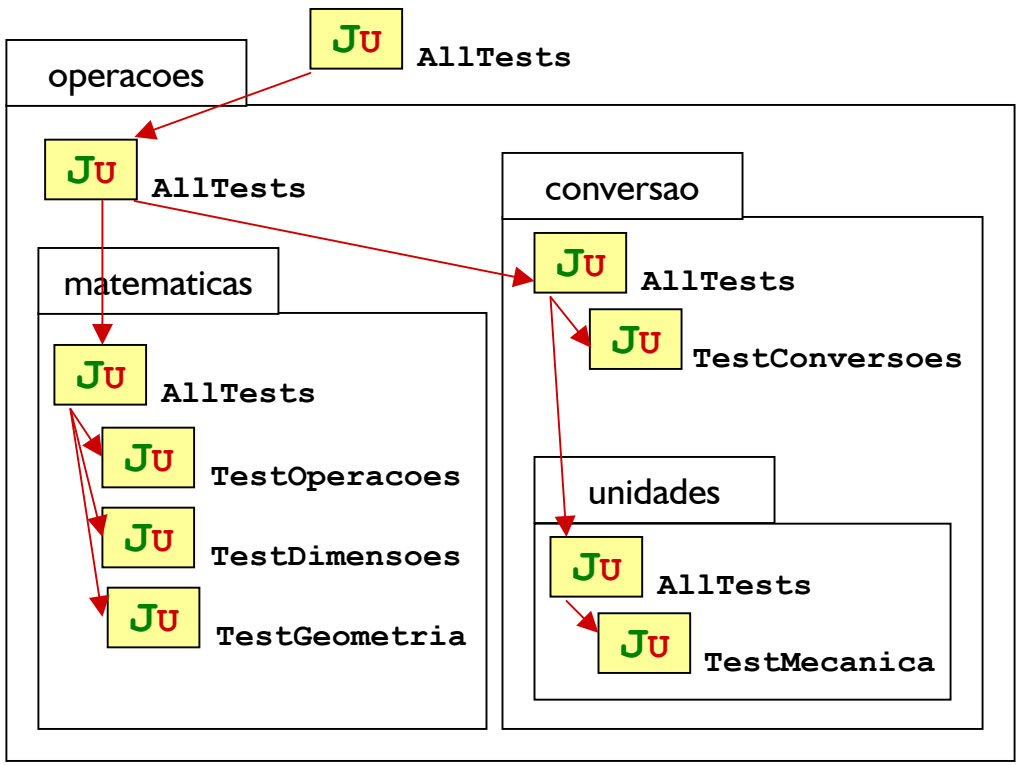
Test cases inteiros

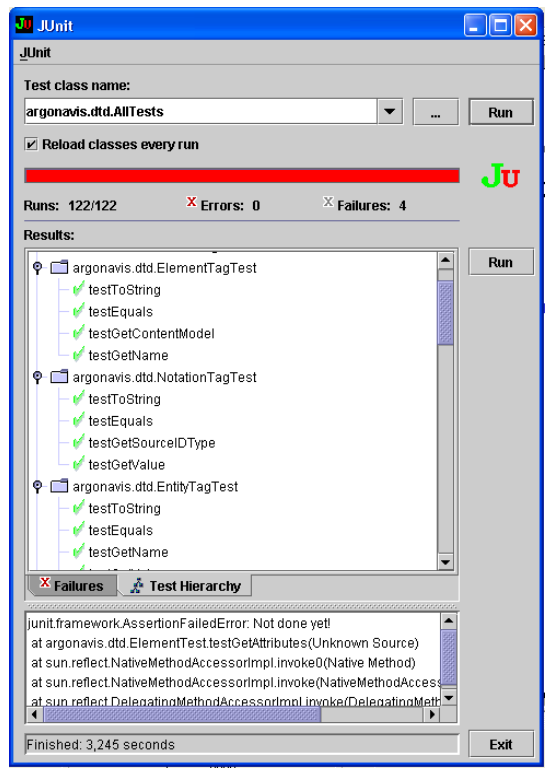
Coleção de test cases de subpacote

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
```

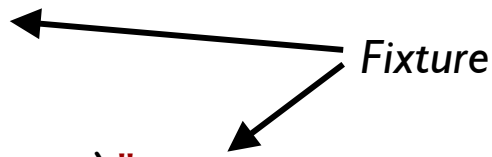




Fixtures

- *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);
    }

(...)
```



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*

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*

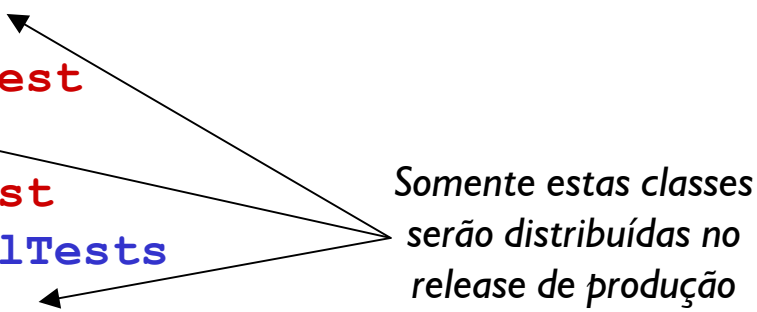
Limitações do JUnit

- 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());
```

Como lidar com testes difíceis

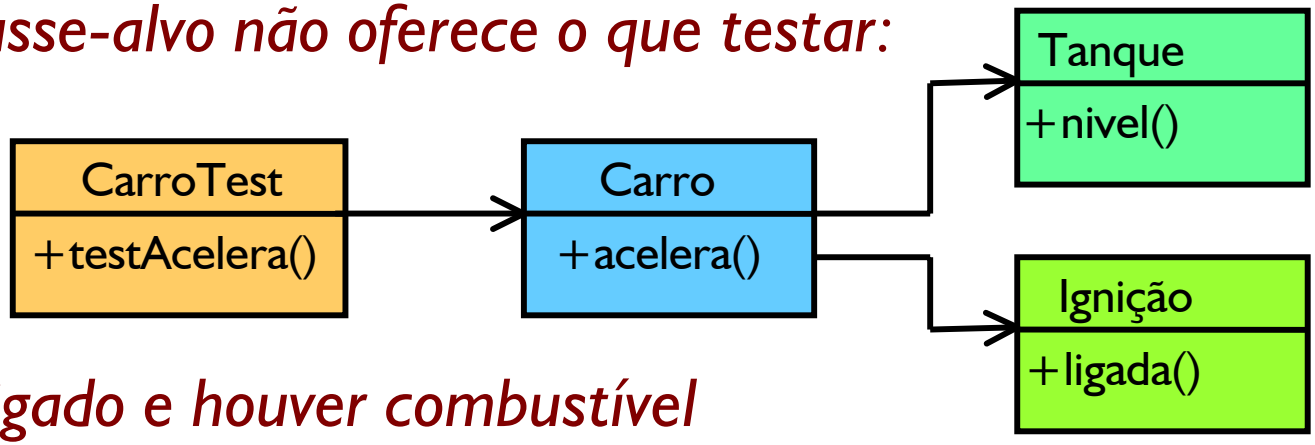
- Testes devem ser simples e suficientes
  - Comece com testes mais importantes
  - Sempre pode-se escrever novos testes, quando necessário
- Não complique
  - Não teste o que é responsabilidade de outra classe/método
  - Assuma que outras classes e métodos funcionam
- Testes difíceis (ou que parecem difíceis)
  - Aplicações gráficas: eventos, layouts, threads
  - Objetos inacessíveis, métodos privativos, Singletons
  - Objetos que dependem de outros objetos
  - Objetos cujo estado varia devido a fatores imprevisíveis
- Soluções
  - Alterar o design da aplicação para facilitar os testes
  - Simular dependências usando proxies e stubs

Como testar GUIs

- *O que testar?*
  - *Assumir que GUI (Swing, AWT, etc.) funciona*
  - *Concentrar-se na lógica de negócio e não na UI*
- *Como testar?*
  - *"Emagrecer" o código para reduzir a chance de falha*
  - *Usar MVC: isolar lógica de apresentação e controle*
  - *Separar código confiável do código que pode falhar*
  - *Usar mediadores (Mediator pattern) para intermediar interações entre componentes*
- *Exemplo: event handlers*
  - *Devem ter 0% de lógica de negócio: "A Stupid GUI is an Unbreakable GUI" (Robert Koss, Object Mentor)* **[5]**
  - *Responsabilidades delegadas a mediadores testáveis*

Dependência de código-fonte

- *Problema*
  - *Como testar componente que depende do código de outros componentes?*
  - *Classe-alvo não oferece o que testar:*



  - *Se ligado e houver combustível método* `void acelera()` *deve funcionar*
  - *Como saber?*

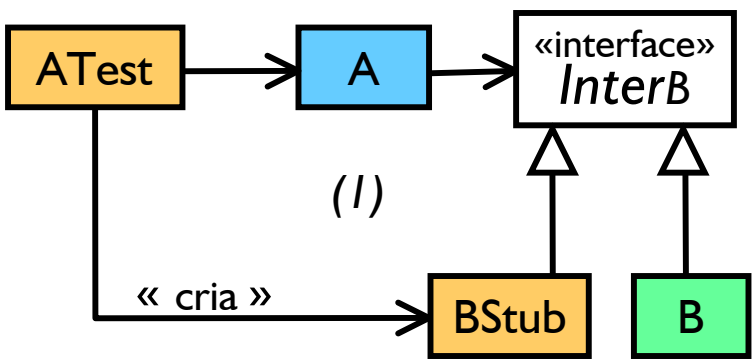
```
public void testAcelera() {
    Carro carro =
                new Carro();
    carro.acelera();
    assert???(???);
}
```

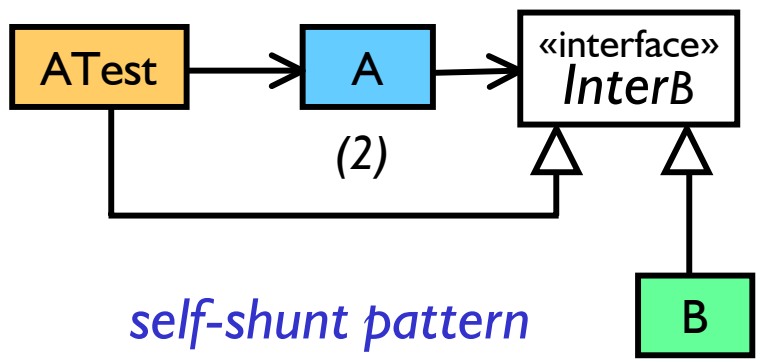

Fonte: www.objectmentor.com, 2002

Stubs: objetos "impostores"

- *É possível remover dependências de código-fonte refatorando o código para usar interfaces*

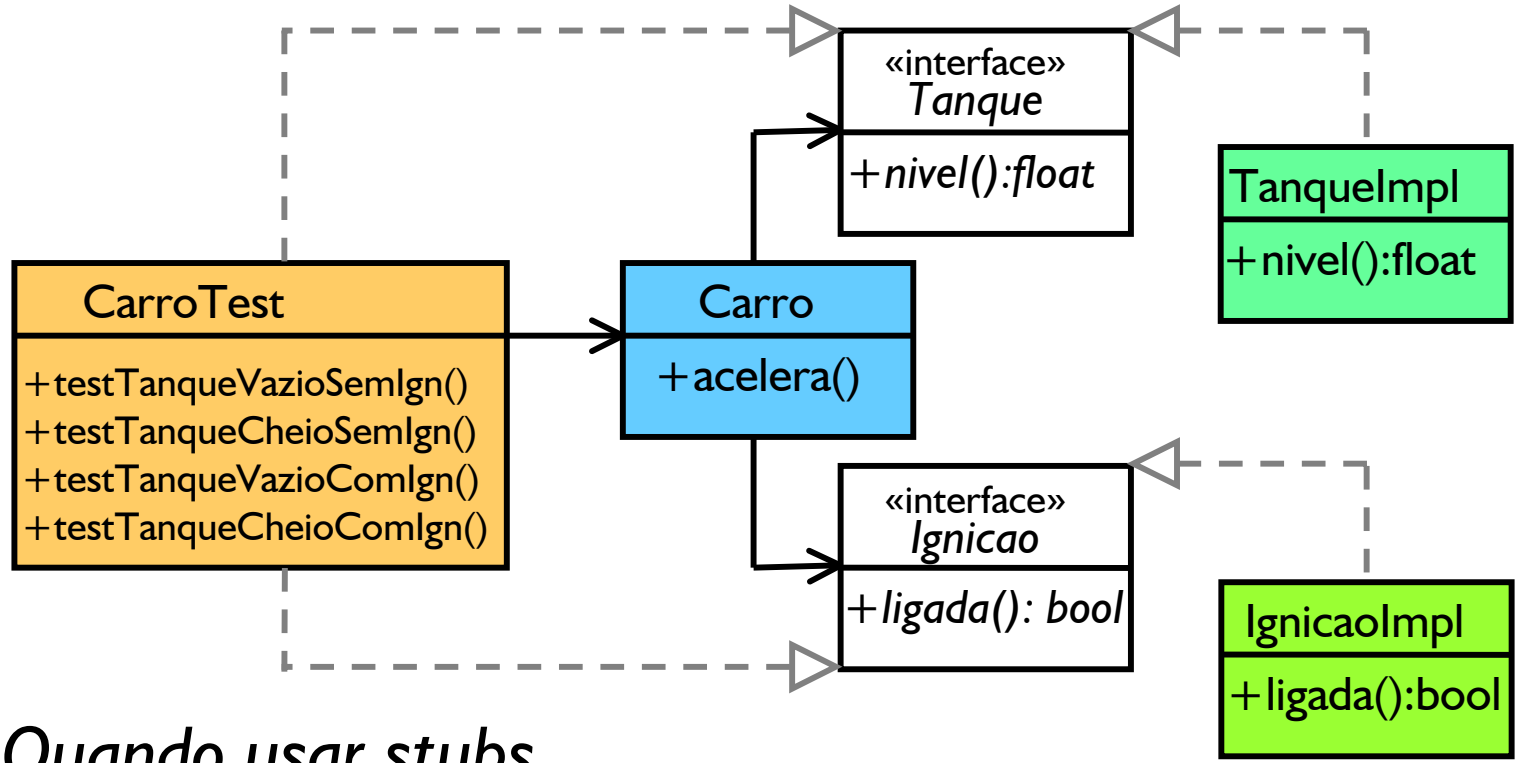
- *Agora B pode ser substituída por um stub*
  - *BStub está sob controle total de ATest (1)*
  - *Em alguns casos, ATest pode implementar InterB (2)*





Fonte: www.objectmentor.com, 2002

Dependência: solução usando stubs



- *Quando usar stubs*
  - *Dependência não existe ainda (não está pronta)*
  - *Dependências tem estado mutante, imprevisível ou estão indisponíveis durante o desenvolvimento*
    - *BDs, servidores de aplicação, servidores Web, hardware*

Fonte: www.objectmentor.com, 2002

Dependências de servidores

- *Usar **stubs** para simular serviços e dados*
  - *É preciso implementar classes que devolvam as respostas esperadas para diversas situações*
  - *Complexidade muito grande da dependência pode não compensar investimento (mas não deixe de fazer testes por causa disto!)*
  - *Vários tipos de stubs: mock objects, self-shunts.*
- *Usar **proxies** (mediadores) para serviços reais*
  - *Oferecem interface para simular comunicação e testa a integração real do componente com seu ambiente*
  - *Não é teste unitário: teste pode falhar quando código está correto (se os fatores externos falharem)*

Mock Objects

- ***Mock objects*** *(MO) é uma estratégia similar ao uso de stubs mas que não implementa nenhuma lógica*
  - *Um mock object não é, portanto, um stub, pois não simula o funcionamento do objeto em qualquer situação*
- *Comportamento é controlado pela classe de teste que*
  - *Define comportamento esperado (valores retornados, etc.)*
  - *Passa MO configurado para objeto a ser testado*
  - *Chama métodos do objeto (que usam o MO)*
- *Implementações open-source que facilitam uso de MOs*
  - *EasyMock (tammofreese.de/easymock/) e MockMaker (www.xpdeveloper.com) geram MOs a partir de interfaces*
  - *Projeto MO (mockobjects.sourceforge.net) coleção de mock objects e utilitários para usá-los*

Exemplo de Mock Object

Interface

```
public interface Dependencia {
   public boolean metodo();
}
```

Implementação "mock" da interface gerada pelo mock maker

```
import mockmaker.ReturnValues;
import com.mockobjects.*;

public class MockDependencia implements Dependencia{
   private ExpectationCounter metodoCalls = new ... ;
   private ReturnValues metodoReturnValues = new ... ;
   public void setExpectedMetodoCalls(int calls){
      metodoCalls.setExpected(calls);
   }
   public void setupMetodo(boolean arg){
      metodoReturnValues.add(new Boolean(arg));
   }
   public void verify(){
      metodoCalls.verify();
   }
   public boolean metodo(){
      metodoCalls.inc();
      Object nextReturnValue = metodoReturnValues.getNext();
      return ((Boolean) nextReturnValue).booleanValue();
   }
}
```

métodos para configuração do mock object e resultados esperados

implementação

Usando um Mock Object

Objeto que usa Dependência. Em tempo de desenvolvimento, ela será implementada por um Mock object.

```
public class DepClient {
    private Dependencia dep;
    public DepClient(Dependencia dep) {
        this.dep = dep;
    }
    public String operacao(String texto) {
        return dep.metodo(texto);
    }
}
```

```
import junit.framework.*;

public class DepClientTest extends TestCase {
    private DepClient client;
    private MockDependencia mockObject; // implements Dependencia!

    public void setUp() {
        mockObject = new MockDependencia();
        client = new DepClient(mockObject);
        mockObject.setExpectedMetodoCalls(1);
        mockObject.setupMetodo("DADOS"); // define comportamento
    }

    public void testMetodo() throws java.io.IOException {
        String result = client.operacao("dados");
                assertEquals("DADOS", result);
        mockObject.verify();
    }
}
```

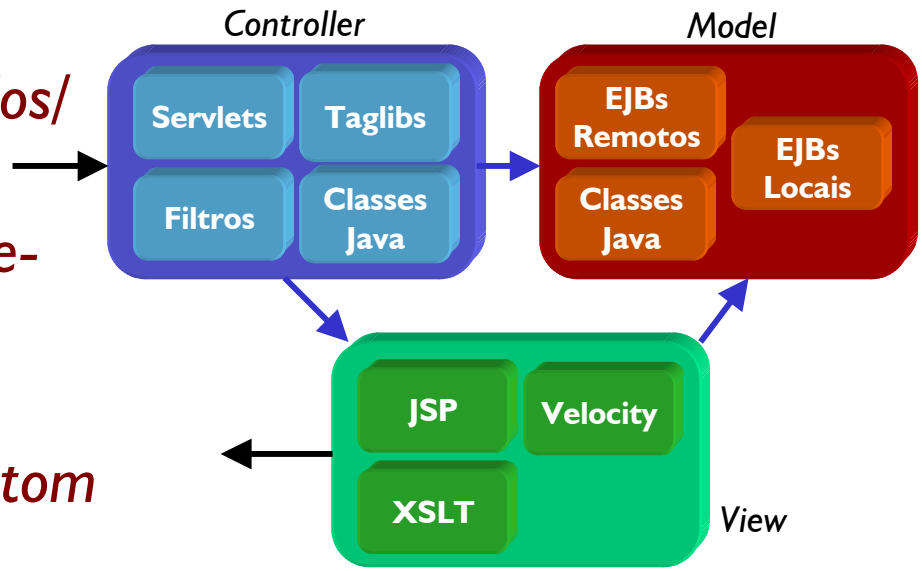
Test Case que usa um Mock Object para simular uma dependência real

Cactus: framework para J2EE

- *Testa componentes J2EE no próprio container*
  - *Componentes Web (Camada de Controle)*
  - *Camada EJB (Model) e cliente (View): indiretamente*
- *Cactus estende o JUnit framework*
  - *Execução dos testes é realizada de forma idêntica*
  - *TestCases são construídos sobre uma subclasse de junit.framework.TestCase*
- *Por que usar?*
  - *Para testes de integração (JUnit testa lógica local)*
  - *Aplicações J2EE possuem muitas dependências (é preciso simular a rede, o servidor, o EJB container, etc.)*
  - *Mock objects podem ficar muito complicados*
  - *Testar a integração com o servidor pode ser mais fácil*

Para que serve?

- *Use Cactus para testar a integração de aplicações Web (servlets, JSP, filtros) e aplicações EJB com o container*
- *Arquitetura do Cactus*
  - ***M**: EJB (Modelo de dados/ lógica de negócios)*
  - ***V**: JSPs (camada de apresentação: View, através de controladores)*
  - ***C**: Servlets, filtros e custom tags (Controladores)*



- *Se não precisar testar a integração, use JUnit*
  - *Pode-se usar ambos em conjunto, uma vez que Cactus é extensão do JUnit e usa a mesma interface*

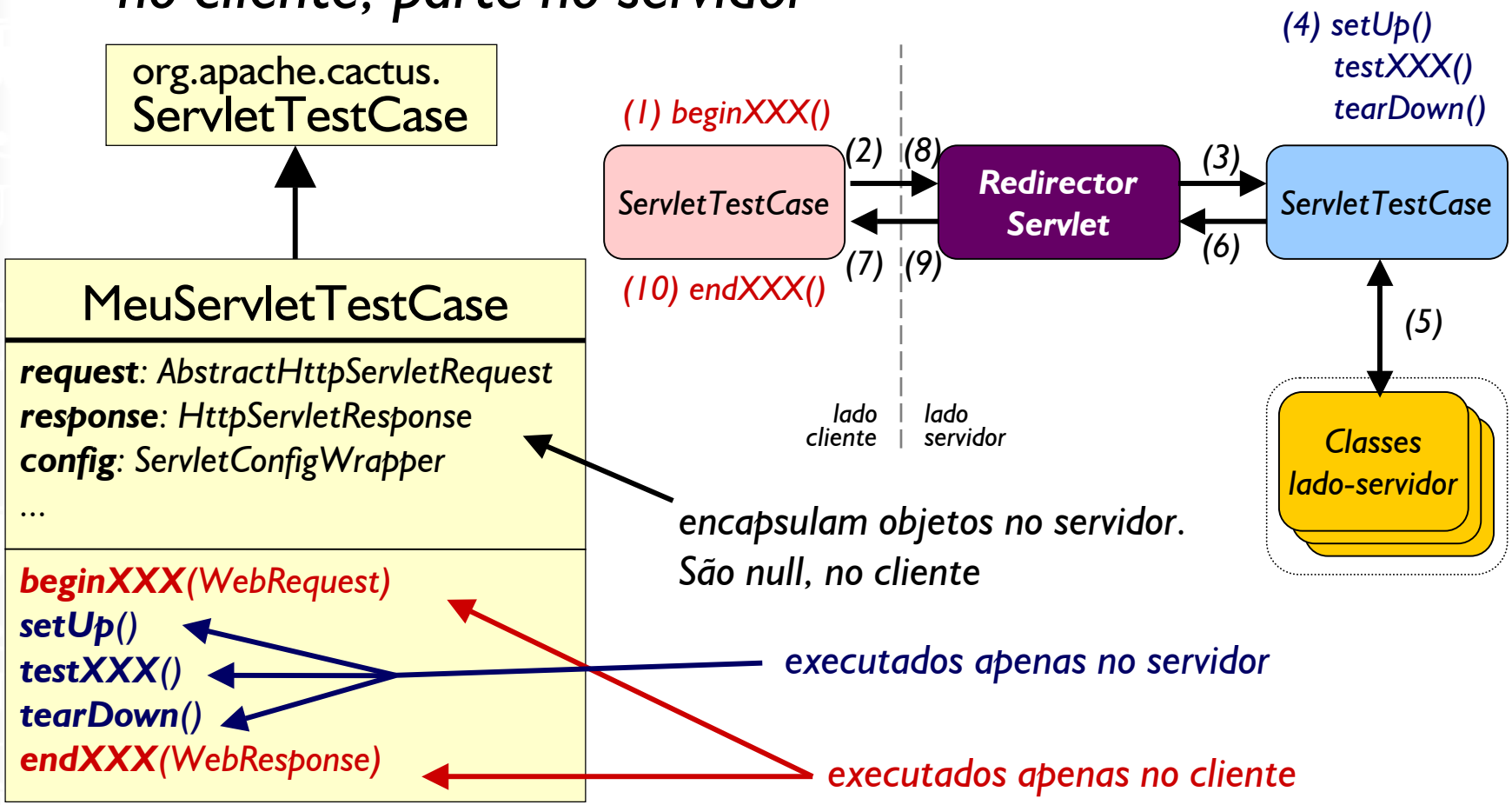
Ilustração: Manual do Cactus

Como funciona?

- Cactus utiliza os test cases simultaneamente no cliente e no servidor: duas cópias
  - Uma cópia é instanciada pelo servlet container
  - Outra cópia é instanciada pelo JUnit
- Comunicação com o servlet container é feita através de um proxy (Redirector)
  - JUnit envia requisições via HTTP para proxy
  - Proxy devolve resultado via HTTP e TestRunner mostra
- Há três tipos de proxies:
  - ServletRedirector: para testar servlets
  - JSPRedirector: para testar JSP custom tags
  - FilterRedirector: para testar filtros de servlets

Arquitetura

- *Parte da mesma classe (ServletTestCase) é executada no cliente, parte no servidor*



ServletTestCase (ou similar)

- Para cada método XXX() a ser testado, pode haver:
  - Um `beginXXX()`, para inicializar a requisição do cliente
    - Encapsulada em um objeto WebRequest a ser enviado ao servidor
  - Um `testXXX()`, para testar o funcionamento do método no servidor (deve haver ao menos um)
  - Um `endXXX()`, para verificar a resposta do servidor
    - Devolvida em um objeto WebResponse retornada pelo servidor
- Além desses três métodos, cada TestCase pode conter
  - `setUp()`, opcional, para inicializar objetos no servidor
  - `tearDown()`, opcional, para liberar recursos no servidor
- Os métodos do lado do servidor têm acesso aos mesmos objetos implícitos disponíveis em um servlet ou página JSP: request, response, etc.

Exemplo de uso de Cactus

- *O objetivo deste servlet é*
  - *1) gravar qualquer parâmetro que receber na sessão (objeto session)*
  - *2) devolver uma página contendo os pares nome/valor em uma tabela*
  - *3) imprimir resposta em caixa-alta*

```
public void doGet(...) throws IOException {
    Enumeration e = request.getParameterNames();
    HttpSession s = request.getSession();
    while (e.hasMoreElements()) {
       String param = (String)e.nextElement();
       s.setAttribute(param, request.getParameter(param));
    }
    writer.println("<html><body><table border='1'>");
    String str = "";
    while(e.hasMoreElements()) {
        String param = (String)e.nextElement();
        key = param.toUpperCase();
        val = request.getParameter(param).toUpperCase();
      }
      str = "<tr><td><b>"+key+"</b></td><td>"+val+"</td></tr>";
    }
    writer.println(str);
    writer.println("</table></body></html>");
}
```

(1) *Grava request em session*

(3) *Transforma nome e valor para caixa-alta*

(2) *Devolve tabela*

Testes com o Cactus

```
public class MeuServletTest extends ServletTestCase { (...)
  private MeuServlet servlet;
  public void beginDoGet(WebRequest cSideReq) {
     cSideReq.addParameter("user", "Jabberwock");
  }

  public void setUp() throws ServletException {
    servlet = new MeuServlet();
    servlet.init(this.config);
  }
  public void testDoGet() throws IOException {
    servlet.doGet(this.request, this.response);
    String value = (String) session.getAttribute("user");
    assertEquals("Jabberwock", value);
  }

  public void endDoGet(WebResponse cSideResponse) {
    String str = cSideResponse.getText();
    assertTrue(str.indexOf("USER</b></td><td>JABBERWOCK") > -1);
  }
}
```

Simula servlet container

Verifica se parâmetro foi mapeado à sessão

Verifica se parâmetro aparece na tabela HTML e em maiúsculas

Exemplo: funcionamento

Cliente (JUnit)

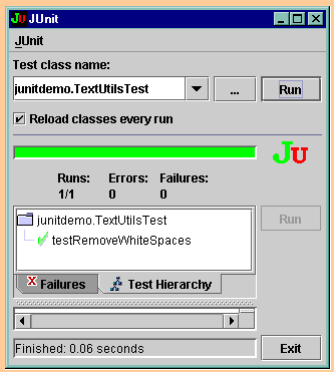
`beginDoGet(WebRequest req)`

- *Grava parâmetro:*

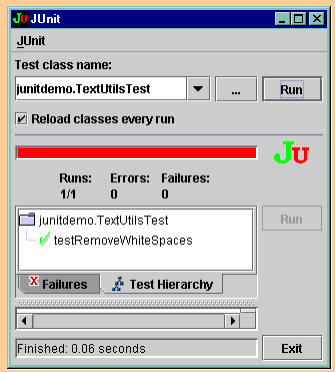
nome=`user`

value=`Jabberwock`

SUCCESS!!



FAIL!



falha local

falha remota

&

`endDoGet(WebResponse res)`

- Verifica se resposta contém

`USER</b></td><td>JABBERWOCK`

ReqInfo

2 conexões HTTP:

- *Uma p/ rodar os testes e obter saida do servlet*
- *Outra para esperar resultados de testes (info sobre exceções)*

TestInfo

Output

Servidor (Tomcat)

`setUp()`

- *Define init-params no objeto* `config`
- *Roda* `init(config)`

`testDoGet()`

- *Roda* `doGet()`
- *Verifica se parâmetro (no response) foi mapeado à sessão*

`tearDown()`

Teste de interface Web com HttpUnit

- *Onde encontrar*
  - `http://httpunit.sourceforge.net`
- *Para testes funcionais de interface (tipo "caixa-preta")*
  - *Verifica a resposta de uma aplicação Web ou página HTML*
  - *Oferece métodos para "navegar" na resposta: links, tabelas, imagens, objetos DOM (Node, Element, Attribute)*
- *Pode ser combinado com Cactus no endXXX()*
  - *Argumento* `com.meterware.httpunit.WebResponse` *substitui* `WebResponse` *do Cactus*
- *Testes HttpUnit podem ser complexos*
  - *São testes funcionais (não são unitários) e, portanto, envolvem várias classes, páginas, tecnologias, etc.*
  - *Geralmente há menos testes funcionais que unitários*

Principais classes da API do HttpUnit

- *WebConversation*
  - *Representa uma sessão de cliente Web (usa cookies)*

```
WebConversation wc = new WebConversation();
WebResponse resp = wc.getResponse("http://xyz.com/t.html");
```

- *WebRequest*
  - *Representa uma requisição*
- *WebResponse*
  - *Representa uma resposta. A partir deste objeto pode-se obter objetos WebLink, WebTable e WebForm*
- *WebLink*
  - *Possui métodos para extrair dados de links de hipertexto*
- *WebTable*
  - *Possui métodos para navegar na estrutura de tabelas*
- *WebForm*
  - *Possui métodos para analisar a estrutura de formulários*

HttpUnit com Cactus

- *Troque o* **WebResponse** *em cada* endXXX() *por* **com.meterware.httpunit.WebResponse**

```
public void endDoGet(com.meterware.httpunit.WebResponse resp)
                                        throws org.xml.sax.SAXException {
    WebTable[] tables = resp.getTables();
    assertNotNull(tables);
    assertEquals(tables.length, 1);  // só há uma tabela
    WebTable table = tables[0];
    int rows = table.getRowCount();
    boolean keyDefined = false;
    for (int i = 0; i < rows; i++) {
        String key   = table.getCellAsText(i, 0); // col 1
        String value = table.getCellAsText(i, 1); // col 2
        if (key.equals("USER")) {
            keyDefined = true;
            assertEquals("JABBERWOCK", value);
        }
    }
    if (!keyDefined) {
        fail("No key named USER was found!");
    }
}
```

Resumo e conclusões

- *JUnit é uma ferramenta open-source que ajuda a implementar testes em projetos Java*
  - *Com JUnit, é fácil viabilizar práticas como Refactoring, Integração Contínua e Test-Driven Development (TDD)*
- *Em um processo TDD, testes que falham são escritos no JUnit para que, a partir deles, se possa implementar código que os faça rodar com sucesso*
- *Mock objects podem ser usados para representar dependência e diminuir as responsabilidades de testes*
- *Cactus é uma extensão do JUnit para testar a integração de aplicações J2EE com seu container*

Use JUnit para desenvolver código de maior qualidade, maior confiabilidade, aumentar sua produtividade, reduzir o stress e aposentar o seu debugger!

Fontes (1)

[1] *R. Hightower, N. Lesiecki. Java Tools for eXtreme Programming. Wiley, 2002.* Explora ferramentas Ant, JUnit, Cactus e outras usando estudo de caso com processo XP.

[2] *Jeffries, Anderson, Hendrickson. eXtreme Programming Installed, Addison-Wesley, 2001.* Contém exemplos de estratégias para testes.

[3] *Kent Beck, Erich Gamma. JUnit Test Infected: programmers love writing tests. (JUnit docs).* Aprenda a usar JUnit em uma hora.

[4] *Andy Schneider. JUnit Best Practices. JavaWorld, Dec. 2000.* Dicas do que fazer ou não fazer para construir bons testes.

[5] *Robert Koss, Testing Things that are Hard to Test. Object Mentor, 2002* http://www.objectmentor.com/resources/articles/TestingThingsThatAreHa~9740.ppt. Mostra estratégias para testar GUIs e código com dependências usando stubs.

[6] *Mackinnon, Freeman, Craig. Endo-testing with Mock Objects.* http://mockobjects.sourceforge.net/misc/mockobjects.pdf. O autor apresenta técnicas para testes usando uma variação da técnica de stubs chamada de "mock objects".

Fontes (2)

[7] William Wake. *Test/Code Cycle in XP. Part I: Model, Part II: GUI.* http://users.vnet.net/wwake/xp/xp0001/. Ótimo tutorial em duas partes sobre a metodologia "test-first" mostrando estratégias para testar GUIs na segunda parte.

[8] Steve Freeman, *Developing JDBC Applications Test First. 2001.* Tutorial sobre metodolodia test-first com mock objects para JDBC.

[9] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley, 2000. Cap 4 (building tests) é um tutorial usando JUnit.

[10] Kent Beck. *Test-driven development.* Addison-Wesley, 2002. Mostra como utilizar testes como "requerimentos executáveis" para guiar o desenvolvimento.

[11] *Apache Cactus User's Manual.* Contém tutorial para instalação passo-a-passo.

helder@argonavis.com.br

Selecione o link relativo a esta palestra no endereço

www.argonavis.com.br

Recursos disponíveis no site:

- *Palestra completa em PDF*
- *Código-fonte usado nos exemplos e demonstrações*
- *Instruções sobre como rodar e instalar os exemplos*
- *Links para software utilizado e documentação*

© 2001-2003, Helder da Rocha