Testes de Unidade em Java com JUnit

Helder da Rocha (helder@acm.org)

Objetivos

1. Apresentar o framework **JUnit** e demonstrar como utilizá-lo para testar aplicações em Java
2. Discutir **dificuldades** relativas à prática de testar, como descobrir testes, dependências, GUIs, etc.
3. Apresentar **extensões do JUnit** para lidar com dificuldades e cenários específicos de teste

Público-alvo:
Programadores Java

Por que testar?

- Por que não?
  - Como saber se o recurso funciona sem testar?
  - Como saber se ainda funciona após refatoramento?
- 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!

O que é JUnit?

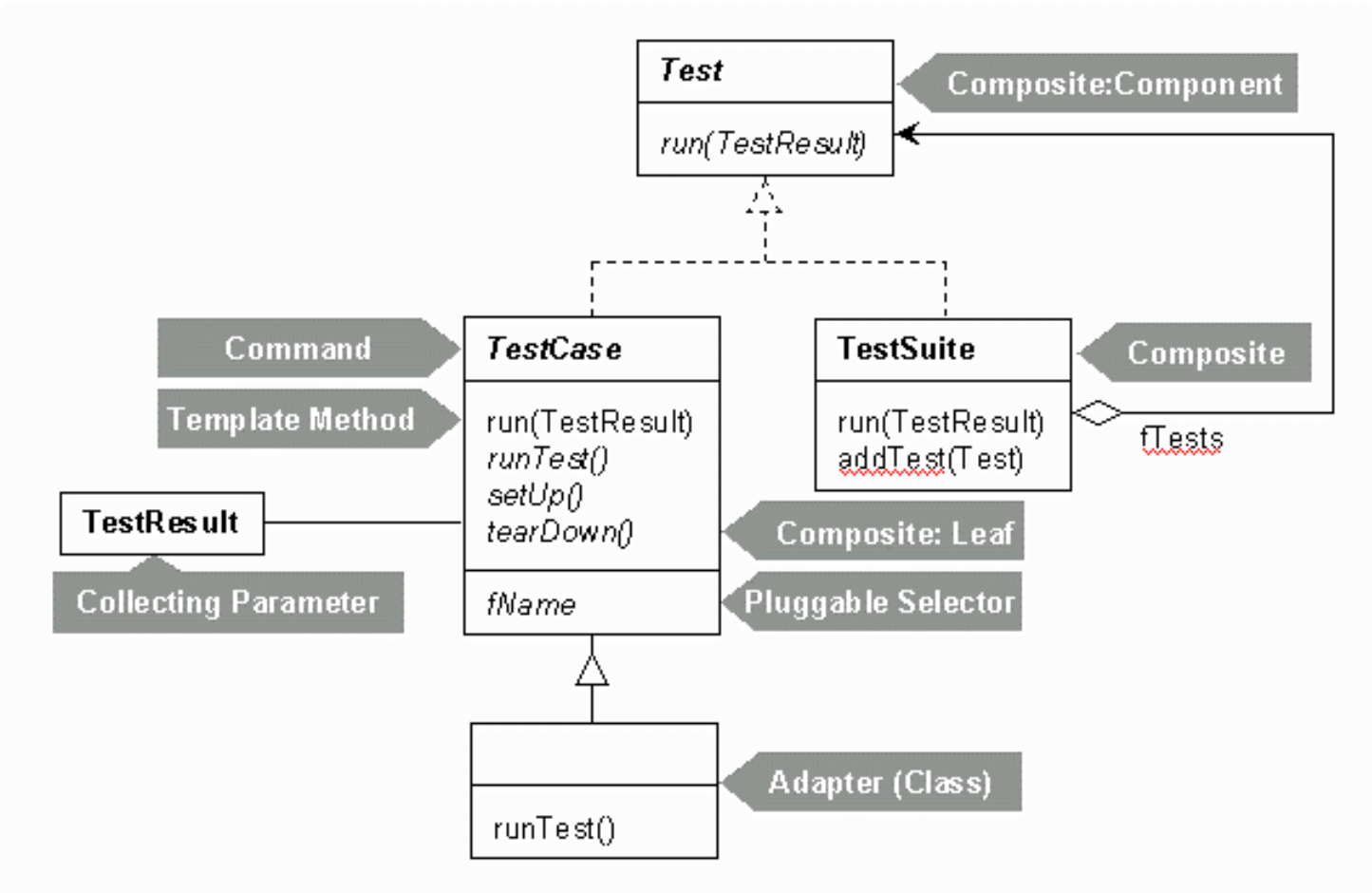
- Um **framework** que facilita o desenvolvimento e execução de **testes de unidade** em código Java
  - Uma API para construir os testes
  - Aplicações para executar testes
- A API
  - Classes Test, TestCase, TestSuite, etc. oferecem a infraestrutura necessária para criar os testes
  - Métodos assertTrue(), assertEquals(), fail(), etc. são usados para testar os resultados
- Aplicação TestRunner
  - Roda testes individuais e suites de testes
  - Versões texto, Swing e AWT
  - Apresenta diagnóstico sucesso/falha e detalhes

Para que serve?

- 'Padrão' para testes de unidade em Java
  - Desenvolvido por Kent Beck e Erich Gamma
  - Design muito simples
- Testar é uma boa prática, mas é chato; JUnit torna as coisas mais agradáveis, facilitando
  - A criação e execução automática de testes
  - A apresentação dos resultados
- JUnit pode verificar se cada unidade de código funciona da forma esperada
  - Permite agrupar e rodar vários testes ao mesmo tempo
  - Na falha, mostra a causa em cada teste
- Serve de base para extensões

Arquitetura do JUnit

- *Diagrama de classes*



Fonte: Manual do JUnit (Cooks Tour)

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?
  - www.junit.org
- Como instalar?
  - Incluir o arquivo junit.jar no classpath para compilar e rodar os programas de teste

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() {...}`
- *Sobreponha o método setUp(), se necessário*
- *Sobreponha o método tearDown(), se necessário*

JUnit para guiar o desenvolvimento

Cenário de Test-Driven Development (TDD)

1. *Defina uma lista de tarefas a implementar*
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*

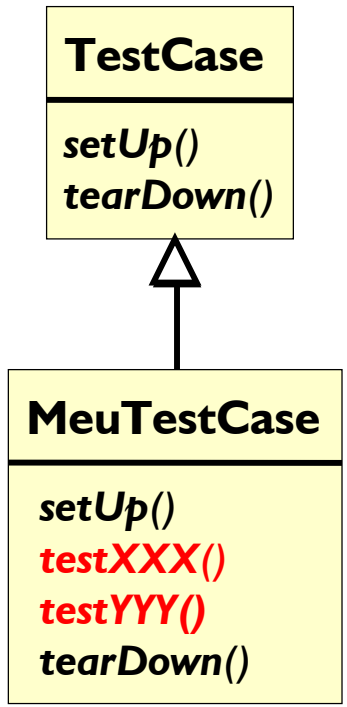
Como implementar?

- *Dentro de cada teste, utilize os métodos herdados da classe TestCase*
  - *assertEquals(objetoEsperado, objetoRecebido),*
  - *assertTrue(valorBooleano), assertNotNull(objeto)*
  - *assertSame(objetoUm, objetoDois), fail (), ...*
- *Exemplo de test case com um setUp() e um teste:*

```
public class CoisaTest extends TestCase {
  // construtor padrão omitido
  private Coisa coisa;
  public void setUp() { coisa = new Coisa("Bit"); }
  public void testToString() {
    assertEquals("<coisa>Bit</coisa>",
                 coisa.toString());
  }
}
```

Como funciona?

- O TestRunner recebe uma subclasse de junit.framework.TestCase
  - Usa reflection para descobrir seus métodos
- Para cada método testXXX(), 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 provocar **exceção**



Exemplo: uma classe

```
package junitdemo;
import java.io.*;

public class TextUtils {

  public static String removeWhiteSpaces(String text)
                                                throws IOException {
    StringReader reader = new StringReader(text);
    StringBuffer buffer = new StringBuffer(text.length());
    int c;
    while( (c = reader.read()) != -1) {
      if (c ==' '||c =='\n'||c =='\r'|| c =='\f'||c =='\t') {
         ; /* do nothing */
      } else {
         buffer.append((char)c);
      }
    }
    return buffer.toString();
  }
}
```

veja código em

junitdemo.zip

Exemplo: um test case para a classe

```
package junitdemo;

import junit.framework.*;
import java.io.IOException;

public class TextUtilsTest extends TestCase {

    public TextUtilsTest(String name) {
        super(name);
    }

    public void testRemoveWhiteSpaces() throws IOException {
        String testString = "one, ( two | three+) ,       "+
                            "(((four+ |\t  five)?\n \n,  six?";
        String expectedString = "one,(two|three+)"+
                                ",(((four+|five)?,six?";
        String results = TextUtils.removeWhiteSpaces(testString);
        assertEquals(expectedString, results);
    }
}
```

Construtor precisa ser publico, receber String name e chamar `super(String name)`

*Método começa com "**test**" e é sempre **public void***

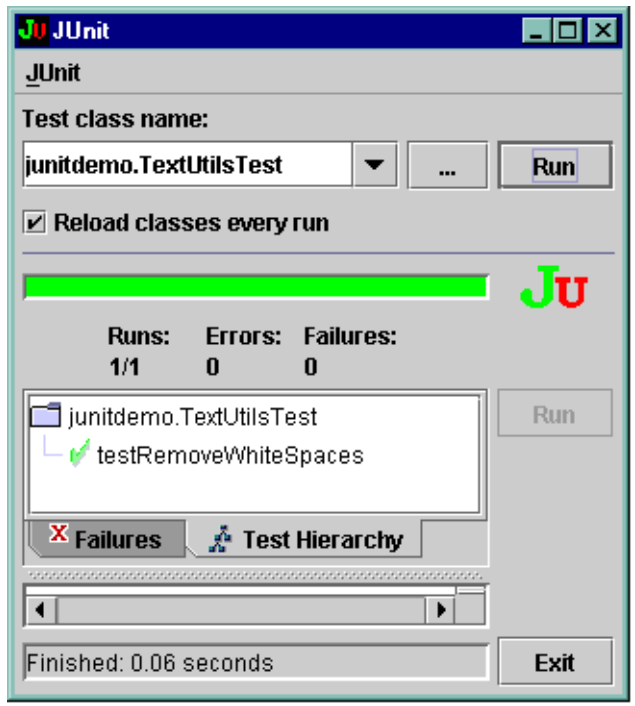
Exemplo: como executar

- Use a interface de texto
  - java -cp junit.jar junit.textui.TestRunner junitdemo.TextUtilsTest
- Ou use a interface gráfica
  - java -cp junit.jar junit.swingui.TestRunner junitdemo.TextUtilsTest
- Use Ant <junit>
  - tarefa do Apache Ant
- Ou forneça um main():

```
public static void main (String[] args) {
    TestSuite suite =
      new TestSuite(TextUtilsTest.class);
    junit.textui.TestRunner.run(suite);
}
```

veja demonstração em

junitdemo.zip



TestSuite

- Permite executar uma coleção de testes
  - Método addTest(TestSuite) adiciona um teste na lista
- Padrão de codificação (usando reflection):
  - retornar um TestSuite em cada test-case:

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

  - criar uma classe AllTests que combina as suites:

```
public class AllTests {
    public static Test suite() {
        TestSuite testSuite =
                        new TestSuite("Roda tudo");
        testSuite.addTest(pacote.AllTests.suite());
        testSuite.addTest(MinhaClasseTest.suite());
        testSuite.addTest(SuaClasseTest.suite());
        return testSuite;
    }
}
```

Pode incluir outras suites

veja demonstração em junitdemo.zip

Fixtures

- São os dados reutilizados por vários testes
  - Inicializados no setUp() e destruídos no tearDown() (se 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);
    }
(...)
```

Fixture

- Extensão **JXUnit** (jxunit.sourceforge.net) permite manter dados de teste em arquivo XML (*.jxu) separado do código
  - Mais flexibilidade. Permite escrever testes mais rigorosos, com muitos dados

veja exemplo

jxunitdemo.zip

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
    }
}
```

Afirmações do J2SDK1.4 (assertions)

- São expressões booleanas que o programador define para afirmar uma condição que ele acredita ser verdade
  - Afirmações são usadas para validar código (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
- Afirmações são um recurso novo do JSDK1.4.0
  - Nova palavra-chave: **`assert`**
  - É preciso compilar usando a opção -source 1.4:

```
> javac -source 1.4 Classe.java
```

  - Para executar, é preciso habilitar afirmações (enable assertions):

```
> java -ea Classe
```

JUnit vs. afirmações

- Afirmações 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)
- Afirmações do JUnit são usadas em classe separada (TestCase)
  - Não têm acesso ao interior dos métodos (verificam se a interface dos métodos funciona como esperado)
- Afirmações do J2SDK1.4 e JUnit são complementares
  - JUnit testa a interface dos métodos
  - assert testa trechos de lógica dentro dos métodos

veja exemplo

junitdemo.zip

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)
- Soluções com refatoramento
  - Isolar em métodos private apenas código inquebrável
  - Transformar métodos private em package-private
    - Desvantagem: quebra ou redução do encapsulamento
    - Classes de teste devem estar no mesmo pacote que as classes testadas para ter acesso
- Solução usando extensão do JUnit
  - **JUnitX**: usa reflection para ter acesso a dados private
  - http://www.extreme-java.de/junitx/index.html

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 seu código para que fique mais fácil de testar

Como descobrir testes?

- Listas de tarefas (to-do list)
  - Comece pelas mais simples e deixe os testes "realistas" para o final
  - Requerimentos, use-cases, diagramas UML: rescreva os requerimentos em termos de testes
- Dados
  - Use apenas dados suficientes (não teste 10 condições se três forem suficientes)
- Bugs revelam testes
  - Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)!

Test-Driven Development (TDD)

- Desenvolvimento guiado pelos testes
  - Só escreva código novo se um teste falhar
  - Refatore 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 simplesmente fazendo método retornar constante
  - 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

Como lidar com testes difíceis

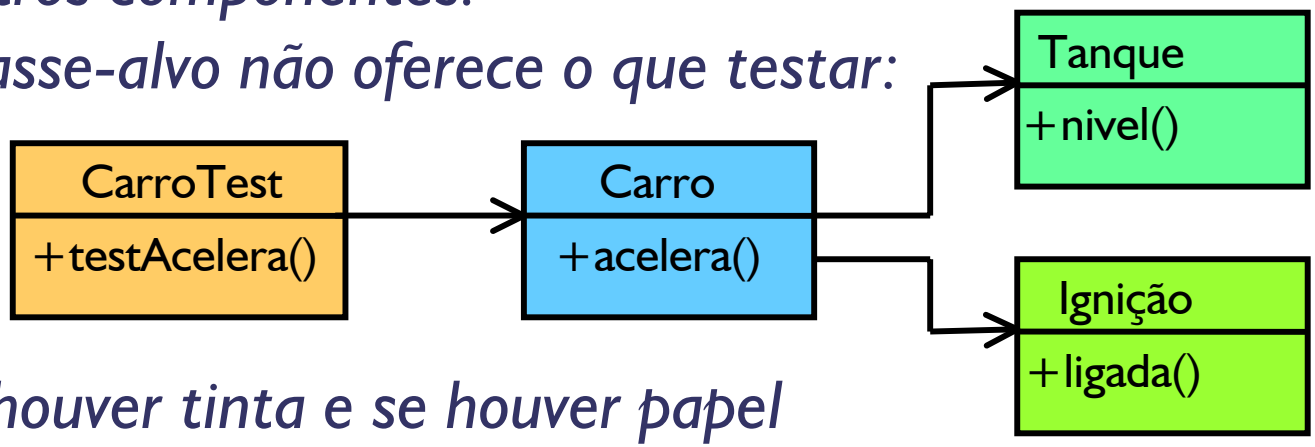
- Testes devem ser simples e suficientes
  - **XP**: design mais simples que resolva o problema; sempre pode-se escrever novos testes, quando necessário
- Não complique
  - Não teste o que é responsabilidade de outra classe/método
  - Assuma que outras classes e métodos funcionam
- Testes difíceis (ou 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?* [11]
  - *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:*



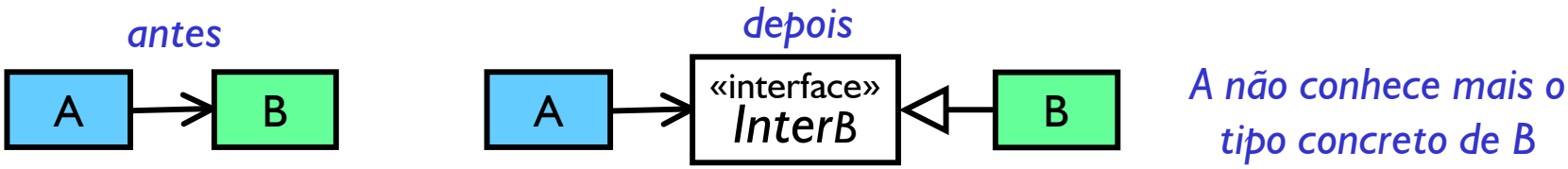
  - *Se houver tinta e se houver papel método* **`void imprime()`** *deve funcionar*
  - *Como saber se há ou não tinta e papel?*

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

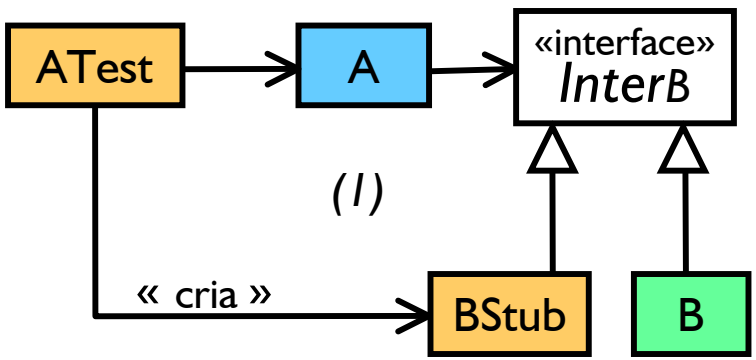
Fonte: www.objectmentor.com, 2002

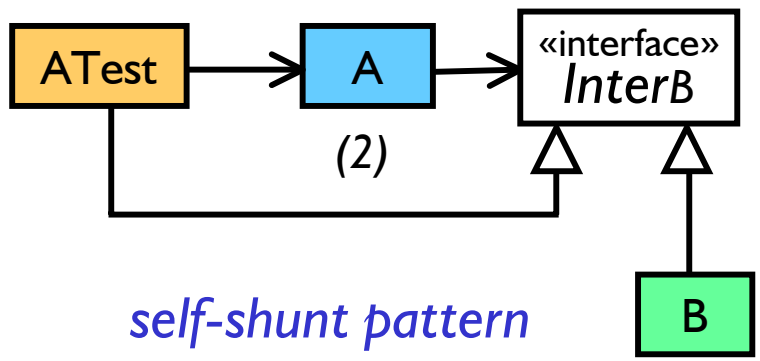
Stubs: objetos "impostores"

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



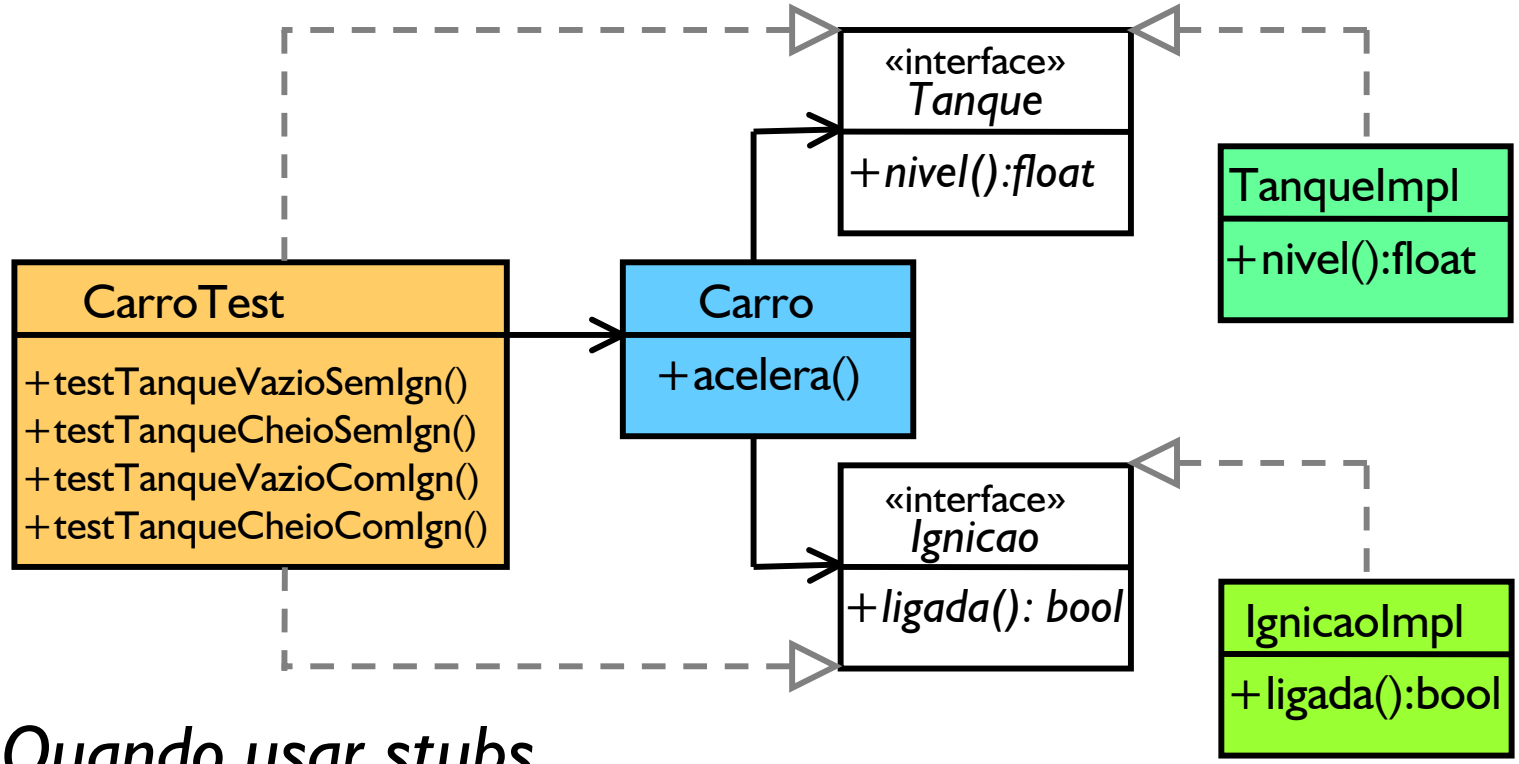
- *Agora B pode ser substituída por um stub*
  - *BStub está sob controle total de ATest (1)*
  - *Em alguns casos, ATest pode implementar InterB (2)*





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 (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

veja exemplo mockdemo.zip

Testes de interface

- Caso específico: resposta de servidores Web
  - Verificar se uma página HTML ou XML contém determinado texto ou determinado elemento
  - Verificar se resposta está de acordo com dados passados na requisição: testes funcionais tipo "caixa-preta"
- Soluções (extensões do JUnit)
  - **HttpUnit** e **ServletUnit**:
    - permite testar dados de árvore DOM HTML gerada
  - **JXWeb** (combinação do JXUnit com HttpUnit)
    - permite especificar os dados de teste em arquivos XML
    - arquivos de teste Java são gerados a partir do XML
  - **XMLUnit**
    - extensão simples para testar árvores XML
  - Onde encontrar: (httpunit|jxunit|xmlunit).sourceforge.net

Testes de performance

- **JUnitPerf** (www.clarkware.com)
  - Coleção de decoradores para medir performance e escalabilidade em testes JUnit existentes
- TimedTest
  - Executa um teste e mede o tempo transcorrido
  - Define um tempo máximo para a execução. Teste falha se execução durar mais que o tempo estabelecido
- LoadTest
  - Executa um teste com uma carga simulada
  - Utiliza timers para distribuir as cargas usando distribuições randômicas
  - Combinado com TimerTest para medir tempo com carga
- ThreadedTest
  - Executa o teste em um thread separado

JUnitPerf: exemplo

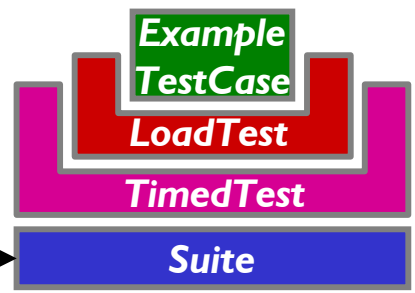
```
import com.clarkware.junitperf.*;
import junit.framework.*;

public class ExampleLoadTest extends TestCase {
  public ExampleLoadTest(String name) { super(name); }
  public static Test suite() {
     TestSuite suite = new TestSuite();
     Timer timer = new ConstantTimer(1000);
     int maxUsr = 10;
     int iters = 10;
     long maxElapsedTime = 20000;
     Test test = new ExampleTestCase("testOneSecondResp");
     Test loadTest = new LoadTest(test, maxUsr, iters, timer);
     Test timedTest = new TimedTest(loadTest, maxElapsedTime);
     suite.addTest(timedTest);
     return suite;
  }
  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

TestCase existente

Decorador de carga

Decorador de tempo



TestRunner

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>
```

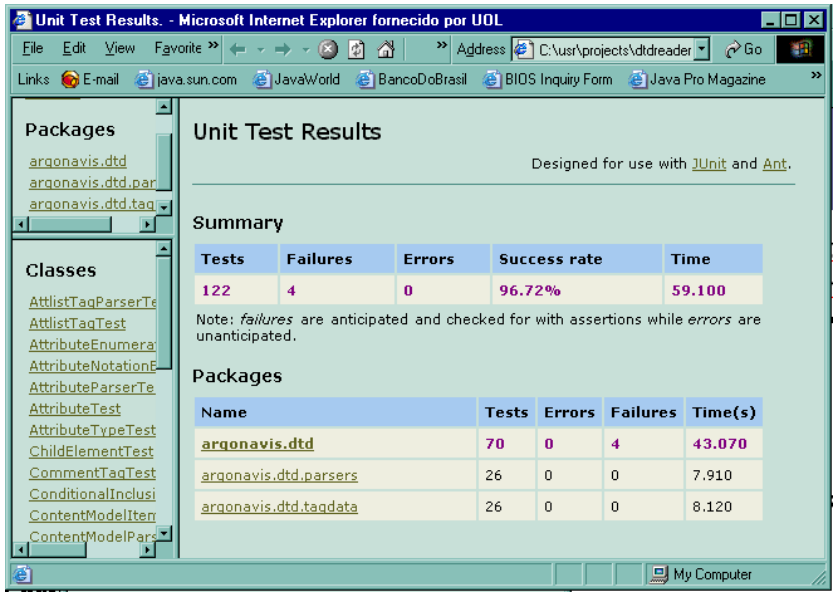
Usa arquivos XML gerados por <formatter>

veja demonstração

junitdemo.zip

veja demonstração

dtdreader.zip



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*
- *Para testes em domínios específicos e para lidar com limitações do JUnit há várias extensões open-source*
  - *HttpUnit, Cactus, JUnitPerf, JXUnit, JUnitX, JUnitEE ...*

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.*

Fontes (2)

[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".

[7] William Wake. *Test/Code Cycle in XP. Part 1: 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.

helder@argonavis.com.br

Selecione o link relativo a esta palestra no endereço

www.argonavis.com.br/xpbrasil2002

Recursos disponíveis no site:

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

Tutorial: Implementando eXtreme Programming em Java
XP Brasil 2002, São Paulo