



Senac

Teste de Software

Aula 01

---

JUnit

- *Este módulo é mais sobre boas práticas de desenvolvimento e menos sobre Java*
- *Abordagem será superficial* (material é extenso), mas visa despertar seu interesse no hábito de escrever testes.
- **Objetivos**
- *Apresentar e incentivar a prática de testes de unidade durante o desenvolvimento*
- *Apresentar a ferramenta JUnit, que ajuda na criação e execução de testes de unidade em Java*
- *Discutir as dificuldades relativas à "arte" de testar e como podem ser superadas ou reduzidas*
- *Torná-lo(a) uma pessoa "viciada" em testes: Convencê-lo(a) a nunca escrever uma linha sequer de código sem antes escrever um teste executável que a justifique.*

# O que é "Testar código"?



- *É a coisa 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?*
  - *São 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 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?



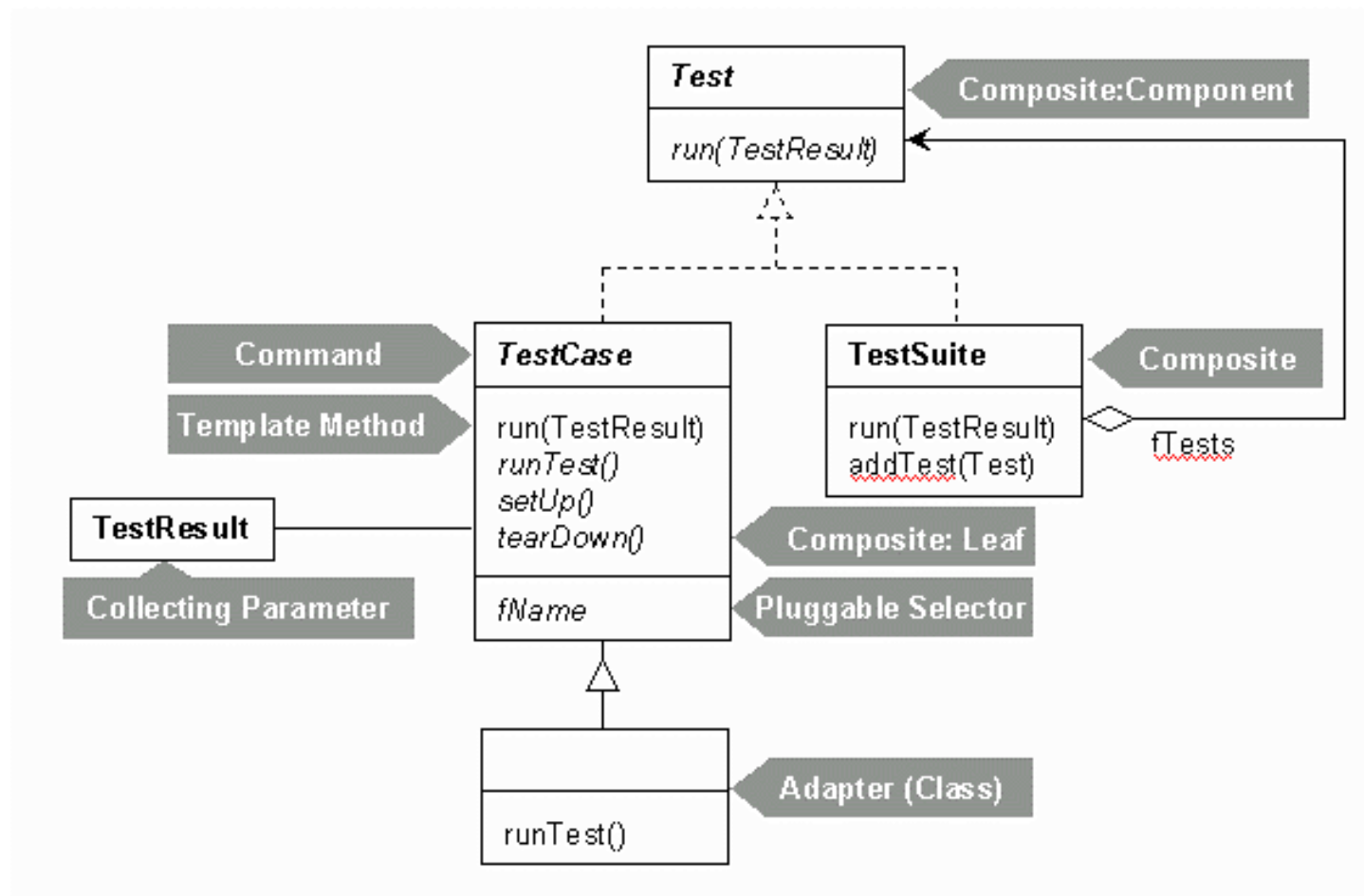
- Um **framework** que facilita o desenvolvimento e execução de **testes de unidade** em código Java
  - Uma **API** para **construir** os testes
  - **Aplicações** para **executar** testes
- **A API**
  - Classes **Test**, **TestCase**, **TestSuite**, etc. oferecem a infraestrutura necessária para criar 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 (XP) e Erich Gamma (GoF)
  - 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

- *Diagrama de classes*



# 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](http://www.junit.org)
- *Como instalar?*
  - *Incluir o arquivo junit.jar no classpath para compilar e rodar os programas de teste*
- *Para este curso*
  - *Inclua o junit.jar no diretório lib/ de seus projetos*



# 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?



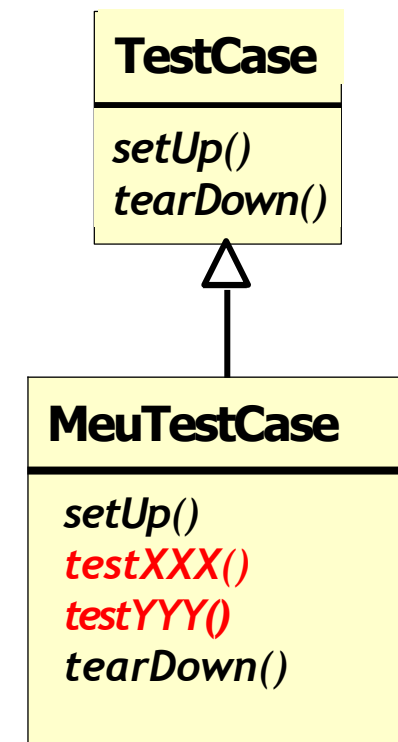
- *Dentro de cada teste, utilize os métodos herdados da classe TestCase*
  - *assertEquals(objetoEsperado, objetoRecebido),*
  - *assertTrue(valorBooleano), assertNotNull(objeto)*
  - *assertSame(objetoUm, objetoDois), fail (), ...*
- *Exemplo de test case com um setUp() e um teste:*

```
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 (Cap 14) para achar 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: um test case



```
package junitdemo;

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

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

```
public class TextUtilsTest extends TestCase {
```

*(JUnit 3.7 ou anterior)*

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

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

```
    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);
    }
}
```

# Exemplo: uma classe que faz o teste passar



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

public class TextUtils {

    public static String removeWhiteSpaces(String text)
                                                throws IOException {
        return "one, (two|three+), (((four+|five)?, six?";
    }
}
```

- *O teste passa... e daí? A solução está pronta? Não! Tem dados duplicados! Remova-os!*
- *Escreva um novo teste que faça com que esta solução falhe, por exemplo:*

```
String test2 = " a    b\nc    ";
assertEquals("abc", TextUtils.removeWhiteSpaces(test2));
```

# Outra classe que faz o teste passar



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

public class TextUtils {

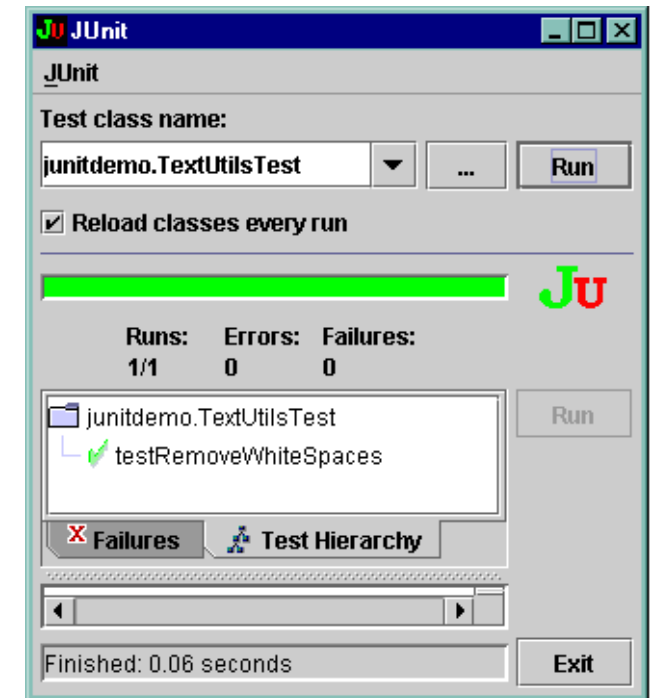
    public static String removeWhiteSpaces(String text)
                                                throws IOException {
        StringReader reader = new StringReader(text);
        StringBuffer buffer = new StringBuffer(text.length());
        int c;
        while( (c = reader.read()) != -1) {
   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

# 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);  
}
```





# 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() {
```

*Pode incluir  
outras suites*

```
        TestSuite testSuite =  
            new TestSuite("Roda tudo");  
        testSuite.addTest(pacote.AllTests.suite());  
        testSuite.addTest(MinhaClasseTest.suite());  
        testSuite.addTest(SuaClasseTest.suite());  
        return testSuite;
```

```
    }
```

```
}
```


# Fixtures



- *São os dados reutilizados por vários testes*

```
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



- *Se os mesmos dados são usados em vários testes, inicialize-os no setUp() e faça a faxina no tearDown() (se necessário)*
- *Não perca tempo pensando nisto antes. Escreva seus testes. Depois, se achar que há duplicação, monte o fixture.*

# Teste situações de falha



- *É tão importante testar o cenário de falha do seu código 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  
    }  
}
```

# JUnit vs. asserções



- *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*

# 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 (open-source)*
  - **JUnitX**: usa reflection para ter acesso a dados **private**
  - <http://www.extreme-java.de/junitx/index.html>

# Resumo: JUnit



- *Para o JUnit,*
  - *Um teste é um método*
  - *Um caso de teste é uma classe contendo uma coleção de testes (métodos que possuem assertions)*
  - *Cada teste testa o comportamento de uma unidade de código do objeto testado (pode ser um método, mas pode haver vários testes para o mesmo método ou um teste para todo o objeto)*
- *Fixtures são os dados usados em testes*
- *TestSuite é uma composição de casos de teste*
  - *Pode-se agrupar vários casos de teste em uma suite*
- *JUnit testa apenas a interface das classes*
  - *Mantenha os casos de teste no mesmo diretório que as classes testadas para ter acesso a métodos package-private*
  - *Use padrões de nomenclatura: ClasseTest, AllTests*
  - *Use o Ant para separar as classes em um release*

## **Apêndice: boas práticas e dificuldades com testes**

# 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?



- *Escreva 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)!*
- *Teste sempre! Não escreva uma linha de código sem antes escrever um teste!*

# 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



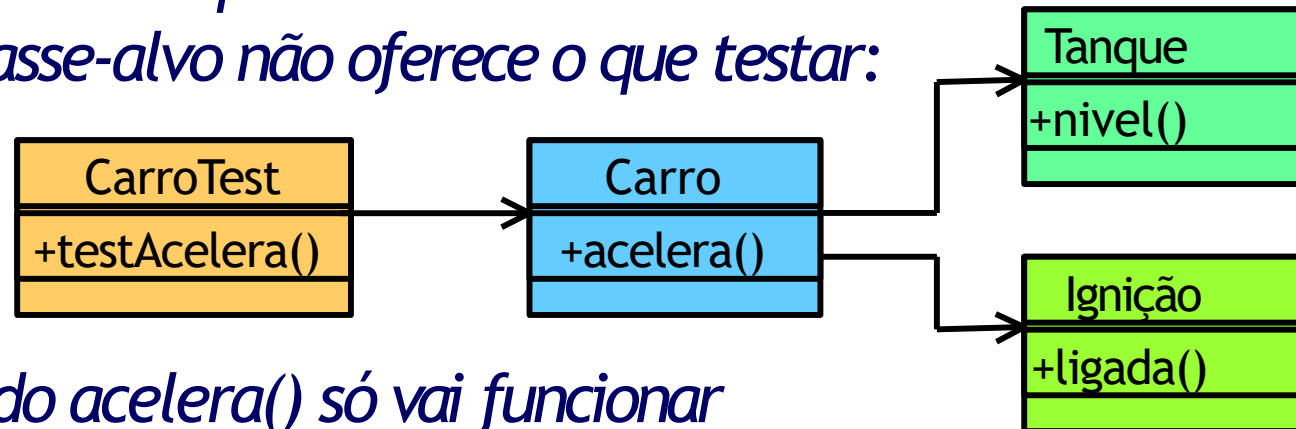
- *Testes devem ser simples e suficientes*
  - *XP: design mais simples que resolva o problema; sempre pode-se escrever novos testes, quando necessário*
- *Não complique*
  - *Não teste o que é **responsabilidade** de outra classe/método*
  - ***Assuma** que outras classes e métodos funcionam*
- *Testes difíceis (ou que parecem difíceis)*
  - *Aplicações gráficas: eventos, layouts, threads*
  - *Objetos inacessíveis, métodos privados, 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*

# Dependência de código-fonte



- *Problema*

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



- *Método acelera() só vai funcionar se nível() do tanque for > 0 e ignição estiver ligada()*
- *Como saber se condições são verdadeiras se não temos acesso às dependências?*

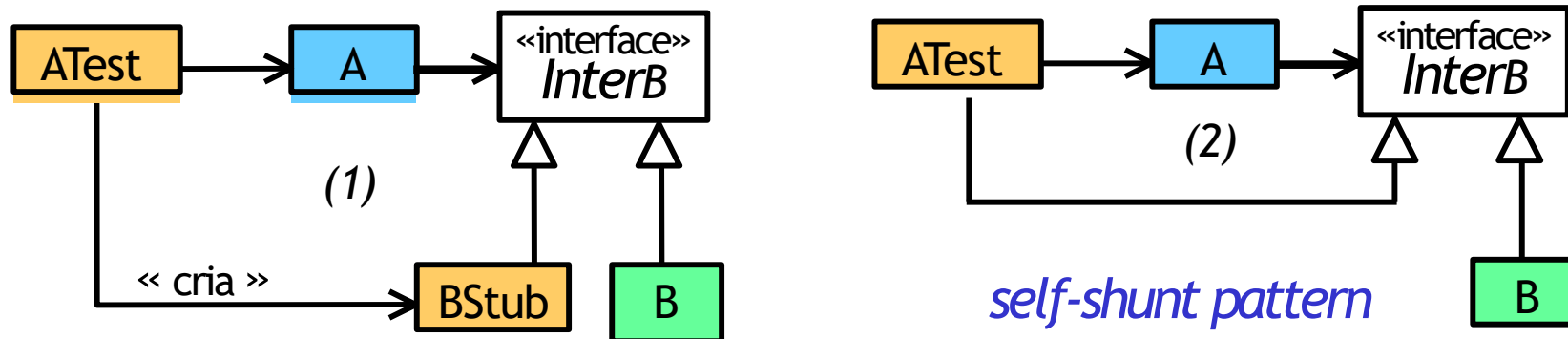
```
public void testAcelera() {
    Carro carro =
        new Carro();
    carro.acelera();
    assert??(???);
}
```

# Stubs: objetos "impostores"

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



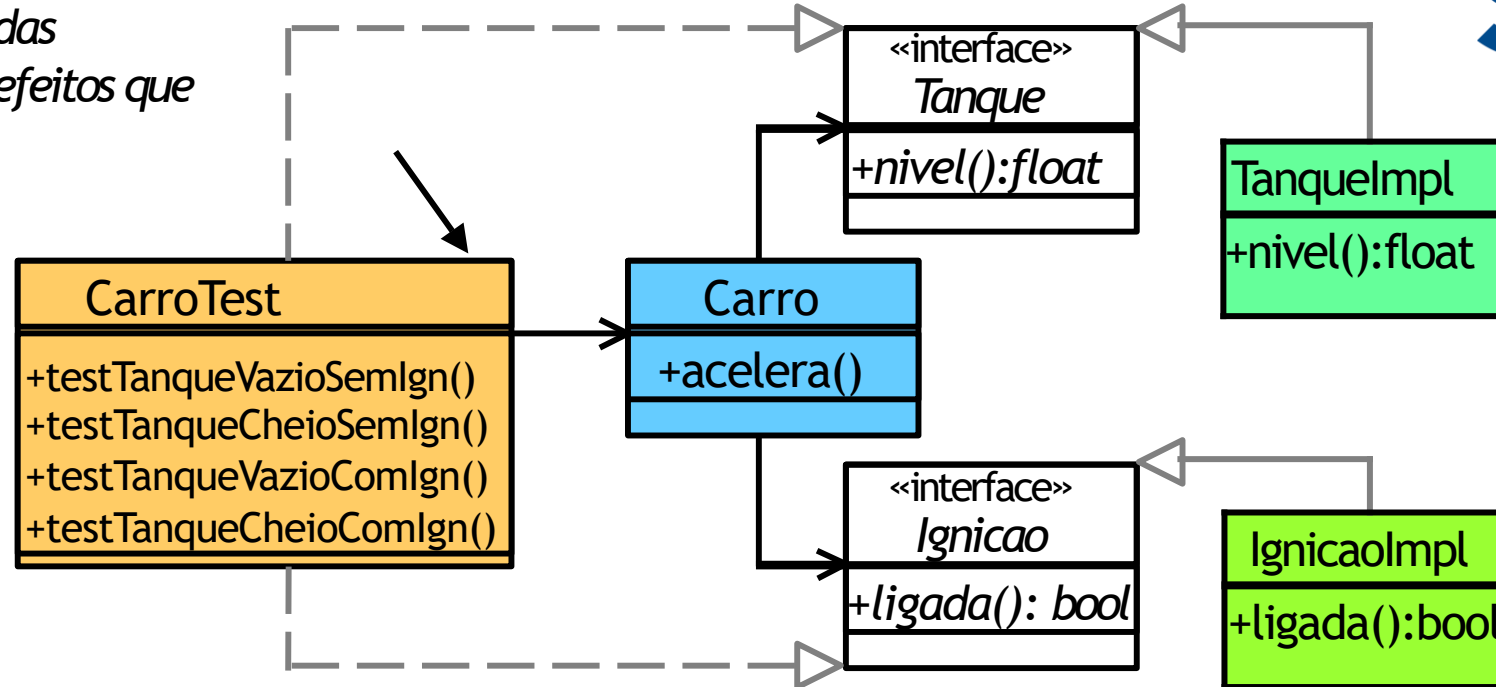
- *Agora B pode ser substituída por um stub*
  - *BStub está sob controle total de ATest (1)*
  - *Em alguns casos, ATest pode implementar InterB (2)*



# Dependência: solução usando stubs



*"Mock implementation" das dependências simula os efeitos que terão sobre Carro*



- Quando criar o objeto, passe a implementação falsa
  - `carro.setTanque(new CarroTest());`
  - `carro.setIgnicao(new CarroTest());`
- Depois preencha-a com dados suficientes para que objeto possa ser testado

# 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)
  - Exemplo em J2EE: **Jakarta Cactus**

# Mock Objects



- **Mock objects** (MO) é uma estratégia de uso de stubs que **não implementa nenhuma lógica**
  - Um mock object não é exatamente 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/](http://tammofreese.de/easymock/)) e **MockMaker** ([www.xpdeveloper.com](http://www.xpdeveloper.com)) **geram MOs a partir de interfaces**
  - **Projeto MO** ([mockobjects.sourceforge.net](http://mockobjects.sourceforge.net)) **coleção de mock objects e utilitários para usá-los**



# Ant + JUnit



- *Com Ant, 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 em \$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>
```

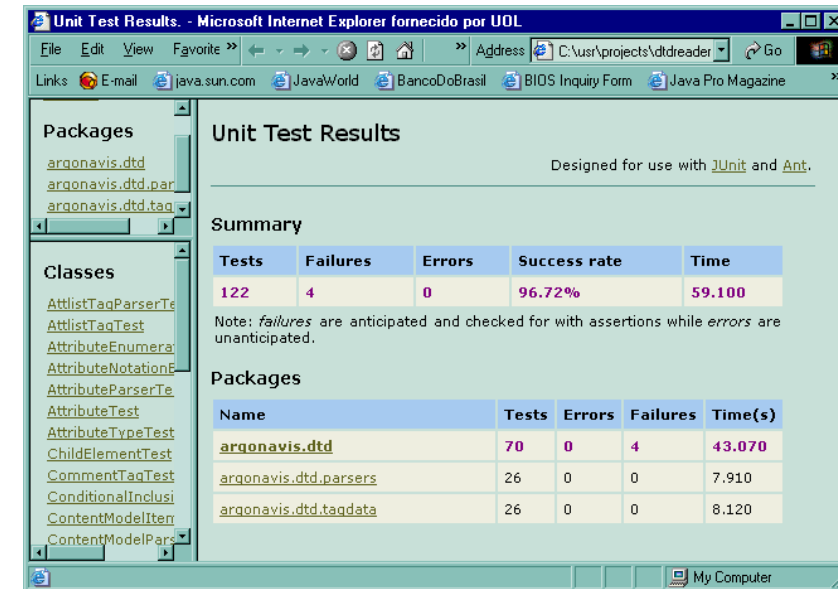
Gera arquivo XML  
Inclui todos os arquivos que  
terminam em TEST.java

# <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>



The screenshot shows a web browser window titled 'Unit Test Results. - Microsoft Internet Explorer fornecido por UOL'. The address bar shows 'C:\usr\projects\vtddreader'. The page content includes a summary table and a packages table.

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.

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

# Resumo



- *Testar é tarefa essencial do desenvolvimento de software.*
- *Testar unidades de código durante o desenvolvimento é uma prática que traz inúmeros benefícios*
  - *Menos tempo de depuração (muito, muito menos!)*
  - *Melhor qualidade do software*
  - *Segurança para alterar o código*
  - *Usando TDD, melhores estimativas de prazo*
- *JUnit é uma ferramenta open-source que ajuda a implementar testes em projetos Java*
- *TDD ou Test-Driven Development é uma técnica onde os testes são usados para guiar o desenvolvimento*
  - *Ajuda a focar o desenvolvimento em seus objetivos*
- *Mock objects ou stubs podem ser usados para representar dependência e diminuir as responsabilidades de testes*

# Exercício



- 1. A classe *Exercicio.java* possui quatro métodos vazios:

```
int soma(int a, int b) long fatorial(long  
n) ;  
double fahrToCelsius(double fahrenheit) ;  
String inverte(String texto) ;
```

Escreva, na classe *ExercicioTest.java*, test-cases para cada método, que preencham os requisitos:

- **Soma:**  $1 + 1 = 2$ ,  $2 + 4 = 4$
- **Fatorial:**  $0! = 1$ ,  $1! = 1$ ,  $2! = 2$ ,  $3! = 6$ ,  $4! = 24$ ,  $5! = 120$   
A fórmula é  $n! = n(n-1)(n-2)(n-3)...3*2*1$
- **Celsius:**  $-40C = -40F$ ,  $0C = 32F$ ,  $100C = 212F$   
A fórmula é  $F = \frac{9}{5} * C + 32$
- **Inverte** recebe "Uma frase" e retorna "esarf amU"
- Implemente um teste e execute-o (o esqueleto de um deles já está pronto). Ele deve falhar.
- Implemente os métodos, um de cada vez, e rode os testes até que não falhem mais (tarja verde), antes de prosseguir.