Testes de Unidade

prof. Angelo Loula

Baseado em http://junit.sourceforge.net/doc/testinfected/testing.htm e http://www.dsc.ufcg.edu.br/~jacques/cursos/p2/html/oo/testes.htm

Realizando Testes

- Você faz testes no seu programa?
- Todo programador sabe que deve testar seus programas, mas pouco realmente o fazem de forma completa.
- Assuma que: um código não testado, é um código que não funciona.
  - Isso é melhor que: se o desenvolvedor disse que funciona, então funciona agora e sempre

Realizando Testes

- Como você faz testes no seu programa?
- **`System.out.println()`** ?
  - Muitas saídas no console
  - Inspeção visual da saída
  - Insere código de teste no meio do código de execução
  - Manutenção trabalhosa

Realizando Testes

- Como você faz testes no seu programa?
- Usando Debug?
  - Executa o programa passo-a-passo, inspecionando o valor das variáveis e verificando se correspondem ao esperado
  - Uma checagem manual entre valores reais e valores esperados
  - Processo lento e tedioso!

Realizando Testes

- Porque não automatizar seus testes?
  - Você identifica os valores esperados e ocorre uma checagem automática se correspondem aos valores reais
  - Testes Automatizados!
- Mas o que devemos testar?
  - As classes?
  - Os métodos?
  - O sistema?

Realizando Testes

- Testes de Unidade
  - Unidade é a menor parte testável de um sistema
  - A unidade deve seguir um ‘contrato’ sobre o que ela deve saber, e esse contrato precisa ser testado
  - Em OO, a unidade é a classe.

Realizando Testes

- Testes de Unidade
  - Benefícios:
    - Melhor manutenção, integração, documentação, projeto (teste antes do código)
  - Testar as classes da menos acoplada para a mais acoplada
  - Em geral, leva menos tempo codificar testes automatizados do que realizar repetidos testes manuais.
  - Se for difícil escrever um teste para verificar valores esperados de execução, pode ser um indicativo que os métodos precisam ser mais coesos e curtos, o que melhora seu projeto de software.

JUnit

- JUnit é um framework para testes automatizados em Java
- JUnit segue a arquitetura xUnit para testes de unidade
- Inclui:
  - Assertions para testar valores esperados
  - Fixtures para compartilhar dados de teste em comum
  - Runners para executar testes

Testando uma classe

```
class Money {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }

    public Money add(Money m) {
        return new Money(amount()+m.amount(), currency());
    }
}
```

JUnit

- Testando nossa classe
  - Crie uma subclasse de TestCase
  - Coloque no mesmo pacote para ter acesso a métodos *protected*

```
public class MoneyTest extends TestCase {
    //…
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);
        Assert.assertTrue(expected.equals(result));
    }
}
```

JUnit

- MoneyTest envolve três passos:
  - Criar objetos que serão manipulados (contexto, *fixture*)
  - Manipular os objetos
  - Verificar o resultado com o esperado

- Mas como funciona o *equals* aqui?

```
Assert.assertTrue(expected.equals(result));
```

JUnit

- Precisamos sobrescrever o método equals de objeto para nossa classe, mas antes disso vamos escrever um teste para isso:

```
public void testEquals() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");

    Assert.assertTrue(!m12CHF.equals(null));
    Assert.assertEquals(m12CHF, m12CHF);
    Assert.assertEquals(m12CHF, new Money(12, "CHF"));
    Assert.assertTrue(!m12CHF.equals(m14CHF));
}
```

JUnit

- Depois podemos escrever o método para Money:

```
public boolean equals(Object anObject) {
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

JUnit

- Temos dois testes prontos, mas eles duplicam algum código.
- Podemos reutilizar código de teste e estabelecer fixtures comuns.
  - método setUp()
  - método tearDown()
  - chamados a cada teste, para um não influenciar o outro

```
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f14CHF;

    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
    }
}
```

JUnit

- Rescrevendo nossos testes:

```
public void testEquals() {
    Assert.assertTrue(!f12CHF.equals(null));
    Assert.assertEquals(f12CHF, f12CHF);
    Assert.assertEquals(f12CHF, new Money(12, "CHF"));
    Assert.assertTrue(!f12CHF.equals(f14CHF));
}

public void testSimpleAdd() {
    Money expected= new Money(26, "CHF");
    Money result= f12CHF.add(f14CHF);
    Assert.assertTrue(expected.equals(result));
}
```

JUnit

- Executando os testes
  - Testes individuais
  - *Test Suites* (conjunto de testes)
- Forma estática:
  - Sobrescrever o método runTest():

```
TestCase test= new MoneyTest("simple add") {
    public void runTest() {
        testSimpleAdd();
    }
};
```

JUnit

- Forma dinâmica:
  - Nome do teste é o nome do método:

```
TestCase test= new MoneyTest("testSimpleAdd");
```

  - Mais flexível, porém mais sujeita a erros

JUnit

- Falta rodar os dois testes, em um só *Test Suite*

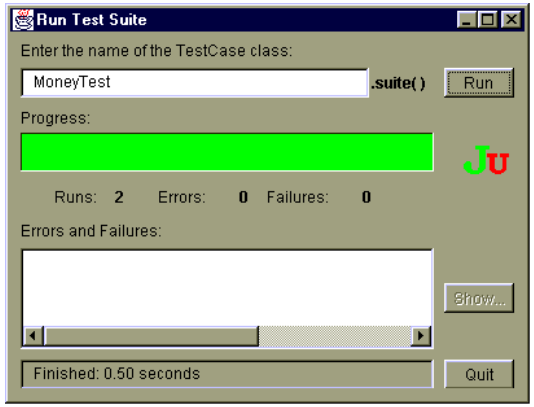
```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
```

  - Ou ainda assim :

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

(todos métodos começados com 'test')

JUnit



Testar tudo?

- "Testes provam a presença de erros, mas nunca sua ausência (Dijkstra)"
- Tente testar todo o comportamento
- Faça testes para várias condições, imaginando entrar em todos if's e else's
- Faça testes para condições de contorno (condições limites e para entradas inválidas)
- "Não inclua testes 'besta'. Porém, cuidado para não achar que 'tudo é teste besta'".

Testar tudo?

- If it can't break *on its own*, it's too simple to break.
  - getX(), normalmente não faz nada
  - setX(), pode também não fazer nada, mas se tiver validação pode precisar de teste
  - Métodos simples

```
public void myMethod(final int a, final String b) {
    myCollaborator.anotherMethod(a, b);
}
```

Se myCollaborator é nulo, é responsabilidade de outro método

Se anotherMethod não funciona, ele é o responsável