Introdução ao JUnit

Testes Unitários

O que são Teste Unitários?

Imagine por exemplo, se um avião só fosse testado após a conclusão de sua construção, com certeza isso seria um verdadeiro desastre, é nesse ponto que a engenharia aeronáutica é uma boa referência em processos de construções de projetos de software, principalmente em sistemas de missão crítica, pois durante a construção e montagem de um avião todos os seus componentes são testados isoladamente até a exaustão, e depois cada etapa de integração também é devidamente testada e homologada.

O teste unitário, de certa forma se baseia nessa ideia, pois é uma modalidade de testes que se concentra na verificação da menor unidade do projeto de software. É realizado o teste de uma unidade lógica, com uso de dados suficientes para se testar apenas à lógica da unidade em questão.

Em sistemas construídos com uso de linguagens orientadas a objetos, essa unidade pode ser identificada como um método, uma classe ou mesmo um objeto.

Tipos de Teste

Test Case (para cada classe) - Desenvolvedor (Projeta, escreve e roda).

Test Suite(Roda vários test cases) - Coordenador e Desenvolvedor (Projeta, escreve e roda).

Vale lembrar que o Teste de aceitação (homologação) é feito junto ao cliente.

Outra visão nova, interessante e muito polêmica, é a aproximação da responsabilidade dos testes ao programador, o que em algumas outras abordagens metodológicas é feito somente por equipes separadas, como por exemplo, uma equipe de teste/homologação.

Porém esse contexto é a base de qualquer metodologia ágil, pois dessa forma, o próprio programador, ao criar e executar os testes, adquiri um controle maior e imediato na prevenção e correção de bugs, contribuindo substancialmente para redução do tempo de vida de um projeto.

O que testar?

Sempre nós ficamos em dúvida sobre o que devemos testar em nossas classes, existem alguns macetes que podem nos ajudar a descobrir quais e quantos testes deverão ser escritos:

- A principal regra para saber o que testar é: “Tenha criatividade para imaginar as possibilidades de testes”.
- Comece pelas mais simples e deixe os testes “complexos“ para o final.
- Use apenas dados suficientes (não teste 10 condições se três forem suficientes)
- Não teste métodos triviais, tipo get e set.
- No caso de um método set, só faça o teste caso haja validação de dados.
- Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)!

Para fixar bem essas dicas, na Figura 1, temos um exercício de imaginação, onde você deverá achar as possibilidades de testes neste diagrama de classe.

Claro que não existe uma resposta única para esse exercício, pois você, baseando-se em suas experiências anteriores e sua criatividade, pode ter várias visões acerca dessa classe.

Figura 1 - Classe para exercício de imaginação.

JUnit

O JUnit é um framework que facilita o desenvolvimento e execução de testes unitários em código Java. Além disso o JUnit já vem instalado e configurado nas versões recentes de IDE’s como Eclipse, NetBeans, JBuilder, BlueJ entre outros.

Usando o JUnit na Prática

A idéia principal é sempre lembrar das cores **vermelho**, **verde** e **verde**. Porque essas cores? Antes de começar a desenvolver o exemplo de uma calculadora devemos pensar nos testes que devem ser feitos, antes de começar a implementar. Essa é uma das metodologias utilizadas no *Test-driven development* (TDD). O vermelho é etapa inicial, onde não existe nada implementado, pensamos apenas nas assinaturas de cada método.

Crie um projeto Java Desktop com o nome CalculadoraTeste e o pacote **projCalculadora**.

```
package projCalculadora;

public class Calculadora {

  public int soma(int a, int b){
     return 0;
  }

  public int subtracao(int a, int b){
     return 0;
  }

  public double multiplicacao(int a, int b){
     return 0;
  }

  public double divisao(int a, int b){
     return 0;
  }

}
```

Classe Inicial da Calculadora

Para criar um Teste usando o Netbeans 7 clique com o botão direito sobre o projeto e escolha a opção Testes Junit (lembrando que você precisa adicionar o JUnit ao classpath do seu projeto) como na Figura 2 abaixo.

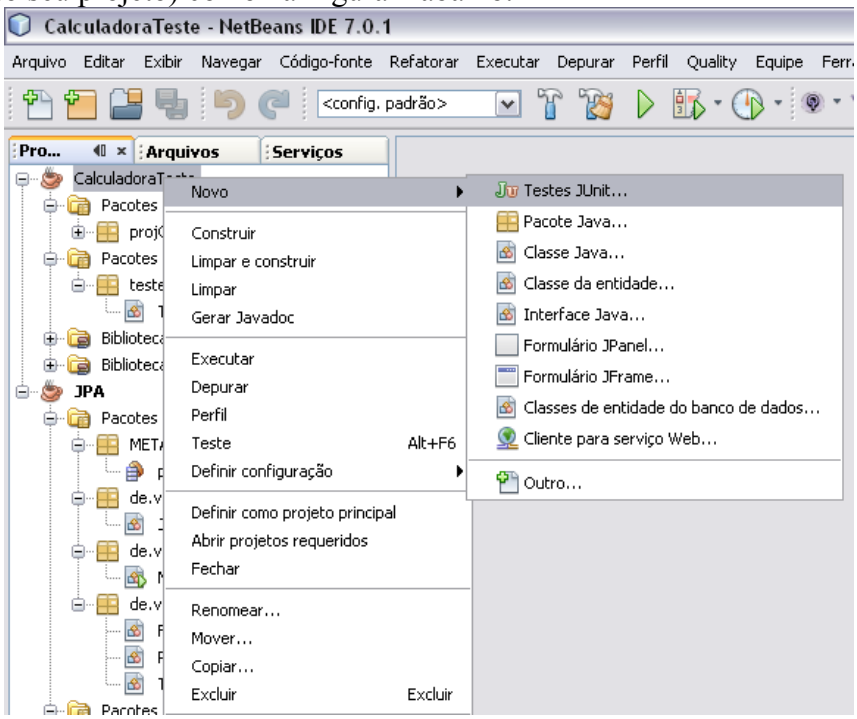


Figura 2 – Criando uma classe de Teste

Caso não apareça a opção de Testes JUnit faça o seguinte clique o botão direito sobre o seu projeto escolha Novo -> Outro -> JUnit e escolha a opção Testes JUnit. Como mostrado na Figura 3.

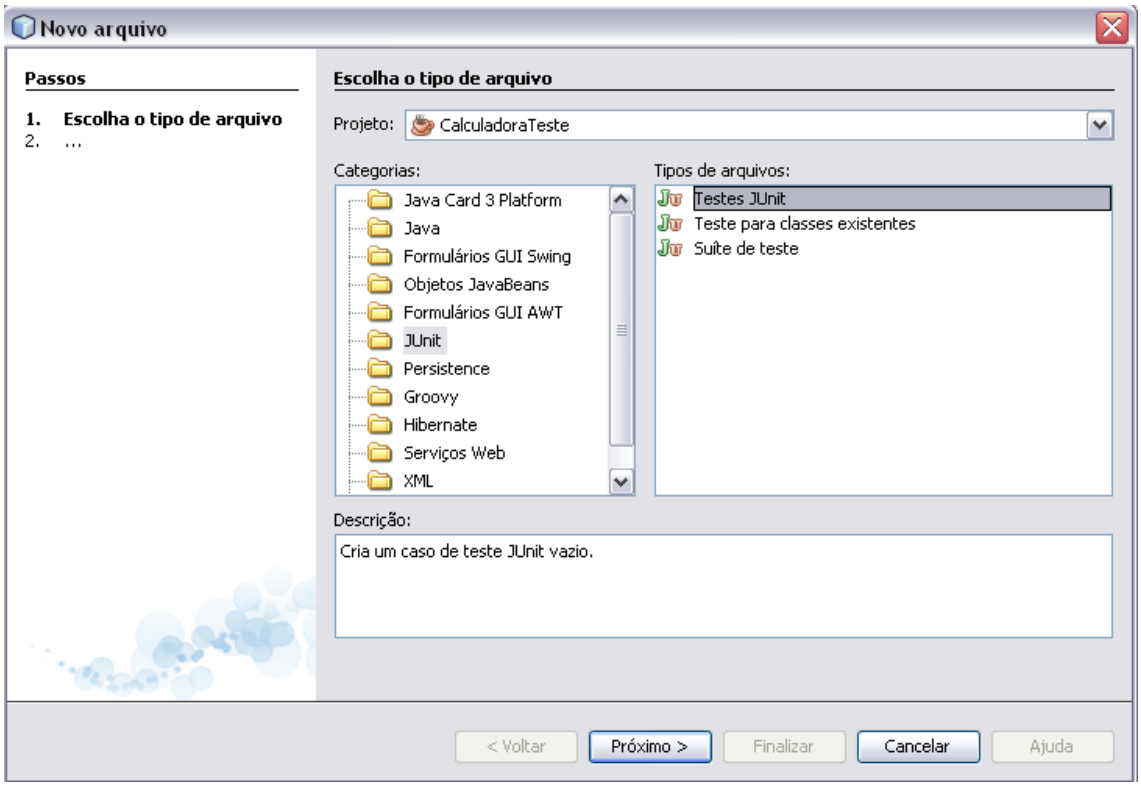


Figura 3 – Criando a classe de Testes JUnit

Preencha o nome da classe com o nome TestaCalculadora como na Figura 4 abaixo.

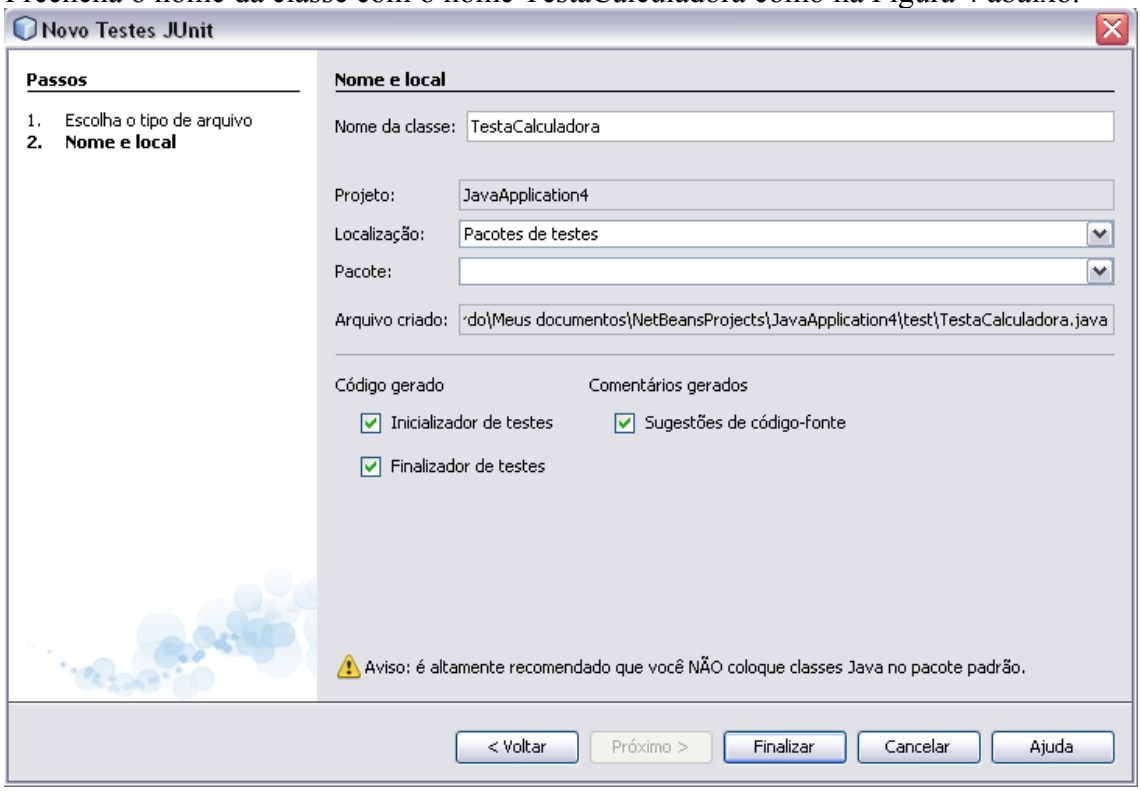


Figura 4 – Classe de Teste

Escolha a opção JUnit 4.x e clique em Selecionar como na Figura 5.

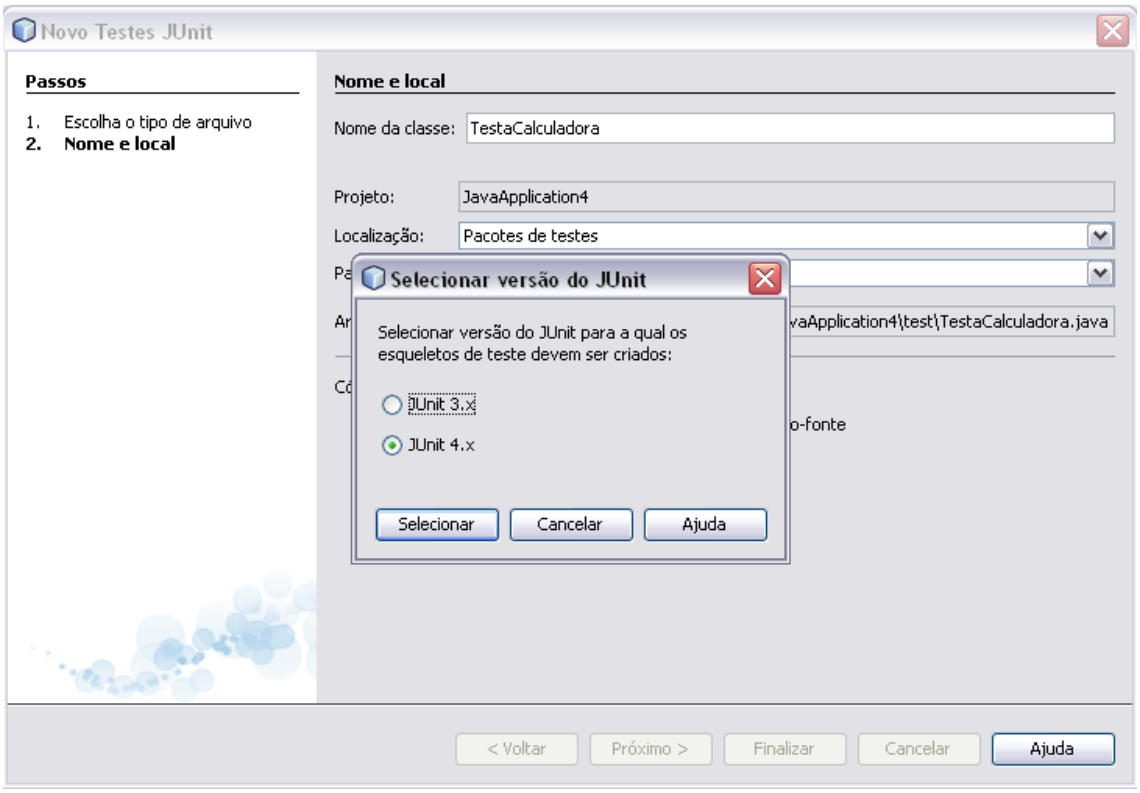


Figura 5 – Selecionando a versão do JUnit

Após esses passo será criada a nossa classe de testes TestaCalculadora.java, retire todos os métodos e construtor criados por padrão. E adicione na classe o seguinte o código:

```
import projCalculadora.Calculadora;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestaCalculadora {

    @Test
    public void assertSoma(){
        Calculadora c = new Calculadora();
        assertEquals(2, c.soma(1, 1));
        assertEquals(3, c.subtracao(6, 3));
        //0,1 é o delta – o delta é o valor máximo entre o valor real (c.multiplicacao(2,
2)) // e o esperado (4.0) para o qual ambos os números ainda são considerados iguais.
//      assertEquals(new Double(4.0), new Double(c.multiplicacao(2, 2)));
        assertEquals(4.0, c.multiplicacao(2, 2), 0.1);
        assertEquals(5.0, c.divisao(25, 5), 0.1);
    }
}
```

Classe de Teste

Coloque 2 no assertEquals porque você sabe que a soma de 1 + 1 tem retornar 2, quem define isso é o programador. Execute o caso de teste como mostrado na Figura 6.

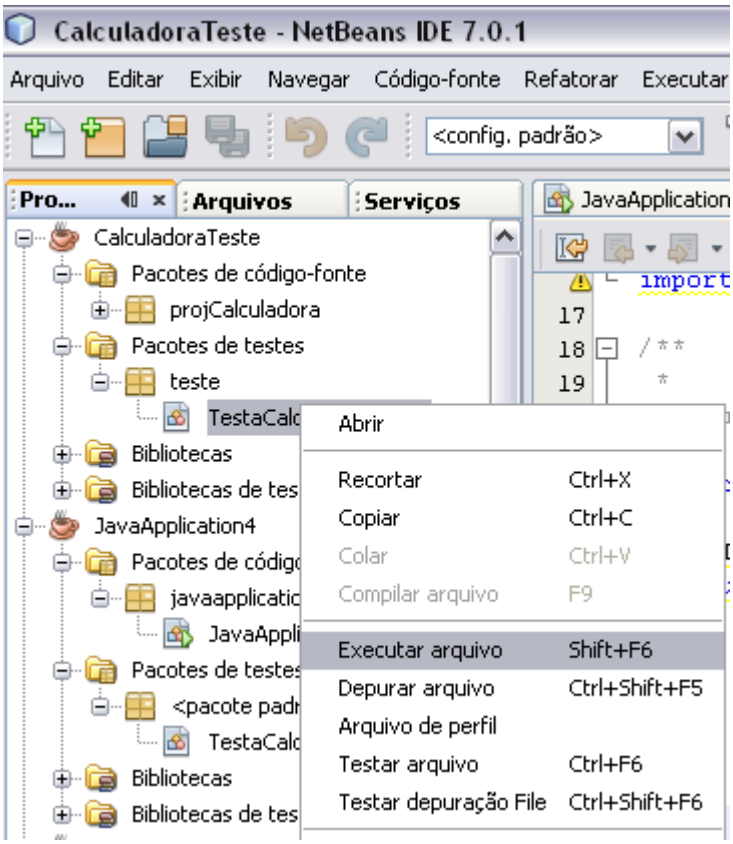


Figura 5 – Executando o caso de Teste

Após executar o caso de teste, o teste vai falhar o que é normal porque não temos nada implementado. **Falha do Caso de Teste.**

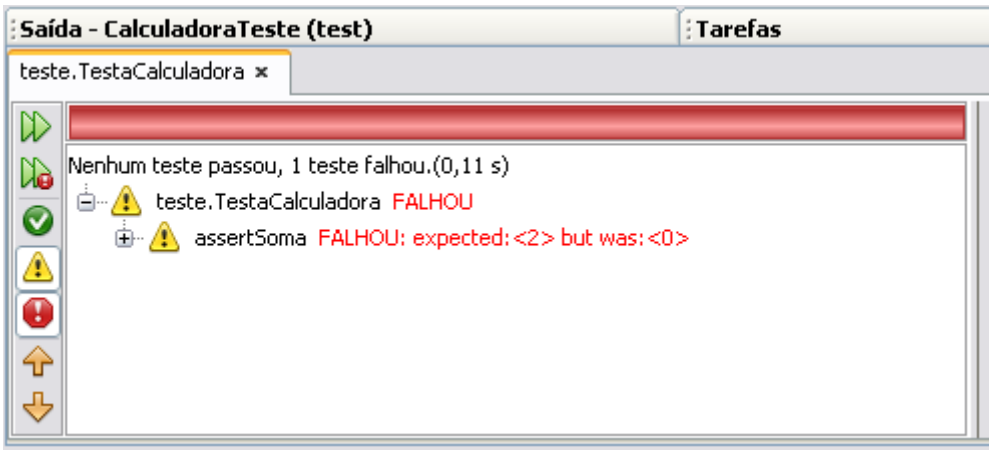


Figura 5 – Primero Vermelho

Lembrando das cores **vermelho**, **verde** e **verde**. Agora precisamos forçar a aparecer o primeiro verde, só para garantir que o nosso caso de teste está funcionando normalmente.

```
package projCalculadora;

public class Calculadora {

  public int soma(int a, int b){
     return 2;
  }

  public int subtracao(int a, int b){
     return 3;
  }

  public double multiplicacao(int a, int b){
     return 4;
  }

  public double divisao(int a, int b){
     return 5;
  }

}
```

Classe Intermediária da Calculadora

Após executar o teste novamente depois da modicação dos métodos da classe calculadora temos que todos os testes passam no teste.

```
import projCalculadora.Calculadora;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestaCalculadora {

   @Test
   public void assertSoma(){
      Calculadora c = new Calculadora();
      assertEquals(2, c.soma(1, 1));
      assertEquals(3, c.subtracao(6, 3));
      //0,1 é o delta – o delta é o valor máximo entre o valor real (c.multiplicacao(2,
2)) // e o esperado (4.0) para o qual ambos os números ainda são considerados iguais.
      assertEquals(4.0, c.multiplicacao(2, 2), 0.1);
      assertEquals(5.0, c.divisao(25, 5), 0.1);
   }
}
```

Classe de Teste

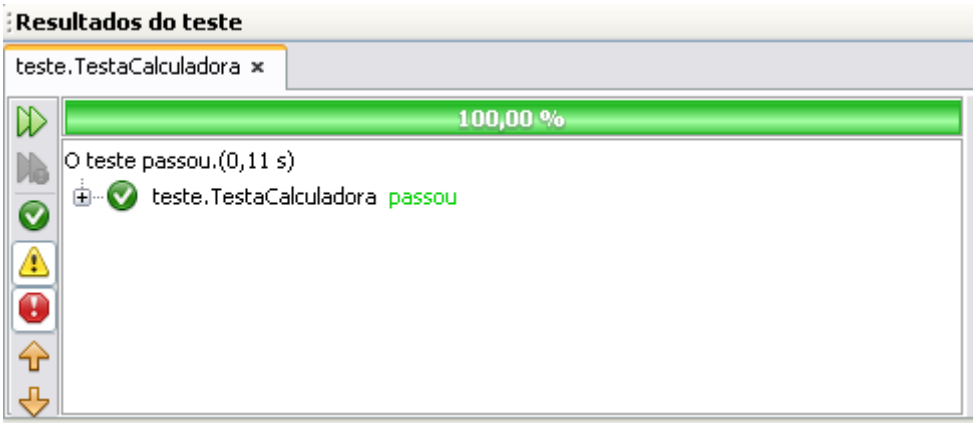


Figura 6 – Primeiro Verde

Agora precisamos implementar os métodos que estão sendo testados e sempre testa-lo a cada modificação.

Após todos os métodos como no código a seguir.

```
package projCalculadora;

public class Calculadora {

  public int soma(int a, int b){
     return a + b;
  }

  public int subtracao(int a, int b){
     return a - b;
  }

  public double multiplicacao(int a, int b){
     return a * b;
  }

  public double divisao(int a, int b){
     if(b = = 0){
        return 0;
     }
     return a / b;
  }

}
```

Classe Final da Calculadora

Precisamo executar o teste novamente para verificar se está tudo OK, pois os casos de teste que passaram no teste anteriormente precisam continuar passando no teste após a implementação.

```
import projCalculadora.Calculadora;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestaCalculadora {

   @Test
   public void assertSoma(){
      Calculadora c = new Calculadora();
      assertEquals(2, c.soma(1, 1));
      assertEquals(3, c.subtracao(6, 3));
      //0,1 é o delta – o delta é o valor máximo entre o valor real (c.multiplicacao(2,
2)) // e o esperado (4.0) para o qual ambos os números ainda são considerados iguais.
      assertEquals(4.0, c.multiplicacao(2, 2), 0.1);
      assertEquals(5.0, c.divisao(25, 5), 0.1);
   }
}
```

Classe de Teste

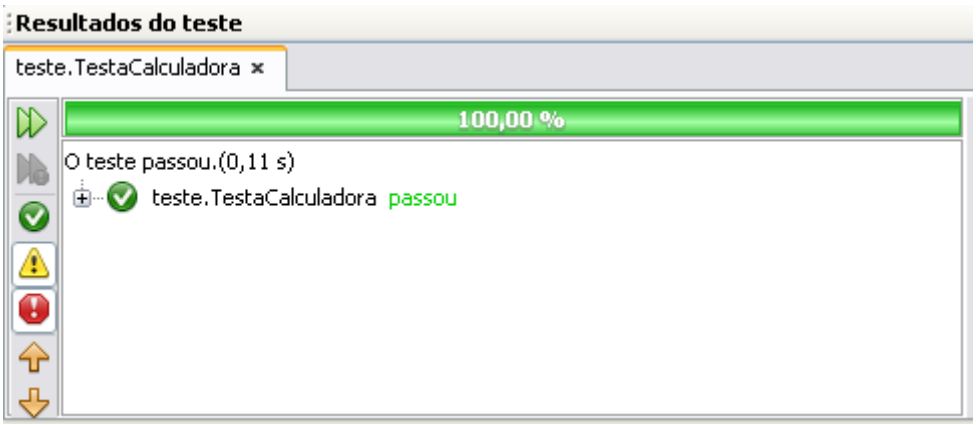


Figura 6 – Segundo Verde

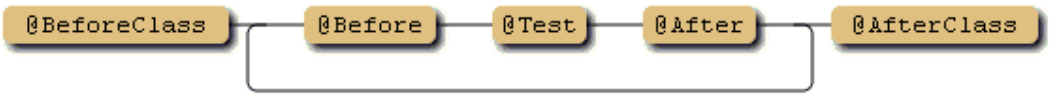
Método	Descrição
assert( )	Este foi substituído em JUnit 3.7, porque interfere com a palavra-chave assert do o J2SE 1.4. Você deve usar assertTrue () ao invés desse método. Este método foi totalmente removido em JUnit 3.8.
assertEquals( )	Compara dois valores de igualdade. O teste passa se os valores são iguais.
assertFalse( )	Avalia uma expressão booleana. O teste passa se a expressão é falsa.
assertNotNull( )	Compara uma referência de objeto nula. O teste passa se a referência não é nula.

assertNotSame( )	Compara o endereço de memória de duas referências de objeto usando o operador = =. O teste passa se ambos se referem a objetos diferentes.
assertNull( )	Compara uma referência de objeto nula. O teste passa se a referência é nula.
assertSame( )	Compara o endereço de memória de duas referências de objeto usando o operador = =. O teste passa se ambos se referem ao mesmo objeto.
assertTrue( )	Avalia uma expressão booleana. O teste passa se a expressão é verdadeira.
fail( )	Faz com que o teste atual falhe. Isto é comumente usado com a manipulação de exceção.

Anotações JUnit4.x

● @Test - Identifica que este método é um método de teste.
● @Before - Executará o método antes de cada teste. Este
método pode ser usado para preparar o ambiente de teste, por exemplo, lendo dados do usuário.
● @After - Executará o método após cada teste.
● @BeforeClass - Executará o método antes do início de todos os testes. Por exemplo: usado para se conectar a um banco de dados.
● @AfterClass - Executára o método após todos os testes finalizarem. Por exemplo: usado para desconectar a base de dados.
● @Ignore - O método será ignorado. Usado quando fizemos alguma mudança e ainda não adaptamos o teste.
● @Test (expected=IllegalArgumentException.class) – Testa se o método levanta a exceção especificada.
● @Test (timeout=100) – Falha se o teste levar mais que 100 milisegundos

Então todo o período de desenvolvimento ocorrerá da seguinte forma:



Contato:

Universidade de São Paulo (USP)
Ricardo Ramos de Oliveira 7250350
Email: ricardoramos.usp@gmail.com

Referências

Livro:
Java Extreme Programming Cookbook, O'REILLY. Eric M. Burke & Brian M. Coyner

Caelum F-16

Links:
http://luizgustavoss.wordpress.com/2009/04/05/testes-unitarios-com-junit/

http://www.vogella.de/articles/JUnit/article.html

http://www.devmedia.com.br/articles/viewcomp.asp?comp=1432

http://www.google.com/url?sa=t&source=web&cd=1&ved=0CBUQFjAA&url=http%3A%2F%2Fwww.cefetrn.br%2F~fellipe%2Flib%2Fexe%2Ffetch.php%3Fid%3Ddisc%253Atads%253Apds%253Ahome%26cache%3Dcache%26media%3Ddisc%3Atads%3Apds%3Atestes_junit4x.pdf&rct=j&q=www.cefetrn.br%2F~fellipe%2Flib%2Fexe%2Ffetch.php%3F%20%5BPDF%5D%20Testes%20com%20Junit%204.4&ei=oMBTTqyuE4fD0AH77_DSBQ&usg=AFQjCNHlj9BtgBnXQAfaT4r56m8KrCwgAQ&sig2=kKxDoEI29TUsWl90QOohiw&cad=rja

http://www.devmedia.com.br/post-4798-Criando-Testes-de-Unidades-com-o-Junit-4-usando-anotacoes.html
http://pt.wikipedia.org/wiki/JUnit
http://www.guj.com.br/articles/40
http://blog.jromay.es/2010/04/29/pruebas-unitarias-con-junit-basico/
http://javafree.uol.com.br/dependencias/tutoriais/testes_junit.pdf
http://edgarddavidson.com/?p=537
http://junit.sourceforge.net/doc/cookstour/cookstour.htm
http://www.junit.org/