Testes Unitários com JUnit

Por: Luiz Gustavo Stábile de Souza
http://luizgustavoss.wordpress.com

Abril/2009

Motivação

Apesar de já fazer um certo tempo que escrevi a primeira versão deste tutorial, o assunto de testes unitários, e a utilização do JUnit ainda são para alguns uma novidade.
Acredito que tanto a utilização de testes unitários quanto a utilização do JUnit deverão ser, cada vez mais, constantes no dia-a-dia dos profissionais.
Portanto, eis aqui a minha contribuição para quem está começando!

O que são Testes Unitários?

Falando de forma simples e direta, um teste unitário é um teste realizado para verificar a funcionalidade de um determinado trecho de código, verificar se ele realmente faz o que se propõe a fazer.
O objetivo de testes unitários não é testar toda a funcionalidade do sistema, ou a integração de várias partes do sistema de uma única vez, mas realizar testes isolados, testando blocos específicos do sistema, mais comumente os métodos das classes.
Para entender melhor esse conceito, vamos a exemplos práticos, primeiramente vendo como seria executar um teste sem a utilização de um framework de testes.

Testes sem o JUnit – Um exemplo prático

Para a realização dos exemplos deste tutorial, vou utilizar o IDE Netbeans. Aconselho que você também siga este tutorial executando os exemplos no Netbeans, pois alguns passos são específicos

para este IDE, a exemplo da última parte. Mas saiba que a criação de testes unitários não é de forma alguma dependente de uma IDE, e você também poderá criar testes unitários para suas classes, por exemplo, no Eclipse. A escolha aqui é meramente "didática".

Vamos começar criando um projeto Java simples com o nome de *Calculadora*. Neste projeto crie uma classe chamada *Calculadora*, e inclua nesta classe o código abaixo:

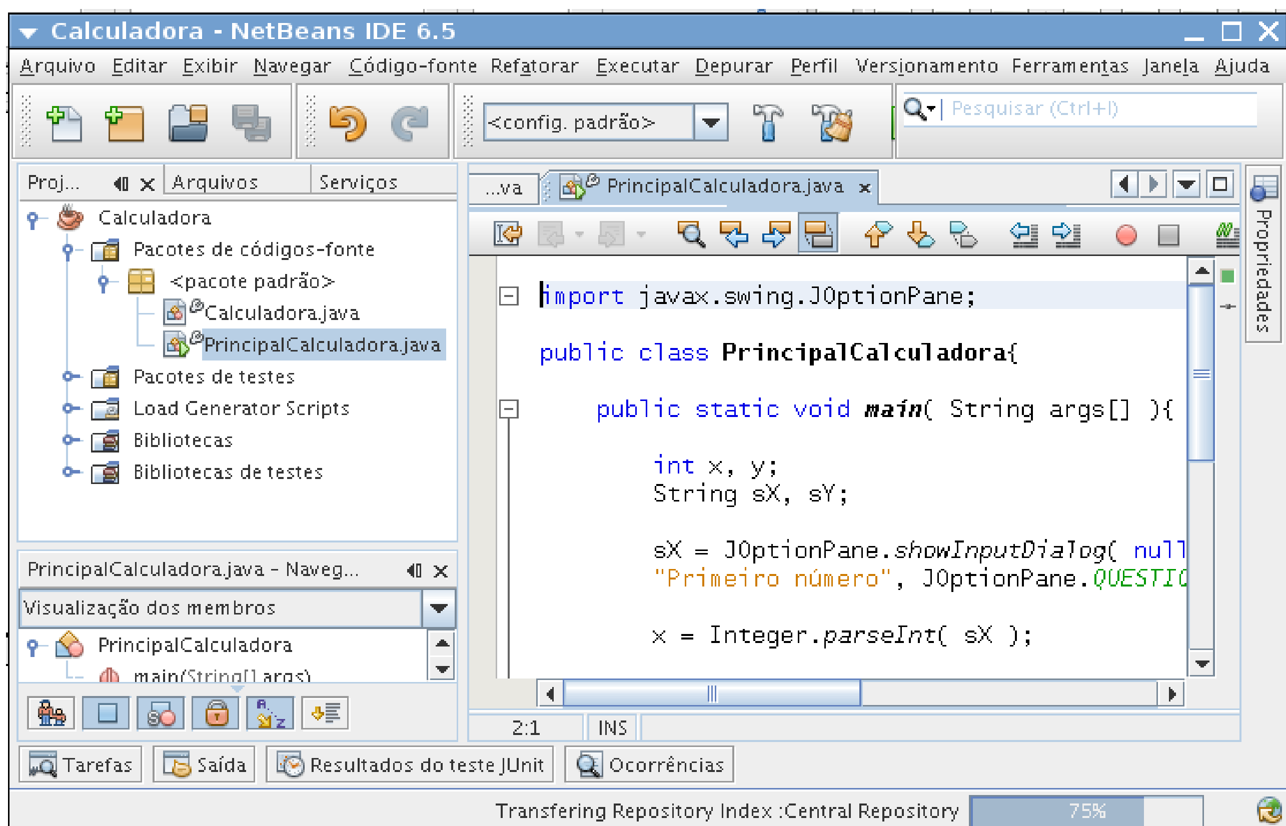
```
public class Calculadora{


        // atributo
        private int resultado = 0;


        // método somar
        public double somar( int n1, int n2 ){

            resultado = n1 + n2;
            return resultado;
        }

        // método subtrair
        public double subtrair( int n1, int n2 ){

            resultado = n1 - n2;
            return resultado;
        }

        // método multiplicar
        public double multiplicar( int n1, int n2 ){

            resultado = n1 * n2;
            return resultado;
        }

        // método dividir
        public double dividir( int n1, int n2 ){

            resultado = n1 / n2;
            return resultado;
        }
}
```

Repare que, no método *dividir*, propositadamente retiramos o teste que verifica a possibilidade de uma divisão por zero.

Crie agora uma classe chamada *PrincipalCalculadora*, que utilizará a primeira classe:

```
import javax.swing.JOptionPane;

public class PrincipalCalculadora{

    public static void main( String args[] ){
```

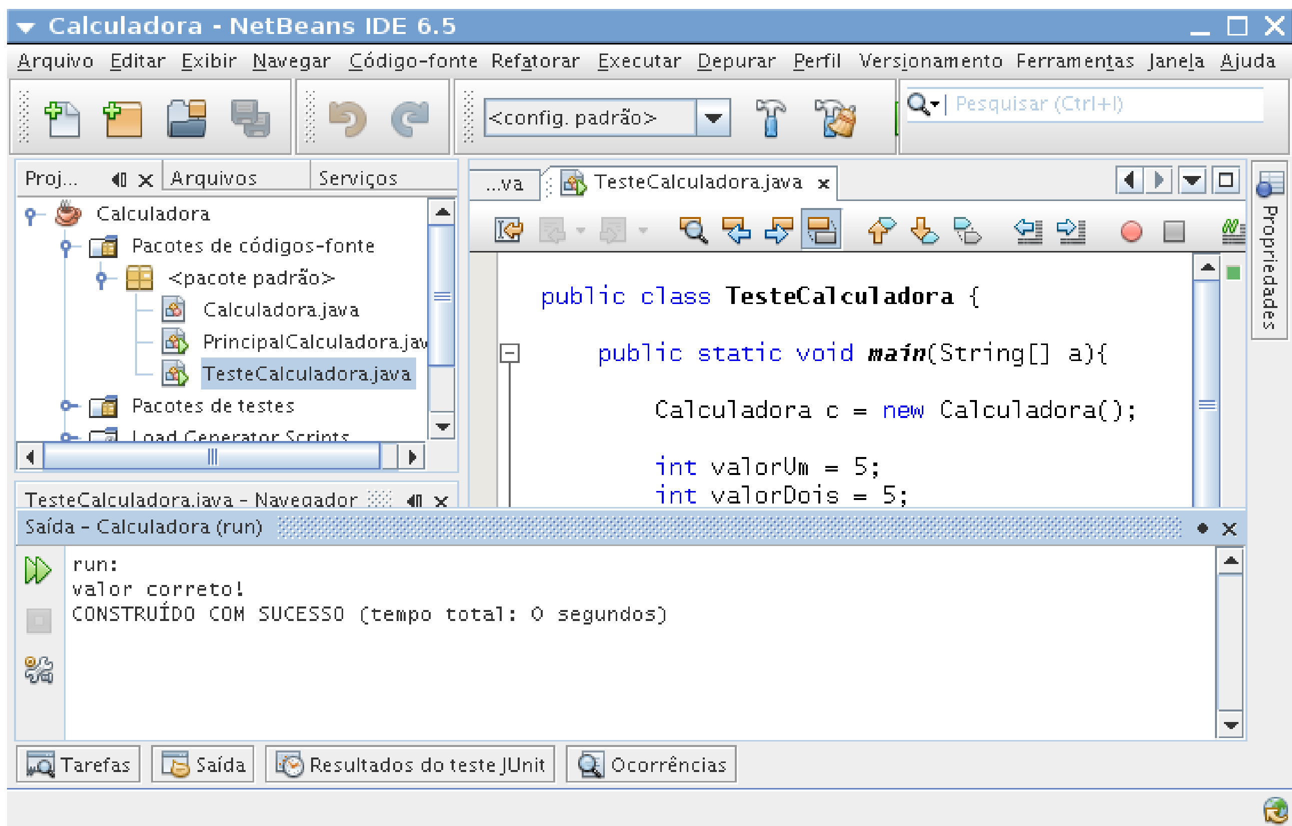
```
        int x, y;
        String sX, sY;

        sX = JOptionPane.showInputDialog( null, "Digite o primeiro número:",
        "Primeiro número", JOptionPane.QUESTION_MESSAGE );

        x = Integer.parseInt( sX );

        sY = JOptionPane.showInputDialog( null, "Digite o segundo número:",
        "Segundo número", JOptionPane.QUESTION_MESSAGE );

        y = Integer.parseInt( sY );

        // instanciação de um objeto da classe calculadora
        Calculadora calc = new Calculadora();

        JOptionPane.showMessageDialog(null, "somar: " +  calc.somar( x, y ) );
          JOptionPane.showMessageDialog(null, "subtrair: " +  calc.subtrair( x,
y ) );
        JOptionPane.showMessageDialog(null, "multiplicar: " +  calc.multiplicar(
x, y ) );
           JOptionPane.showMessageDialog(null, "dividir: " +  calc.dividir( x,
y ) );

        System.exit( 0 );
    }
}
```



Vamos agora criar uma classe chamada *TesteCalculadora* para testar nossa classe *Calculadora*. Adicione a ela o seguinte trecho de código:

```
public class TesteCalculadora {

    public static void main(String[] a){

        Calculadora c = new Calculadora();

        int valorUm = 5;
        int valorDois = 5;

        double valorTotal = c.somar(valorUm, valorDois);

        if(valorTotal == 10){
            System.out.println("valor correto!");
        }
        else{
            System.out.println("valor errado!");
        }
    }

}
```

Clique sobre a classe *TesteCalculadora* com o botão direito e escolha a opção *Executar Arquivo* (*Run File*). A execução deve apresentar o resultado esperado, ou seja, o teste foi executado e o método se comportou como esperado:



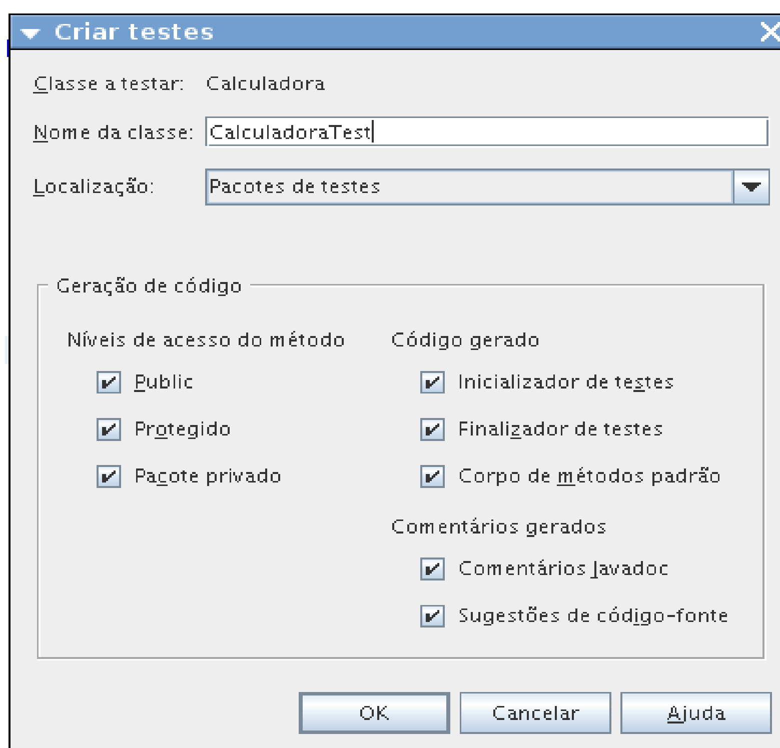
Nossa classe de teste é funcional, mas para a boa prática de programação, erros devem ser tratados com exceções (blocos try-catch), e não com avaliações condicionais. Criar testes desta maneira, com controle de exceções, além de poluir o código com blocos try-catch, é muito trabalhoso.
É nesse momento que se mostra útil a utilização de um framework de testes.

O Framework de Testes JUnit

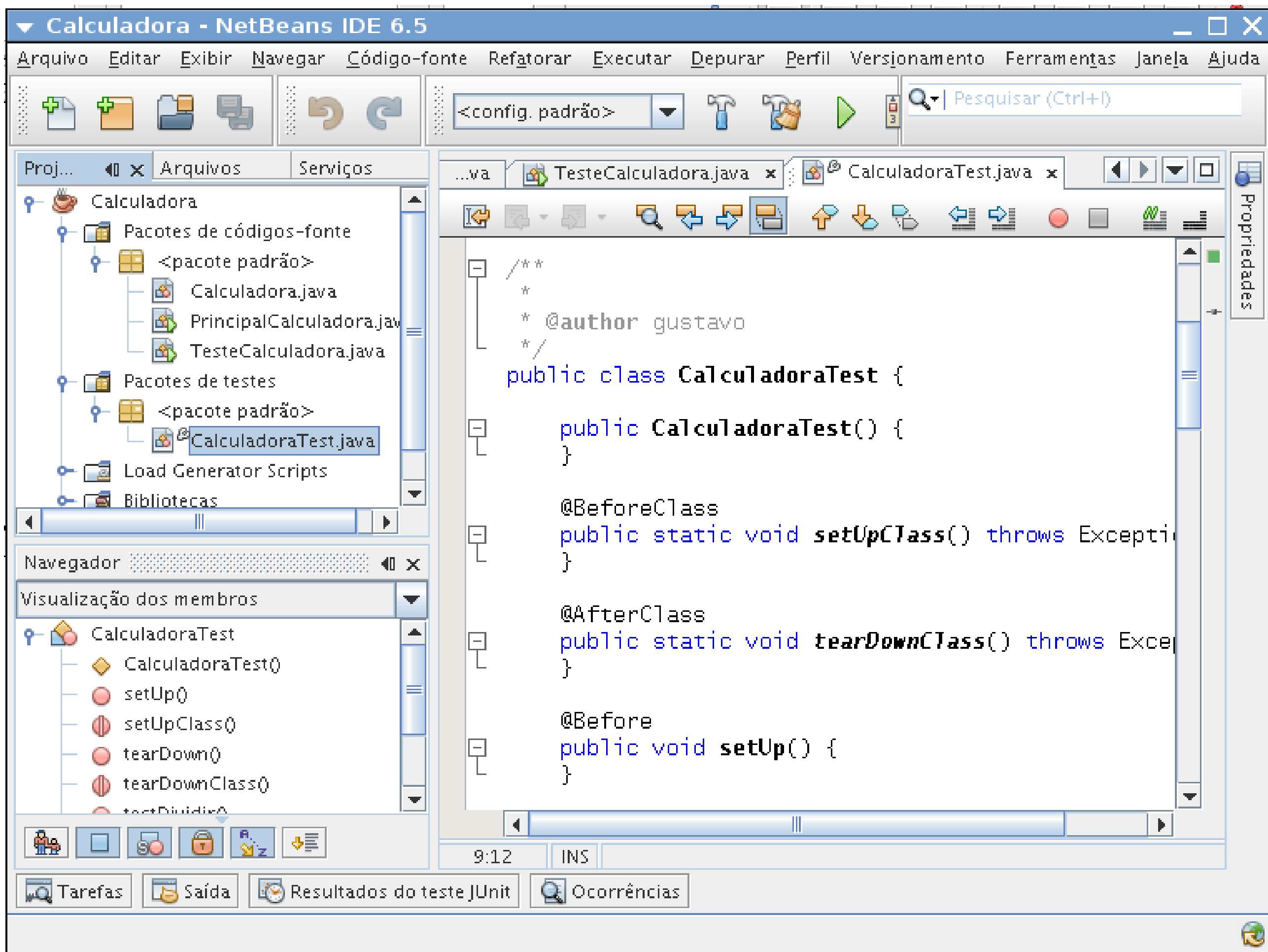
Existem vários frameworks para testes unitários na plataforma Java, e o mais famoso deles é o JUnit, um projeto open-source que atualmente está em sua versão 4.5. Na página do projeto [1] você poderá encontrar mais informações, além de tutoriais.
O Netbeans 6 dá suporte à versão mais recente do JUnit, que incorpora o uso de anotações para facilitar a criação de testes. A integração do Netbeans com o JUnit nos permite criar classes de teste rapidamente.

Testes com JUnit – Um exemplo prático

Agora que já vimos como criar uma classe de teste sem o JUnit, e vimos alguns dos inconvenientes dessa abordagem, vamos criar uma classe de teste para a nossa classe *Calculadora*, utilizando os recursos de integração do Netbeans ao JUnit.
Clique com o botão direito do mouse sobre a classe *Calculadora.* No menu que aparece, escolha a opção *Ferramentas > Criar testes JUnit (Tools > Create JUnit Tests).* Na próxima janela que aparece, escolha a opção *JUnit 4.x* e clique em *Select.* A seguinte tela será apresentada:



Nesta tela encontramos alguns recursos a serem configurados para a classe de teste, como a presença de métodos de inicialização e finalização de casos de testes, o nível de acesso dos métodos a serem testados, e a possibilidade de geração de comentários nos códigos. Repare também que é sugerido um nome para a classe de teste (*CalculadoraTest*). Podemos aceitar as opções sugeridas, clicando em *OK*. Após este procedimento, será criada a classe de teste no diretório *Pacote de Testes (Test Packages)*:



Além dos casos de teste simples, criados para cada método da classe *Calculadora*, foram criados outros 4 métodos que merecem comentário, são eles:

@BeforeClass
setUpClass()

Neste método devem ser colocados códigos que precisam ser executados antes da criação de um objeto da classe de teste, ou seja, um código do qual todos os métodos de teste podem tirar algum proveito. Pode ser a criação de uma conexão com o banco de dados, por exemplo, ou a leitura de um arquivo no sistema de arquivos.
A anotação que acompanha o método (@BeforeClass) pode ser adicionada a qualquer método, e

nesse caso, todos os métodos que tiverem essa anotação serão executados na ordem em que aparecem declarados, e antes de qualquer caso de teste específico.

@AfterClass
tearDownClass()

Neste método deverão ser colocados códigos que precisam ser executados assim que todos os casos de teste tiverem sido executados. Tais códigos podem ser referentes a liberação de recursos adquiridos no método setUpClass(), como o fechamento de conexões com o banco de dados, ou à liberação de arquivos.
Assim como acontece com a anotação @BeforeClass, a anotação @AfterClass pode acompanhar qualquer método, e nesses casos todos os métodos serão executados para a liberação de recursos, na ordem em que aparecem declarados.

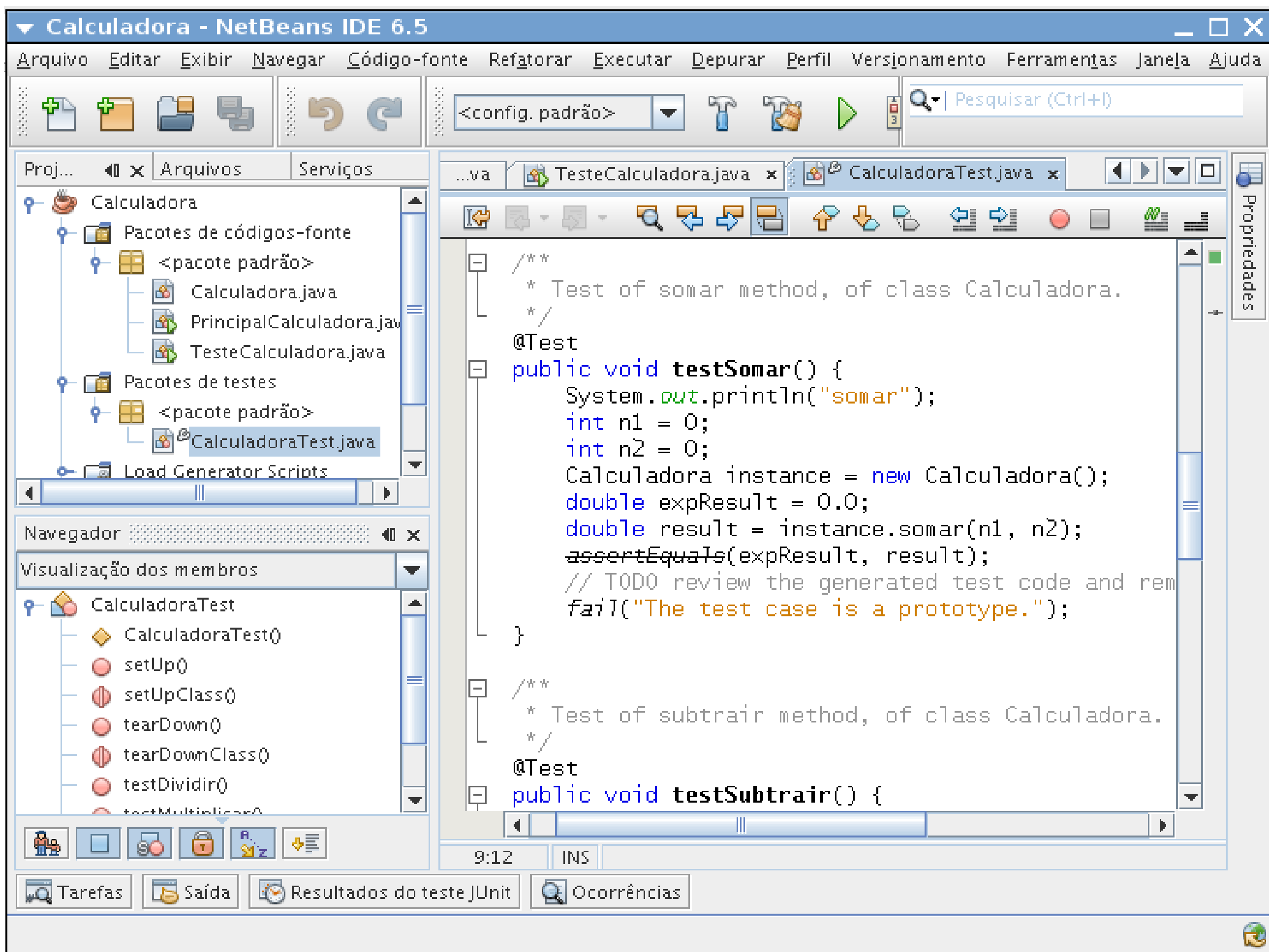
@Before
setUp()

O método setUp() pode ser utilizado para a inicialização de recursos antes da execução de cada método de teste. É o local ideal para obter e inicializar recursos que precisam ser reiniciados a cada teste.
Assim como as outras anotações, @Before pode ser adicionado a outros métodos.

@After
tearDown()

O método tearDown() é utilizado para a liberação de recursos ao final de cada método de teste. Estes recursos geralmente são os que foram obtidos no método setUp().
A anotação @After pode, assim como as demais, ser utilizada com outros métodos.

Agora vamos observar o método de teste criado para o método somar da classe *Calculadora*:



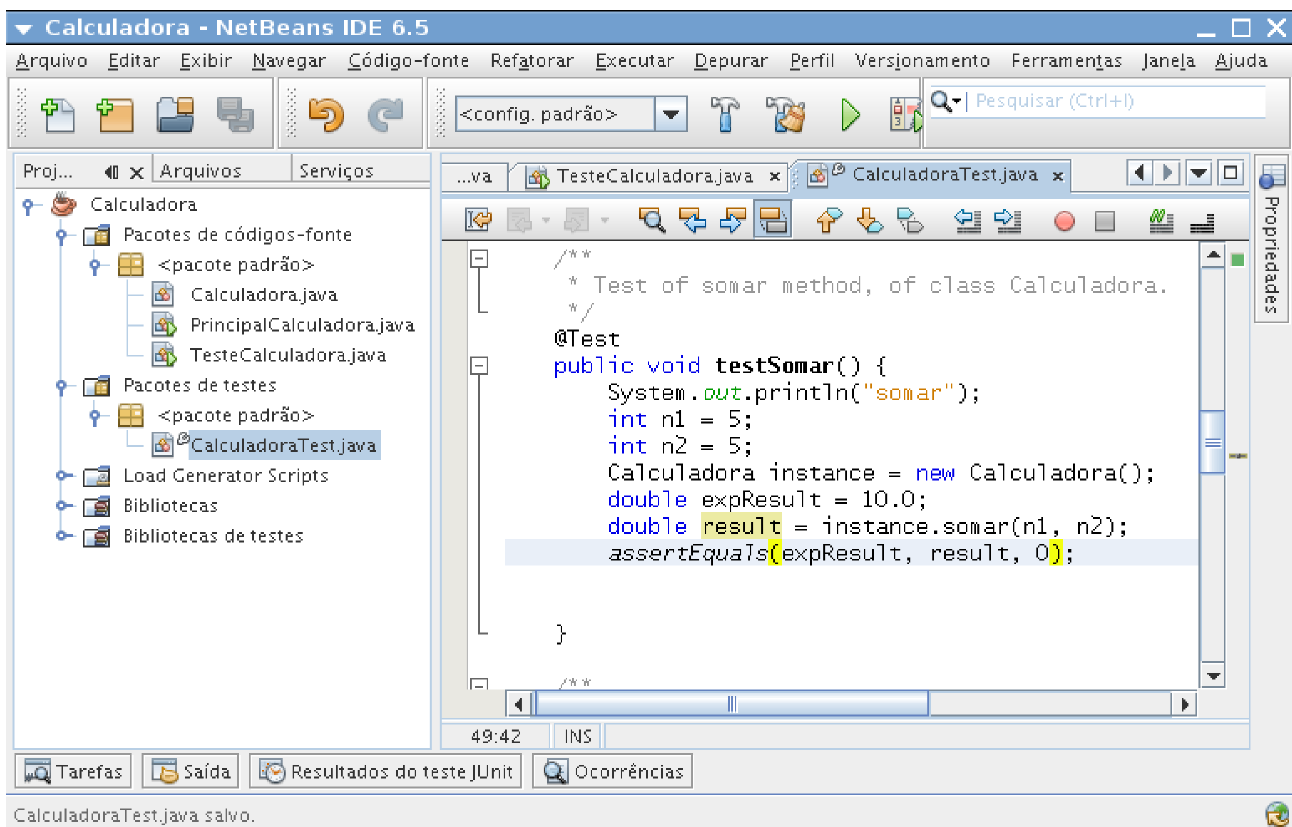
A anotação @*Test* indica que este método de teste deve ser executado pelo framework. Quando não quisermos que um método de teste seja executado basta remover a anotação. O que o método de teste faz é criar uma instância da classe *Calculadora*, e passar dois parâmetros para o método somar, verificando em seguida o resultado.

Como você pode observar, os valores iniciais das variáveis de teste não são nada otimizados, isto porque o framework não tem como conhecer as regras pertinentes ao nosso sistema, colocando então valores padrões. Cabe a nós a "lapidação" do caso de teste, passando valores mais adequados para o teste.

Repare também que a verificação do resultado é feita através de um método de asserção, chamado *assertEquals()*, que recebe dois parâmetros. O que este método faz é verificar a igualdade entre os parâmetros, e caso estes valores não sejam iguais, é retornada uma exceção, e o teste falha. Veja que não precisamos utilizar blocos condicionais, nem estruturas de controle de exceções, o que torna nosso código de teste mais limpo e direto.

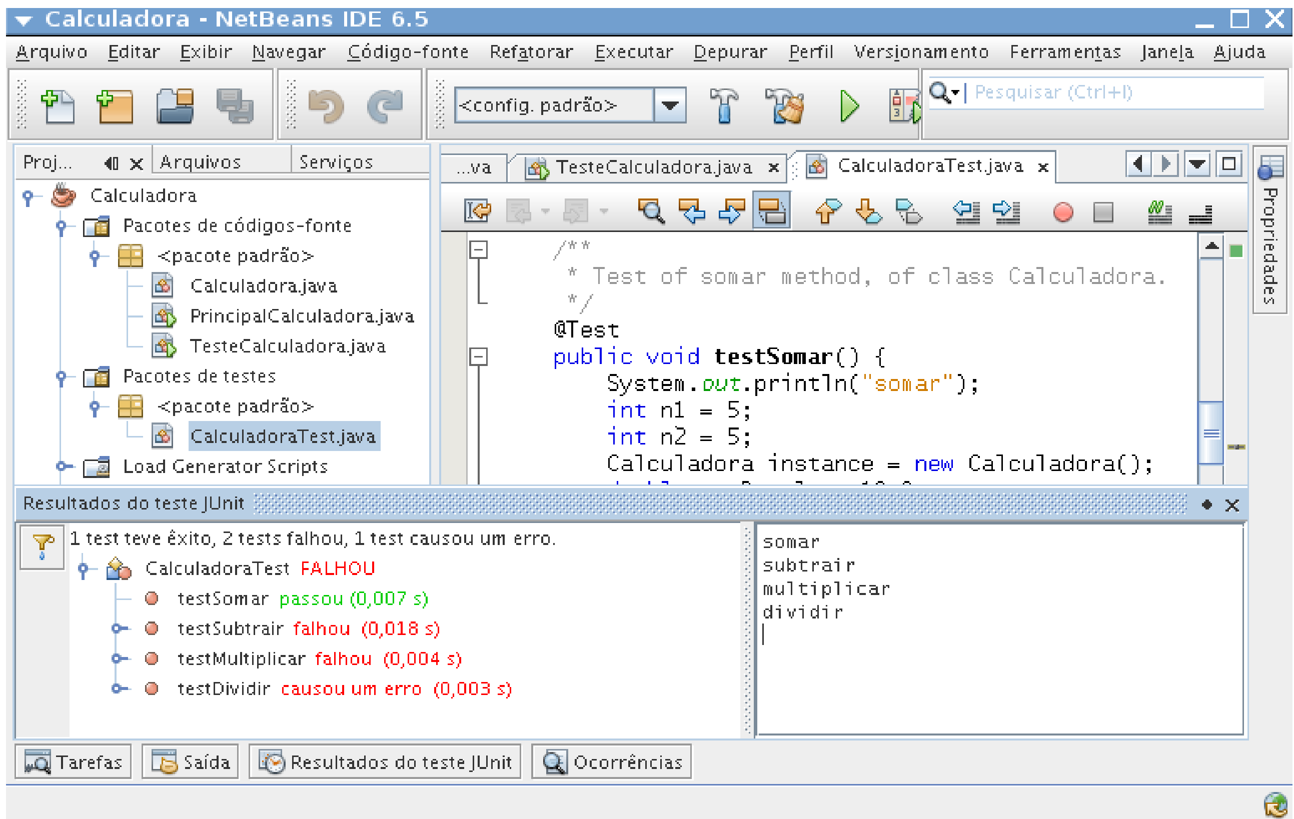
O comando fail() força a falha do teste, e no geral deve ser removido (a não ser que se queira realmente forçar a falha de um teste).

Vamos refinar nosso caso de teste:



Repare que melhoramos os valores das variáveis utilizadas no teste do método *testSomar*, e no método *assertEquals*, passamos um terceiro parâmetro, que especifica uma variação decimal aceitável para a comparação de números de ponto-flutuante.

Para rodar o teste, clique com o botão direito do mouse sobre o ícone que representa a classe *CalculadoraTest*, na árvore de recursos de projeto (à esquerda), e escolha a opção *Executar Arquivo (Run File)*. Você verá que todos os métodos de teste anotados com *@Test* serão executados, e como retiramos o comando *fail* somente da classe de teste *testSomar*, todos os outros testes falharão:

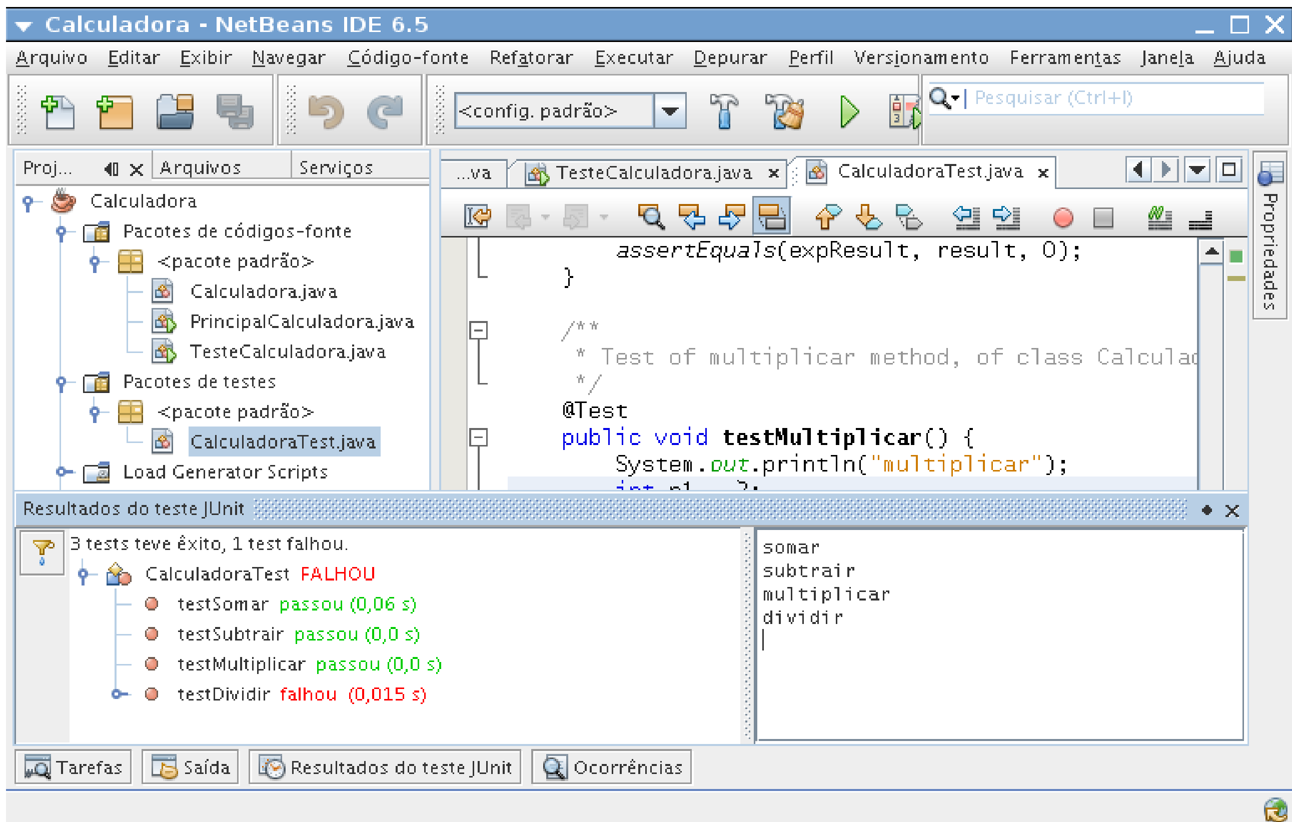


Vamos ajustar os demais testes. O código final da classe de testes, até agora, deverá ser o seguinte:

```
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 CalculadoraTest {

    public CalculadoraTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
```

```java
     * Test of somar method, of class Calculadora.
     */
    @Test
    public void testSomar() {
        System.out.println("somar");
        int n1 = 5;
        int n2 = 5;
        Calculadora instance = new Calculadora();
        double expResult = 10.0;
        double result = instance.somar(n1, n2);
        assertEquals(expResult, result, 0);
    }

    /**
     * Test of subtrair method, of class Calculadora.
     */
    @Test
    public void testSubtrair() {
        System.out.println("subtrair");
        int n1 = 5;
        int n2 = 3;
        Calculadora instance = new Calculadora();
        double expResult = 2;
        double result = instance.subtrair(n1, n2);
        assertEquals(expResult, result, 0);
    }

    /**
     * Test of multiplicar method, of class Calculadora.
     */
    @Test
    public void testMultiplicar() {
        System.out.println("multiplicar");
        int n1 = 2;
        int n2 = 3;
        Calculadora instance = new Calculadora();
        double expResult = 6;
        double result = instance.multiplicar(n1, n2);
        assertEquals(expResult, result, 0);
    }

    /**
     * Test of dividir method, of class Calculadora.
     */
    @Test
    public void testDividir() {
        System.out.println("dividir");
        int n1 = 5;
        int n2 = 2;
        Calculadora instance = new Calculadora();
        double expResult = 2.5;
        double result = instance.dividir(n1, n2);
        assertEquals(expResult, result, 0);
    }

}
```

Substitua o código da classe de testes pelo código acima e execute o teste. O último teste, do método dividir falha. Vamos analisar o que aconteceu:

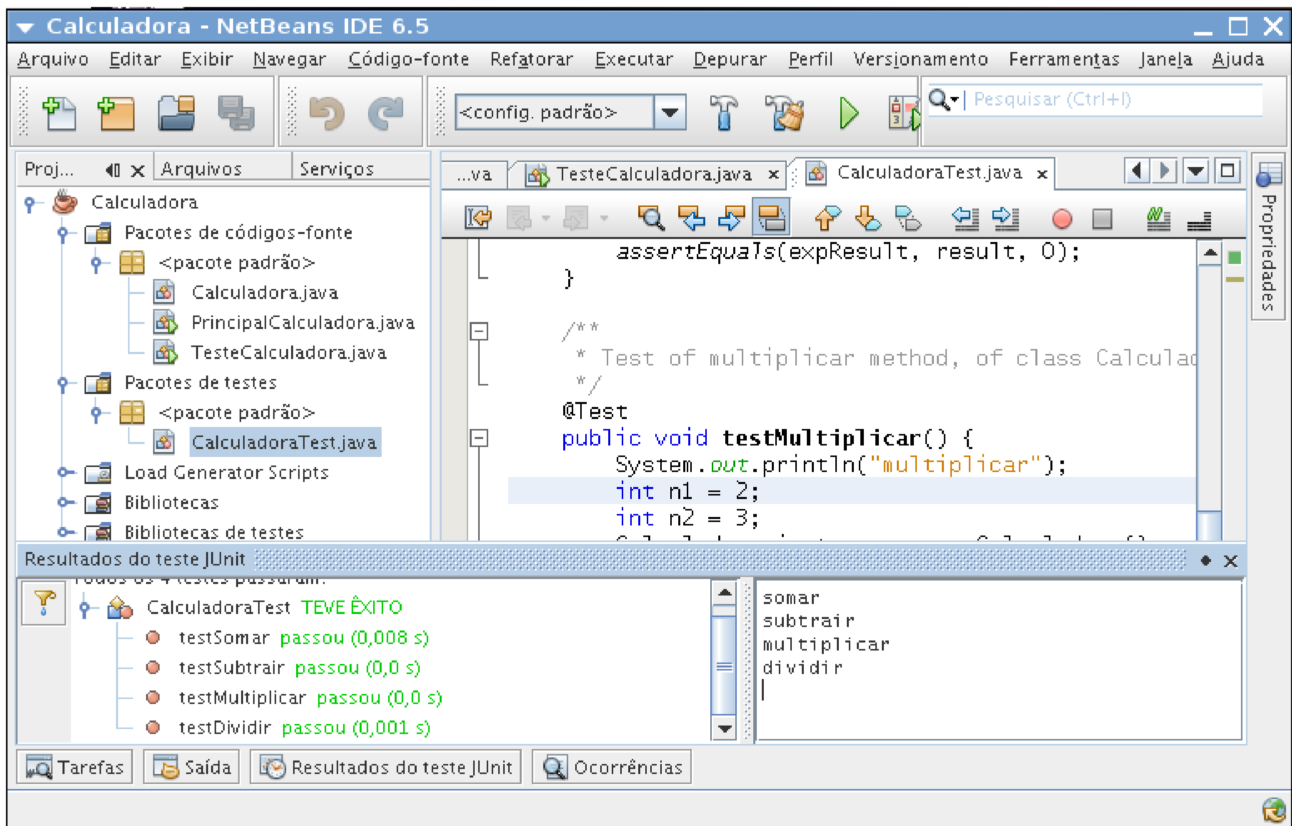


O método de teste para o método *dividir* determina que a divisão de 5 por 2 deve resultar em 2.5. Isto está correto, e devemos então verificar o método *dividir* na nossa classe *Calculadora*, para encontrar a causa do problema.

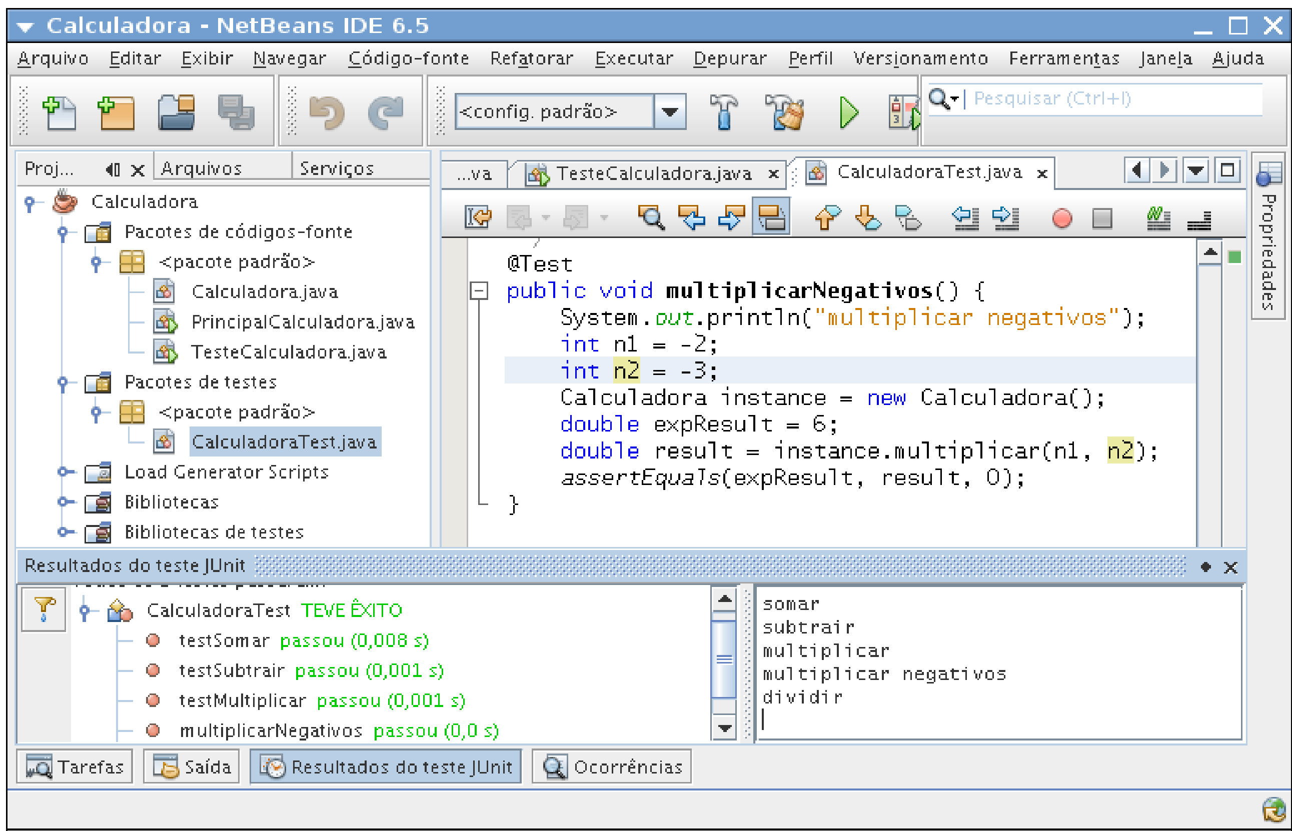
Ao analisar o código do método *dividir*, da classe *Calculadora*, vemos que ele retorna como resultado o valor da divisão direta de dois números inteiros. Qualquer operação que envolva um número inteiro retornará um número inteiro, e quando fazemos a divisão de 5 por 2, ele retorna somente o valor inteiro 2, é esse o problema de nosso método. Vamos alterar o código de nosso método *dividir*, na classe *Calculadora* para o seguinte:

```
public double dividir( int n1, int n2 ){

        double d1 = Double.valueOf(n1);
        double d2 = Double.valueOf(n2);

        double r = d1 / d2;
        return r;
}
```

Agora, sem nenhuma alteração em nossa classe de teste, execute os testes. Agora todos os testes passam:



Conforme vimos até agora, temos um caso de teste para cada método da nossa classe *Calculadora*. Isso é bom, mas não o bastante. Precisamos criar casos de teste para outras situações. Por exemplo, vamos testar o comportamento de nosso método *multiplicar* para o caso de parâmetros com valores negativos:



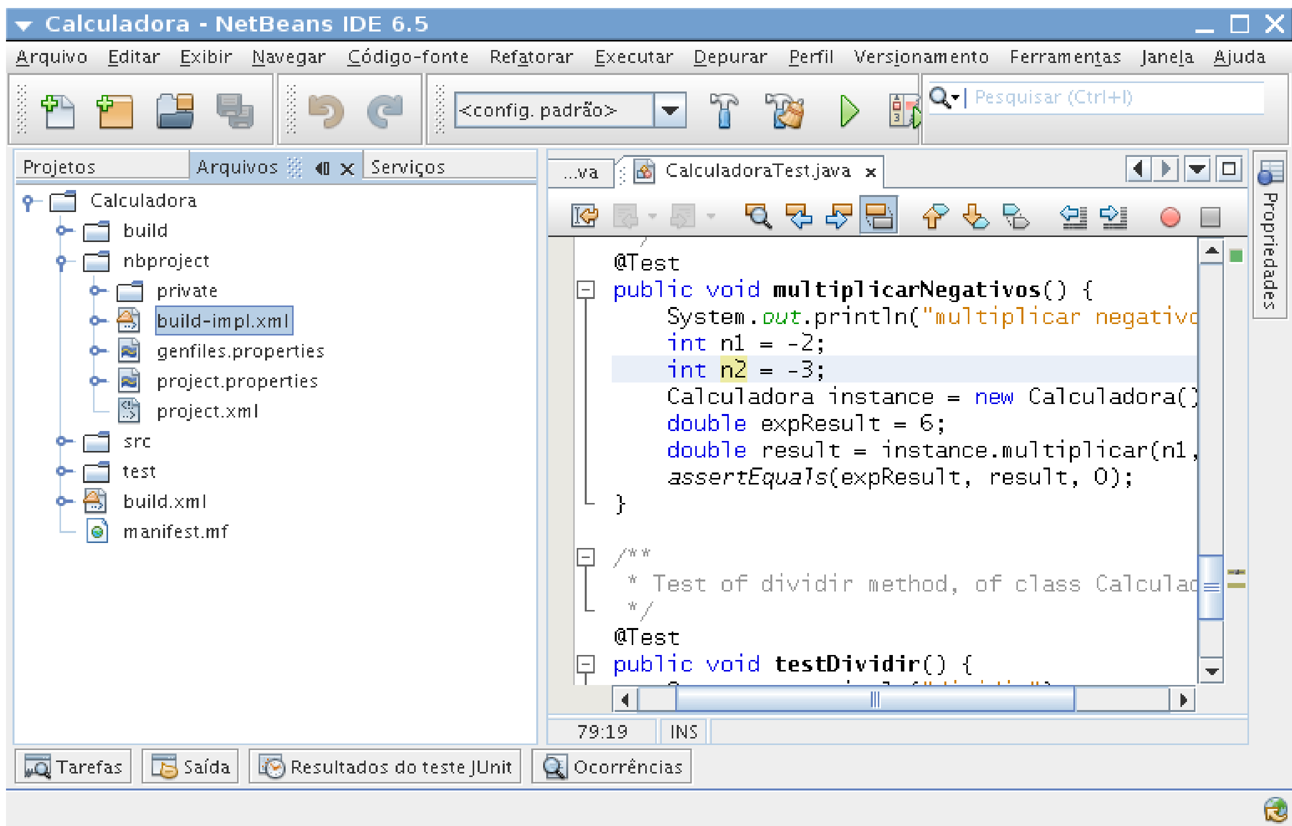
Como se pode ver, passamos dois valores negativos ao método *multiplicar*, e o mesmo nos retornou um valor positivo.

Consulte a API do framework JUnit na web e conheça os diversos métodos assertivos disponíveis. Há métodos assertivos para comparação de Strings, para garantir que um valor não é nulo, etc...

Gerando Relatórios de Teste com JunitReport

Podemos aplicar algumas modificações em um script de configuração do Netbeans para que sejam gerados relatórios dos testes realizados.

Na tela de arquivos (*Files*) procure pelo arquivo build-imp.xml. Trata-se de um arquivo de script *ant*, que o Netbeans utiliza para algumas configurações internas:

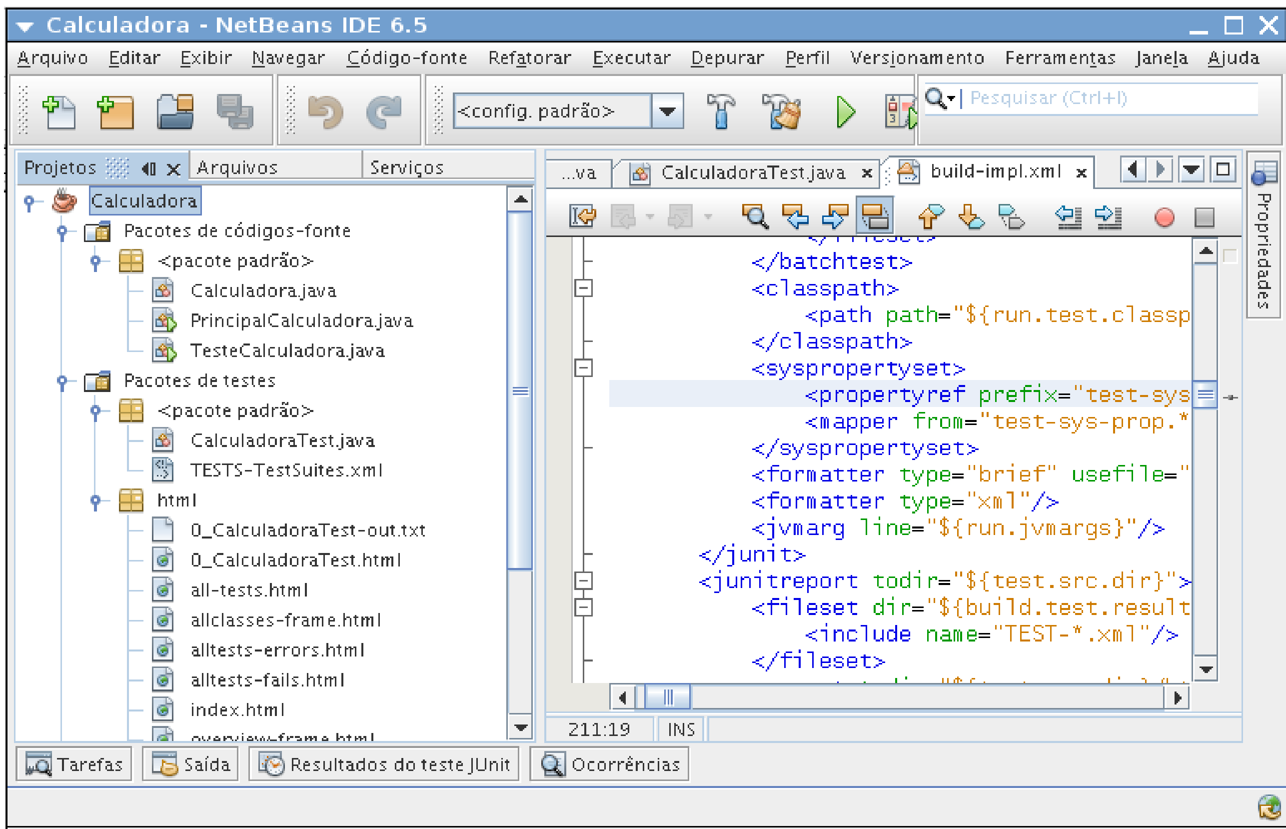


Clique duas vezes sobre o arquivo, para abri-lo no editor. Procure pela tag de fechamento </junit> e abaixo dela digite o seguinte código:

```
<junitreport todir="${test.src.dir}">
          <fileset dir="${build.test.results.dir}">
                    <include name="TEST-*.xml"/>
          </fileset>
          <report todir="${test.src.dir}/html"/>
</junitreport>
```

```
<junit dir="${work.dir}" errorproperty="tests.failed" failurepro
    <batchtest todir="${build.test.results.dir}">
        <fileset dir="${test.src.dir}" excludes="@{excludes},${e
            <filename name="@{testincludes}"/>
        </fileset>
    </batchtest>
    <classpath>
        <path path="${run.test.classpath}"/>
    </classpath>
    <syspropertyset>
        <propertyref prefix="test-sys-prop."/>
        <mapper from="test-sys-prop.*" to="*" type="glob"/>
    </syspropertyset>
    <formatter type="brief" usefile="false"/>
    <formatter type="xml"/>
    <jvmarg line="${run.jvmargs}"/>
</junit>
<junitreport todir="${test.src.dir}">
    <fileset dir="${build.test.results.dir}">
        <include name="TEST-*.xml"/>
    </fileset>
    <report todir="${test.src.dir}/html"/>
</junitreport>
```

Depois dessa modificação, clique sobre o projeto com o botão direito e escolha a opção *Teste*. Você perceberá que será criada uma estrutura a mais junto com as classes de teste. Esta estrutura passa a armazenar as páginas que apresentam os testes executados.



Clique com o botão direito do mouse sobre o arquivo *index.html* e escolha a opção *Visualizar*:

Home

Packages

<none>

Classes

CalculadoraTest

Unit Test Results.

Designed for use with JUnit and Ant.

Class CalculadoraTest

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
CalculadoraTest	5	0	0	1.430	2009-04-05T06:04:53	anion

Tests

Name	Status	Type	Time(s)
testSomar	Success		0.016
testSubtrair	Success		0.002
testMultiplicar	Success		0.001
multiplicarNegativos	Success		0.005
testDividir	Success		0.001

Properties »
System.out »

É isso pessoal! Espero que este pequeno tutorial seja útil.

Até o próximo!

Referências

[1] - http://www.junit.org/