Utilizando Testes de Unidade no Ciclo de Desenvolvimento de *Software* para Processamento e Análise de Imagens

Sabrina Borba Dalcin, Adriane Pedroso Dias Ferreira, Marcos Cordeiro d'Ornellas

Programa de Pós-Graduação em Engenharia de Produção (PPGEP)
Grupo de Processamento de Informações Multimídia (PIGS)
Centro de Tecnologia
Universidade Federal de Santa Maria (UFSM) – Santa Maria, RS – Brasil

{sabrinadalcin, dias.adriane, marcosdornellas}@gmail.com

Resumo

Este artigo tem por objetivo descrever algumas aplicações de casos de testes de unidade que foram desenvolvidas no decorrer do ciclo de desenvolvimento de *software*. Neste trabalho, é apresentada a utilização do *framework* JUnit para a realização dos testes de unidade em um *software* para processamento e análise de imagens, desenvolvido em Java. Finalizando, são apresentados alguns casos de testes que irão verificar a corretude do *software* analisado e demonstrar a eficácia dos casos de testes de unidade que foram produzidos.
Palavras-chave: Engenharia de *Software*, Processamento e Análise de Imagens, Testes de Unidade.

Abstract

The aim of this paper is to describe some applications of cases of unit tests that had been developed during the cycle of development of software. In this work, JUnit was used to perform tests of unit in a software for image processing and analysis, which is being developed in Java. Finally, a test to verify the correctness of the analyzed software was presented, whose results demonstrates the efficiency of the cases of unit tests that have been developed.
Key-words: Software Engineering, Image Analysis and Processing, Tests of Unit.

1 Introdução

A indústria do *software* vem experimentando um grande crescimento nas últimas décadas. Hoje, o *software* está presente na vida de praticamente todas as pessoas, seja através do uso de computadores, sistemas de automação comercial e industrial ou *software* embutido em eletrodomésticos e telefones celulares, entre outros.

As principais conseqüências desse crescimento são o aumento da complexidade do *software* e as exigências cada vez maiores do mercado [2]. É exigido das empresas de *software* que os sistemas sejam desenvolvidos com prazo e custo determinados e obedeçam a padrões de qualidade.

Segundo [3], a garantia da qualidade do *software* é a principal atividade com a qual uma equipe de desenvolvimento deve se preocupar para entregar um produto confiável a seus usuários. Essa atividade deve ser conduzida em todas as fases do processo de *software* para que os defeitos introduzidos sejam eliminados na fase em que surgiram, a fim de evitar o aumento dos custos da remoção desses em fases posteriores a sua criação.

Dentre as atividades de garantia de qualidade de *software* estão os testes. Os testes de *software,* segundo [4], consistem na execução do produto de *software* visando verificar a presença de erros e aumentar a confiança de que tal produto esteja correto.

Conforme [5], a atividade de teste é o processo de executar um programa com a intenção de encontrar um erro: um bom caso de teste é aquele que tem alta probabilidade de revelar a presença de erros e um teste bem sucedido é aquele que detecta a presença de erros ainda não descobertos. Por outro lado, o custo para correção de um erro na fase de manutenção é de sessenta a cem vezes maior que o custo para corrigi-lo durante o desenvolvimento [6].

Um dos objetivos de testar programas é maximizar a sua abrangência, isto é, permitir que o teste alcance um maior número possível de defeitos, deixando, assim, o programa com um melhor desempenho. Outro objetivo é eliminar esforços duplicados, visto que no momento que se cria um

programa, já deve ser elaborado o seu teste, diminuindo com isso a probabilidade de problemas futuros, principalmente se houver a necessidade *refactoring* de código.

A organização da atividade de teste deve, de alguma forma, poder refletir a organização da atividade do projeto. Isto implica que a arquitetura modular do *software* seja uma candidata natural a dirigir a verificação da aplicação. Neste contexto, procurou-se realizar casos de testes de unidade de *software* no decorrer do desenvolvimento do mesmo, utilizando o *framework* JUnit.

Neste artigo, são descritos casos de testes de unidade utilizando o *framework* JUnit. Fazendo uso dessa tecnologia, serão apresentados casos de testes realizados em um componente de *software* para Processamento e Análise de Imagens.

Este artigo está organizado da seguinte maneira: a seção 2 aborda o *framework* JUnit como ferramenta de testes de unidade; a seção 3 apresenta uma aplicação utilizando o *framework* Junit; por fim, na seção 4, são apresentadas as conclusões.

2 Framework JUnit

O JUnit é um *framework* horizontal de código aberto, desenvolvido por Kent Beck e Erick Gamma, com suporte à criação de testes automatizados em Java. Esse *framework* facilita a criação de código para a automação de testes com apresentação dos resultados. Com ele, pode ser verificado se cada método de uma classe funciona da forma esperada, exibindo possíveis erros ou falhas, podendo ser utilizado tanto para a execução de baterias de testes como para extensão.

Uma vez que todas as condições testadas obtiveram o resultado esperado após a execução do teste, considera-se que a classe ou método está de acordo com a especificação, incrementando a qualidade daquela unidade.

Com JUnit, o programador tem uma ferramenta muito poderosa que o ajudará a eliminar todos (ou quase todos) os *bugs* de seu código de maneira mais atraente. Ou seja, os programadores gostam de programar, então se criou uma forma interessante de realizar testes onde é possível a criação de programas que realizem os testes pelo programador. É utilizando esse conceito que JUnit permite deixar a fase de teste de unidade bem mais agradável ao programador.

Para utilizar o JUnit, é necessário criar uma classe que estenda `junit.framework.TestCase`. A partir daí, para cada método a ser testado é necessário definir métodos, os quais devem ser públicos e sem retorno de argumentos. Métodos testeXxx() na classe de teste devem ser `public void` e não podem receber nenhum parâmetro. Eles criam um objeto definindo o ambiente de teste, executam esses testes utilizando o ambiente que foi criado e verificam os resultados que podem terminar, falhar ou provocar uma exceção.

A Figura 1 apresenta o diagrama das principais classes da API para construir os testes.

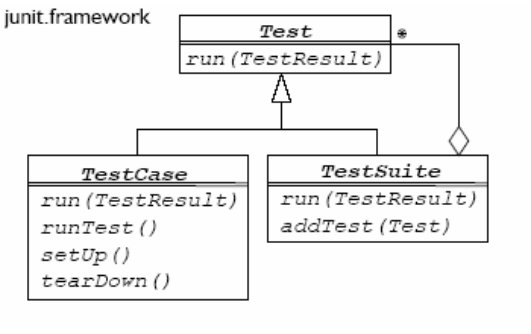


Figura 1 – Principais classes da API.

Segundo [1], estas são algumas vantagens de se utilizar JUnit:

- Permite a criação rápida de código de teste enquanto possibilita um aumento na qualidade do sistema sendo desenvolvido e testado;
- Não é necessário escrever o próprio *framework*;
- Amplamente utilizado pelos desenvolvedores da comunidade código-aberto, possuindo um grande número de exemplos;
- JUnit é elegante e simples. Quando testar um programa se torna algo complexo e demorado, então não existe motivação para o programador fazê-lo;
- Uma vez escritos, os testes são executados rapidamente sem que, para isso, seja interrompido o processo de desenvolvimento;
- JUnit verifica os resultados dos testes e fornece uma resposta imediata;
- Pode-se criar uma hierarquia de testes que permitirá testar apenas uma parte do sistema ou todo ele;
- Escrever testes com JUnit permite que o programador perca menos tempo depurando seu código;
- Todos os testes criados utilizando o JUnit são escritos em Java;
- JUnit é LIVRE.

O JUnit possui uma grande integração com outras ferramentas de desenvolvimento, como Jbuilder, Kawa, Jdeveloper, entre outros. Além disto, foram projetados as extensões do JUnit voltados para diversos segmentos como banco de dados, XML, J2EE e WEB.

3 StoneAge: Uma Aplicação

A ferramenta **StoneAge** foi inicialmente desenvolvida[1] dentro do Grupo de Processamento de Informações Multimídia - PIGS da Universidade

[1] Desenvolvida por Daniel Welfer [Welfer, 2005], Gabrielle Dias Freitas [Freitas, 2004], Grasiela Peccini [Peccini, 2004] e Marcos Cordeiro d'Ornellas.

Federal de Santa Maria, e provê funcionalidades para processamento e análise de imagens. Essa ferramenta foi implementada utilizando a linguagem de programação Java, a *Application Programming Interface (API) Swing*[2] e *Java Advanced Image*[3], em conjunto com o desenvolvimento baseado em componentes e padrões de projeto.

O grupo PIGS tem como objetivo produzir e implementar métodos de processamento e análise de imagens em microscopia quantitativa e anatomopatologia, contemplando as seguintes áreas de atuação: processamento e filtragem de imagens para melhoramento na qualidade da análise, classificação automática de imagens indexadas por conteúdo, desenvolvimento de software para processamento de imagens e imageamento médico.

Atualmente, o grupo está desenvolvendo a ferramenta **StoneAge**, que utiliza testes de unidade como forma de garantir a qualidade do que está sendo desenvolvido.

A ferramenta **StoneAge** possui duas classes principais que são a `Main` e a `LittleWindow`. A classe `Main` (classe principal do sistema) representa a classe mãe onde é efetuada a abertura dos arquivos gráficos, as informações de ajuda do sistema e outras. É a classe principal por gerenciar a chamada aos pacotes os quais os componentes estão inseridos. Toda vez que um arquivo gráfico é aberto essa classe invoca um objeto da classe `LittleWindow` que tem por finalidade exibir essa imagem na tela e proporcionar todos as operações até então desenvolvidas [7].

A classe `LittleWindow` (unidade principal da interface com o usuário) tem por finalidade exibir a imagem na tela e proporcionar todos as operações até então desenvolvidas. Assim, esta classe comporta-se como um processo filho capaz de apresentar diferentes comportamentos [8].

O presente artigo explora casos de testes de unidade realizados nas classes `Main`, `LittleWindow` e `ShowImage` da ferramenta **StoneAge** utilizando *framework* JUnit. O JUnit facilita a criação automática de testes, com apresentação dos resultados. Pode-se verificar se cada método de uma classe funciona de forma esperada, exibindo uma possível falha.

Dentre as funcionalidades existentes na classe `Main`, realizou-se testes de unidade na função `Open`, a qual é responsável por abrir um arquivo gráfico. Posteriormente, realizou-se teste na função `SaveAs` da classe `LittleWindow` que é responsável por salvar um arquivo gráfico e, por fim, realizou-se testes na classe `ShowImage`, a qual é responsável por exibir uma janela interna do tipo `LittleWindow`, que mostra um arquivo gráfico aberto.

[2]Java Foundation Classes (JFC/Swing): http://java.sun.com/products/jfc.

[3] Página Oficial de *Java Advanced Image:* http://java.sun.com/products/java-media/jai.

Na função `Open` da classe `Main` testou-se a chamada de abertura de um objeto gráfico (imagem) escolhido e, também as extensões desse arquivo aceitas pela ferramenta **StoneAge**. Na Figura 2 pode-se visualizar a função `Open` da classe `Main`, a qual realizou-se os testes.

```
public void Open(){
     PlanarImage src;
     if ( getOpenNumber() == 0 )
     {
      jfilechooser = new JFileChooser( "C:/WINDOWS/Desktop" );
     }
     else
     { String ultimoCaminho = (String)
          ultimoPath.elementAt( getOpenNumber()-1 );
       jfilechooser = new JFileChooser( ultimoCaminho );
     }
     jfilechooser.setFileSelectionMode
          ( JFileChooser.FILES_ONLY );

     ExampleFileFilter filterOpen = new ExampleFileFilter();
     filterOpen.addExtension( "JPG" );
     filterOpen.addExtension( "JPEG" );
     filterOpen.addExtension( "BMP" );
     filterOpen.addExtension( "PNG" );
     filterOpen.addExtension( "GIF" );
     filterOpen.addExtension( "TIFF" );
     filterOpen.addExtension( "PNM" );

     jfilechooser.setFileFilter( filterOpen );

     result = jfilechooser.showOpenDialog( this );

     if ( result == JFileChooser.APROVE_OPTION )
     {
     this.filename = jfilechooser.getSelectedFile();
     String name = jfilechooser.getName( this.filename );
     SetImageName( name );

     File imagePath = jfilechooser.getSelectedFile();
     String path = imagePath.getPath();
     setImagePath( path );
     setImageExtension( filterOpen.getExtension( filename ));
     src = JAI.create( "fileload",getImagePath() );
     setSrc( src );

     new ShowImage( src,getImagePath(),desktop );
     setOpenNumber( getOpenNumber() + 1 );
     ultimoPath.add( path );
     }
     else
          jfilechooser.cancelSelection();
     }
```

Figura 2 – Função `Open`.

A função `Open` instancia um objeto do tipo `JFileChooser`, que é a janela interna que permite localizar e abrir um arquivo gráfico (imagem). Após foi criado um construtor do tipo `ExempleFileFilter`, o qual é uma classe da *Sun* que é utilizada para filtrar a extensão dos arquivos que serão abertos. As extensões permitidas para abertura de um arquivo pela ferramenta **StoneAge** são JPG, JPEG, BMP, PNG, GIF, TIFF e PNM. Por fim, é passada a classe `ShowImage`.

Os testes de unidade realizados nos métodos da função `Open`, Figura 2, tiveram como objetivo verificar se o arquivo está sendo aberto corretamente e suas possíveis extensões. Neste caso, para desenvolver os testes com o JUnit, foi necessário criar uma classe que estenda `junit.framework.TestCase`, para em seguida definir um método `public void testImagemAberta()` na classe de teste.

A Figura 3 mostra os casos de teste de unidade realizados para os métodos da Figura 2.

```
public void testImagemAberta(){
     main.open();
     try{
     System.out.println ("Extensão" +
                main.getImageExtension());
     } catch( Exception e) {}
     extension = main.getImageExtension();

     if((extension.equals("jpg")) ||
       ((extension.equals("JPG")) ||
       ((extension.equals("jpeg")) ||
       ((extension.equals("JPEG")))
       assertEquals("2.0", "jpg",extension);
     else
     if((extension.equals("bmp")) ||
       ((extension.equals("BMP"))
       assertEquals("3.0", "bmp",extension);
     else
     if((extension.equals("tiff")) ||
       ((extension.equals("TIFF"))
        assertEquals("4.0", "tiff",extension);
     else
     if((extension.equals("png")) ||
       ((extension.equals("PNG")) ||
        assertEquals("5.0", "png",extension);
     else
     if((extension.equals("pnm")) ||
       ((extension.equals("PNM"))
       assertEquals("6.0", "pnm",extension);
     else
     if((extension.equals("gif")) ||
       ((extension.equals("GIF"))
       assertEquals("7.0", "gif",extension);
     else
     if((extension != "jpg") || (extension != "JPG") ||
        (extension != "jpeg") || (extension != "JPEG") ||
        (extension != "bmp") || (extension != "BMP") ||
        (extension != "tiff") || (extension != "TIFF") ||
        (extension != "png") || (extension != "PNG") ||
        (extension != "gif") || (extension != "GIF"))
         assertFalse(extension, true);
}
```

Figura 3 – Casos de testes realizados para a função `Open`.

No caso de teste `testImagemAberta` iniciou-se o teste chamando a função `Open`, o qual abre uma caixa de diálogo para escolha do arquivo que será aberto. Posteriormente a escolha do arquivo, é feito o teste para ver se a extensão do arquivo escolhido pelo usuário é permitida pela ferramenta **StoneAge**.

Para verificar se a extensão do arquivo aberto é igual à extensão permitida fez-se uso do método `assertEquals()`, o qual tem como objetivo comparar dois valores (o valor esperado e o valor recebido).

Na função `SaveAs` da classe `LittleWindow` testou-se as extensões do arquivo gráfico salvo aceitas pela ferramenta **StoneAge**. Na figura 4 pode-se visualizar a função `SaveAs` da classe `LittleWindow`, onde foram realizados os testes.

```
public void saveAs(PlanarImage src)
{
   String  saveName;
   JFileChooser salvaArquivoAs =
               new JFileChooser();
   String extension;
   int i = pathGlobal.indexOf(".");
   extension = pathGlobal.substring(i+1).trim();
   ExampleFileFilter filterSaveAsPNM =
               new ExampleFileFilter();
   filterSaveAsPNM.addExtension("PNM");
   salvaArquivoAs.setFileFilter
              (filterSaveAsPNM);
   ExampleFileFilter filterSaveAsJPG =
               new ExampleFileFilter();
   filterSaveAsJPG.addExtension("JPG");
   SalvaArquivoAs.setFileFilter
               (filterSaveAsJPG);
   ExampleFileFilter filterSaveAsBMP =
               new ExampleFileFilter();
   filterSaveAsBMP.addExtension("BMP");
   salvaArquivoAs.setFileFilter
               (filterSaveAsBMP);
   ExampleFileFilter filterSaveAsPNG =
               new ExampleFileFilter();
   filterSaveAsPNG.addExtension("PNG");
   salvaArquivoAs.setFileFilter
               (filterSaveAsPNG);
   ExampleFileFilter filterSaveAsTIFF =
               new ExampleFileFilter();
   filterSaveAsTIFF.addExtension("TIFF");
   salvaArquivoAs.setFileFilter
               (filterSaveAsTIFF);
   salvaArquivoAs.setDialogType
               (JFileChooser.SAVE_DIALOG);
   JFrame frameMensSalvar = new JFrame();
   Dialogo caixaDialogo= new
               Dialogo(frameMensSalvar);
   int result =
         salvaArquivoAs.showSaveDialog(this);
   if (result == JFileChooser.APPROVE_OPTION)
   {
   savefilename =
             salvaArquivoAs.getSelectedFile();
   System.out.println("Gravando imagem em: " +
             savefilename+"."+extension);
```

```
    pathGlobal = returnFilter.getDescription();
    i = pathGlobal.indexOf("(");
    extension =
   pathGlobal.substring(i+1).trim();
    int size=extension.length();
    extension = extension.substring(0, size-1);
    saveName = savefilename+extension;
    System.out.println("saveNome "+saveName);
    setImageExtension( extension );
    if (  (extension.equals(".gif")) ||
        (extension.equals(".GIF")) ) {
         System.out.println("Formato gif não
          disponível.");
    }
    if (  (extension.equals(".jpg"))||
       (extension.equals(".JPG"))||
       (extension.equals(".jpeg"))||
       (extension.equals(".JPEG") )  ){
          strategy.choice(1,src, saveName);
    }
    if (  (extension.equals(".bmp")) ||
       (extension.equals(".BMP")) ){
           strategy.choice(2,src, saveName);
  }
  if ((extension.equals(".png")) ||
    (extension.equals(".PNG"))){
           strategy.choice(3,src, saveName);
  }
  if ((extension.equals(".tiff")) ||
   (extension.equals(".TIFF"))){
            strategy.choice(4,src, saveName);
  }
  if ((extension.equals(".pnm")) ||
     (extension.equals(".PNM"))){
           strategy.choice(5,src, saveName);
  }
 }
}
```

Figura 4 – Função `SaveAs`.

A função `SaveAs` é responsável por salvar um arquivo gráfico no diretório escolhido pelo usuário. Bem como na função `Open`, essa função também instancia um objeto do tipo `JFileChooser`, e após é criado um construtor do tipo `ExemploFileFilter` para filtrar a extensão dos arquivos que serão salvos. As extensões permitidas para salvar um arquivo gráfico são JPG, JPEG, BMP, PNG, GIF, TIFF e PNM.

Os testes de unidade realizados nos métodos da função `SaveAs`, Figura 4, tiveram como objetivo verificar se o arquivo está sendo salvo corretamente com suas possíveis extensões. Neste caso, primeiramente foi chamada a função `Open` para abrir um arquivo gráfico existente e, após, foi chamada a função `SaveAs` para permitir realizar os testes para salvar o arquivo corretamente.

A Figura 5 mostra os casos de teste de unidade realizados para os métodos da Figura 4.

```
public void testImageReturn() {
   main.open();
   try{
      littleWindow = new LittleWindow
          ( main.getImagePath(),
           main.getSrc(), desk );
      littleWindow.saveAs(main.getSrc());
   }
   catch( Exception e ){
   System.out.println( "Erro + " +   e.getMessage());
   }
   extension = littleWindow.getImageExtension();
         System.out.println( "Extensão Retorna:  " + extension );
  if ((extension.equals(".jpg")) || (extension.equals(".JPG"))||
     (extension.equals(".jpeg")) || (extension.equals(".JPEG")))
      assertEquals( "2.0", ".jpg", extension );
  else
  if ((extension.equals(".bmp")) || (extension.equals(".BMP")))
     assertEquals( "3.0", ".bmp", extension );
  else
  if ((extension.equals(".tiff")) || (extension.equals(".TIFF")))
     assertEquals( "4.0", ".tiff", extension );
  else
  if ((extension.equals(".png")) ||  (extension.equals(".PNG")))
     assertEquals( "5.0", ".png", extension );
  else
  if ((extension.equals(".pnm")) || (extension.equals(".PNM")))
     assertEquals( "6.0", ".pnm", extension );
  else
  if ((extension.equals(".gif")) || (extension.equals(".GIF")))
     assertEquals( "7.0", ".gif", extension );
  else
  if ( (extension != ".jpg")  || (extension != ".JPG")  ||
       (extension != ".jpeg") || (extension != ".JPEG") ||
       (extension != ".bmp")  || (extension != ".BMP")  ||
       (extension != ".tiff") || (extension != ".TIFF") ||
       (extension != ".png")  || (extension != ".PNG")  ||
       (extension != ".gif")  || (extension != ".GIF"))
            assertFalse(extension, true);
  }
}
```

Figura 5 – Casos de testes da função `SaveAs`.

No caso de teste `testImageReturn` iniciou-se o teste chamando a função `Open`, o qual abre uma caixa de diálogo para escolha do arquivo que será aberto. Posteriormente a escolha do arquivo, é aberta uma nova caixa de diálogo onde o usuário deve escolher o diretório, a extensão e o nome do arquivo para posteriormente salvar o arquivo gráfico e então testar se a extensão escolhida pelo usuário é permitida pela ferramenta **StoneAge**. Para verificar se a extensão do arquivo salvo é igual à extensão permitida fez-se uso do método `assertEquals()`.

Na classe `ShowImage` realizou-se testes de unidade para verificar se os métodos e variáveis responsáveis por abrir uma janela interna estão recebendo os valores atribuídos. A Figura 6, mostra a classe `ShowImage`, a qual se realizou os testes.

```
public ShowImage(PlanarImage src, String imagePath,
     JDesktopPane desktop) {
     setShowImage( src );
     sourcesVector.addSources( src );
     int svs = sourcesVector.getVSources().size();

     try{
        LittleWindow littleWindow = LittleWindow(imagePath,
                                              dst, desktop);
        desktop.add(littleWindow);
        littleWindow.setLocation(10 * svs, 10 * svs);
        littleWindow.setSelected( true );
        }catch(Exception ex){
        JOptionPane.showMessageDialog( null, ex.getMessage(),
               "Try Again", JOptionPane.INFORMATION_MESSAGE),
        }
     }
     public void setShowImage( PlanarImage dst){
         this.dst = dst;
     }
     public PlanarImage getShowImage( ){
         return dst;
     }
}
```

Figura 6 – Classe `ShowImage`.

Esse método é responsável por exibir uma janela interna, do tipo `LittleWindow`, onde passa três parâmetros. O primeiro é uma `planarImage`, o segundo é uma `string` que mostra o caminho do arquivo e, o terceiro é o `JDesktopPane` a ser instanciado. A Figura 7 mostra os casos de testes de unidade realizados para a classe `ShowImage`.

```
     public void testImageReturn() {
          main.open();
          showImage.setShowImage(main.getSrc());
          try{
               main.getSrc();
          } catch( Exception e) { }

          assertNotNull(showImage.dst);
          assertNotNull(showImage.getShowImage());
     }
```

Figura 7 – Casos de testes da função `ShowImage`.

Os testes realizados na classe `ShowImage` tiveram como objetivo verificar se o método e a variável está recebendo e mostrando o arquivo aberto. Para realizar esse teste fez-se uso do método `AssertNotNull()`, cujo objetivo é verificar se a referência ao objeto passado não é nula.

Posteriormente criou-se uma "classe de teste mãe", chamada `Suite`, a qual representa uma composição de testes, cuja finalidade é acionar várias classes de teste em um único lugar, sendo usado pelo `TestRunner` para saber quais métodos devem ser executados como teste. A Figura 8 mostra um exemplo de unidade de teste "mãe".

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class TestaTudo {
     public static void main ( String [] args )
     {
          junit.textui.TestRunner.run( suite() );
     }
     public static Test suite()
     {
          TestSuite suite = new TestSuite();
          suite.addTest( TestMain.suite() );
          suite.addTest( TestImageReturn() );
          suite.addTest( TestShowImage.suite() );
          return suite;
     }
}
```

Figura 8 – Exemplo de unidade de teste "mãe".

Dessa maneira é possível construir uma hierarquia de testes onde uma `Suite` chama a outra e assim por diante. Na Figura 9, é apresentado o resultado com sucesso da classe "mãe" `TestaTudo`.

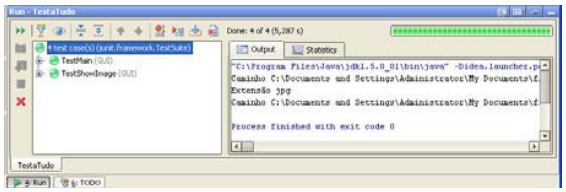


Figura 9 - Caso de teste com sucesso.

Na Figura 9, foi apresentado o caso de teste `TestaTudo`, o qual se verificou através do modo gráfico que os testes `testImagemAberta()`, `testImageReturn()` e `testShowImage()` obtiveram sucesso, garantindo a qualidade das funções básicas da ferramenta.

Por conseguinte, vale lembrar que em caso de insucesso a ferramenta mostra uma barra vermelha. Na Figura 10, é apresentado um caso de teste mal sucedido.

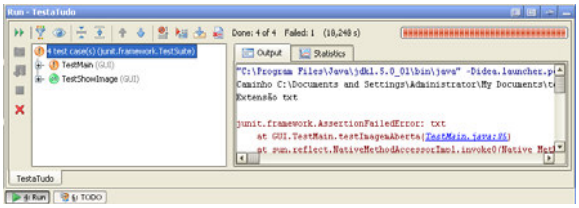


Figura 10 - Caso de teste mal sucedido

No caso de teste apresentado na Figura 10, constatou-se que a classe de teste `TestMain` teve um erro na função `testImagemAberta`, tendo em vista que a mesma recebeu um arquivo com formato "`txt`" sendo que essa extensão não é suportada pela ferramenta **StoneAge**, a qual só aceita extensões do tipo gráfico.

4 Conclusão

O objetivo principal do teste de software é aumentar a confiabilidade e garantir a qualidade do software para que este seja liberado com o mínimo de erros possível. Esses testes devem ser simples e eficientes dando ênfase ao *design* simples para resolução do problema, permitindo sempre que necessário escrever novos testes. A realização de testes de *software* permite a criação de códigos bem mais confiáveis e estáveis em um tempo de desenvolvimento bem menor, pois os usuários estão cada vez mais informados e principalmente críticos, quanto à qualidade dos sistemas adquiridos.

No âmbito de Processamento e Análise de Imagens, apesar de ainda não haver referências específicas para essa área, notou-se que os testes são de vital importância para garantir a qualidade da ferramenta desenvolvida, além de garantir a confiabilidade para os usuários que utilizarão as funções que a ferramenta **StoneAge** disponibiliza.

Neste contexto, o *framework* JUnit mostrou-se um ótimo recurso para testes de unidade, principalmente por prover a separação do código de teste do código do produto, facilitando sua integração com a maioria das ferramentas de desenvolvimento Java, bem como extensões para vários segmentos. É de fácil utilização permitindo visualizar os testes em formato texto, GUIs AWT e Swing, permitindo ainda a extensão do *framework*.

Referências

[1] CLARCK, M. JUnit Primer. Web Journal, 2005.

[2] COELHO, C. C. Maps: Um Modelo de Adaptação de Processo de Software. Dissertação, CI/UFP, Recife, 2003.

[3] FUGGETA, A. Software Process: a Roadmap. In: 22nd International Conference on the Future of Software engineering. Limerick, Ireland: ACM, 2000, p.25-34.

[4] HARROLD, M. J. Testing: a roadmap. In: ICSE'2000, 22nd International Conference on Software Engineering. The Future of Software Engineering, New York, NY, USA: ACM Press, 2000.

[5] MYERS, G. J. The Art of Testing. Addison Wesley Publishing Company, 2004, 2ªed.

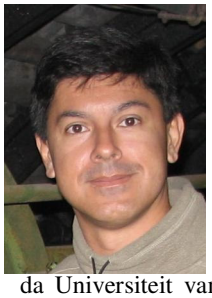
[6] PRESSMAN, R. Engenharia de Software: Uma Abordagem Prática. McGraw-Hill, 2002, 5ªed., RJ.

[7] WELFER, D. Padrões de Projeto no Desenvolvimento de Sistemas de Processamento de Imagens. Dissertação de mestrado, Universidade Federal de Santa Maria, RS, 2005.

[8] FREITAS, G. D. IgraPI: Interface Gráfica da Ferramenta para Processamento de Imagens StoneAge. Monografia, Universidade Federal de Santa Maria, RS, 2004.

Sabrina Borba Dalcin é mestranda do Programa de Pós-Graduação em Engenharia de Produção, subárea Tecnologia da Informação, da Universidade Federal de Santa Maria – UFSM, RS. Obteve o título de Bacharel em Sistemas de Informação em 2002 pelo Centro Universitário Franciscano – UNIFRA, Santa Maria, RS. Integrante do Grupo de Processamento de Informação Multimídia (PIGS). Suas áreas de interesse são: Engenharia de Software e Testes.

Adriane P. Dias Ferreira é mestranda do Programa de Pós-Graduação em Engenharia de Produção, subárea Tecnologia da Informação, da Universidade Federal de Santa Maria – UFSM, RS. Obteve o título de Bacharel em Ciência da Computação em 2003 pela Universidade de Cruz Alta – UNICRUZ, Cruz Alta, RS. Integrante do Grupo de Processamento de Informação Multimídia (PIGS). Sua área de interesse é Engenharia de Software.



Marcos Cordeiro d'Ornellas é Engenheiro e Mestre em Engenharia Eletrônica e Computação pelo Instituto Tecnológico de Aeronáutica e Doutor em Informática na área de processamento de imagens, pelo Intelligent Systems Laboratory Amsterdam (ISLA) da Universiteit van Amsterdam (UvA). Suas áreas de interesse são: processamento e análise de imagens, engenharia de software e segurança na Internet.