UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA DE PRODUÇÃO

IMAGETEST: UMA PROPOSTA DE METODOLOGIA PARA APLICAÇÃO DE TESTES DE SOFTWARE NO DOMÍNIO DE TRANSFORMAÇÕES NO ESPAÇO DE CORES

DISSERTAÇÃO DE MESTRADO

Sabrina Borba Dalcin

Santa Maria, RS, Brasil

2007

IMAGETEST: UMA PROPOSTA DE METODOLOGIA PARA APLICAÇÃO DE TESTES DE SOFTWARE NO DOMÍNIO DE TRANSFORMAÇÕES NO ESPAÇO DE CORES

por

Sabrina Borba Dalcin

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-graduação em Engenharia da Produção, Área de Concentração em Tecnologia da Informação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Engenharia da Produção**

Orientador: Prof. Marcos Cordeiro d'Ornellas, PhD.

Santa Maria, RS, Brasil

2007

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Engenharia da Produção

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

IMAGETEST: UMA PROPOSTA DE METODOLOGIA PARA APLICAÇÃO DE TESTES DE SOFTWARE NO DOMÍNIO DE TRANSFORMAÇÕES NO ESPAÇO DE CORES

elaborada por
Sabrina Borba Dalcin

como requisito parcial para obtenção do grau de
Mestre em Engenharia da Produção

COMISÃO EXAMINADORA:

Marcos Cordeiro d'Ornellas, PhD.
(Presidente/Orientador)

Cesar Tadeu Pozzer, Dr. (UFSM)

Lisandra Manzoni Fontoura, Dra. (URI)

Santa Maria, 24 de agosto de 2007.

Dedico essa dissertação
aos meus pais Reneu e
Maria Célia.

AGRADECIMENTOS

Agradeço inicialmente a Deus pelo dom da vida e por estar presente sempre.

A minha querida família, em especial ao meu pai Reneu e a minha mãe Maria Célia pelo amor, dedicação, paciência e incentivo em todos os momentos da minha vida. Vocês são meus exemplos de vida.

Ao meu irmão Tiago que mesmo de longe sempre me incentivou com suas palavras. Sem o apoio dessa família esse trabalho não teria se concretizado.

Ao meu orientador, Prof. Dr. Marcos d'Ornellas pela confiança, incentivo, amizade e profissionalismo nesses anos de orientação.

A minha co-orientadora, Prof. Dra. Maria Istela Cagnin, que aceitou me co-orientar. Obrigada pelo seu voto de confiança, pela sua orientação sempre me fornecendo informações que foram muito relevantes para a realização desse trabalho.

Aos meus colegas de laboratório Diego, Adri, Patrícia e Daniel pelo apoio, pelas conversas, almoços, cafés e principalmente pela amizade nesses anos de convívio. Lembrarei de vocês sempre com muito carinho.

Aos demais professores e colegas do grupo GPIM que sempre contribuíram de alguma maneira.

Aos meus amigos de longa data que também sempre se preocuparam em saber como estava o trabalho, sempre com palavras de incentivo.

Ao apoio do CNPq.

"É muito melhor arriscar coisas grandiosas, alcançar triunfo, glória, mesmo se expondo à derrota, do que formar fila com os pobres de espírito que nem gozam muito, nem sofrem muito, porque vivem numa penumbra cinzenta, que não conhece vitória nem derrota."
(Theodore Roosevelt)

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Engenharia da Produção
Universidade Federal de Santa Maria

IMAGETEST: UMA PROPOSTA DE METODOLOGIA PARA APLICAÇÃO DE TESTES DE SOFTWARE NO DOMÍNIO DE TRANSFORMAÇÕES NO ESPAÇO DE CORES

AUTORA: Sabrina Borba Dalcin
ORIENTADOR: Marcos Cordeiro d'Ornellas
Data e Local da Defesa: Santa Maria, 24 de agosto de 2007.

Este trabalho propõe o desenvolvimento da metodologia ImageTest, a qual é dedicada a aplicações que envolvem transformações no espaço de cores. Essa metodologia descreve o processo de desenvolvimento e aplicação de testes de software, desde a entrada de um participante na equipe de testes até a finalização da aplicação. Sendo assim, para melhorar a execução dos testes, a metodologia visa a utilização de duas técnicas existentes na literatura, a técnica estrutural e funcional. Para a devida aplicação das técnicas, são utilizados os critérios todos-nós, particionamento de equivalência e análise de valor limite. Por fim, são realizados os testes de aceitação, os quais auxiliam na obtenção da garantia de que o software satisfaz o cliente/usuário. Dessa forma, a ImageTest serve de auxílio ao desenvolvimento e aplicação dos casos de testes, visando o aumento da produtividade e qualidade do software, e a redução de manutenção.

Palavras-chave: Testes de Software, Engenharia de Software, Transformações no Espaço de Cores.

ABSTRACT

Master's Dissertation
Post-Graduate Program in Production Engineering
Federal University of Santa Maria

IMAGETEST: A METHODOLOGY PROPOSAL TO THE APPLICATION OF SOFTWARE TESTS IN THE DOMAIN OF THE TRANSFORMATIONS IN THE COLORS SPACE

AUTHOR: Sabrina Borba Dalcin
ADVISOR: Marcos Cordeiro d'Ornellas
Date and city: Santa Maria, 24th august 2007.

This work is about the development of the ImageTest methodology which is dedicaded to applications that envelop transformations in the color space. This methodology describes the development and applications process of software tests, since the entrance of a participant in the team tests until the final application. This way, to emprove the test performances, the methodology aims the use of two techniques, both of them are in the bibliography, the structural and functional techniques. It has been used to the applications techniques the criterions all-of-us, equivalence partitioning, and boundary value analysis. Finally, the acceptance tests are done, which help the warranty that the software pleases the customer/user. So, the ImageTest can help the develoment and application of the cases of tests, aiming at the productivity increase and software quality, and the maintenance decrease.

Key-words: Software Tests, Software Engineering, Transformations in Colors Space.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*(Aplication Programming Interface).* Interface de programação para construção de software.
CoFI	Abordagem de teste que emprega Conformidade e Falhas Injetáveis.
IEEE	*Institute of Electrical and Electronics Engineers.*
JAI	*(Java Advanced Image).* API Java para processamento de imagem.
OO	Orientação a objetos.
GPIM	Grupo de Processamento de Informação Multimídia.
UFSM	Universidade Federal de Santa Maria.

SUMÁRIO

1 INTRODUÇÃO

Sistemas de software estão se tornando cada vez mais importantes na sociedade moderna e progressivamente mais complexos, devido a grande concorrência do mercado e a elevada exigência por parte do cliente/usuário. A indústria passou a valorizar todas as fases que envolvem a elaboração do software e a posterior entrega, preocupando-se com seu capital intelectual e a comprometer-se com qualidade, a qual é fator de competitividade e se mostra cada vez mais presente no plano estratégico das organizações. Atualmente na área da informática, é extremamente difícil falar em desenvolvimento de software sem falar em qualidade, no uso de padrões de projeto e componentes, assim como testes de software.

O processo de desenvolvimento de software envolve uma série de atividades observando que, mesmo com o uso de métodos, técnicas e ferramentas, os produtos de software desenvolvidos ainda podem conter defeitos. Com isso, para que o produto de software atinja um grau de qualidade aceitável, são necessárias atividades de garantia de qualidade. As atividades de verificação, validação e teste (VV&T) vêm sendo utilizadas ao longo do processo de desenvolvimento com o intuito de minimizar esses erros e riscos (Maldonado et al., 2003). A atividade de teste tem como objetivo principal identificar e eliminar a presença de erros em um determinado software. De acordo com Harrold (2000), os testes de software consistem na execução do produto de software, visando verificar a presença de erros e aumentar a confiança de que tal produto esteja correto.

O teste de software é uma das fases do processo de engenharia de software que tem como objetivo aprimorar a produtividade e fornecer evidências da confiabilidade e da qualidade do software em complemento com outras atividades de garantia de qualidade. O teste de produto de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes (Myers, 1979), (Beizer, 1990), (Maldonado, 1991), (Pressman, 2001). Essas etapas devem ser realizadas ao longo do processo de desenvolvimento de software e se concretizam em três fases de teste: testes de unidade, de integração e testes de alto nível.

A atividade de teste de software é uma das atividades mais onerosas do processo de desenvolvimento de software, chegando a consumir 50% dos custos (Harrold, 2000). Essa atividade é considerada complexa, pois exige planejamento, projeto, execução, acompanhamento, integração com outras áreas e recursos como equipe, processo, treinamento e ferramentas adequadas. Por isso, é responsável pela maior porcentagem de esforço técnico no processo de desenvolvimento do software.

Sem uma infra-estrutura para a realização dos testes, torna-se impraticável a sua aplicação de forma adequada. Todavia, se a atividade de teste for executada como parte integrante do desenvolvimento de software, casos de teste podem ser criados nas diferentes fases do ciclo de vida para testar os produtos da própria fase e para serem usados na implementação do código (Inthurn, 2001).

Para tanto, vista como atividade que promove o aumento na qualidade do software, a realização de testes de acordo com uma metodologia torna-se cada vez mais freqüente na indústria de software. A atividade deve ser realizada segundo um processo bem definido que valorize a qualidade do projeto dos testes, independente do tipo de suporte possível: manual ou automático.

1.1 Motivação e Relevância

De acordo com Crespo (2004), a dificuldade em testar software é caracterizada por alguns pontos importantes como: o teste de software é um processo caro; existe uma falta de conhecimento sobre a relação custo/benefício do teste; existem dificuldades em implantar um processo de teste; há o desconhecimento de um procedimento e de técnicas de teste adequadas; há o desconhecimento sobre como planejar a atividade de teste; e finalmente, a preocupação com a atividade de teste ocorre somente na fase final do projeto.

Devido a esses fatores, a preparação e o desenvolvimento dos testes são feitos sem planejamento. É muito freqüente que os casos de teste sejam desenvolvidos de uma forma não estruturada e não sistemática, ou seja, são selecionados de forma aleatória ou *ad-hoc* (momentânea). Esta é uma realidade encontrada no desenvolvimento de softwares, onde comumente há recursos limitados e tempo escasso.

Atualmente a área de processamento e análise de imagens vem crescendo consideravelmente, sendo amplamente utilizada tanto no meio acadêmico e científico como em áreas de imageamento médico, na análise de imagens de satélite, no sensoriamento remoto, no domínio de transformações no espaço de cores, entre outras. Conforme destaca Gonzalez e Woods (2002), o uso de cores no processamento de imagens ocorre em função de dois fatores: (1) cores são poderosas para identificar e distinguir objetos, e (2) o sistema visual humano pode distinguir muito mais facilmente entre as muitas cores existentes do que entre os poucos níveis existentes em imagens monocromáticas.

Devido ao aumento da utilização das imagens digitais nas mais diversas áreas, os espaços de cores existem para tornar possível a exibição de imagens reais em um meio digital. Dessa forma, um modelo de cor pode ser definido como um sistema de coordenadas tridimensionais e um subespaço dentro deste sistema, onde cada cor é representada por um único ponto contido dentro deste subespaço (Gonzales; Woods, 2002).

Baseado neste crescimento considerável da utilização de imagens digitais, tem-se a necessidade de testar a qualidade dos softwares que fazem uso do domínio de transformações no espaço de cores. Para tanto, a atividade de teste deve ser organizada para que seja realizada de forma adequada.

Apesar do desenvolvimento de metodologias de teste de software estar sendo alvo de pesquisas recentes, percebe-se a carência de metodologia de teste no campo de processamento e análise de imagens, que descrevam detalhadamente todo o processo. Segundo Molinari (2003), existem modelos que descrevem os níveis de teste e diversas técnicas que podem ser usadas em conjunto para cobrir as necessidades de cada projeto. Entretanto, esses modelos apenas referenciam as técnicas e tipos de testes que podem ser utilizados de acordo com a metodologia de desenvolvimento de software adotada. Ou seja, esses modelos não especificam detalhes de participantes, atividades, documentos e outros fatores que envolvem uma metodologia de teste de software.

Percebe-se a existência de uma lacuna no campo de transformações no espaço de cores, o que se deve à carência de uma metodologia de teste que seja aplicada ao longo do ciclo de desenvolvimento do software. Dessa forma, esse

trabalho tenta preencher essa falta, contribuindo com o desenvolvimento de uma metodologia de testes que seja aplicada ao domínio de transformações no espaço de cores.

A metodologia proposta orienta o projeto de casos de teste no domínio de transformações no espaço de cores, obedecendo alguns critérios existentes na literatura, cujas contribuições são descritas a seguir.

1.2 Contribuições

A principal contribuição deste trabalho é a definição de uma metodologia de testes no domínio de transformações no espaço de cores, que orienta a equipe de testes a projetar casos de testes e aplicá-los de forma organizada, fazendo parte do ciclo de desenvolvimento do mesmo, promovendo a documentação e estando de acordo com as necessidades que envolvem esse domínio.

O desenvolvimento de software em aplicações no domínio de transformações no espaço de cores carece de uma metodologia para apoio as atividades de teste. Na metodologia proposta, o desenvolvimento dos testes parte de uma especificação textual baseada na norma IEEE 829 – 1998 (*Standard for Software Test Documentation*), onde os testes são aplicados de forma incremental ao longo do processo de desenvolvimento do software, apoiando o uso de ferramentas de domínio público e, finalizando a atividade com a emissão de relatórios.

Dessa forma, busca-se a melhoria de processos bem sucedidos de testes em aplicações de transformações no espaço de cores, propiciando o aumento da produtividade do software, redução no número de defeitos no produto visto que os algoritmos da área são bastante complexos, maior previsibilidade nos processos, além de garantir a qualidade das transformações realizadas na ferramenta desenvolvida.

1.3 Trabalhos relacionados

Conforme as pesquisas realizadas na literatura sobre assuntos relacionados à qualidade de software na área de processamento e análise de imagens, percebe-se que tanto a área de testes como a de imagens são áreas recentes, as quais ainda

não possuem muitos trabalhos referenciando o uso de metodologias de testes e suas aplicações. Pode-se citar alguns trabalhos encontrados com abordagens semelhantes:

- O trabalho de Crespo (2004), descreve o desenvolvimento de uma metodologia de teste de software no contexto de Melhoria de Processo, o qual é dedicada a implantação ou melhoria do processo de teste em empresas desenvolvedoras de software, que tem como finalidade viabilizar o uso das práticas de teste pelas empresas. Os testes, nesse trabalho, são formalizados utilizando a norma IEEE 829[1].

- Andrade (2005), propõe um método que utiliza a cobertura estrutural na seleção dos dados de entrada para a geração da curva ROC (*Receiver Operating Characteristic*) para avaliar os diagnósticos realizados com auxílio do computador (CAD), onde utiliza a técnica de teste estrutural para aperfeiçoar a avaliação de um sistema de auxílio ao diagnóstico. Esse método procura normalizar a escolha dos casos de teste através da análise de cobertura.

- Ambrosio (2005) descreve um processo e uma metodologia de teste de software visando reduzir custos e diminuir o número de incidentes em missões espaciais. A proposta de metodologia consta de um processo de teste baseado na IS-9646 que abriga uma metodologia para orientar o projeto de testes. Ambos, o processo e a metodologia são denominados CoFI, pois combinam teste de conformidade e validação por falhas injetáveis. A metodologia caracteriza-se por tratar falhas que imitam os problemas físicos causados pela radiação que afetam a comunicação computacional a bordo de satélites. Uma análise da eficiência da seqüência de teste CoFI relacionada ao critério de adequação de falhas de mutação em máquinas de estado foi realizada.

1.4 Estrutura da dissertação

Esta dissertação está organizada em seis capítulos. Neste capítulo, foram apresentados o contexto em que se insere o trabalho, a motivação para o seu desenvolvimento, trabalhos relacionados e suas contribuições. No Capítulo 2 é abordada uma introdução sobre a área de processamento e análise de imagens e o

[1] Disponível em: http://standards.ieee.org/reading/ieee/std_public/description/se/829-1983_desc.html

domínio de transformações no espaço de cores. No Capítulo 3 são abordados os principais tópicos relacionados aos fatores de qualidade de software. No Capítulo 4 são apresentados os conceitos sobre testes de software abordando suas técnicas e critérios, ferramentas e modelos de testes de software existentes. O Capítulo 5 apresenta a proposta de metodologia para aplicação de testes de software no domínio de transformação no espaço de cor – ImageTest, sendo abordada em detalhes suas etapas. O Capítulo 6 apresenta o estudo de caso da metodologia ImageTest em um projeto real de desenvolvimento de software, apresentando a realização dos testes de software de acordo com a metodologia proposta e os resultados alcançados. Finalmente, o Capítulo 7 apresenta as conclusões e perspectivas para os trabalhos futuros.

2 PROCESSAMENTO E ANÁLISE DE IMAGENS

Segundo Shapiro e Stockman (2000), a computação voltada para imagens existe há mais de trinta anos sendo que, no começo, restringia-se aos grandes centros e laboratórios de pesquisa devido aos caros recursos computacionais que se dispunha na época.

A área de processamento de imagens é certamente uma área em crescimento, onde diversos temas científicos são abordados e em alguns casos de caráter interdisciplinar. Conforme Albuquerque (2000), entre eles podemos citar: a compreensão de imagens, a análise em multi-resolução e em multi-frequência, a análise estatística, a codificação e a transmissão de imagens, entre outras.

Segundo Russ (1999), as operações de processamento de imagens consistem em melhorar a aparência visual da imagem para a visão humana e/ou preparar a imagem para análise e mensuração de características e estruturas presentes. O processamento de imagens parte da imagem (de uma informação inicial que é geralmente captada por uma câmera) ou de uma seqüência de imagens para obter a informação. Segundo Andrade (2005), existem três níveis da área de processamento de imagem: baixo, intermediário e alto nível.

O primeiro nível, ou seja o baixo nível, corresponde ao pré-processamento que é a etapa onde os defeitos e imperfeições da imagem são corrigidos. Existe um grande número de técnicas de pré-processamento como: realce da imagem, a qual tem como objetivo livrar a imagem de formações e distorções; a restauração, que melhora as propriedades visuais da imagem; a filtragem de ruído, que remove borrões e ruídos em uma imagem; e, a segmentação que divide a imagem em partes com a finalidade de detectar descontinuidades ou semelhanças.

Logo, o nível intermediário consiste em realizar um mapeamento do processamento realizado no baixo nível, para entidades significativas do próximo nível, na etapa de classificação. O propósito da extração de características é a redução da quantidade de dados ou redução da dimensionalidade dos dados, obtidos por meio da observação de certas características ou propriedades que distinguem os padrões de entrada (Gonzalez, 2000) (Facon, 2002).

A última etapa corresponde ao processamento de alto nível, onde fazem parte deste nível de processamento o reconhecimento de objetos e a interpretação da

cena. A tarefa de classificação consiste em associar rótulos a um dado objeto, com base na informação fornecida pelos seus descritores.

Russ (1999), destaca que a visão computacional e o imageamento médico são campos da computação de processamento e análise de imagens que também vem se destacando. Em visão computacional, o objetivo é possibilitar que as máquinas sejam capazes de processar e analisar as imagens obtidas. Um exemplo encontrado em Shapiro e Stockman (2001) é a aplicação do processamento de imagens na área é o do reconhecimento de padrões, ou seja, a identificação de imagens de acordo com características previamente definidas. Por conseguinte, no campo de imageamento médico, devido ao avanço da medicina e da tecnologia envolvida, o uso de técnicas de processamento de imagens tornou-se essencial tanto na aplicação de exames como na realização de processos mais complexos como uma cirurgia realizada via computador, permitindo a remoção de ruídos existentes na imagem evitando possíveis erros.

Contudo, para que uma imagem seja processada, Ima (2004) destaca que existe a necessidade que se disponha de um sistema adequado e eficiente para tal fim, onde há diversas ferramentas existentes no mercado que permitem a realização de tal processamento. Entretanto, a maioria das ferramentas representa um alto custo além de serem ferramentas específicas as quais não abrangem todas as operações que envolvem a área.

Processar uma imagem não é uma tarefa trivial. É extremamente importante conhecer alguns conceitos básicos utilizados na área, a fim de aplicar os métodos disponíveis na obtenção de melhores resultados.

2.1 O domínio de transformações no espaço de cores

O processamento digital de imagens atua sobre imagens digitais. Por isso, tem-se a necessidade de converter um campo de imagem contínuo para uma imagem digital equivalente (Pratt, 2001). Essa conversão ocorre através do mapeamento da imagem em uma matriz de *pixels* (*picture elements*). Cada *pixel* é composto por uma coordenada (x,y), que fornece a sua posição na imagem, e um modelo de cor, definido de acordo com o objetivo de uso da imagem.

Atualmente existem diversos modelos de cores, os quais são utilizados para facilitar e tornar possível a representação de cores no meio digital de várias formas de acordo com as necessidades. Cada modelo possui sua própria forma de representação das cores, e segundo Gonzales e Woods (2002), podem ser definidos como um sistema de coordenadas tridimensionais e um subespaço dentro deste sistema, onde cada cor é representada por um único ponto contido dentro deste subespaço.

Os três principais modelos de cores são o RGB, o CMY e o HSI. Tais modelos podem ser definidos em dois grandes grupos, de acordo com sua aplicabilidade. O primeiro grupo é composto pelos modelos utilizados em dispositivos de hardware, onde os modelos utilizados são categorizados como dependentes de dispositivo. Dessa forma, tem-se o modelo RGB (*red, green, blue*) que é utilizado em televisores e monitores de computadores, e o modelo CMYK (*cyan, magent, yellow, key*) utilizado em impressoras.

O segundo grupo é formado pelos modelos criados para utilização em softwares que visam a manipulação. Conforme Russ (1999), neste grupo estão os modelos que auxiliam nas tarefas de análise e processamento de imagens, por serem relacionados a percepção humana das cores. Alguns dos modelos foram criados com base na forma com que o sistema visual humano diferencia as cores, utilizando conceitos como a existência dos cones no olho humano, os variados comprimentos de onda que são percebidos, entre outros. Então, neste segundo grupo podem ser citados modelos como o modelo HSI e o modelo CIE L*a*b*.

Cabe lembrar que um espaço de cores é absoluto quando seu espaço possui uma única maneira de representar cada uma das cores que fazem parte do seu *gamut*[2]. Estes espaços possuem diversas especificações de como as cores devem ser visualizadas, o que garante que ele seja absoluto e o torna independente de fatores externos. Já, um espaço não absoluto é quando não possui uma única forma de representar suas cores e é dependente de fatores externos. Portanto, conversões entre espaços não absolutos e entre um espaço não absoluto e um absoluto podem ser feitas, porém normalmente não possuem muita precisão (as cores podem sofrer

[2] Refere-se ao conjunto de todas as cores que podem ser geradas através da combinação das coordenadas utilizadas pelo espaço de cores em questão.

alterações após a conversão) e, por vezes, podem ser úteis para análise visual por mais que não sejam absolutamente precisas (Icc, 2007).

2.1.1 Modelos de cores

Um modelo de cores é um modelo matemático que especifica como as cores podem ser representadas utilizando conjuntos de números, normalmente com 3 ou 4 valores. Quando se tem uma maneira de conectar este modelo com um espaço de cores absoluto através de uma função de mapeamento (função que delimita o *gamut* desse modelo de cores no espaço utilizado) faz-se deste modelo um espaço de cores. Através da Figura 2.1, extraída de Daronco (2007), pode-se observar um exemplo de uma mesma imagem sendo exibida em quatro diferentes espaços de cores.

Figura 2.1: Uma mesma imagem em quatro modelos diferentes: a) RGB b) CMY c) YIQ d) HSI

A seguir são mostrados alguns dos modelos de cores existentes na literatura.

2.1.1.1 RGB

O modelo RGB é composto de três coordenadas, R (*red*), G (*green*) e B (*blue*), que especificam o brilho dos sinais vermelho, verde e azul (as cores primárias da luz), respectivamente, cada um representado por um comprimento de onda definido. O RGB é um modelo aditivo, que combina vermelho, verde e azul aditivamente para formar uma cor resultante.

O RGB é um modelo muito utilizado principalmente por dispositivos com capacidade de emissão de luz, como monitores, televisores, câmeras de vídeo, como também é muito utilizado para o armazenamento de imagens digitais, o que facilita a tarefa de exibição das imagens. Segundo Russ (1999) o modelo CMY (*Cyan, Magenta, Yellow*) é o complemento do RGB, uma vez que as cores ciano, magenta e amarelo são consideradas complementos de vermelho, verde e azul, respectivamente.

O RGB é um modelo não absoluto e orientado à *hardware*. A exibição de uma mesma imagem em dois dispositivos pode resultar em cores aparentemente diferentes. Quando é necessária a conversão do modelo RGB para espaços absolutos como o CIE L*a*b*, por exemplo, é necessário que primeiro seja efetuada uma conversão do modelo RGB para algum espaço absoluto baseado neste modelo (Russ, 1999).

A seguir, têm-se as equações utilizadas para a conversão do modelo RGB para os espaços absolutos, onde (*r*, *g*, *b*) são os valores RGB convertidos, (*R*, *G*, *B*) são os valores RGB originais e *y* representa o *gamma* do espaço de cor escolhido (Lindbloom, 2007).

Para o espaço sRGB:

$$r = \begin{cases} R/12.92 & \text{se } R \leq 0.04045 \\ ((R + 0.055)/1.055)^{2.4} & \text{se } R > 0.04045 \end{cases}$$

$$g = \begin{cases} G/12.92 & \text{se } G \leq 0.04045 \\ ((G + 0.055)/1.055)^{2.4} & \text{se } G > 0.04045 \end{cases}$$

$$b = \begin{cases} B/12.92 & \text{se } B \leq 0.04045 \\ ((B + 0.055)/1.055)^{2.4} & \text{se } B > 0.04045 \end{cases}$$

Para outros espaços:

$$r = R^{\gamma}$$

$$g = G^{\gamma}$$

$$b = B^{\gamma}$$

2.1.1.2 HSI

O modelo HSI (*Hue, Saturation, Intensity*) baseia-se em três propriedades das cores: tonalidade, que é a cor descrita em comprimento de onda (por exemplo a diferença entre azul e verde); saturação, que é a quantidade de cor presente (diferença entre vermelho e rosa); e intensidade, que corresponde à quantidade de luz presente (imagem mais clara ou mais escura). Segundo Gonzales e Woods (2002), o homem não descreve a cor de um objeto através da quantidade de cada cor primária que ela possui (como no modelo RGB), mas sim através de seu matiz, saturação e brilho. Por isso esse modelo é considerado baseado na maneira como o homem descreve as cores.

Esse modelo é bastante utilizado para o processamento de imagens, pois permite que a tonalidade, a saturação e a intensidade sejam analisadas separadamente. Conforme Shapiro e Stockman (2001), as equações usadas na conversão do modelo RGB para o HSI são:

$$S = 1 - \frac{3}{(R + G + B)} [min(R, G, B)]$$

$$I = \frac{1}{3} (R + G + B)$$

$$H = \begin{cases} \theta & \text{se } B \leq G \\ 360 - \theta | & \text{se } B > G \end{cases}$$

onde: $$\theta = \cos^{-1}\left\{ \frac{\frac{1}{2}[(R-G)+(R-B)]}{[(R-G)^2+(R-B)(G-B)]^{\frac{1}{2}}} \right\}$$

2.1.1.3 CMY e CMYK

O CXMY (*Cyan, Magenta, Yellow*) é o complemento do RGB, onde o magenta é a cor complementar do verde, assim como ocorre entre o ciano e o vermelho, e entre o amarelo e o azul.

É um modelo subtrativo, isto é, as cores são subtraídas da luz branca para produzir a cor desejada (RUS, 1998). Da mesma forma que o modelo RGB, esse modelo também é não absoluto e orientado a hardware.

Os valores dos componentes do CMY variam conforme os valores RGB. Segundo Gonzales e Woods (2002), a equação utilizada para converter o modelo RGB para CMY é simples:

$$C = 1.0 - R$$

$$M = 1.0 - G$$

$$Y = 1.0 - B$$

2.1.2.3 YIQ e YUV

Os modelos YIQ e YUV são utilizados para transmitir imagens de televisão analógica. O componente Y de ambos os modelos possui o mesmo valor e corresponde à luminância da imagem, o valor do brilho monocromático desta imagem de acordo com a sensibilidade do olho humano a cada um dos componentes RGB. IQ e UV estão associados à cromaticidade, ou seja, às cores da imagem. As equações utilizadas para conversão de RGB para YIQ e para YUV são (Russ, 1999), (Colantoni, 2004), (Pratt, 2001), (Shapiro; Stockman, 2001):

$$Y = 0.29889531 * R + 0.58662247 * G + 0.11448223 * B$$

$$I = 0.59597799 * R - 0.27417610 * G - 0.32180189 * B$$

$$Q = 0.21147017 * R - 0.52261711 * G + 0.31114694 * B$$

$$Y = 0.30 * R + 0.59 * G + 0.11 * B$$

$$U = 0.493 * (B - Y)$$

$$V = 0.877 * (R - Y)$$

Além disso, deve-se considerar o tipo da imagem utilizada para as transformações. O tipo da imagem pode ser classificado em binária, níveis de cinza ou colorida, onde os valores atribuídos ao modelo de cor variam. Por exemplo, em uma imagem colorida, utilizando o modelo RGB. Cada componente pode assumir valores reais que variam de 0.0 a 1.0. Já em uma imagem em níveis de cinza, R = G = B, e os valores permitidos são os números inteiros contidos no intervalo [0, 255], em que o 0 corresponde a cor preta e o 255 equivale a cor branca (ou vice-versa), enquanto que os valores intermediários representam os níveis de cinza.

Além do modelo de cor, o *pixel* possui relações de vizinhança e a conectividade Conforme Meyer (1994) as relações de vizinhança mais utilizadas são a 4-vizinhança e a 8-vizinhança, em grade quadrada, e a 6-vizinhança em uma grade hexagonal.

Entre os modelos existentes, pode-se citar também o modelo YCbCr que é utilizado em sistemas de vídeo.

2.1.2 Espaços de cores

Os espaços de cores definidos pela CIE, os quais têm a característica comum de serem baseados na percepção humana das cores, nos valores dos *tristimulus,* e procuram representar todas as variações de cores que podem ser vistas pelo homem. Os *tristimulus* são representados pelos componentes X, Y e Z.

2.1.2.1 CIE XYZ

O espaço CIE XYZ é composto pelos três componentes primários X, Y e Z que foram definidos a partir de combinações dos valores vermelho, azul e verde. Segundo Lindblom (2007) essas combinações foram realizadas com o objetivo de definir um espaço com algumas características importantes, como: ser relacionado à percepção humana das cores, ser capaz de representar qualquer cor visível, ter os valores de seus componentes sempre positivos e separar a informação das cores da informação que representa a luminância (representada apenas pelo Y).

A conversão de RGB para o espaço CIE XYZ é muito mais complexa e envolve algumas operações a mais do que as simples conversões citadas anteriormente, onde maiores detalhes podem ser encontrado em Lindbloom (2007).

2.1.2.2 CIE xyY

O espaço CIE xyY é composto pelos três componentes utilizados no diagrama de cromaticidade CIE. Os valores x, y e Y são chamados de valores de cromaticidade e dependem somente da matiz ou do comprimento de onda dominante e da saturação, portanto são independentes da luminância (Hoffman, 2005b). Eles são calculados através das equações a seguir:

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

$$z = \frac{Z}{X + Y + Z}$$

Além dos espaços citados, existe também o espaço CIE L*a*b*, que é um espaço independente de dispositivo que foi diretamente baseado no CIE XYZ, onde suas cores aparentemente iguais são codificadas da mesma maneira. O espaço CIE

L*u*v* foi desenvolvido a partir dos espaços CIE L*a*b* e CIE U*V*W* e, assim como no CIE L*a*b*, separa-se a informação de cromaticidade nos componentes u* e v*, enquanto L* corresponde a *lightness*.

3 FATORES GERAIS DA QUALIDADE DE SOFTWARE

Nos últimos anos, a qualidade vem representando um conceito de extrema importância para a indústria em geral. Durante os anos 90, a melhoria da qualidade (e redução de custos) constituiu um dos principais desafios para a produção de software, fazendo com que os desenvolvedores de software procurassem seguir orientações de modelos, normas e padrões de qualidade de processo.

Conforme Rocha (2001), a qualidade de software pode ser vista como um conjunto de atributos que devem ser alcançados de forma que o produto atenda as necessidades de seus usuários. As diferenças entre os diversos domínios da aplicação e entre as tecnologias utilizadas na construção dos produtos determinam diferenças nos atributos a serem considerados na avaliação e, na importância destes atributos para o alcance da qualidade desejada.

Produzir software de qualidade é uma meta básica da engenharia de software, que é um termo utilizado para referir-se a modelos de ciclo de vida, metodologias de rotina, técnicas de estimativa de custo, estruturas de documentação, ferramentas de gerenciamento de configuração, técnicas de garantia de qualidade e outras técnicas de padronização da atividade de produção de software.

Logo, a qualidade de um produto tem um propósito: satisfazer o cliente (Koscianski, 2006). Portanto, a qualidade de um software depende de se decidir o que significa qualidade. Então é preciso adotar uma perspectiva gerencial e considerar diversos fatores que afetam a construção do produto e que influenciam no julgamento dos usuários, tais como: tamanho e complexidade do software sendo construído; número de pessoas envolvidas no projeto; ferramentas utilizadas; custos associados a existência de erros; custos associados a detecção e a remoção de erros.

Dessa forma, fazer uso das técnicas de engenharia de software em sistemas de processamento e análise de imagens, empregando de maneira correta boas metodologias pelos desenvolvedores, garante a qualidade do produto desenvolvido. Outro aspecto que contribui com a qualidade é o desenvolvimento de tecnologias e ferramentas, as quais vem crescendo cada vez mais na área de processamento de imagens. Por conseguinte, existem várias tarefas que podem ser automatizadas,

diminuindo a carga de trabalho das pessoas e, ao mesmo tempo, garantindo uniformidade: se for uma ferramenta que executa a tarefa, há menos chances do resultado ser diferente porque diversas pessoas a utilizaram.

3.1 Garantia de qualidade no domínio de transformações no espaço de cores

A Garantia de Qualidade de Software ou *SQA (Software Quality Assurance*) é uma atividade que é aplicada ao longo de todo o processo de engenharia de software em qualquer tipo de aplicação. No domínio de transformações no espaço de cores deve-se ter um cuidado especial com a escolha de vários itens importantes para a construção do software, pois essas escolhas são extremamente relevantes visto que tais aplicações utilizam bastantes recursos de memória. A garantia de qualidade abrange os seguintes fatores:

- métodos e ferramentas de análise, projeto, codificação e teste;
- revisões técnicas formais que são aplicadas durante cada fase da engenharia de software;
- uma estratégia de teste de múltiplas fases;
- controle da documentação do software e das mudanças feitas nela;
- um procedimento para garantir a adequação aos padrões de desenvolvimento de software, se eles forem aplicados;
- mecanismos de medição e divulgação.

Geralmente, a garantia de qualidade consiste daqueles procedimentos, técnicas e ferramentas aplicadas por profissionais para assegurar que um produto atinge ou excede padrões pré-especificados durante o ciclo de desenvolvimento do produto; se tais padrões são aplicados, a garantia de qualidade assegura que um produto atinge ou excede um nível de excelência (industrial ou comercial) mínimo aceitável.

Para garantir a qualidade de um sistema, primeiramente deve-se fazer a escolha de ferramentas que suportem o tratamento de imagens, assim como bibliotecas específicas para a construção da aplicação. Após, deve-se prevenir o software dos defeitos ou das deficiências, dessa forma aplicando testes de software desde as

fases iniciais do projeto, e sobretudo fazer com que o produto possa ter uma garantia de qualidade através de medições ou envio de relatórios sobre as atividades de testes. Conforme Molinari (2003) pode-se realmente gerenciar aquilo que se consegue medir, e vice-versa, e ainda algumas medidas de qualidade incluem: estruturação de um processo de desenvolvimento com métodos, técnicas e ferramentas.

Também, deve-se levar em consideração o uso de padrões de projeto para o desenvolvimento de aplicações no domínio de cores, pois de acordo com Gamma *et. al* (2000), um padrão de projeto nomeia, abstrai e identifica aspectos-chave de uma estrutura de projeto para torná-lo reutilizável. Sabe-se também da importância de ter a especificação dos requisitos dessas aplicações ao alcance para que os testes possam ter inicio nas primeiras fases do desenvolvimento fazendo parte do ciclo de vida do mesmo, visto que o domínio faz uso de muitos cálculos matemáticos;

Por fim, Molinari (2003) destaca que o gerenciamento de qualidade diminui os custos porque cedo um defeito será localizado e corrigido, e o custo de defeitos encontrados será menor ao longo do tempo. Portanto um investimento em termos de recursos, disponibilidade de tempo e mão de obra qualificada deve ser significativo no domínio de transformações no espaço de cores de modo a garantir a qualidade das aplicações desenvolvidas.

3.2 Fatores relevantes para a qualidade do software

Segundo Cortês (2001), a qualidade pode ser considerada sob diferentes aspectos obtendo-se várias definições para um software de qualidade. A melhoria na qualidade do software é obtida pela melhoria da qualidade dos processos envolvidos no seu desenvolvimento (Medeiros, 2001). Desta forma, a análise da qualidade de um software pode ser realizada tanto no processo de produção do software quanto no produto final.

Esta seção apresenta alguns dentre tantos fatores relevantes que contribuem com a qualidade do software.

3.2.1 Fatores Internos e Externos

Qualidade é um conceito multidimensional, realizado por um conjunto de atributos, representando vários aspectos relacionados ao produto: desenvolvimento, manutenção e uso. Com isso, percebe-se que podem existir vários observadores da qualidade de um produto: desenvolvedores e usuários (Pfleeger, 1991).

Diz-se que um software é de qualidade quando ele funciona adequadamente, não usa muita memória, é rápido, fácil de usar e satisfaz o usuário. Assim, a noção de qualidade de software pode ser descrita por um grupo de fatores. Esses fatores são classificados em dois tipos principais: *externos* e *internos*. Os fatores internos são percebidos apenas pelas pessoas que desenvolvem software, como, por exemplo, modularidade e legibilidade. Esses fatores são extremamente importantes tratando-se de aplicações no domínio de espaços de cores, pois são aplicações que utilizam muitas fórmulas matemáticas e fazem uso de algoritmos muito complexos, entre a utilização de outros recursos como bibliotecas próprias, que tornam muitas vezes aplicações difíceis de serem compreendidas por demais desenvolvedores.

A modularidade ajuda consideravelmente na compreensão do funcionamento do sistema, e a legibilidade é um fator relevante a medida que se procede a rotatividade de desenvolvedores. Dessa forma, esses dois fatores são de extrema relevância para que não aconteça a demanda de horas de compreensão sobre o que foi desenvolvido, ou até mesmo a perda do sistema por falta de entendimento.

Logo, os fatores externos são percebidos tanto pelas pessoas que desenvolvem software quanto pelos usuários. Por exemplo, confiabilidade, eficiência e facilidade de uso de um sistema são considerados fatores externos. Neste caso, principalmente os usuários do domínio de espaços de cores, que são pessoas especialistas no assunto, podem fazer uma boa avaliação sobre tais fatores. Também, considera-se que extensibilidade é um fator externo, dado que as pessoas (usuários) que encomendam um software podem notar se uma extensão requisitada pode ser facilmente realizada ou não.

3.2.2 Interface

O software tem como uma de suas características mais marcantes a usabilidade. A interação entre programa e usuário exerce influência determinante sobre a impressão de qualidade percebida. Embora outros fatores como precisão ou segurança possam ser de importância particular em uma determinada aplicação, problemas com o uso de um software devem ser tratados com atenção pelos desenvolvedores (Koscianki, 2003).

De acordo com Wood (1999), as interfaces devem prover um meio de interação com a aplicação de uma forma intuitiva e natural. Portanto, as preocupações dos projetistas no sentido de criar tipos legíveis, melhorar barras de rolagem, definir ícones, integrar cor, som, imagem e voz, são extremamente importantes, mas não são essenciais (Freitas, 2004). Deve-se ter uma preocupação com a maneira com a qual as pessoas utilizam os computadores, pois um usuário frustrado ou irritado pela sua experiência com um programa certamente desempenhará mal suas tarefas.

Koscinaki (2006) relata alguns critérios gerais para garantir uma qualidade razoável, observando-se que somente eles não são suficientes para obter uma solução ideal. São eles: 1) o primeiro princípio de construção é projetar. As interfaces merecem um tratamento a parte dentro do software; 2) priorizar a simplicidade. A interface deve evitar o máximo possível depender da habilidade ou da memória do usuário; 3) o vocabulário deve estar correto, onde a língua utilizada e os termos técnicos devem estar claros e corretos; 4) as mensagens e textos devem conter informações úteis e fáceis de serem compreendidos, sendo que as mensagens de erro devem sempre sugerir como resolver o problema; 5) referente a quantidade de comandos, deve-se reduzir o número de opções disponíveis simultaneamente ao usuário; 6) deve-se evitar telas sobrecarregadas de dados; 7) cuidar a consistência e não usar variações na interface sem motivos para isso; 8) a disposição, onde o conteúdo das janelas deve ser bem organizado; 8) e por fim, a navegabilidade, onde a seqüência de apresentação de comandos ou de dados deve ser logicamente organizada e clara.

Nesse sentido, sabe-se que um mesmo software pode ser interpretado de maneiras diferentes pelos utilizadores, em função do contexto de uso. Dessa forma,

programas que realizam funções complicadas são operados provavelmente por usuários qualificados, não impedindo, que falhas no projeto da interface possam ocasionar dificuldades no uso do software.

Entretanto, atualmente encontram-se diversos aplicativos no domínio de transformações no espaço de cores, os quais suas interfaces deixam a desejar em termos de usabilidade. Nesse contexto, deve-se compartilhar informações e conhecer bem os usuários da área, fazendo com que eles participem de alguma forma da elaboração da interface gráfica, reunindo as informações adquiridas, organizando-as e analisando-as, de forma que sejam devidamente representadas de modo a desenvolver uma interface amigável aos seus usuários.

O domínio de espaços de cores requer um gerenciamento nada trivial em relação a sua interface, devido principalmente a complexidade das aplicações inseridas nesse contexto. Por isso, Sommerville (2004) diz que um bom projeto de interface com o usuário é fundamental para o sucesso de um sistema, e mesmo com o uso de diversas técnicas para sua avaliação, muitos problemas de projeto de interface com o usuário podem ser descobertos.

3.2.3 Padrões de projeto

De acordo com Gamma et. al (2000), um padrão de projeto nomeia, abstrai e identifica aspectos-chave de uma estrutura de projeto para torna-lo reutilizável. Cada um dos padrões existentes focaliza um problema particular, e também identifica as classes e instâncias participantes, seus papéis, colaborações e responsabilidades.

Uma das referências mais conhecidas para padrões de projeto de software são os chamados *Gof patterns* (Gamma et al, 1995). A sigla *Gof* vem de *Gang of Four*, em referência aos quatro autores do livro. Esses padrões são associados ao paradigma de programação orientada a objetos e definidos em 23 padrões de projeto. Eles são classificados em três tipos: *Creational* (criacional ou de criação), *Structural* (estrutural) e *Behavioral* (comportamental)

O padrão Criacional ajuda a tornar o software independente de como os objetos são criados, compostos e representados. Permite, por exemplo, encapsular a criação de objetos e tornar mais organizada a programação para instanciar objetos

de tipos diferentes (Freeman et al., 2004). O padrão Estrutural é usado para identificar como agrupar objetos e classes para formar uma estrutura maior. Por fim, o padrão Comportamental que é usado em colaborações dos objetos para atingir um objetivo, com enfoque na interconexão dos objetos. Através dos padrões propostos pelo catálogo de padrões, consegue-se solucionar problemas de projeto assegurando a concepção de componentes adaptáveis e portanto reutilizáveis de fragmentos de código.

Além desses padrões, atualmente são descritos diversos tipos de padrões na literatura, como por exemplo, padrões de testes, de documentação, de interfaces gráficas, de projeto, arquiteturais, entre outros. Conforme Metsker (2002), existem outros 77 padrões que, juntos, provavelmente caracterizam os 100 mais usuais.

De acordo com Buckmann et. al (BUS 1996), um padrão é constituído por três componentes básicos: contexto, problema e solução. O contexto é uma situação onde o problema a ser resolvido é encontrado. A solução descreve uma abordagem de sucesso para resolver tal problema.

O uso de padrões está se tornando cada vez mais freqüente no desenvolvimento de software, onde é muito importante o seu uso na área de processamento e análise de imagens. Segundo Welfer (2003), a medida que o sistema de processamento e análise de imagens vai crescendo, mais e mais chamadas aos componentes deverão ser asseguradas pelo *framework desenvolvido.* Dessa forma, para gerenciar essa e as demais atividades de sistemas de imagens e promover o reuso de seus componentes que são bastante complexos, faz-se necessário o uso de diversos padrões, tanto na fase de projeto e arquitetura do sistema, quanto nas demais fases do desenvolvimento.

Portanto, a aplicação de padrões em processamento e análise de imagens ajuda a tornar o sistema mais gerenciável em relação as suas funcionalidades e as chamadas a novos componentes, auxilia a ocultar a complexidade de algumas soluções e manipula-la de forma mais flexível permitindo dessa maneira seu reuso.

3.2.4 Verificação e Validação

As atividades de verificação e validação (V&V) servem para assegurar que o software funcione de acordo com o que foi especificado. A verificação tem como

propósito de averiguar se o software está de acordo com as especificações pré-estabelecidas, e a validação é o processo de confirmação de que o sistema está apropriado e consistente com os requisitos.

Observa-se que os defeitos de software que produzem uma falha (inviabilizando a continuidade da execução de um determinado método/tarefa) são certamente bastante inconvenientes, porém a gravidade de uma falha de software também é relativa. Segundo Koscianki (2006), existem falhas com as quais usuários podem conviver, a tal ponto que o sucesso de aplicação de um produto não seja afetado. Em outros casos, a falha do programa representa um completo fracasso. Finalmente, há programas de computador responsáveis pelo controle de equipamentos valiosos ou que podem colocar em risco a segurança física de pessoas.

A evolução na área de qualidade de software tem como um dos pontos focais o processo de tecnologias abertas, seja ela de arquitetura aberta ou código aberto, é natural que os testes cada vez mais tendam a ser um processo que permita um gerenciamento global de testes. Vários autores de teste, dentre eles Crespo (2004), têm chamado a atenção para mais uma tendência que é a revolução dos *times* de testes, onde a tarefa inicial do testador tende a ser aparentemente repetitiva, porém, segundo eles, existe muito mais do que somente encontrar e testar *bugs* em softwares (Molinari, 2003).

O próximo capítulo apresentará em detalhes o processo de testes de software como meio de garantir a qualidade do produto desenvolvido.

4 TESTES DE SOFTWARE

O desenvolvimento de software envolve uma série de atividades de produção nas quais as chances de ocorrência de falhas humanas são grandes. Enganos podem acontecer em todos os processos de desenvolvimento, tanto no início do processo como nas fases de análise e especificação e nas fases de projeto e implementação. Devido a esta característica, o desenvolvimento de software é acompanhado por atividades de testes que visam garantir a qualidade (Rocha et al., 2001).

Teste de software é uma etapa fundamental do ciclo de desenvolvimento e representa uma importante premissa para alcançar padrões de qualidade no produto criado. O teste diz respeito à análise dinâmica do programa e consiste na execução do produto com o intuito de revelar a presença de erros. Segundo Whittaker (2000), teste de software é o processo de execução de um produto para determinar se ele atingiu suas especificações e funcionou corretamente no ambiente para o qual foi projetado.

Tendo em vista a importância dos testes como forma de garantir a qualidade do que está sendo desenvolvido, nesse capítulo são apresentados alguns conceitos que envolvem a atividade de teste e que farão parte da metodologia desenvolvida no domínio de transformações no espaço de cores. Inicialmente, são feitas considerações a respeito da visão geral dos testes de software e sua importância durante o processo de desenvolvimento. Após, são apresentadas as fases de teste e as principais técnicas e critérios que são utilizadas em cada uma das fases. Por fim são apresentados alguns modelos de teste existentes.

4.1 Visão geral sobre testes de software

O teste é uma atividade crucial no processo de desenvolvimento do software e tem como objetivo principal revelar a presença de defeitos em um determinado produto, como em um componente ou programa (Gimenes, 2005). Segundo Myers (1979), 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 um erro ainda não descoberto.

Maldonado (2003) classifica os erros em dois tipos: 1) computacionais, que ocorre quando a computação é incorreta, sendo que a seqüência de comandos (caminho) executada é igual à esperada; 2) de domínio, o qual acontece quando o caminho executado é diferente do esperado (um caminho errado é selecionado). Logo, o padrão IEEE 610.12-1990 (IEEE Standards Board, 1990) distingüe os termos utilizados no contexto de testes de software: Defeito (*fault*) – um passo, processo ou definição de dados incorretos; Engano (*mistake*) – uma ação humana que produz um resultado incorreto; Erro (*error*) – diferença entre o valor computado, observado ou medido e o valor real; e, Falha (*failure*) – um resultado incorreto. Contudo, para fins de entendimento deste trabalho, assume-se que erro, falha, defeito e engano são sinônimos e ocorrem quando o software não executa de acordo com sua especificação.

Conforme Pressman (1997), a atividade de teste, de maneira geral, pode ser considerada como uma atividade incremental realizada em três fases: teste de unidade, teste de integração e testes de alto nível. Os testes de unidade se preocupam em testar cada unidade do programa para garantir que a implementação de cada uma esteja correta. Logo, para garantir o correto funcionamento das unidades quando integradas, deve-se realizar o teste de integração, cujo objetivo é garantir que as interfaces entre as unidades do programa funcionem sem erros. Por fim, os testes de alto nível, divididos em teste de “validação” e teste de “sistema”, são realizados após a integração do sistema e visam garantir que a aplicação e os demais elementos que o compõe, como o sistema operacional, banco de dados, entre outros, se comuniquem adequadamente entre si e que os requisitos não funcionais estabelecidos sejam atendidos pela aplicação.

Mesmo utilizando as técnicas e critérios existentes, dividindo a atividade de teste em várias fases e utilizando ferramentas de teste, não se pode garantir um software livre de erros. Nesse sentido, as técnicas e critérios de teste têm sido elaborados com o objetivo de fornecer uma maneira sistemática e rigorosa para selecionar um subconjunto do domínio de entrada e ainda assim ser eficiente para apresentar os erros existentes, respeitando-se as restrições de tempo e custo associados a um projeto de software (Vincenzi, 1998).

Cabe ressaltar que a atividade de teste não é uma atividade trivial, mas sim uma atividade que exige muito conhecimento e planejamento. Dessa forma, segundo

Crespo (2004), na elaboração do planejamento do teste, uma das etapas é a elaboração da estratégia de teste, a qual, tem por objetivo compreender a definição dos seguintes itens: o nível de teste, isto é, a definição da fase do desenvolvimento do software em que o teste será aplicado; a técnica de teste a ser utilizada; o critério de teste a ser adotado; o tipo de teste a ser aplicado no software.

Os testes somente contribuem para aumentar a confiança de que o software funciona de acordo com o esperado, de modo que grande parte dos defeitos já foi detectada (Beizer, 1995). Portanto, um sistema de software somente é considerado portador de boa qualidade, quando atinge níveis satisfatórios e adequados de confiabilidade na realização de sua funcionalidade.

4.2 Fases de teste de software

Como mencionado anteriormente, a atividade de teste pode ser considerada como uma atividade realizada em três fases: unidade, integração e de alto nível. Segundo Gimenes (2005) assim como os métodos de desenvolvimento de software são divididos em várias fases, de modo a permitir que o engenheiro de sistemas implemente a solução do problema passo a passo, a atividade de teste também é dividida em fases. Por isso, além de fazer uso de técnicas e critérios de teste, quando programas complexos são testados é necessário dividir a atividade de teste em fases.

Conforme destaca Linnenkugel & Mullerburg (1990), com a divisão da atividade de teste em várias fases, o testador pode se concentrar em aspectos diferentes do software e em diferentes tipos de erros e utilizar diferentes estratégias de seleção de dados de teste e medidas de cobertura em cada uma delas. A cobertura do código é uma medição que determina trechos de código que foram ou não exercitados através da execução de um teste. Segundo Rocha (2005), a análise permite identificar áreas do programa não exercitadas por um conjunto de casos de teste e dessa forma avaliar a qualidade desse conjunto, sendo possível também medir o progresso do teste e decidir quando finalizar essa atividade.

Dessa forma, a seguir são apresentadas as principais fases de teste descritas na literatura, as quais são utilizadas pela ImageTest. É importante que no domínio de transformações no espaço de cores sejam realizados teste em diferentes fases,

para assim garantir a qualidade tanto na parte estrutural da aplicação quanto na funcional.

4.2.1 Teste de unidade

Nessa fase se procura identificar erros na lógica e na implementação de cada módulo do software, isoladamente. O teste de unidade se concentra na verificação da menor unidade de projeto de software: módulo, métodos, classes ou até mesmo, trechos de códigos confusos. Usando a descrição do projeto como guia, caminhos de controle importantes são testados para descobrir erros dentro das fronteiras do módulo (Pressman, 2000). Durante esta fase utiliza-se muito a técnica de teste estrutural que requer a execução de elementos específicos da estrutura de controle de cada unidade. A técnica de teste baseada em erros também tem sido usada nesta fase (Vincenzi, 2000).

Num software orientado a objetos, torna-se mais complexo trabalhar com as funções de forma isolada, isto é, testar uma função sem levar em conta a sua classe. Além disso, a herança permite abrir um leque maior de possibilidades quando fala-se em unidade, pois tanto uma classe quanto uma hierarquia de classes podem ser vistas como uma unidade. No desenvolvimento de software orientado a objetos é relevante tratar cada unidade, mas não necessariamente de uma maneira seqüencial, ou seja, não é preciso que um determinado teste chegue ao fim, para que outro possa ser iniciado.

Esse teste, no domínio de transformações no espaço de cores, faz-se necessário sua aplicação para a verificação dos valores referentes as fórmulas matemáticas e as diferentes transformações que envolvem as conversões. Para cada transformação de modelos e espaços de cores deve-se verificar a estrutura interna da implementação. Além disso, é muito comum a verificação dos valores retornados dos métodos *get()* e *set()* presentes na aplicação.

4.2.2 Teste de integração

O teste de integração verifica basicamente se as unidades testadas de forma individual executam corretamente quando colocadas juntas, isto é, quando

integradas. Nessa fase, procura-se descobrir erros nas interfaces dos módulos, durante a integração da estrutura do programa. No caso de um programa orientado a objetos, por exemplo, um tipo de teste de integração consiste em testar cada método juntamente com os métodos chamados direta ou indiretamente por ele dentro da mesma classe, isto também é chamado de teste inter-método (Lemos, 2005).

Conforme Pressman (2002), as técnicas de projeto de casos de teste funcional são as mais utilizadas durante essa fase, e o teste de integração é classificado em dois tipos: incremental e não-incremental. Na integração não incremental todos os módulos são combinados antecipadamente e o programa completo é testado. Na abordagem incremental o software é construído e testado em blocos e as interfaces têm mais probabilidade de serem testadas completamente, o que facilita o isolamento e a correção de erros. Ainda, na integração incremental podem-se utilizar duas estratégias: descendente (*top-down*) ou ascendente (*bottom-up*).

O teste de integração não é abordado diretamente pela ImageTest, porém ele pode se fazer presente no momento que é realizado um teste de unidade sobre algum método *execute()*. Ao mesmo tempo em que um teste unitário pode ser considerado teste funcional, pois verifica os resultados da execução do método, ele não deixa de estar fazendo também um teste de integração entre as classes de um sistema no domínio de transformações no espaço de cores.

4.2.3 Teste de sistema

No teste de sistema verifica-se se a integração de todos os elementos que compõem o sistema e o seu funcionamento (Rocha, 2005). O objetivo desse teste é assegurar que o software e os demais elementos que o compõem, tais como, hardware e banco de dados, se combinam adequadamente e que a função/desempenho global desejada é obtida. A técnica de teste funcional é que tem sido mais utilizada nesta fase de teste (Pressman, 2000).

O teste de sistema consiste em uma série de testes de diferentes tipos cuja finalidade principal é exercitar todo o software. Segundo Pressman (2002), esse tipo

de teste se subdivide em quatro outros testes: testes de recuperação; teste de segurança; teste de estresse e teste de desempenho.

Visto que a estrutura do software não é visualizada no teste de sistema, isto faz com que o teste de sistema aplicado ao software estruturado possa ser aplicado igualmente ao software orientado a objetos (Inthurn, 2001). Contudo, esse teste não será aplicado na metodologia proposta, porém fez-se necessário sua abordagem para o melhor entendimento do processo de teste visto na Figura 4.1 descrita a seguir.

4.2.4 Teste de validação

O teste de validação ou aceitação, como também é chamado, começa no fim do teste de integração, quando componentes individuais já foram exercitados, o software está completamente montado como um pacote, e os erros de interface foram descobertos e corrigidos. Esse teste é realizado com o propósito de avaliar a qualidade externa do produto e, na medida do possível, também a qualidade em uso.

O teste de validação é um teste com forte relação com o cliente, que participa do planejamento e realização dessa atividade. O teste de aceitação é geralmente realizado de duas maneiras: 1) Teste alfa: realizado no ambiente de desenvolvimento com os usuários finais onde o desenvolvedor ou testador registra todos os erros e problemas de uso; 2) Teste beta: realizado pelo usuário final em seu próprio ambiente, registrando todos os problemas que são encontrados e os relata ao desenvolvedor que realiza as modificações necessárias.

Os clientes que devem ser escolhidos para o teste beta devem possuir algumas das características como: capacidade crítica no uso do *software*; bom entrosamento com a empresa ou instituição desenvolvedora; preferencialmente, deve ser leigo em informática, não fazendo considerações que podem levar a interpretações equivocadas do erro encontrado; ser organizado e dispor de tempo para o teste. Com essas características o cliente passa a interagir com o processo de teste obtendo melhores resultados, fazendo assim com que após a implantação definitiva do sistema o *software* não precise passar por constantes manutenções.

O teste de aceitação é muito importante em qualquer domínio de aplicação, porém em processamento de imagens ele torna-se essencial, pois os usuários do domínio de transformações no espaço de cores devem ser especialistas para utilizar o sistema e da mesma maneira poder avaliá-lo.

Por fim, a Figura 4.1, extraída de Vincenzi (2004), ilustra as três fases de teste mencionadas acima, bem como os componentes utilizados em cada uma das fases tanto para programas procedimentais como para programas OO.

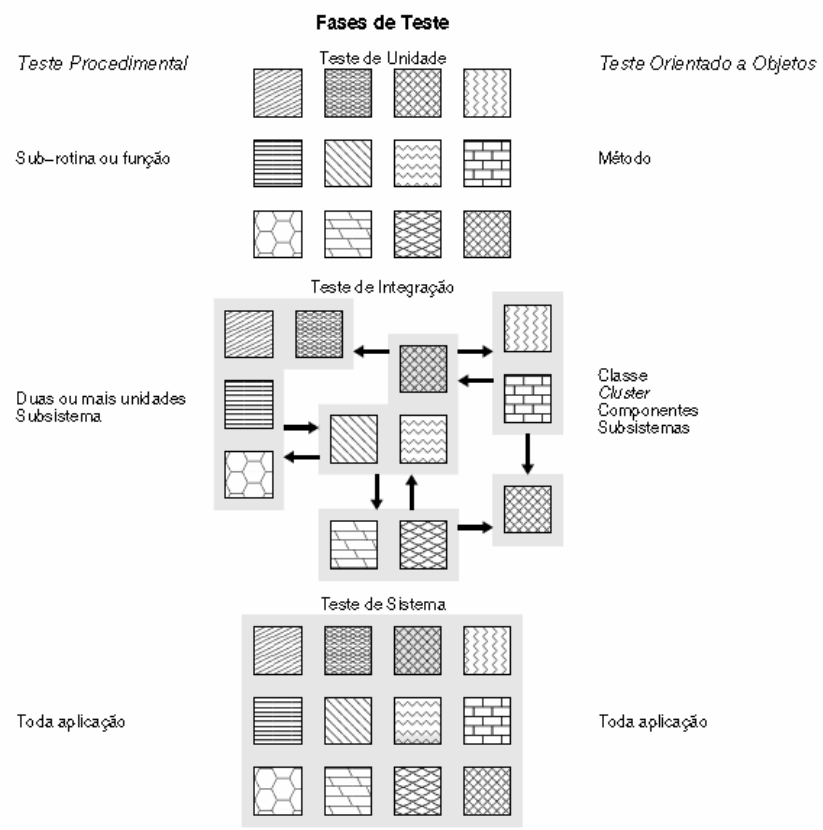

Figura 4.1: Relacionamento entre teste de unidade, de integração e de sistema: programas procedimentais e OO (Binder, 1999).

4.3 Técnicas e critérios de teste de software

As técnicas de teste visam contribuir na estruturação do processo de teste, fornecendo um esquema sistemático e rigoroso para revelar a presença de erros existentes, respeitando as restrições de tempo e custo associados a um projeto de software (Martin, 1991). Conforme Pressman (2002), técnicas de teste fornecem

diretrizes para projetar testes que exercitam a lógica interna dos componentes do software, assim como seus domínios de entrada e saída.

Dentre as várias técnicas propostas se encontram as técnicas de teste funcional, estrutural e baseada em erros. Segundo Howden (1978), o teste pode ser classificado de duas maneiras: teste baseado em especificação e teste baseado em programa. De acordo com tal classificação, pode-se enquadrar o teste funcional na primeira categoria, enquanto que as técnicas estrutural e baseada em erros enquadram-se na segunda.

As técnicas e critérios de teste fornecem ao desenvolvedor uma abordagem sistemática e teoricamente fundamentada para se conduzir e avaliar a qualidade dos testes de software (Domingues, 2002). No processo de testes as tarefas distribuídas devem ser realizadas de forma sistemática, o que deve ser feito através da utilização dessas técnicas de teste. Dessa maneira, as técnicas de teste propiciam a confecção de um plano de teste e a determinação/distribuição das tarefas que irão efetivamente compor uma determinada atividade de teste.

Entretanto, apenas utilizando a técnica de teste por si só, o conjunto de elementos a ser testado torna-se demasiado e impraticável. Por isso, são utilizados critérios de teste que auxiliam o testador fornecendo: 1) um método para avaliação de conjunto de casos de teste e 2) uma base para a seleção de casos de teste. No primeiro caso os critérios de adequação servem para evidenciar a suficiência da atividade de teste, e no segundo caso, para ajudar na construção de casos de teste (Frankl e Weyuker, 2000).

Como pode ser observado, estas técnicas devem ser aplicadas de forma complementar, de maneira que a metodologia desenvolvida explore cada uma das técnicas e critérios a fim de constituir uma abordagem de teste mais completa, eficaz, de baixo custo e de boa qualidade. A seguir, é apresentada a descrição de cada uma das técnicas que fazem parte da ImageTest.

4.3.1 Técnica funcional

O teste funcional ou caixa-preta tem esse nome por se tratar de um teste que verifica o software como se fosse uma caixa, da qual o seu conteúdo é desconhecido, sendo possível somente visualizar o lado externo dessa caixa (dados

de entrada e saída). Deste modo, o testador utiliza essencialmente a especificação de requisitos do programa para derivar os requisitos de teste que serão empregados, sem se importar com os detalhes de implementação (Beizer, 1990). Sendo assim, segundo Vincenzi (1998) uma especificação correta e de acordo com os requisitos do usuário é essencial para esse tipo de teste.

De acordo com Ostrand e Balcer (1998), o objetivo do teste funcional é encontrar discrepâncias entre o comportamento atual do sistema e o descrito em suas especificação. Dessa forma, são levadas em consideração apenas as entradas, saídas e estado do programa e o testador não tem necessariamente acesso ao código fonte do software.

O teste funcional procura basicamente revelar erros nas seguintes categorias: funções incorretas ou omitidas; erros de interface; erros nas estruturas de dados ou no acesso a banco de dados externo; erros de desempenho; e erros de iniciação e término.

Visto que os critérios desta técnica se baseiam exclusivamente na especificação do software para derivar os requisitos de teste, esses critérios podem ser aplicados indistintamente a programas procedimentais e a programas OO (Hoffman & Strooper, 1997). A seguir, são apresentados os critérios mais abordados nessa técnica.

- **Particionamento de Equivalência:** os programas normalmente se comportam de maneira comparável para todos os membros de uma classe. Devido a esse comportamento equivalente, essas classes são, algumas vezes, chamadas de partições de equivalência ou domínios (Beizer, 1990). Conforme Pressman (2000), o particionamento de equivalência divide o domínio de entrada de um programa em classes de equivalências válidas e inválidas, a partir das condições de entrada de dados identificadas na especificação.

Esse critério visa minimizar o número de casos de teste, selecionando apenas um caso de teste de cada classe de equivalência, pois em princípio, todos os elementos de uma classe devem se comportar de maneira equivalente (Maldonado e Fabbri, 2001b). Logo, se um caso de teste de uma determinada classe de equivalência revela um erro, qualquer outro caso de teste nessa classe deveria revelar o mesmo erro (Myers, 2004a).

Uma vez identificado um conjunto de partições, escolhem-se então casos de teste a partir de cada uma dessas partições. Uma boa diretriz a seguir para a seleção de casos de teste é escolher casos de teste nos limites das partições e também casos próximos ao ponto médio da partição. As partições de equivalência podem ser identificadas utilizando a especificação de programa ou a documentação de usuário e pelo testador, utilizando sua própria experiência para prever que classes de valores de entrada podem detectar erros (Sommerville, 2003).

Segundo Roper ((1994) apud (Cagnin, 2005)), a vantagem do critério particionamento de equivalência é que ele ajuda reduzir o tamanho aparente do domínio de entrada e se preocupa em criar casos de teste baseados somente na especificação. Esse critério é de difícil aplicação quando o domínio de entrada do software é simples e o processamento é complexo.

- **Análise de Valor Limite:** este critério é um complemento ao critério particionamento de equivalência, sendo que os limites associados às condições de entrada são exercitados de forma mais rigorosa. Ao invés de selecionar-se qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, pois nesses pontos pode-se concentrar um grande número de erros. Isto é, ele preocupa-se em criar casos de teste que considerem valores diretamente acima e abaixo dos limites das classes de equivalência.

O espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída. Uma vantagem adicional é que esse critério fornece mais orientação na criação de dados de teste e esses dados de teste tendem a se manifestar onde falhas podem ser encontradas (Roper, 1994).

Conforme Rocha (2005), algumas diretrizes podem ser utilizadas nesse critério:

- se uma condição de entrada especifica uma gama de valores, devem ser escritos casos de teste para os extremos dessa gama e para suas proximidades (ex., se os valores válidos forem números reais entre –1.0 e 1.0, deve-se testar –1.0, 1.0, -1.001, 1.001);
- valores de saída também devem ser particionados e devem ser escritos casos de teste que exercitem seus extremos (ex., se o programa produz

como resultados um número inteiro que varia entre 0 e 25, devem ser escritos casos de teste que façam o programa produzir esses dois extremos e, se possível, um valor negativo e outro maior que 25);

- se a condição de entrada especifica um número de valores, devem ser desenvolvidos casos de teste para o número mínimo e máximo de valores e suas proximidades (ex., se o tamanho da entrada varia de 1 a 4 números, deve-se testar o tamanho da entrada sendo 0, 1, 4 e 5);
- de forma semelhante, os valores de saída devem ser analisados e casos de teste devem ser elaborados para testar seus limites e vizinhanças (ex., se um programa calcula de 1 a 3 valores, deve-se criar casos de teste que levem ao cálculo de 0, 1, 4 e, se possível, 5 valores).

4.3.2 Técnica estrutural

A técnica de teste estrutural, conhecida como caixa-branca (em oposição ao nome caixa-preta), leva em consideração os aspectos de implementação na escolha dos casos de teste (Vincenzi, 1998). Nesta técnica, segundo Rocha (2005), os requisitos são estabelecidos com base na implementação do programa e a estrutura e os detalhes do código são considerados para derivar os casos de teste.

Este teste permite ao avaliador concentrar a atenção nos pontos mais importantes do código, de uma maneira mais precisa do que no caso do teste caixa-preta. Tais pontos podem até ser identificados durante o desenvolvimento e cobertos com o uso de assertivas e testes.

Há, entretanto, um elemento comum entre os testes caixa-preta e caixa-branca: em ambos os casos, o avaliador não sabe qual será o comportamento do produto em uma determinada situação. Esse fato pode parecer, a princípio, paradoxal para o teste caixa-branca, uma vez que o avaliador tem total acesso aos componentes internos do produto.

O teste estrutural em softwares orientados a objetos é fundamentalmente diferente dos testes em software não orientado a objetos. Desta forma, os testes convencionais poderão ser utilizados, mas deverão ser adaptados para orientação a objetos. O teste estrutural objetiva verificar se a estrutura interna da unidade está

correta, e esta verificação é efetuada através de casos de teste que visam percorrer todos os caminhos internos possíveis da unidade.

Percorrer todos os caminhos, no entanto, acaba tornando a atividade de teste muito mais complexa, devido principalmente ao polimorfismo, pois não é possível ver no código qual a unidade que está ativa em um determinado tempo, visto que, quando um estímulo é mandado, qualquer instância de uma classe pode receber este estímulo (Inthurn, 2001).

O teste estrutural é uma técnica de projeto de casos de teste que usa a estrutura de controle e o fluxo de dados para derivar os requisitos de teste (Pressman, 2000). Os critérios de teste estrutural se baseiam em diferentes tipos de estruturas para determinar quais partes do programa tem sua execução requerida. Esses critérios podem ser classificados em: baseados na complexidade, baseados no fluxo de controle e baseados em fluxo de dados. Contudo, a ImageTest fará uso somente do critério baseado no fluxo de controle.

No caso do critério grafo de fluxo de controle, o qual é um grafo dirigido, com um único nó de entrada e um único nó de saída, cada arco representa um possível desvio de um bloco para outro. Cada bloco tem as seguintes características: uma vez que o primeiro comando do bloco é executado, todos os demais são executados seqüencialmente; e não existe desvio de execução para nenhum comando dentro do bloco. Através desse grafo podem ser escolhidos os componentes que devem ser executados.

Conforme Domingues (2002), os critérios baseados no fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para derivar os requisitos de teste necessários. Os critérios mais conhecidos dessa classe são: todos-nós, todos-arcos e todos-caminhos. O critério todos-nós é abordado pela ImageTest, o qual ele exige que a execução do programa passe, ao menos uma vez, em cada nó do grafo de fluxo, ou seja, que cada comando do programa seja executado pelo menos uma vez.

4.4 Ferramentas de testes

Como descreve Binder (1999), a automatização permite verificação rápida e eficiente das correções de defeitos, agiliza o processo de depuração, permite a

captura e análise dos resultados de teste de forma consistente e facilita também a execução de testes de regressão, nos quais os casos de teste podem ser reutilizados para revalidação do software após modificação. A utilização de ferramentas de teste também facilita a adoção de critérios de teste pela indústria e a condução de estudos empíricos com vistas a comparar diversos critérios de teste existentes (Vincenzi, 2004).

A disponibilidade de ferramentas de teste permite a transferência de tecnologia para as indústrias e contribui para uma contínua evolução de tais ambientes, fatores indispensáveis para a produção de software de alta qualidade (Domingues, 2002). Além disso, tais ferramentas auxiliam pesquisadores e alunos de Engenharia de Software a adquirir os conceitos básicos e experiência na comparação, seleção e estabelecimento de estratégias de teste (Vincenzi, 2000).

A seguir é apresentada a descrição de algumas ferramentas de teste obtidas por meio da pesquisa na Internet e de artigos, sendo sua descrição baseada na documentação existente de cada uma. A seleção de algumas das diversas ferramentas existentes, foi baseada por serem ferramentas populares na área de testes, tais como *Jtest*[3], *TestComplete*[4], *XDETester*[5], *JUnit*[6], *PureCoverage*[7], *Jabuti*[8], *JprobeSuite*[9]*, Cobertura*[10] e *ANT*[11]*.* São elas:

- ***Jtest (Parasoft Corporation):*** é uma ferramenta integrada de teste Java, a qual oferece apoio a verificação de padrões de codificação e ao teste de unidade, e opera diretamente sobre o código fonte que deve estar disponível. No teste de unidade os casos de teste são gerados e executados automaticamente a partir da análise da classe em teste. Realiza o teste funcional, em que os casos de teste gerados devem ser alterados para verificar se as interfaces públicas da classe operam como esperado. Ela disponibiliza informações sobre a cobertura de métodos e classes segundo o paradigma do teste estrutural. Contudo, essa ferramenta não é gratuita.

---

[3] Disponível em http://www.parasoft.com/jtest
[4] Disponível em http://www.automatedqa.com/products/testcomplete/index.asp
[5] Disponível em http://www-136.ibm.com/developerworks/rational/products/xdetester
[6] Disponível em http://www.junit.org/
[7] Disponível em http://www.rational.com
[8] A descrição dessa ferramenta pode se encontrada no texto de Nardi (2005)
[9] Disponível em http://www.quest.com/jprobe/
[10] Disponível em http://cobertura.sourceforge.net/
[11] Disponível em http://ant.apache.org/

- ***TestComplete:*** oferece apoio ao teste de programas implementados em diversas linguagens, incluindo Java. Para testes em Java, o testador deve inserir comandos especiais nas classes em teste e criar um *script* para sua execução. O relatório de teste denominado *TestLog* exibe todos os métodos executados em uma estrutura hierárquica. Essa ferramenta não oferece diretamente a análise de cobertura, além disso, ela não utiliza qualquer critério do teste funcional.

- ***XDETester:*** ferramenta para teste funcional Java. Tem a capacidade de gravar *scripts* e posteriormente reproduzi-los, de forma semelhante ao *TestComplete*. Não implementa a análise de cobertura e não utiliza o conceito de classes de equivalência ou valor-limite.

- ***JUnit:*** é um *framework* livre para apoio ao teste de unidade, que contém classes e métodos específicos para declaração e checagem de asserções em Java. O *Junit* disponibiliza uma versão gráfica e outra via linha de comando para execução dos casos de teste e exibição do relatório do teste. Não oferece análise de cobertura ou utiliza qualquer critério específico de teste.

- ***PureCoverage (Rational Software Corporation):*** é uma ferramenta de análise de cobertura para códigos C++ e Java que aponta as áreas de códigos que foram exercitadas ou não durante os testes, disponibilizando a cobertura do código através de arquivos, módulos e linhas de código. Coleta e exibe interativamente dados de cobertura para execuções individuais e múltiplas. E, também, uni dados de cobertura de diferentes programas que compartilham código fonte comum e de diferentes *builds*.

- ***JaBUTI:*** Automatiza parte das etapas necessárias para a aplicação da técnica estrutural em código Java. Suporta os seguintes critérios: todos-nós, todos-arcos, todos-usos e todos-potenciasi-usos. É de licença própria, de uso livre para pesquisa. Porém, ainda não se encontra disponível para *download* devido a detalhes de direitos autorais.

- ***JprobeSuite:*** Conjunto de três ferramentas que ajuda a eliminar gargalos de execução causados por algoritmos ineficientes em Java apontando causas de perdas de memória; identifica potencias perigos de concorrência e *deadlocks*; e também mede quanto do código está sendo exercitado.

- ***Cobertura:*** Realiza teste de cobertura sobre o código especificado permitindo identificar o percentual de código testado. Indica o quanto cada parte do código foi testado ou se alguma parte não foi. Utiliza o critério todos-nós da técnica estrutural e, fornece relatórios do projeto testado.

- ***ANT:*** O *ANT* é um projeto *open source,* produzido pelo grupo Jakarta da Fundação Apache, é considerada uma poderosa ferramenta que auxilia no desenvolvimento, testes e instalações de aplicações Java (The Apache Ant Project, 2006). Com o *ANT* pode-se escrever um pequeno arquivo XML descrevendo várias funções (*tasks*) que deseja automatizar. Dentre algumas das operações do *ANT* inclui-se a execução de funções do *JUnit.*

Contudo, segundo Domingues (2002) como ferramentas de apoio à aplicação dos critérios baseado em análise de fluxo de controle e de dados em programas procedimentais pode-se citar as ferramentas PokeToll (Chaim, 1991; Maldonado et al., 1989), ATACOBOL (Sze, 2000) e xSuds (Agrawal et al., 1998; Telcordia Technologies, Inc., 1998). E também para o teste de programas OO a ferramenta Panorama (International Software Automation, 1999a,b).

Nessa dissertação procurou-se fazer uso de ferramentas gratuitas, por isso serão utilizados o *JUnit* e a *Cobertura*, as quais foram escolhidas pela facilidade da execução automática dos testes e por fazer uso do critério *todos-nós* da técnica estrutural. Maiores detalhes podem ser vistos no Capítulo 5. As demais ferramentas gratuitas analisadas ou não fazem uso de critérios, ou não executam testes na linguagem de programação Java.

5 IMAGETEST: UMA PROPOSTA DE METODOLOGIA PARA APLICAÇÃO DE TESTES DE SOFTWARE NO DOMÍNIO DE TRANSFORMAÇÕES NO ESPAÇO DE CORES

Embora a qualidade do software dependa significativamente de um processo de desenvolvimento bem-estruturado, o principal indicador de qualidade no desenvolvimento do software é a satisfação do cliente.

Percebe-se que as aplicações de processamento e análise de imagens estão cada vez mais presentes nas nossas vidas. Essas aplicações de imagens abrangem várias áreas e subáreas como astronomia, biologia, medicina nuclear, geografia, imageamento médico, a análise de imagens de satélite, o processamento e filtragem de imagens, a classificação automática de imagens indexadas por conteúdo, o sensoriamento remoto, aplicações industriais, a análise de minerais, segmentação, transformação no espaço de cor, entre outros.

Devido à diversidade de subáreas no contexto de processamento e análise de imagens, adicionando a complexidade da área por compor-se basicamente de métodos matemáticos por trás de um contexto do problema implementado, e também, muitas vezes por haver a necessidade de mais recursos computacionais e de memória, torna-se necessário a aplicação de testes para garantir a qualidade das ferramentas.

Contudo, sabe-se que o sucesso tanto do desenvolvimento de software quanto da aplicação de testes de software está intrinsecamente relacionada a metodologia de desenvolvimento. Sem uma metodologia para a atividade de teste, os riscos são maiores uma vez que existe maior dependência das pessoas envolvidas e a verificação do andamento se torna uma atividade complexa e indecisa (Dias, 2006)

Para a obtenção de bons resultados na execução do processo de testes, o presente trabalho visa propor o desenvolvimento de uma metodologia que proporcione suporte aos testes de software no domínio de transformações no espaço de cores.

Embora técnicas, critérios e ferramentas sejam utilizadas durante todo o processo de desenvolvimento de software a fim de evitar que erros sejam introduzidos no produto, a atividade de teste continua sendo de fundamental

importância para a eliminação de erros que persistem (Maldonado, 1991). Como observado por Inthurn (2001), técnicas e critérios de testes têm sido desenvolvidos para fornecer a geração e avaliação de conjuntos de teste que melhor se enquadrem no contexto analisado, fornecendo assim um mecanismo que permita auxiliar e avaliar a qualidade da atividade de teste.

Dessa forma, a presente proposta de metodologia para aplicação de testes no domínio de transformações no espaço de cores faz uso da técnica estrutural e funcional. Primeiramente é verificada a estrutura interna do software, bem como se suas funcionalidades estão de acordo com as especificações dos requisitos, finalizando o processo com a aplicação de testes de validação juntamente com o usuário.

Levando em consideração que, cada uma das técnicas possui um conjunto de critérios de teste, cabe lembrar que essas técnicas são complementares e a questão que deve ser ressaltada está no fato de como utilizá-las. As vantagens de cada uma delas deve determinar a atividade de teste de forma produtiva.

A metodologia propõe o uso de alguns critérios existentes na literatura para o devido desenvolvimento e aplicação dos testes, tais como o critério todos-nós, que pertence a técnica estrutural e que será aplicado nos testes de unidade de forma automatizada; e os critérios particionamento de equivalência e análise de valor limite que correspondem a técnica funcional.

Sabe-se que no caso de teste de software, existem vários modelos de testes que visam otimizar, criar ou representar um processo de teste (Molinari, 2003). No âmbito de processamento e análise de imagens, apesar de ainda existirem poucas referências na literatura, percebe-se que os testes são de vital importância para diminuir os erros e garantir a qualidade da ferramenta desenvolvida, além de garantir a confiabilidade para os usuários que utilizarão as funções disponibilizadas.

Dessa forma, devido a lacuna existente na literatura, referente ao domínio de transformações no espaço de cores, torna-se necessário desenvolver uma metodologia de testes de software para aplicá-la de acordo com as necessidades do domínio e disponibilidade de recursos, visando organizar toda a atividade de teste e documentá-la com o propósito de minimizar os erros durante todo o processo de desenvolvimento de software.

5.1 Descrição da ImageTest

Tendo em vista a grande importância que vem sendo dada ao desenvolvimento de metodologias para produzir software com maior qualidade, e testá-los de forma organizada e coerente, elaborou-se uma proposta de metodologia de testes de software para processamento e análise de imagens no domínio de transformação no espaço de cor.

A ImageTest é uma proposta de metodologia para a aplicação de testes de software no domínio de transformações no espaço de cores. Ao desenvolvê-la, levou-se em consideração alguns fatores importantes para o domínio, a saber são:

- complexidade dos algoritmos implementados: essa complexidade é decorrente do fato de que o domínio de transformações no espaço de cores utiliza diversas fórmulas matemáticas para o desenvolvimento das funcionalidades;
- planejamento dos testes de forma incremental: visto que os requisitos para o desenvolvimento do sistema são levantados aos poucos, o planejamento dos testes é feito também de maneira incremental, de acordo com o andamento do desenvolvimento do software.

Essa metodologia contempla o planejamento dos casos de teste e um modelo de documentação para guiar os mesmos. A implantação do processo de teste envolve um conjunto de atividades que vai desde o levantamento das necessidades do ambiente a qual está sendo empregada, até o acompanhamento das atividades realizadas como, planejamento, aplicação e documentação dos testes, constituindo assim, um completo ciclo de implantação da atividade de teste.

A metodologia proposta se subdivide em cinco atividades, descritas na Seção 5.4, que buscam mostrar o ciclo de vida do processo de teste. A ImageTest propõe que as atividades de teste sejam aplicadas de forma iterativa e incremental, ou seja, ela será integrada a cada fase de desenvolvimento de software, em que os casos de testes devem ser gerados para testar cada unidade do software que estiver pronta e o seu produto final, sem aumentar muito o custo e o prazo de desenvolvimento.

Segundo Inthurn (2001), essa integração da atividade de teste em um processo de desenvolvimento provê uma oportunidade de melhorar a coordenação entre os produtos desenvolvidos e os testes relacionados, a fim de encontrar e

remover as falhas. Essa coordenação pode levar a uma menor redundância de trabalho no processo de desenvolvimento de software, visto que ao se construir ou modificar um produto do processo de desenvolvimento é importante verificar a sua correção. A integração da atividade de teste ao processo de desenvolvimento de software proporciona uma melhoria na produtividade e na qualidade do produto final.

Uma das atividades mais relevantes e pertinentes ao teste é o projeto de casos de teste. Nesta atividade concentram-se um conjunto de técnicas, critérios e métodos para elaborar casos de teste que forneçam ao projetista de software, uma abordagem sistemática e teoricamente fundamentada (Inthurn, 2001). Como destacado por Maldonado et al. (1998), tem-se observado que a própria atividade de projeto de casos de teste é bastante efetiva em evidenciar a presença de defeitos de software. Além disso, Binder (1994a) também destaca a necessidade de iniciar os testes o quanto antes no processo de desenvolvimento.

Por meio do uso de ferramentas é possível reduzir o esforço necessário para a realização do teste, bem como diminuir os erros que são causados pela intervenção humana nessa atividade (Vincenzi, 1998). Na prática, o critério de teste está fortemente condicionado a sua automatização (Vincenzi, 2004). Para tanto, a metodologia proposta fará uso de testes automatizados em conjunto com testes manuais, buscando dessa maneira extrair as vantagens que cada uma das formas proporciona.

Dessa forma, o uso de uma metodologia de testes voltada para o domínio de transformação no espaço de cor visa aumentar a qualidade do software desenvolvido, proporcionado assim, maior confiabilidade e segurança aos usuários do sistema.

5.2 Tópicos de teste cobertos pela ImageTest

Conforme destacado por Harrold (2000), a realização de estudos comparando critérios de teste e procurando estabelecer uma relação entre quais tipos de defeitos cada um dos critérios de teste é capaz de revelar, é de fundamental importância para se acumular conhecimento sobre os diversos critérios de teste que irá auxiliar na definição de estratégias adequadas a cada tipo ou tipos de defeitos que se deseja revelar.

De acordo com Myers (2004 a) e Roper (1994), critérios de testes funcionais e critério de testes estruturais devem ser utilizados em conjunto para que um complemente o outro. Dessa forma, a metodologia propõe a utilização dos testes especificados a seguir, visando estabelecer uma metodologia de testes que seja de possível aplicação sem que haja um número demasiado de casos de teste. São eles:

- **Teste de Unidade:** O teste de unidade se concentra na verificação da menor unidade de projeto de software: o módulo. No contexto de programas orientado a objetos, o teste de unidade classifica como sendo a menor unidade o método. Tendo por base que esse tipo de teste identifica erros de lógica e de implementação, cabe lembrar que a maioria dos métodos ou funcionalidades de uma ferramenta de processamento e análise, no domínio de transformações no espaço de cores de imagens, produz como resultado uma matriz de *pixels*. Isto é, a implementação de um método tem como entrada uma imagem e produz como saída outra imagem.

Logo, uma forma de realizar o teste de unidade em alguns métodos existentes nas aplicações de processamento de imagens, é analisando a matriz de *pixels* da imagem, isto é, verificar o valor *pixel* a *pixel* da matriz. Dessa maneira, quando necessário, deve-se capturar os valores dos canais RGB de uma matriz de *pixel* para poder fazer as devidas verificações nos testes. Entretanto nos demais métodos do domínio de transformações do espaço de cores, faz-se necessário a verificação dos métodos relacionados as transformações dos modelos de cores.

Inicialmente, de acordo com a metodologia, para realizar o teste de unidade deve-se fazer uma comparação da funcionalidade que se deseja testar da ferramenta em análise com uma ferramenta já existente e bem conceituada na área de imagens que disponibilize o domínio de espaços de cores. Por exemplo, para realizar o teste de unidade na função “Separar Bandas - Split CMY” do software Arthemis[12], o qual foi desenvolvido pelo Grupo de Processamento de Informação Multimídia – GPIM (UFSM, Santa Maria), antes de aplicá-lo deve-se verificar se a funcionalidade “Separar Bandas - Split CMY” está sendo executada conforme a mesma funcionalidade de algum software conceituado na área de imagens, escolhido pelo testador. Isso se deve ao fato de poder garantir, de alguma forma,

[12] Maiores informações podem ser encontradas no *website* do grupo em http://w3.ufsm.br/gpim

que a funcionalidade está realizando sua função corretamente para que então o código dessa funcionalidade possa ser testado.

Depois que a comparação foi realizada, a elaboração e aplicação dos testes nessa fase têm início após a implementação de cada unidade do software. Assim, a equipe de programação ao terminar a implementação de cada unidade deve passar o que foi implementado a equipe de testes, a qual poderá dar início ao desenvolvimento dos testes. Dessa maneira, é contemplada a automação dos casos de teste, utilizando o *framework* JUnit, que é específico para a realização de testes de unidade em conjunto com a ferramenta *Cobertura* que utiliza o critério todos-nós da técnica estrutural. Por fim, a ferramenta *Cobertura* faz a verificação da cobertura dos testes realizados, informando através de gráficos no formato *html* quais as classes e métodos testados e qual a porcentagem do quanto cada classe foi testada.

- **Teste Funcional:** O teste funcional ou caixa-preta tem por objetivo testar se as funcionalidades presentes na documentação do sistema funcionam como especificadas. Dessa maneira, após ser realizado o teste de unidade deve-se realizar o teste funcional da mesma funcionalidade baseando-se nos dois critérios apresentados a seguir.

Nesse teste foram escolhidos os critérios de particionamento de equivalência e análise de valor limite, os quais estão explicados na Seção 4.3.1 do Capítulo 4, para selecionar um conjunto de casos de teste. Esses critérios permitem que sejam aplicados em conjunto sendo que um complementa o outro, e verificam os valores extremos que podem ser testados, escolhendo casos de testes nas fronteiras das classes de saída.

A aplicação dos critérios do teste funcional é realizada de forma manual, visto que não há ferramentas disponíveis que suportem os critérios citados. Porém, pode-se fazer uso do *framework JUnit* para a automatização dessa atividade quando for conveniente. Esse teste, também é aplicado após cada término da implementação das unidades do software, desde que as mesmas funcionem adequadamente. Com isso, o próximo passo dos testes é verificar com o cliente se o sistema está de acordo com as expectativas traçadas.

Cabe lembrar que o testador que realiza esses testes deve conhecer as peculiaridades das operações que envolvem o domínio de transformação no espaço de cor, para então poder fazer um julgamento correto das funcionalidades testadas.

- **Testes de Aceitação:** O teste de aceitação é um teste exploratório voltado para validar aquilo que o usuário deseja. Esse teste foi escolhido para fazer parte da ImageTest, pois, trata-se de um teste que por ser realizado pelo usuário, tem grande probabilidade de encontrar defeitos que não são visualizados pelo testador e pelo programador.

Esse teste é aplicado na fase final dos testes pelo usuário do sistema. Então após a realização dos testes de unidade e funcional, o software é instalado na máquina do usuário, onde o mesmo deve verificar se as funcionalidades do software estão de acordo com o que foi solicitado. Para isso, o usuário deve preencher os relatórios para a formalização dos testes propostos pela ImageTest, o qual pode ser verificado com maiores detalhes na Seção 5.5.

5.3 Estrutura da ImageTest

A metodologia está dividida em cinco atividades, como ilustrado na Figura 5.1. Essas atividades foram projetadas visando estabelecer um conjunto de ações necessárias para a aplicação do processo de teste em softwares de processamento e análise de imagens no domínio de transformações no espaço de cores, de forma clara, objetiva e sistematizada, fornecendo assim uma atividade que seja de fácil entendimento e aplicação.

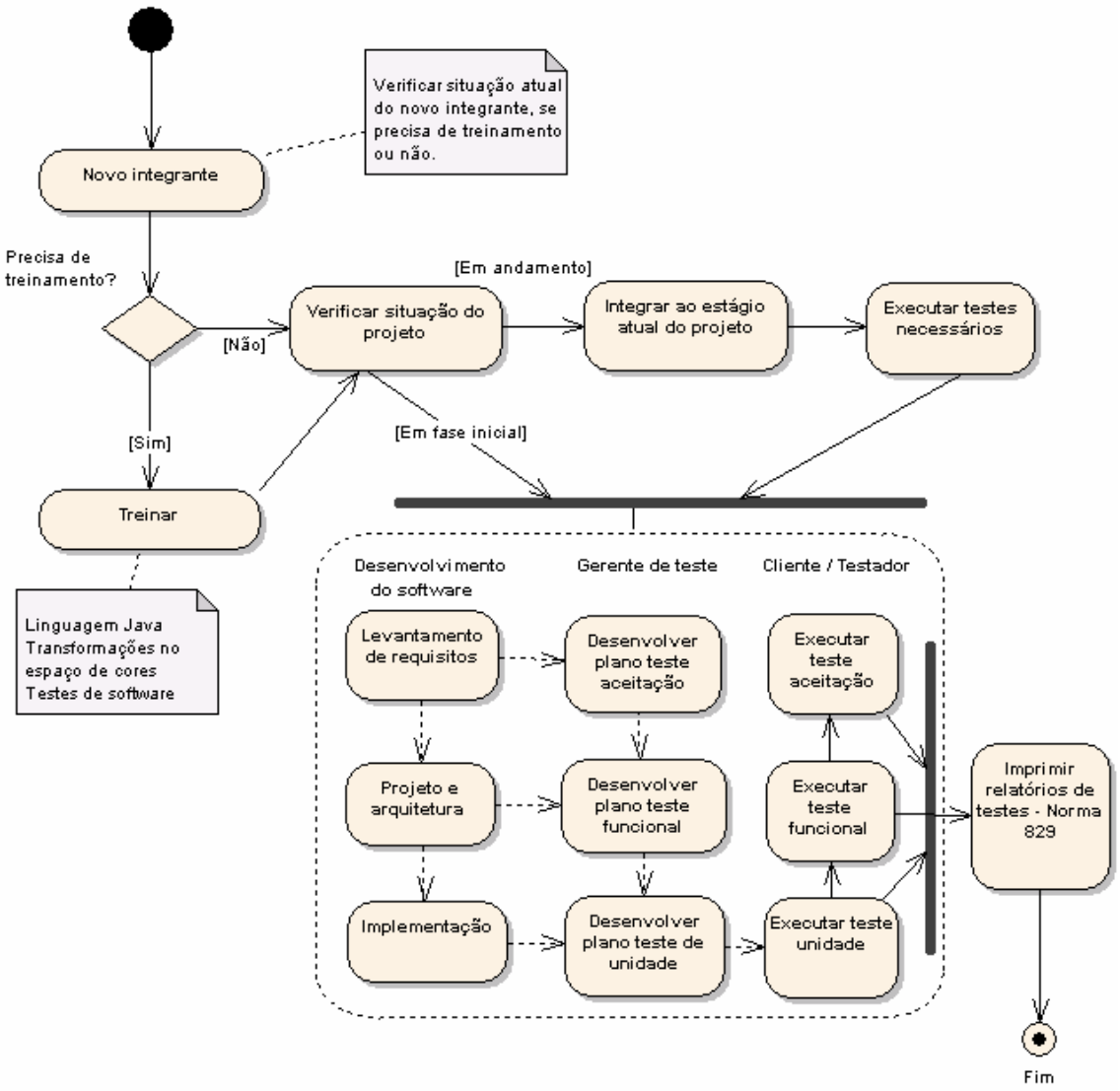

Figura 5.1 – Processo das atividades da ImageTest.

As cinco atividades pertencentes à estrutura da metodologia são:

1. ***Treinar (aprendizado):*** a atividade de treinamento e/ou aprendizado é a etapa inicial da ImageTest. Esta atividade consiste na capacitação de novos integrantes da equipe de testes às atividades referentes à área de processamento e análise de imagens, a subárea de transformação no espaço de cor e ao processo de testes de software. Leva-se em consideração que os novos integrantes ainda não detenham conhecimento das atividades, dessa forma, essa atividade divide-se em duas etapas. São elas:

   1ª - Estudo sobre conceitos básicos de testes de software, *framework*, técnicas, processo de teste e documentação.

2ª - Estudo sobre a API JAI, Java e conceitos básicos que envolvem a área de processamento e análise de imagens e o domínio da transformação no espaço de cor.

Contudo, se o novo membro já possuir conhecimento sobre as etapas pertencentes a essa atividade, ele poderá passar automaticamente para a próxima atividade.

2. ***Verificar situação do projeto:*** nesta etapa deve-se somente verificar se o projeto está em andamento ou está em fase inicial, para então verificar em qual estágio ele se insere.

   Se o projeto está em fase inicial, o analista de teste deverá elaborar o plano dos testes de acordo com o andamento da equipe de desenvolvimento. Após a codificação o testador poderá iniciar a execução dos testes e passar para a última etapa que é a fase de documentação.

3. ***Integrar ao estágio atual do projeto:*** nessa atividade, é apresentada ao novo membro da equipe de testes toda a documentação existente referente aos testes realizados na ferramenta em análise e, também, lhe é mostrado o banco de imagem para teste, o qual consiste em um banco de dados com imagens para a realização dos testes. Esse banco é elaborado pela equipe de testes em conjunto com a equipe da área de imagens, para então definir as imagens que melhor se enquadram nos testes que serão realizados. Dessa forma o integrante poderá ter conhecimento sobre os seguintes aspectos relacionados aos testes:

   - Quanto do sistema já foi testado;
   - Qual o estágio atual em que se encontram os testes; e,
   - Quais os documentos que serão preenchidos pelo testador durante a atividade de testes.

   Caso a metodologia seja aplicada no início do desenvolvimento da ferramenta, essa atividade então não é utilizada, ou seja, não é obrigatória. Ela somente é aplicada quando entrar um novo integrante na equipe de testes e o projeto já estiver em andamento.

4. ***Executar testes necessários**:* nessa atividade a metodologia define um processo de teste que tem por objetivo prever a realização das atividades de planejamento de testes, execução e acompanhamento dos testes de unidade, funcionais e aceitação.
5. ***Imprimir relatórios de testes:*** por fim, esta última atividade consiste na aplicação de uma série de documentos que serão utilizados para a gerência do processo de teste, tanto na fase de preparação quanto na fase de registro dos resultados. Essa documentação será baseada na norma IEEE 829 (1998), que é um padrão para a documentação do processo de teste, a qual pode ser vista com mais detalhes na Seção 5.5.

Através da Figura 5.1 é possível visualizar as atividades da metodologia, partindo do princípio da chegada de um novo membro a equipe de testes.

Por meio da Figura 5.1, observa-se que qualquer novo membro da equipe de testes deverá possuir domínio sobre as atividades propostas para então, se tornar um integrante capaz de realizar as devidas funções. Para passar por todas as atividades propostas ImageTest, o projeto deve estar em andamento e o novo integrante não possuir conhecimento sobre a área de imagens ou testes.

O ambiente o qual será aplicada a metodologia proposta, define dois integrantes para a equipe de testes: o gerente de testes e o testador. Sendo assim, os integrantes da equipe trabalham de forma contínua, ou seja, os dois terão participação em quase todas as atividades relacionadas ao testes.

As tarefas distribuídas devem ser realizadas de forma sistemática, o que deve ser feito através da utilização de técnicas de teste. As técnicas de teste propiciam a confecção de um plano de teste e a determinação/distribuição das tarefas que irão efetivamente compor uma determinada atividade de teste.

Ao utilizar uma metodologia que possua um ciclo de vida na forma incremental e iterativa, é necessário especificar o modelo de desenvolvimento que é utilizado para a construção do software. Isso se faz relevante para que a equipe de testes possa definir em quais as fases de desenvolvimento as suas atividades irão se enquadrar.

5.4 Automação dos testes

Esta seção tem por objetivo descrever sobre as ferramentas que serão utilizadas para realizar os testes de unidade de forma automatizada.

A ferramenta *Cobertura* é gratuita e encontra-se disponível na Internet, assim como o *JUnit*. Com o uso da *Cobertura* é possível verificar qual o percentual do código foi validado pelos testes unitários. Cabe ressaltar que todos os testes unitários são levados em consideração na realização de testes de cobertura, mesmo aqueles que apresentam falhas ou erros. Essa ferramenta faz uso do critério todos-nós que pertence a técnica de teste estrutural. Com isso, a *Cobertura* permite verificar o percentual de linhas de código cobertas (*line coverage*) e o percentual de desvios lógicos cobertos (*branch coverage*) através de gráficos. Caso não seja possível realizar algum tipo de teste, a ferramenta gera os relatórios informando onde estes não são aplicáveis. A *Cobertura* fornece relatórios que possuem níveis de detalhe do projeto como um todo ou também de linhas de uma classe particular.

O *JUnit* foi escolhido como a ferramenta utilizada para realizar os testes de unidade pois é uma ferramenta gratuita e de fácil utilização. 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, se considera que a classe ou método está de acordo com a especificação, incrementando a qualidade daquela unidade.

Para utilizar o *JUnit*, é necessário criar uma classe que estenda `junit.framework.TestCase`. Segundo Massol (2005), o *framework JUnit* possui três classes principais: `TestCase`, `TestSuite` e `TestRunner`. O `TestCase`, ou caso de teste, contém um ou mais testes representados pelos métodos `testNomedoMetodo`. Um caso de teste é usado para agrupar mais testes, que exercitem comportamentos comuns. O `TestSuite`, ou conjunto de testes, roda um conjunto de casos de testes, os quais podem incluir outros conjuntos de testes. Por fim, o `TestRunner`, ou

executor de testes, é uma interface de usuário para lançamento de testes, onde a BaseTestRunner é a superclasse para todos os executores de testes.

O *JUnit* facilita bastante a criação e execução de testes, mas elaborar bons testes exige mais. Dentro deste contexto, incluem-se até mesmo os métodos get/set. Segundo Clark (2005), 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;
- é elegante e simples. Quando testar um programa se torna algo complexo e demorado, então não existe motivação para o programador fazê-lo;
- permite que, uma vez escritos, os testes sejam executados rapidamente sem que, para isso, seja interrompido o processo de desenvolvimento;
- verifica os resultados dos testes e fornece uma resposta imediata;
- possibilita a criação de uma hierarquia de testes que permite testar apenas uma parte do sistema ou todo ele;
- permite que o programador perca menos tempo depurando seu código;
- todos os testes criados são escritos em Java;
- é LIVRE; e
- possui integração com ferramentas populares como Ant e Maven, e IDEs populares como Eclipse, IntelliJ, e Jbuilder.

Contudo, fazendo uso somente do *JUnit* não é possível identificar qual o percentual de código testado, assim, partes importantes podem não ter sido testadas. Para identificar o que foi e o que não foi testado, a metodologia propõe a utilização da ferramenta *Cobertura* (Mark, 2006), a qual realiza teste de cobertura sobre o código especificado, sobretudo dos testes executados no *JUnit*.

5.5 Formalização do processo de teste

Durante a atividade de teste são elaborados alguns documentos baseados na norma IEEE 829 (1998) – *Standard for Software Test Documentation*, que visam colaborar com a garantia da qualidade do processo de teste.

A documentação proposta pela norma IEEE 829 é composta de oito documentos, sendo que a ImageTest vai fazer uso de apenas quatro documentos, baseando-se no fato de não fazer uso exagerado de documentação. Os documentos selecionados pela metodologia são:

- Plano de teste;
- Especificação dos casos de teste;
- Diário de teste;
- Relatório - resumo de teste.

Plano de teste: o plano apresenta o planejamento para execução do teste, incluindo a abrangência, abordagem, recursos e cronogramas das atividades de teste. Ele é preenchido pelo gerente de testes.

Especificação dos casos de teste: a especificação define os casos de teste, incluindo dados de entrada, resultados esperados, ações e condições gerais para a execução do teste. Esse relatório é preenchido pelo gerente de teste em conjunto com o testador.

Diário de teste: O diário de teste é preenchido logo após o testador executar algum teste. Esse relatório registra os detalhes relevantes relacionados com a execução dos testes.

Relatório - resumo de teste: Esse relatório apresenta de forma resumida os resultados das atividades de teste associadas com uma ou mais especificações de projeto de teste, sendo preenchido também pelo testador.

Na escolha desses quatro documentos levou-se em consideração os de maior relevância para o contexto da metodologia proposta, visando garantir a documentação das principais atividades realizadas e também visando à diminuição do excesso de documentação. Por meio da Figura 5.2, pode-se visualizar o processo de documentação dos testes da ImageTest.

Inicialmente, a metodologia de teste deve possuir um planejamento global para organizar todo o processo. Fica sob responsabilidade do gerente de teste elaborar o planejamento dos testes e preencher também o relatório de especificação de casos de testes e, então enviar os documentos e informações pertinentes a essas atividades ao testador. O testador por sua vez, tem a responsabilidade de executar os testes conforme lhe foi solicitado e preencher os documentos cabíveis a suas responsabilidades que são o “diário de teste” e o “relatório resumo de teste”.

No caso de ocorrência de erros no produto analisado, o relatório “diário de teste” deve ser enviado à equipe de desenvolvimento relatando os defeitos encontrados, para que assim possam ser feitas as devidas correções. Os modelos de documentos gerados pela ImageTest encontram-se na seção de Apêndices.

Esse planejamento de testes é desenvolvido de forma incremental, visto que, assim que a equipe de desenvolvimento elaborar o projeto e levantar os requisitos para dar início ao desenvolvimento do software, a equipe de testes começa a elaborar o planejamento dos testes que serão realizados.

O propósito desta metodologia consiste em apresentar de forma simplificada, detalhando a importância e o modo como são realizadas as atividades de testes nas diversas fases do ciclo de vida do software para o domínio de transformação no espaço de cor, descrevendo em detalhes toda atividade desde a fase de aprendizagem até a fase de documentação.

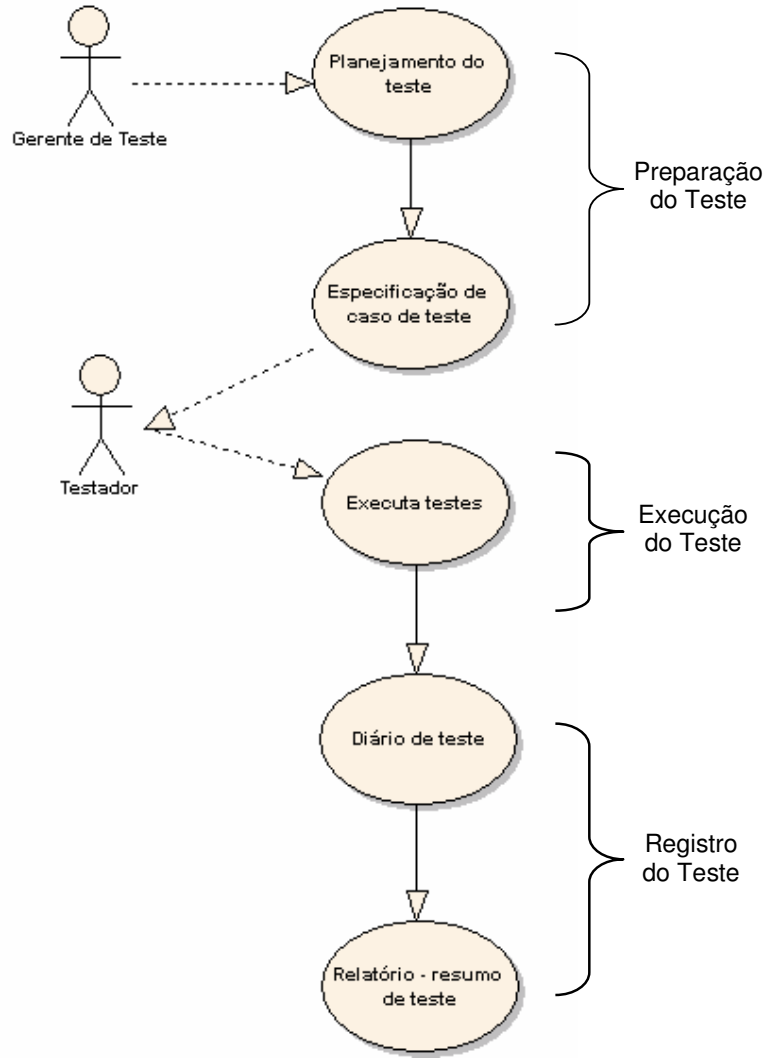

Figura 5.2 – Processo de documentação dos testes da ImageTest

Essa metodologia visa corrigir o maior número de erros encontrados nas funcionalidades da transformação no espaço de cor, mesmo porque de acordo com Myers (2004), possivelmente nem todos os métodos e funções da ferramenta serão testados.

6 ESTUDO DE CASO

Neste capítulo é apresentada a aplicação da proposta da metodologia ImageTest. A aplicação dessa metodologia ocorreu em um projeto real de desenvolvimento e manutenção de software, com o objetivo de avaliar a qualidade do software, assim como avaliar o quão importante é fazer uso de uma metodologia de testes no domínio de transformações no espaço de cores. Esta aplicação ocorreu na forma de um estudo de caso, pois resultados e análises foram feitos sobre a metodologia utilizada.

6.1 Aplicação da ImageTest

A aplicação da metodologia ImageTest foi realizada no contexto de um grupo de pesquisa em um projeto de desenvolvimento e manutenção de software Arthemis para processamento e análise de imagens, com o objetivo de garantir a qualidade do software desenvolvido, verificando possíveis falhas existentes no domínio de transformações no espaço de cores.

O grupo GPIM teve início no ano de 2002 como resultado do crescimento das atividades de processamento e análise de imagens e da diversificação das atividades de pesquisa dentro da UFSM. Atualmente, o grupo conta com uma equipe multidisciplinar composta por 30 pessoas.

A ferramenta Arthemis é implementada utilizando a linguagem de programação Java, a API Swing[13] (*Application Programming Interface*) e Java Advanced Image[14], em conjunto com o desenvolvimento baseado em componentes e padrões de projeto. Sua funcionalidade é voltada para 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.

O domínio de transformações no espaço de cores foi desenvolvido como um módulo independente dos outros módulos do sistema, sendo possível seu uso em

[13] Java Foundation Classes (JFC/Swing): http://java.sun.com/products/jfc.
[14] Página oficial de Java Advanced Image: http://java.sun.com/product/java-media/jai.

outras aplicações ou outros módulos da aplicação na qual a ferramenta esta inserida. A ferramenta facilita a visualização das transformações através de ambientes tridimensionais, fazendo uso de bibliotecas gráficas que disponibilizam rotinas para tais finalidades, tais como OpenGL e Java3D.

6.1.1 Estrutura e interface do Arthemis

A estrutura de pacotes e pastas do software Arthemis tem como objetivo facilitar a compreensão do software como um todo. Chama-se de pacotes todos os diretórios que contém arquivos de código fonte da linguagem Java, já os diretórios que não contém nenhum arquivo fonte são chamados de pastas. A seguir, por meio da Figura 6.1 pode ser visualizada a estrutura dos pacotes da ferramenta Arthemis.

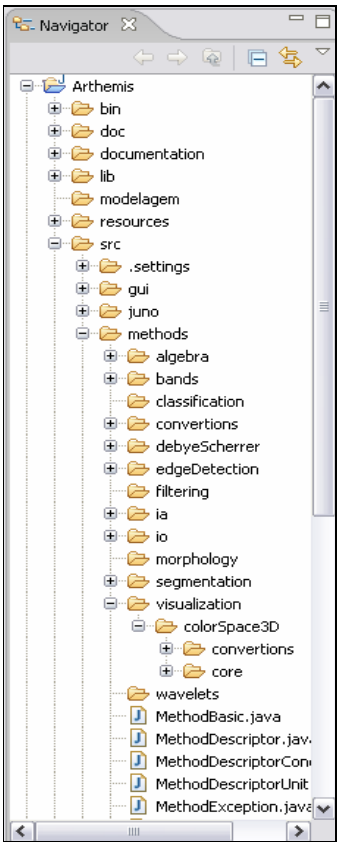

Figura 6.1 – Estrutura dos pacotes do Arthemis

Como se observa na Figura 6.1, o pacote “*src*” é o diretório raiz do código fonte responsável pelas funcionalidades relacionadas ao processamento e análise de imagens do Arthemis. No pacote ”*visualization*” encontra-se o pacote das

conversões dos modelos de cores, chamado "*convertions*". Nesse pacote têm-se os métodos para dar suporte aos seguintes espaços de cores: CIE XYZ, CIE xyY, CIE L*a*b*, CIE L*u*v*, CMY, HSI, YCbCr, YIQ, YUV e RGB que na aplicação assume-se que as imagens carregadas já encontram-se neste espaço. Cabe lembrar que os espaços de cores suportados pela aplicação derivam do espaço RGB.

As equações utilizadas na aplicação de transformações dos espaços de cores são as mesmas que foram apresentadas no Capítulo 2, podendo conter algumas modificações na implementação. Algumas das operações de conversões necessitam de parâmetros tais como: *RGB working space*, que é o espaço de cores absoluto baseado no RGB; o *white point*, que consiste no ponto de luz branca que é utilizado para representação do espaço; o método de adaptação cromática e o campo de visão que são necessários para obter a cromaticidade dos *white points* utilizados. Através da Figura 6.2 é possível visualizar a interface da aplicação com seus parâmetros e uma transformação realizada.

6.1.2 Aplicação da ImageTest no Contexto de um grupo de Pesquisa

A metodologia proposta foi aplicada ao projeto Arthemis, tendo como objetivo avaliar a importância de uma metodologia de testes para o contexto de transformações no espaço de cores, avaliando, dessa forma, a qualidade da aplicação desenvolvida e a validade das técnicas e critérios propostos por ela.

A aplicação da ImageTest ocorreu dentro de um grupo de pesquisa – Grupo de Processamento de Informação Multimídia GPIM, o qual o desenvolvimento da ferramenta Arthemis já se encontrava em andamento. Fizeram parte da equipe de testes dois integrantes do grupo, o gerente de teste que é o responsável pelo desenvolvimento do planejamento dos testes, e o testador que é o responsável pela aplicação direta dos testes na ferramenta.

Ao desenvolver a proposta de metodologia para a aplicação de testes, o projeto não utilizava uma metodologia sistematizada e documentada para testar suas aplicações, bem como não havia atividade de teste fazendo parte do seu desenvolvimento. Assim, à medida que as funcionalidades das aplicações solicitadas são terminadas, elas são alocadas no Arthemis e ficam disponíveis para os demais integrantes do grupo utilizar.

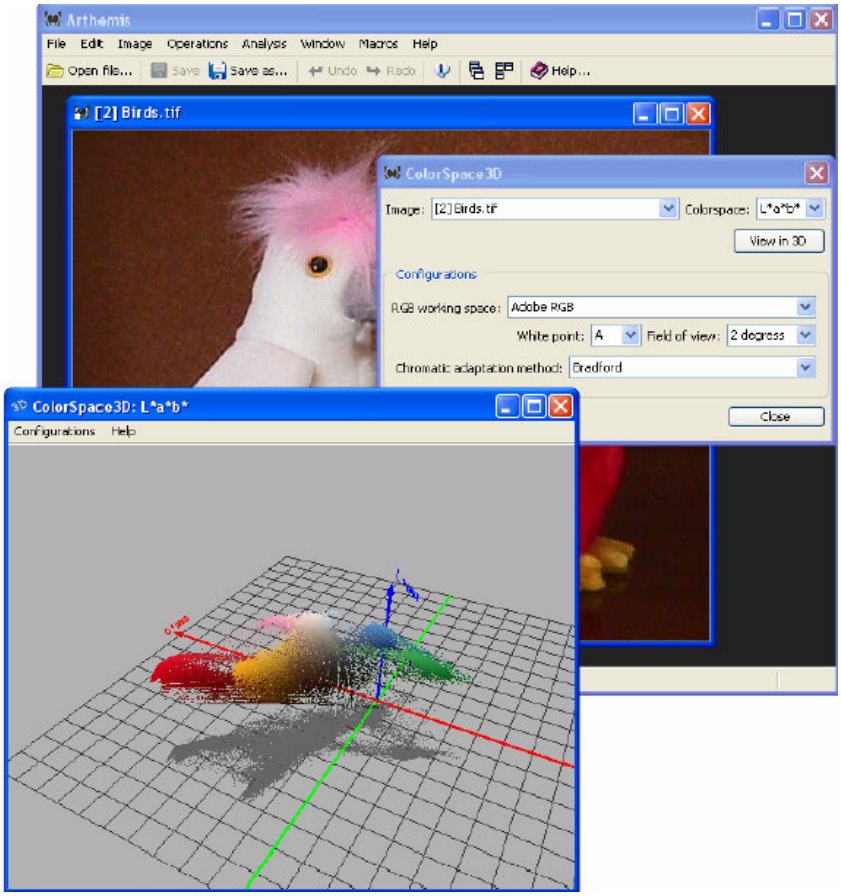

Figura 6.2 – Interface das funcionalidades e seus parâmetros no domínio de transformações no espaço de cores

Dessa forma, o desenvolvimento do Arthemis é realizado de forma incremental. Aos poucos, o gerente do projeto faz o levantamento dos requisitos do sistema, e à medida que essa atividade é realizada, o mesmo passa as informações sobre o que deve ser implementado para os desenvolvedores do grupo. Assim que as aplicações ficam prontas, elas são alocadas a estrutura do Arthemis tornando-se disponível para os demais integrantes do grupo, que podem ser considerados os usuários internos que são alunos e professores, os quais fazem uso da ferramenta para fins didáticos.

A seguir são descritas as 5 etapas realizadas de acordo com a ImageTest.

1. *Treinar:* Nessa fase inicial da metodologia, verificou-se que os dois integrantes que formaram a equipe de testes deveriam realizar treinamento sobre o processo de testes de software e a execução automática dos mesmos, juntamente com as ferramentas que são utilizadas para sua execução.

2. *Verificar situação do projeto:* a estrutura do projeto Arthemis já existia e continha algumas funcionalidades da área de processamento e análise de imagens. Porém o domínio de transformações de cores estava em fase inicial, o qual pode-se realizar o processo de testes desde o início.

3. *Integrar ao estágio atual do projeto:* Como dito anteriormente, o desenvolvimento do projeto Arthemis não possuía nenhuma atividade de teste de software, fazendo com que essa fase não precisasse ser aplicada.

4. *Executar testes necessários:* Essa fase é a mais importante da metodologia. É onde a equipe de testes realizou o seu trabalho de acordo com o andamento do projeto Arthemis. Primeiramente, à medida que a equipe de desenvolvimento fez o levantamento dos requisitos, a equipe de testes estudava esses requisitos e, a partir disso, o gerente de teste elaborou o planejamento dos testes de aceitação. Para isso ele utilizou o relatório de "Plano de teste" baseado na norma IEEE 829. O plano de teste descreve o próprio sistema e o plano para executar todas as funções e características. Ele encontra-se em anexo e aborda os seguintes tópicos:

- nome do sistema;
- nome dos módulos a serem testados;
- local e data de definição do plano;
- data sugerida para término da atividade;
- objetivo do plano;
- resumo do sistema;
- referências a documentos e observações;
- riscos associados com a atividade de teste;
- cronograma de teste com nome das funcionalidades a serem testadas e nome dos testes;
- tempo total,
- data final;
- recursos necessários; e,
- nome do testador.

Após a arquitetura do sistema ser construída, o gerente de teste passa a elaborar o plano de testes funcionais para as funcionalidades que existirão no sistema. Dessa forma, utiliza-se o mesmo relatório que foi descrito anteriormente, preenchendo o plano conforme informações mais precisas sobre as funcionalidades que farão parte do sistema.

Depois de construída a arquitetura do sistema, os desenvolvedores do projeto passaram a codificar a ferramenta. À medida que a implementação foi realizada, as funcionalidades acabadas foram passadas para a equipe de testes. O gerente de teste fez o plano de testes de unidade, colocando todas as informações necessárias. Após fazer o plano, o gerente preencheu o relatório de "Especificação de caso de teste", onde este relatório contém detalhes mais específicos dos testes, como:

- descrição resumida dos testes;
- entrada e resultados esperados;
- procedimentos que podem ser utilizados na execução dos testes; e,
- o nome do gerente responsável.

Posteriormente, o gerente passou o relatório de "Especificação de caso de teste" ao testador para ele finalmente executar os testes de unidade com a ferramenta *JUnit*. O gerente de teste assumiu a responsabilidade de gerenciar e, também testar. Por meio da Figura 6.3 pode-se visualizar um método referente à classe de transformações do espaço de cores CIECalculator do Arthemis, e por conseguinte, na Figura 6.4 o teste relacionado ao método.

```
public float[][] getMatrixRGBtoXZY(String ws, String wp, String method, String fieldOfView) {

    float[][] result = null;
    float[][] M1 = getMatrixRGBtoXZYNoChromAdapt(ws, fieldOfView);

    M1 = MatrixOp.transpose(M1);
    String wpSource = getWorkingSpaceWhitePoint(ws);

    if (!wpSource.equals(wp)) {
        float[][] M2 = getMatrixChromaticAdapt(wpSource, wp, method, fieldOfView);
        result = MatrixOp.multiply_3x3_3x3(M1, M2);
    } else
        result = M1;
    return result;
}
```

Figura 6.3 – Método `getMatrixRGBtoXYZ` responsável por calcular a matriz [M] para a conversão do espaço RGB para CIE XYZ

```
public void testgetMatrixRGBtoXZY(){
    String[] wss = new String[]
            { "Adobe RGB", "Apple RGB",  "Best RGB", "Beta RGB", "Bruce RGB", "CIE RGB",
              "ColorMatch RGB", "DonRGB4", "ECI RGB", "Ekta Space PS5", "NTSC RGB",
              "PAL/SECAM RGB", "ProPhoto RGB", "SMPTEC-C", "sRGB", "Wide Gamut RGB"};
    String[] sources = new String[]{"A", "B", "C", "D50", "D55", "D65", "D75", "E"};
    String[] methods = new String[]{"XYZ Scaling", "Bradford", "Von Kries" };
    String[] fieldsOfView = new String[]{"2 degress", "10 degress"};

    CIECalculator ciecalculator = new CIECalculator();
    for (int i = 0; i < wss.length; i++){
        String ws = wss[i];
        for (int j = 0; j < sources.length; j++){
            String wp = sources[j];
            for (int k = 0; k < methods.length; k++){
                String method = methods[k];
                for (int l = 0; l < fieldsOfView.length; l++){
                    String fieldOfView = fieldsOfView[l];
                    float[][] res;
                    res = ciecalculator.getMatrixRGBtoXZY(ws, wp, method, fieldOfView);
                    float[][] result = null;
                    float[][] M1 = ciecalculator.getMatrixRGBtoXZYNoChromAdapt(ws, fieldOfView);
                    M1 = MatrixOp.transpose(M1);
                    String wpSource = ciecalculator.getWorkingSpaceWhitePoint(ws);
                    if (!wpSource.equals(wp)) {
                        float[][] M2 = ciecalculator.getMatrixChromaticAdapt(wpSource, wp, method,
                                                                             fieldOfView);
                        result = MatrixOp.multiply_3x3_3x3(M1, M2);
                    } else result = M1;
                    assertEquals(result, res);
                }
            }
        }
    }
```

Figura 6.4 – Implementação do teste de unidade realizado sobre o método `getMatrixRGBtoXYZ`

Através da Figura 6.4, pode-se perceber a importância do testes de unidade nesse contexto. É preciso verificar se os parâmetros que estão sendo passados estão realmente retornando os valores esperados. Dessa forma, é necessário testar todas as possibilidades ou valores possíveis dos parâmetros dos métodos para então verificar se o resultado da transformação está correto. Assim, para todas as classe referentes às transformações do espaço de cores, foram realizados testes de unidade nos métodos para verificar se os valores passados para as transformações nos espaços contidos no Arthemis estão sendo passados de forma correta. Os testes de unidade foram os que mais despenderam tempo. Eles foram aplicados pelo testador durante 3 meses e não foram encontrados erros na estrutura interna do sistema.

Sabe-se que apenas com o *JUnit* não é possível identificar qual o percentual de código testado, assim partes importantes podem não ter sido testadas. Para realizar essa identificação foi utilizada a ferramenta *Cobertura*. Após a execução dos testes de unidade, o relatório “Diário de teste” foi preenchido pelo testador.

Através da Figura 6.5, pode-se visualizar a chamada da execução da ferramenta *Cobertura* no projeto Arthemis. Logo, a Figura 6.6 exibe o relatório de cobertura, revelando níveis de detalhe do projeto como um todo.

```
<!-- Execute coverage tests -->
<!-- <target name="cobertura" depends="unitTests-optional" -->
<target name="cobertura" depends="compile"
        description="execute Arthemis coverage tests">
    <delete file="${basedir}/cobertura.ser" />

    <cobertura-instrument todir="${bin.instrumented}" >
        <fileset dir="${bin.methods.visualization.colorSpace3D.convertions}" >
            <include name="**/*.class" />
            <exclude name="**/*Test*.class" />
        </fileset>
    </cobertura-instrument>
        <java fork="yes" classname="junit.textui.TestRunner" taskname="unit" >
            <sysproperty key="net.sourceforge.cobertura.datafile" file="${basedir}/cobertura.ser" />
            <classpath location="${bin.instrumented}" />
            <classpath location="${bin}"/>
            <classpath refid="project.classpath" />
            <arg line="test.AllTests" />
        </java>

    <cobertura-report destdir="${documentation.coverage}" >
        <fileset dir="${src.methods.visualization.colorSpace3D.convertions}" >
            <include name="**/*.java" />
        </fileset>
        <fileset dir="${src.test}" >
            <include name="**/*.java" />
        </fileset>
    </cobertura-report>
</target>
```

Figura 6.5 – Descrição da chamada a ferramenta *Cobertura*, localizada no arquivo `build.xml` do Arthemis.

O arquivo `build.xml` encontra-se no projeto Arthemis e foi criado para fazer a chamada a ferramenta *Cobertura*. Ele possui várias *target* (ou alvo) referenciando o local do projeto, as pastas onde os relatórios devem ser gerados, entre outras informações necessárias para sua compilação. Na Figura 6.5 foi exibida apenas a *target* que faz referência ao caminho dos testes e a ferramenta Cobertura. Para apoiar o projeto Arhtemis, além da *target* exibida, o projeto define outros alvos descritos, tais como: 1) *clean:* exclui todos os arquivos presentes nos diretórios; 2) *compile:* compila todo o código fonte presente nos pacotes *"src"* (dos métodos) e *"test"* (pacote dos testes) para a pasta de classes compiladas *"bin"*; 3) *javadoc:* gera páginas *HTML* de documentação da *API* no estilo *javadoc* ; 4) *uniTests:* executa todos os testes unitários do Arthemis, criados com o *JUnit*; 5) *cobertura:* executa os testes de cobertura sobre todos os arquivos fonte dos pacotes especificados, utilizando a ferramenta *Cobertura*; 6) *all*: executa de forma seqüencial todos as *targets* (alvos) definidos.

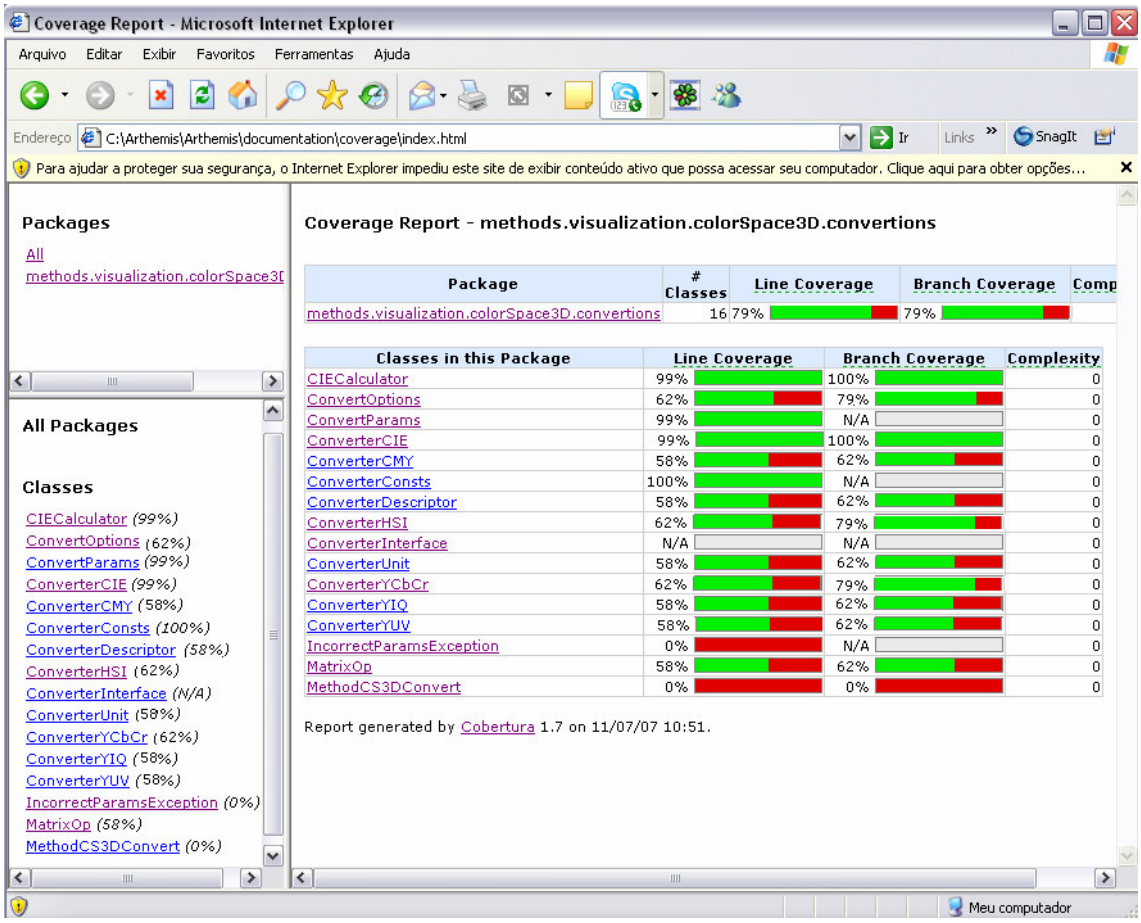

Figura 6.6 – Relatório dos testes de cobertura realizados pela ferramenta *Cobertura*.

Através do relatório gerado pela ferramenta *Cobertura*, pode-se verificar quantas classes existem no pacote testado e qual a porcentagem dessas classes que foram testadas. Além disso, pode-se selecionar as classes exibidas no relatório geral, a qual irá abrir um novo relatório contendo o código da classe mostrando em vermelho o que não foi testado e em verde o que foi testado. Dessa forma, a ferramenta *Cobertura* utiliza o critério todos-nós, verificando a execução dos testes linha a linha. Em diversos arquivos do código fonte, existem trechos de código que estavam comentados, alguns que realizavam a impressão do texto no console (onde se pode solicitar a execução da aplicação), outros trechos forneciam informações sobre as tarefas dos métodos, onde esses trechos não são executados pela *Cobertura*.

Após a execução dos testes de unidade, foram executados os testes funcionais, os quais são baseados nos requisitos do sistema, e foram feitos a partir dos critérios particionamento de equivalência e análise de valor limite, utilizando a

ferramenta *JUnit*. A tabela a seguir mostra as classes de equivalência e análise de valor limite para a realização dos testes.

Critério: Particionamento de equivalência		
Operação	**Classes válidas**	**Classes inválidas**
Arquivo	Imagem com 3 canais de bandas	Imagem com um canal
Nome do arquivo	Tamanho > 0	Tamanho = 0
Extensões do arquivo	tif, gif, jpg, pnm, png, bitmap	Txt, doc, pdf, ppt
Conversores	CIE XYZ, CIE xyY, CIE L*a*b* e CIE L*u*v*	Não selecionar o conversor desejado

Tabela 6.1: Critério de particionamento de equivalência

Critério: Análise de valor limite		
Operação	**Entrada**	**Saída**
Tamanho arquivo	1300 x 1030	Mostrar gráfico de conversão
Valores arquivo	0 a 255	0 a 255
Extensões do arquivo	TIF, TIFF, GIF, JPEG, JPG, BITMAP, PNM, PNG	Abertura da imagem
Conversores	CIE XYZ, CIE xyY, CIE L*a*b* e CIE L*u*v*	Imagem com respectivo espaço de cor convertida

Tabela 6.2: Critério de análise de valor limite

Através desses dois critérios complementares da técnica funcional, o testador realizou os casos de testes na ferramenta Arthemis. Esses testes foram realizados à medida que o os testes de unidade ficavam prontos. Para cada classe obteve-se em média 4 casos de testes que levaram cerca de 2 meses para serem concluídos. Como resultados desses testes, obteve-se uma falha referente às conversões nos espaços de cores quando o arquivo (imagem) possui uma matriz de 1300x1030 *pixels*. Neste caso foi passado aos desenvolvedores o "Diário de teste" contendo informações relevantes para que a funcionalidade fosse corrigida.

Por fim, os testes de aceitação foram realizados por alguns usuários internos do grupo GPIM que detinham o conhecimento das funcionalidades no domínio de transformações no espaço de cores. Essa avaliação foi realizada quando as funcionalidades já estavam integradas no sistema, e ela tem como objetivo avaliar tais funcionalidades, identificar problemas relacionados a interface, desempenho e algumas sugestões de melhorias para a utilização do software.

O software foi utilizado pelo usuário, no laboratório do grupo GPIM, registrando os dados dos testes no "Diário de teste" juntamente com o testador acompanhando e registrando os erros e problemas de uso. Com a realização dos testes de aceitação pode-se observar algumas sugestões identificadas pelos usuários do sistema:

- melhorar o desempenho das transformações em imagens iguais ou superiores a 1300x1030 *pixels*, onde as transformações nestes casos não estão ocorrendo;
- no "histograma", deve-se ter a opção de aumento de escala para melhor visualização dos dados;
- padronização da linguagem utilizada na interface;
- o botão "`Ver em 3D`" poderia ser localizado na parte inferior da tela abaixo da caixa de configurações;
- fazer com que as transformações referentes às conversões L*a*b* e L*u*v* sejam mais eficazes, elas estão muito demoradas relacionadas as demais;

5. *Imprimir relatórios de testes:* essa etapa foi realizada após cada atividade relacionada ao planejamento até a execução dos testes. Dessa forma, todos os

relatórios de testes realizados ficam guardados juntamente com os demais arquivos do projeto como forma de permitir o reuso.

7 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. Essa atividade, no entanto, consome grande parte dos custos envolvidos durante o desenvolvimento e, apesar disso, não garante um produto completamente livre de erros.

Procurando reduzir o custo e o tempo associado ao teste de software, bem como, aumentar sua eficácia, várias técnicas e critérios têm sido propostos (Domingues, 2002). Devido ao aspecto complementar dessas técnicas, o testador deve aplica-las em conjunto para obter um teste de boa qualidade. O uso de ferramentas de teste também contribui para a redução dos custos associados a essa atividade, além de evitar que erros sejam introduzidos pela intervenção humana durante a realização dos testes.

Tendo como desafio propor uma metodologia para aplicar ao domínio de transformações no espaço de cores, a principal contribuição do trabalho é a própria metodologia que apresenta uma estratégia para aplicar testes nesse domínio, a qual foi aplicada em um grupo de pesquisa que trabalha com a área de processamento de imagens. O objetivo dessa proposta foi auxiliar o processo da atividade de teste tornando-a presente em todo o ciclo do desenvolvimento do software. Com essa finalidade, buscou-se aplicar as técnicas e critérios existentes na literatura para o domínio, assim como ferramentas disponíveis que suprissem as técnicas de teste escolhidas visando custos mínimos com a atividade.

Com a aplicação da metodologia em um grupo de pesquisa que desenvolve software no domínio de transformações no espaço de cores, os quais foram obtidos bons resultados, podê-se concluir que:

- a metodologia é viável de ser aplicada em um grupo de pesquisa, como parte do processo de desenvolvimento do software;
- a técnica estrutural é necessária para validar os métodos desenvolvidos que realizam o processo de transformações no espaço de cores, bem como a técnica funcional que tem a finalidade de verificar o resultado das funcionalidades;

- o uso da ferramenta *JUnit* juntamente com a *Cobertura* foi muito importante para a execução automática dos testes e obtenção de relatórios informando quanto de cada classe foi testada;
- o processo de teste implantado pela metodologia gera melhorias visíveis aos clientes e desenvolvedores, melhorando a qualidade do software por fazer com que o cliente interaja juntamente com o ciclo do desenvolvimento;
- o processo padrão de documentação possibilita o reuso dos testes, visto que o sistema pode estar sendo modificado constantemente;

A ImageTest ajuda a estruturar as atividades de teste desde o início de um projeto de desenvolvimento de software, tornando claro para todas as partes envolvidas o relacionamento entre o processo de desenvolvimento de software e as atividades de teste. A definição pela metodologia dos principais artefatos do processo de teste, baseado na Norma IEEE 829, é o principal orientador das atividades de teste.

7.1 Trabalhos Futuros

Como continuidade deste trabalho e como forma de obter um maior grau de precisão dos experimentos realizados, pode-se destacar:

- Modificar a metodologia no sentido de agilizar o processo dos testes de unidade fazendo com que os próprios desenvolvedores realizem os testes unitários visto que já detém o conhecimento da estrutura interna do sistema;
- Aplicar a metodologia proposta em um projeto de desenvolvimento de software que trabalhe com processamento de imagens e transformações no espaço de cores no contexto de uma empresa;
- Utilizar a ferramenta *Jabuti*, desenvolvida na USP de São Paulo, a qual faz uso de diversos critérios da técnica estrutural, podendo então verificar diversas condições, diversos laços, tratamento de exceções entre outras práticas da técnica estrutural.

REFERÊNCIAS

ALBUQUERQUE, M. P; ALBUQUERQUE, M. P. ***Processamento de imagens: métodos e análises.*** Centro Brasileiro de Pesquisas Físicas – CBPF/MCT, Coordenação de Atividades Técnicas – CAT, RJ, 2000.

ANDRADE, M. S. S; ***Utilização de teste estrutural para aperfeiçoar a avaliação de um sistema de auxílio ao diagnóstico.*** Dissertação de Mestrado, Centro Universitário Eurípedes de Marília, Marília, SP, 2005.

AGRAWAL, H.; ALBERI, J.; HORGAN, J. R.; LI, J.; LONDON, S.; WONG, W.E.; GHOSH, S; WILDE, N. ***Mining system tests to aid software maintenance***. IEEE Computer, p. 64-73, 1998;

AMBROSIO, A. M. ***CoFI – uma abordagem combinando teste de conformidade e injeção de falhas para validação de software em aplicações espaciais***. Tese de doutorado, INPE, São José dos Campos, 2005.

BEIZER, B. ***Software testing techniques***, 2nd ed., Van Nostrand Reinhold Co., New York, NY, 1990.

BEIZER, B. ***Black box testing: techiniques for functional testing of software and systems.*** John Willey, 1995.

BINDER, R. V. ***Design for testability in object oriented systems.*** Communications of the ACM, v. 37, n. 9, p. 87-101, 1994a.

BUSCHMANN, F. et al. ***Pattern – oriented software architeture: a system of patterns.*** New York: John Wiley & Sons Ltd, 1996.

BUDD, T. A. ***Mutation analysis: Ideas, example, problems and prospects, cáp.*** Computer Program Testing North-Holand Publishing Company, 1981.

CAGNIN, M. I. ***Parfait: uma contribuição para reengenharia de software baseada em linguagens de padrões e frameworks.*** 2005. 241f. Tese (Doutorado em Ciências da Computação e Matemática Computacional) – Universidade de São Paulo, São Carlos, 2005.

CHAIM, M. L. ***Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados.*** Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, SP, 1991;

CLARK, M. ***JUnit Primer***. *Web Journal*, 2005.

COLANTONI, P. ***Color space transformations.*** Disponível online em: http://colantoni.nerim.net/download/colorspacetransform-1.0.pdf. [02/02/2007].

CORTÊS, M.L., CHIOSSI, T. C. S. ***Modelos de Qualidade de Software.*** Editora da Unicamp, Instituto de Computação, Campinas, SP, 2001.

CRESPO, A. N.; SILVA, O. J.; BORGES, C. A.; SALVIANO, C. F.; ARGOLLO, M.; JINO, M. ***Uma metodologia para teste de Software no Contexto da Melhoria de Processo,*** In: III Simpósio Brasileiro de Qualidade de Software (SBQS 2004), Brasília, 2004.

DELAMARO, M. E. ***Aspectos teóricos e empíricos de teste de cobertura de software.*** Relatório Técnico 31, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, 1998.

DEMILLO, R. A. ***Software testing and evaluation.*** The Benjamin/Cummings Publishing Company, Inc., 1987.

DOMINGUES, A. L. dos S. ***Avaliação de critérios e ferramentas de teste para programas OO*****.** Dissertação, Instituto de São Carlos – USP. São Paulo, 2002.

FACON, J. ***Processamento e análise de imagens.*** Curitiba: PUC, Feb 2002.

FRANKL, P. G.; WEYUKER, E. J. ***Testing Software to Detect and Reduce Risk.*** Journal of Systems and Software, v. 53, n. 3, p. 275-286, 2000.

FREITAS, G. D. ***IgraPI: Interface Gráfica da Ferramenta para o Processamento de Imagens StoneAge.*** Dissertação de Mestrado – Programa de Pós-Graduação em Engenharia de Produção – Universidade federal de Santa Maria, 2004.

WHITTAKER, J.A. ***"What Is Software Testing? And Why Is It So Hard?",*** IEEE Software, January/February, 2000.

GAMMA, E. et al***. Padrões de projeto – soluções reutilizáveis de software orientado a objetos.*** Porto Alegre: Bookman, 2000.

GAMMA, E.; BECK, K. ***JUnit, Testing Resources for Extreme Programming.*** 2002. Disponível em: http://www.junit.org/ [16/11/2005].

GIMENES, I. M. DE S.; HUZITA, E. H. M. ***Desenvolvimento baseado em componentes: conceitos básicos e técnicas.*** Rio de Janeiro: Editora Ciência Moderna. 2005.

GONZALES, R. C.; WOODS, R. E. ***Processamento de imagens digitais.*** São Paulo: Edgard Blücher, 2000.

GONZALEZ, R. C.; WOODS, R. E. ***Digital Image Processing***. 2.ed. Upper Saddle River, New Jersey: Prentice Hall, 2002. cap 6.

HARROLD, M. J. ***Testing: A roadmap***. In: 22th International Conference on Software Engineering – Future of SE Track, 2000, p. 61-72.

HERBERT, J. S.; PRICE, A. M. A. ***Técnicas de testes de software orientado a objetos.*** Laboratório de Qualidade de Software, Instituto de Informática. São Leopoldo: UNISINOS, 2000.

HOFFMAN, D.; STROOPER, P. ***Classbrench: A framework for autmated class testing.*** Software Practice and Experience, p. 573-597, 1997.

HOFFMAN, G. ***CIE Color Space.*** Disponível online em: http://fhoemden.de/~hoffmann/ciexyz29082000.pdf. Acesso em 05 fev. 2007.

HOWDEN, W. E. ***Theoretical and empirical studies of program testing***. IEEE Transactions on Software Engineering, v. 4, n. 4, p. 293-298, 1978.

IEEE Computer Society; ***IEEE Std 829: Standard for Software Test Documentation.*** New York, September, 1998.

ICC. [Introduction to the ICC profile format]. Website do ICC. Disponível online em: http://www.color.org/iccprofile.html. [07/02/2007].

IMA. ***IMAGE processing toolkits and program***. Disponível online em: http://image.soongsil.ac.kr/tools_mis.html. [12/07/2005].

INTHURN, C. ***Qualidade e Teste de Software.*** Ed. Visual Books, Florienópolis – SC, 2001.

KOSCIANSKI, A.; SOARES, M. S. ***Qualidade de Software.*** 1 ed. Editora Novatec, São Paulo, 2006.

LEMOS, O. A. L. ***Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ.*** Dissertação, Instituto de Ciências Matemáticas e Computação – ICMC/USP, 2005.

LINDBLOOM, B. J. [BruceLindbloom.com]. Disponível online em: http://brucelindbloom.com. [02/02/2007].

LINNENKUGEL, U.; MÜLLERBURG, M. ***Test selection criteria for (software) integration testing.*** In: First Iternational Conference on System Integration, Morristown, NJ, 1990, p.709-717.

MALDONADO, J. C. ***Critérios potenciais usos: Uma contribuição ao teste estrutural de software.*** Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. ***Seleção de casos de testes baseada na análise de fluxo de dados através dos critérios potencias usos.*** In: IISBES – Simpósio Brasileiro de Engenharia de Software, Canela, RS, 1998, p. 24-35.

MALDONADO, J. C. FABBRI, S. C. P. F. ***Teste de software.*** In: DA ROCHA, A. R. C.; MALDONADO, J.C; WEBER, K. C., eds. Qualidade de Software – Teoria e Prática, 1 ed, São Paulo: Prentice Hall, p. 73-84, 2001.

MALDONADO, J. C. et al. ***Teste de software: teoria e prática***, In: Minicurso – XVII Simpósio Brasileiro de Engenharia de Software (SBES 2003). Manaus – AM: [s.n], 2003.

MARK, D. ***Cobertura.*** Disponível online em: http://cobertura.sourceforge.net/ [01/07/2006].

MARTIN, James, McCLURE, Carma. ***Técnicas estruturadas e Case***. São Paulo: Makron, McGraw-Hill, 1991.

MASSOL, V.; HUSTED, T. ***JUnit em Ação***. Ed. Ciência Moderna Ltda, Rio de Janeiro, 2005.

MEDEIROS, J. D. R. V. ***Uma implementação da Norma ISO 9001: 2000 para Implementação em Empresas de Software***. Dissertação de Mestrado, Universidade Federal de Pernambuco, Centro de Ciências Exatas e da Natureza, Departamento de Informática, Recife- PE, 2001.

MEYER, F. ***Topographic distance and watershed lines.*** Signal Processing, v.38, p.113-125, 1994.

MYERS, G. J. ***The art of software testing*****.** Wiley, New York, 1979.

MYERS, G. J. ***The art of software testing***. Second ed. Wiley, 2004.

MYERS, G. J. ***The art of software testing***, cap. Extreme Testing. In: (Myers, 2004a), p.177-191, 2004b

MOLINARI, L. ***Testes de software - Produzindo Sistemas Melhores e mais Confiáveis.*** 2 ed. Editora Érica, São Paulo, 2003.

NARDI, P. A.; DELAMARO, M. E.; SPOTO, E. S.; VINCENZI, A. M. R. ***JaBUTI/BD: Utilização de Critérios Estruturais em Aplicações de Banco de Dados Java.*** Simpósio Brasileiro de Engenharia de Software, Uberlândia, MG, 2005.

OFFUT, A. J.; IRVINE, A. ***Testing object-oriented software using the category-partition method.*** In: 17th Iternational Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, CA, 1995, p. 239-304.

OSTRAND, T. J.; BALCER, M. J. ***The category-partition method for specifying and generating functional tests.*** Communications of the ACM, v. 31, n. 6, p. 676-686, June 1998.

PFLEEGER, S. L. ***Software engineering: The production of quality softwre***. Second Ed., Macmillan, New York. In: Belchior, A. D., 1991,

PARASOFT CORPORATION. ***Jtest.*** Disponível online em http://www.parasoft.com/jtest [05/06/2006].

PRATT, W. K. ***Digital image processing: PIKS inside.*** 3.ed. New York: John Wiley and Sons, 2001.

PRESSMAN, R. S. ***Engenharia de software.*** A Practioner´s Approach – McGraw-Hill Co., Inc. Fourth Edition, 1997.

PRESSMAN, R. S. ***Software engineering – a practitioner's approach.*** 5 ed. McGraw-Hill, 2000.

PRESSMAN, R. S. ***Software engineering - a practitioner's approach*****.** 5 ed. McGrawHill, 2001.

PRESSMAN, R. S. ***Engenharia de software.*** 5 ed. McGrawHill, Rio de Janeiro, 843 p., 2002.

RATIONAL CORP. IBM Rational. ***XDE Tester***. Disponível online em http://www-136.ibm.com/developerworks/rational/products/xdetester [09/07/2006].

ROCHA, A. R., OLIVEIRA, K. M., Jr., A. R. ***Qualidade de Software Médico.*** IV Simpósio Argentino de Informática e Saúde – SADIO, Argentina, 2001.

ROCHA, A. D. ***Uma ferramenta baseada em aspectos para apoio ao teste funcional de programas Java*****.** Dissertação apresentada ao instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, 2005.

ROPER, M. ***Software testing.*** The International Software Engineering Series. McGraw-Hill, 1994.

RUSS, J. C. ***The image processing handbook.*** 3.ed. Raleigh, North Carolina: CRC Press LLC, 1999.

SHAPIRO, L; STOCKAMN, G. ***Computer vision.*** New Jersey: Prentice Hall, 2001.

SOMMERVILLE, I. ***Engenharia de software.*** São Paulo: Addison-Wesley, 2004.

SZE, L. ***Atacobol: A cobol test coverage analysis tool and its applications*****.** In: ISSRE'2000 – International Symposium on Software Reliability Engineering, San Jose, CA, 2000.

THE APACHE ANT PROJECT. ***ANT.*** Disponível online em: http://jakarta.apache.org/ant/ [01/08/2006].

TSUKUMO, A.N., et al. ***Qualidade de Software: Visões de Produto e Processo de Software.*** In: VIII CITS – Conferencia Internacional de Tecnologia de Software: Qualidade de Software, Curitiba, Paraná, Brasil. Anais, 1997.

UTOMATEDQA CORP. ***TestComplete***. Disponível online em http://www.automatedqa.com/products/testcomplete/index.asp [07/07/2006].

VINCENZI, A. M. R. ***Subsídios para o estabelecimento de estratégias de teste baseadas na técnica de mutação****.* Dissertação de Mestrado, ICMC-USP, São Carlos – SP, 1998.

VINCENZI, A. M. R. ***Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação.*** Tese de doutorado, ICMC-USP, São Carlos – SP, 2004.

WELFER, D. ***Padrões de projeto no desenvolvimento de sistemas de processamento de imagens.*** 2005. Dissertação de Mestrado – Programa de Pós-Graduação em Engenharia de Produção (PPGEP) – Universidade Federal de Santa Maria.

WOOD, L. E. ***User interface design – bridging the gap from user requirements to design.*** Utah: CRC Press LLC, 1999.

.

APÊNDICE A – Plano de Teste

PLANO DE TESTE	
Cabeçalho **Sistema:**	
Nome dos módulos a serem testados:	
Plano	
Nº plano:	
Local da definição:	
Data da definição:	Data sugerida para término:
Gerente responsável:	
Objetivo do plano	
Detalhamento – Resumo do Sistema	
Referências a documentos e observações	
Riscos associados com a atividade de teste	

Cronograma de testes		
Nome métodos/funcionalidades a serem testados	Nome dado ao teste	Nº teste
Tempo total:		
Data inicial:	Data final:	
Recursos necessários:		
Nome Testador:		

APÊNDICE B – Especificação de Caso de Teste

ESPECIFICAÇÃO DE CASO DE TESTE		
Cabeçalho		
Sistema:	Nº plano:	Nº especificação:
Nome módulo(s) a ser testado:		

Descrição resumida dos testes	
Nº teste	Descrição resumida do teste

Resultados esperados		
Nº teste	Entrada	Resultado esperado

Procedimentos utilizados

Gerente responsável

APÊNDICE C – Diário de Teste

DIÁRIO DE TESTE		
Cabeçalho		
Sistema:	Nº diário teste:	Nº especificação:
Nome do módulo testado:		
Fase do sistema: Implementação [ ] Validação [ ]		
Data:	Tempo gasto:	

N° teste	**Nome do método/funcionalidade testado**	**Tipo de teste realizado**

Resultados obtidos		
N° teste	**Entrada**	**Resultado obtido**

Descrição de erros encontrados	
N° teste	**Descrição do erro**

Testador responsável

APÊNDICE D – Relatório Resumo de Teste

RELATÓRIO - RESUMO DE TESTE	
Cabeçalho	
Sistema:	
Nº plano teste:	
Nº especificação teste:	
Nº diário(s) teste:	
Nome dos módulos testados:	
Nome dos métodos testados	**Breve resultado dos testes aplicados**
Nome dos métodos/funcionalidades com defeitos	
Riscos associados	
Testador responsável	