FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL ALVES PAES DE OLIVEIRA

ESTRUTURA PARA UTILIZAÇÃO DE CBIR EM ORÁCULOS GRÁFICOS

MARÍLIA
2008

RAFAEL ALVES PAES DE OLIVEIRA

ESTRUTURA PARA UTILIZAÇÃO DE CBIR EM ORÁCULOS GRÁFICOS

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. MÁRCIO EDUARDO DELAMARO

MARÍLIA
2008

OLIVEIRA, Rafael Alves Paes

Estrutura para utilização de CBIR em oráculos gráficos / Rafael Alves Paes de Oliveira; orientador: Prof. Dr. Márcio Eduardo Delamaro. Marília, SP: [s.n], 2008.

87 f.

Trabalho de Conclusão de Curso (Graduação em Bacharel em Ciência da Computação) – Curso de Bacharelado em Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2008.

1. Engenharia de *Software* 2. Teste de *Software* 3. CBIR

CDD: 005.14

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Rafael Alves Paes de Oliveira

ESTRUTURA PARA UTILIZAÇÃO DE CBIR EM ORÁCULOS GRÁFICOS

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10 (Dez)

Orientador: Márcio Eduardo Delamaro

1°. Examinador: Fátima L. S. Nunes Marques

2°. Examinador: Elvis Fusco

Marília, 21 de novembro de 2008.

AGRADECIMENTOS

A DEUS por ter me dado a chance de estudar, saúde para trabalhar/pesquisar e a capacidade de aprender com meus erros.

À minha mãe pelas incessantes doses de carinho, atenção, confiança e amor.

Ao CNPQ (processo nº 551002/2007-7) - Conselho Nacional de Desenvolvimento Científico e Tecnológico pelo apoio financeiro dado a este trabalho.

Ao meu orientador Delamaro e à professora Fátima pela oportunidade, confiança depositada, investimento, crédito desprendido e principalmente pelos muitos ensinamentos. Muito Obrigado.

A gratidão é o mais nobre sentimento humano, base para todos sentimentos bons. MUITO OBRIGADO!

OLIVEIRA, Rafael Alves Paes. **Estrutura para utilização de CBIR em Oráculos Gráficos.** 2008. 87 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

RESUMO

Neste trabalho é apresentado o protótipo de um sistema cujo objetivo é evidenciar a possibilidade de explorar a visão computacional aplicada na Recuperação de Imagem Baseada em Conteúdo (CBIR), para a confecção de oráculos de teste destinados a *software* que geram saídas gráficas. Oráculos são mecanismos que servem para decidir se a saída ou comportamento de um programa em teste estão corretos ou não. CBIR é a técnica para recuperar imagens de um banco de dados seguindo juízos de similaridade com uma imagem modelo.Usando bibliotecas da linguagem de programação Java, uma estrutura que possibilita a aliança de duas áreas tão díspares no cenário atual da Computação foi implementada. Grande flexibilidade foi atribuída à estrutura, o que possibilita a intervenção do usuário no modo de análise das imagens. Nesse contexto é possível que tal usuário escolha as características que devem ser extraídas das imagens e como as quais devem ser consideradas no teste, criando assim um ambiente que foi batizado de "*Oráculo Gráfico*".

Palavras-chave: Teste de *Software*. Engenharia de *Software*. CBIR

OLIVEIRA, Rafael Alves Paes, **Estrutura para utilização de CBIR em Oráculos Gráficos.** 2008. 87 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

ABSTRACT

This paper presented a prototype of a system whose goal is to highlight the opportunity to explore computer vision applied in the Content-based Image Retrievial (CBIR), in order to testing oracles *software* that generate graphical output. Oracles are mechanisms used to decide whether to leave or behavior of a program to test are correct or not. CBIR is the technique to retrieve images from a database using judgments of similarity with an image model. Using libraries of Java programming language Java, a structure that allows the alliance of two such disparate areas in the current scenario of Computing has been implemented. A great flexibility was given to the structure, which enables the user intervention in order to analyze the images. In this context it is possible such a user choose the characteristics that must be extracted from the images and how these should be considered in the test, thus creating what was called 'Graph Oracle'.

Keywords: *Software* Testing. *Software* Engineering. CBIR.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API: Application Program Interface

BNF: Backus-Naur Form (Forma de Backus-Naur)

CAD: Computer-Aided Diagnosis (Auxílio Computadorizado ao Diagnóstico)

CBIR: Content-Based Image Retrieval (Recuperação de Imagem Baseada em Conteúdo)

GLC: Gramática Livre de Contexto

GUI: Graphical User Interface

IEEE: Institute of Electrical and Electronics Engineers

JAI: Java Advanced Imaging

LAPIMO: Laboratório de Análise e Processamento de Imagens Médicas e Odontológicas

PAC: Picture Archiving and Communication system

SMA: Segundo Momento Angular

SQA: *Software* Quality Assurance (Garantia de Qualidade de *Software*)

SRIM: Sistema de Recuperação de Imagens Mamográficas

VV&T: Validação, Verificação e Teste

XML: eXtensible Markup Language

SUMÁRIO

INTRODUÇÃO

Para que o trabalho a ser apresentado seja considerado um avanço no universo da Computação é necessário que se aborde a evolução dos microcomputadores. Diz-se isso porque os computadores pessoais da década de 70 eram utilizados apenas por empresas que tinham altas condições financeiras para tal aquisição. Em meados da década de 80, as maiores empresas que desenvolviam *software* investiram em novos modelos de microcomputadores contendo sistemas com interface gráfica. Começava aí a popularização do computador pessoal, que hoje permite a inclusão digital de milhões de pessoas em todo o mundo.

Como tudo que é visível aos nossos olhos torna-se objetivamente mais agradável e interessante quando comparado a algo que nos transmita abstração, nota-se a evolução positiva que interfaces gráficas têm proporcionado ao homem moderno. Partindo desse princípio, chega-se até a evolução dos sistemas que têm como finalidade o processamento gráfico, comuns em diversas áreas da ciência. Em especial pode-se citar a área médica, que ganhou muito conforto com os esquemas CAD (Computer-Aided Diagnosis), usados com a finalidade de auxiliar diagnósticos médicos (GATO et al., 2004; SANTOS, 2006).

Tais sistemas, não somente aqueles que compõem os esquemas CAD, mas quaisquer *software*s com algum processamento gráfico, têm o acerto como característica fundamental (OLIVEIRA et al., 2008). É essencial que tais programas forneçam alto grau de confiança a seus usuários. Em vista disso, faz-se necessário que sejam implantadas técnicas de teste de *software* em seu processo de desenvolvimento.

Objetivos e Justificativas

O presente trabalho é inserido nesse contexto e se enquadra na proposta do projeto "*Definição de Oráculos de Teste para programas com saída gráfica usando Recuperação Baseada em Conteúdo*", descrito em Delamaro (2007a – processo CNPQ nº 551002/2007-7). Sua idéia principal é a construção de um paradigma de estrutura que use a Recuperação de Imagem Baseada em Conteúdo, referida no trabalho apenas pela sigla CBIR (*Content-Based Image Retrieval*), para compor critérios de teste de *software*, ou melhor, Oráculos de Teste capazes de testar o produto final de *software*s que tenham saída gráfica, avaliando seu funcionamento de uma forma objetiva.

O CBIR é usado para fazer buscas em uma base de imagens por meio de juízos de

similaridade (SANTOS, 2006). Juízos de similaridade entre imagens são obtidos por meio da extração de suas características para serem comparadas com as mesmas características de outras imagens (processo de indexação) por funções de similaridade específicas (MOURA et al., 2003). Em linhas gerais, tem-se uma imagem de consulta qualquer da qual se podem extrair características e compará-las com as características das imagens armazenadas em uma base de dados, seguindo um critério, e recuperando as que lhe forem convenientes.

Segundo Hoffman (2001), oráculo é uma estrutura que se utiliza para definir a saída ou a conduta esperada de uma execução. Sendo assim pode-se referir a oráculo como sendo o mecanismo que define e dá um veredicto acerca da correção de uma execução de um programa em teste. Num ambiente de desenvolvimento e teste, um oráculo pode assumir diversas formas e deve basear-se nas particularidades do programa a ser testado.

O fato que deixa o trabalho complexo e incomum se resume na dificuldade encontrada em definir oráculos de teste para programas com processamentos não usuais dos testes. Isso implica dizer que a definição de oráculos para testar programas com processamento gráfico, é uma atividade complexa (OLIVEIRA et al., 2008.; DELAMARO, 2007a).

O foco deste projeto é, então, a construção de um paradigma de ambiente flexível o bastante para possibilitar a instalação de objetos (‘extratores de características de imagens’ e ‘funções de similaridade entre imagens’) a qual viabilize ao testador escolher a função de similaridade e as características que serão consideradas em um teste, criando seus próprios oráculos. Assim a realização de um teste, quando utilizar esse oráculo, dependerá diretamente do processamento de imagens para comparar o resultado da execução de um programa. Como tal processamento deve ser realizado é descrito no oráculo gerado pelo usuário.

Entende-se o porquê de se considerar a idéia do trabalho como um passo que segue as tendências naturais dos avanços computacionais. Como a Computação como um todo migrou para o “universo” gráfico, a inserção e parceria das atividades automatizadas de teste de *software* com tal mundo não podem ser vistas com olhos de surpresa.

O custo da atividade de um bom teste de *software* realizado, em qualquer tipo de aplicação, por uma série de fatores, sempre é muito alto (DELAMARO et al., 2007b), haja vista a dificuldade de automatização de tais atividades. Por a mecanização de funcionalidades testadoras fiéis não ser uma tarefa de simples realização, isso se reflete em alto custo. A automatização de oráculos de teste não é uma tarefa comum e vários trabalhos têm sido divulgados sobre o tema (DELAMARO et al., 2008).

Neste contexto, a proposta de trabalho apresentado se valoriza, pois nela são utilizados conceitos de CBIR, que se resumem em um processo que, por meio de técnicas de

processamento de imagens, é capaz de extrair características de imagens e utilizá-las em buscas por similaridade (GATO et al., 2004), para auxiliar nas atividades de teste, permitindo a automatização de *"ORÁCULOS GRÁFICOS"*. *Oráculo gráfico* é definido por Delamaro et al. (2008) como o ambiente que utiliza técnicas de processamento de imagens para confrontar o resultado de execução de um programa na forma gráfica, com o resultado esperado também na forma gráfica. Enquadra-se, pois como um tópico de automatização dos mecanismos de oráculos.

Conclui-se que este protótipo vem para contribuir com a definição de técnicas e ferramentas que possibilitem ou facilitem a automatização do mecanismo conhecido como oráculo, que é classificado como componente das atividades coletivamente chamadas de "VV&T" ou Validação, Verificação e Teste (MALDONADO et al., 2004).

Disposição do Trabalho

O trabalho está dividido nos seguintes capítulos:

- Capítulo 1: Apresentação de conceitos básicos sobre Teste de *Software*, Oráculos de Teste e CBIR;
- Capítulo 2: Descrição da metodologia utilizada para utilizar técnicas de CBIR em um ambiente batizado de *"oráculo gráfico"*;
- Capítulo 3: Demonstração de um exemplo de utilização da estrutura por meio de um estudo de caso.
- Conclusões
- Referências

CAPÍTULO 1 – TESTE DE *SOFTWARE*, CBIR e ORÁCULOS

O presente capítulo tem como finalidade apresentar alguns conceitos e definições considerados pré-requisitos fundamentais para o bom entendimento do trabalho que é apresentado. Algumas demonstrações de trabalhos anteriores ajudam-nos a entender o contexto no qual está inserido.

Em linhas gerais, é importante que se esclareça o que são atividades de Teste de *Software*, técnicas de CBIR e o que pode ser esperado da aliança destas duas áreas tão díspares da Computação.

Após a apresentação dos conceitos julgados importantes, iniciam-se subseções que visam a dar entendimento do significado do termo "*oráculo gráfico*", que é o grande propósito da estrutura desenvolvida.

1.1 Teste de *Software*

Como conseqüência do intenso uso dos sistemas computacionais na vida de muitas sociedades do século XXI, é necessário cada vez mais garantir a fidedignidade desses tipos de produto. Em outras palavras, a produção de um *software*, seja este um sistema de controle de processo de alto risco ou um comum programa de controle de estoque, deve passar por uma forma de avaliação de qualidade. Com esse propósito, constituiu-se um dos desafios da Computação, discutido no seminário "Grandes Desafios Pesquisa em Computação no Brasil" (GRANDES, 2006). Trata-se do "Desenvolvimento Tecnológico de Qualidade: sistemas disponíveis, corretos, seguros, escaláveis, persistentes e ubíquos". Para que se atinja este objetivo são necessárias as atividades de teste de *software*.

Segundo Pressmam (1995, p.724), a Garantia de Qualidade de *Software* ou SQA, do termo inglês *Software Quality Assurance*, é uma atividade que deve ser aplicada ao longo de todo o processo de engenharia de *software* e abrange, entre outros itens, uma estratégia para testes de fases e formas de medição e divulgação. Atividades conjuntas sob o título de SQA têm sido introduzidas ao longo de todo o processo de desenvolvimento, entre elas atividades de "VV&T" (Validação, Verificação e Teste de *Software*), com o objetivo de minimizar a ocorrência de erros (MALDONADO et al., 2004). Dentre os métodos de verificação e validação, a atividade de teste é uma das mais empregadas, e compõe um dos elementos para

analisar a confiabilidade do *software* em complemento a outras atividades (DELAMARO et al., 2007b).

Em outras palavras, o teste de *software* serve para auxiliar as empresas que desenvolvem sistemas computacionais de forma que seus produtos atinjam níveis e padrões de qualidade esperados. Técnicas e ferramentas são indispensáveis para a prática de atividades de teste de maneira sistematizada e com embasamento científico (DELAMARO et al., 2007b). Estudos evidenciam problemas quanto à falta de qualidade dos *software*s no mercado e à disposição dos usuários, bem como em empresas e indústrias. Estima-se que, na atmosfera atual, quase metade dos *software*s que são colocados à disposição do consumidor contêm erros ou falhas complexas (MYERS, 2004).

Intertextualizando, a evolução de todo e qualquer *software* é vista como natural e sabe-se que ela acontece independentemente da disciplina que tenha sido imposta em seu processo de criação (MYERS, 2004). Esta propriedade não é observada apenas em sistemas computacionais, ou seja, de uma maneira global, todo produto em uso tende a evoluir. No entanto, a evolução de tais produtos, inclusive os *software*s, não pode influenciar ou comprometer sua qualidade final de nenhuma forma.

É senso comum entre os programadores reconhecer que a aplicação das atividades de teste, durante um processo de criação de um *software*, é uma atividade cara e complexa. Diante disso é importante replicar que com o passar o tempo a busca pelo barateamento e facilitação da prática do teste fez com que diferentes técnicas, ferramentas e critérios fossem criados. Tal pluralização dos testes cresceu ainda mais em conformidade com a criação de novas linguagens de programação e aplicações para web (MYERS, 2004).

Autores compartilham a idéia de que todo *software* contém erros. Tal mentalidade pode ser explicada partindo-se do princípio de que os sistemas são desenvolvidos a partir de técnicas oriundas do que é conhecido por Engenharia de *Software*. Esta é resumida por Pressman (1995, p.31) como disciplina que aproveita os princípios da engenharia com o objetivo de produzir *software* de alta qualidade e baixo custo. Seguindo esta linha de raciocínio, nota-se que a criação de um sistema não é uma atividade trivial, haja vista que o sucesso de seu desenvolvimento deve ser conseqüência de uma série de atividades de engenharia. Segundo Delamaro et al. (2007b) o desenvolvimento de *software* está sujeito a erros, pois depende principalmente da habilidade, da interpretação e da execução das pessoas que o constrõem. O desenvolvimento de sistemas de *software* envolve uma série de atividades de produção em que as oportunidades de injeção de falhas são enormes (DEUTSCH, 1979).

Logo, erros acabam surgindo, ainda que sejam utilizados métodos e ferramentas de engenharia de *software* adequados.

É comum a informatização de trabalhos que, há algum tempo, eram exclusiva e exaustivamente realizados por seres humanos. Isto leva à exposição de tais trabalhos aos mais diferenciados tipos de erros e falhas que podem resultar de um processo computacional. Nota-se que a informatização de um determinado trabalho ou setor, pode não ser condição suficiente para a isenção total de possíveis falhas relacionadas a ele.

Quando um Sistema de Informação não processa dados da forma aguardada é porque ele contém algum defeito. As atividades "VV&T" devem ser inseridas para que não persistam defeitos em um sistema e estes sejam corrigidos antes da liberação do *software* para uso (DELAMARO et al., 2007b). Atividades "VV&T" estão diretamente relacionadas com detecção e correção de erros ou defeitos em *software*s, desde seu processo de desenvolvimento até seu produto final em conformidade com o especificado.

Segundo Myers (2004, p.6), atividades de teste de *software* podem ser definidas como o processo de execução de um programa com a intenção de encontrar seus erros. O autor condena definições que têm por base a idéia de que as atividades de teste são um ou mais processos que têm a finalidade de descobrir se um *software* desempenha o que lhe é definido de maneira correta. Tal definição é rotulada como equívoca baseando-se em dois argumentos; o primeiro é que um *software* pode fazer o que dele é esperado e mesmo assim conter erros; o segundo é que quando um testador assimila tal definição, ele deve buscar que o programa desempenhe o que lhe é atribuído de maneira correta, desviando assim seu foco principal: detectar e corrigir erros em um programa.

O mesmo Myers (2004, p.2) é menos radical em sua abordagem que posiciona o teste como processo ou série de processos concebidos para certificarem se um código realmente realiza o objetivo para o qual foi projetado e nada adicional a isso. Tal abordagem aproxima-se muito da mencionada por Pressman (1995, p.786) que classifica o teste como um elemento crítico da garantia de qualidade de um *software* e representa a última revisão de especificação, projeto e condições.

O contexto de teste no qual a estrutura desenvolvida neste trabalho está inserida é abordado por Delamaro et al. (2007b, p.2). Segundo esse autor, o teste de *software* é uma atividade dinâmica que tem o intuito de executar o programa ou modelo, utilizando algumas entradas em particular, e verificar se seu comportamento está de acordo com o esperado. Obviamente, se a execução apresentar resultados não especificados, diz-se que um erro ou

defeito foi identificado. Por residir em tal abordagem grande correlação com o protótipo desenvolvido no trabalho, ela será utilizada como modelo de definição de Teste de *Software*.

1.1.1 Defeitos, Erros e Falhas

Conhecer a diferença entre Defeitos, Erros e Falhas é fundamental para entender definições futuras. As definições utilizadas aqui seguem a terminologia padrão para Engenharia de *Software* do IEEE (IEEE,1990) e são, portanto, voltadas para o contexto no qual foi inserido o trabalho.

- Defeito - ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.
- Erro - manifestação concreta de um defeito num artefato de *software*. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.
- Falha - comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros. Salienta-se que alguns erros podem nunca resultar em falha.

Figura 1 - Defeito x Erro x Falha

Fonte: Revista Engenharia de Software edição especial Rio de Janeiro, v.01, n.01, p.54-59, 2007.

Verificam-se na Figura 1 as diferenças entre esses conceitos. *Defeitos* fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, pelo mau uso de uma tecnologia. É sabido que defeitos podem ocasionar a manifestação de *erros* em um produto, ou seja, a construção de um *software* de forma diferente do que foi especificado (universo de informação). Por fim, os erros geram *falhas*, que são

comportamentos inesperados em um *software* e afetam diretamente o usuário final da aplicação (universo do usuário), podendo inviabilizar a utilização de um *software* (DIAS NETO, 2007).

1.1.2 Elementos do Teste de *Software*

A atividade de teste é constituída de elementos essenciais que ajudam na formalização desta prática como um todo. Definem-se esses elementos como *caso de teste, procedimento de teste e critério de teste.*

- Caso de Teste - elemento que descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado (NETO, 2007).
- Procedimento de Teste - descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (ROCHA et al., 2001).
- Critério de Teste - seleciona e avalia casos de teste de forma a aumentar as possibilidades de provocar falhas ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA et al., 2001).

1.2 Fases de Teste

Em função de sua complexidade, as atividades de teste geralmente são divididas em fases que diferem quanto ao alvo de avaliação, entretanto compartilham de uma finalidade comum: revelar um erro no produto testado. A divisão em fases mais utilizada pelos autores define teste de unidade, teste de integração ou incremental e teste de sistema como etapas fundamentais da atividade avaliadora (DELAMARO et al., 2007b, p.4).

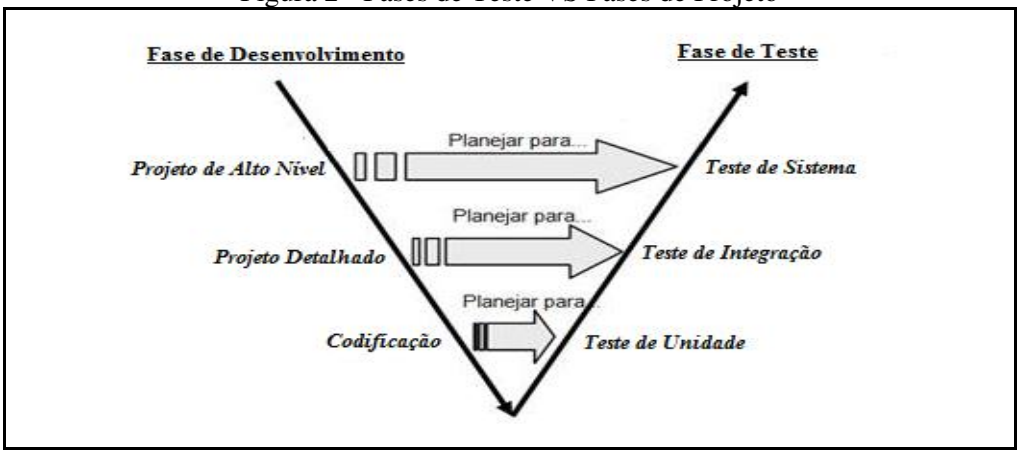
Segundo Delamaro et al. (2007b, p.4), teste de unidade é aquele que tem como foco as menores unidades de um programa, ou seja, funções, procedimentos, métodos ou classes. De acordo com Myers (2004, p.91), teste unitário é um processo de teste individual para subprogramas, sub-rotinas ou procedimentos em um programa, visto que, antes de testar um programa como um todo, deve existir um controle de qualidade que incida sobre os blocos que o constroem. Este controle é o teste de unidades.

A fase de teste de integração é definida por Myers (2004, p.105) como a abordagem que testa, posteriormente ao teste unitário, a combinação dos módulos que formam o

programa conjuntamente. Portanto, deve ser conseqüência do teste de unidade e proporcional à adição das diversas partes do *software* em um contexto de forma que trabalhem juntas por um ideal. Devem ser realizadas avaliações para que o acréscimo do sistema não leve a erros (DELAMARO et al., 2007b, p.4).

O teste de sistemas, realizado após a integração, visa à identificação de erros de funções e características de desempenho que não estejam de acordo com a especificação (MALDONADO et al., 2004, p.3). Segundo Delamaro et al. (2007b, p.4), esta fase tem como objetivo verificar se as funcionalidades, especificadas nos documentos de requisitos, estão todas corretamente implementadas.

Figura 2 - Fases de Teste VS Fases de Projeto



A Figura 2 ilustra uma forma de relacionar as fases de teste de *software* às fases de um projeto de *software*. Em decorrência deste paralelismo, muitas organizações adotam a estratégia de designar uma equipe independente para realizar os testes do sistema. Observa-se que as fases de codificação, projeto detalhado e alto nível são paralelamente relacionadas com as fases de teste de unidade, integração e sistemas respectivamente.

1.3 Economias e Benefícios

É de conhecimento geral que em qualquer linha de produção, seja ela de larga ou pequena escala, existe o setor ou departamento responsável por realizar testes. Nenhuma empresa deseja colocar produtos à disposição de seus clientes sem que antes sejam testados adequadamente para uso. Contrastando com alguns produtos que admitem testes apenas quando seu processo de produção é encerrado, o *software* deve ser avaliado desde a primeira

instância do processo produtor. Conseqüentemente o custeio do projeto e o tempo necessário para a elaboração de um produto final são elevados, o que se percebe pela grande relutância por parte de alguns gestores em realizar aplicações de teste desde o inicio do seu processo de desenvolvimento. Considera-se, no caso, tal atividade com um policiamento que resulta em altos encargos financeiros e em um considerável aumento no prazo de entrega do produto final. (MYERS, 2004)

O fato é que as estatísticas têm denunciado exatamente o contrário. Pesquisas recentes mostram que executivos têm desembolsado quantias consideráveis para corrigir *software*s prontos em que, se tivessem sido realizadas atividades testadoras adequadas durante seu processo de produção, a resultante seria uma economia considerável.

Quando um *software*, já em atividade, precisa ser alterado ou corrigido para sanar algo que não está sendo executado em conformidade com o esperado, um processo investigativo e de pesquisa deve ser realizado, visando a encontrar as partes defeituosas para que possam ser corrigidas e o corretor deve transmitir para o usuário o diagnóstico das causas e conseqüências dos problemas. Em suma, este é o grande causador do encarecimento da alteração de um sistema já formado. Por isso, algumas organizações de *software* têm gastado até 40% dos esforços totais de projetos com as atividades testadoras (PRESSMAN, 1995, p.786).

Nota-se que o teste proporciona um custo benefício evidente, quando feito de forma adequada desde o início do projeto. No aspecto operacional, os testes geram um produto final confiável e que atende em todos os sentidos às necessidades do usuário. Segundo Myers (2004) avaliar um sistema adequadamente pode transformar o risco de grandes perdas em uma positiva vantagem competitiva.

1.4 Testadores versus Desenvolvedores

Atividades de teste têm propósito e base totalmente diferentes das práticas de desenvolvimento de *software*, uma vez que processos testadores não podem revelar a ausência de *bugs*, mas detectar a presença deles (PRESSMAN, 1995, p.789; MYERS, 2004). O teste se resume em uma prática que deve ser aplicada durante todo o ciclo de vida de um programa, ou seja, desde seus primeiros passos de desenvolvimento até o produto final, o que implica dizer que desenvolvimento e testes devem se harmonizar paralelamente à correção do *software*.

É importante lembrar que, nos dias atuais, o mercado da Computação encara a função de desenvolvimento de *software* de forma distinta da função do teste de *software*. Isso levou à valorização de uma nova profissão nas indústrias atuantes em tecnologias da informação, a profissão de testador de *software* (DELAMARO et al., 2007b). Um testador de *software* tem por função maior montar, estrategicamente, uma série de atividades definidas que busquem demolir tudo aquilo que foi construído por um desenvolvedor. Esse tem por objetivo maior montar estratégias e, muitas vezes, usar de técnicas de engenharia de *software* para arquitetar um programa que processe dados de forma fiel e execute tudo o que dele for esperado (MYERS, 2004).

Evidencia-se, então, a importante figura de um testador. Levando-se em conta que um desenvolvedor não tenha percebido a presença de um *bug* na hora de programar, não se pode afirmar que este desenvolvedor irá revelar tal falha na hora de testar. (PRESSMAN, 1995). É por isso que a independência dos processos de testes e de desenvolvimento, com base em uma relação cliente-fornecedor, na visão de que quem desenvolve não testa, resulta no aumento da qualidade e da produtividade das equipes (MOREIRA FILHO, 2008).

Cabe ressaltar que são metas distintas durante um projeto de construção de *software*, as de testador e desenvolvedor. Este tem a meta de sintetizar e usufruir de diferentes metodologias para fazer um produto conciso, preciso e exato em relação à sua aplicação destino. Já aquele, segundo Myers (2004), tem sucesso quando revela um novo erro e projeta bons casos de teste capazes de revelar falhas obscuras.

1.5 Técnicas e Critérios de Teste

Teste Funcional (Caixa Preta), Teste Estrutural (Caixa Branca), Teste Baseado em Modelos, Teste de Mutação, Teste Orientado a Objetos e Teste de Aspectos são, entre outras, algumas técnicas de teste, com fundamentações científicas, à disposição dos testadores (DELAMARO et al, 2007b). Tais técnicas foram desenvolvidas objetivando conduzir e avaliar a qualidade da atividade de teste e se diferenciam pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste (MALDONADO, 91). Nesta seção apresentam-se, com mais detalhes, os testes funcional e estrutural.

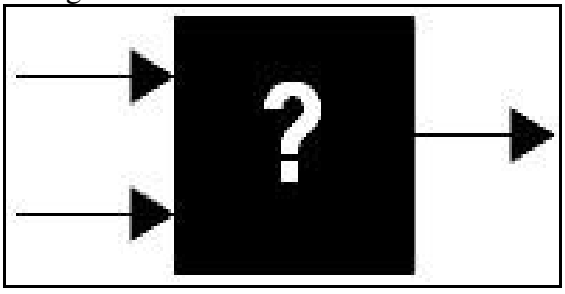
1.5.1 Teste Funcional (Caixa Preta)

A técnica do Teste Funcional é de simples entendimento. Nela o programa é visto como uma caixa preta (por isso alguns a chamam de Teste Caixa Preta) em que o testador não tem idéia de nenhum detalhe de implementação do sistema em teste. Dessa forma para ser realizada uma avaliação basta que o testador forneça entradas para o *software* e observe as saídas por ele geradas, verificando se elas estão em conformidade com os objetivos especificados (DELAMARO et al., 2007b, p.9).

De acordo com Pressman (1995, p.816), o teste funcional, quando os testadores têm em seu domínio as funções específicas do *software*, é realizado com a finalidade de demonstrar que cada função do sistema é operacional. Desta forma, não há interesse algum em observar a estrutura lógica do programa, ou seja, entradas adequadamente aceitas e saídas adequadamente produzidas, mantendo-se assim a integridade das informações externas (PRESSMAN, 1995).

Como é apresentado na Figura 3, nada inerente ao programa é transparente para o testador, sendo assim, não é possível prever qualquer comportamento do *software* durante a realização do teste funcional. Tomando esta idéia como base, elucida-se que a única maneira de encontrar todos os erros de um *software*, por intermédio do teste caixa preta, é pelo esgotamento de todas as possíveis combinações de entrada do programa. Em outras palavras, deve ser feita uma exaustão dos casos de teste no *software*. Tal tarefa é conhecida pelos profissionais do ramo como teste de exaustão (MYERS, 2004).

Figura 3 - Técnica de Teste Funcional



É possível afirmar que a geração de todos os possíveis casos de teste de um programa é impraticável, porque há programas que aceitam infinitas entradas diferentes. Um exemplo disso é que se pode imaginar, mas não praticar, a quantidade de casos teste de um compilador da linguagem C.

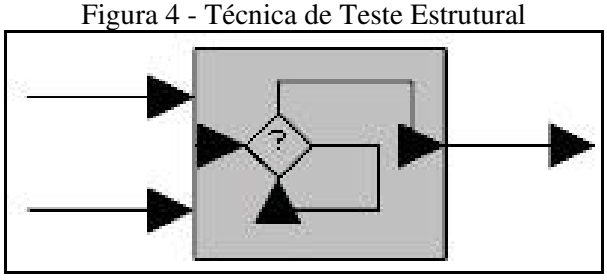
Conclui-se que deve haver uma estratégia para sintetizar casos de teste que tenham maior probabilidade de revelar falhas ainda não descobertas, pois também é objetivo das atividades de teste maximizar seus rendimentos. Isso implica desenvolver formas de fazer com que um número finito de casos de teste encontre um bom número de falhas no *software* testado.

1.5.2 Teste Estrutural (Caixa Branca)

O Teste Funcional, observado na seção acima, pode ser considerado como uma avaliação iniciada às avessas em relação ao Teste Estrutural. Enquanto naquele o testador não tem nenhuma especificação de lógica do *software* inserida por parte do programador, neste ele não só deve ter em mãos a estrutura interna do sistema, como deve basear-se nela para aplicar os mais diferenciados casos de teste (MYERS, 2004).

A ilustração da Figura 4 é um paradigma de um teste estrutural ou Caixa Branca (assim chamado porque é transparente ao testador a estrutura interna do programa). Os requisitos de teste devem ser estabelecidos, baseados exatamente na implementação realizada pelo desenvolvedor, requerendo a execução de partes ou componentes elementares do programa, que é de domínio do testador (MYERS, 2004; PRESSMAN, 1995). Sendo assim, as atividades de teste devem ser realizadas para que haja a garantia de que as especificações internas do produto foram adequadamente postas à prova.

Figura 4 - Técnica de Teste Estrutural



A fundamentação do teste estrutural (caixa branca) consiste de um minucioso exame de detalhes procedimentais, ou seja, avaliações dos caminhos lógicos por meio do *software* em teste (PRESSMAN, 1995). Chega-se à conclusão de que há programas que proporcionariam exaustivos casos de teste, da mesma forma que se observou na avaliação do teste funcional (caixa preta). Um exemplo disso seria um programa que contivesse comandos

de repetição atrelados a condições de parada diversas. Indubitavelmente se chegaria aos testes exaustivos (DIAS NETO, 2007).

A grande diferença entre teste estrutural e funcional está na fonte utilizada para definir requisitos de teste. Nas duas técnicas apresentadas, a aplicação do teste exaustivo seria a única de forma de afirmar que um programa foi completamente testado. No entanto, pode-se inferir que até mesmo um teste de exaustão não é suficiente para garantir que um *software* é totalmente correto, uma vez que pode haver incompletude do domínio de entradas possíveis ou falta de caminhos lógicos internos. Myers (2004) sugere que talvez fosse necessária uma comparação entre elementos funcionais e estruturais, com a finalidade de obtenção de uma estratégia de teste razoável, mas ainda assim incompleta.

1.6 Oráculos de Teste

Já foi citado que o foco central deste trabalho é a construção de um ambiente que, no contexto apresentado por Delamaro (2007a), é batizado de *oráculo gráfico*. Para o completo entendimento de tal conceito, é necessário que o conceito genérico de oráculo seja compreendido sem obscuridades.

O termo oráculo é originário da mitologia grega e geralmente era conhecido por se tratar da resposta de um sacerdote ou divindade, geralmente subjetiva e obscura, dada a uma questão de um consulente qualquer (GRANDE, 1998, p.4325). Quando usado na área da Computaça, mais precisamente na área de teste de *software*, difere muito disso.

O encarecimento e a complexidade das atividades de teste em um processo de desenvolvimento de *software* se dão em função da grande dificuldade encontrada quando é buscada uma definição precisa do modo de avaliar a qualidade de determinado processamento. Dificuldade similar é encontrada na procura de um conjunto ideal de testes que seja completo o bastante para revelar os mais diferentes defeitos (DELAMARO et al, 2001).

Nesse contexto é inserido o conceito genérico de oráculo, definido como mecanismo que se utiliza para definir a saída ou comportamento esperado de uma execução qualquer (HOFFMAN, 2001). É importante salientar que diferentes aspectos influenciam no modo de obtenção de um oráculo para determinado sistema.

Caso seja possível extrair o comportamento de determinado processamento usando um modelo formal, como um autômato, é possível automatizar uma função de oráculo por

intermédio de um elemento comparador entre a saída produzida e a saída esperada, esta definida por meio do modelo.

Oráculos automatizados de teste são componentes essenciais na atividade de teste de *software* (DELAMARO et al., 2008). Oráculo é parte essencial do teste de *software*. Muitos trabalhos da área vêm para abordar exatamente a subjetividade de se decidir sobre a correção ou não de um programa em teste. O trabalho apresentado por Hoffman (2001) cita que há uma classificação utilizada por testadores que divide oráculos em:

- *ativos*: responsáveis por diretamente dirigir ou coordenar as atividades de testes ;
- *passivos*: oráculos menos complexos e que apenas recebem como entrada o par comportamento desejado, comportamento produzido.

O referido trabalho trata especificamente dos benefícios que a utilização de um oráculo automatizado pode trazer ao processo de desenvolvimento de um programa. Segundo as idéias do autor, os oráculos automatizados podem ser classificados em quatro classes bem distintas quanto à sua forma de automatização. São eles:

- *oráculos verdadeiros* geram respostas de maneira independente do programa a ser testado e devem prever respostas apenas para entradas utilizadas no teste;
- o *oráculo(estratégia) de consistência* baseia-se na execução de um programa para avaliar a correção de outras execuções;
- o *oráculo(estratégia) de referência própria* consiste na soma de dados ao resultado esperado juntamente com a própria estrutura e com os dados do teste;
- o *oráculo (estratégia) heurístico* é muito simples e consiste da verificação de algumas características que relatem se a execução está correta ou não.

Em Hoffman (2006) é apresentado um resumo de uma série de artigos que visam a descrever as diferentes finalidades e utilizações dos oráculos automatizados de verificação e validação de *software*. A Tabela 1 é baseada na pesquisa realizada pelo referido autor. Ela exibe vários tipos de oráculos, algumas de suas características mais relevantes, as vantagens, desvantagens, e as implicações em que estão incluídos para automatização de testes.

Tabela 1 - Comparação de tipos de oráculos

Classificação	**Definição**	**Vantagens**	**Desvantagens**
Sem Oráculo	• Sem verificação de resultados	• Pode rodar qualquer quantidade de dados	• Apenas erros notáveis são percebidos

Classificação	Definição	Vantagens	Desvantagens
Oráculo Verdadeiro	• Geração independente de todos os resultados esperados	• Todos os erros são encontrados na área de avaliação	• Implementação cara e complexa.
Estratégia de *Consistência*	• Verifica resultados correntes com resultados anteriores	• Método mais rápido usando um oráculo. • Simples verificação. • Pode verificar grandes quantidades de dados	• Pode não detectar erros originalmente.
Estratégia de Referência Própria	• Incrementa respostas aos dados nas mensagens	• Permite extensa análise pós-teste. • A confirmação está baseada na mensagem conteúdo. • Pode gerar grandes quantidades de dados complexos.	• Deve definir respostas, logo tem que gerar mensagens.
Estratégia Heurística	• Verifica algumas características úteis de valores	• Mais rápido e fácil do que Oráculo Verdadeiro • Barato na maioria das vezes.	• Pode perder erros. • Pode gerar alarmes falsos

Cabe ressaltar que definir um oráculo implica sintetizar uma estrutura formal, ou até mesmo informal, automatizada, que seja capaz de oferecer ao usuário um veredicto indicativo da exatidão de uma execução do sistema ao final das aplicações do teste. Sendo assim, pode ser dito que oráculo é o mecanismo que define e dá um veredicto acerca da correção de uma execução de um programa em teste (HOFFMAN, 2001 ; DELAMARO et al., 2008).

1.6.1 Contexto de Oráculo no Trabalho Proposto

Já foi dito que este trabalho é inserido no contexto do projeto descrito em Delamaro et al. (2007b). Logo não poderia ter outro parâmetro, que não fosse o referido projeto. Sendo assim, baseia-se na definição de técnicas e desenvolvimento de ferramentas que possibilitem a automatização do mecanismo de oráculo, de fundamental importância no contexto de teste de *software*.

Os Oráculos são classificados por Delamaro (2007a), em relação à sua capacidade de automatização, nas seguintes categorias:

- *Baseados em especificações informais, em geral textuais*. Dificilmente automatizáveis, uma vez que a decisão sobre a correção de uma determinada execução depende da interpretação da especificação.
- *Baseados em modelos formais*. Em geral, de fácil automatização, uma vez que as saídas esperadas são definidas no modelo formal e dele podem ser extraídas. Além disso, o tipo de saída tende a ser simples e facilmente tratado.
- *Baseados em resultados armazenados da execução de outros programas*. Nesse acaso, a automatização é simples, se o tipo de resultados a ser considerado for simples, como saída textual ou eventos discretos. A automatização é mais complicada quando a saída é complexa, em particular, no formato gráfico.

Este trabalho foca especificamente a automatização de oráculos de programas cuja saída é apresentada na forma de uma imagem, seja ela uma janela de uma interface gráfica, seja qualquer outra imagem produzida em teste, logo é inserido no terceiro caso (DELAMARO, 2007a).

1.6.2 Trabalhos Relacionados a Oráculos

Há autores que afirmam que qualquer trabalho que envolva Teste de *Software* e critérios de teste correlacionam-se com o tema oráculo. No entanto há trabalhos que são bem mais específicos e têm como foco principal uma forma de gerar e trabalhar adequadamente com oráculos de teste de *software*.

Foi proposta por AZEVEDO (2004) uma estratégia para geração de oráculos para teste de *software* a partir de especificação formal que visava a um aumento na abrangência da aplicabilidade e usabilidade dos geradores de oráculo. Tal estratégia possibilita a geração de oráculos para implementações não derivadas diretamente da estrutura da especificação, os quais podem ser aplicados a casos de testes derivados de qualquer técnica de seleção de casos de teste.

Proposto por Takahashi (2001), o trabalho denominado “Uma automatização de oráculo para objetos GUI” é interessado em criação de oráculos para programas com interface gráfica, ou seja, interfaces GUI. O autor identifica algumas das formas pelas quais um

programa pode produzir suas saídas, como texto, arquivo, memória, janela, dados de comunicação ou imagem.

Disposto a evitar a comparação pixel a pixel entre as saídas de duas execuções de um programa com saída gráfica, o autor propõe um sistema que intercepta chamadas a API gráfica e guarda/compara tais informações.

1.7 Recuperação de Imagem Baseada em Conteúdo (CBIR)

É notável o aumento do uso de documentos digitalizados por parte de diversas organizações no contexto atual. Seu uso está em constante crescimento, sendo armazenados em diferentes bases de dados que podem ser acessadas por meio de redes de comunicação como a internet. O uso em larga escala deste tipo de documento levou à necessidade de criação de eficientes algoritmos de recuperação, indexação e técnicas de classificação de imagens.

Cabe ainda citar que diversas áreas da ciência foram amplamente beneficiadas por tal evolução, com destaque especial para a área das ciências médicas. No diagnóstico auxiliado por computador, conhecido como esquema CAD, o médico sintetizará um diagnóstico levando em consideração o resultado de processamento de um sistema computadorizado, que pode executar baseando-se em processamentos e análises de imagens ou até em dados clínicos de pacientes (GATO et al., 2004).

Esse cenário favorável forçou o aprimoramento da tecnologia e o barateamento das operações digitalizadas, como exemplo, os exames médicos. Naturalmente isso fez com que fossem criados sistemas computacionais que suportassem a demanda das tarefas de análise de imagens (TRAINA, 2006). Em paralelo, muitos pesquisadores voltaram suas pesquisas para a área do processamento de imagens e da computação gráfica como um todo, fazendo com que novas tecnologias e métodos de trabalho surgissem na área. O CBIR foi um deles.

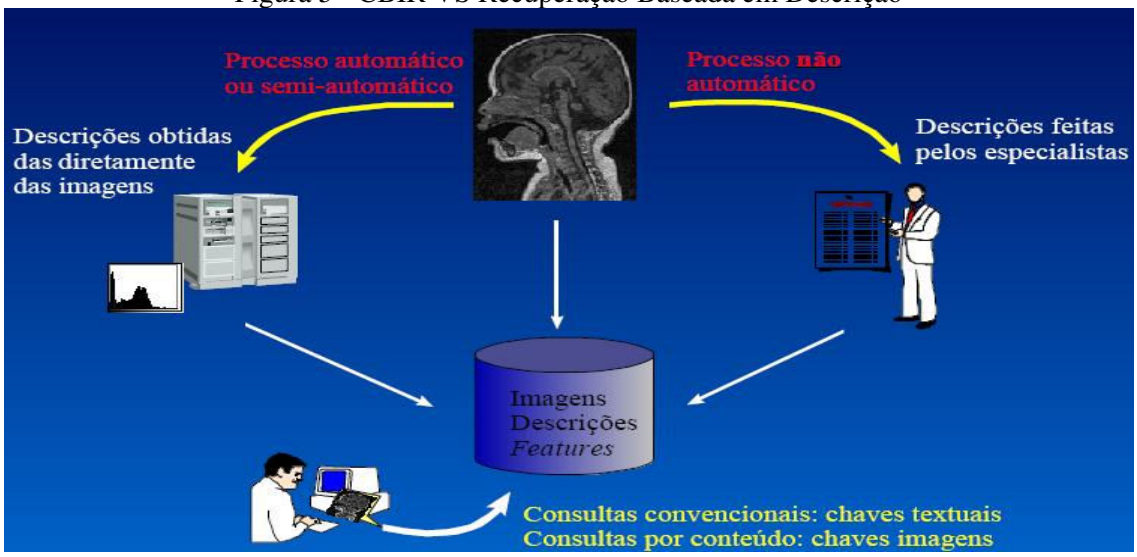
O CBIR é uma técnica de recuperação de imagem que tem como princípio básico a realização de uma consulta em um banco de dados com uma determinada quantidade de imagens similares a uma imagem de referência, baseando-se em um ou mais critérios fornecidos (SANTOS, 2006; OLIVEIRA et al., 2008). Tais critérios se resumem nas características das imagens e podem tanto ser obtidas por especialistas como ser extraídas das imagens por algoritmos automáticos alcançando melhores resultados (SANTOS, 2006, p.21;

ARAUJO et al., 2002). São exemplos de características de imagens: forma, textura, borda, cor etc.

O propósito do CBIR nem sempre é buscar a imagem exatamente igual à imagem de consulta, o que poderia acontecer por uma comparação pixel a pixel, e sim buscar a imagem mais parecida ou similar. A consulta por distância recupera objetos dentro de certo grau de similaridade, um número fixo de objetos similares. A comparação de objetos por similaridade é essencial para tratar dados complexos e capturar o que é mais representativo a fim de extrair informações que os representem de modo mais fiel (FILARDI, 2007, p5).

Pode-se classificar CBIR como um processo que exige muito tempo de processamento, por isso a comparação entre as imagens é feita utilizando um conjunto de características extraídas das imagens que as descrevem (SANTOS, 2006). Esse conjunto de características extraídas de uma imagem forma o seu vetor de características, que é utilizado na sua indexação e recuperação. Pode-se concluir que as características extraídas representam a imagem no momento de sua busca, pois é a partir delas que uma determinada imagem é recuperada do banco de imagens (DELAMARO, 2007a). O conjunto de características em si não é suficiente para determinar o resultado da recuperação, visto que a escolha da medida de similaridade entre as imagens vai, também, ter influência nele.

Figura 5 - CBIR VS Recuperação Baseada em Descrição



A busca acontece pelas imagens mais relevantes segundo tais medidas, ou melhor, funções de similaridade entre uma imagem de consulta e as que estão armazenadas no banco de dados (KINOSHITA, 2004). Logo, o CBIR fornece benefícios notórios, quando

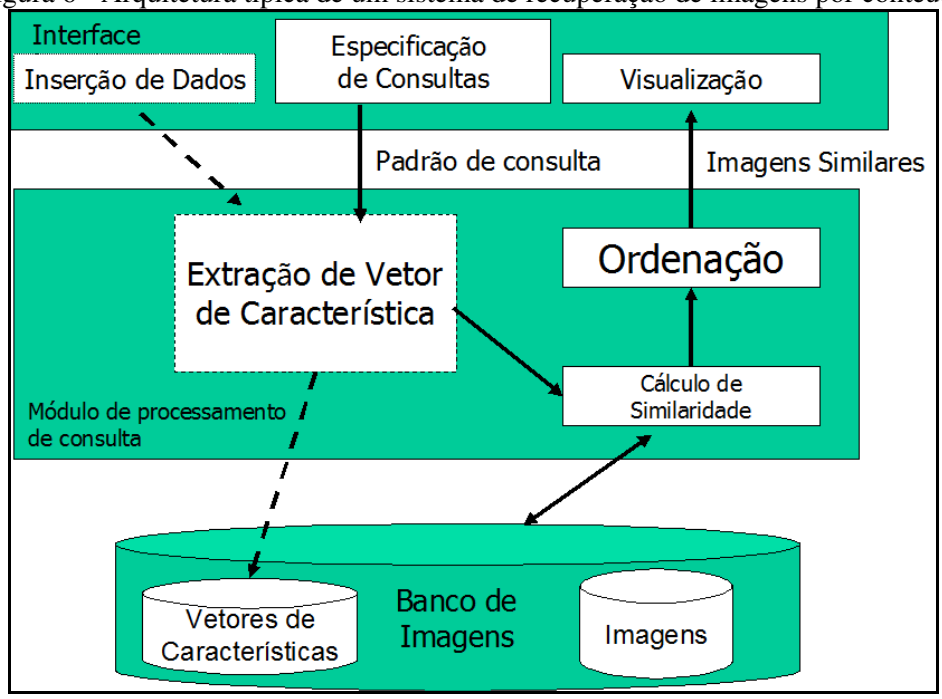
comparado a métodos de recuperação baseados em descrição textual (TRAINA, 2006). A Figura 5 ilustra uma paralelização de CBIR com recuperação baseada em descrição textual.

A definição mais adequada de CBIR, considerando o contexto do trabalho apresentado, é encontrada em Santos (2006, p.21-22):

> "... conclui-se que sistemas CBIR permitem a recuperação de um conjunto finito de imagens similares a uma imagem exemplo. Para isso é feito o uso de informações inerentes à própria imagem. Tais similaridades devem ter o nível de semelhança pré-determinado pelo usuário. As comparações entre as imagens podem ser realizadas através de características extraídas e automaticamente agrupadas em um vetor de características que tem por finalidade armazenar a essência da imagem ..."

Uma possível arquitetura para sitemas CBIR é descrita em Torres (2006). Tal é apresentada na Figura 6.

Figura 6 - Arquitetura típica de um sistema de recuperação de imagens por conteúdo



Fonte: Revista de Informática Teórica e Aplicada, 13(2):161–185, 2006.

1.7.1 Etapas do CBIR

São etapas básicas do CBIR: aquisição, pré-processamento, extração de características, indexação e recuperação: (ARAUJO et al., 2002; SANTOS, 2006, p.22)

- *Aquisição:* consiste em, por algum meio físico, obter as imagens digitais a serem utilizadas.

- *Pré-processamento:* baseia-se em melhorar a imagem, por meio de técnicas para realce de contraste, remoção de ruído, isolamento de regiões e etc. Esta etapa tem, indiretamente e como objetivo maior, aumentar as chances de sucesso dos processos seguintes.
- *Extração de características:* tem por meta extrair informações ou dados particulares da imagem, as quais serão utilizadas na busca na base de dados.
- *Indexação:* consiste em aperfeiçoar e facilitar a consulta à base de dados de imagens.
- *Recuperação:* consiste da varredura de toda a base de dados, de forma a recuperar as imagens mais similares à imagem exemplo.

Um processo de extração de característica de uma imagem deve acontecer sempre após a realização de uma etapa de segmentação, que faz com que o foco de interesse seja identificado e separado das demais partes da imagem. A comparação de características, e conseqüentemente a recuperação, será realizada acerca da parte de interesse na imagem, ou seja, os critérios de similaridade podem ser aplicados apenas nas regiões segmentadas (ARAUJO et al., 2002). No decorrer do trabalho será descrito como esse fato transformou-se virtude para flexibilidade do ambiente desenvolvido.

É sabido que em função do objetivo que se tenha, a extração de características de uma imagem pode variar de muitas maneiras (DELAMARO, 2007a). Na busca por imagens médicas, Moshfeghi et al. (2004) utilizam entropia e distância de *Kullback-Liebler* para recuperar informação em uma base de dados contendo imagens médicas provenientes de modalidades diversas. Corboy et al. (2005) propõem um sistema para recuperar imagens de regiões anatomicamente normais, presentes em estudos de Tomografia Computadorizada do pulmão e do abdômen, usando descritores de pixels locais e globais, além de descritores provenientes de matrizes de co-ocorrência para representar texturas.

Pode-se concluir que a Recuperação de Imagem Baseada em Conteúdo é uma tecnologia que visa a resgatar um conjunto finito de imagens similares a uma imagem exemplo. Como fundamento de busca, são utilizadas características obtidas da própria imagem no momento de sua análise.

1.7.2 Sistemas CBIR

É vasta a quantidade de trabalhos encontrados que têm em sua base o uso de técnica de CBIR. A grande maioria volta-se para esquemas CAD, ou seja, têm como finalidade

processamentos o auxílio a algum tipo de diagnóstico médico. Como já foi citado, uma das áreas da ciência mais beneficiadas, não só com as técnicas de CBIR, mas com a evolução da computação gráfica, foi a área médica. Dois trabalhos desta família serão apresentados nas seções seguintes.

1.7.2.1 Recuperação de Imagens Mamográficas Baseada em Conteúdo

O trabalho proposto por Santos (2006) visa à construção de uma ferramenta que aproveite técnicas de CBIR para que imagens mamográficas sejam recuperadas de uma base dados. Diversos extratores foram pesquisados e apresentados. Interessada em CBIR para imagens mamográficas, a autora emprega classes abstratas da linguagem Java para implementar versões de extratores de área, forma, SMA (Segundo Momento Angular), densidade entre outros.

A base do trabalho constituiu-se de estudos médicos anteriores que, alarmantemente, denunciavam que 10% a 30% das mulheres que tiveram o câncer de mama foram submetidas ao exame de mamografia, que por sua vez revelou um resultado negativo. Conclui-se que estas mulheres foram avaliadas por um radiologista que tinha em mãos um exame aparentemente normal. Sabe-se que casos de neoplasia têm as chances de uma recuperação bem sucedida inversamente proporcionais ao tempo da doença no organismo. Em outras palavras, quanto mais cedo um enfermo começar a se tratar corretamente, maiores serão suas chances de recuperação.

A idéia central do trabalho utiliza uma base de dados, criada em trabalhos anteriores, povoada com imagens mamográficas digitalizadas de exames que antecederam a um exame detector da neoplasia. Foi criado um programa que recupera tais imagens usando critérios de similaridade com as características de uma imagem mamográfica digitalizada. Tais características são escolhidas pelo usuário do sistema.

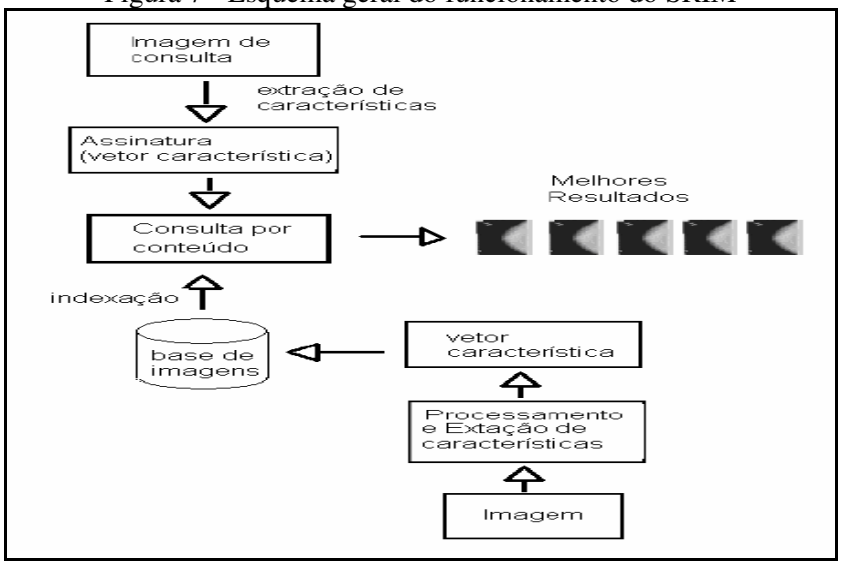
Conclui-se que é possível utilizar a ferramenta desenvolvida em diversos tipos de aplicações na área médica, mais precisamente no auxílio a diagnósticos de mamografias. Ela possibilita que sejam recuperadas imagens da base de dados que ajudem muito no diagnóstico final do radiologista. Neste contexto o trabalho realizado pela autora veio para auxiliar esquemas CAD e também pode servir como ferramenta didática para o ensino das áreas de saúde que utilizam as imagens médicas como material didático.

1.7.2.2 SRIM

O trabalho apresentado em Gato et al.(2004) explorou a tecnologia de um sistema CBIR para construir um sistema que serve como base de apoio para agilizar avaliações de imagens mamográficas com características peculiares.

Convém citar que o sistema computacional batizado de SRIM (Sistema de Recuperação de Imagens Mamográficas) pode ser utilizado também como ferramenta de apoio para o desenvolvimento de pesquisas, ajudando a acelerar o processo de seleção de imagens com casos de interesse, além de servir como ferramenta de ensino (treinamento de diagnóstico médico). O sistema faz parte de um CAD do LAPIMO (Laboratório de Análise e Processamento de Imagens Médicas e Odontológicas) do Departamento de Engenharia Elétrica da EESC/USP.

Figura 7 - Esquema geral do funcionamento do SRIM



Uma particularidade do sistema SRIM é que, simultaneamente ao projeto, foi alimentado um banco de dados que atualmente conta com 3300 imagens mamográficas digitalizadas com as mais diversas características dos pacientes. Logo o projeto rendeu uma ótima fonte para pesquisas inovadoras da área. A Figura 7 ilustra um esquema que demonstra de maneira objetiva o funcionamento do SRIM.

1.7.3 Setores que Utilizam CBIR

São diversas as áreas de pesquisa que lançam mão da tecnologia de CBIR com as mais heterogêneas finalidades. Em Gudivada & Woods (1995), é disposta uma numerosa lista de tais áreas, entre elas:

- galerias de artes e administração de museu;
- projetos de engenharia e arquitetura;
- *design* de interiores;
- percepção e administração remota dos recursos da terra;
- sistemas de informações geográficas;
- administração de banco de dados científicos;
- previsão de tempo;
- vendas no varejo;
- projetos de tecido e moda;
- administração de banco de dados de marca registrada e direito de cópia;
- execução da lei e investigação criminal;
- sistemas de arquivamento e comunicação de imagem (*Picture Archiving and Communication System* - PACS).

O projeto apresentado por Delamaro (2007a), cujo contexto é aproveitado pelo presente trabalho, visa a inserir técnicas de CBIR em uma nova área da pesquisa científica ainda não explorada conjuntamente com o processamento de imagens: a área de teste de *software*. Nele é proposto um protótipo que utiliza técnicas de CBIR para compor oráculos de teste para programas com saída gráfica, ou seja, uma nova área de aplicação para CBIR é proposta: a automação de testes por meio de oráculos.

1.8 CBIR e Teste de *Software*

A estrutura desenvolvida neste trabalho explora a tecnologia CBIR para definição de oráculos flexíveis de teste. Já foi citado que o teste de *software* globalmente é uma atividade cara para um projeto de desenvolvimento de um *software* qualquer em função das dificuldades encontradas para automatização das atividades de teste. Logo o protótipo aqui apresentado é inserido neste contexto (DELAMARO, 2007a).

É inovadora a idéia de aliar a técnica de recuperação de imagem baseada em conteúdo como forma de auxílio à atividade de teste para viabilizar a automatização do que foi batizado de *oráculo gráfico*. Experiências anteriores já foram realizadas com a intenção da criação de oráculos para programas com interface gráfica, mas para tanto não foi cogitada a possibilidade do uso de técnicas de CBIR. Muitas descobertas são esperadas de tal aliança e muitos trabalhos futuros podem ser idealizados.

Quando se trata de processamento com saídas gráficas, o problema de determinar a correção torna-se mais complexo, uma vez que nem sempre estão definidas quais são as características que devem ser ponderadas (OLIVEIRA et al, 2008). Outro obstáculo se deve ao fato de o formato gráfico não ser trivial para a automação de testes. Em particular, decidir se a saída gerada por uma execução corresponde à saída esperada – papel do oráculo de teste – é um trabalho complexo (DELAMARO et al., 2008). É exatamente o mesmo problema que se tem ao realizar uma busca em uma base de dados. Quando a busca é textual, ela pode ser realizada facilmente, existindo para tanto métricas conhecidas e firmadas. Quando se trata de busca baseada em conteúdo gráfico, a situação se torna mais complicada (DELAMARO, 2007a).

A idéia de aplicar técnicas de CBIR para realizar teste de *software* é um tanto quanto inovadora, haja vista que não há trabalhos nem artigos na literatura nos quais seja mencionada a aliança desses dois tópicos da Computaça. Myers (2004) cita critérios de teste pra interfaces gráficas (GUI's) e propõe que testes sejam realizados por meio do desenvolvimento de uma biblioteca, de forma que os objetos sejam pré-programados e cheguem para os usuários após terem passado por testes anteriores, de forma que a necessidade de testá-las nos aplicativos nos quais forem inseridas é reduzida.

1.9 Considerações Finais

Muito tem sido feito com o objetivo de encontrar um ponto de referência para que seja possível afirmar que um programa em teste está correto. Este é o papel de um Oráculo de Teste. Todavia cabe ao testador gerir formas e critérios com os quais o oráculo sintetizado conduza o teste com uma única finalidade que é apontar erros no *software* testado.

O presente trabalho tem por objetivo maior definir uma estrutura que, por técnicas de recuperação de imagem, sintetize um tipo de oráculo, ou melhor, um ambiente que será chamado de '*oráculo gráfico*'. Técnicas e critérios de teste têm que ser muito bem escolhidos e

objetivamente aplicados, uma vez que os programas alvo do trabalho são *software*s que trabalham com algum tipo de processamento gráfico.

Nesta seção introdutória foram apresentados os mais importantes conceitos para o entendimento do contexto de engenharia de *software* no qual o trabalho será inserido, haja vista que o protótipo apresentado é considerado multidisciplinar, pois busca aliar teste de *software* com processamento de imagem.

Em adição a isso, foi apresentado um embasamento teórico sobre a tecnologia CBIR. Tal teorização é necessária para o bom entendimento do trabalho proposto. A metodologia utilizada para a implementação da pesquisa e outros objetivos indiretos do produto final do trabalho estão descritos no capítulo seguinte.

CAPÍTULO 2 – METODOLOGIA

Esta seção visa a apresentar a forma com a qual conceitos da tecnologia CBIR foram utilizados em uma estrutura, contribuindo para a automatização de oráculos gráficos. Figuras e trechos de código foram utilizados para facilitar o entendimento da engenharia usada.

2.1 Modelo de CBIR para *Oráculo gráfico*

O objetivo principal do protótipo desenvolvido é servir de base para um ambiente denominado de *oráculo gráfico* (DELAMARO, 2007a). Para a sua concepção aproveitou-se o princípio de CBIR para criar uma estrutura flexível de tal maneira que seu próprio usuário-testador fosse capaz de selecionar quais características devem ser consideradas em um teste e como elas devem ser parametrizadas.

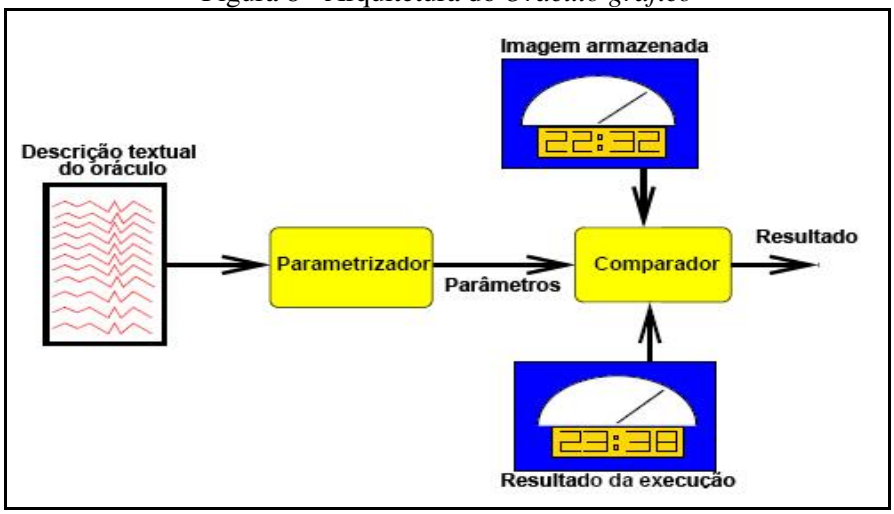
Dependendo do objetivo que se tenha, a extração de características de imagens pode variar sobremaneira. No caso de busca por imagens mamográficas, Santos (2006) utiliza extratores de Área, Forma e Densidade entre outros. Por fim a autora apresenta, como critério de comparação de características, a função de similaridade chamada Distância Euclidiana, considerada a similaridade mais simples dentre algumas outras elencadas por ela.

O conjunto de características que se deseja utilizar seguramente não se aplica a nenhum domínio ou sequer a nenhum programa dentro de um mesmo domínio (DELAMARO, 2007a). Sendo assim, um dos princípios do protótipo é dar flexibilidade ao testador para que este possa escolher quais serão as características a serem usadas na comparação de resultados. Por isso, ao construir a estrutura para um ambiente de *oráculo gráfico*, deve-se ter em mente a sua flexibilidade para permitir que:

- O testador escolha a similaridade e as características que serão utilizadas
- O testador escolha como as características devem ser comparadas (parametrização)
- Novos Extratores e Funções de Similaridade sejam adicionados (instalados) de maneira simples.

Um diagrama do funcionamento do protótipo de um *oráculo gráfico* é apresentado em Delamaro (2007a). A estrutura deste protótipo pode ser observada na Figura 8.

Figura 8 - Arquitetura do *Oráculo gráfico*



O componente denominado de *Comparador* tem o objetivo de extrair o vetor de características das imagens a serem comparadas e apresentar o veredicto. Para que isso seja realizado, além das imagens, o componente conta com um conjunto de parâmetros que indicam quais características serão consideradas e como elas devem ser comparadas. Para isso o testador deve fornecer uma *Descrição Textual* desses parâmetros. A construção flexível de tal descrição dentro do ambiente foi implementada por completo neste trabalho. Essa descrição textual é processada por um *Parametrizador* que traduz o texto para um formato que possa ser utilizado pelo comparador. Resume-se que, para realizar tal tradução, o parametrizador conta com um *parser*, implementado no trabalho, que lê o arquivo de parâmetros e cria objetos (funções de similaridade e extratores) a serem utilizados pelo oráculo (DELAMARO et al., 2008).

A Figura 9 mostra um descritor textual válido para a ferramenta. Exemplos e metodologia utilizada para a confecção desta descrição textual, foco maior do trabalho, serão apresentados no capítulo seguinte.

Figura 9 - Descrição Textual para *Oráculo gráfico*

```
similarity Euclidean

extractor area  { thr = 0 window = [25.25 , 300, 200]}

extractor Forma { }

precision 0.05
```

Obtém-se, assim, um ambiente que possibilita a instalação de extratores e funções de similaridade particulares. Dessa forma, os critérios do teste que serão realizados estarão

sujeitos ao processamento de imagens para confrontar o resultado da execução de um programa que tenha saída gráfica com as características de uma imagem modelo.

2.2 Tecnologias Utilizadas

Neste trabalho utiliza-se a linguagem de programação Java (SUN, 2008) em função de sua portabilidade e suas técnicas que ajudam a escrever programas que podem ser executados, com pouca ou nenhuma modificação, em uma variedade de computadores. Outra característica que pesou na escolha do Java como linguagem de programação foi a disponibilidade de ricas bibliotecas para as mais diversas áreas de pesquisa (DEITEL ; DEITEL, 2005).

Em complemento a esta linguagem de programação foi utilizada a API (*Application Program Interface*) JAI (*Java Advanced Image*) (JAI, 2008), que facilita a manipulação de imagens, implementando uma grande quantidade de operadores e transformações sobre elas (OLIVEIRA et al., 2008).

2.3 Instalações de Extratores de Características de Imagens e Funções de Similaridade

A flexibilidade da estrutura possibilita que extratores das mais variadas características de imagens possam ser instalados na ferramenta e usados em diversos testes e o mesmo acontece com funções de similaridade entre imagens. Isso ocorre porque é fácil a adaptação de classes de funções de similaridade e extratoras de características para sua utilização na estrutura.

A instalação de um pacote de classes acessórias na ferramenta, nada mais é do que a realização de uma cópia do pacote principal das classes Java de tal extrator ou função de similaridade para um diretório de conhecimento da estrutura, ou núcleo que administrará o ambiente.

2.3.1 Tipos de Instalações

Para evitar problemas de restrições de uso de classes em casos particulares, a ferramenta disponibiliza dois tipos de instalação de pacote de classes acessórias, são elas:

Instalações Globais e Instalações Locais.

Rotulam-se *Instalações Globais* aquelas em que o pacote de classes acessórias é copiado para um diretório específico dentro da ferramenta de modo que possa ser mais facilmente manipulado.

Por *Instalações Locais* compreende-se a realização da cópia do pacote de classes para um diretório de conhecimento do núcleo administrador do ambiente, dentro do diretório HOME do usuário corrente. Este tipo de instalação visa a sanar todo e qualquer tipo de problema relacionado aos acessos de pacotes de classes de instalações globais durante uma execução de comando do usuário ao *oráculo gráfico*.

2.4 Controle de Acessórios Instalados

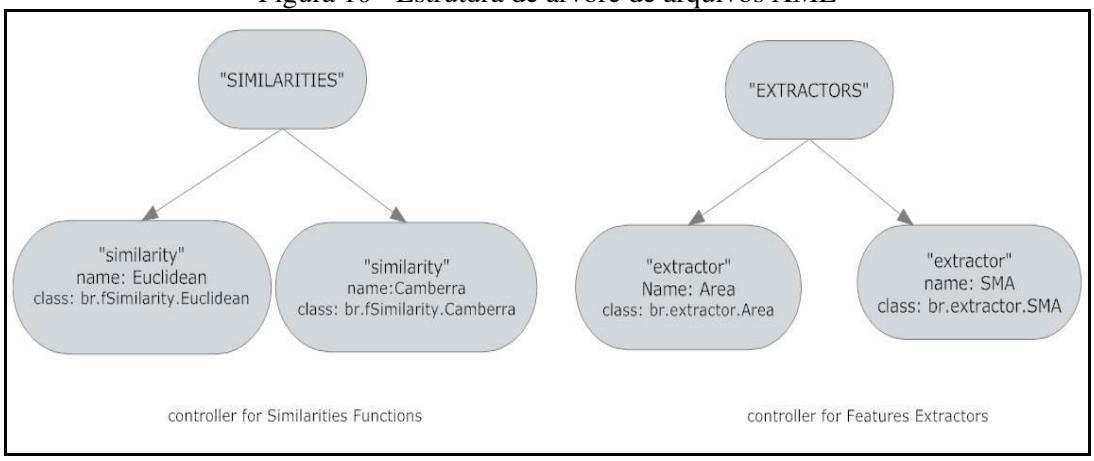
Para que o ambiente desenvolvido obtivesse pleno controle de todos acessórios à disposição para a criação de oráculos gráficos, foi idealizada e posta em prática uma estrutura em arquivos XML (*eXtensible Markup Language*). Resume-se XML a uma linguagem de marcação, ou seja, conjunto de códigos aplicados a um texto ou a dados, com o fim de adicionar informações particulares sobre esse texto ou dado, ou sobre trechos específicos (McLAUGHLIN, 2001). Tal linguagem foi escolhida em função de sua portabilidade, da disposição de API's Java que facilitam a manipulação e estruturação destes tipos de arquivos e por ser extensível, ou seja, não se limita a determinado número de *tags*.

A arquitetura de um arquivo XML permite sua disposição na forma estrutural de árvore, o que facilita a navegação entre os nós para a adição ou recuperação de dados (McLAUGHLIN, 2001). A API utilizada para a manipulação de arquivos XML no programa foi a JDom (JDOM, 2008), que disponibiliza um conjunto de classes e métodos que fornecem meios para acesso aos arquivos XML por meio de uma estrutura de árvore construída especificamente para a linguagem Java (McLAUGHLIN, 2001, p.141).

Para facilitar ao máximo a administração do ambiente por meio deste tipo de arquivo, foi escolhida uma estrutura XML simples, porém suficiente para gerir o fluxo de acessórios na ferramenta. A estrutura consiste em uma *tag* raiz, que deve fazer referência ao tipo de acessório administrado por aquele arquivo. Sendo assim, quando um arquivo mantiver dados a respeito de extratores de características de imagens, ele terá a *tag* raiz denominada de "EXTRACTORS". Em contrapartida, quando um arquivo mantiver dados acerca de funções de similaridade, ele deverá ter a *tag* raiz denominada de "SIMILARITIES".

Dando continuidade à estrutura da árvore, surgem os nós filhos da raiz, e cada um destes representa um acessório instalado corretamente e à disposição de uso. Tais nós, ou *tags* filhas da *tag* raiz, são nomeadas de *"similarity"* ou *"extractor"* para referência à função de similaridade ou ao extrator de característica respectivamente. Elas têm em particular dois atributos: *name* e *class*. O primeiro faz referência ao nome utilizado durante o processo de instalação e o segundo é utilizado para guardar pacote e nome da classe principal do acessório. A Figura 10 ilustra exemplos da estrutura desenvolvida para esses arquivos que servem como uma referência ao banco de acessórios.

Figura 10 - Estrutura de árvore de arquivos XML



As estruturas observadas na Figura 10 correspondem às árvores dos arquivos XML ilustrados na Figura 11. Conclui-se que o ambiente, durante uma operação do usuário, ou seja, um ajuste de oráculo, navegará por esses nós buscando recuperar informações a respeito das diversas classes instaladas.

Figura 11 - Exemplos de arquivos XML *controller*

```
<?xml version="1.0" encoding="UTF-8" ?>
- <SIMILARITIES>
    <similarity name="Camberra" class="br.fSimilarity.Camberra" />
    <similarity name="Euclidean" class="br.fSimilarity.Euclidean" />
  </SIMILARITIES>
```

controller.xml for Similarities Functions

```
<?xml version="1.0" encoding="UTF-8" ?>
- <EXTRACTORS>
    <extractor name="SMA" class="br.extractor.SMA" />
    <extractor name="Area" class="br.extractor.Area" />
  </EXTRACTORS>
```

controller.xml for Features Extractors

Em função da possibilidade de haver duas formas de instalações de acessórios e por existirem dois tipos de acessórios a serem administrados pelo ambiente gerado, lançou-se mão de quatro arquivos denominados de *controller.xml*. Estes arquivos realizam o controle dos seguintes pacotes de classe; classes extratoras locais, classes extratoras globais, classes de funções de similaridade locais e classes de funções de similaridade globais, respectivamente.

Conclui-se, então, que, para cada tipo de acessório, extrator de característica ou função de similaridade, o programa mantém um arquivo no formato XML de estrutura simples, que contém os nomes de instalação dos acessórios e os nomes das classes principais dentro do pacote de classes que está instalado. Cada vez que o núcleo do sistema é executado, ele consulta esses arquivos para saber quais são os extratores e funções de similaridade disponíveis para uso (OLIVEIRA et al., 2008).

2.5 Instalações de Extratores de Características de Imagens

Etapa fundamental do CBIR, a extração de características de imagens consiste de uma medida numérica que tem por objetivo capturar determinada propriedade visual da imagem, podendo ser de escopo global, ou seja, de toda a área da imagem, ou local, em que se focam pequenas regiões da imagem (BUGATTI, 2008, p.12). Alguns extratores de características aceitam parâmetros ajustáveis de acordo com a necessidade particular de uma recuperação. Nas seções seguintes será mencionado o modo que isso foi utilizado como um incremento para a flexibilidade dada à estrutura durante o ajuste de um oráculo.

A adaptação de tais tecnologias para o funcionamento correto na estrutura pode ser obtida de maneira simples. Basta fazer com que classes Java contendo tais extratores implementem determinada interface Java, chamada de *IExtractor*. Uma interface Java descreve um conjunto de métodos que podem ser chamados em um objeto, porém não são fornecidas implementações concretas para tais métodos (DEITEL ; DEITEL, 2005, p.337). No trabalho, a referida interface consiste basicamente de alguns métodos que facilitam a manipulação dos extratores escolhidos pelo usuário para serem usados na geração de oráculos gráficos e realização de testes de recuperação de imagem. A Figura 12 representa a interface *IExtractor.*

Figura 12 - Interface Java IExtractor

```
1.  package br.oraculos.plugins;
2.
3.  import javax.media.jai.*;
4.
5.  public interface IExtractor {
6.
7.      String getName();
8.
9.      void setProperty(String propertyName, Object propertyValue) ;
10.
11.     Object getProperty(String propertyName);
12.
13.     Object[] getProperties();
14.
15.     String[] getPropertiesNames();
16.
17.     double computeValue(PlanarImage imageLoaded);
18. }
```

O método IExtractor.getname() é responsável identificar o pacote de classes na estrutura. IExtractor.setProperty(String, String) é a sub-rotina que visa a ajustar um parâmetro do extrator de características de imagens de acordo com as especificações do usuário. O método IExtractor.getProperty(String) retorna um objeto contendo a instância de determinado parâmetro ajustável. IExtractor.getPropertiesNames() deve retornar um vetor com o nome de todos os parâmetros ajustáveis do extrator. Por fim, a interface exige a implementação de um método de escritura IExtractor.computeValue(PlanarImage) e de retorno do tipo primitivo Java double. Espera-se deste método a principal função de um extrator de características: a utilização de seus parâmetros aplicados em uma imagem para o cômputo de um valor quantitativo.

2.6 Instalações de Funções de Similaridade

Todo esquema CBIR tem como componente essencial a função para medida de similaridade, chamada também de distância de similaridade. É importante salientar que a recuperação de imagem baseada em conteúdo não realiza buscas exatas de imagens. Isso se dá pelo fato de fundamentar-se em um cálculo de similaridade entre uma imagem de consulta e as imagens contidas em um banco de dados. Como já foi citado, o grau de similaridade entre as imagens é obtido pela comparação entre seus vetores de características (BUGATTI, 2008).

Existem diversas distâncias de similaridade apresentadas por pesquisadores da área de processamento de imagem. Exemplos comuns são as medidas *Euclidiana, Camberra, Minkowski, Manhattan, Chebyshev, Mahalanobis*, entre outras que são descritas no trabalho

de Ferreira (2005).

A instalação de um pacote de classes correspondente a uma função de similaridade na ferramenta segue o mesmo mecanismo utilizado para instalações dos extratores. Especificamente, a interface Java que deve ser implementada para este tipo de acessório é a *ISimilarity,* ilustrada pela Figura 13. Tal interface consiste de um conjunto de métodos que visam a identificar, localizar e utilizar classes extratoras no ambiente de um *oráculo gráfico*.

Figura 13 - Interface Java *ISimilarity*

```
1.  package br.oraculos.plugins;
2.
3.  import javax.media.jai.*;
4.  import java.util.Vector;
5.
6.  public interface ISimilarity {
7.
8.      String getName();
9.
10.     void addExtractor(IExtractor extractor);
11.
12.     double computeSimilarity(Vector vectModel, Vector vectTesting);
13.
14.     Vector getVectorSimilarity(PlanarImage image);
15.
16.     void removeExtractors(Vector vectorOfExtractors);
17.
18.                             }
```

Resumidamente, o método ISimilarity.getName() carrega a função de nomear a classe que o implementa, ou seja, é usado para dar um nome à função de similaridade dentro da estrutura. O método ISimilarity.addExtractor(IExtractor) é responsável por adicionar um novo objeto de extrator de característica à recuperação de imagem. O método ISimilarity.computeSimilarity(Vector, Vector) é o método principal da interface e deve retornar a computação da similaridade entre dois vetores de características. Finalizando, a interface ISimilarity.getVectorSimilarity(PlanarImage) é o método encarregado de extrair um vetor de características da imagem recebida como parâmetro.

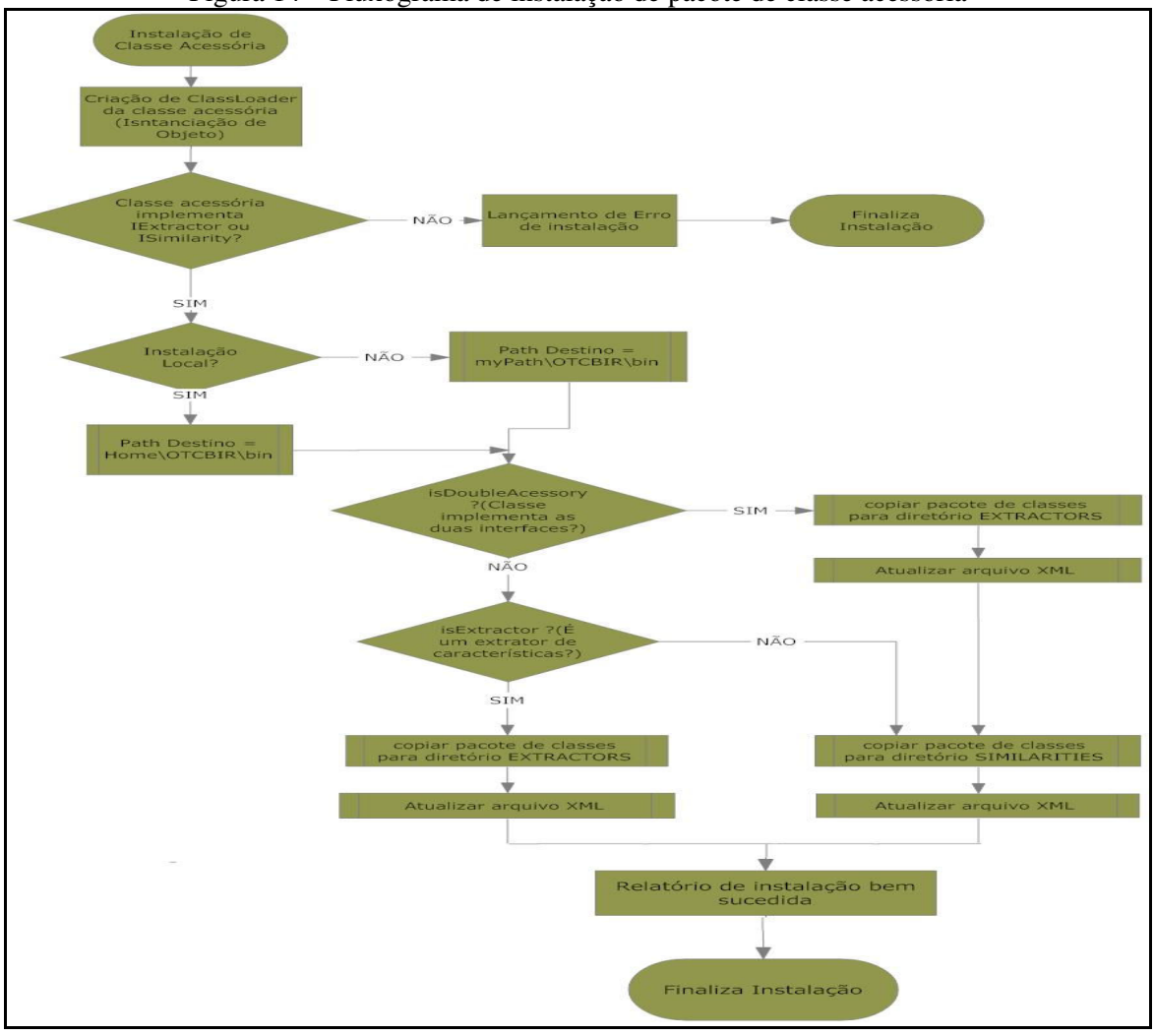
2.7 Instalações: Organização Interna

O mecanismo de instalação por protocolos dispostos em forma de interfaces Java consiste em um dos fatores que deram flexibilidade à estrutura. Como foi visto, o CBIR tem sido usado em diversas áreas e cada uma, em particular, requer um conjunto de distâncias de similaridade e extratores de características de imagens próprios. Portanto a estrutura pode

apoiar a todas elas sem maiores problemas adaptativos.

Salienta-se que a utilização de interfaces Java também é explorada por alguns desenvolvedores como uma forma de herança múltipla, que resumidamente possibilita o compartilhamento de atributos e operações de duas ou mais classes em uma subclasse (DEITEL ; DEITEL, 2005). No contexto do trabalho aqui apresentado, isso implica dizer que existe a possibilidade de que um objeto em processo de instalação implemente, simultaneamente, *IExtractor* e *ISimilarity*. Quando a ferramenta estiver diante de tal situação, o acessório será considerado duplo, e será feita a instalação de uma função de similaridade e de um extrator de características de forma normal. Ainda não foi estudada uma forma de reverter este recurso da linguagem Java em benefícios para o ambiente de oráculos gráficos tampouco para o CBIR, no contexto do protótipo, de uma forma geral.

Figura 14 – Fluxograma de instalação de pacote de classe acessória



A Figura 14 ilustra um fluxograma que contém os principais processos e tomadas de

decisão durante a realização da instalação de um acessório na ferramenta. Nota-se que a instalação de um acessório duplo é tratada de forma natural pelo ambiente.

Observa-se que um processo de instalação deve ser iniciado com a instanciação de um objeto da classe principal do acessório. Desta forma, é possível que o objeto seja explorado sobremaneira e, por conseguinte, não é preciso que um usuário informe para a estrutura qual é o tipo do acessório. Resume-se que alguns métodos, todos de simples funcionamento e entendimento, precisaram ser implementados para extrair o tipo do acessório. Dois destes métodos são ilustrados pela Figura 15.

Figura 15 - Métodos do processo de instalação

```
1.  public static boolean isExtractor (Object classTested)
2.  {
3.
4.  if ((classTested  instanceof IExtractor)&&
5.  (!(classTested instanceof ISimilarity)))
6.
7.  isExtractor = true;
8.
9.  else
10. isExtractor = false;
11.
12. return isExtractor;
13. }
```

method: static Boolean isExtractor(java.lang.Object)

```
1.  public static boolean doubleAcessory (Object classTested)
2.  {
3.
4.  if ((classTested  instanceof IExtractor)&&
5.     (classTested instanceof ISimilarity))
6.
7.  doubleAces = true;
8.
9.  return doubleAces;
10. }
```

method: static Boolean isExtractor(java.lang.Object)

No diagrama ilustrado na Figura 14, fica claro que o primeiro requisito para a realização de uma instalação qualquer na ferramenta é o acessório implementar pelo menos uma interface requerida pelo sistema. No mínimo um protocolo, ou seja, um contrato de métodos em forma de interface Java, deve ser assinado pela classe principal do pacote de classes em processo de instalação. A verificação destas condições é feita por meio de um conjunto de métodos e de atributos de formatos similares aos apresentados pela Figura 15. Na figura referida são apresentados dois métodos estáticos de retorno do tipo primitivo Java

boolean: isExtractor(java.lang.Object) e doubleAcessory(java.lang.Object). Ambos recebem um parâmetro do tipo Object, do pacote default Java (*java.lang*), e retornam uma característica a respeito da instância da classe principal. O primeiro retorna *true*, se o objeto, seu parâmetro, for extrator de características e o segundo retorna *true*, se receber como parâmetro um objeto que implemente *IExtractor e ISimilarity,* simultâneamente.

Verificadas as especificações particulares do objeto a ser instalado, confere-se o tipo de instalação requerida pelo usuário. Esta etapa do processo é importante, pois visa a exibir qual é o destino do pacote de classes, *diretório global ou diretório local*. Isto pode ser feito pela simples verificação de comando passado ao método de instalação.

2.8 Estrutura das Descrições para Oráculos Gráficos

Os oráculos são baseados no armazenamento dos resultados da execução de outros programas. Observa-se que a complexidade de sua automatização é diretamente proporcional à complexidade de sua saída. Dessa forma, quando a saída de processamento é complexa, em particular, no formato gráfico, sua automatização também será complexa (OLIVEIRA et al., 2008).

Já foi abordado que o objetivo maior deste trabalho é fornecer requisitos necessários para um ambiente em que o usuário possa criar seus próprios oráculos para programas com saídas gráficas (DELAMARO, 2007a). Tais oráculos devem ser o mecanismo que permite que se decida se uma execução do programa em teste está correta ou não. Como é demonstrado na Figura 8, o ambiente proposto pelo trabalho depende de uma descrição textual por meio da qual o usuário/testador pode exprimir suas especificações particulares para um teste.

Esta descrição textual deve seguir regras sintáticas para que possa ser entendida e assim utilizada de modo eficiente por ferramentas para as quais for destinada. Em resumo, ferramentas de teste podem receber essa descrição textual em formas arquivo de parâmetro.

O reconhecimento e manuseio de tais arquivos descritivos são função de um *parser* que foi implementado dentro do Parametrizador, que pode ser visualizado na Figura 8. Tal *parser*, quando reconhece uma descrição, faz a criação dos objetos a serem utilizados pelo oráculo.

2.8.1 Gramática de Descritores de Oráculos Gráficos no Formato BNF

Uma estrutura de descritor para *oráculo gráfico* simples e eficiente foi idealizada e posta em prática no trabalho. Um exemplo de tal estrutura foi ilustrado, no capítulo inicial, pela Figura 8. Para facilitar o entendimento, a Figura 16 exibe as regras gramaticais de formação do oráculo no formato BNF (*Backus-Naur Form*). Tal formato é uma maneira alternativa de se definirem linguagens livres de contexto, semelhantes a uma GLC (Gramática Livre de Contexto), entretanto permite que o lado direito das produções possua alguns operadores especiais (DELAMARO, 2004, p.12).

Figura 16 - Gramática de oráculos gráficos no formato BNF

```
1-   <unit>   :: (<fsimi>)* <EOF>
2-   <fsimi>  ::  <simiDecl> (<extractorDecl>)+  [ <precisionDecl> ]
3-  <EOF>   ::  final de arquivo
4-  <simiDecl> :: "similarity" <IDENT>
5-   <extractorDecl> ::
                    "extractor"  <IDENT>
                    "{"
                          (  <IDENT> "="(<simpleConst> | <arrayDecl>)) *
                    "}"
6-  <precisionDecl>  :: "precision"    "="  <DOUBLE>
7-  <IDENT> :: identificador válido
8-  <simpleConst> ::   <LONG> | <STRING> | <FLOAT> |
                                <DOUBLE> | <BOOLEAN> | <INT> | "null"
9-  <arrayDecl> :: "[" ( <simpleConst> | <arrayDecl> )
                          ( "," (<simpleConst> | <arrayDecl>) )* "]"
10-  <LONG> :: valor do tipo long
11-  <STRING> :: array de caracteres
12-  <FLOAT> :: valor do tipo float
13-  <DOUBLE> :: valor do tipo Double
14-  <BOOLEAN>  :: "true" | "false"
15-  <INT> :: valor do tipo int
```

Visando a facilitar o entendimento do leitor, a gramática dos arquivos descritores é exibida pela Figura 16. Um paralelo com as GLC pode ser feito durante a visualização da referida figura. Observa-se que os símbolos não terminais estão entre os caracteres comparadores (‘**<**’ e ‘**>**’) e os símbolos terminais estão entre aspas duplas (‘ **”** ’e ‘ **“** ‘).

O descritor do oráculo deve ser iniciado com a palavra reservada *“similarity”*, seguida de um identificador que represente o nome da função de similaridade ajustada. Feita a identificação da função de similaridade, devem ser identificados extratores presentes na descrição e seus parâmetros.

A descrição de um extrator no arquivo de ajuste do oráculo deve ser iniciada com a palavra reservada *"extractor"*, seguida de um identificador qualquer que represente o nome pelo qual o acessório é identificado pelo núcleo da estrutura. O ajuste de parâmetros para este determinado extrator deve ser apresentado na seqüência da seguinte maneira- abertura de chaves*"{"*, nome do acessório ajustável, sinal de atribuição *"="* e valor desejado de ajuste. Podem ser ajustados quantos parâmetros forem necessários com esse formato antes do fechamento de chaves *"}"*.

Já foi citado que o formato BNF é referência por possibilitar a utilização de alguns operadores especiais. Isso é explorado na gramática dos descritores e pode ser observado na linha 2 da Figura 16. Na ocasião, o operador +, da sentença *(<extractorDecl>)*+, reflete na declaração de, no mínimo, um extrator de características de imagens em um *oráculo gráfico*. Logo a declaração descrita no parágrafo anterior deve acontecer no mínimo uma vez, mas pode ocorrer indefinidas vezes para diferentes extratores durante um ajuste de descrição.

Haja vista que toda distância de similaridade retorna um valor numérico representante da semelhança entre duas imagens (BUGATTI, 2008), o formato da descrição do *oráculo gráfico* propõe uma variável de valor numérico para funcionar como parâmetro de recuperação de imagens em possíveis sistemas CBIR que venham a utilizar as descrições geradas pela estrutura. Tal variável é de simples declaração e inicia-se com a palavra reservada *"precision"*, seguida do símbolo de atribuição *"="* e finalizada com um valor double qualquer.

Mais uma vez utilizam-se de recursos BNF para facilitar confecção da gramática da estrutura descritora para o oráculo. Observa-se, na linha 2 da Figura 16, que a sentença *<precisionDecl>* é colocada entre abertura e fechamento de colchetes. Isso replica em uma parte da estrutura que poderá estar presente ou não durante um ajuste de *oráculo gráfico*. A Figura 9 ilustra uma descrição de *oráculo gráfico* que utiliza tal variável de precisão (*precision*) com um valor de 0.05. Uma proposta é utilizar um critério de proximidade (que é esse valor) para recuperar apenas imagens que estejam dentro de um intervalo, que inicia em zero e vai até *precision*, pré-determinado de modo particular pelo usuário durante a confecção da descrição. A Figura 17 exibe dois exemplos de descrição textual para *oráculo gráfico*.

A descrição textual de oráculo, totalmente abstrata, apresentada pela Figura 17, visa a exibir como são postas em práticas as regras definidas pela representação gramatical da BNF (Figura 16). Numa descrição informal do oráculo definido pelo trecho 2 da referida figura deve ser utilizada uma função de similaridade denominada de *"simi2"*, com dois extratores de características de imagens; *"extract3"* e *"extract4"*- este não requer ajuste de parâmetros,

enquanto aquele tem uma propriedade denominada de *"y1"* que receberá um vetor com cinco elementos de valores distintos.

Figura 17 - Exemplo de descrição textual para *oráculo gráfico*

```
similarity simi1

extractor
      extract1
      {
            v1 = 0 v2 = "STRING" v3 = 0.378E+9
            v4 = 9.0 v5 = .22 v6 = .22E-1919
      }


extractor
      extract2
      {
            x1 = 0x8fd91 x2 = 0.3893 x3 = true x4 = FALSE x5 = null
      }
```

1

```
similarity simi2

extractor
      extract3
            {y1 = [0, "STRING",  0.378E+9,  9.0,  .22E+1919 ] }

extractor
      extract4  { }
```

2

É interessante salientar que a gramática, exposta pela Figura 16 em formato BNF, permite que mais de uma descrição textual de oráculo seja ajustada simultaneamente. Desta forma, as estruturas apresentadas na Figura 17, enumeradas de 1 e 2, são exemplos de descrições textuais de *oráculo gráfico* válidas, separada ou simultaneamente. Isso se dá em função de a sentença *(<fSimi>)* *, exposta na linha 1 da Figura 16, representar que podem ser reconhecidas zero ou inúmeras descrições. Isso implica dizer que o trecho identificado como 1 (Figura 17) é uma descrição válida tanto quanto o trecho 2 (Figura 17), e não menos que os trechos 1 e 2 juntos e associados. Formas de associar mais de uma descrição textual dentro dos oráculos podem ser estudadas e exploradas em pesquisas futuras com ambientes administradores de oráculos gráficos.

2.8.2 JavaCC: de BNF para Código Java

O formato BNF, utilizado para representações de GLC's, exige muita atenção e destreza no tocante à implementação de um *parser* reconhecedor para suas especificações no Parametrizador do ambiente. Em uma linguagem orientada a objetos, como é o caso da utilizada neste trabalho, talvez a presença de algum outro modelo formal que auxilie tal tarefa, seja quase indispensável.

O trabalho apresentado tem como função criar a base de um ambiente de *oráculo gráfico*. Dentro do contexto dessa base, incluem-se não só a geração arquivos descritores de oráculos, bem como seu reconhecimento. Em adição a isso há uma funcionalidade implementada no Comparador da estrutura, que visa a realizar um teste de recuperação entre duas imagens, modelo e teste, a partir de arquivos descritores gerados pela própria ferramenta. Sendo assim os descritores de oráculos servirão como arquivo parâmetro para tal funcionalidade.

Por meio do parametrizador, comandos passados ao protótipo possibilitam ajustar o oráculo de forma que qualquer função de similaridade, previamente instalada na ferramenta possa ser associada a quaisquer extratores de características de imagens com parâmetros particulares. Como já foi citado, o parametrizador lê um arquivo com a descrição do oráculo e passa essa descrição para o núcleo Comparador da ferramenta (OLIVEIRA et al, 2008), para o que um confiável reconhecedor de descrições de oráculos gráficos teve de ser implementado.

É nesse contexto que é inserida a ferramenta JavaCC (JAVACC, 2008). Este programa é um gerador de compiladores, mais precisamente um gerador de analisador sintático (DELAMARO, 2004, p.7). A ferramenta toma como entrada uma gramática, cuja descrição é uma associação entre o formato BNF e código Java, disposta em um arquivo de extensão *".jj",* e transforma-a em um programa Java capaz de analisar um arquivo e processar se o mesmo satisfaz ou não as regras especificadas nessa gramática.

A ferramenta foi escolhida pelo fato de que o programa gerado por ela realiza a análise sintática top down (descendente) e fornece base para definição de analisadores léxicos e sintáticos de uma só vez (DELAMARO, 2004). A Figura 18 ilustra um trecho, referente ao analisador sintático, de um arquivo .JJ, para geração de um código Java reconhecedor de descrições de oráculos gráficos no formato de arquivos parâmetros.

O trecho de código apresentado pela Figura 18, guardadas as devidas proporções, deve ser visto como uma conversão da gramática definida, em formato BNF, para a descrição de oráculo textual (Figura 16). Como se pode- perceber, há grande semelhança entre os formatos. Logo a tarefa do programador será muito facilitada quando o gerador de analisador sintático JavaCC for utilizado para implementações de reconhecedores deste caráter. Adiciona-se que o trecho apresentado faz referência às definições pertinentes ao analisador sintático. Grandezas léxicas, como palavras reservadas (similary ou extractor) e tokens (=, {, }, [,]) devem ser definidas no mesmo arquivo, antes das referências sintáticas, e foram omitidas na Figura 16.

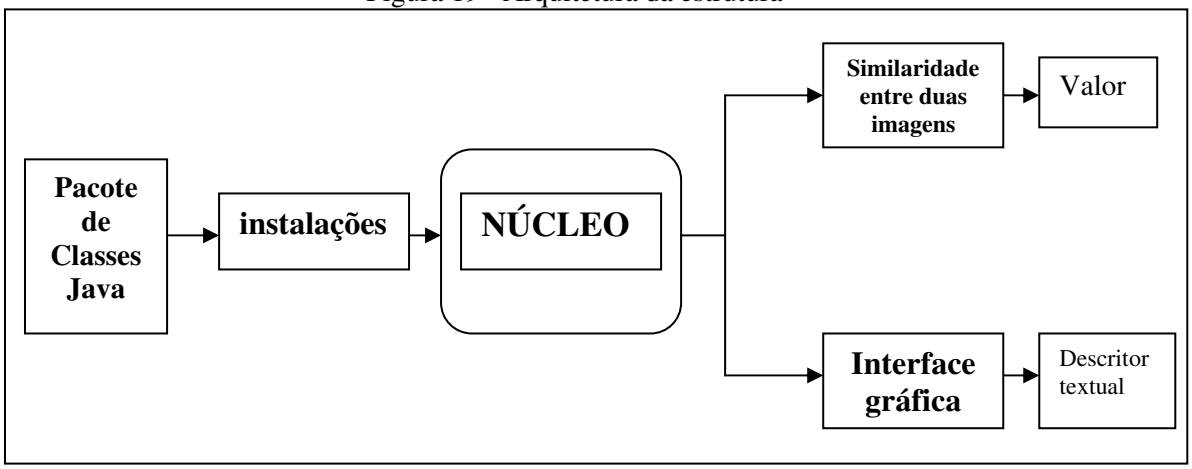
Figura 18 - Trecho de gramática de arquivo JJ reconhecedor de arquivos descritores de oráculos gráficos

```
void unit() :
{
}
{
     ( fsimi() )* <EOF>
}

void fsimi():
{
}
{
   simidecl() ( extractdecl() )+ [precisonDecl()]
}

void simidecl():  // declara a função de similaridade a usar
{
}
{
   <SIMILARITY> <IDENT>
}

void extractdecl(): // declara os extratores a usar
{
}
{
       <EXTRACTOR> <IDENT>
       <LBRACE>
             ( <IDENT> <ASSIGN> ( simple_const() | array() ) )*
         <RBRACE>
}

void precisionDecl() //declara a prcisão de recuperação
{
}
{
       <PRECISION> <ASSIGN> <DOUBLE_CONSTANT>
}
```

2.9 Visão Geral do Protótipo

O protótipo desenvolvido no trabalho, em uma visão global, apresenta relacionamentos interessantes. Salienta-se que tais relacionamentos são base da arquitetura do ambiente que é descrita no Capítulo 1 e ilustrada na Figura 8. Tais relacionamentos são ilustrados na Figura 19 e constituem pacotes ou módulos distintos dentro de um mesmo ambiente.

Figura 19 - Arquitetura da estrutura



Destacam-se três módulos principais, que visam a realizar as diversas funcionalidades do programa e que se relacionam diretamente com a arquitetura do ambiente: São elas:

- *Instalação de Objetos:* pacote de classes de funções de similaridade e/ou extratores. São viabilizadas por meio do núcleo do Comparador.
- *Geração de Descritores de Oráculos Gráficos:* Arquivos parâmetros. Gerados apenas por interface gráfica.
- *Testes de Recuperação:* Realizados entre duas imagens. Possibilitados por objetos instanciados pelo *parser* do Parametrizador e da resultante de comparação entre as imagens, feita pelo núcleo do Comparador.

Os dois primeiros módulos de funcionalidades, Instalações de objetos e Geração de Descritores Textuais de Oráculo, já foram abordados nas subseções anteriores do capítulo. O módulo *Testes de Recuperação* resume-se em uma funcionalidade que visa a aplicar as definições de um, arquivo parâmetro, descritor textual de *oráculo gráfico*, definido na ferramenta ou até mesmo importado por ela, em duas imagens, computando assim a similaridade entre elas. Logo, não se trata de um processo CBIR, pois o oráculo não age em uma base de dados de consulta, mas em apenas uma imagem de consulta.

Salienta-se que os módulos definidos anteriormente carregam uma dependência hierárquica. Embora ocorram separadamente, devem obedecer a uma ordem lógica.

A proposta de ordem lógica é a seguinte:

1. Instalação de Pacotes de Classes
2. Geração de Descritores de Oráculos Gráficos
3. Testes de Recuperação de Imagem

O que foi apresentado implica dizer que se não forem instalados pacotes de classes no protótipo, não é possível que sejam realizadas as gerações arquivos descritores particulares por parte do usuário. A impossibilidade da geração de parâmetros inviabiliza a realização do Teste de Recuperação entre imagens.

O entendimento da estrutura global do trabalho passa pelo modo de interação e comunicação entre os módulos. Infere-se que a estrutura toda é controlada por um núcleo que sempre será acionado, em toda e qualquer operação realizada no protótipo, seja via interface gráfica ou via linha de comando.

Define-se o referido núcleo como sendo um conjunto de classes Java que identifica qual a interação do usuário com o programa, direciona seus parâmetros e retorna resultados.

Em linhas gerais, o núcleo faz interface entre todas as funcionalidades da estrutura, provendo os serviços requeridos por cada uma delas. Um dos papéis do núcleo é a filtragem das ações que são realizadas pelo usuário, ou seja, identificação e direcionamento de serviços requeridos durante uma execução, seja ela por linha de comando ou via interface.

A Figura 19 ilustra a forma de relacionamento entre os módulos e o núcleo do protótipo. Ao observar a referida figura, nota-se que apenas a funcionalidade de geração de descritores textuais é restrita à interface gráfica do sistema.

Ressalta-se que a realização de instalações de classes Java e realizações de testes de recuperação entre duas imagens podem ser realizadas pela execução do programa por intermédio da interface gráfica e via linha de comando, sendo necessária a passagem de alguns parâmetros que serão detalhados nas seções seguintes.

Resume-se a divisão de tarefas do núcleo, fazendo interface com cada módulo de operações, da seguinte maneira:

- *Instalações de Objetos*: acesso ao núcleo do Comparador (Figura 8). Disparo de processos que validam, tipificam (extrator ou distância similaridade), nomeiam e copiam as classes na ferramenta. Quando bem sucedidas todas essas operações, aciona-se a atualização dos arquivos XML, que objetiva notificar outros módulos da presença de um novo objeto. Retorno de relatório de instalação ao usuário.
- *Geração de Oráculos Gráficos*: disparo de conjunto de processos que fazem uma completa varredura da instância da interface gráfica obtendo dados acerca dos ajustes feitos pelo usuário. Gravação de *oráculo gráfico* em um arquivo texto. Exibição de *oráculo gráfico* ou mensagem indicativa de erro.
- *Testes de Recuperação de Imagem*: acesso ao *parser* do Parametrizador (Figura 8). Ativação de processos responsáveis por reconhecer e parametrizar tudo que foi estabelecido pelo arquivo parâmetro. Disparo de processos de objetos de extratores para obtenção de vetor de características das duas imagens e cômputo do valor de similaridade entres elas (Comparação).

As sub-seções seguintes visam a apresentar, de um modo mais prático, as funcionalidades disponibilizadas no projeto, bem como a interface gráfica desenvolvida para o sistema.

2.9.1 Funcionalidades do Sistema

Dentro do cenário apresentado pela seção anterior, que ilustra módulos do sistema, pode-se inferir que a ferramenta é composta por um conjunto de funcionalidades, inseridas em tais módulos, que visam a apoiar e compor o objetivo maior do sistema, que é a de construir uma estrutura que utilize técnicas de CBIR para construção de oráculos gráficos.

A Tabela 2 apresenta essas funcionalidades associadas ao modo pelo qual o usuário pode acessá-las (interface gráfica ou comandos) e uma breve descrição do contexto no qual elas se inserem dentro do trabalho.

Tabela 2 – Funcionalidades do sistema

FUNCIONALIDADE	DISPONIBILIDADE	DESCRIÇÃO
Instalações de novos objetos	• Interface Gráfica • Linha de Comando	Instalar novos objetos (funções de similaridade ou extratores de características) na ferramenta.
Geração e armazenamento de descritores de oráculos gráficos	• Interface Gráfica	Consiste na leitura uma instância da interface para a geração de um arquivo com o formato especificado que sirva de parâmetro ao *oráculo gráfico*. Arquivos parâmetro podem ser gerados e depois salvos em disco.
Teste de Recuperação entre duas imagens	• Interface Gráfica • Linha de Comando	Para ilustrar o funcionamento da comparação realizada por duas imagens por meio de um descritor de oráculo, gerado pelo protótipo ou até mesmo importado, foi desenvolvido um mecanismo que extrai a similaridade entre duas imagens, modelo e testada. Para tanto é necessário um arquivo descritor de *oráculo gráfico* e as referências das imagens.
Importação de descritores de Oráculos Gráficos	• Interface Gráfica	Esta funcionalidade visa a permitir que descritores de oráculos gráficos, externos à estrutura, possam ser utilizados em testes de recuperação entre duas imagens. Obviamente para que o arquivo seja carregado com sucesso para a estrutura ele deve respeitar a gramática dos descritores. É necessário que os objetos referenciados por ele estejam instalados no protótipo.
Salvamento de imagens para realização de testes de recuperação	• Interface Gráfica	Salva a imagem modelo carregada na tela principal da interface em um diretório particular do ambiente (OTCBIR/myImages). Isso visa a facilitar que imagens modelo, uma vez utilizadas no ambiente, possam ser resgatadas para nova utilização.
Pesquisa sobre acessórios	• Linha de Comando	Verificação de quais são os acessórios que estão disponíveis para que ele os utilize da forma como desejar para customizar um *oráculo gráfico* qualquer.

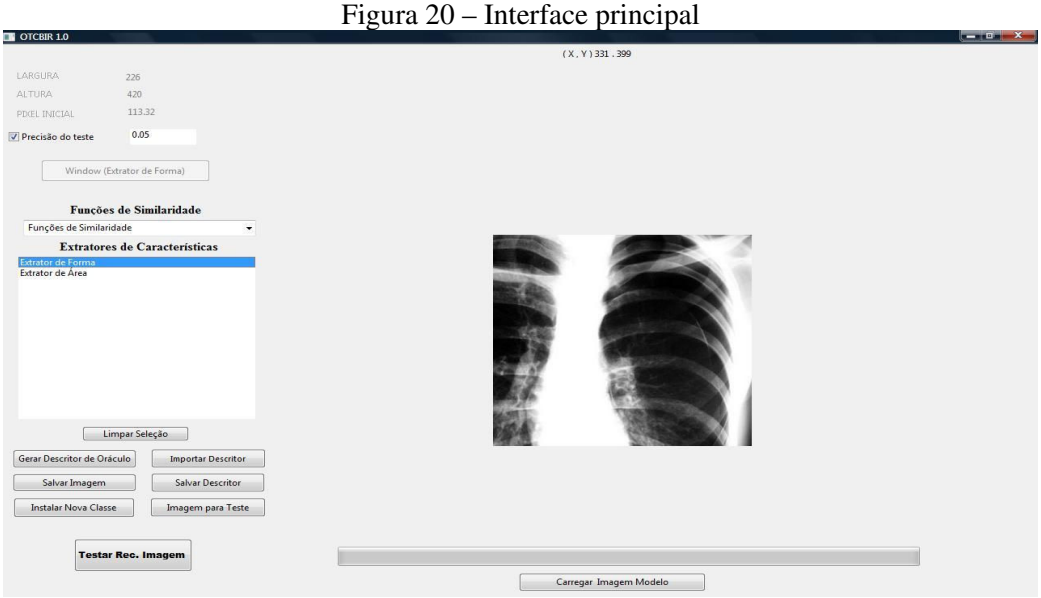
As sub-seções seguintes visam a apresentar e exemplificar, de um modo mais prático, as funcionalidades disponibilizadas no trabalho, bem como a interface gráfica desenvolvida para o sistema.

2.10 Implementação da Interface Gráfica do Sistema

Em função do que foi apresentado até aqui, percebe-se que o trabalho realizado envolve conceitos multidisciplinares e engloba diversas operações que visam a atingir um objetivo comum: criar a base para um ambiente de *oráculo gráfico*. Como pode ser observada na Figura 19, a funcionalidade de geração de arquivo parâmetro abre um leque de diversas outras funcionalidades para o ambiente.

É importante salientar que instalações de novos objetos (similaridades ou extratores) e reconhecimento de arquivos parâmetro, por meio do *parser* do Parametrizador, foram implementados, primeiramente, para serem executados em linha de comando, mas a geração de arquivos descritores para oráculos gráficos de forma dinâmica e flexível trouxe a necessidade de uma interface gráfica convincente. Em função das referidas funcionalidades já estarem prontas antes de idealização da interface, elas puderam ser importadas com facilidade e eficiência para o meio gráfico. Em adição a elas, foram criadas algumas outras operações visando a auxiliar o operador da ferramenta.

Figura 20 – Interface principal



No contexto inserido pela Figura 20, que ilustra a interface principal da ferramenta, são exibidas diversas operações e ajustes que podem ser realizados via interface gráfica. É importante salientar que apenas uma funcionalidade é exclusivamente disponibilizada em linha de código. Trata-se da pesquisa por objetos, funções e extratores instalados corretamente na ferramenta. Isso se dá pelo fato de que, quando um acessório estiver instalado e pronto para ser usado no ambiente, seu nome de identificação pode ser visualizado na tela principal da interface do programa, fato que dispensa a funcionalidade da pesquisa.

Resume-se que um comando de pesquisa dado à ferramenta deve ter um vetor de Strings passado como parâmetro para o método principal da classe br.oraculos.oraculo.Main, com a seguinte sintaxe:

- "search" - corresponde ao parâmetro entendido pela estrutura como comando de busca (args[0]);
- "extractor" ou "function" – trata-se do parâmetro que determina o tipo de acessório que está sendo buscado nos arquivos XML comandados pela estrutura (args[1]);
- "nome_do_acessorio" - deve corresponder ao nome de instalação do item buscado (args[2]);

A estrutura permite também a busca por todos os itens de determinado tipo de acessório. Esse tipo de busca retorna uma lista de dados contendo o nome de instalação e a classe principal de todos os acessórios de determinado tipo passado como parâmetro à estrutura. Os comandos de busca são similares aos comandos da pesquisa por nome. No entanto, no comando que faz referência ao nome do item a ser buscado, deve ser passado como parâmetro o nome "*all*", para referenciar a todos os acessórios (OLIVEIRA et al., 2008).

2.11 Instalando Novos Objetos

Toda a parte metodológica acerca de instalações já foi detalhada, no entanto nada foi demonstrado a respeito. A presente sub-seção visa a preencher esta lacuna demonstrando de forma prática a flexibilidade atingida com instalações de novos acessórios à estrutura.

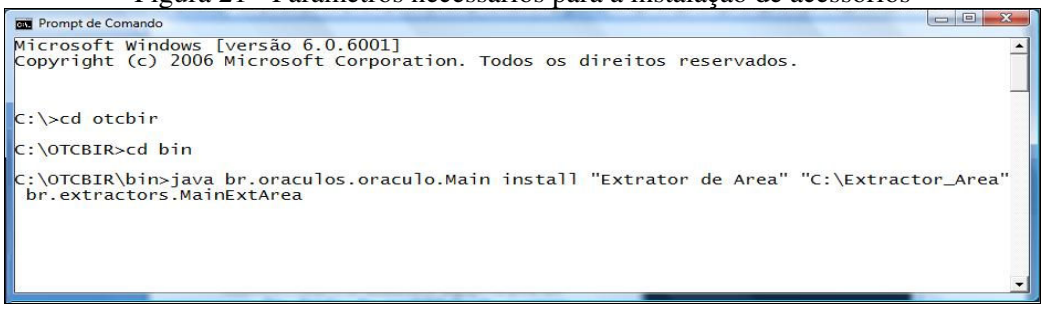
2.11.1 Instalações Via Linha de Comando

Para realizar a instalação de um acessório, via linha de comando, é necessário que sejam passados alguns argumentos ao método Main.main(), do pacote “br.oraculos.oraculo” da ferramenta. Argumentos e respectivas descrições são exibidos na seqüência.

- *"install"* - corresponde ao comando de instalação de acessórios na estrutura, logo deve ser o primeiro parâmetro do programa (args[0]); Caso seja requerida uma instalação local, o argumento deve ser *“localinstall”*;
- "Nome_do_Acessório" - corresponde ao nome pelo qual o acessório será identificado na ferramenta. Deve ser inédito para que não haja conflito com outro acessório instalado. Este deve ser o segundo argumento de chamada do programa (args[1]);
- “Diretório da raiz do acessório" - corresponde ao caminho em disco até o diretório raiz do acessório a ser instalado. O caractere "/" deve ser usado como separador de pastas, este deve ser o terceiro parâmetro passado à estrutura quando se deseja instalar um acessório local (args[2]);
- "Nome completo da classe principal do acessório" - trata-se do nome da classe, usando a notação de “.” para separar pacotes e classe. Este deve ser o quarto e último argumento exigido para a instalação bem sucedida de um extrator de características de imagem ou uma função de similaridade (args[3]);

A Figura 21 ilustra como podem ser passados os comandos de instalação de acessórios globais à estrutura.

Figura 21 - Parâmetros necessários para a instalação de acessórios



No exemplo, um extrator de características de imagens denominado pelo usuário de “Extrator de Área” e com classe principal br.extractors.MainExtArea é instalado. Isso implica informar ao sistema que tal classe implementa IExtractor e/ou ISimilarity. Após o reconhecimento do comando, a estrutura verificará qual interface é implementada, ou até

mesmo se as duas interfaces são implementadas pela classe referenciada, como já foi visto. Caso isso se confirme, ela fará a cópia do pacote inteiro para um diretório de seu conhecimento e a instalação é concluída com sucesso (OLIVEIRA et al., 2008).

2.11.2 Instalações Via Interface Gráfica

As instalações de objetos, similaridades ou extratores, via interface gráfica, são realizadas de maneira muito simples e confortável para o usuário em comparação com instalações via comandos. Para que um processo de instalação seja iniciado basta que o usuário utilize o mouse para clicar no botão "Instalar Nova Classe", disponível na tela principal do ambiente. Uma caixa de diálogo igual à apresentada pela Figura 22 será aberta.

Figura 22 - Caixa de diálogo de instalação

Quanto a caixa de diálogo de instalação é muito simples e auto-explicativa, o usuário pode escolher o tipo de instalação, local ou global, acionando com o mouse o radioButton referente ao tipo desejado. Para o ajuste do nome do objeto, deve ser utilizado o campo de texto rotulado de "Nome". Para selecionar o caminho do diretório principal do pacote de classes a ser instalado, o caminho para o referido pode ser digitado manualmente. O botão "search", representado pela figura de uma lupa, pode ajudar nesse tipo de ajuste, pois abre uma caixa de diálogo com um navegador de pastas, que permite ao usuário ajustar o diretório de modo seguro. O último dado requerido é uma referência para a classe principal do acessório e pode ser ajustada no campo "Classe Principal". Para finalizar uma instalação, o usuário deve clicar no botão "Instalar Classe".

Como pôde ser observado, os dados requeridos para instalações, via linha de comando, são muito similares aos campos de instalações globais. Isso ocorre porque os mecanismos daquele são aproveitados neste. Isso implica dizer que as classes de instalação desenvolvidas para linha de comando foram reusadas na interface gráfica, ou seja, foi

aproveitado o artifício do reuso de código proporcionado pela linguagem Java.

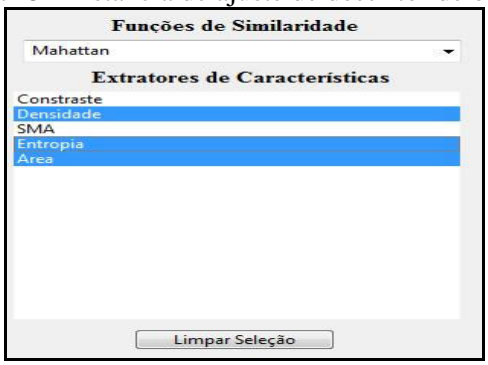
2.12 Mecanismo de Geração de Descritor para *Oráculo gráfico*

A geração de descritores de oráculos gráficos (arquivo parâmetro) é disponibilizada somente via interface gráfica e depende diretamente dos acessórios instalados na ferramenta. Isso implica dizer que caso não tenha sido instalada nenhuma função de similaridade no ambiente será impossível a geração de oráculos. O mesmo problema irá acontecer caso a estrutura interna não contiver nenhum extrator de características de imagem em seu domínio.

Quando o ambiente do trabalho dispuser de classes extratoras e de funções de similaridades para sua utilização nas mais diversas funcionalidades, os nomes de tais objetos irão aparecer na janela principal da interface. Funções de similaridade, por serem unitárias em um *oráculo gráfico*, são apresentadas em uma estrutura de Combo, que faz com que apenas uma função esteja selecionada em uma instância qualquer da interface. Extratores de características de imagens, pelo fato de não serem unitários em um *oráculo gráfico*, são exibidos em uma estrutura de lista e sempre estão visíveis para o usuário, sendo identificados pelo nome de instalação.

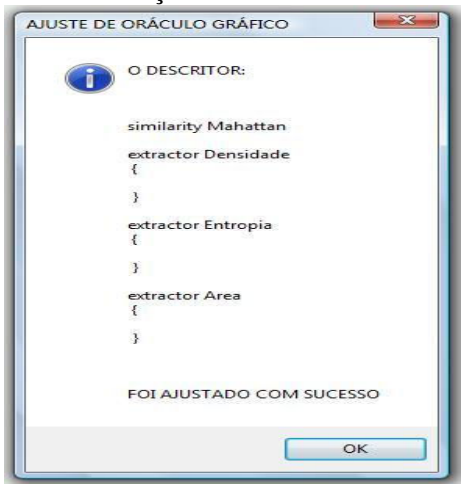
A seleção dos objetos para confecção de um descritor de *oráculo gráfico* qualquer é feita de modo muito simples. Um clique do mouse sobre o nome da função de similaridade escolhida pelo usuário é suficiente para ela ser ajustada ao oráculo. Não acontece diferente com os extratores de características de imagens, com a ressalva de que a seleção de um extrator de características provoca a abertura de uma caixa de diálogo, visando a fazer com que o usuário ajuste os parâmetros do extrator escolhido. Tal propriedade é descrita detalhadamente na seção seguinte.

Figura 23 - Instância de ajuste de descritor de oráculo



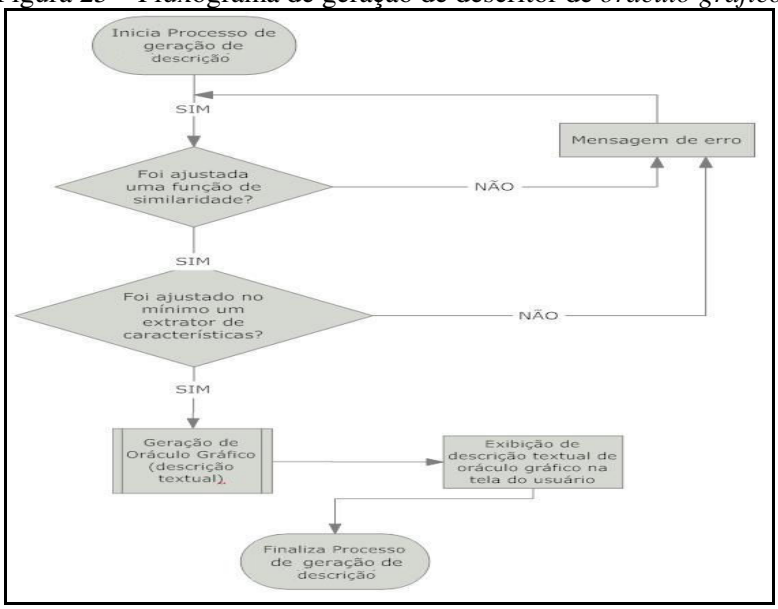
A Figura 23 ilustra a parte da interface principal referente à instância de um *oráculo gráfico* que será gerado. Mesmo sem ter conhecimento completo da instância da interface, é possível afirmar que o descritor de *oráculo gráfico* que está sendo manipulado utiliza função de similaridade, denominada Manhattan, com Extratores de Área, Densidade e Entropia.

Figura 24 - Tela exibição de descritor de *oráculo gráfico*



Para concretizar a geração de uma descrição de um oráculo, o usuário deve clicar com o mouse no botão “Gerar Descritor de Oráculo”, na interface principal. Quando um arquivo parâmetro é gerado com sucesso, sua descrição é exibida em tela, como ilustra a Figura 24.

Figura 25 – Fluxograma de geração de descritor de *oráculo gráfico*



O botão “Gerar Descritor de Oráculo” possui um escutador que inicia um processo que varre os campos da interface referentes aos ajustes do usuário para o oráculo, buscando as informações necessárias para gerá-lo com sucesso. Em linhas gerais, tal processo não passa de uma série de simples concatenações e ajustes de objetos java.lang.String que formarão a descrição do oráculo.

A Figura 25 ilustra um fluxograma que visa a esclarecer melhor a forma de funcionamento da geração de um descritor de *oráculo gráfico*.

2.13 Mecanismo de Ajuste de Parâmetros de Extratores

Já foi dito que grande parcela da flexibilidade obtida pelo protótipo é devida ao fato de que diferentes extratores de características de imagens podem ter parâmetros ajustados, de maneira particular, de acordo com as necessidades do usuário para uma extração de característica de imagem. O trabalho realizado tirou proveito disso pelo fornecimento de mecanismos, utilizados via interface gráfica, que possibilitam ao usuário o ajuste, de maneira simples, de todos os parâmetros ajustáveis dos extratores de características de imagens instalados com sucesso na ferramenta.

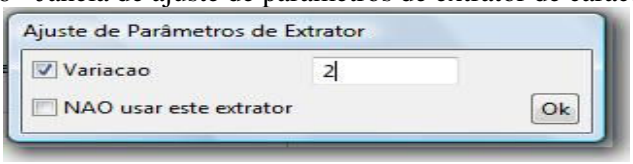
O mecanismo utilizado pelo protótipo para o ajuste de parâmetros de extratores é feito totalmente via interface gráfica, e fornece meios para o usuário ajustar os parâmetros sem desprendimento de esforços. Quando um usuário selecionar um extrator qualquer para ajustar um oráculo, a estrutura tira proveito do conhecimento dos métodos do protocolo firmado pela da interface *IExtractor()*, que lança mão de um seus métodos para obter os parâmetros ajustáveis de tal extrator. O método IExtractor.getPropertiesNames(), como já foi citado, deve retornar um vetor com o nome de todos os parâmetros ajustáveis do extrator. Imediatamente após a seleção do extrator, é aberta com usuário uma caixa de diálogo com o usuário, contendo todas as grandezas que podem ser ajustadas. Esta caixa de diálogo visa a receber os valores que o usuário deseja que sejam ajustados para o acessório no oráculo.

Quando um extrator de características não permitir que nenhum de seus parâmetros seja ajustado externamente, uma mensagem é lançada para o usuário na própria caixa de dialogo aberta após sua seleção. As restrições de valores e dados devem ser tratadas pelo próprio extrator de característica em seus métodos internos de ajuste.

A Figura 26 ilustra uma interface de ajuste de parâmetros da estrutura. Neste caso em particular, pode-se observar que o extrator selecionado pelo usuário tem apenas um parâmetro

ajustável, chamado de Variacao. Logo, o retorno do método IExtractor.getPropertiesNames(), implementado pelo extrator é um vetor contendo apenas um valor; o nome Variacao.

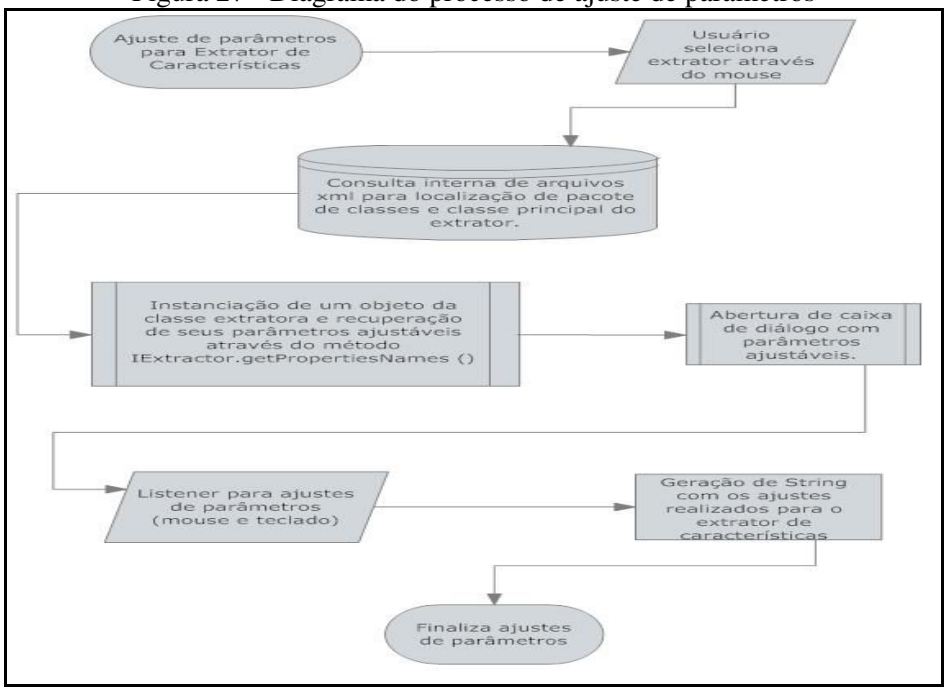
Figura 26 - Janela de ajuste de parâmetros de extrator de características



Por ser fornecido somente um parâmetro ajustável, um check-box (botão de seleção por meio de checagem) contendo o nome do parâmetro e uma caixa de texto são criados pela estrutura. Ao detectar um clique do mouse no botão referido, a interface habilita o campo de texto para que o usuário possa editá-lo da forma que quiser. Para finalizar um ajuste, o usuário deve clicar no botão "OK". Caso o usuário selecione novamente um extrator já indicado para um uso em um *oráculo gráfico*, os dados ajustados anteriormente são recuperados imediatamente para a caixa de diálogo que será criada.

Em toda caixa de diálogo, um botão de checagem é criado com o rótulo "NÃO usar este extrator". Este botão pode ser utilizado toda vez que o usuário quiser desprezar um extrator de características já selecionado em um oráculo. Isto é feito pela checagem do referido botão, seguida de uma confirmação com um clique no botão "OK".

Figura 27 - Diagrama do processo de ajuste de parâmetros



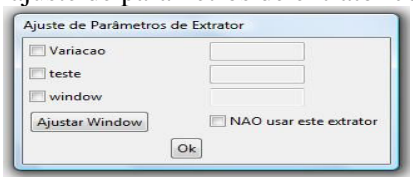
A Figura 27 ilustra, por meio de um diagrama, o processo inteiro de ajuste de parâmetros para um extrator de características em um *oráculo gráfico*. É importante saber que, antes de qualquer coisa, é realizada uma pesquisa interna, visando a resgatar informações básicas a respeito da classe extratora. Este resgate é feito para possibilitar que um objeto da classe extratora seja instanciado.

Quando um usuário finaliza um ajuste de parâmetros para um extrator, por um clique do mouse no botão "OK", uma string que segue a regra sintática apresentada pela sentença <*extractorDecl*>, apresentado linha 5 da Figura 16, é gerada. É exatamente essa cadeia de caracteres que será inserida na descrição final do *oráculo gráfico* e fará referência ao modo de ajustar os parâmetros de tal objeto extrator.

2.13.1 Mecanismos de Ajustes de Janelas para Extratores (window)

É sabido que extratores de características têm o objetivo de capturar uma determinada propriedade visual de uma imagem. Tal característica pode representar a imagem globalmente ou apenas em um foco ou região de interesse. Seguindo a definição apresentada, no segundo caso em especial, apenas uma região da imagem é considerada.

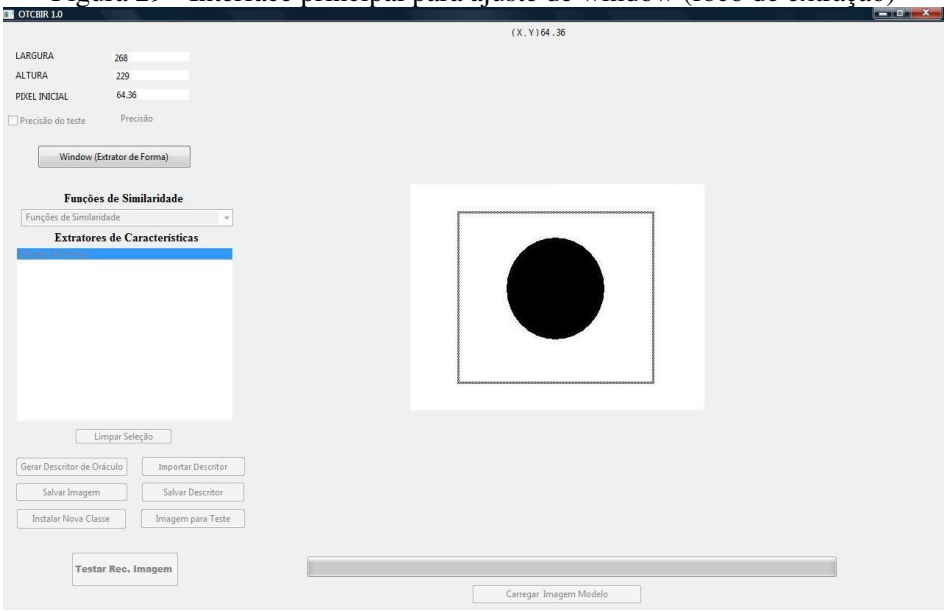
Figura 28 - Janela de ajuste de parâmetros de extrator com window ajustável



Há extratores que têm a propriedade de ajustar sua região de interesse pelo recebimento de um parâmetro. Para este caso particular de extratores, a ferramenta fornece uma forma muito simples e eficiente de ajuste de foco para sua extração. Para que essa interação seja bem sucedida, o parâmetro que faz referência à região de interesse deve ser nomeado de *"window"*. Quando o núcleo da estrutura se depara com o ajuste de parâmetros de um extrator de características com um parâmetro ajustável com nome de "window", um diálogo especial para recebimento dos valores será aberto. A Figura 28 exemplifica tal diálogo. Nela se observa que o extrator de características de imagens permite que três parâmetros sejam ajustados: Variação , teste e window.

Quando reconhece o parâmetro *"window"*, a ferramenta gera, além de um check-button e uma caixa de texto para seu ajuste de forma convencional, um botão denominado de "Ajustar Window". Um clique com o mouse nesse botão transporta o usuário para a tela principal da interface e aciona um escutador do mouse sobre a imagem modelo presente na tela, obviamente a imagem deve estar carregada antes do início ajuste para evitar que uma mensagem de erro seja exibida. O usuário poderá ajustar a janela de extração para o acessório clicando e arrastando sobre a imagem para demarcar sua área de interesse. Todas as outras funcionalidades da interface principal são desligadas durante um ajuste de janela, com exceção de um botão que deve ser acionado por meio do mouse para indicar que o ajuste de janela foi terminado. Tal botão será nomeado de acordo com o nome extrator para o qual está sendo realizado o ajuste de janela, e seguirá a seguinte forma "Window(nome do extrator)". A Figura 29 exibe a interface principal do programa quando está sendo realizado um ajuste de foco de extração.

Figura 29 - Interface principal para ajuste de window (foco de extração)



Nota-se que os valores de Pixel Inicial, Largura e Altura da janela ajustada são carregados em caixas de texto do lado esquerdo da tela em que está ocorrendo o ajuste. Isso visa a facilitar a visualização e quantificação dos valores que estão sendo ajustados.

2.14 Realização de Teste de Recuperação

O teste de recuperação é tratado como mais uma funcionalidade implementada em decorrência da necessidade da realização de teste sobre os arquivos descritores de oráculos gerados na estrutura. A necessidade de apreciar o comportamento de um sistema cujo parâmetro de funcionamento fosse um descritor válido para o ambiente de *oráculo gráfico* – gerado pela própria estrutura – fez com que a funcionalidade fosse adicionada ao trabalho.

O teste de recuperação, citado em seções anteriores, nada mais é do que a obtenção do valor de similaridade entre duas imagens nos moldes de um sistema CBIR. No entanto não há consulta em banco, pois as duas imagens são ajustadas pelo usuário. Até aqui nada em especial, a não ser pelo fato que a função de similaridade a ser usada, bem como os extratores de características e os valores de seus parâmetros, serão acionados a partir de um arquivo texto contendo uma descrição textual. Trata-se de um descritor de *oráculo gráfico*.

2.14.1 Teste de Recuperação por Linha de Comando

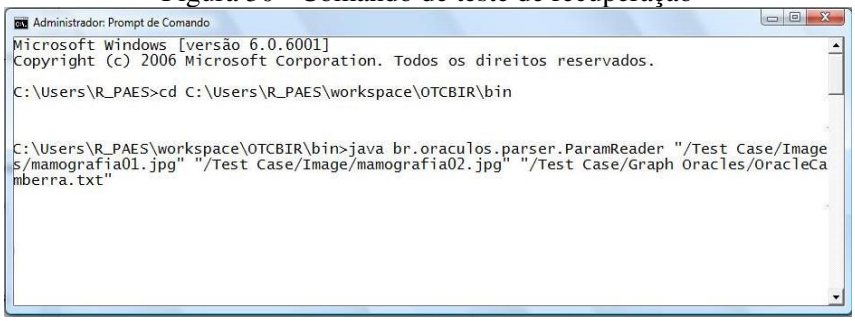
Para que um teste de recuperação de imagem seja realizado pela linha de comando é necessário que três argumentos sejam passados, por meio de um vetor de strings ao método *main* da classe *br.oraculos.parser.ParamReader*. Os argumentos devem representar caminho em disco para a imagem modelo, caminho em disco para a imagem em teste e caminho em disco para o arquivo parâmetro descritor de *oráculo gráfico*, respectivamente.

A classe Java ParamReader é um leitor de parâmetros que tem como função principal verificar a validade de um arquivo passado como parâmetro, neste caso em particular, um arquivo descritor de *oráculo gráfico* que represente uma descrição textual válida. A referida classe é a principal de um conjunto de classes geradas pelo JavaCC durante a implementação de um *parser* para o Parametrizador que funciona como um reconhecedor de parâmetros.

Para que a funcionalidade realizadora do teste de recuperação funcionasse dentro da classe ParamReader, foi necessário lançar mão de uma associação especial de código Java ao arquivo ParamReader.jj, que se trata do arquivo considerado pelo JavaCC. Acrescenta-se que além de reconhecer no arquivo a referida classe, também carrega a função de instanciar os objetos referenciados no arquivo de ajuste

A Figura 30 representa um comando válido dado à ferramenta para realização de um teste de recuperação entre duas imagens chamadas de “mamografia01.jpg” (modelo) e “mamografia02.jpg” (testada) utilizando um arquivo descritor, denominado “OracleCamberra.txt”

Figura 30 - Comando de teste de recuperação



2.14.2 Teste de Recuperação Via Interface

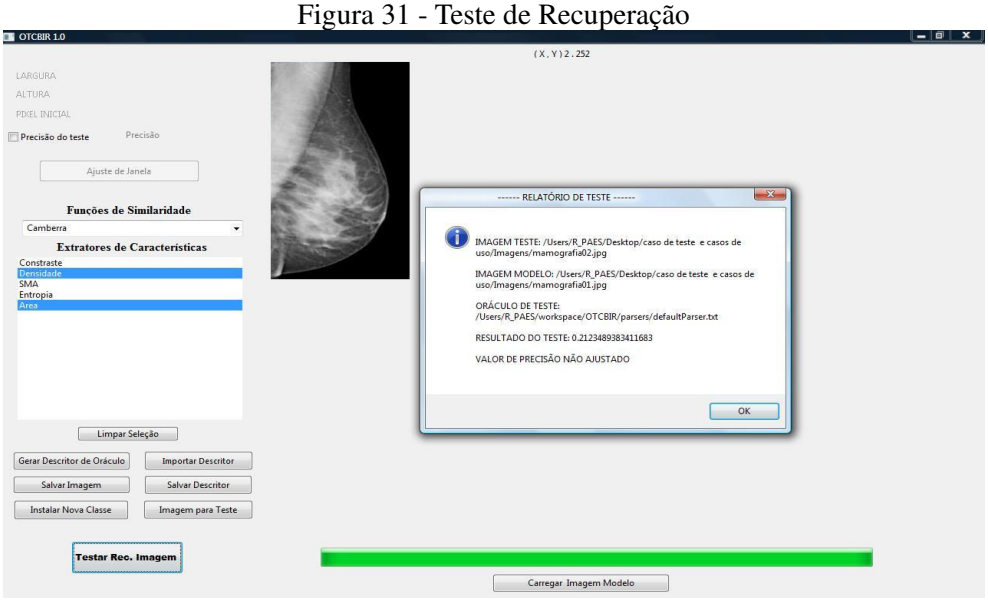
O teste de recuperação pela interface gráfica da ferramenta é mais estruturado, quando comparado ao teste via linha de comando, pois pode ser mais bem visualizado pelo usuário, haja vista que basta observar a instância da interface pra ter idéia dos ajustes que serão seguidos pelo oráculo no cômputo da similaridade entre as imagens.

Convém ressaltar que para a realização do teste é necessário que o usuário tenha carregado a imagem modelo e a imagem a ser testada. Este ajuste é realizado facilmente quando o usuário clica com o mouse nos botões “Carregar Imagem Modelo” e “Imagem para Teste” respectivamente. O critério da comparação, ou melhor, a descrição do *oráculo gráfico* a ser utilizada, deve estar carregada na ferramenta ou deve ser importada para ela. Isso pode ser obtido com o acionamento dos botões “Gerar Descritor de Oráculo” ou “Importar Descritor”, respectivamente. A diferença entre os dois botões é que, na geração de parâmetro, uma instância da interface é criada e, em contraste a isso, na importação de parâmetro, um arquivo descritor externo é trazido para a estrutura do protótipo.

A realização do teste ocorrerá, de fato, quando o usuário clicar no botão nomeado de “Testar Rec. Imagem”, que irá acionar a parametrização e a comparação baseada nos ajustes feitos descritos pelo arquivo parâmetro. A realização bem sucedida de um teste desse caráter gera um relatório com as informações fundamentais acerca do teste realizado. Tais informações consistem em imagem em teste e modelo, descrição utilizada, resultado numérico da similaridade imposta pelo oráculo e, se acionado, critério *precision*. Quando utilizado, tal critério consistirá de um valor limite para ser comparado com o valor gerado no cômputo de similaridade. Em geral, valores de similaridades menores ou iguais ao valor de *precision* fariam com que a imagem em teste fosse considerada similar à modelo. É importante lembrar, que a sentença *precision* é declarada na gramática referente ao arquivo descritor de *oráculo gráfico*, sendo assim pode ser observada na Figura 16, mais precisamente na linha 2. A Figura

31 mostra a interface de um teste de recuperação, ou melhor, da exibição de um do relatório de um teste já realizado.

Figura 31 - Teste de Recuperação



2.15 Considerações finais

Foi introduzida a arquitetura do protótipo proposto em Delamaro (2007a), que é a fonte de contexto do ambiente implementado no trabalho, e a fonte descritiva para aliar CBIR à automatização de oráculos.

Este capítulo visa a inteirar o leitor acerca de uma série de metodologias e particularidades sobre as quais o trabalho foi realizado. Busca-se passar à frente tudo que foi idealizado e posto em prática no protótipo. Detalhamentos, capturas de telas, diagramas e até mesmo trechos importantes de código Java foram utilizados para alcançar seu objetivo maior: uma visualização qualitativa dos testes e exemplo de utilização apresentados no capítulo seguinte.

CAPÍTULO 3 – Resultados, Discussões e Trabalhos Futuros

O objetivo desta seção é apresentar casos de teste, introduzir um estudo de caso que pode ser definido como um exemplo de utilização, pois se trata de um estudo de caso completo, e mostrar algumas comparações e resultados obtidos no final da implementação do protótipo. Como foi apresentado na seção anterior, o trabalho descrito se resume em três funcionalidades principais; instalações de objetos, testes de recuperação e geração de arquivos parâmetro. O propósito principal do trabalho se resume nesta última funcionalidade, além da criação do ambiente de um *oráculo gráfico*. Por isso testes e geração de arquivos parâmetro são o tema dos casos de testes apresentados.

No final da seção são apresentadas algumas possibilidades de incremento a tudo que foi desenvolvido.

3.1 Apresentação dos Casos de Teste

Sabe-se que boa parte dos benefícios obtidos por meio dos sistemas CBIR é direcionada para esquemas CAD e sistemas de auxílio e treinamento médico em geral. Em função disso, foram confeccionados casos de teste baseados em extratores de características e funções de similaridade desenvolvidos para trabalharem com imagens médicas, mais precisamente imagens mamográficas digitalizadas.

As seções seguintes visam a descrever os objetos (funções e extratores) e imagens que foram utilizados para avaliar a eficiência da flexibilidade do sistema e o comportamento de *software*s que utilizem o ambiente de "*oráculo gráfico*", bem como seus arquivos parâmetro.

3.2 Extratores de Características Utilizados

Foram utilizados cinco extratores de características, em um primeiro momento apresentados por Santos (2006), que foram adaptados para o funcionamento flexível proposto pelo projeto descrito em Delamaro (2007a). São eles: Extrator de Área, Contraste, SMA (Segundo Momento Angular), Entropia e Densidade. O trabalho *"Implementação e Indexação de Novas Características em um Sistema de Recuperação de Imagens Mamográficas Baseada em Conteúdo"*, apresentado por Neves (2008), foi responsável pelas adaptações necessárias.

A Tabela 3 exibe um resumo das descrições apresentadas por Neves (2008) e Santos (2006) acerca de tais extratores.

Tabela 3 – Identificações de extratores de características de imagens

Extrator	Fórmula	Descrição
Área	–	Soma dos *pixels* da área pertencente à mama dividida pela quantidade total de *pixels* da imagem.
Densidade	–	Média dos valores dos *pixels* da imagem dividida pelo maior valor de cinza possível (no caso 65536).
Entropia	$-\sum_i \sum_j p(i, j) \cdot \log(\ p(i, j))$	Representa o grau de desordem dos *pixels* da imagem.
Segundo Momento Angular	$\sum_i \sum_j p(i, j)^2$	Extrai o nível de homogeneidade da imagem.
Contraste	$\sum_{n=0}^{N_g-1} n^2 \left\{ \sum_i \sum_j p(i, j),\ se\ \lvert i - j \rvert = n \right\}$	Identifica as variações locais das imagens.

A partir da tabela é possível observar que extratores, como densidade e área, não dispõem de uma fórmula, que sirva como função para obtenção de seu valor de extração. Isso acontece porque elas não necessitam de uma maneira especial de medir a textura da imagem (SANTOS apud MARQUES et al. ,FERRERO et al., 2006). No caso de extratores de Entropia, SMA e Contraste, utiliza-se uma matriz denominada de co-ocorrência para obter a textura de uma imagem. Segundo Santos (2006), a matriz de co-ocorrência corresponde à freqüência de um nível de cinza na imagem, considerando uma distância em uma direção. Logo, a posição p (i, j) da matriz utilizada nas fórmulas indica a freqüência de ocorrência de um particular par de nível de cinza i e j, obtido a partir de uma distância e de um ângulo (direção) (SANTOS, 2006).

Em linhas gerais, os extratores cujas fórmulas são apresentadas na Tabela 3, utilizam uma matriz de co-ocorrência para o cálculo de suas extrações. Isso significa que valores de extração dependem diretamente da freqüência do nível de cinza nas imagens. Como foi exposto no capítulo de metodologia, depois da adaptação do extrator, seu valor de extração deve ser obtido por meio do retorno do método computeValue(PlanarImage) da interface

IExtractor. A Figura 32 mostra o referido método para o extrator de área adaptado por Neves (2008).

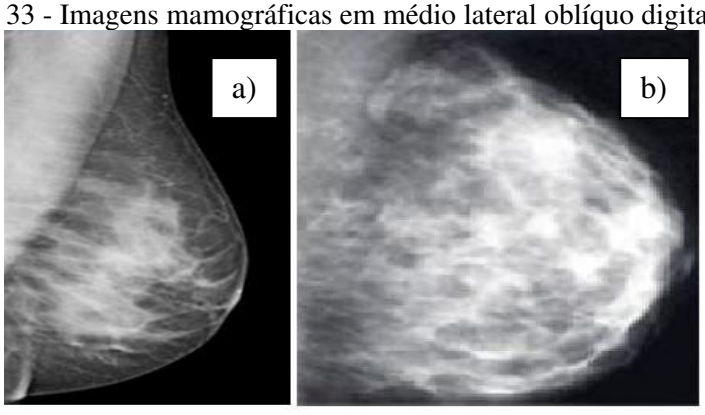
Figura 32 – Método computeValue() do extrator de área

```
1.  public double computeValue(PlanarImage imageLoaded)
2.  {
3.
4.  this.imTesting = imageLoaded;
5.  int[] im1 = getData();
6.  int area = 0;
7.  int total = 0;
8.  for( int i = 0; i < im1.length; i++) {
9.  if( im1[i] != 0 )
10. area++;
11. total++;
12.
13. }
14. return (double) area / total;
15. }
```

3.2.1 Comportamentos dos Extratores

Com a função de elucidar o comportamento dos extratores selecionados para o caso de teste, extraiu-se das imagens apresentadas pela Figura 33 ((a) e (b)) cada uma das características exibidas na Tabela 3. As referidas imagens consistem em mamas digitalizadas e apresentadas na posição em médio lateral oblíquo. Salienta-se que para todos os extratores há um critério diferente de normalização imposto, fazendo com que sempre o valor retornado esteja no intervalo entre zero e um ([0,1]). Isso visa a evitar problemas de similaridade entre imagens de tamanhos distintos entre outros. Um exemplo disso é ilustrado na Figura 32, na qual a área encontrada é dividida pelo tamanho total da imagem (SANTOS, 2006, p.64).

Figura 33 - Imagens mamográficas em médio lateral oblíquo digitalizadas



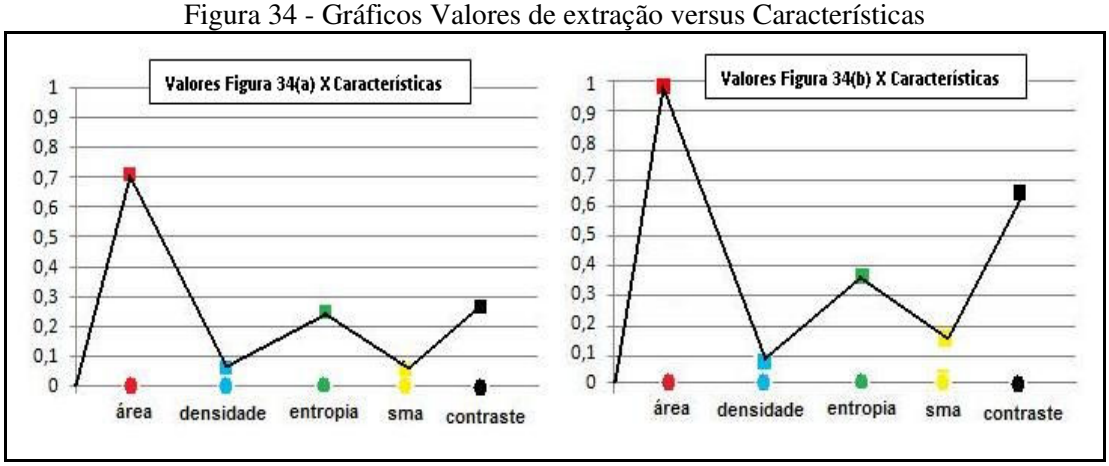
A Figura 34 “a” apresenta um gráfico que relaciona os valores de extração da imagem apresentada na Figura 33 “a” com as respectivas características consideradas. A referida figura consiste na representação gráfica das seguintes associações; Extrator x Valor:

- Área = 0.712442374125832
- Densidade = 0.001861572265625
- Entropia = 0.2403252270404100
- SMA = 0.021647132607827028
- Contraste = 0.2581971747605000

A Figura 34 “b” ilustra um gráfico, nos mesmos moldes da Figura 34 “a”, que relaciona os valores de extração de características da imagem exibida na Figura 33 “b”. Os valores obtidos foram:

- Área = 0.9999917642601835
- Densidade = 0.002044677734375
- Entropia = 0.376586401211376
- SMA = 0.01762180391458643
- Contraste = 0.6938475388503561

Figura 34 - Gráficos Valores de extração versus Características



3.2.2 Funções de Similaridade Utilizadas

A exemplo dos extratores de características, as funções de similaridade utilizadas nos casos de teste e estudo de caso, que serão apresentados nesta seção, foram adaptadas para o funcionamento flexível proposto em Delamaro (2007a). O trabalho *“Implementação e*

Avaliação de Funções de Similaridade em Sistema de Recuperação de Imagem Baseada em Conteúdo", apresentado em Bisconsin (2008), foi responsável por tais adaptações.

O referido trabalho, que fornece funções de similaridade implementadoras da interface ISimilarity, será inserido como estudo de caso do ambiente aqui apresentado, uma vez que é proposto um sistema CBIR para avaliar funções de similaridade, sendo que para isso é necessário utilizar um arquivo parâmetro com regras sintáticas de um descritor de *oráculo gráfico*. Logo, arquivos gerados pelo protótipo descrito neste trabalho foram utilizados.

Tabela 4 - Identificações de funções de similaridade

Função de Similaridade	Fórmula	Descrição
Camberra	$D(x,y)=\sum_{i-1}^{m}\frac{\lvert x_i-y_i\rvert}{\lvert x_i-y_i\rvert}$	Examina a soma de uma série de diferenças entre as coordenadas da fração de um par de objetos. Cada termo da fração diferença tem valor entre 0 e 1.
ChebyChev	$D(x,y)=\max_{i-1}^{m}\lvert x_i-y_i\rvert$	Também chamada de valor de distância máxima. Estuda e analisa a magnitude absoluta das diferenças entre as coordenadas de um par de objetos.
Manhatan	$D(x,y)=\sum_{i-1}^{m}\lvert x_i-y_i\rvert$	Função de distância Manhattan é uma função alternativa, pois exige menos tempo computacional.
Euclidean	$D(x,y)=\sqrt{\sum_{i-1}^{m}(x_i-y_i)^2}$	Identifica parâmetros que estão próximos entre si com a comparação dos valores de um vetor.

Define-se uma função de distância como um algoritmo que compara dois vetores de características e retorna um valor não negativo. Quanto menor o valor retornado, maior é a semelhança entre a imagem modelo e a imagem procurada (BISCONSIN apud ARAÚJO, 2008). Para a confecção do caso de teste, foram escolhidas quatro funções de similaridade, a saber: Camberra, ChebyChev, Manhatan e Euclidiana. A Tabela 4 resume cada uma dessas funções e apresenta o modo pelo qual fazem as comparações entre os elementos de cada vetor de características.

A Figura 35 exibe o método computeSimilarity da função de similaridade Euclidiana, já adaptada para o funcionamento em um ambiente de *oráculo gráfico*. A referida foi desenvolvida por Bisconsin (2008).

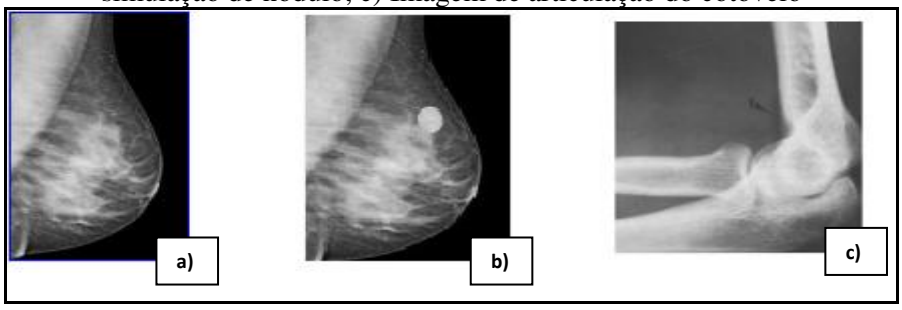
Figura 35 - Método computeSimilarity da função Euclidiana

```
1.  public double computeSimilarity(Vector vModel, Vector vTest)
2.  {
3.  Double d1,d2,distancia=0.0;
4.  for (int i=0 ; i < vModel.size() ; i ++)
5.  {
6.  d1 = (Double) vModel.elementAt(i);
7.  d2 = (Double) vTest.elementAt(i);
8.  distancia += (d1 - d2)*(d1 - d2);
9.  }
10. return Math.sqrt(distancia);
11. }
```

3. 3 Conjunto de Imagens Utilizadas

Um conjunto de 15 imagens médicas foi utilizado para a ilustração do estudo de caso. As referidas consistem em imagens mamográficas e algumas imagens de articulações digitalizadas com resolução 180 x 300 pixels. Salienta-se que foram inseridas algumas imagens repetidas com variação de brilho e alternância de clareza em seu ambiente, ou seja, as mesmas imagens ora mais claras ora mais escuras. A Figura 36 ilustra algumas destas imagens. Tais imagens foram inseridas no banco de dados da ferramenta apresentada por Bisconsin (2008).

Figura 36 - Imagens para teste: a) Imagem mamográfica original; b) Imagem mamográfica com simulação de nódulo; c) Imagem de articulação do cotovelo

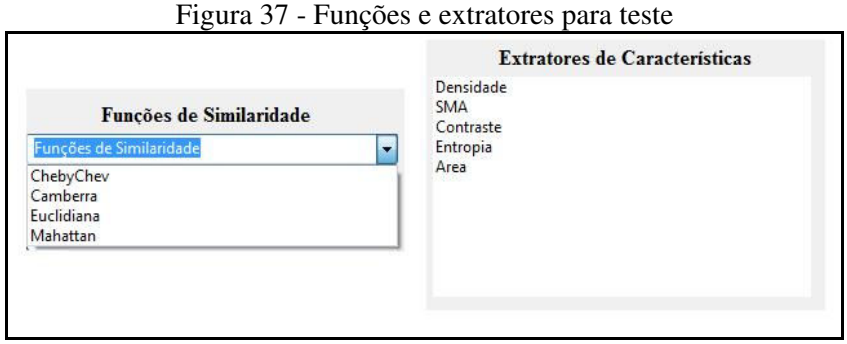


3.4 Instalações de Objetos

Obviamente que para realizar os mais diversos testes e gerar material para estudo de caso, foi necessário povoar a ferramenta descrita no trabalho com os acessórios pertinentes. Logo os extratores de área, densidade, entropia, SMA e contraste foram instalados no ambiente administrado pelo sistema. O mesmo aconteceu com objetos referentes às funções de similaridade, ou seja, distâncias de Camberra, Chebychev, Manhatan e Euclidiana. A

Figura 37 ilustra setores referentes aos objetos disponíveis do ambiente de oráculo após a instalação de todos os acessórios descritos anteriormente necessários.

Figura 37 - Funções e extratores para teste



3.5 Paralelo entre Caso de Teste e Estudo de Caso

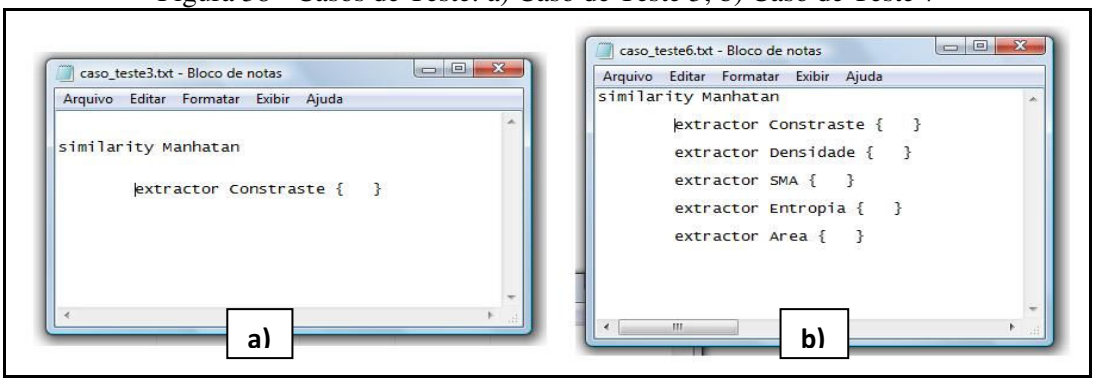
Para ilustrar um dos contextos no qual os oráculos gráficos e descritores para oráculos gráficos podem ser inseridos, foi feito um paralelo entre caso de teste e estudo de caso. Tal paralelo foi direcionado de modo a exibir a geração arquivos parâmetro, por meio da ferramenta, e em seguida demonstrar como seria sua utilização em um sistema que os consumisse, constituindo um exemplo de utilização de tais arquivos.

O sistema cliente dos descritores de oráculos gráficos, como já foi dito, é um sistema, proposto por Binsconsin (2008), que busca avaliar o funcionamento de funções de similaridade em sistemas CBIR, utilizando como parâmetro um arquivo descritor de *oráculo gráfico*.

3.6 Geração de Arquivos Parâmetros para Casos de Teste

A partir dos objetos instalados e descritos nas seções anteriores, um conjunto de oito arquivos parâmetros foi gerado na ferramenta. Buscou-se criar dois casos de teste para cada função de similaridade. De uma forma geral, cada distância de similaridade terá um caso de teste com apenas um extrator e outro caso com todos os extratores. Aleatoriamente foram utilizados extratores de características com ajustes de parâmetros. A Figura 38 exibe os dois casos de teste criados para a função de similaridade Manhatan que foram criados por meio da interface gráfica do protótipo.

Figura 38 - Casos de Teste: a) Caso de Teste 3; b) Caso de Teste 7



A forma com a qual os casos de teste foram nomeados e descritos é exposta na Tabela 5.

Observa-se que o extrator de contraste foi o escolhido para compor os casos de teste nos quais apenas um extrator foi utilizado. Isso implica dizer que tal extrator esteve presente em todos os descritores de oráculos gerados para o estudo de caso.

Tabela 5 - Descrição de casos de Teste

NOME	F. SIMILARIDADE	EXTRATORES
Caso_Teste1.txt	• Camberra	• Contraste
Caso_Teste2.txt	• ChebyChev	• Contraste
Caso_Teste3.txt	• Manhatan	• Contraste
Caso_Teste4.txt	• Euclidiana	• Contraste
Caso_Teste5.txt	• Camberra	• Contraste, Densidade, Entropia, SMA e Area
Caso_Teste6.txt	• ChebyChev	• Contraste, Densidade, Entropia, SMA e Area
Caso_Teste7.txt	• Manhatan	• Contraste, Densidade, Entropia, SMA e Area
Caso_Teste8.txt	• Euclidiana	• Contraste, Densidade, Entropia, SMA e Area

3.7 Arquivos Parâmetros de *Oráculo gráfico*: Exemplo de Utilização

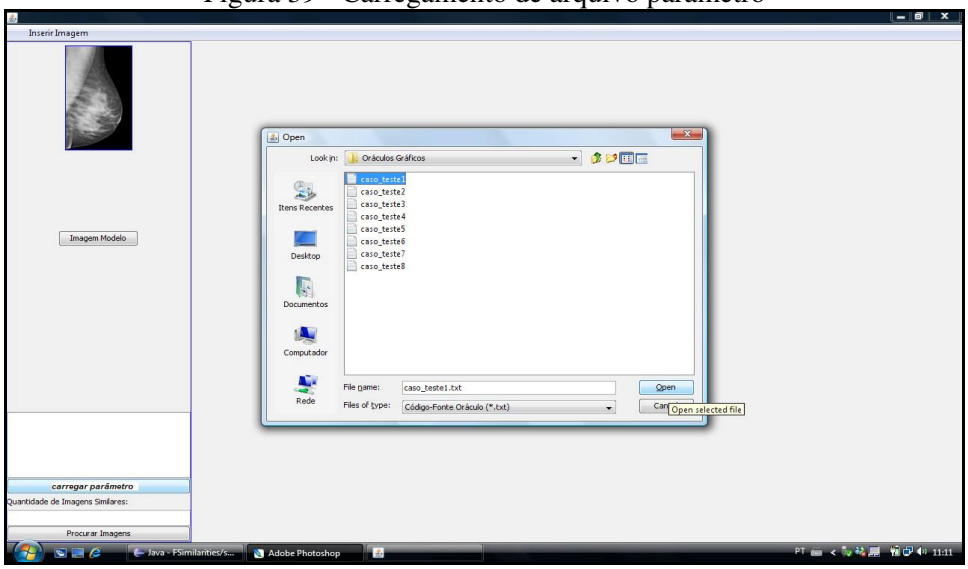
Em linhas gerais, o propósito da utilização de um arquivo parâmetro descritor seria o seu aproveitamento em uma ferramenta de teste de *software*, mostrando como o ambiente de

oráculo gráfico deve se comportar, servindo a este como referência de um processamento qualquer com saída gráfica. No entanto o que pode ser observado é que estes arquivos podem ser inseridos em um leque maior de utilidades. Esse é o contexto do estudo de caso descrito a seguir.

O trabalho apresentado por Bisconsin (2008) é um sistema CBIR, ou seja, trata-se de um *software* que recupera imagens de um banco de dados levando em consideração a similaridade de tais imagens com uma imagem de consulta. Para a realização desta funcionalidade o trabalho proposto pelo autor lança mão de um arquivo parâmetro para processamento. O referido deve descrever as características bem como a função de similaridade que deve ser utilizada em um processamento CBIR. Como já foi citado, os arquivos reconhecidos pela ferramenta devem seguir as regras sintáticas de um descritor de *oráculo gráfico*.

Os descritores de oráculos gráficos gerados no caso de teste e descritos na Tabela 5 foram utilizados como parâmetro da ferramenta de Bisconsis (2008). Salienta-se que essa ferramenta tem, em sua estrutura interna, todos os objetos que foram instalados no protótipo para confecção dos arquivos parâmetro. A Figura 39 mostra como é feito o carregamento de um parâmetro na ferramenta proposta por Bisconsin (2008).

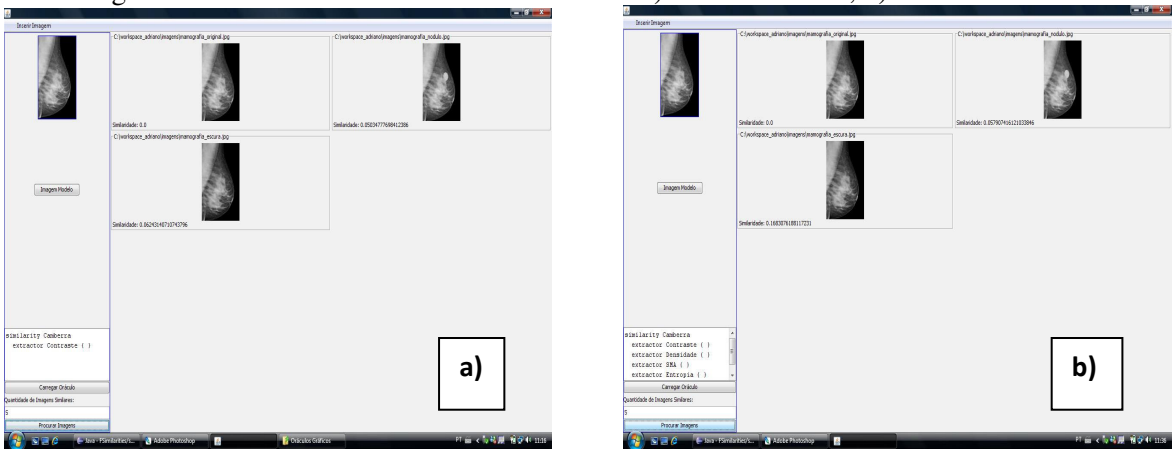
Figura 39 - Carregamento de arquivo parâmetro



Após o carregamento de um arquivo parâmetro válido, a ferramenta permite que o usuário carregue uma imagem modelo e realize a recuperação de imagens nos moldes de um sistema CBIR, seguindo os critérios impostos pelo parâmetro. Também é permitido que o usuário escolha o número máximo de imagens que serão recuperadas.

A Figura 40 exibe como foram realizadas algumas consultas. É possível observar que a ferramenta exibe, em forma de relatório, as imagens recuperadas, bem como o valor de similaridade obtido entre cada uma delas e a imagem modelo. A Figura 40 “a” exibe o estudo de caso utilizando os parâmetros definidos pelo caso de teste denominado de Caso_Teste1. A Figura 40 “b” exibe o funcionamento da ferramenta de Bisconsin (2008) quando é utilizado um arquivo parâmetro tal qual o caso de teste denominado Caso_Teste5.

Figura 40 – Consultas utilizando caso de teste: a) Caso de teste 1; b) Caso de Teste 5



Salienta-se que a imagem modelo utilizada é ilustrada pela Figura 36 ”a”. É possível perceber que, nos dois casos demonstrados, as imagens recuperadas, em ambas as consultas, foram as mesmas, no entanto, os valores de similaridade foram diferentes. Para algumas imagens a similaridade foi acrescida e, em outros casos, ela foi reduzida.

A Tabela 6 mostra um relatório final com o comportamento das consultas para todos os casos de teste gerados no protótipo e aplicados como estudo de caso. Foram relacionados os casos de teste que envolviam funções de similaridade comuns, a exemplo dos exibidos pela Figura 38 “a” e “b” que envolvem Manhatan. A tabela relaciona os exemplos de utilização mostrando a diferença máxima apresentada entre as similaridades da mesma imagem para os diferentes casos de teste. Isso implica dizer, portanto, dizer que, nos exemplos exibidos pela Figura 40, as consultas com o Caso_Teste1 e Caso_Teste5 recuperaram as mesmas imagens, sendo que a maior diferença de similaridade para uma mesma imagem recuperada foi de 0,51788. Logo, houve alterações impostas pelos arquivos parâmetros, que fizeram com que as similaridades entre as imagens fossem alteradas.

Tabela 6 - Relatório de teste

Função de Similaridade	Casos de teste envolvidos	Imagem Modelo	Máx. diferença entre similaridades	Houve Diferença de Recuperação
CAMBERRA	**Caso_Teste1.txt e Caso_Teste5.txt**	**Figura 36a**	**0,51788**	NÃO
CHEBYCHEV	**Caso_Teste2.txt e Caso_Teste6.txt**	**Figura 36a**	**0,45121**	NÃO
MANHATAN	**Caso_Teste3.txt e Caso_Teste7.txt**	**Figura 36a**	**0,37465**	NÃO
EUCLIDIANA	**Caso_Teste4.txt e Caso_Teste8.txt**	**Figura 36a**	**0,48454**	NÃO

3.8 Considerações Finais

Logicamente os resultados apresentados, no que se refere a valores, dependem da fidelidade de extratores de características de imagens e de funções de similaridade implementadas. Cabe ressaltar que um estudo de caso completo foi apresentado por meio da estrutura desenvolvida em Binsconsin (2008), o que caracteriza um exemplo de utilização dos arquivos descritores de oráculos gráficos gerados no protótipo desenvolvido.

CONCLUSÕES

A utilização de oráculos é parte essencial da atividade de teste. Muitos trabalhos relacionados com a área de teste acabam abordando, direta ou indiretamente, dificuldades relacionadas a decidir sobre a correção de uma execução do programa em teste. Muitos destes trabalhos abordam problemas de como e o que comparar ao se testar ou decidir sobre a correção de uma execução (DELAMARO, 2007a).

No caso de programas com processamento gráfico, o problema de determinar a execução torna-se mais complexo, pois nem sempre está claro quais são as características que devem ser consideradas para que seja avaliada a correção. Além disso, o formato gráfico dificulta a automatização (DELAMARO, 2007a).

O objetivo deste trabalho é a construção de um ambiente, batizado de "*ORÁCULO GRÁFICO*" no qual o usuário pode criar seus próprios oráculos para programas com saída gráfica de forma flexível. Tais oráculos são definidos a partir da técnica de processamento de imagem conhecida por Recuperação de Imagem Baseada em Conteúdo. Sabe-se que, dependendo do objetivo que se tenha, a extração de características pode variar de modos particulares. Nesse contexto, o usuário pode determinar por meio de um oráculo como e quais características de um processamento gráfico devem ser esperadas.

A ferramenta desenvolvida conta com:

- um núcleo de um Comparador que permite a instalação de extratores de características e funções de similaridade. É desenvolvida uma metodologia simples que utiliza interfaces Java para realizar tais instalações.
- um *parser* para um Parametrizador, ou melhor, um reconhecedor de descrições de oráculos para programa com saída gráfica.
- uma interface gráfica que cria arquivos de parâmetros

A estrutura desenvolvida mostra-se satisfatória para o contexto na qual foi inserida, ou seja, a geração de arquivos de parâmetro, que sirvam como uma descrição para o comportamento de um oráculo de um programa com saída gráfica, bem como um ambiente de *oráculo gráfico*. A forma com a qual os parâmetros de extratores são ajustados corresponde às necessidades do usuário, e a geração de arquivos parâmetro está disposta de modo simples. Enfim um protótipo de um ambiente de oráculo para programas com saída gráfica foi desenvolvido.

Um possível incremento da estrutura apresentada pode ser o ajuste da ferramenta de modo que sejam realizados testes em grande escala, com uma grande quantidade de imagens captadas de um programa que tenha saída gráfica. Logo, um mecanismo deve ser adicionado à ferramenta na forma de um pluggin que acompanhe um processamento gráfico e armazene suas saídas para que sejam submetidas aos devidos testes.

Experiência com a pesquisa, ampliação do conhecimento e exercício de forma positiva das capacidades de inovação e criação são algumas das contribuições proporcionadas pelo trabalho apresentado. Além disso, há um protótipo de uma ferramenta de automatização de oráculo que emprega técnicas de recuperação de imagem baseada em conteúdo, o que já constitui sólida contribuição (OLIVEIRA et al., 2008).

Trabalhos Futuros

Pretende-se ainda medir a eficiência dessa tecnologia e ter uma visão mais aprimorada sobre os problemas relacionados com a automatização de oráculos que utilizem informações na forma de imagens, em lugar de informações convencionais como texto ou sinais (OLIVEIRA et al., 2008).

Uma continuação do trabalho pode ser iniciada com atividades teóricas, ou seja, pesquisas bibliográficas visando a constituir uma base para a definição de extratores e funções de similaridade, bem como arquivos parâmetro, haja vista que neste trabalho nenhum extrator foi desenvolvido. Desta forma poderá ser avaliado o desempenho dos extratores e do CBIR como um todo dentro do contexto de teste de *software* para processamentos gráficos.

Novas atividades de automatização podem ser inseridas no contexto da ferramenta, bem como atividades experimentais tal qual a avaliação da efetividade do ambiente na automatização de teste.

Enfim, por se tratar de um trabalho que engloba duas áreas de pesquisa, ou seja, é multidisciplinar, ele tem muitas faces que permitem ser exploradas, tanto no que tange a teste de *software* quanto a processamento de imagens.

REFERÊNCIAS

ARAUJO, M. R. B.; TRAINA JR., C; TRAINA A.; MARQUES, P. M. A. *Recuperação de exames em sistemas de informação hospitalar com suporte a busca de imagens baseada em conteúdo*. In: VII Congresso Brasileiro de Informática em Saúde – CBIS'S 2002, Natal – RN, outubro de 2002.

AZEVEDO, M. C. *Uma Estratégia para Geração de Oráculos de Teste de Software a Partir de Especificação Formal*, In: V Congresso Brasileiro de Informática., v. 1.Porto Alegre-RS, 2004.

BINCONSIN, A. M. *"Implementação e avaliação de funções de similaridade em sistema de recuperação de imagem baseada em conteúdo"*. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

BUGATTI, P. H., *Análise de influência de funções de distância para o processamento de consulta por similaridade em recuperação de imagem baseada em conteúdo.* 2008. 91 f. Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP – para obtenção do título de Mestre em ciência da computação e matemática computacional, São Carlos, 2008.

CORBOY, A, et al. *Texture-based image retrieval for computerized tomography databases.* In: The 18th IEEE International Symposium on Computer-Based Medical Systems (CBMS'05), June, 2005.

DEITEL, H. M.; DEITEL, P. J. *Java Como Programar.* trad. Édson Furmankiewicz. ed.6. São Paulo, Pearson Prentice Hall, 2005, 1110p.

DELAMARO, M. E., *Como construir um compilador utilizando ferramentas Java.* São Paulo: Novatec, 2004, 308p.

DELAMARO, M. E., *Definição de oráculos de teste para programas com saída gráfica usando recuperação baseada em conteúdo.* Projeto de Pesquisa apresentado ao Conselho Nacional do Desenvolvimento Científico e Tecnológico – CNPQ, 2007a.

DELAMARO, M. E.; MALDONADO, J. C.; JINO Mário. *Introdução ao teste de software*. Rio de Janeiro: Elsevier, 2007b. 394p.

DELAMARO, M. E.; MALDONADO, J. C.; MARTHUR, A. P. *Interface Mutation: An approach for integration testing.* IEEE Transactions on *Software* Engineering, v. 27, n. 3, p. 228-247, 2001.

DELAMARO, M. E.; OLIVEIRA, R. A. P. "*Apoio à automatização de oráculos de teste para programas com saídas gráficas"*. Projeto de pesquisa para bolsa mestrado apresentado à Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP, 2008.

DEUTSCH, M. *Verification and Validation.* In *Software* Engineering, (R. Jensen e C. Tonies, eds). Prentice-Hall: 1979, p. 329-408.

DIAS NETO, A. C. "Engenharia de *Software* - Introdução a Teste de *Software*". *Revista Engenharia de Software edição especial.* Rio de Janeiro, v.01, n.01, p.54-59, 2007.

FERREIRA, M. J. R., *Um modelo de recuperação de imagens por conteúdo através da quantização do espectro de Fourier* .2005, 112 f. Dissertação apresentada ao Programa de Pós-Graduação da Faculdade de Ciência da Computação da Universidade Federal de Uberlândia como requisito para obtenção do grau de Mestre em Ciência da Computação, 2005.

FILARDI, A. L., *Análise e avaliação de técnicas de interação humano-computador para sistemas de recuperação de imagem por conteúdo baseadas em um estudo de caso.* 2007, 149 f. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional). Instituto de Ciências e Matemática Computacional, ICMC-USP, São Carlos, 2007.

GATO, H. E. R.; NUNES, F. L. S.; SCHIABEL, H. *Uma proposta de Recuperação de Imagens Mamográficas Baseada em Conteúdo*. In: IX Congresso Brasileiro de Informática em Saúde, 2004, Ribeirão Preto. Anais do Congresso Brasileiro de Informática em Saúde, 2004. v. 1.

GRANDE *Enciclopédia Larousse Cultural.* ed.18, Nova Cultural, Larousse. São Paulo, 1998.

GRANDES *desafios da Pesquisa em Computação no Brasil* – 2006 – 2016, Relatório sobre o seminário realizado em 8 e 9 de maio de 2006 – SBC. São Paulo, em: http://www.sbc.org.br./index.php?language=1&content=downloads&id=272.

GUDIVADA, V. N.; WOODS, R. E. *Content-Based Image Retrieval Systems,* IEEE Computer-Special Issue, Vol. 28, p.18-62, Setembro de 1995.

HOFFMAN, D. *Using Oracles in Testing and Test Automation (1-3). Software* Quality Methods, LLC. em: http://www.logigear.com/newsletter/using_oracles_in_testing_and_test_automation_part1.asp , 2006.

HOFFMAN, D. *Using oracles in testing automation.* In: Pacific Northwest *Software* Quality Conference (PNSQC 2001), 2001.

IEEE Standard 610-1990: IEEE *Standard Glossary of Software Engineering Terminology,* IEEE Press, 1990.

JAI, (2008) disponível em http://java.sun.com/products/java-media/jai/downloads/download-1_1_2.html, acessada em 05 de maio de 2008.

JAVACC, (2008) disponível em https://javacc.dev.java.net/, acessada em 20 de maio de 2008.

JDOM, (2008) disponível em http://www.jdom.org/dist/binary/jdom-1.1.zip, acessada em 23 de março de 2008.

KINOSHITA, S. K. et al.; *Recuperação baseada em conteúdo de imagens mamográficas: atributos visuais de forma, espectrais no domínio de Radon e granulometria.* In: IX Congresso Brasileiro de Informática em Saúde, 2004, Ribeirão Preto. CBIS'S2004, 2004.

MALDONADO, J. C. et al. *Introdução ao teste de software*. Notas didáticas do ICMC-USP, versão 2004-01. São Carlos, 2004. 56p.

MALDONADO, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de Software.* PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MARQUES, J.; TRAINA, A. J. M. *Realimentação de Relevância: Integração do Conhecimento do especialsta com a Recuperação de Imagens por Conteúdo*. In: VI Workshop de Informática Médica - WIM'2006 (junto ao V Simpósio Brasileiro de Qualidade de Software - SBQS), 2006, Vitória. Anais do VI Workshop de Informática Médica WIM'2006. Vitória. 2006. v.1. p. 83-93.

McLAUGHLIN, B. *Java & XML: solutions to real world problems.* O'Reilly, Ed.2, 2001, 528p.

MOREIRA FILHO, T. *Execução do processo de teste tratado como projeto.*In: I Seminário Catariense de Qualidade e Teste de *Software*, 1, Florianópolis, 2008.

MOSHFEGHI, M.; SAIZ, C.; Yu, H. *Content –based retrieval of medical images with relative entropy.* In: Proceedings of SPIE Medical Imaging 2004, Picture Archiving and Communication Systems (PACS) and Imaging Informatics, vol. 5371, p.31-42, 2004.

MOURA, D. M.; et al. *Recuperação de imagens baseada em conteúdo.* In: IV Workshop em Tratamento de Imagens, 2003, Belo Horizonte. Anais do IV Workshop em Tratamento de Imagens, NPDI/DCC/ICEx/UFMG, p. 101-109.

MYERS, G. J. *The art of software testing.* 2. ed. Hoboken, New Jersey: John Wiley & Sons, Inc, 2004. 234p.

NEVES, A. C. *"Implementação e indexação de novas características em um sistema de recuperação de imagens mamográficas baseada em conteúdo"*. 2008, Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

OLIVEIRA, R. A. P.; DELAMARO, M. E.; NUNES. F. L. S. *Estrutura para utilização de recuperação de imagens baseada em conteúdo em oráculos de teste com saída gráfica.* In: IV Workshop de Visão Computacional, WVC'2008. UNESP-Bauru, Bauru, outubro de 2008.

PRESSMAN, R. S. *Engenharia de Software.* tradução José Carlos Barbosa dos Santos; revisão técnica José Carlos Maldonado, Paulo Cesar Masiero, Rosely Sanches. São Paulo: Makron Books, 1995. 1056p.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. et al., *Qualidade de software – Teoria e prática.* Prentice Hall, São Paulo, 2001.

SANTOS, A. P. O. *Recuperação de imagens mamográficas baseada em conteúdo.* 2006. 154 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

SUN M., *JAVA document home page.* Disponível em: http://java.sun.com/j2se/1.5.0/docs/api/. Acesso em abril de 2008.

TAKAHASHI, J., *An automated Oracle for verifing GUI objects. Software* Engineering Notes, vol 26(4), 99 83-88, 2001.

TORRES, R. S.; FALCÃO, A. X.. *Content-Based Image Retrieval: Theory and Applications*. Revista de Informática Teórica e Aplicada, 13(2):161–185, 2006.

TRAINA, A. J. M., *Técnicas de* CBIR *aumentando a usabilidade de exames por imagens.* Notas didáticas do ICMC-USP. São Carlos, 2006.