**UFSM**

**Dissertação de Mestrado**

# PADRÕES DE PROJETO NO DESENVOLVIMENTO DE SISTEMAS DE PROCESSAMENTO DE IMAGENS

**Daniel Welfer**

**PPGEP**

**Santa Maria, RS, Brasil**

**2005**

# PADRÕES DE PROJETO NO DESENVOLVIMENTO DE SISTEMAS DE PROCESSAMENTO DE IMAGENS

---

por


**Daniel Welfer**


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

**PPGEP**

**Santa Maria, RS, Brasil**

**2005**

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

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

**PADRÕES DE PROJETO NO DESENVOLVIMENTO DE SISTEMAS DE PROCESSAMENTO DE IMAGENS**

elaborada por
**Daniel Welfer**

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

COMISSÃO EXAMINADORA:

---

Prof. Ph.D. Marcos Cordeiro d'Ornellas
(Presidente/Orientador - UFSM-Departamento de Eletrônica e Computação)

---

Prof. Dr. Felipe Martins Müller
(UFSM-Departamento de Eletrônica e Computação)

---

Prof. Dr. José Antônio Trindade Borges da Costa
(UFSM-Departamento de Física)

Santa Maria, 01 de fevereiro de 2005

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Welfer, Daniel

Padrões de Projeto no Desenvolvimento de Sistemas de Processamento de Imagens / Daniel Welfer. – Santa Maria: Programa de Pós-Graduação em Engenharia da Produção, 2005.

85 f.: il.

Dissertação (mestrado) – Universidade Federal de Santa Maria. Programa de Pós-Graduação em Engenharia da Produção, Santa Maria, BR–RS, 2005. Orientador: Marcos Cordeiro d'Ornellas.

1. Engenharia de software . 2. Padrões de projeto. 3. Programação orientada por objetos. 4. Linguagem de Modelagem Unificada. I. d'Ornellas, Marcos Cordeiro. II. Título.

UNIVERSIDADE FEDERAL DE SANTA MARIA
Reitor: Prof. Dr. Paulo Jorge Sarkis
Vice-Reitor: Prof. Dr. Clovis Silva Lima
Pró-Reitor de Pós-Graduação e Pesquisa: Prof. Dr. Ney Luis Pippi
Diretor do Centro de Tecnologia: Prof. Dr. Felipe Martins Müller
Coordenador do PPGEP: Prof. Dr. João Hélvio de Oliveira Righi
Coordenador da Área de Tecnologia da Informação: Prof. Dr. Marcos C. d' Ornellas

✍ Essa dissertação foi escrita utilizando a ferramenta de edição $\LaTeX\ 2_\varepsilon$

*"Tell me and I forget.*
*Teach me and I remember.*
*Involve me and I learn."*
**Benjamin Franklin**

Dedico esta dissertação aos meus pais
Paulo Welfer e Renata Welfer
e aos meus irmãos Marcos, Tiago e Márcia

# AGRADECIMENTOS

Em primeiro lugar a DEUS que, dentre todas as outras coisas, me deu a vida.

Quero agradecer também, de forma muito especial, ao meu orientador Prof. Marcos Cordeiro d'Ornellas que pela sua paciência, incentivo e auxílio propiciou um ambiente de trabalho agradável e produtivo.

À Coordenação de Apoio ao Pessoal de Ensino Superior(CAPES) por ter financiado esse estudo. E, finalmente, a todos os integrantes do grupo PIGS (Grupo de Processamento de Informações Multimídia) que indiretamente, contribuiram para o desenvolvimento desse trabalho. De forma especial aos alunos de graduação em Ciência da Computação Grasiela, Mônica e Gabrielle ("Turma da Mônica"), que pelo exímio conhecimento em programação ajudaram a aumentar a aplicação desse trabalho além de apontar alguns equívocos que inicialmente apresentava.

# RESUMO

Essa dissertação apresenta componentes de software para processamento e análise de imagens digitais e um sistema capaz de controlá-los de forma organizada. Os componentes precisam ser funcionais, flexíveis, legíveis e de fácil manutenção. Assim, alcançar esses requerimentos básicos é uma necessidade no processo de desenvolvimento de software. Componentes de software são projetados para serem usados em uma variedade de ambientes. A arquitetura da linguagem de programação utilizada, no caso Java, permite que os programas sejam montados a partir de blocos de software. Dessa forma o projetista pode incorporar facilmente esse componente em uma aplicação, necessitando conhecer apenas sua interface de entrada e saída. No entanto, encontrar a abstração certa para construir software reutilizável não é uma tarefa fácil. Mesmo os projetistas mais experientes em orientação por objetos freqüentemente precisam revisar um projeto várias vezes antes de conseguir uma solução apropriada. Por essa razão, a idéia de padrões de projeto tem ganhado terreno rapidamente, desde que estabelece uma solução para um certo problema de projeto. Estes padrões especificam uma maneira para construir, estruturar e manipular entidades de software em um estilo racional visando, principalmente, gerenciar a sua complexidade no domínio de processamento digital de imagens para assegurar a sua qualidade. Nesse trabalho foram construídos componentes para segmentação de imagens baseado em filtros de convolução, para melhoria da qualidade da imagem através do processo de equalização automática, para armazenamento da imagem em arquivos gráficos de diferentes formatos, na criação de um componente para visualizar o histograma, na visualização da imagem na tela através de interface gráfica, na conversão da imagens coloridas para tons de cinza e no processo de limiarização.

**Palavras-chave:** Engenharia de software , padrões de projeto, programação orientada por objetos, Linguagem de Modelagem Unificada.

# ABSTRACT

*This dissertation presents software components for processing and analysis of digital images and a system capable to control them in an organized way. The Components must be functional, context-free, readable and maintainable. So far, achieving these basic requirements is a need in the software development process. Software components are designed to be reusable in a variety of different environments. The architecture of programming language used, in the case Java, allows programs to be assembled from software building blocks. In that way the designer can incorporate easily these components into an application, needing just to know your entrance and exit interface. However, to find the right abstractions to build extensible and reusable software is not an easy task. Even experienced object-oriented designers often need to review a design several times before getting to one appropriate solution. Therefore, the idea of design patterns has gained ground quickly since it provides a solution to a certain design problem. These patterns specify a way to build, structure, and manipulate software entities in a reasonably fashion, aiming mainly, to management your complexity in the domain of digital imaging processing to assure your quality. In this paper were built components for image segmentation based on convolution filters, for improvement of the quality of the image through the process of automatic equalization, for image storage in graphic files of different formats, in the creation of a component to visualize the histogram, in the visualization of the image in the screen through graphic interface, in the conversion of the colored images for grayscale formats and in the thresholding process.*

**Keywords:** Software Engineering, Design Patterns, Object Oriented Paradigm, Unified Modeling Language.

# LISTA DE FIGURAS


Figura 2.1: Os três personagens do processo de componentização segundo Padmal Vitharana, [VIT 2003]. a) pode-se observar que os componentes são fabricados para vários fins sendo que, não há uma relação explícita entre eles, isto é, são concebidos para serem independentes. b) O montador precisa encontrar e inter-relacionar os componentes a fim de resolver o problema do cliente. c) O produto de software final pronto para ser utilizado. . . . . . . . . . . 21

Figura 2.2: Esquema do sistema projetado para suportar a componentização. . . . . . . 22

Figura 3.1: Uso do diagrama de classes para o problema da equalização de imagens . . 26

Figura 3.2: A operação de equalização demonstrada via diagrama de atividades. a ) exibe o processo de equalização mais genérico se preocupando apenas com a entrada e saída de informações. b ) "Explode" o processo de equalização original permitindo assim visualizar a seqüência lógica exigida pelo algoritmo. Em ambas ilustrações o círculo preto representa o início do processo e o círculo circunscrito em outro o seu término. . . . . . . . . . . . . . . 28

Figura 4.1: Desenvolvimento de software em camadas. . . . . . . . . . . . . . . . . . 30

Figura 4.2: a) Máscara 3 x 3 hipotética. b)Imagem original tons de cinza. . . . . . . . 31

Figura 4.3: a) Conhecido como RectIter e percorre os pixels no sentido de cima para baixo obedecendo a ordem da esquerda para direita. b)Conhecido como RookIter e permite uma navegação arbitrária no que diz respeito a ordem. Nesse exemplo é seguido o sentido de cima para baixo porém a ordem varia entre esquerda e direita e da direita para a esquerda. . . . . . . . . . . . . 32

Figura 4.4: Esquema para o uso dos operadores JAI. . . . . . . . . . . . . . . . . . . . 37

Figura 4.5: Implementação da lógica central do componente de conversão para tons de cinza. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

Figura 4.6: a )Imagem original colorida. b )Imagem em tons de cinza. . . . . . . . . . 40

Figura 4.7: Visualização do histograma da imagem da figura 4.6 b). . . . . . . . . . . 41

Figura 4.8: a ) Interface gráfica confusa e desorganizada. b ) Uma interface com a mesma eficácia porém organizada através do agrupamento de funcionalidades similares. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

Figura 5.1: Um exemplo de aplicação do padrão arquitetural Layer: o modelo OSI. . . 46
Figura 5.2: visão das Camadas de um sistema com seus componentes internos e como eles se relacionam. Adaptado de Buschmann *et al.* [BUS 1996]. . . . . . . 48
Figura 5.3: Uma ferramenta de construção de diagramas UML para a modelagem de software orientado por objetos. . . . . . . . . . . . . . . . . . . . . . . . . . 50
Figura 5.4: Iterface gráfica construída com a mescla dos três *Look and Feel* nativos do Java. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Figura 5.5: Um exemplo clássico do uso do padrão Abstract Factory: mudando o estilo visual dos componentes gráficos do JDK. . . . . . . . . . . . . . . . . . . 53
Figura 5.6: Componente gráfico construído a partir de uma hierarquia de pacotes Java. (Imagem obtida da documentação HTML do j2sdk1.4.1_02) . . . . . . . . 55
Figura 5.7: O padrão de projeto Façade atuando para tornar um componente mais flexível. **a)** mostra a maneira confusa com que as novas classes E e F se comunicam com as classes do componente. **b)** a aplicação do padrão Façade para fazer o meio campo entre as interfaces dos componentes e das classes novas E e F. 56
Figura 5.8: Duas funções na linguagem C que expressam a mesma intenção. Adaptação de Buschmann *et al.* [BUS 1996]. . . . . . . . . . . . . . . . . . . . . 59

Figura 6.1: Usando os comentários de documentação para explicar um componente. . . 63
Figura 6.2: As três classes básicas do framework StoneAge. . . . . . . . . . . . . . . 64
Figura 6.3: Como é feito o controle das chamadas dos métodos por componentes gráficos. 65
Figura 6.4: Uma alternativa de controle mais legível. . . . . . . . . . . . . . . . . . . 66
Figura 6.5: Uma visão mais completa da estruturação do sistema base segundo o padrão de projeto de controle denominado de Command. . . . . . . . . . . . . . . 67
Figura 6.6: A primeira interface já funcional do StoneAge. . . . . . . . . . . . . . . 68
Figura 6.7: A coleção Vector utilizada para resolver o problema da operação “undo”. . 69
Figura 6.8: As operações de Undo e Redo sob a perspectiva de um padrão idiomático. . 70
Figura 6.9: A modelagem UML das operações `Save` e `Save As` . . . . . . . . . . . 71

Figura 7.1: O *kernel* de convolução utilizado: a ) filtro vertical e b ) filtro horizontal de Frei e Chen. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
Figura 7.2: Modelo Algébrico para encontrar o Thresholding, onde k é o limite especificado pelo usuário. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
Figura 7.3: As várias etapas necessárias para detectar a borda. . . . . . . . . . . . . . 75
Figura 7.4: O modelo estrutural do padrão Builder. O objeto Image é o produto que está sendo construído. . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
Figura 7.5: A visão de projeto completa do componente de segmentação baseado em máscaras. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
Figura 7.6: A interface gráfica do componente de segmentação baseado em contornos. 78

# LISTA DE TABELAS

Tabela 4.1: Alguns operadores pontuais presentes na JAI. . . . . . . . . . . . . . . . . . 35
Tabela 4.2: Operadores de área implementados pela JAI. . . . . . . . . . . . . . . . . . 35
Tabela 4.3: Operadores Geométricos: manipulam a forma, tamanho e orientação da imagem. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
Tabela 4.4: Operadores de arquivo: essenciais para iniciar qualquer processamento e/ou armazenar os resultados do mesmo. . . . . . . . . . . . . . . . . . . . . . 36
Tabela 4.5: Outros operadores muito usuais. . . . . . . . . . . . . . . . . . . . . . . . 36

Tabela 5.1: Os padrões de projeto de Gamma agrupados em categorias afins. . . . . . . 51

# LISTA DE ABREVIATURAS E SIGLAS

API *(Application Programming Interface)*. Interface de programação para construção de software.

AWT *(Abstract Window Toolkit)*. API que provê, entre outros, componentes gráficos de interface com o usuário.

CBSD *(Component-Based Software Development)*. Desenvolvimento de software baseado em componentes.

CVS *(Concurrent Version System)*. Mecanismo para gerenciar versões de software.

GUI *(Graphics User Interface)*. Interface gráfica com o usuário.

HCI *(Human-Computer Interaction)*. Interação entre homem e máquina.

JAI *(Java Advanced Imaging)*. API Java para processamento de imagens.

JDK *(Java Development Kit)*. Kit de Desenvolvimento Java.

OOP *(Object-Oriented Programming)*. Programação orientada por objetos.

PIGS *(Programmer's Imaging Generic System)*. Grupo de pesquisa dedicado ao estudo de imagens digitais da UFSM.

RMI *(Remot Method Invocation)*. Invocação de Métodos Remotos na construção de software cliente-servidor.

STL *(Standart Template Library)*. Biblioteca C++ para programação genérica.

UML *(Unified Modeling Language)*. Linguagem de Modelagem Unificada.

# SUMÁRIO

1 INTRODUÇÃO . . . . . 15
**1.1 Estrutura da dissertação** . . . . . 16

2 REFERENCIAIS TEÓRICOS . . . . . 17
**2.1 Construção de Software Adaptável** . . . . . 17
**2.2 Padrões de Projeto** . . . . . 18
**2.3 Desenvolvimento de Software Baseado em Componentes** . . . . . 20
**2.4 Linguagem de Modelagem Unificada - UML** . . . . . 21

3 A MELHORIA DA QUALIDADE DE PRODUTOS E PROCESSOS NA ENGENHARIA DE SOFTWARE . . . . . 23
**3.1 Utilizando um Processo de Software** . . . . . 24
**3.2 Programação Orientada por Objetos** . . . . . 25
**3.3 A Ferramenta de Modelagem Unificada para Projetar Software no Modelo de Objetos** . . . . . 27

4 DESENVOLVIMENTO DE SOFTWARE NO DOMÍNIO DE PROCESSAMENTO DE IMAGENS . . . . . 29
**4.1 A Camada Central** . . . . . 30
**4.2 A Camada de Aplicação** . . . . . 38
**4.3 A camada de Interface com o Usuário** . . . . . 41

5 PADRÕES DE CONSTRUÇÃO DE SOFTWARE PARA O PROCESSAMENTO DE IMAGENS . . . . . 43
**5.1 Os padrões Arquiteturais** . . . . . 43
5.1.1 Da Lama à Estrutura . . . . . 45
**5.2 Padrões Gamma** . . . . . 49
5.2.1 Padrões de Criação . . . . . 51
5.2.2 Padrões Estruturais . . . . . 54
5.2.3 Padrões Comportamentais . . . . . 56
**5.3 Padrões Idiomáticos** . . . . . 58

6 STONEAGE: UMA FERRAMENTA JAVA BASEADA EM COMPONENTES PARA O PROCESSAMENTO DE IMAGENS . . . . . . . . . . . . . . . . . . . . . . . . . 61
**6.1 A Linguagem de Programação** . . . . . . . . . . . . . . . . . . . . . . . . . 61
**6.2 A Aplicação dos Padrões no StoneAge** . . . . . . . . . . . . . . . . . . . . . . 64

7 APLICAÇÃO: UM COMPONENTE PARA SEGMENTAÇÃO DE IMAGENS BASEADO EM FILTROS DE CONVOLUÇÃO . . . . . . . . . . . . . . . . . . . . . . . . . 73
**7.1 O processo de segmentação** . . . . . . . . . . . . . . . . . . . . . . . . . 73
**7.2 Os padrões utilizados para projetar o componente** . . . . . . . . . . . . . . . . 75

8 CONCLUSÃO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

REFERÊNCIAS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

# 1 INTRODUÇÃO

Atualmente, o processamento e a análise de imagens estão sendo empregados nas mais diferentes áreas do conhecimento. Na área médica, por exemplo, as imagens são utilizadas para diagnosticar patologias. No domínio geoespacial, elas são utilizadas para visualizar o estado climático de uma região ou até mesmo para registrar o relevo de outros planetas. No campo comercial, as imagens estão cada vez mais presentes no cotidiano das pessoas através das câmeras digitais e scanners cada vez mais portáteis.

Porém, as imagens digitais, normalmente são dependentes de um software que gerencia todo o seu processamento ou análise. E é, justamente nesse contexto, que os padrões de projeto são utilizados. Os padrões de projeto são formas de construção de software comprovadamente funcionais que garantem legibilidade, fácil manutenção e reutilização de código-fonte no desenvolvimento de alguma aplicação computacional.

Os softwares existentes para computação científica envolvendo imagens, geralmente são concebidos para funcionarem sob uma única plataforma de sistema operacional além de, não serem de domínio livre, isto é, são proprietários. Um bom exemplo é a biblioteca MMORPH que só funciona se agregada no programa MATLAB e que, por sua vez, não disponibiliza livremente seu código-fonte para que seja aprimorado pela comunidade de usuários.

Nessas circunstâncias, essa dissertação tem por objetivo desenvolver um software para o processamento e análise de imagens que tenha como principais características a independência de plataforma operacional, código-livre e cuja implementação tenha sido baseada em padrões de projeto. Esse software recebeu o nome de *StoneAge*[1] e sua codificação foi feita através da linguagem de programação Java associada com a interface para programação voltada para imagens *Java Advanced Imaging* (JAI).

O *StoneAge* utiliza a idéia de componentização para resolver os vários problemas que envolvem as imagens. Ele funciona como um repositório de componentes independentes que, por sua vez, abrigam diferentes algoritmos para manipular as imagens. Porém, tanto esse repositório quanto os inúmeros componentes, se utilizam da idéia de padrões de projeto para assegurar as características anteriormente citadas como reuso, legibilidade e fácil manutenção de código-fonte. Essas características tornam um componente de software adaptável, isto é, que pode ser reaplicado em outros contextos de forma rápida e funcional. Assim, o problema apresentado

[1] O StoneAge pode ser obtido livremente em `(http://www.inf.ufsm.br/~welfer)`.

por essa dissertação é o de identificar e implementar padrões de projeto para o desenvolvimento de componentes de software adaptáveis para problemas de processamento e análise de imagens.

## 1.1 Estrutura da dissertação

O capítulo 2 apresenta a revisão bibliográfica dos principais tópicos abordados nessa dissertação como os padrões de projeto propriamente dito, o processo de componentização e a ferramenta de modelagem unificada (UML) que foi utilizada para projetar o *StoneAge*. No capítulo 3 é descrito a aplicação de processos de engenharia de software, como a UML e programação orientado por objetos, para obter software de qualidade. O capítulo 4 apresenta como ocorre o desenvolvimento de software especialmente para o processamento de imagens. O capítulo 5 descreve os padrões de construção arquitetural, de projeto e idiomáticos para a construção de software. O capítulo 6 apresenta os padrões aplicados no *StoneAge* além da linguagem de programação utilizada. A seguir, no capítulo 7, é mostrado um componente para segmentação de imagens presente no *StoneAge* e os padrões utilizados para a sua implementação. Por fim, no capítulo 8, são apresentadas as considerações finais desse trabalho.

# 2 REFERENCIAIS TEÓRICOS

Neste capítulo é apresentado uma visão global sobre a geração de software adaptável. Para isso, será descrito na seção 2.1 algumas noções de engenharia de software para a produção de componentes flexíveis. Na seção 2.2 descreve-se as principais características dos padrões de projeto. A seção 2.3 aborda o desenvolvimento de software baseado em componentes assim como seus riscos e desafios. A seção 2.4 encerra o capítulo abordando a linguagem de modelagem unificada - UML (*Unified Modeling Language*) e como a mesma é empregada para projetar componentes de software tanto em nível conceitual como na fase de implementação.

## 2.1 Construção de Software Adaptável

A implementação de um componente capaz de resolver problemas específicos em diferentes contextos e aplicações é uma tarefa difícil de ser realizada. Isso ocorre porque essa implementação não é concebida para ser um módulo ou artefato de código isolado, mas sim, para atuar em extensos, inúmeros e distintos pacotes de software que necessitam da mesma solução [WEI 1999]. Para isso se faz necessário incorporar ao desenvolvimento desse componente, técnicas de engenharia de software capaz de torná-lo, acima de tudo, adaptável.

Porém, para tornar uma solução adaptável é necessário prever as diversas situações onde ela pode ser utilizada, o que, na maioria das vezes, é de difícil percepção, principalmente para um desenvolvedor com pouca experiência. Nesse contexto, segundo Karsten Weihe [WEI 1999], os projetistas desse componente reutilizável devem se preocupar em usar uma abordagem tanto em baixo quanto em alto nível. A etapa de tornar o componente adaptável através de técnicas em baixo nível refere-se a reutilização de funções através de tipos parametrizados de dados, conhecidos como genéricos pelos usuários das linguagens Eiffel e Ada ou gabaritos STL *(Standart Template Library)* da linguagem C++ [GAM 2000]. A idéia central dessa abordagem é manipular tipos primitivos de dados como inteiros, reais, caracteres e outros de maneira simplificada. Para isso, uma das formas possíveis é definir um tipo abstrato de dados, como uma função ou uma classe, capaz de aceitar, sem especificações predefinidas, todos os tipos primitivos implementados pela linguagem de programação usada [JOS 1999][DEI 2001]. Dessa forma, ganha-se flexibilidade e independência de dados levando a implementação de um algoritmo a se tornar um componente de software adaptável a várias circunstâncias dentro de uma ou diversas aplicações.

Já na construção de código adaptável através de uma abordagem em alto nível, utiliza-se técnicas de programação orientada por objetos associada com as metodologias de construção de software baseadas principalmente em padrões arquiteturais e de projeto [BUS 1996] [GAM 2000]. Um padrão nada mais é do que a descrição de uma determinada solução para um problema que aparece com freqüência no projeto de software [COO 1998]. Dessa forma, uma vez que a equipe de projeto esteja familiarizada com certos padrões, a mesma pode aplicá-las na solução de problemas semelhantes [GAM 2000]. Assim, um padrão é um conjunto de informações sobre um dado problema que capta a sua estrutura essencial e caracteriza um conjunto de soluções comprovadamente bem sucedidas para o mesmo.

Dessa forma, para gerar componentes reutilizáveis e, naturalmente adaptáveis, para problemas de processamento de imagens digitais, não basta apenas identificar os objetos que são freqüentemente usados pelos programadores, como também interpretar esses objetos e manipulá-los de forma que suas interfaces[1] e seus relacionamentos com outros objetos, propiciem o seu acoplamento com outras aplicações de forma satisfatória. Esse é o grande desafio desse trabalho e um importante passo para auxiliar na implementação de componentes complexos muito comuns em problemas de processamento de imagens e que, posteriormente, possam ser necessários em muitos outros contextos. Assim, concluindo essa seção, deve-se projetar um componente que atenda o maior número de requerimentos possíveis, o que, muitas vezes, não é claro na fase de projeto, precisando assim, construir esse conhecimento antes da aplicação dos padrões de projeto anteriormente citados. Esse conhecimento, em engenharia de software, faz parte da definição do modelo conceitual e sua preocupação é a especificação do domínio do problema no mundo real para só então, iniciar a descrição dos componentes de software [LAR 2000].

## 2.2 Padrões de Projeto

A idéia de uma linguagem de padrões é originária do arquiteto e matemático Christopher Alexander, que no final da década de 70 escreveu dois livros que especificavam e descreviam o uso de padrões em projetos arquitetônicos [ALE 1977, ALE 1979] mais especificamente no planejamento de projetos de edifícios e cidades [GAM 2000]. Alexander provou que os métodos de construção até então utilizados geravam produtos que não satisfaziam os indivíduos e a sociedade, ou seja, não atendiam as suas reais necessidades. Na busca pela qualidade dos produtos e conseqüente satisfação das pessoas, Alexander estabeleceu 253 padrões que descreviam, em uma forma de catálogo, os problemas que se repetem em diferentes contextos ao mesmo tempo que, formalizava uma solução para os mesmos. Esses padrões alcançavam a qualidade porque, na sua solução, Alexander enfatizava a comunicação entre as pessoas que participavam, mesmo que indiretamente, da elaboração do projeto [SAL 1997] [LEA 1993]. A partir dessa idéia pode-se reutilizar uma solução na construção de casas, edifícios e outros obtendo resultados diferentes, otimizando o tempo de construção de alguma estrutura e, principalmente, sendo

[1] A interface aqui tem o sentido de especificar as operações que uma determinada unidade de programa define

capaz de atender os desejos dos usuários finais de alguma obra ou produto.

Quando essa idéia de identificar padrões de problemas foi utilizada no processo de desenvolvimento de software, principalmente através dos catálogos de Gamma *et al.* [GAM 2000] e Buschmann *et al.* [BUS 1996], uma nova maneira de pensar em projetos de software surgiu [MAY 2003].

Essa nova abordagem associada com o paradigma de programação orientada a objetos definiu 23 padrões de projeto comumente conhecidos como padrões da Gangue dos Quatro *(Gang of Four)* "GoF" referentes a Erich Gamma e seus colegas. Esses padrões são divididos em três categorias: os padrões de criação, padrões estruturais e os padrões comportamentais. A primeira categoria intuitivamente representa formas para o processo de criação ou instanciamento de objetos. A segunda identifica os padrões que compõe as classes ou grupos de classes e o último, isto é, os padrões comportamentais caracterizam a interação e responsabilidades entre os objetos, isto é, a comunicação entre eles [COO 1998]. Através desse catálogo de padrões consegue-se solucionar problemas de projeto assegurando a concepção de componentes adaptáveis e portanto reutilizáveis de fragmentos de código.

Porém esses não são os únicos padrões existentes. Segundo Steven Metsker [MET 2002] existem outros 77 padrões que, juntos, provavelmente caracterizam os 100 mais usuais. Entre esses outros padrões pode-se citar os de Buschmann *et al.* [BUS 1996], que se preocupa em aplicar formas ótimas de construção de software sob vários níveis como o arquitetural, o de projeto e o idiomático. O Padrão arquitetural cataloga formas para organizar a estrutura fundamental de um sistema, isto é, aquela parte que irá dar suporte básico a várias operações básicas. Já o de projeto é muito similar aos do Gamma, ou seja, normaliza a construção de componentes que serão agregados ao sistema e toda a comunicação envolvida entre os objetos responsáveis por essa tarefa. Por fim o padrão idiomático utiliza vantagens específicas da linguagem de programação que se está sendo utilizada para otimizar o desenvolvimento de algum componente. Esse padrão idiomático ocorre em baixo nível e nele se enquadram os *generics* das linguagens anteriormente descritas. Um bom exemplo do uso desses padrões idiomáticos é encontrado em [D´OR 2001] e [D'OR 2003], que utiliza o paradigma genérico promovido pela biblioteca STL da linguagem de programação C++ para amenizar a tríade dependência entre tipo de imagem, estruturas de dados e algoritmos utilizados na resolução de problemas de processamento de imagens.

Atualmente a comunidade que trabalha com padrões de construção de software tem crescido muito. Vários congressos sobre o tema foram implantados em várias partes do mundo. Entre esses congressos[2] cita-se o PLoP (*Pattern Languages of Programs*), SugarloafPLoP (Conferência Latino-Americana em Linguagens de Padrões para Programação), OOPSLA (*Object-Oriented Programming, Systems, Languages and Applications*), ECOOP (*European Conference on Object-Oriented Programming*), EuroPLoP (*European Conference on Pattern Languages of Programs*) e o VikingPLoP(*The Nordic Conference on Pattern Languages of Programs*).

[2]Informações mais detalhadas podem ser obtidas no site do grupo dedicado a aumentar a qualidade do desenvolvimento de software (`http://hillside.net/conferences/`).

Na próxima seção é descrito um modelo de organização de software conhecido como componentes. Esses componentes estão intimamente ligados aos padrões pois sua estrutura interna é concebida baseada em padrões de construção. Assim, conclui-se que os padrões são um importante passo para o processo de geração de código-fonte reutilizável porém, precisam ser complementados com a idéia de componentes para atribuir uma estrutura organizacional para os códigos desenvolvidos.

## 2.3 Desenvolvimento de Software Baseado em Componentes

Essa nova abordagem de planejamento de software adaptável exige uma implementação através de módulos denominados componentes. Na mesma medida com que o paradigma de desenvolvimento estruturado e orientado a objetos influenciaram programadores e projetistas do mundo inteiro, também assim, o desenvolvimento de software baseado em componentes tem emergido como a próxima revolução na concepção de sistemas computacionais [VIT 2003]. Essa tendência é conseqüência natural de uma série de vantagens como fácil manutenção, maior qualidade e tempo e custos reduzidos de desenvolvimento. Eles possuem manutenção acessível pois permitem substituir componentes obsoletos por outros mais robustos e estáveis; apresentam maior qualidade porque são testados e reutilizados em várias aplicações; e por fim, apresentam menor tempo e custo de desenvolvimento porque, utilizando-o para fins mais amplos, a sua codificação dá-se apenas uma vez e não proporcional ao número de vezes em que o problema surge.

Através das vantagens oferecidas pelo CBSD (*Component-Based Software Development*) é possível identificar três personagens principais: o desenvolvedor do componente, o montador e o cliente ou usuário final. De forma simples, o desenvolvedor é aquele que cria o componente, isto é, o programador ou fabricante. O montador é aquele que, conhecendo apenas as interfaces do componente, utiliza-o para resolver algum problema de forma totalmente aplicável, isto é, usando-o diretamente. E por último o cliente é o que recebe uma aplicação final construída com componentes previamente selecionados e que atendam sua lista de requisitos. A figura 2.1 demonstra a atuação desses três elementos que implementam o processo de desenvolvimento de software desde a sua concepção até o seu uso pelo cliente final.

Porém, como esses três elementos atuam sob fases e graus de conhecimento distintos sobre o processo de componentização, acabam se submetendo a um grande risco que pode comprometer a eficácia geral ou parcial de um sistema. O principal problema que pode ocorrer nessa abordagem é o efeito cascata[VIT 2003], isto é, qualquer erro na fase de desenvolvimento do componente, acarreta erros ou dificuldades na fase de montagem não atendendo, dessa forma, os desejos do cliente de forma satisfatória ou obrigando-o a se adequar ao sistema.

Um modelo parecido com o da figura 2.1 é proposto por Mohamed Fayad *et al.*, [FAY 2003]. Nele, a construção de um componente estável ocorre em três camadas. São elas: a EBTs (*Enduring Business Themes*), BOs (*Business Objects*) e a IOs (*Industrial Objects*). A camada EBT representa as classes que implementam a lógica permanente de um componente, isto é, o núcleo

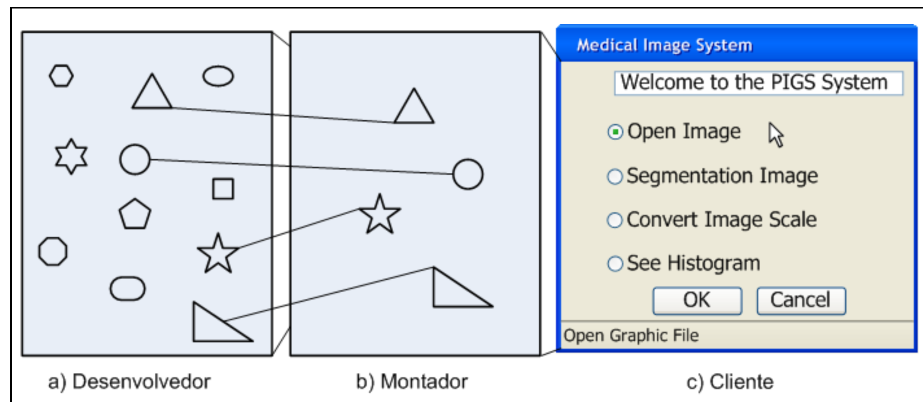


Figura 2.1: Os três personagens do processo de componentização segundo Padmal Vitharana, [VIT 2003]. a) pode-se observar que os componentes são fabricados para vários fins sendo que, não há uma relação explícita entre eles, isto é, são concebidos para serem independentes. b) O montador precisa encontrar e inter-relacionar os componentes a fim de resolver o problema do cliente. c) O produto de software final pronto para ser utilizado.

da implementação. Os BOs são classes que se utilizam da lógica oferecida pelo EBT para implementar mais objetos concretos. A camada mais externa IO é aquela que mapeia os BOs em objetos de aplicação, isto é, que associa um determinado objeto ou conjunto deles para resolver um problema real.

Dessa forma, o principal desafio do desenvolvimento de software baseado em componentes é o de criar mecanismos para gerenciar a refatoração de código que sempre ocorre quando os requisitos do sistema se modificam de tal forma que sua manutenção seja simples e rápida [BOO 1999]. Entre esses mecanismos cita-se o controle de versões, isto é, CVS *(Concurrent Version System)*, identificação profunda do domínio do problema, testes de validação, avaliação do grau de interoperabilidade entre os componentes, localização de componentes adequados durante o processo de montagem, checagem da procedência do componente, avaliação de como um sistema legado pode interagir com os componentes e qual o risco dessa operação entre outros. Como o processo de refatoração enfatiza outros aspectos que não o processo de criação de componentes adaptáveis, ele foge do escopo desse trabalho.

Na seção 2.4, é apresentado ainda a UML como uma importante ferramenta seja para trabalhar com padrões de projeto ou componentes. Ela pode ser vista como o complemento final necessário para a construção de software adaptável uma vez que é capaz de especificar todas as etapas necessárias para a sua implementação.

## 2.4 Linguagem de Modelagem Unificada - UML

Para projetar os componentes de software para os problemas de processamento de imagens, foi utilizada a Linguagem de Modelagem Unificada (UML). Atualmente ela é a notação gráfica mais amplamente utilizada para a modelagem de sistemas orientados a objetos pois sua linguagem é visual e rica em recursos necessários para formalizar, documentar e demonstrar a

criação, comportamento e estrutura dos objetos [DEI 2001], [ARM 2003], [ENG 2000]. Para que isso ocorra há uma série de convenções que são utilizadas para expressar a construção e o relacionamento dessas entidades de software tornando-a assim uma linguagem padrão onde qualquer pessoa pode, facilmente, interpretá-la e então entender a complexidade que o sistema apresenta. Esse estágio de alto nível da modelagem é mais conhecido como modelo estrutural e sua principal característica é a definição das classes utilizadas e como essas se relacionam entre si na solução do problema. No caso de um componente de software, por exemplo, pode-se ter um ou mais modelos estruturais dependendo da complexidade do sistema, isto é, utiliza-se uma hierarquia de componentes e sub-componentes que implementam modelos próprios a fim de modularizar a solução de forma mais legível. Dessa forma, a UML torna-se uma ferramenta essencial não apenas para documentar um sistema mas como também para desenvolver sua solução antes do processo de codificação.

A figura 2.2 utiliza a notação UML e a noção de componentização para demonstrar como o *StoneAge* foi projetado. O ícone central representa o framework, isto é, o software base implementado para abrigar vários componentes que foram desenvolvidos para um determinado fim. Esse nó central é representado por uma estrutura do tipo pacote com o estereótipo intuitivo à sua função, ou seja, um framework. Os componentes são representados pelos retângulos com duas guias horizontais e possuem um relacionamento de dependência com outra estrutura que representa seu código fonte gerador, isto é, a classe de visão pública que implementa toda a complexidade do componente. Já o ícone do framework apresenta também um relacionamento de dependência porém em relação aos componentes executáveis. Esta é uma forma bastante simplificada do esquema de desenvolvimento uma vez que, na prática, um componente pode ser formado por várias classes e/ou interfaces. Assim, para fins organizacionais, todos os componentes presentes no *StoneAge* foram abrigados em sub-pacotes.

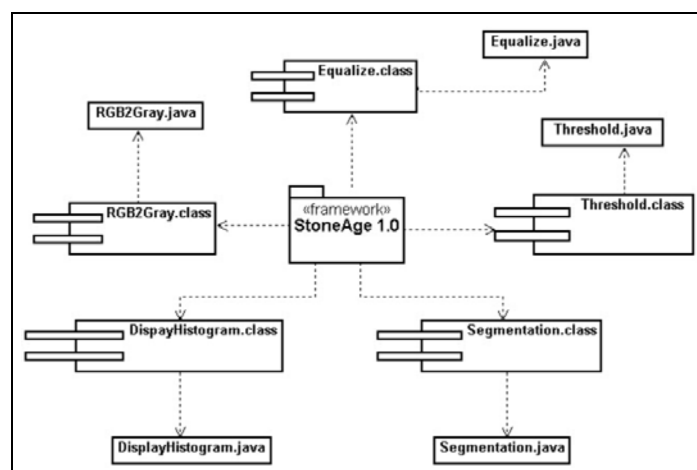


Figura 2.2: Esquema do sistema projetado para suportar a componentização.

No capítulo 3, é descrito mais detalhadamente as características da UML e a programação de software orientado por objetos como mecanismos para propiciar melhor qualidade tanto no processo quanto no produto de software que está sendo desenvolvido.

# 3 A MELHORIA DA QUALIDADE DE PRODUTOS E PROCESSOS NA ENGENHARIA DE SOFTWARE

A construção de um software de qualidade depende, principalmente, da sua conformidade com as especificações de seus requisitos [POP 2003]. Essas especificações, comumente conhecidas como engenharia de requisitos, ocorrem antes da fase de projeto e implementação pois é ela que vai formalizar o que se espera do artefato de software que será desenvolvido. Independente se é um componente ou um *framework*[1] que se está querendo construir, é essa especificação que irá descrever as necessidades e os desejos que essas entidades devem possuir e intuitivamente devem ser encontradas antes de qualquer passo posterior de desenvolvimento. Segundo Craig Larman [LAR 2000], essa etapa pode ser dividida em cinco fases para fins organizacionais: a descrição global sobre o que se quer desenvolver, quem é o cliente, quais são os objetivos, quais as funções do sistema e por fim quais os atributos desse sistema.

Tome-se como exemplo o desenvolvimento de uma classe para efetivar o realce de bordas de imagens visando possivelmente uma segmentação, ou em outras palavras a implementação de um componente para encontrar o gradiente de uma imagem. A primeira fase, consiste de uma simples especificação dos objetivos gerais onde se esclarece a finalidade do sistema, o que corresponde a fase de definir a interface do componente, ou seja, a entrada de uma imagem e a obtenção do resultante gradiente da mesma. A próxima fase é reconhecer o cliente desse componente. Aqui a unidade especialista de software é projetada para ser agregada a um *framework*. Dessa forma sua construção deve implementar primitivas que permitam sua união à essa entidade maior de software. Essas primitivas podem ser, por exemplo, características da linguagem de programação utilizada, isto é, se um *framework* foi projetado com determinada linguagem, possivelmente, para fins de compatibilidade, o componente fará uso da mesma ferramenta. Nos objetivos, pode-se enfatizar o processo de realce de bordas de forma mais detalhada, isto é, sob qual tipo de semântica o algoritmo funcionará, isto é, em imagens tons de cinza ou multiespectrais, qual algoritmo será utilizado para otimizar esse processo, quais as vantagens dessa solução em relação aos outros algoritmos. Já nas funções do sistema, especifica-se como ele opera, em relação às suas entradas, por exemplo, se o usuário pode escolher qual algoritmo de uma coleção de outros vai ser aplicado na imagem ou se o sistema detectará isso automaticamente. Na última

[1] software que agrega os componentes desenvolvidos.

fase, isto é, na identificação dos atributos do sistema pode-se visualizar um componente com interface gráfica para facilitar a sua utilização por parte do usuário, outro atributo seria a sua modularidade, isto é, se ficaria mais legível o separar em diversas classes entre outros.

Dessa forma, construir e refinar um repositório de requisitos é um importante passo para assegurar a construção de um software de qualidade. Porém, isso não garante que o produto final foi implementado da melhor forma possível uma vez que, por motivos de pouca experiência, o código fonte produzido pode carecer em simplicidade tornando-se muito extenso e pouco lógico. Assim, ele pode sofrer refatoração várias vezes durante seu ciclo de desenvolvimento a fim de tornar-se mais enxuto porém sem alterar sua funcionalidade original. Sua operacionalidade original é mantida porque a interface do componente é respeitada, porém a sua complexidade interna pode sofrer alterações [FOW 1999]. Construir um código "limpo" auxilia na manutenção que ele virá a sofrer com as mudanças em suas especificações funcionais em decorrência de alguma nova necessidade por parte do usuário.

## 3.1 Utilizando um Processo de Software

Através do que foi escrito até o momento pode-se perceber que o desenvolvimento de software que almeja certas qualidades operacionais não é uma tarefa trivial. Isso deve-se principalmente ao fato de serem construções extremamente abstratas e, portanto, complexas. Peters e Pedrycs [PET 2001] expressam bem esse problema: *"...eles estão entre as construções humanas contemporâneas mais abstratas, pois não são regidos por nenhuma lei da física. Particularmente eles não envelhecem, não ocupam espaço, não se desgastam e não apresentam continuidade."*

Nesse contexto faz-se necessário utilizar uma metodologia que auxilie o seu desenvolvimento desde o princípio, garantindo que o produto seja confiável, eficaz, eficiente e de fácil manutenção. A esse recurso utilizado dá-se o nome de *processo de software* e seu objetivo é especificar métodos e técnicas a serem utilizados pelas pessoas, no caso projetistas e programadores, a fim de desenvolverem e manterem em algum tipo de artefato de software [DON 2001]. Nesse estágio são utilizadas ferramentas de engenharia de software que foram criadas e vem evoluindo há mais de 30 anos e sua função é prover mecanismos para analisar e validar as três fases principais do processo de desenvolvimento de software que são: a análise dos requisitos, a fase de projeto e a fase de implementação. A fase de requisitos é a mesma descrita no início desse capítulo, ou seja, visualiza o sistema como uma "caixa-preta" se preocupando apenas com o que é entrada ou é saída de dados. A fase de projeto trata das especificidades do sistema, isto é, qual o algoritmo será utilizado para prover a solução exigida pelos requisitos, qual a linguagem de programação, entre outros. Já a fase de implementação refere-se a geração de código-fonte, documentação e testes de validação. Para gerenciar essas três fases vários modelos de ciclo de vida de software vem sendo estudados entre eles: o modelo cascata, incremental, espiral, evolucionário, de prototipação, orientado a objetos entre outros [PET 2001]. Esses modelos são chamados de universais pois descrevem apenas as fases

básicas do processo de desenvolvimento, o que os tornam bastante genéricos e, portanto, pouco precisos.

Mesmo assim faz-se necessário escolher um deles porém com um auxílio adicional. Dessa forma, o próximo passo necessário é escolher uma ferramenta que dê suporte à esse modelo escolhido. No caso dessa pesquisa foi utilizado o modelo de programação orientado por objetos associado com a ferramenta de modelagem gráfica UML para construir o *StoneAge*.

## 3.2 Programação Orientada por Objetos

Na perspectiva de programação por objetos, surgida na segunda metade da década de 70, o desenvolvimento de um sistema é centrado em uma entidade denominada classe, enquanto que a visão tradicional enfatiza os algoritmos de forma seqüencial. Até aqui não há nenhum problema, pois ambas as formas conseguem resolver os problemas exigidos pelos requisitos dos usuários. O problema é que a abordagem de codificação estruturada de software, isto é, aquela centrada nos algoritmos com suas funções ou procedimentos, torna-se muito complexa quando o sistema vai ganhado proporções maiores o que, conseqüentemente, torna sua manutenção muito difícil [BOO 1999]. Esse problema é fácil de enxergar mesmo que teoricamente pois se o algoritmo é a unidade que implementa toda a lógica de uma solução e o processo de desenvolvimento é atrelado a ele, logo todo o código produzido fica dependente da complexidade que ele implementa.

Já no modelo de objetos, comumente chamado de OOP *(Object-Oriented Programming)* consegue-se ocultar a complexidade de alguma solução e manipulá-la de forma mais flexível, ou seja, com a possibilidade de criação de várias instâncias de classes, com a utilização das vantagens do encapsulamento, e com o reuso mais efetivo de objetos para resolver problemas específicos em outros contextos [PET 2001]. Com essa modelagem outra característica muito desejável e importante em um sistema é alcançada, isto é, a legibilidade. A legibilidade é conseguida porque o desenvolvedor/programador pode organizar seu código fonte em classes, que nada mais são que objetos em tempo de codificação, a corresponderem ao problema que se está sendo analisado [COA 1993]. Por exemplo, na construção de um componente que realiza a operação de equalização em uma imagem o programador pode criar uma série de classes que resolvem várias etapas desse processo isoladamente. Por exemplo, o primeiro passo é adquirir a imagem, então uma classe deverá ser criada para permitir o acesso a matriz de pixels de determinado arquivo gráfico. Após isso outra classe deve ser desenvolvida para computar o histograma dessa imagem uma vez que a equalização é uma operação baseada no histograma. O próximo passo é computar, na mesma classe, o histograma normalizado e acumulado encontrando assim a função de transferência que irá refletir a nova distribuição das intensidades dos pixels na imagem [SAN 1995]. Essas últimas operações são implementadas na mesma classe porque não são complexas e dispensam qualquer modularidade. Por fim deve-se criar uma classe para exibir a imagem na tela através de uma janela. A figura 3.1 demonstra esse processo.

Os três retângulos dessa figura representam as classes utilizadas para resolver o problema,

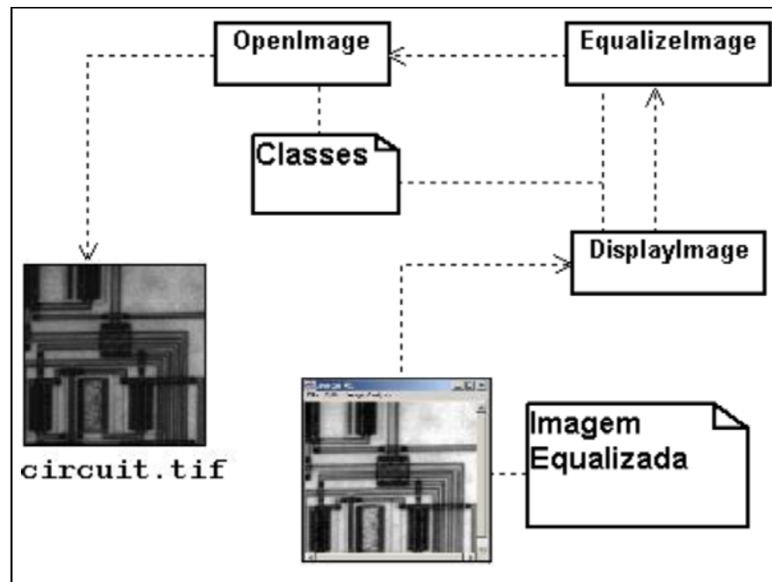


Figura 3.1: Uso do diagrama de classes para o problema da equalização de imagens

sendo que, cada uma delas é representada por um arquivo com extensão `class`[2]. Os métodos e atributos de cada classe foram abstraídos por motivos de simplificação. Outra notação importante e muito útil presente na figura 3.1 é o retângulo com a borda superior direita dobrada. Esse é o ícone de uma nota, ou um *note* e é muito utilizado para explicar visualmente algum processo ou entidade. Ele pode ser visto como uma documentação visual resumida do diagrama. Assim, para o funcionamento eficaz do programa a classe implementada pelo arquivo `OpenImage.class` necessita do arquivo gráfico que irá conter a imagem[3] a ser equalizada. Após isso a classe representada pelo arquivo `EqualizeImage.class` fará todo o processamento que irá efetivar a operação de equalização. Após isso a classe `DisplayImage` irá exibir essa imagem no monitor através dos componentes de interface gráfica nativos da linguagem Java. Essas classes apresentam-se sob uma relação de dependência. Por exemplo, a classe `DisplayImage` depende da classe `EqualizeImage` para funcionar corretamente. Essa relação de dependência ocorre através da convenção gráfica de uma linha tracejada com uma seta aberta em uma das extremidades.

Essa operação de equalização é muito importante para facilitar a visualização de uma imagem. Comparando a imagem original com a pós-processada pode-se notar que essa última está mais nítida pois suas bordas foram realçadas por esse processo de distribuição mais uniforme da luminosidade, que é o objetivo da equalização. Através da equalização, por exemplo, pode-se avaliar melhor visualmente o resultado de uma segmentação [4].

[2]Arquivo com essa extensão representa o programa executável que é interpretado pela máquina virtual Java.

[3]A imagem utilizada nesse exemplo é parte integrante do toolbox de processamento de imagens do MatLab - (`http://www.mathworks.com/`) © 1994-2004 The MathWorks, Inc.

[4]O uso de algoritmos de segmentação foi abordado em trabalho publicado no XXIV Encontro Nacional de Engenharia de Produção, [WEL 2004] e no X International Conference on Industrial Engineering Management, [D'OR 2004].

## 3.3 A Ferramenta de Modelagem Unificada para Projetar Software no Modelo de Objetos

A UML foi desenvolvida logo após as primeiras linguagens de programação orientada por objetos como um meio para analisar e projetar os novos sistemas de software centrados nesse novo paradigma [BOO 1999]. Criada por Grady Booch, Ivar Jacobson e James Rumbaugh em 1995, com a denominação de método unificado, foi aperfeiçoada e tornou -se a linguagem padrão de modelagem [5] da indústria de desenvolvimento de software [KOB 1999].

Essa ferramenta possui diversas notações gráficas capaz de atender as várias construções propiciadas por uma linguagem orientada por objetos. Sua operacionalidade pode ser dividida em quatro categorias: estrutural, comportamental, de agrupamento e de anotação. A seguir é apresentado uma descrição sobre cada grupo:

- o grupo estrutural da UML abriga, principalmente, as notações de diagrama de classes, objetos, componentes, interfaces e os seus relacionamentos. Por exemplo, as figuras 2.2 e 3.1 utilizam- -se respectivamente dos diagramas de componentes e de classes para demonstrar exemplos concretos do projeto de software em diferentes níveis de complexidade.

- o grupo comportamental modela as características dinâmicas de um sistema, ou seja, os passos que ele realiza para resolver algum problema no decorrer do tempo. São representantes dessa categoria o diagrama de atividades, de estados, de casos de uso, de colaboração e de seqüências. A figura 3.2 mostra o uso do diagrama de atividades para projetar e documentar o fluxo de informações manipuladas para distribuir mais uniformemente a luminosidade em uma imagem.

- na terceira categoria, isto é, no agrupamento cita-se o diagrama de pacotes que pode abrigar em seu interior outros pacotes ou estruturas a fim de criar uma hierarquia para fins organizacionais.

- O anotacional apenas existe para fornecer informação visual sobre algum processo ou esquema. Por exemplo, a figura 3.1 utiliza duas dessas estruturas para explicar visualmente o que está acontecendo ou como o sistema foi modelado, isto é, ele pode servir como uma espécie de lembrete para o desenvolvedor.

A UML não serve apenas para projetar um sistema de software, ela serve também, para documentá-lo. Dessa forma a tradicional documentação de código via estrutura de comentários ganha um grande aliado uma vez que, a UML, é visual e assim consegue-se enxergar e entender os relacionamentos que as várias entidades de software podem implementar entre si. Esse processo ameniza a complexidade do sistema, o que por sua vez facilita a manutenção do mesmo.

[5] A UML é padronizada e mantida pela OMG - Object Management Group. (`http://www.omg.org`)

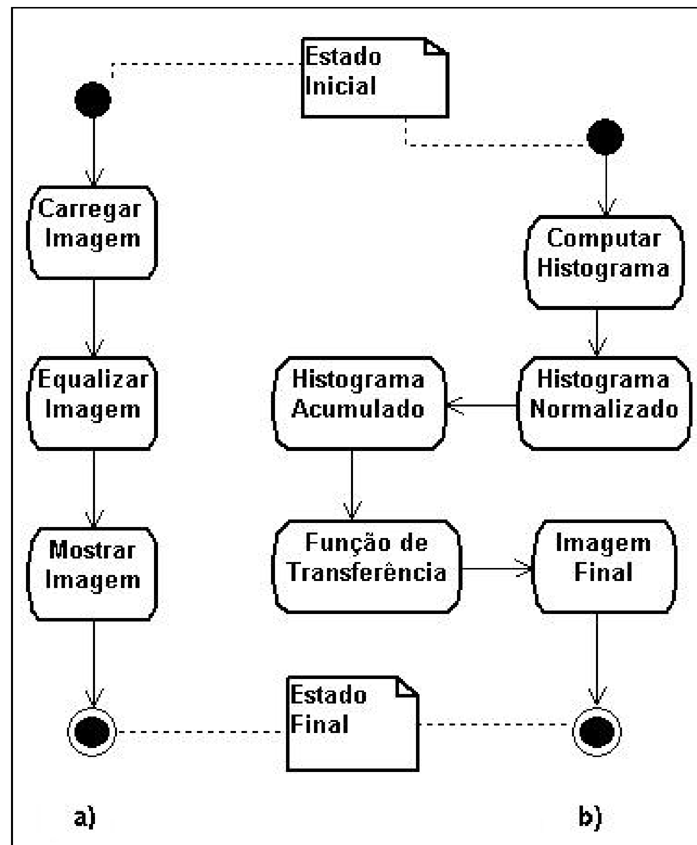


Figura 3.2: A operação de equalização demonstrada via diagrama de atividades. a ) exibe o processo de equalização mais genérico se preocupando apenas com a entrada e saída de informações. b ) "Explode" o processo de equalização original permitindo assim visualizar a seqüência lógica exigida pelo algoritmo. Em ambas ilustrações o círculo preto representa o início do processo e o círculo circunscrito em outro o seu término.

Além disso, ela permite refinar o processo de concepção do sistema por qualquer desenvolvedor através de suas notações gráficas padrão.

Portanto, um modelo de programação associado com uma linguagem gráfica de modelagem tende a produzir um software de qualidade, uma vez que, em seu processo de projeto deve-se conseguir tratar o maior número possível de exceções evitando assim a futura refatoração de código o que significa economia de tempo e dinheiro.

No próximo capítulo, é apresentado as principais características do processo de desenvolvimento de software no domínio do processamento e análise de imagens. Essa etapa é fundamental para posteriormente identificar e aplicar os padrões de construção na implementação do *StoneAge*. Assim, é essencial para a correta construção de código adaptável entender tanto o domínio da aplicação quanto dos conceitos de engenharia de software.

# 4 DESENVOLVIMENTO DE SOFTWARE NO DOMÍNIO DE PROCESSAMENTO DE IMAGENS

A computação voltada para imagens existe há mais de trinta anos. No começo, ela se restringia aos grandes centros e laboratórios de pesquisa devido aos caros recursos computacionais que eram empregados[SHA 2000]. Segundo Gonzalez e Woods [GON 2000], o processamento de imagens começou a ganhar importância em 1964 quando o Laboratório de Propulsão a Jato do Centro de Tecnologia de Pasadena, na Califórnia, utilizou um computador de grande porte associado a técnicas de processamento para realizar o melhoramento de imagens da Lua que foram transmitidas pela sonda Ranger 7.

Porém, com a evolução do hardware no que diz respeito ao aumento de poder de processamento, redução dos custos de memória, advento das linguagens de programação de alto nível, aumento da capacidade dos dispositivos de armazenamento aliados a existência de sistemas operacionais voltados para usuários não especialistas, a computação voltada para imagens tornou-se acessível a quase todos. Dessa forma, o desenvolvimento de software para processamento de imagens chegou ao alcance de qualquer pessoas dotada de um simples computador de mesa ou até mesmo portátil tornando-se assim, um tema muito atrativo e útil para pesquisadores das mais diferentes áreas do conhecimento. Para Gonzalez e Woods [GON 2000] aplicações que envolvem imagens podem ser encontradas em astronomia, biologia, defesa, medicina, apoio a lei, arqueologia, física, geografia, em aplicações industriais, geologia, engenharia e ciência dos materiais.

No entanto, apesar do fácil acesso aos recursos computacionais atuais necessários para a programação de software para resolver problemas de imagens, deve haver um certo cuidado quanto ao seu projeto, isto é, na forma como será implementado. Normalmente, o desenvolvimento de grandes sistemas de software, o que é muito característico no desenvolvimento voltado para imagens, ocorre em camadas [VÖL 2002]. Dentre essas, as mais utilizadas são: a camada central ou núcleo, a camada de aplicação e a de apresentação. O principal objetivo dessas camadas é tornar o software flexível em relação as suas funcionalidades, isto é, decompô-lo em blocos de funções similares no seu nível de abstração [BUS 1996]. Como por exemplo pode-se citar a última camada que tem a função de interagir com o usuário e, cujo objetivo, é se preocupar apenas em propiciar uma entrada e saída de informações que torne o sistema fácil de usar. A figura 4.1 apresenta um esquema organizacional baseado na idéia de camadas.

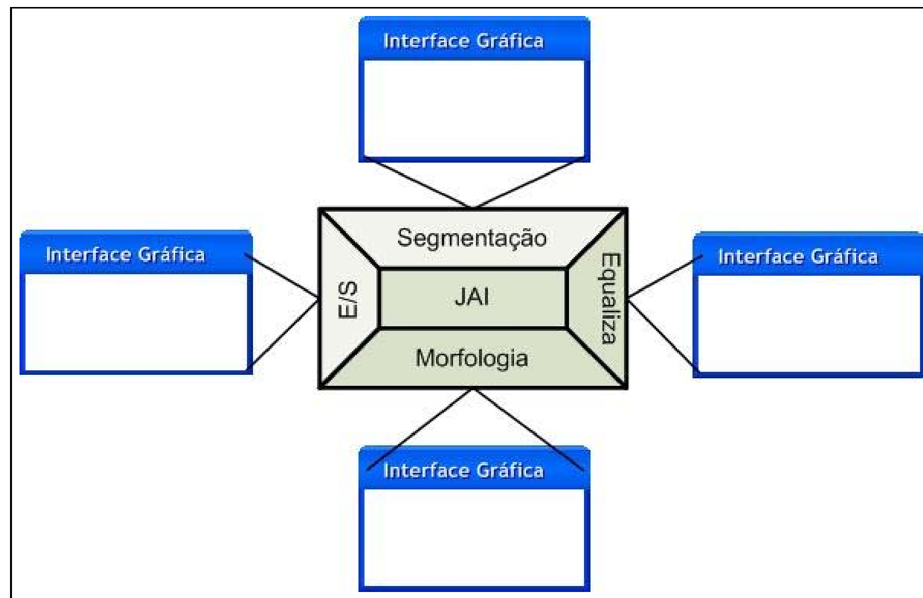


Figura 4.1: Desenvolvimento de software em camadas.

Na figura 4.1 o núcleo é representado pela interface de programação para imagens chamada JAI (*Java Advanced Imaging*). Essa camada central é responsável pelos algoritmos essenciais na área de processamento de imagens. Em sua volta encontra-se a camada de aplicação onde, utiliza-se uma série de operadores providos pelo núcleo para resolver determinados problemas de manipulação de imagens. Nessa figura, a título ilustrativo, são citadas aplicações de morfologia, abertura de arquivos gráficos, segmentação e uma aplicação para linearizar um histograma, isto é, a operação de equalização. A última camada é a de interface com o usuário e comumente chamada de camada GUI (*Graphics User Interface*). Essa camada é de fundamental importância pois é através dela que o usuário será capaz de utilizar o sistema. Dessa forma, uma camada de interface gráfica bem projetada pode diminuir o grau de complexidade necessária para operar um sistema.

## 4.1 A Camada Central

A camada central ou núcleo é responsável pela implementação de um conjunto de funcionalidades básicas para processamento de imagens. Enquadram-se nessas funcionalidades operadores pontuais, de área, geométricos, de manipulação de arquivos gráficos, de realce de bordas, de quantização, de região de interesse e operações morfológicas. Todas essas implementações são utilizadas em diferentes situações como especificado a seguir:

- no momento de extrair alguma informação de determinado arquivo gráfico. Com os inúmeros formatos atualmente existentes é muito útil que a camada central se preocupe em decodificar pelo menos os tipos mais conhecidos como, por exemplo, JPEG, BMP, TIFF, PNG, PGM e GIF. Esse processo é também crucial uma vez que, quase todo o processamento ou análise de imagens necessita de matrizes bidimensionais que representam os

pixels da imagem e que, por sua vez, são as principais informações presentes nos arquivos gráficos mais comumente usados em laboratório.

- no momento de "realçar" uma imagem, o que pode ser realizado em dois domínios básicos: O domínio espacial e o domínio da freqüência. No domínio espacial pode-se citar os filtros ou máscaras de convolução utilizados para amenizar o efeito do ruído em imagens, realçar detalhes em imagens borradas e outros. Ele é dito espacial pois opera sob o próprio plano da imagem, isto é, diretamente sob seus pixels [GON 2000]. A figura 4.2 ilustra a aplicação de uma máscara. Já o domínio da freqüência baseia-se na transformada de Fourier da imagem, isto é, na taxa de transição que o brilho apresenta em uma imagem [GON 2000].

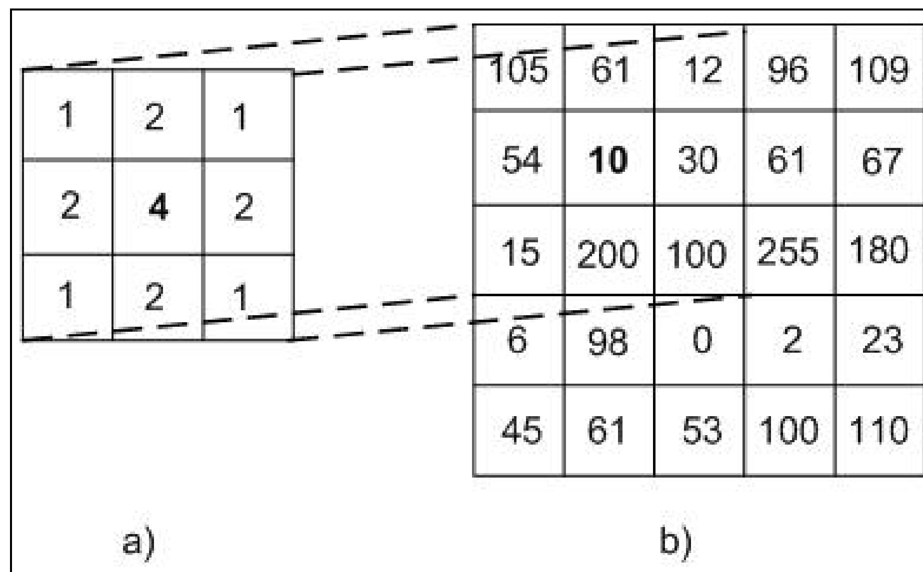


Figura 4.2: a) Máscara 3 x 3 hipotética. b)Imagem original tons de cinza.

A máscara da figura 4.2 é também dita oito-conectada ou $N_8$ porque utiliza oito pixels vizinhos em relação ao seu centro. O pixel central da máscara, que também pode ser vista como uma imagem, é computado para todos os pixels da imagem original mais a sua vizinhança local. Por exemplo, para encontrarmos o valor dos pixels resultantes da convolução da máscara 3 x 3 sobre uma imagem hipotética representada na figura 4.2 deve-se multiplicar cada valor do pixel da máscara por cada valor do pixel da imagem original e por fim efetuar a soma. Como esses novos pixels podem exceder o intervalo de intensidades que se está sendo representado eles podem sofrer uma normalização de acordo com o peso da máscara [SHA 2000]. Segundo Gonzalez e Woods [GON 2000], as técnica espaciais são muito mais freqüentemente usadas do que as técnicas baseadas no domínio da freqüência pela sua fácil implementação e performance computacional.

- nas operações geométricas da imagem como a rotação, translação, shear, escala e outros.

- no processo de análise de imagens através da detecção de arestas, geração de histograma,

encontro do máximo e ínfimo valor de uma imagem.

- no armazenamento de novas imagens em arquivos. Esse processo é de extrema importância uma vez que através dele é possível ter uma saída para o processamento realizado mantendo assim a imagem original inalterada. Para isso, o núcleo deve ser capaz de codificar os algoritmos usados pelos principais formatos gráficos permitindo assim, anexar a matriz de pixel resultante de alguma operação realizada em um novo e válido arquivo de imagens.

- no acesso aos pixels de uma imagem o núcleo deve implementar formas para que isso ocorra de forma simples e rápida. Para isso vários tipos de iteradores[1] devem ser implementados permitindo que, a matriz de pixel seja percorrida de diferentes formas visando assim, uma melhor performance de processamento. A figura 4.3 demonstra dois exemplos desses iteradores e como eles funcionam.

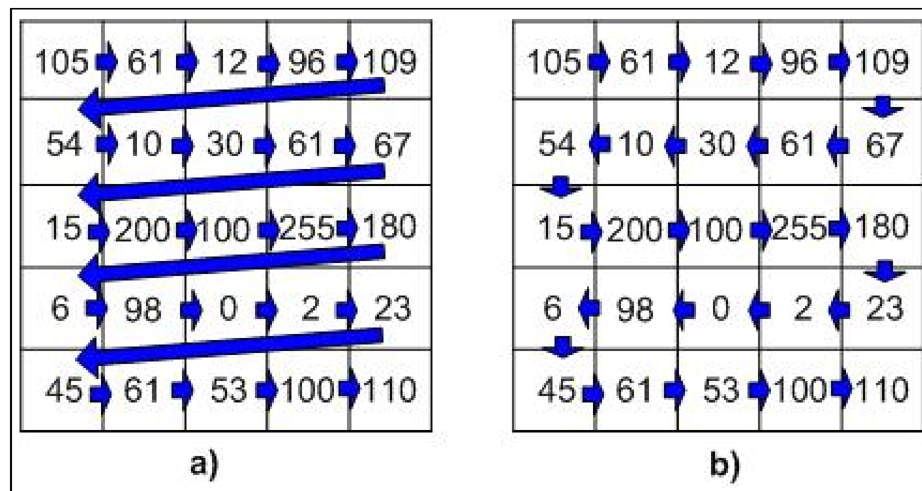


Figura 4.3: a) Conhecido como RectIter e percorre os pixels no sentido de cima para baixo obedecendo a ordem da esquerda para direita. b)Conhecido como RookIter e permite uma navegação arbitrária no que diz respeito a ordem. Nesse exemplo é seguido o sentido de cima para baixo porém a ordem varia entre esquerda e direita e da direita para a esquerda.

Esse núcleo geralmente é implementado através de bibliotecas de programação. Porém, muitas delas são especialistas,isto é, atendem apenas funcionalidades específicas [Sun 1999]. Um bom exemplo é a biblioteca Mmorph[2] que contém o estado da arte no que se refere a operadores morfológicos [D´OR 2001]. No entanto, essa biblioteca além de apenas se preocupar com os aspectos relativos à morfologia das imagens seja ela cinza, binária ou colorida, ela só funciona agregada em um outro software chamado MATLAB. Esse sistema que é a abreviatura para MATrix LABoratory manipula essa biblioteca para fins de interação com o usuário além de,

[1] mecanismo que propicia o percorrimento dos pixels de uma imagem através de diferentes maneiras.

[2] Essa biblioteca pode ser obtida em `http://www.mmorph.com` com licença para trinta dias.

agregar novas funções. O MATLAB[3] nada mais é que um software que agrega várias bibliotecas especializadas como a de processamento de imagens, de morfologia, de otimização linear, análise financeira, lógica fuzzy, equações diferenciais parciais, estatística, e outros.

Nessa dissertação foi utilizada a interface de programação para imagens denominada JAI que, atualmente, está na versão 1.1.2. Essa API (*Application Programming Interface*), como é comumente chamada, é a biblioteca central sobre a qual a maioria das operações com imagens foram desenvolvidas. Essa API foi desenvolvida pela Sun Microsystems [4] em conjunto com outras grandes empresas como a Siemens Corporate Research, Eastman Kodak e Autometric visando, principalmente, atender as necessidades da área médica, geoespacial, comercial, governamental e da internet [Sun 2004]. A seguir são abordados alguns aspectos dessas necessidades.

- Na área médica essa biblioteca permite a construção de aplicativos capazes de auxiliar no diagnóstico médico por imagens através de processamentos como: segmentação, determinação de regiões de interesse, equalização, filtros anti-ruídos ou/e de desborramento, aumento escalar da imagem entre outros. Algumas dessas formas de processamento são utilizadas para o processo de quantificação de imagens, ou seja, para conseguir extrair determinadas informações úteis para o profissional da saúde que depende da imagem para uma avaliação mais criteriosa sobre determinada doença;

- No âmbito geoespacial como destaca [Sun 2004], essa API pode ser utilizada para manipular grandes imagens obtidas via satélite obtendo assim informações pertinentes, por exemplo, do estado climático de determinadas regiões, seu comportamento atual, sua previsão, seus recursos naturais entre outras. Esses sistemas são conhecidos como SIGs, isto é, Sistemas de Informações Geográficas;

- Quanto às necessidades governamentais poder-se-ia citar também, o uso de imagens obtidas via satélite para analisar o prejuízo que um tornado ou um terremoto causou ou causaria em algum lugar, para analisar se determinada lavoura foi cultivada ou não, entre outros. Um exemplo atual é a utilização da API JAI em conjunto com outra API chamada Java 3D pela Agência Espacial Norte-Americana - NASA. Os cientistas da NASA usaram essas bibliotecas Java para controlar as sondas *Spirit* e *Opportunity* na missão de explorar o planeta Marte. O software construído para isso foi chamado de MAESTRO[5] e seu objetivo era criar um mundo tridimensional baseado nas imagens captadas por essas sondas ao mesmo tempo que, permitia a manipulação mecânica desses robôs exploradores [GOS 2004].

- No domínio comercial, atualmente com a existência de dispositivos de aquisição de imagens mais acessíveis a população em geral, como por exemplo, as câmeras digitais e scan-

[3]Maiores informações sobre esse sistema, que é uma verdadeira linguagem para computação técnica, podem ser obtidas em `http://www.mathworks.com`.

[4]© 2000 Sun Microsystems, Inc. (`http://www.sun.com`)

[5]Esse sistema pode ser baixado livremente mediante cadastro em (`http://mars.telascience.org/`)

ners de mesa, a atenção dada para a criação de novos softwares que manipulam imagens é evidente. Dessa forma, para não reimplementar uma série de funcionalidades básicas o uso dessa API pode ser visto como uma alternativa lógica para otimizar o processo de desenvolvimento de software.

- As demandas referentes à Internet referem-se a capacidade que essa biblioteca, baseada na linguagem Java, possuí para manipular imagens remotamente através do modelo cliente-servidor. Na realidade essa vantagem adicional é conseqüência da linguagem Java que já permitia a construção de aplicações baseada em objetos remotos através de RMI[6](Remote Method Invocation) [DEI 2001].

Além da vantagem de possibilitar a implementação de objetos distribuídos de imagens, a API JAI é multiplataforma. Isso ocorre porque ela foi escrita em Java, e, conseqüentemente a idéia de "escrever uma vez e executar em qualquer lugar" é válida [Sun 1999]. Claro que essa independência de plataforma é relativa, isto é, ela funciona em qualquer sistema operacional desde que haja uma máquina virtual apropriada instalada para que a mesma seja interpretada. Outra característica muito positiva que essa biblioteca herdou da linguagem Java diz respeito a metodologia utilizada para sua construção. Essa API foi escrita com o paradigma orientado por objetos, apresentando-se dessa forma uma organização baseada em classes e métodos. Essa é uma característica positiva, pois facilita o reuso, legibilidade e a modularidade do código fonte. Em nível de implementação, por exemplo, se quisermos utilizar a JAI para manipular um histograma apenas importamos a classe `Histogram` pertencente ao pacote `javax.media.jai`. Dessa forma tem-se acesso a todos os atributos e serviços (métodos) providos por essa classe relativo ao histograma de uma imagem de entrada.

Como foi dito no início desse capítulo, a camada central oferece uma série de operações básicas para a implementação de software para processamento de imagens. Nesse contexto pode-se identificar nove categorias de operadores implementados pela biblioteca JAI, são eles: Operadores pontuais, operadores de área, operadores geométricos, de cor, de arquivo, de freqüência, de estatística, de extração de bordas e operadores gerais [AKR 2003]. E é justamente essa gama de funcionalidades que a difere das bibliotecas especialistas como a Mmorph anteriormente citada.

A seguir são apresentados os principais operadores segundo as categorias mais usuais dentre as nove anteriormente citadas. Todos esses operadores serão mostrados, para melhor organização, em tabelas como a 3.1. A primeira coluna dessas tabelas especifica o nome do operador que, intuitivamente será passado como parâmetro posteriormente, isto é, na hora de programar. A segunda coluna dá a funcionalidade desse operador, isto é, o que será feito a partir de uma ou duas imagens de entrada. E, por fim, se existir, a terceira coluna mostra como esse operador deve ser utilizado durante o processo de codificação através de uma forma adequada, isto é, sem erros em tempo de compilação ou execução.

[6]O uso de objetos distribuídos via RMI aplicado a imagens foi descrito em trabalho publicado no XXIII Encontro Nacional de Engenharia de Produção e IX International Conference on Industrial Engineering Management. [WEL 2003]

- Os operadores pontuais atuam apenas sob os atributos de cada pixel da imagem, isto é, os pixels da nova imagem dependem somente da intensidade isolada da cada pixel da imagem de entrada. A JAI implementa quatro dezenas de operadores pontuais. A tabela 4.1 demonstra alguns desses operadores.

| Operador | Função | Estrutura Básica |
|---|---|---|
| Add | Realiza a operação de adição de duas imagens. | dst = JAI.create("Add", pb, null);. |
| And | Realiza a operação de AND lógico entre duas imagens . | dst = JAI.create("And", pb, null);. |
| BandCombine | Usado para manipular as bandas de uma imagem. | dst = JAI.create("BandCombine", pb, null);. |
| Threshold | Computa o limiar de uma imagem. | dst = JAI.create("Threshold", pb, null);. |
| Invert | Inverte os valores dos pixels da imagem. | dst = JAI.create("Invert", pb, null);. |
| Subtract | Subtrai os pixels de duas imagens. | dst = JAI.create("Subtract", pb, null);. |

Tabela 4.1: Alguns operadores pontuais presentes na JAI.

- Os operadores de área podem ser utilizados para implementar as máscaras anteriormente citadas, para localizar as coordenadas espaciais de uma sub-imagem, para adicionar borda ao redor de uma imagem, para implementar filtros de medianas capazes de suavizar uma imagem entre outros. A JAI implementa quatro desses operadores que podem ser conferidos na tabela 4.2.

| Operador | Função |
|---|---|
| Border | Adiciona bordas ao redor da imagem. |
| BoxFilter | Determina a intensidade de um pixel de acordo com a média dos pixels pertencentes a uma determinada área retangular da imagem fonte. |
| Convolve | Implementa a operação espacial entre máscara e imagem. É ele que é utilizado para implementar a operação representada pela figura 4.2 |
| Crop | `Recorta` uma área retangular da imagem. |
| MedianFilter | Filtro de mediana utilizado para remover linhas ou pixels isolados. |

Tabela 4.2: Operadores de área implementados pela JAI.

- Há também a categoria dos operadores geométricos que modificam a orientação, forma e tamanho de uma imagem. Na tabela 4.3 é possível ver alguns desses operadores existentes.

- Outra importante categoria é a de manipulação de arquivos gráficos. Essa categoria é essencial porque permite o acesso a informação referente a matriz de pixel da imagem. Essa manipulação pode ser de escrita ou apenas leitura conforme mostrado na tabela 4.4.

| Operador | Função |
|---|---|
| Affine | Altera a distância e o ângulo entre as linhas de uma imagem. |
| Rotate | Rotaciona a imagem |
| Scale | Aumenta ou diminui o tamanho da imagem. |
| Translate | Translada a imagem para um outro local do plano |
| Transpose | Combina a operação de flip com a de rotação; |

Tabela 4.3: Operadores Geométricos: manipulam a forma, tamanho e orientação da imagem.

| Operador | Função |
|---|---|
| BMP | Codifica e decodifica arquivos com extensão .BMP |
| FileLoad | Lê a imagem de um arquivo gráfico |
| FileStore | Grava uma imagem em um determinado tipo de arquivo gráfico. |
| GIF | Lê uma imagem de arquivo com extensão .GIF |
| JPEG | Abre uma imagem que está dentro de um arquivo com extensão .JPEG |
| PNG | Interpreta os arquivos com extensão .PNG |
| PNM | Lê os arquivos do padrão PNM e também seus similares PBM, PGM e PPM |
| TIFF | Lê os arquivos com extensão .TIFF |
| URL | Interpreta uma imagem especificada por um endereço web |

Tabela 4.4: Operadores de arquivo: essenciais para iniciar qualquer processamento e/ou armazenar os resultados do mesmo.

- Há outras categorias que contém muitos outros operadores como o de freqüência e quantização de cores. Assim, na tabela 4.5, ainda é mostrado outros operadores que foram muito utilizados nesse trabalho.

| Operador | Função | Categoria |
|---|---|---|
| GradientMagnitude | Realça as bordas de uma imagem baseado em duas direções ortogonais, isto é, com uma máscara vertical e depois horizontal | Extração de bordas |
| Histogram | Computa o histograma de uma imagem, isto é, a distribuição das intensidades dos pixels dessa imagem | Estatístico |

Tabela 4.5: Outros operadores muito usuais.

No entanto, faz-se necessário ainda entender como relacionar esses operadores com as imagens de entrada e saída sob a ótica da construção do código-fonte. Para isso, deve-se saber que:

1. Um objeto estático `PlanarImage` da classe `javax.media.jai.PlanarImage` carrega a matriz de pixel de determinada imagem previamente aberta ou processada. Esse objeto pode ser encontrado, por exemplo, na tabela 3.1 sob o nome de `dst`, significando

imagem de destino, isto é, aquele objeto que terá a matriz de pixel resultante de determinada operação;

2. Necessita-se de um objeto não estático denominado `ParameterBlock` provido pela classe `java.awt.image.renderable.ParameterBlock`. Esta é uma classe que não pertence ao pacote nativo da JAI, porém ela é totalmente compatível e conveniente para manipular operadores da JAI. Ela vem do pacote `awt` e é muito útil porque permite abrigar diferentes informações que formam um único bloco de dados e que, posteriormente, serão associadas com o operador utilizado. Essas informações podem ser uma matriz alusiva a uma máscara de convolução e/ou a própria imagem. Na tabela 4.1 esse objeto é representado pelo parâmetro `pb` da terceira coluna do método `JAI.create()`.

3. E por fim, faz-se necessário um objeto estático que representa o núcleo do processamento com operadores chamado `JAI` e do seu método `create` providos pela classe mãe `javax.media.jai.JAI`. Ele é o mais importante porque é nele que serão passados os parâme-tros de operação e dados de entrada providos pela classe `ParameterBlock`. A figura 3.4 ilustra a utilização precisa dessas três entidades de software, isto é, a imagem, a operação, e o bloco de informações.

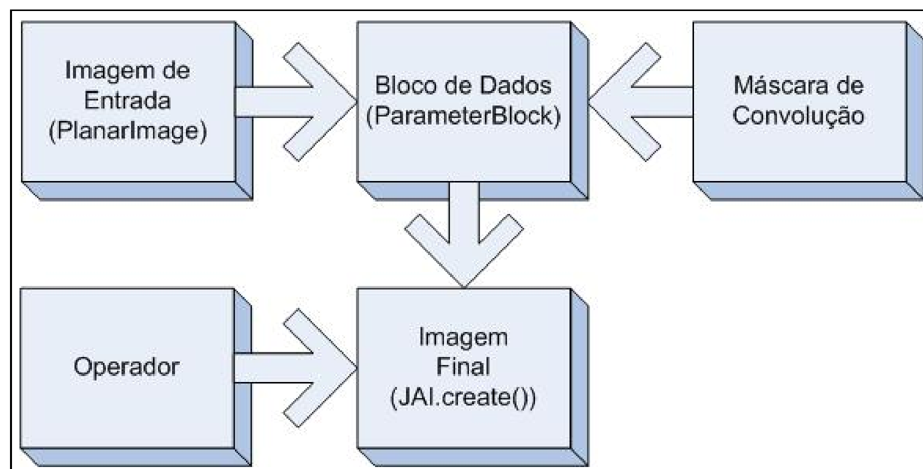


Figura 4.4: Esquema para o uso dos operadores JAI.

Através da descrição desses três passos básicos, mas, essenciais para a manipulação de operadores com a biblioteca JAI pode-se concluir também que, a construção de software em três camadas necessita de outros pacotes Java. Como especificado no item dois anterior o objeto `ParameterBlock` não pertence ao JAI mas sim, ao pacote `awt` (Abstract Window Toolkit). Isso também ocorre na implementação da última camada ou camada de interface gráfica onde utiliza-se pacotes como o `javax.swing` para fins de reuso de componentes gráficos como janelas, caixas de diálogo, botões, barras de rolagem, menus, campos de entrada de dados entre outros.

Entretanto, essa biblioteca não permite apenas utilizar essa ampla gama de operadores para construir aplicações completas, mas como também permite criar outros novos operadores. Isso

é muito positivo em se tratando de codificação pois permite encapsular a complexidade presente em um ou vários componentes e que, por sua vez, podem apresentar-se sob inúmeras classes e arquivos. Dessa forma com uma chamada apenas do objeto estático `JAI.create()`, que utiliza apenas uma linha de código, é possível resolver problemas de processamento de imagens de forma muito simples e rápida. Por exemplo, para o problema da equalização de imagens mostrado pela figura 3.1 é possível criar um operador chamado "Equalize" para resumir toda a manipulação desse processamento. Esse procedimento foi criado com êxito e sua chamada final ficou na forma `JAI.create("Equalize",pb ,null)`, onde o objeto `pb` representa o bloco de informações que carrega a matriz de pixels da imagem a ser equalizada.

## 4.2 A Camada de Aplicação

A camada de aplicação, representada pela camada intermediária da figura 4.1, consiste nos componentes de software responsáveis por resolver diversos problemas envolvendo imagens. Porém, esses componentes se utilizam das funcionalidades implementadas pelo núcleo para resolver algum problema mais amplo. Por exemplo, para binarizar uma imagem, utiliza-se operadores do núcleo para decodificar o formato do arquivo gráfico que contém a imagem, para transformá-la em tons de cinza caso seja necessário, para implementar o limiar propriamente dito e, por fim, salvar as mudanças em um novo arquivo gráfico. Os componentes de aplicação podem ser interpretados como os objetos que compilam e agregam diversas operacionalidades da camada básica representada nesse trabalho pela JAI.

Essa camada, usualmente referenciada como a camada da lógica de negócio [VÖL 2002], é a responsável por abrigar implementação de diversos algoritmos para imagens na forma de componentes de classes. Esses componentes interrelacionam-se entre si objetivando resolver sempre um problema maior a partir de pequenos artefatos de software. Essa é a fase de montagem e seu processo assemelha-se a manipulação dos blocos do kit Lego porém de uma forma muito mais difícil [CRN 2002]. Componentes de software são mais complexos que o Lego pois devem se preocupar também em como interagir com outros componentes cuja única operacionalidade conhecida é sua interface, isto é, como montar componentes que foram desenvolvidos sob aspectos distintos como uma linguagem diferente ou para um sistema ou problema específico. Com o Lego isso não ocorre pois todos suas peças são rapidamente e perfeitamente compatíveis entre si. Segundo CrnKovic e Larsson [CRN 2002], programar baseado em componentes é como ter uma banheira com diferentes kits de brinquedos como Lego, Thinkertoy, Erector, Block City e outros sendo que deve-se integrar todas as peças proveniente de cada um a fim de montar um único brinquedo. Seguindo esse raciocínio baseado na necessidade de interoperabilidade entre componentes obteve-se nesse trabalho um certo grau de facilidade uma vez que, todos os componentes foram criados utilizando a mesma linguagem, técnica e metodologia de programação. Além disso, o framework que os suporta também foi concebido nos mesmos moldes, isto é, foi arquiteturado utilizando a linguagem Java que exige um estilo de programação orientada por objetos e, que por sua vez, foi racionalmente otimizada através do uso de padrões de construção.

Isso garantiu desde o início gerenciar a complexidade do software à medida que ele ia crescendo permitindo que os padrões de desenvolvimento fossem rapidamente visualizados e incrementados a nível de implementação de soluções. Porém, se em algum estágio de desenvolvimento futuro houver a necessidade de interoperar componentes de origens distintas pode-se executar essa ação utilizando componentes de comunicação habitualmente denominados de `middlewares` como CORBA (Common Object Request Broker Architecture), Microsoft OLE (Object Linking and Embedding)e COM (Component Object Model)[BUS 1996] e [CAI 2000].

A figura 4.5 mostra o código de um componente completo que transforma uma imagem colorida para tons de cinza. Para isso utilizou-se de uma série de primitivas proporcionadas pela JAI para montar a lógica necessária para resolver esse problema específico.

**Código 1:** Método construtor do componente para converter a imagem em tons de cinza.

```
01.  public RGBtoGRAY()
02.  {
03.     //Define o nome do arquivo a ser processado
04.    String path="Angio.jpg";
05.     //Guarda a matriz de pixel da imagem no objeto chamado src
06.    PlanarImage src = JAI.create("fileload", path);
07.     //Matriz que guarda a equação de luminância
08.    double[][] matrix = {
09.                                { 0.114, 0.587, 0.299, 0 }
10.    };
12.    //Bloco de informações da imagem e da equação
13.    ParameterBlock pb = new ParameterBlock();
14.    pb.addSource(src);
15.    pb.add(matrix);
16.    //Realiza a operação de conversão
17.    PlanarImage dst = JAI.create("BandCombine", pb, null);
18.    //Grava as informações em novo arquivo gráfico
19.    JAI.create("filestore",dst,"AngioCinza.jpg");
20.  }
```

Figura 4.5: Implementação da lógica central do componente de conversão para tons de cinza.

Na figura 4.5 consegue-se visualizar na prática[7], o que foi demonstrado na figura 4.4 isto é, o esquema proposto para a utilização de operadores da biblioteca JAI. Para isso foi utilizado o mesmo esquema composto da imagem, do bloco de informações e da operação. No caso da figura 4.5 no lugar da máscara de convolução, foi utilizada a equação de luminância. Na linha 6 dessa figura obtém-se a imagem de entrada, na linha 8 é construído a equação de luminância, nas linhas 13 a 15 constrói-se o bloco de informações através do objeto `ParameterBlok`, na

[7]Um bom recurso para programação com JAI é a lista de discussão mantida oficialmente pela Sun em (`http://archives.java.sun.com/cgi-bin/wa?A0=jai-interest`).

linha 17 efetiva-se a operação de conversão para tons de cinza e por fim, na linha 19 um novo arquivo gráfico é gerado contendo a imagem recém processada. O resultado dessa operação pode ser visto na figura 4.6. Nessa figura é demonstrado o processo de angiogênese em células cancerígenas. Esse processo é caracterizado pela formação de novos vasos sangüíneos cuja função é prover a nutrição para essas células que estão em proliferação.

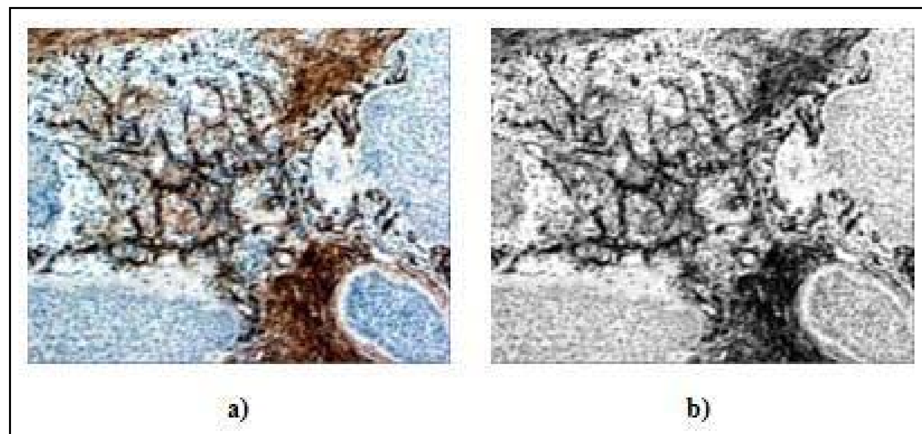


Figura 4.6: a )Imagem original colorida. b )Imagem em tons de cinza.

Muitos outros componentes foram desenvolvidos para incrementar a operacionalidade dessa camada. Por exemplo, o componente de equalização automática descrito no capítulo 3 mais especificadamente pela figura 3.2, o componente de visualização de histograma e suas informações, componentes para morfologia[8] e segmentação por watersheds[9] e componentes de interface gráfica[10] que fazem parte da camada a ser abordada na próxima seção. No entanto, cabe ressaltar que, apesar de se ter construídos componentes para certos problemas de processamento de imagens, o principal enfoque dado a esse trabalho é o da construção de um mecanismo capaz de gerenciar a complexidade provida pelo framework que irá agregar todos esses componentes. É nesse contexto que entraram os padrões arquiteturais e de projeto que serão abordados no próximo capítulo.

A figura 4.7 demonstra, ainda, outro componente desenvolvido, que a biblioteca JAI não disponibiliza e que é de essencial importância para a parte de análise de imagens, isto é, um componente para a visualização de histograma. Esse componente utiliza duas classes para resolver esse problema. A primeira manipula a entrada da imagem e a segunda tem como responsabilidade mostrar na tela o histograma recebendo como parâmetro apenas o objeto `PlanarImage` (que carrega a matriz de pixel). Dessa forma, toda a complexidade exigida principalmente pelas primitivas de exibição gráficas do java, como o objeto `Graphics2D` que irá exibir o histograma na tela, é encapsulada dentro de seu método construtor. Porém, esse componente, não demonstra apenas o gráfico da distribuição dos pixels em relação as suas intensidades mas também

[8]Construído pela acadêmica do curso de Ciência da Computação Mônica Marcuzzo.

[9]Desenvolvido pela acadêmica do curso de Ciência da Computação Grasiela Peccini.

[10]A organização dos componentes gráficos do *stoneAge* foi desenvolvido pela também acadêmica do curso de Ciência da Computação Gabrielle Dias Freitas.

informa dados mais precisos para o usuários como a intensidade de maior ocorrência, o ponto de threshold, o total de pixels analizados e outros.

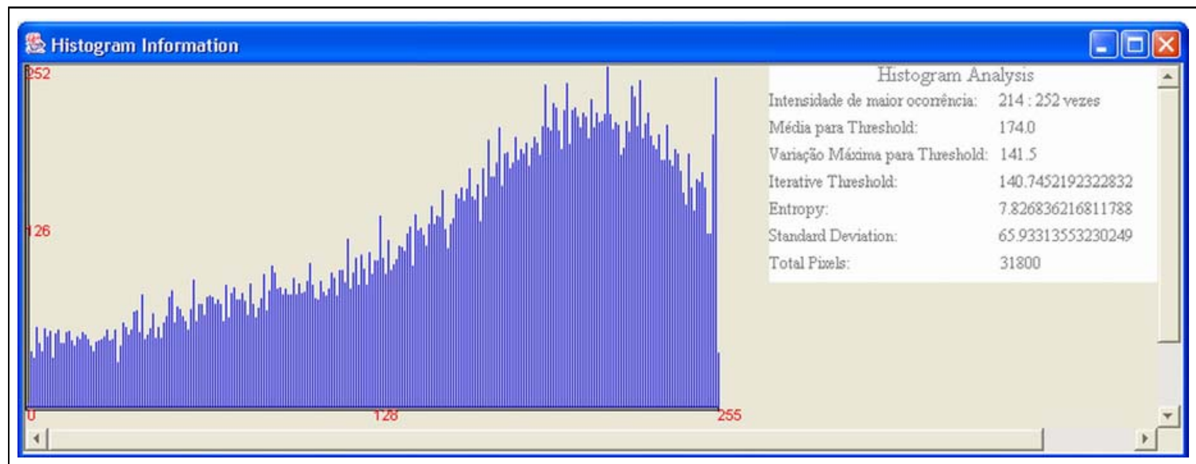


Figura 4.7: Visualização do histograma da imagem da figura 4.6 b).

## 4.3 A camada de Interface com o Usuário

A camada de interface gráfica é a parte mais importante de um sistema uma vez que, é ela que a maioria dos usuários utiliza para resolver seus problemas [GAL 2002]. Para isso, é de fundamental importância que a mesma seja funcional, agradável e acessível. Segundo ainda Wilbert Galitz [GAL 2002], um bom projeto de interface gráfica pode aumentar a produtividade de 25 a 40%.

Devido a tal importância, o projeto de interfaces está sendo atualmente visto como um campo de estudo chamado HCI (Human-Computer Interaction) cujo principal objetivo é pesquisar e desenvolver mecanismos capazes de aperfeiçoar o desenvolvimento dessa importante ferramenta. Foi o que aconteceu em 1970 quando pesquisadores da Xerox do Centro de Pesquisas de Palo Alto apresentaram o mouse com rodas como uma ferramenta alternativa para as entradas de dados provenientes do teclado. Graças a isso muita coisa mudou, uma vez que, os programas puderam ser gerenciados de outra forma ,ou seja, com janelas , caixas de diálogo, botões e outros componentes gráficos. Assim, 14 anos depois o mouse, já com uma bola, fez sucesso com o Macintosh da Apple revolucionando a interação entre o homem e a máquina para computadores domésticos [GAL 2002].

Entretanto, as pesquisas baseadas em HCI não se restringem apenas em inventar novas formas que permitam o manipulação de software através de novas entidades de hardware mas como também, de investigar como organizar a ampla gama de componentes gráficos já existentes de tal forma que o usuário aprove. Essa é a tendência atual no desenvolvimento de interfaces uma vez que, hardwares de interação com o usuário, como o mouse por exemplo, já se tornaram um padrão e desde então não sofreram enormes mudanças. Porém, encontrar um padrão para o uso de interfaces é algo complexo uma vez que elas são dinâmicas, isto é, dependem da aplicação.

Um software cujo objetivo é editar texto terá, intuitivamente, uma interface diferente de um programa que implementa soluções para o processamento de imagens digitais.

Nesse contexto, a montagem dos componentes gráficos para resolver algum problema normalmente ocorre com base na experiência que o desenvolvedor tem no que diz respeito a satisfação do usuário. Assim, para evitar demasiadas pesquisas de campo, o desenvolvimento da camada de interface presente nesse trabalho, foi embasada nos inúmeros programas já existentes para imagens como o GIMP[11], Photoshop[12] e CorelDraw[13]. Revendo a figura 4.1 pode-se visualizar também que cada componente na camada de aplicação possui a sua interface pois cada um desses componentes requer uma interação distinta do usuário e adequada à sua lógica operacional. Na figura 4.8 pode-se observar como a organização dos componentes de uma interface pode fazer a diferença no que diz respeito a funcionalidade que ela implementa. Nessa figura é apresentado, de maneira bem simples, os principais operadores da biblioteca JAI mencionados anteriormente e como a sua utilização pode ser confusa ou bem projetada para o usuário final em um exemplo hipotético de uso.

Figura 4.8: a ) Interface gráfica confusa e desorganizada. b ) Uma interface com a mesma eficácia porém organizada através do agrupamento de funcionalidades similares.

Assim, concluindo esse capítulo, foi possível perceber algumas características e requisitos exigidos pelo desenvolvimento de software na área de processamento de imagens. De posse dessas informações, o próximo passo é o de conhecer mais detalhadamente os padrões de construção. Essa etapa associada com a identificação e aplicação de alguns padrões na área de imagens, ocorre no capítulo seguinte.

[11]©2001-2004 -GNU Image Manipulation Program (http://www.gimp.org/)
[12]©2004 - Adobe Systems Incorporated (http://www.adobe.com/)
[13]©2004 - Corel Corporation (http://www.corel.com/)

# 5 PADRÕES DE CONSTRUÇÃO DE SOFTWARE PARA O PROCESSAMENTO DE IMAGENS

Nesse capítulo será abordado os padrões de projeto que, até o presente momento, apenas foram citados superficialmente nos capítulos e seções anteriores e que, por sua vez, representam o âmago dessa dissertação. Esses padrões foram utilizados tanto para a criação do framework que abrigará todos os componentes para processamento de imagens, quanto para os próprios componentes a serem continuamente agregados. Porém, sob essa óptica de implementação, os padrões aplicados no desenvolvimento desses artefatos de software, atuam em diferentes níveis de abstração. Esses níveis são o de arquitetura (alto nível), de projeto (médio nível) e idiomáticos (baixo nível).

## 5.1 Os padrões Arquiteturais

Como foi mencionado no capítulo 2, o padrão arquitetural proposto por Buschmann *et al.* [BUS 1996], cataloga formas para organizar a estrutura fundamental de um sistema, isto é, aquela parte que irá dar suporte a várias operações básicas do sistema para imagens chamado de StoneAge.

A literatura apresenta oito padrões arquiteturais: Pipes and Filters, Broker, Model-View-Controller, MicroKernel, Reflection, Blackboard, Presentation-Abstraction-Control e Layers.

O primeiro deles, isto é, Pipes and Filters que pode ser traduzido para "Tubos e Filtros" apresenta uma solução para desenvolver sistemas baseados em fluxo de dados. O próximo padrão denominado de Broker (intermediário) é muito usado na implementação de sistemas distribuídos pois propicia a interoperabilidade entre diferentes componentes de software sejam eles remotos ou locais. Exemplos do uso dessa solução são o CORBA descrito na seção de aplicação do capítulo 4, a tecnologia Microsoft Ole 2.x e navegadores internet como HotJava[1], Mosaic[2] e Netscape[3] que agem como intermediários entre os clientes e os provedores de serviço WWW [BUS 1996].

[1]©Copyright 1994-2004 Sun Microsystems, Inc. (`http://java.sun.com/products/archive/hotjava/`)

[2]©2003 Board of Trustees of the University of Illinois (`http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html`)

[3]©2004 Netscape. All Rights Reserved. (`http://www.netscape.com/`)

O padrão Model-View-Controller (Modelo/Vista/Controlador) é utilizado para construir interfaces para o usuário através da aplicação de três componentes: O Modelo contém a lógica básica utilizada na resolução do problema. A Vista exibe as informações para o usuário e o controlador manipula a entrada de dados pelo usuário. Segundo Gamma *et al.* [GAM 2000] e Buschmann *et al.* [BUS 1996] os usos conhecidos desse padrão são a construção de interfaces para a linguagem VisualWorks Smalltalk-80, e a Microsoft Fundation Class (MFC) para desenvolver aplicações para a plataforma Windows.

O próximo padrão apontado pela literatura é o Microkernel e que, de forma muito intuitiva, serve para desenvolver software que precisa estender várias aplicações a partir de um núcleo central. Essa abordagem é típica de sistemas operacionais e, dessa forma, ele tem seu uso prático na implementação do sistema operacional Mach[4], Amoeba[5], Windows NT[6], para sistemas em tempo real como o Chorus[7] e para sistemas de busca de dados como o MKDE.

O Padrão arquitetural Reflection (Reflexão) serve para sistemas que precisam mudar sua estrutura e comportamento dinamicamente. Pode também ser visto como um sistema adaptável como o MicroKernel anteriormente descrito. Buschmann cita o exemplo de uma aplicação que precisa guardar de forma persistente seus dados sendo que, para isso, é necessário armazenar e ler arbitrariamente objetos da memória principal. Para isso, faz-se necessário ter acesso dinâmico aos dados por esses objetos implementados[BUS 1996].

Já o padrão BlackBoard implementa uma solução em um contexto em que determinado problema precisa de vários programas independentes e que trabalham cooperativamente sob a mesma estrutura. Nesse caso Buschmann descreve o problema de um software que precisa realizar o reconhecimento de voz. A entrada de dados desse sistema pode ser uma simples palavra ou uma sentença inteira e a saída de dados é a representação desses sinais na forma de frases completas exibidas na tela. Para isso, deve haver um programa que segmenta a onda sonora de forma a ter sentido, outro programa ou procedimento deve checar a sintaxe da frase e outras operacionalidades. São exemplos de sistemas que usam esse padrão o HEARSAY-II que é um reconhecedor de voz de 1970, o HASP/SIAP utilizado para detectar submarinos inimigos e o TRICERO de 1984 cuja função era monitorar o tráfego de aeronaves.

O padrão Presentation-Abstraction-Control (Apresentação/Abstração/Controle) também é destinado a implementação de software interativo. Com ele pode-se implementar interfaces com o usuário (não necessariamente apenas interfaces gráficas) baseando-se no conceito de agentes colaborativos. O exemplo dado para esse padrão é um sistema para visualizar os resultados de uma eleição. O sistema oferece várias formas de visualização como gráfico de setores, gráfico de barras, tabelas e outros. Assim, a pedido do usuário, o sistema pode retornar diferentes abstrações dos resultados, isto é, uma versão do sistema pode estar mostrando apenas a tabela

[4] ©Desenvolvido pela Carnegie Mellon University (`http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html`)

[5] ©1996 Vrije Universiteit te Amsterdam, The Netherlands. All Rights Reserved.(`http://www.cs.vu.nl/pub/amoeba/`)

[6] ©2004 Microsoft Corporation. All rights reserved.(`http://www.microsoft.com/`)

[7] Desenvolvido pelo Instituto Nacional de pesquisa em Informática e em Automação da França(`http://www.inria.fr/`)

precisa dos resultados da eleição enquanto outra pode estar exibindo informações mais detalhadas sobre o número de cadeiras que determinado partido alcançou no parlamento ao final da eleição [BUS 1996].

O último padrão arquitetural é o Layers (camadas). Com ele pode-se modelar uma aplicação, intuitivamente, sob camadas onde, cada uma delas representa um domínio na aplicação. Esse padrão arquitetural foi utilizado para projetar o StoneAge. É dele que surgiu o modelo de camadas descrito no capítulo 4, isto é, a camada central que é dominada pela biblioteca para imagens JAI, a camada de aplicação que corresponde aos componentes desenvolvidos até o presente momento e a camada de interface gráfica com o usuário que permite a entrada e visualização das imagens.

### 5.1.1 Da Lama à Estrutura

A *Big Ball of Mud*[8] é o nome que caracteriza os sistemas que são estruturados por conveniência, isto é, de acordo com a experiência do programador. Não são levados em consideração as noções de projeto pois não há preocupação em uma construção capaz de permitir que outras pessoas possam facilmente entender e manter sua implementação.

Nesse contexto de desenvolvimento pode-se dizer que utilizar uma abordagem empírica no desenvolvimento de uma ferramenta que estará em contínuo crescimento é inviável. A idéia central do StoneAge é que ele se torne uma verdadeira ”Caixa de ferramentas” no que diz respeito a processamento de imagens. Assim, ele sofrerá modificações constantes seja pelos alunos de ciência da computação, da física ou até mesmo pelos professores do grupo de pesquisa.

Dessa forma, para assegurar que esse sistema cresça à medida que novas necessidades de aplicação vão surgindo no grupo de pesquisa[9] sua concepção básica não pode ser proveniente das idéias baseadas na observação que o programador tem. Para isso, a melhor forma de implementação é a baseada em padrões. Tomando como base o sentido da palavra, intuitivamente pode-se pensar que essa é uma boa idéia pois, se um padrão é a especificação de algum problema que se repete a sua utilização e conseqüente entendimento pode ser facilmente visualizado com a vantagem de possuir já uma vasta documentação para o seu uso. Assim é conseguido atribuir suporte ao processo de desenvolvimento tornando-se muito útil para os novos programadores e/ou projetistas que irão agregar novas funções de acordo com as necessidades do grupo de pesquisa.

De forma mais específica, entre os oito padrões descritos anteriormente, o padrão que mais se encaixa no contexto de desenvolvimento do grupo é o Layer. Através desse padrão o StoneAge foi dividido, como já descrito anteriormente, em três camadas que correspondem a diferentes graus de abstração. Por exemplo a primeira camada ou camada do núcleo representa o mais

[8]Título da Palestra de Joseph Yoder (University of Illinois/The Refactory, Inc., US) na Terceira Conferência Latino Americana em Linguagens de Padrões para Programação (`http://www.cin.ufpe.br/~sugarloafplop/main_pt.htm`).

[9]Com a aprovação do Edital CT-Info/MCT/CNPq 031/2004 pela coordenação do grupo PIGS o StoneAge será utilizado para dar suporte à implementação da proposta intitulada: *”Ferramentas de Processamento e Análise de Imagens para Microscopia Quantitativa Aplicada a Anatomopatologia e Caracterização Microestrutural de Materiais”*. (`http://www.cnpq.br/resultadosjulgamento/edital_312004_ctinfo.htm`).

baixo nível, enquanto que a interface gráfica com o usuário é apresentada sob o nível mais alto no que se refere a aplicação com imagens.

Buschmann *et al.* [BUS 1996], ilustra a utilização desse componente através da arquitetura do modelo OSI. O modelo OSI (Open Systems Interconnection) foi especificado para padronizar o desenvolvimento de produtos de software voltados para redes de comunicação de dados. Ele foi muito importante pois facilitou a interconexão de sistemas uma vez que, todos os protocolos desenvolvidos pelos mais diversos fabricantes deveriam seguir esse modelo na sua concepção [TOR 2001]. Graças a esse padrão de desenvolvimento, é possível construir uma rede de computadores utilizando produtos de diversos fabricantes. Esse modelo é composto por sete camadas a saber: A camada física, a camada de link de dados, a camada de rede, a de transporte, a de seção, a de apresentação e a de aplicação. A figura 5.1 demonstra essas camadas.

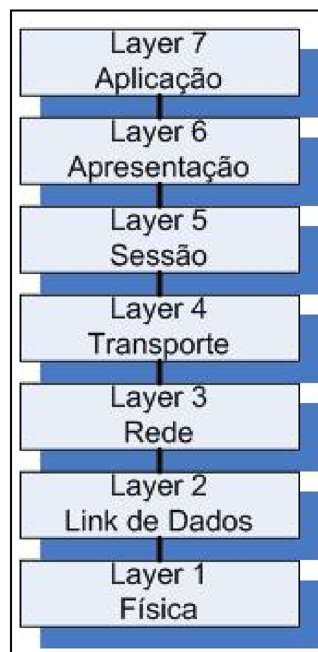


Figura 5.1: Um exemplo de aplicação do padrão arquitetural Layer: o modelo OSI.

Na transmissão de um dado, por exemplo um pacote IP (Internet Protocol), a utilização da camada OSI para tal objetivo ocorre de cima para baixo, isto é, da camada sete que é a de aplicação, até a camada física que irá codificar esse pacote em sinais compatíveis com o meio onde os dados serão transmitidos (que é o que as placas de rede fazem). Na recepção dos dados o processo é inverso, isto é, primeiro a camada física recebe o sinal e os converte para zeros e uns passando-os para a próxima camada denominada de link de dados.

Porém essa não é a única aplicação onde esse padrão é utilizado. Ele é utilizado também na implementação da JVM[10] (Java Virtual Machine). O código construído em Java quando compilado é traduzido para bytecodes que por sua vez são enviados para a JVM para serem interpretados. Esse último passo equivale a execução do programa e as fases de tradução e interpretação são realizadas sob a arquitetura de camadas. Outro exemplo muito intuitivo citado

[10] 1994-2004 Sun Microsystems, Inc. (`http://java.sun.com/`).

por Frank Buschman *et al.* são as APIs. As APIs são conjuntos de entidades de software que podem também possuir diferentes níveis de abstração. Como dito anteriormente a biblioteca JAI é uma API e como o StoneAge pode ainda de subdividir em camadas. Ela apresenta vários níveis de abstração pois provê métodos de baixo nível como os iteradores que realizam a navegação entre os pixels assim como provê também um componente gráfico que exibe a imagem na tela[11] Outro uso conhecido são os Sistemas de Informação (SI) destinados ao comércio e indústria. A camada mais baixa geralmente é representada pelo banco de dados e a mais alta pela interface gráfica e a lógica operacional do sistema. Esses dois últimos normalmente são implementados juntos caracterizando um sistema com duas camadas. Atualmente o modelo mais utilizado é o de três camadas cuja única diferença consiste na separação entre lógica e interface. Por fim outro bom exemplo é o sistema operacional voltado para redes corporativas Microsof Windows NT. Esse sistema foi implementado basicamente sob quatro camadas:

- Serviços: Camada existente entre seu microkernel (denominado de NT Executive) e os demais subsistemas;
- Gerenciamento de Recursos: nessa camada estão implementados os módulos de segurança, de escalonamento do processador, de entrada e saída (mais conhecido como I/O), gerenciamento de memória virtual entre outros;
- Núcleo: que implementa operações de baixo nível como interrupções, tratamento de exceções e outros;
- HAL: Abreviatura de Hardware Abstraction Layer, cuja função é mascarar as diferenças de hardware entre máquinas que possuem diferentes características físicas;

Assim, mediante esses casos de uso que ocorrem nas mais diferentes áreas onde os sistemas de computação estão presentes, justifica-se o uso desse padrão para o desenvolvimento do StoneAge. Ele é utilizado porque o StoneAge é um sistema grande e dessa forma, decompô-lo em camadas o tornará mais gerenciável no que diz respeito a sua complexidade. Deste modo, as vantagens da aplicação desse padrão sobre o sistema desenvolvido pode ser resumida em dois itens: o reuso de camadas e a mudança acessível de código pertencente a cada camada. A camada de aplicação, por exemplo, pode ter seus componentes reutizados para outra aplicação uma vez que cada um encontra-se fisicamente separado para fins organizacionais, de manutenção e de legibilidade. No StoneAge cada componente é encapsulado em uma estrutura chamada pacote, isto é, eles não estão espalhados como mostra a figura 2.2. Essa figura foi assim construída apenas para simplificar a idéia central do funcionamento de um framework. Nesses pacotes, que na verdade é apenas uma hierarquia de diretórios reconhecida pela linguagem Java, são compostos, usualmente, por inúmeras classes que podem ser instanciadas por outras aplicações. Instanciadas significa que as classes podem ser utilizadas em outro ambiente que não o framework chamado de StoneAge. No entanto, para fins de agilidade e simplicidade, o novo ambiente

[11]Esse componente existe mas é depreciado, isto é, teve sua manutenção extinta e possui alguns bugs.

que irá receber esses componentes precisa ser construído na mesma linguagem de origem dos componentes que, no caso, é Java. O StoneAge tem como vantagem também a fácil permutação de código porque cada layer ou camada possui seus próprios componentes, o que significa dizer que na necessidade de mudança de algum componente apenas atua-se nesse local e mais especificadamente sob o pacote que guarda esse componente. Assim, se mantidas sempre a mesma interface dessa entidade de software pode-se trocá-la por outra que implementa uma lógica mais eficiente ou que corrige "bugs" que durante a fase de testes passaram desapercebidos. Outro fato prático que surgiu no processo de desenvolvimento e que, pode ser encaixado nessa vantagem de fácil troca de camadas, diz respeito a nova versão da JAI[12] lançada após, a fase final de implementação do framework e que foi atualizada com êxito. Framework refere-se ao StoneAge e pode ser interpretado como um arcabouço de classes que irão dar suporte a um projeto reutilizável de software [GAM 2000]. Na figura 5.2 é mostrado o processo de componentização associado com a aplicação do padrão arquitetural Layer.

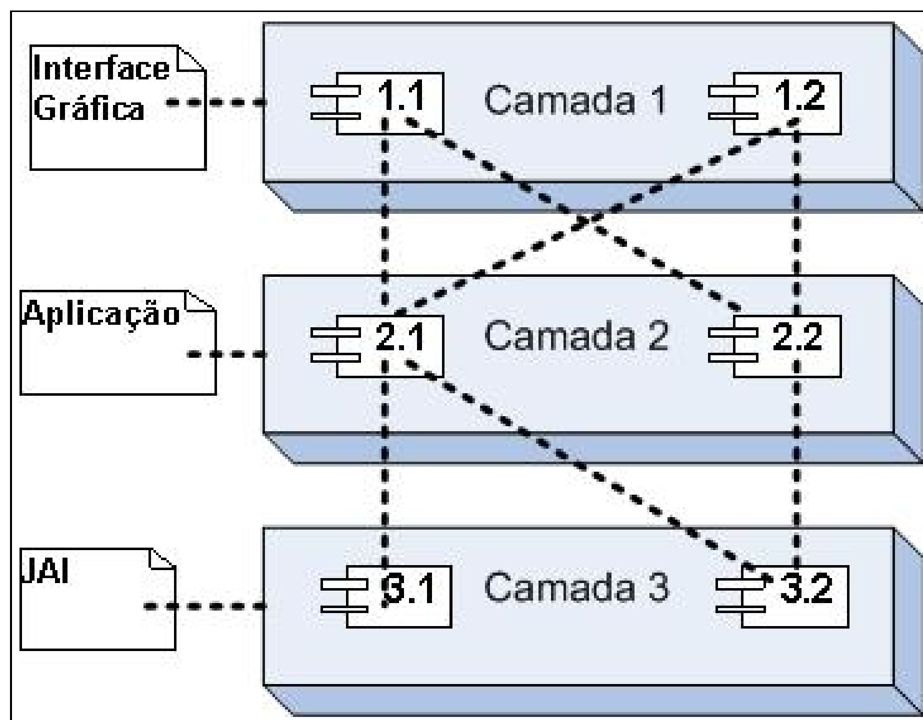


Figura 5.2: visão das Camadas de um sistema com seus componentes internos e como eles se relacionam. Adaptado de Buschmann *et al.* [BUS 1996].

Essa figura representa uma evolução da idéia inicial proposta pela figura 2.2 e uma visão de mais baixo nível se tomado como parâmetro a figura 4.1. Esse baixo nível refere-se a uma visão já de implementação, ou seja, nesse estágio o StoneAge não era mais considerado uma bola de lama, mas sim o projeto inicial do framework que oferece suporte as atividades do grupo de pesquisa no que diz respeito ao processamento de imagens. Nela é possível enxergar que o núcleo e a própria interface também são baseadas no processo de componentização. Esses

[12] A nova versão é a 1.1.2_01 de setembro de 2004. Pode ser efetuado o download livremente em (`http://java.sun.com/products/java-media/jai/`).

componentes internos já não são considerados de arquitetura, mas sim baseados na visão de projeto e idioma.

## 5.2 Padrões Gamma

Os padrões Gamma é como são chamados de maneira mais corriqueira os padrões de projeto especificados por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. A gangue dos quatro, como são mais conhecidos, escreveram um livro onde apresentam vinte e três padrões de construção para software orientado a objetos [GAM 2000].

A diferença entre esses padrões e os arquiteturais descritos na seção anterior, é que eles não especificam maneiras de construir a estrutura fundamental de um sistema. Sublinha-se que, os padrões arquiteturais como o Layer, Microkernel, Pipes and Filters e outros intencionam justamente isso, ou seja, fornecer um modelo básico para o sistema a ser implementado. Entretanto esses padrões não especificam como fazer essa implementação a nível de construção orientada a objetos ou através de outro estilo qualquer de programação. É justamente por isso que eles são denominados padrões de alto nível, isto é, não são utilizados conceitos de linguagem de programação. Porém, cabe ressaltar, que essa fase deve ser realizada antes da utilização dos demais padrões por motivos óbvios uma vez que, eles é que irão fornecer o básico para sustentar a maior parte do sistema.

Nesse contexto, os padrões de projeto possuem uma característica mais implementável, ou seja, apresentam estruturas práticas para o desenvolvimento de software orientado a objetos. Tais estruturas correspondem a especificação de classes, instâncias de classes, pacotes de software e principalmente o relacionamento que as diversas estruturas apresentam como a herança e agregação. Essas entidades de software geralmente são mostradas sob a forma de diagramas UML principalmente os diagramas de classe e de pacotes. E é justamente nesse nível que a UML surge como uma ferramenta que faz a ligação entre a modelagem do programa em alguma ferramenta visual com a parte de código.

Existem muitas ferramentas para essa fase de modelagem como o Rational Rose[13] e o UMLStudio[14] que são versões pagas. Uma boa alternativa para isso é a ferramenta UML chamada ArgoUML[15]. Porém essas ferramentas não são meramente visuais. No UMLStudio por exemplo, é possível, através do diagrama de classes, gerar código em ADA 95, C++, IDL e Java. Obviamente essa geração de código-fonte é parcial, isto é, restringe-se ao escopo das classes, métodos e atributos especificados previamente. Ele não oferece a implementação lógica de determinado componente de forma automatizada. Nesse contexto de operacionalidade essas ferramentas servem apenas para título de modelagem e conseqüente documentação visual do artefatos de software produzidos. A figura 5.3 mostra a interface do software de modelagem UMLStudio sendo utilizado para modelar/documentar o componente de visualização de his-

[13]©IBM Corporation (`http://www.ibm.com/`).
[14]©1996-2001 PragSoft Corp. (`http://www.pragsoft.com/`).
[15]Esse software é “Open Source” e pode ser baixado livremente em (`http://argouml.tigris.org/`).

tograma representado pela figura 4.7 apresentado no capítulo 4.

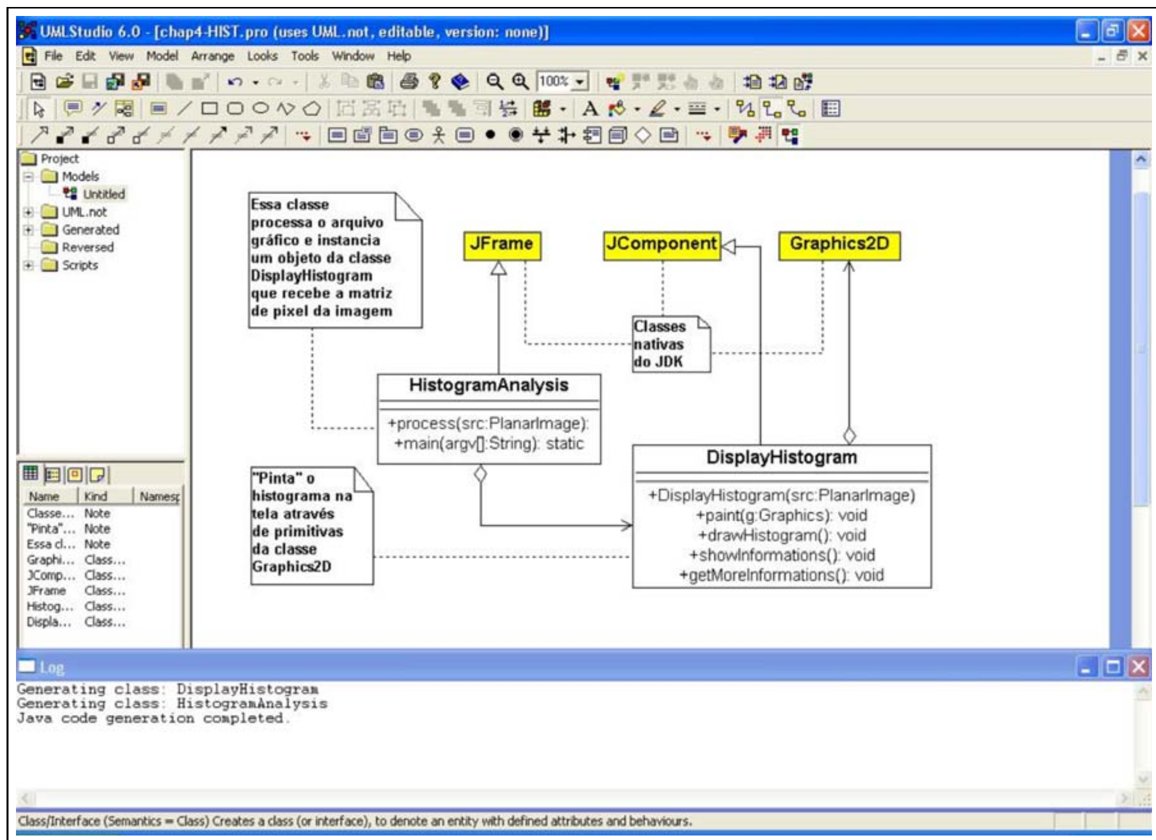


Figura 5.3: Uma ferramenta de construção de diagramas UML para a modelagem de software orientado por objetos.

A figura 5.3 demonstra a implementação de duas classes para resolver o problema da visualização de histograma que são a `HistogramAnalysis` e a `DisplayHistogram`. A primeira apenas passa o objeto de pixels para a segunda. Nesse componente é utilizado a API Java 2D para dar suporte a criação de gráficos de alta qualidade. Com ela é possível manipular até imagens porém de uma forma mais complicada e com pouco suporte no que diz respeito à operações básicas [KNU 1999].

Voltando aos padrões de projeto propriamente dito, pode-se dividi-los em três categorias conforme descrito no capítulo 2. Essa separação não é um padrão propriamente dito, isto é, ela é utilizada por Gamma *et al.* em [GAM 2000] porém Buschmann *et al.* em [BUS 1996] descreve apenas oito padrões e os coloca também em um contexto similar. Mas como os padrões Gamma são em maior número utilizou-se a primeira referência para fins de implementação desse trabalho.

Os 23 padrões descritos por Gamma são: Abstract Factory, Builder, Factory Method, Prototype, Singleton, Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy, Chain of Responsability, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor. Eles são agrupados conforme o seu propósito e isso pode ser observado na tabela 5.1.

| Padrões de Criação | Padrões Estruturais | Padrões Comportamentais |
|---|---|---|
| Factory Method | Adapter | Interpreter |
| Abstract Factory | Bridge | Template Method |
| Builder | Composite | Chain of Responsability |
| Prototype | Decorator | Command |
| Singleton | Façade | Iterator |
| | FlyWeight | Mediator |
| | Proxy | Memento |
| | | Observer |
| | | State |
| | | Strategy |
| | | Visitor |

Tabela 5.1: Os padrões de projeto de Gamma agrupados em categorias afins.

Diferente do que ocorreu com a utilização dos padrões arquiteturais onde apenas um foi identificado e utilizado para fundamentar o sistema, no desenvolvimento de um componente seja na camada que for, pode-se utilizar vários padrões de projeto. Foi o que ocorreu no processo de criação de um componente para segmentação de imagens baseado em contornos que será apresentado no capítulo 7.

Para utilizar esses padrões no processo de criação de componentes em Java primeiro deve-se entender o que cada um representa, isto é, o propósito de cada um deles. Cada um deles possui uma finalidade distinta e, dessa forma, não necessariamente são utilizados em sua plenitude para os problemas que envolvem processamento de imagens.

### 5.2.1 Padrões de Criação

- O Padrão Factory Method (Fábrica de Métodos) serve para retornar um objeto em um contexto de diversas outras classes. Segundo [COO 1998] isso pode ser facilmente realizado através da implementação de classes abstratas, onde cada classe filha dessa, isto é, as concretas, possuem métodos comuns porém com diferentes implementações. Assim, esses métodos podem ser chamados de fábricas porque são os responsáveis pela manufatura dos objetos [GAM 2000]. Uma aplicação desse padrão foi identificada durante no uso dos iteradores descritos no capítulo 4 mais precisamente quando era abordado a camada central do StoneAge, isto é, o núcleo representado pela JAI. Nesse caso, o método que concretiza a manufatura do objeto que irá permitir a navegação nos pixels da imagem é o método `create()` e sua construção na integra ocorre pelo seguinte código: `RectIter readIterator = RectIterFactory.create(input, null);`. Para utilizar o outro iterador (o RookIter) mostrado na figura 4.3 b), basta importar o pacote apropriado e trocar o código anterior por `RookIter readIterator = RookIterFactory.create(input, null);` onde `input` é o objeto PlanarImage que carrega a matriz de pixel. Ele é um exemplo de fábrica de métodos pois consegue instanciar apenas um objeto concreto por vez [Sun 1999]. Outra aplicação desse padrão foi identificada na

criação dos operadores mostrados na tabela 4.1 também através do método `create()` implementado pelo objeto `JAI` .

- O Padrão Abstract Factory[16] funciona de forma equivalente ao anterior, porém nele, é conseguido um nível maior de abstração. Em vez de retornar um objeto apenas, pode-se retornar fábricas inteiras de objetos.

  Um exemplo muito bom da utilização desse padrão pode ser visto no "*Look and Feel*" utilizado pelos componentes swing do Java para mudar o estilo de apresentação dos componentes gráficos. Há três estilos de interfaces implementadas no JDK por default : o estilo Windows, Motif(Unix) e Metal(padrão Java). Eles são suportados pelo pacote `com.sun.java.swing.plaf` [ECK 1998]. PLAF significa "*Pluggable Look-and-Feel*" e sua utilização pode ser vista na figura 5.4. Essa fábrica de objetos pode tornar-se muito importante pois permite deixar a aplicação conforme os padrões visuais existentes o que, agrada o usuário leigo. Aém disso, ela também permite misturar esses vários estilos em uma mesma aplicação e em tempo de execução. Na figura[17] 5.4 foi utilizado na barra de rolagem o padrão Java. No componente `JTabbedPane`(que dá acesso a outros painéis dentro da mesma janela) foi utilizado o padrão Unix e nas janelas de interação utilizadas para abrir o arquivo gráfico foi utilizado o padrão Windows.

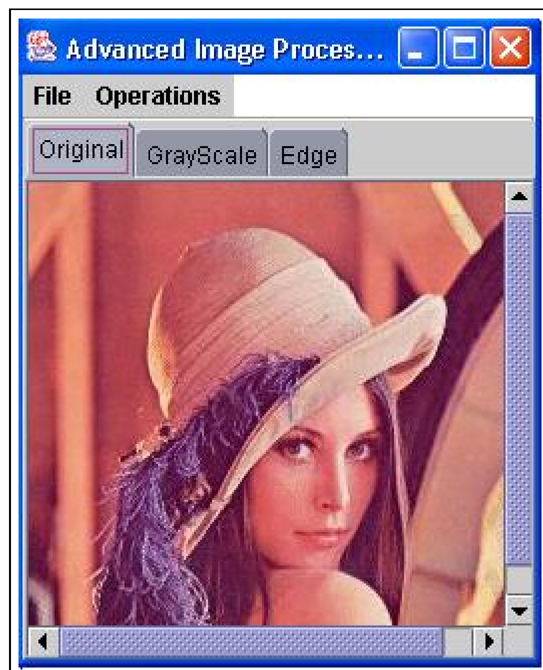


Figura 5.4: Iterface gráfica construída com a mescla dos três *Look and Feel* nativos do Java.

Sublinha-se que esse padrão é uma fábrica abstrata pois toda vez que um determinado *Look and Feel*[18] é utilizado todos os componentes gráficos presentes na aplicação auto-

[16] Também referenciado pela literatura como "Kit".
[17] Usa a imagem da lenna, que é a mais utilizada em testes de imagens. (`http://www.lenna.org`).
[18] Um tutorial on-line pode ser encontrado em `http://java.sun.com/products/jlf/ed1/dg/`.

maticamente são atualizados para esse perfil. E, sendo que, todos esses componentes gráficos possuem sua implementação em diferentes objetos há a necessidade de refletir esse efeito para todos eles. Ele é dito abstrato porque para a sua utilização não é necessário especificar suas classes concretas. No que se refere a implementação ele foi construído de forma a permitir a fácil troca entre diferentes estilos visuais através de uma pequena mudança no código. Essa operação é exibida na implementação da figura 5.5. Observe nessa figura que a utilização desses estilos visuais requer um bloco de tratamento de exceção. Nas linhas 3 e 11 pode-se comprovar que para o uso de determinado estilo basta concatenar as strings Motif, Windows ou Metal com a palavra "LookAndFeel()" que irá formar o objeto invocador.

**Código 2:** Alternando entre diferentes estilos visuais.

```
01.  //Configurando um componente para o estilo Unix
02.  try {
03.    UIManager.setLookAndFeel(new MotifLookAndFeel());
04.  } catch (UnsupportedLookAndFeelException e) {
05.      System.out.println("Problemas com o tema Unix Like");
06.    }
07.  //jtp é o componente que implementa o painel de abas
08.  SwingUtilities.updateComponentTreeUI(jtp);
09.  //Configurando um componente para o estilo Windows
10.  try {
11.    UIManager.setLookAndFeel(new WindowsLookAndFeel());
12.  } catch (UnsupportedLookAndFeelException e) {
13.      System.out.println("Problemas com o tema Windows Like");
14.    }
15.  //bar é o componente que implementa a barra de menus
16.  SwingUtilities.updateComponentTreeUI(bar);
```

Figura 5.5: Um exemplo clássico do uso do padrão Abstract Factory: mudando o estilo visual dos componentes gráficos do JDK.

- Já o padrão Builder tem a finalidade de separar um objeto de todas as partes responsáveis por sua formação. Ele é usado quando o objeto possui um grau de complexidade muito grande e para isso sua construção ocorre em etapas, daí o nome bastante intuitivo desse padrão, isto é, construção. O objeto complexo vai sendo construído por partes, sendo que cada uma dessas partes representam um algoritmo distinto mas que, por sua vez, complementam o objeto final que é chamado pela literatura de produto. A vantagem direta desse padrão é a modularidade que apresenta, pois cada uma dessas separações entre etapas e produtos finais pode ser implementado através de classes distintas. Outra vantagem citada por Erich Gamma *et al.* [GAM 2000], diz respeito ao controle que o

processo de criação possui, o que já não acontece com o padrão Abstract Factory por exemplo. Isso é bem intuitivo de ser entendido pois os padrões de criação anteriores criam os produtos de uma só vez, enquanto que o Builder implementa uma lógica mais precisa e incremental.

- O próximo padrão de criação é o Singleton. Esse padrão tem a finalidade de gerenciar o processo de criação dos objetos. Ele pode ser utilizado, por exemplo, para garantir que haja apenas uma instância de uma determinada classe. Isso pode garantir operacionalidades essenciais como a existência de apenas uma determinada aplicação na memória, um único ponto de acesso a um banco de dados ou apenas um gerenciador de impressão habilitado para uma rede de computadores locais como uma intranet por exemplo [COO 1998]. Além dessas vantagens esse padrão possibilita gerenciar as exceções que podem ocorrer durante o processo de instanciação.

- O último padrão de criação é o Prototype. Esse padrão instancia objetos clonando objetos pré-existente. Esses objetos pré-existentes que foram copiados recebem o nome de protótipos. Esse padrão é usado a todo momento, pois dificilmente alguém constrói uma aplicação do zero, isto é, sem reaproveitar uma série de classes anteriormente codificadas. Esse processo de reuso não ocorre apenas no que se refere a implementação de classes elaboradas pelo desenvolvedor em questão, mas como também, nas várias fábricas de objetos disponíveis pelos kits de desenvolvimento como o JDK (*Java Development Kit*). A utilidade desse padrão é evitar a proliferação desnecessária de código.

### 5.2.2 Padrões Estruturais

Os padrões estruturais se preocupam com a forma com que os objetos são compostos. Há sete desses padrões no catálogo Gamma (que é fiel a tabela 5.1) sendo que, sua maior finalidade, é prover modelos de estruturas de objetos capazes de suportar a interoperabilidade entre si a medida que o sistema cresce.

- O primeiro padrão é o Adapter. Esse padrão tem a finalidade de tornar a interface de uma classe compatível com outra. Isso acarreta na cooperação entre ambas, isto é, permite dois objetos trabalharem em conjunto. Segundo Cooper [COO 1998], existem duas maneiras de fazer isso: através do mecanismo de herança e de composição. No primeiro caso deriva-se uma nova classe a partir da incompatível de forma a ajustar os métodos com a mesma, isto é, que tenha uma mesma assinatura. A segunda é instanciar a classe original na nova (que precisa ser adaptada ao que já existe) e criar métodos que possam se comunicar com a classe original.

- O padrão Bridge separa a abstração da implementação de determinado objeto. Em outras palavras, ele esconde os detalhes de implementação da interface para fins de interoperabilidade entre dois objetos. Buschmann *et al.* [BUS 1996], cita como exemplo um componente distribuído que precisa funcionar em uma rede de computadores e sistemas

operacionais heterogêneos. Esse componente pode ser requisitado sob esses sistemas diferentes mas precisa garantir a comunicação independente dessa peculiaridade. Assim o padrão Bridge implementa em seu interior uma hierarquia de camadas capaz de tratar esses detalhes essenciais de funcionalidade.

- O padrão Composite é utilizado em componentes que precisam representar uma coleção de objetos. Esses objetos geralmente apresentam-se como uma estrutura de árvore para representar hierarquias do tipo todo-parte [GAM 2000]. Um bom exemplo para isso é a API de interface gráfica do JDK, que se utiliza de uma hierarquia para representar seus componentes gráficos. A figura 5.6 demonstra um exemplo de hierarquias de pacotes na qual o componente de interface gráfica `JTextField` foi implementado.

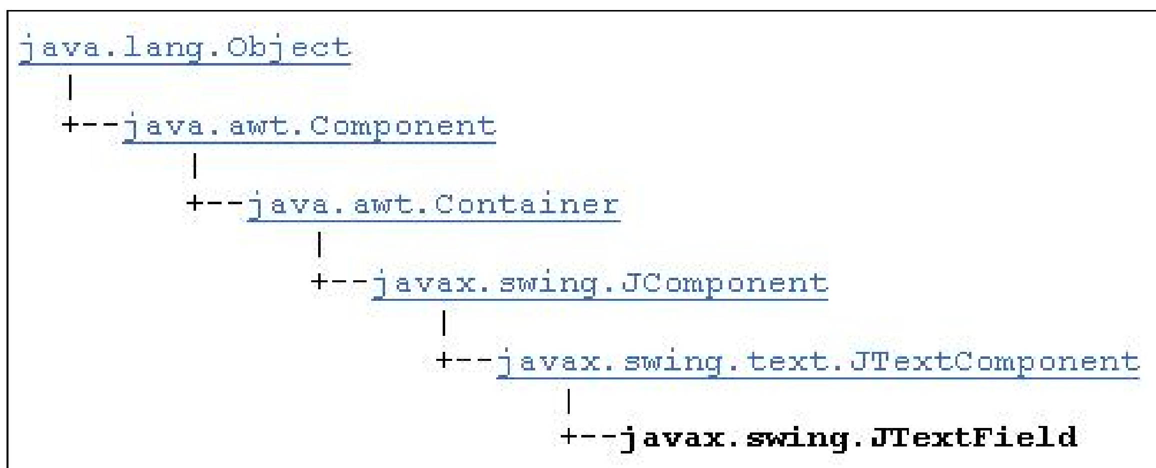


Figura 5.6: Componente gráfico construído a partir de uma hierarquia de pacotes Java. (Imagem obtida da documentação HTML do j2sdk1.4.1_02)

- O padrão Decorator tem a intenção de modificar o comportamento de um objeto sem que haja a necessidade de derivar uma nova classe a partir dessa. Para que essa extensão de funcionalidade ocorra a maneira que a literatura recomenda é anexar ou empacotar esse objeto em outro componente que implemente essas funcionalidades. É devido a esse processo que esse componente é também chamado de `Wrapper` (empacotador).

- O padrão Flyweight(peso-mosca) são usados em aplicações que exigem um grande número de pequenas instâncias de classes para resolver algum problema. Segundo James W. Cooper [COO 1998], ele pode ser utilizado em um editor de texto que trata cada caracter ou cada ícone utilizados em uma tela como um objeto. Porém isso pode gerar dezenas ou até centenas de milhares de objetos acarretando em um uso exagerado de memória. Para resolver isso, basta reconhecer as instâncias que são similares e transformá-las em métodos (operações) reduzindo assim o número de objetos na memória.

- O Proxy é um padrão que implementa o controle de acesso a um objeto para fins de usá-lo sob demanda, isto é apenas quando for necessário. Gamma *et al.* [GAM 2000],

descreve um editor que precisa carregar uma grande imagem e exibir na tela. Para evitar que isso ocorra quando há outras tarefas sendo executadas e que exigem um certo grau de processamento, essa imagem pode ser ocultada, mantendo-se apenas uma referência para a mesma.

- O último padrão é o Façade. Esse padrão provê um objeto que unifica todas as interfaces dos objetos pertencentes a um componente. Ele funciona como uma fachada que esconde as particularidades das assinaturas dos objetos de um componente tornando-o mais fácil de ser utilizado. Na figura 5.7 pode-se ver facilmente como esse processo atua no que diz respeito as classes que pertencem a um pacote pré-existente e as classes novas situadas acima desse componente.

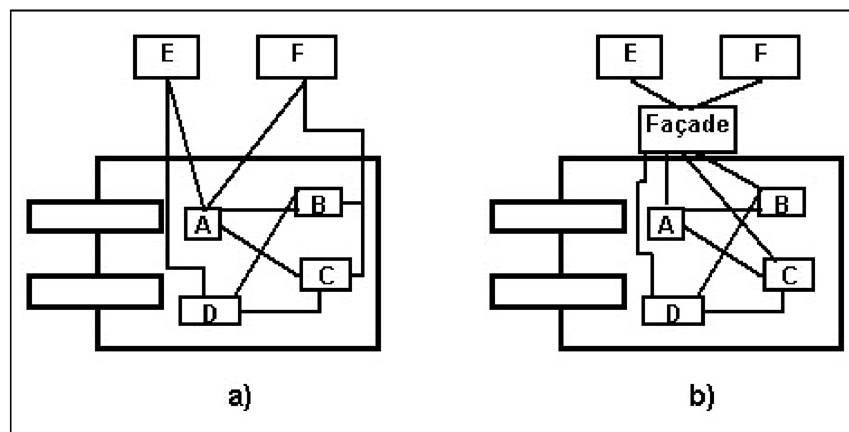


Figura 5.7: O padrão de projeto Façade atuando para tornar um componente mais flexível. **a)** mostra a maneira confusa com que as novas classes E e F se comunicam com as classes do componente. **b)** a aplicação do padrão Façade para fazer o meio campo entre as interfaces dos componentes e das classes novas E e F.

### 5.2.3 Padrões Comportamentais

A preocupação principal desses padrões são os algoritmos e a atribuição de responsabilidades entre objetos [GAM 2000]. Entretanto, para implementar essa idéia ele também especifica estruturas de construção para os objetos. Essa categoria é representada por onze padrões. A seguir é apresentado uma breve descrição de cada um.

- O padrão Chain of Responsibility resolve determinado problema através da atribuição de responsabilidades a uma cadeia de objetos. Nessa cadeia um objeto não está apto a conhecer o comportamento do outro sendo que entre eles apenas há uma comunicação do tipo requisição. Segundo Cooper [COO 1998], esse padrão é utilizado para implementar sistemas de Help onde cada problema a ser pesquisado pelo usuário (que é a requisição) passa por diferentes objetos nessa cadeia até que o usuário consiga ou não a informação desejada.

- O padrão Command é similar ao anterior, isto é, ele atende a um determinado pedido. A diferença é que ele não realiza essa requisição para uma cadeia mas sim, para um objeto específico. Esse padrão foi utilizado na construção do StoneAge e sua descrição mais detalhada será feita no capítulo 6.

- O padrão Interpreter tem a finalidade de interpretar a gramática de uma determinada linguagem (como um projeto de compilador) ou para analisar, por exemplo, palavras de dados referentes a nomes de pessoas em um sistema. Um bom exemplo é descrito por Cooper, [COO 1998] através do software Mathematica[19] onde através de uma equação pode-se visualizar o seu gráfico como uma curva por exemplo. Para isso deve-se criar uma hierarquia de classes capaz de atender as expressões regulares proposta por esse sistema.

- O Iterator tem a intenção de proporcionar o acesso seqüencial aos elementos de um objeto. É o que ocorre com os iteradores implementados pelo JAI. O objeto que implementa o iterador na JAI acessa na ordem anteriormente descrita os pixels de uma imagem que estão encapsulados no objeto `PlanarImage` . A principal característica dele é que ele realiza essa operação de forma alto nível, isto é, não é preciso reconhecer os detalhes internos de sua implementação.

- O Mediator é utilizado para prover um fraco acoplamento entre objetos que precisam mutuamente conhecer seus métodos ou atributos. Quando há muitos objetos envolvidos essa referência entre objetos pode deixar o sistema muito confuso e, portanto, de difícil manutenção. Para resolver isso, cria-se uma classe que irá gerenciar todas essas referências, isto é, um objeto que conhecerá os detalhes de implementação de todos os outros [COO 1998]. Assim quando um objeto precisa se comunicar com outro ele o faz via objeto mediador.

- O padrão Memento (recordação) guarda o estado de um objeto a fim de recuperá-lo posteriormente. É muito utilizado para prover a operação `Undo` [MET 2002]. Essa operação é muito útil em qualquer sistema pois permite recuperar o objeto original de entrada sem que haja a necessidade de executar sua abertura novamente e realizar determinado processamento redundante. Para isso cria-se um nova classe que irá quardar o último estado de determinado objeto.

- O padrão Observer é usado para separar um problema em dois objetos: um objeto que cuida dos dados que serão utilizados e outro que mostra esses dados de alguma forma, isto é, seja por interface ou modo terminal mesmo. Assim esse objeto de exibição precisa sempre observar qualquer mudança nos atributos do objeto de dados para fins de atualização dos dados que podem ter sofrido mudanças. Ele é utilizado na implementação do padrão arquitetura MVC (Model-View-Controller) para gerenciar qualquer atualização

[19] ©2004 Wolfram Research, Inc. (`http://www.wolfram.com/`).

sofrida pelo objeto Model(que representa o núcleo dos dados utilizados) e que irá refletir no objeto View (que mostra na tela). Dessa forma sua utilização garante a fidelidade de visualização da informação.

- O padrão State é utilizada para implementar objetos que dependem de uma determinada situação para manifestar seu comportamento. Por exemplo, um objeto que representa uma conexão de rede pode apresentar os estados "estabelecido", "fechado" ou "escutando". Assim, um pedido de comunicação solicitado por algum outro objeto vai ser atendido dependendo do status do objeto que implementa a conexão com a rede [GAM 2000].

- O Strategy é um padrão que encapsula os detalhes de escolha de determinado algoritmo em um contexto onde podem existir famílias inteiras de algoritmos.

- O padrão Template Method também foi utilizado nesse trabalho e sua finalidade é prover uma lógica comum para todas as subclasses de um componente. Para isso, a classe mãe implementa um método (que por sua vez encapsula um algoritmo) que é utilizado por todas as sub-classes evitando assim sua replicação em cada uma delas.

- O último padrão é o Visitor. Oferece uma estrutura para ser utilizada quando uma classe concreta invoca um método específico implementado por outra classe. Para isso deve ser tratado as questões de visibilidade para permitir tal comunicação. Ele é útil pois permite definir uma nova operação em outro objeto sem mudar as classes que já existem evitando assim, a recompilação de código.[GAM 2000]. Esse objeto separado é o visitante daí o nome desse padrão.

Dentre todos esses padrões, que são os mais conhecidos da literatura, nem todos podem se encaixar em problemas de processamento de imagens. Alguns deles foram usados para a implementação desse trabalho, porém, isso não significa que a medida que o sistema cresce outros padrões sejam reconhecidos e utilizados.

## 5.3 Padrões Idiomáticos

Como dito no capítulo 2, essa é outra maneira de construir software orientado por objetos. Porém nessa categoria não se cria classes e relacionamentos complexos como ocorre com os anteriores. Mesmo assim, ela é citada como baixo nível porque está intimamente ligada a linguagem de programação utilizada para desenvolver os componentes ou até mesmo o próprio framework base.

A idéia central desses padrões é utilizar recursos oferecidos pela linguagem usada na resolução de algum problema. Por exemplo, a linguagem C++ oferece a possibilidade de programar de forma genérica através dos gabaritos STL. Java já implementa uma espécie de padrão Singleton através da palavra chave Static. Esse último não tem exatamente a mesma finalidade do padrão propriamente dito, mas ajuda a sua implementação. Java apresenta mecanismos para guardar

objetos para sua utilização em momentos futuros, substituindo assim a necessidade do padrão de projeto Memento. Esse último diz respeito a implementação do objeto Vector, que permite guardar objetos como se fossem dados primitivos em um vetor simples.

Porém, Buschmann *et al.* [BUS 1996], descreve que o simples ato de manter uma certa característica quando se desenvolve código, está se utilizando um padrão idiomático também. Isto é, nesse caso não significa necessariamente utilizar vantagens oferecidas pelas linguagens de programação, mas sim através da manipulação uniforme de código simples. Por exemplo, se um programador em C faz o possível para utilizar apontadores de memória para implementar alguma solução, mesmo sabendo que existe a possibilidade de outras formas ele está programando com padrões. A idéia é não misturar as formas e procurar propagar um padrão para todo o código-fonte. No que se refere a esse trabalho, isso de certa forma foi utilizado pois, mesmo que alguns componentes tenham sido manipulados ou codificados por pessoas diferentes manteve-se um padrão de codificação no que diz respeito as variáveis, as classes entre outros. A figura mostra essa idéia a partir da codificação de duas funções em C que copiam um arranjo de caracteres para outro.

**Código 3:** Diferentes formas de uma mesma solução.

```
01.  //função para replicar uma Strings usando ponteiro
02.  void copyWithPointer(char *d, const char *s){
03.       while (*d++=*s++);
04.  }
05.  //a mesma função porém sem ponteiros
06.  void copyWithoutPointer(char d[], const char s[]){
07.     int i;
08.     for(i=0;s[i]!='\0';i++)
09.     {
10.         d[i]=s[i];
11.      }
12.     d[i]='\0';
13.  }
```

Figura 5.8: Duas funções na linguagem C que expressam a mesma intenção. Adaptação de Buschmann *et al.* [BUS 1996].

A função da linha dois utiliza ponteiros, enquanto que a da linha seis usa um arranjo propriamente dito. Para formar um padrão idiomático deve-se escolher uma dessas formas de implementação. Assim, pessoas de diferentes graus de entendimento em programação acostumam-se mais rapidamente a um código estranho.

O grande problema que pode vir a ocorrer quando da utilização desses padrões refere-se a portabilidade de código. É fácil entender isso, pois se um componente é escrito utilizando recursos STL da linguagem C++ e há a necessidade de traduzi-lo para Java, possivelmente ele terá de

ser reescrito no que se refere aos seus detalhes internos uma vez que Java não tem a mesma abordagem genérica que os gabaritos C++. A abordagem genérica[20] do Java a partir da versão J2SE 1.5 (*The Java 2 Platform, Standard Edition*) foi projetada apenas para eliminar a necessidade de `casts` que são freqüentemente utilizados no retorno de algum tipo de dado armazenado em alguma coleção. Já, C++ provê mecanismos mais sofisticados para tratar diferentes tipos dados. Esses mecanismos permitem definir um tipo sem especificar todos os outros tipos implementados pela linguagem. Essa abordagem foi descrita no capítulo 2 e suas implicações fogem do contexto dessa dissertação.

Terminada a descrição dos padrões de construção existentes e, das particularidades que envolvem o desenvolvimento de software no domínio de imagens, o próximo capítulo mostra a aplicação desses conceitos na construção do *StoneAge* propriamente dito.

[20]Maiores informações podem ser obtidas em: (`http://java.sun.com/developer/technicalArticles/J2SE/generics/`)

# 6 STONEAGE: UMA FERRAMENTA JAVA BASEADA EM COMPONENTES PARA O PROCESSAMENTO DE IMAGENS

Nesse capítulo é apresentado o framework desenvolvido para controlar os vários componentes de software implementados. Esse framework é a arquitetura base do sistema e foi concebido através da aplicação de determinados padrões especificados no capítulo 5. Para tal propósito, primeiramente será abordado a linguagem de programação utilizada que, no caso, é Java. Posteriormente segue-se uma explicação mais detalhada dos padrões utilizados no processo de codificação. Paralelo a esse último é descrito a operacionalidade do sistema.

## 6.1 A Linguagem de Programação

Segundo Watt e Findlay [WAT 2004], vários critérios devem ser utilizados para escolher uma linguagem de programação. Entre eles: modularidade, reusabilidade, portabilidade, nível de abstração, confiabilidade, eficiência, legibilidade, disponibilidade de ferramentas e outros. Sob essa perspectiva foi identificado no processo de desenvolvimento do StoneAge as seguintes características da linguagem escolhida Java:

- Java é uma linguagem modular pois permite a construção de unidades de compilação o que é muito útil para organizar grandes sistemas. Cita-se aqui, o uso dos pacotes Java que permitem a organização hierárquica das classes (veja figura 5.6) implementando assim um link dinâmico de compilação entre diferentes partes de um sistema. Esses pacotes são estruturas de diretórios utilizados para organizar classes e interfaces [DEI 2001a]. Assim, todos os componentes desenvolvidos nesse trabalho estão inseridos dentro de pacotes. Dessa forma, há apenas duas classes principais que são as responsáveis por gerenciar as chamadas a esses pacotes. Nesse trabalho, elas receberam intuitivamente o nome de `Main` (unidade principal do sistema base) e `LitleWindow` (unidade principal da interface com o usuário). Isso é muito agradável para os futuros desenvolvedores pois não se deparam com uma enxurrada de arquivos relativos as classes do sistema em um único diretório.
- Java também permite o reuso não só de classes, mas como também de pacotes inteiros. Esse aspecto é muito explorado nesse trabalho. Todas as classes que já ofereciam uma determinada solução foram efetivamente utilizadas. Isso ocorreu na camada de aplicação

onde se reutilizava classes do núcleo para resolver um problema maior, quanto para construir a interface com o usuário onde foi utilizado os pacotes ofertados pelo kit de desenvolvimento Java.

- A linguagem Java também é portável. Inclusive essa é umas das principais características dela. Os programas escritos nessa linguagem podem ser movidos de plataforma operacional sem que haja qualquer modificação em seu código fonte. Isso ocorre porque ela é traduzida por uma máquina virtual que é responsável por abstrair as heterogeneidades implementadas pelas diferentes arquiteturas dos sistemas operacionais modernos. Nesse trabalho foi utilizado como plataforma de trabalho o sistema operacional Windows XP[1] porém, o sistema desenvolvido pode ser utilizado, a princípio, sob qualquer outra plataforma cuja máquina virtual é devidamente disponibilizada pelo fabricante que, no caso, é a Sun Microsystems.

- No que se refere a nível de abstração Java apresenta-se sob a forma de alto nível. Por exemplo, em Java, o programador não utiliza apontadores de memória. Entretanto, todos os objetos são acessados através de ponteiros, porém isso está escondido na semântica da linguagem [WAT 2004].

- Pelo fato do programador não manipular apontadores de memória explicitamente já pode ser considerada uma linguagem com um certo grau maior de confiabilidade. Ela também possui checagem de erros em tempo de compilação e execução. Isto é, o compilador acusa o erro que inviabilizou o sucesso na compilação e também acusa de forma bem alto nível possíveis erros em tempo de execução.

- O Java, por ser interpretado, possui uma eficiência menor se comparado com C ou C++. Dessa forma, para algumas aplicações científicas que requerem processamento intensivo ele pode a não vir a ser adequado. Isso é relativo, pois atualmente já estão utilizando Java para programação paralela, isto é, voltada para alto desempenho. Dessa forma, pode-se utilizar Java para executar aplicações em aglomerados de computadores através da tecnologia como a de passagem de mensagens. Uma biblioteca Java que implementa essa tecnologia é a JOPI [MOH 2002]. É normal muitos desenvolvedores utilizarem o critério eficiência para refutar a utilização de Java. Porém esquecem de analisar as outras vantagens que ela oferece como o desenvolvimento rápido de interfaces gráficas(que pode ser essencial para sistemas de imagens), suporte para programação web de alto nível, sua portabilidade operacional entre outros.

- Legibilidade de código é algo que depende muito do programador. Porém, como Java é uma linguagem alto nível o seu código possui uma tendência a se apresentar legível e, dessa forma, torna-se um dos pré-requisitos para a fácil manutenção de código.

[1] ©2004 Microsoft Corporation. All rights reserved.(`http://www.microsoft.com/`)

- Esse último item diz respeito as ferramentas de auxílio ao desenvolvimento Java atualmente existentes. É uma vasta quantidade de ferramentas que possibilitam a construção de código. Entre todas cito as IDEs (Integrated Development Environment) e as ferramentas de documentação de código. As IDEs são ambientes de programação onde é possível configurar um determinado kit de desenvolvimento Java objetivando receber uma série de auxílios como: obter informações sobre os métodos e atributos de determinada classe, conhecer a hierarquia que um objeto possa ter afim de integrá-lo com outro, obter informações sobre erros de compilação ou execução na forma de janelas gráficas entre outros. Nesse trabalho foram utilizados as IDEs Kawa[2] e IntelliJ[3]. Já a ferramenta de documentação[4] é originária do próprio JDK e permite a documentação interna de um código. Assim, por exemplo, é possível descrever qual a intenção de um método ou classe e como ele foi implementado. A partir daí, é possível gerar a documentação em formato HTML que pode, por sua vez, ser automaticamente disponibilizado na internet. Esse processo pode ser visualizado na figura 6.1.

**Código 4:** Documentando o método construtor da classe que efetua a equalização de uma imagem.

```
01.  /**
02.  * Método construtor.  Em seu interior é executado toda a
03.  * operação de equalização.
04.  * @param src O objeto PlanarImage da imagem a ser equalizada.
05.  * @return O objeto representativo da imagem equalizada.
06.  */
07.  public static PlanarImage EqualizeImage(PlanarImage src)
08.  {
09.  ...
10.  }
```

Figura 6.1: Usando os comentários de documentação para explicar um componente.

Para a geração dessa documentação devem ser utilizados comentários especiais que serão ignorados pelo compilador, mas utilizados pela ferramenta Javadoc. Eles recebem o nome de comentários de documentação e na figura 6.1 podem ser vistos da linha 1 até 6. Eles iniciam com `/**` e terminam com `*/` . As tags Javadoc `@param` e `@return` descrevem respectivamente um parâmetro para o método e seu retorno.

[2] ©2000 Allaire Corporation. All rights reserved. (http://www.allaire.com/). Essa IDE foi descontinuada, porém pela sua simplicidade e funcioalidade pode ser vista como uma boa opção.

[3] ©2000 - 2002 JetBrains, Inc. All rights reserved. (http://www.intellij.com/).

[4] ©1994-2004 Sun Microsystems, Inc. (http://java.sun.com/products/jdk/javadoc/index.html).

Assim, mesmo conhecendo alguns dos critérios de avaliação de uma linguagem o contexto em que ela será aplicada pode pesar muito. Por exemplo, do que adianta chegar a conclusão de que Java é a linguagem que mais se ajusta ao problema se no entanto, nenhum dos desenvolvedores possui a experiência necessária para explorá-la ? O efeito disso pode ser muito negativo porque o código dificilmente será legível e provavelmente os programadores não conseguirão seguir um estilo fiel no que diz respeito a orientação por objetos, pois muitos acabam utilizando uma visão estruturada. Além disso, se o fator tempo for importante, isto é, não há muito tempo para obter os conhecimentos necessários sobre a linguagem pois faz-se necessário uma resposta rápida em nível de programação, dificilmente ela será utilizada. Assim, a escolha de uma linguagem depende do contexto.

## 6.2 A Aplicação dos Padrões no StoneAge

Na implementação do framework base foram utilizados quatro padrões de projeto: o Command, um idiomático, o Factory Method e o Strategy.

A idéia do sistema base, como foi descrito anteriormente, é a de suportar os inúmeros componentes que nele serão agregados. Assim, a medida que o sistema vai crescendo, mais e mais chamadas a esses componentes deverão ser asseguradas por esse framework. Porém, o sistema pode se tornar muito complexo pois cada uma dessas chamadas naturalmente serão executadas via interface gráfica formando assim hierarquias de menus que irão fazer a interação entre o usuário e os detalhes de implementação de alguma solução. Dessa forma, para gerenciar essa complexidade de requisições por parte do usuário foi utilizado o padrão de projeto Command.

Esse padrão pode ser melhor visualizado pelo modelo estrutural exibido na figura 6.2.

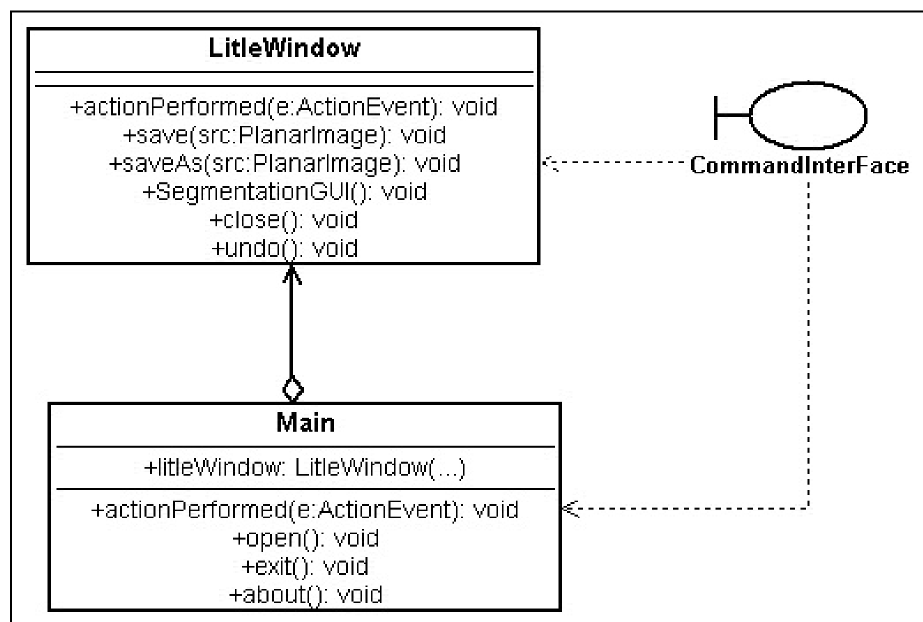


Figura 6.2: As três classes básicas do framework StoneAge.

Na figura 6.2 pode-se ver o esqueleto do funcionamento do padrão Command. A classe `Main` representa a classe "mãe" onde é efetuado a abertura dos arquivos gráficos, as informações de ajuda do sistema e outras. Toda vez que um arquivo gráfico é aberto essa classe invoca um objeto da classe `LitleWindow` que tem por finalidade exibir essa imagem na tela e proporcionar todos as operações até então desenvolvidas. Assim, a classe `LitleWindow` comporta-se como um processo filho capaz de apresentar diferentes comportamentos. E é justamente essa flexibilidade que levou a essa modelagem, isto é, a partir de um objeto mãe é possível ter vários objetos filhos que representam as várias imagens que foram carregadas e que, podem apresentar diferentes formas de processamento. Por exemplo,em um objeto filho pode-se realizar o histograma da imagem enquanto que em outro é realizado a operação de binarização.

Na figura 6.2 essa relação entre as duas classes é realizada através do mecanismo de agregação. Essa forma de relacionamento se dá, segundo as normas da UML, por uma linha onde uma das extremidades possui uma seta que aponta para a classe que está sendo instanciada, e na outra por um diamante que representa a classe agregadora. Porém, essa ainda não é a idéia central do padrão Command. O seu núcleo de operação é implementado pela interface chamada, nesse trabalho, de `CommandInterface`.

Quando um sistema cresce muitas opções de funcionamento podem ser requisitadas via interface gráfica. Em Java, normalmente esse processo é inteiramente gerenciado pelo método `actionPerformed` e do objeto `ActionEvent`. Esse método controla as requisições feitas pelo usuário como, por exemplo, um click em um botão que será seguido pela chamada de algum outro método. Veja na figura 6.3 como isso ocorre.

**Código 5:** Controlando requisições do usuário via Interface Gráfica.

```
01.  public void actionPerformed(ActionEvent w) {
02.     String opcao = w.getActionCommand();
03.     if (opcao.equals("open")){
04.       open();
05.     }
06.     if (opcao.equals("close")){
07.       close();
08.     }
09.     ...
10.  }
```

Figura 6.3: Como é feito o controle das chamadas dos métodos por componentes gráficos.

A palavra `open` e `close` da linha 3 e 6 representam o identificador de determinados componentes gráficos especificados pelo programador. Entretanto, infelizmente, em um sistema que apresenta muitas operações esse método ficaria sobrecarregado além do que, a mesma classe onde esse método é implementado possivelmente também abrigará todos os métodos que im-

plementam as mais distintas operações. Essa é uma forma possível e funcional, porém nada legível e portanto, de difícil manutenção.

Para resolver este problema, é utilizado uma abordagem mais modular. Cada componente gráfico, que requisita alguma operação, é implementado como uma nova classe. Cada uma dessas classes herda as características de uma interface publica chamada `CommandInterFace` que é o objeto Command propriamente dito. A figura 6.4 expõe de maneira mais clara essa solução.

**Código 6:** Manipulando requisições do usuário de forma mais elegante.

```
01.  //especificação do objeto Command em um arquivo
02.  public interface CommandInterFace {
03.     public void ExecuteAction();
04.  }
05.  //invocando o objeto Command na Main
06.  public void actionPerformed(ActionEvent e){
07.     CommandInterFace obj = (CommandInterFace)e.getSource();
08.     obj.ExecuteAction();
09.  }
10.  //Classe alusiva a um componente gráfico interno a classe Main
11.  class ListenOpenFile extends JMenuItem
12.                                  implements CommandInterFace{
13.      public void ExecuteAction(){
14.           open();
15.     }
16.  }
17.  ...
```

Figura 6.4: Uma alternativa de controle mais legível.

Na linha 5 da figura 6.4 pode-se ver que o método `CommandInterFace` ficou reduzido a quatro linhas. Não só nesse caso ele sofreu essa redução, isto é, ele sempre manterá essa característica constante. Através da especificação da interface Command todas as classes criadas como a da linha 11 que tem a intenção de prover um item gráfico para abrir uma imagem de arquivo, terão em seu interior o mesmo nome do método presente em `CommandInterFace`, isto é, `ExecuteAction()` porém com diferentes implementações. Com isso é possível desacoplar o objeto que invoca a operação daquele que tem o conhecimento para executá-la [GAM 2000].

A principal desvantagem desse padrão é a proliferação de pequenas classes [COO 1998]. Porém, essa desvantagem não representa um risco para o sistema, pelo simples fato de que o aspecto organizacional alcançado pelo padrão Command é muito grande. Qualquer alteração em algum componente gráfico pode ser facilmente realizada pois basta encontrar a sua classe e efetuar as mudanças necessárias sem mexer no resto do código. Por exemplo, se a classe da linha 11 necessitar mudar sua finalidade bastaria entrar dentro do seu método `ExecuteAction()` e al-

terar a chamada do método `open()` para outro qualquer. Para fins de simplificação dentro dessas pequenas classes foram invocados métodos locais. Entretanto, na maioria dos casos, acontece o instanciamento dos objetos principais dos componentes que estão alocados em unidades do tipo pacote. Com a intenção de deixar essa idéia clara, a figura 6.5 mostra como esse processo se desenvolve.

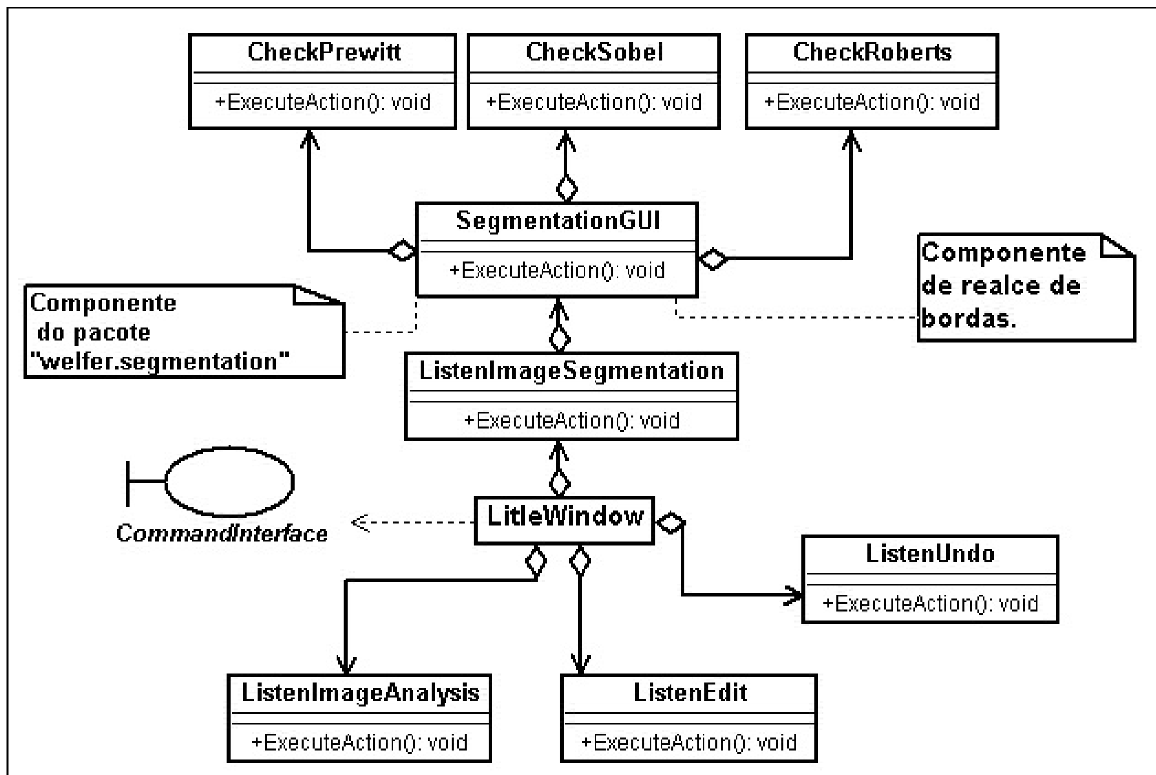


Figura 6.5: Uma visão mais completa da estruturação do sistema base segundo o padrão de projeto de controle denominado de Command.

Na figura 6.5 todas as classes agregadas por `LitleWindow` são componentes gráficos. Nessa figura, para fins de simplificação, são mostrados alguns desses componentes como o `ListenImageSegmentation` (que é um item de menu que controla o processo de segmentação), `ListenImageAnalysis` (é um item de menu global que executa a análise de imagem), `ListenEdit` (item de menu global de edição) e , `ListenUndo` (item de menu que possibilita voltar ao último estado da imagem processada) . Esses componentes, por sua vez, agregam os componentes que estão organizados em pacotes distintos como anteriormente descrito. Na figura 6.5 isso ocorre com o componente de realce de bordas representado pela sua classe principal chamada de `SegmentationGUI` que é instanciada quando o usuário invoca o item de menu gráfico representado pela classe `ListenImageSegmentation`. Assim, cria-se uma espécie de metodologia de desenvolvimento onde, os programadores habituam-se facilmente com o código do sistema possibilitando que o mesmo cresça a partir do desenvolvimento colaborativo. Isso é muito útil em um grupo de pesquisa que, normalmente, possui como característica o dinamismo de desenvolvimento, isto é, o quadro de programadores de tempos em tempos se renova.

Sublinha-se que a criação de ambas as classes `Main` e `LitleWindow` não é uma especificação do padrão de projeto Command. Na verdade ambas se utilizam desse padrão entretanto, sua implementação foi assim construída para gerenciar melhor as imagens que são utilizadas. Para isso a classe mãe apenas gerencia a entrada de dados e outras especificações do sistema enquanto que a classe filha (`LitleWindow`) trata toda nova imagem como um objeto distinto. O processamento realizado em uma imagem é autônomo, não ocorrendo problemas quando várias imagens são abertas. A figura 6.6 mostra esse processo em ação através da primeira interface construída para o StoneAge.

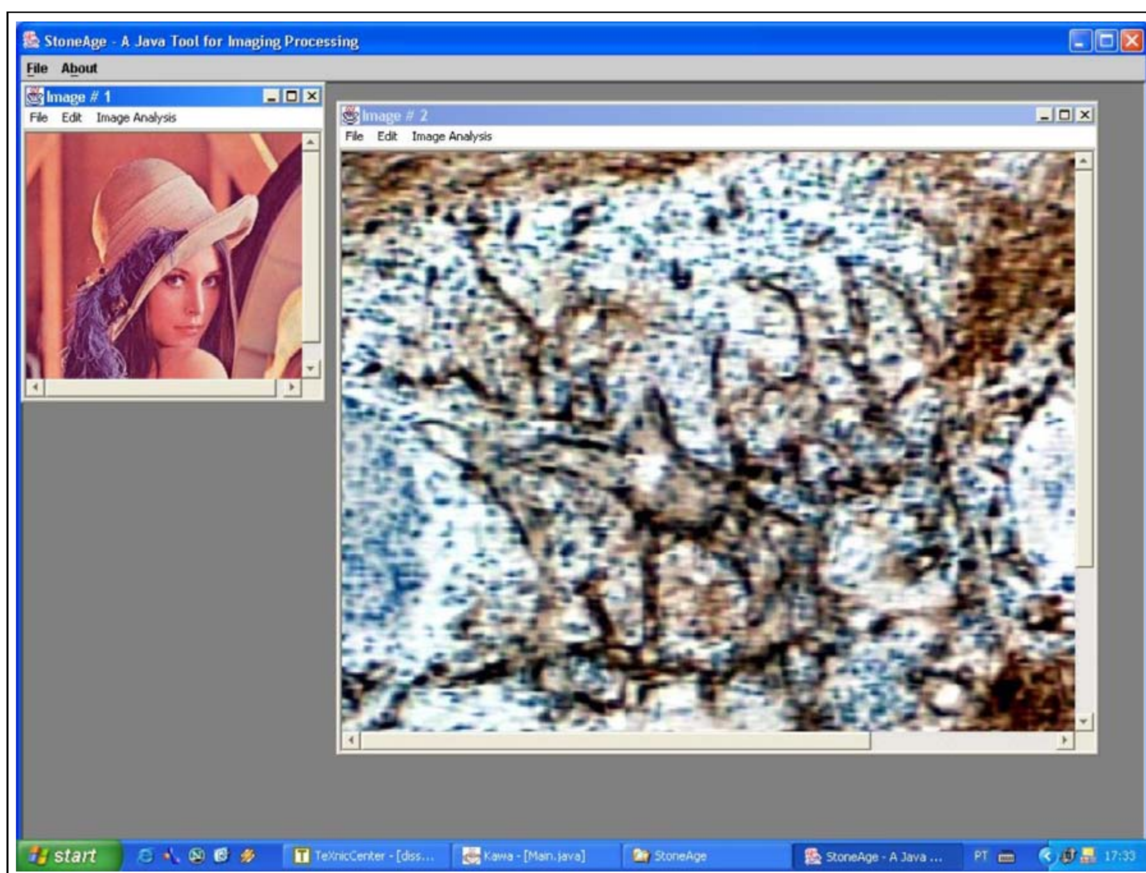


Figura 6.6: A primeira interface já funcional do StoneAge.

O outro padrão utilizado nesse trabalho tem a intenção de guardar as imagens processadas afim de recuperá-las posteriormente. É o que as operações usualmente conhecidas como `Undo` (voltar) e `Redo` (avançar) implementam. Segundo Steven J. Metsker, [MET 2002], a durabilidade desse armazenamento pode ser de momento, horas, dias ou anos. Dessa maneira, a primeira forma de implementação pensada foi o padrão memento(recordação) descrito no capítulo 5. Ele pode ser muito útil pois todos os detalhes de uma possível manipulação de arquivo ou até mesmo base de dados podem ser escondidos dentro de uma classe que será posteriormente agregada a classe `LitleWindow`. Entretanto, não há muito sentido prático em armazenar o estado parcial das imagens processadas em arquivo ou banco de dados. Usualmente essa característica é momentânea, isto é, existe apenas na memória volátil do computador enquanto o

sistema está sendo executado. Isso ocorre com os sistemas citados na seção 4.3.

Porém, sabendo que cada imagem é vista como um objeto do tipo `PlanarImage` conseguiu-se facilmente armazená-la utilizando uma estrutura do tipo coleção. Essa estrutura é provida pela linguagem Java podendo ser considerada, portanto, um padrão idiomático. Essa coleção é o objeto `Vector` que tem a mesma idéia de uma vetor unidimensional primitivo porém, em cada um dos seus índices ele é capaz de armazenar objetos inteiros. Essa implementação resolve o problema do armazenamento temporário de imagens. A nível de estruturas de dados ele opera como uma lista. Cada imagem aberta é adicionada nessa estrutura e quando o usuário precisa de algum processamento anteriormente realizado, ele apenas vai navegando pelos índices dessa lista. No código da figura 6.7 pode ser visto como é simples a manipulação desse objeto.

**Código 7:** A coleção Vector para armazenar imagens

```
01.  //este vector guarda a imagem e suas variações
02.  Vector storeImageProcessed = new Vector();
03.  //armazenando a imagem
04.  storeImageProcessed.add((PlanarImage)src);
05.  //retornando uma imagem da coleção
06.  PlanarImage src =
07.     (PlanarImage)storeImageProcessed.elementAt(posProcessed);
```

Figura 6.7: A coleção Vector utilizada para resolver o problema da operação “undo”.

Na linha 2 da figura 6.7 ocorre o instanciamento da classe Vector. Na linha 4 a imagem é adicionada na última posição desse vetor de objetos. A imagem é representada pelo objeto `src` . Nas linhas 6 e 7 ocorre a recuperação de um determinado objeto imagem de acordo com seu índice `posProcessed` . Toda vez que uma operação `Undo` é requisitada pelo usuário esse gerenciador de índices (que é uma variável inteira) tem seu valor decrementado. O contrário acontece com a operação de `Redo`, isto é, essa variável sofre um incremento unitário. Porém, em ambas deve ser controlado o limite superior e inferior dessa coleção uma vez que, esse vetor possui um início e um fim. Uma vantagem muito grande dessa coleção é que ela possui um crescimento incremental, isto é, ela cresce conforme a necessidade. Ao contrário de um objeto do tipo array primitivo que obriga a alocação de um número especificado de índices(tamanho) no momento da sua criação, a classe Vector[5] aloca um novo espaço apenas quando é necessário. Significa dizer que, se uma imagem foi alterada três vezes, há apenas três índices nessa coleção, de zero a dois. Assim, toda vez que uma imagem previamente aberta sofre algum processamento ela é automaticamente armazenada na última posição dessa estrutura dinâmica. A primeira posição sempre guarda a imagem original. Segundo Deitel e Deitel [DEI 2001a], essas coleções oferecidas pelo Java permitem ao programador utilizar estruturas de dados como listas(como no

[5]Pertencente ao pacote `java.util` a partir da versão dois do Java(1.2).

caso do Undo e Redo), pilhas, filas e árvores sem "reinventar a roda".

Para finalizar a utilização desse padrão idiomático de recordação, sublinha-se que, poderia ser utilizado o padrão de projeto Iterator explicado no capítulo 5 para fins de gerenciar o acesso seqüencial a essa coleção. Entretanto, isso apenas agregaria complexidade ao sistema pois a criação de um objeto para esse fim pode ser intuitivamente substituído por um simples contador. A figura 6.8 mostra a utilização desse objeto dinâmico de uma forma mais simples.

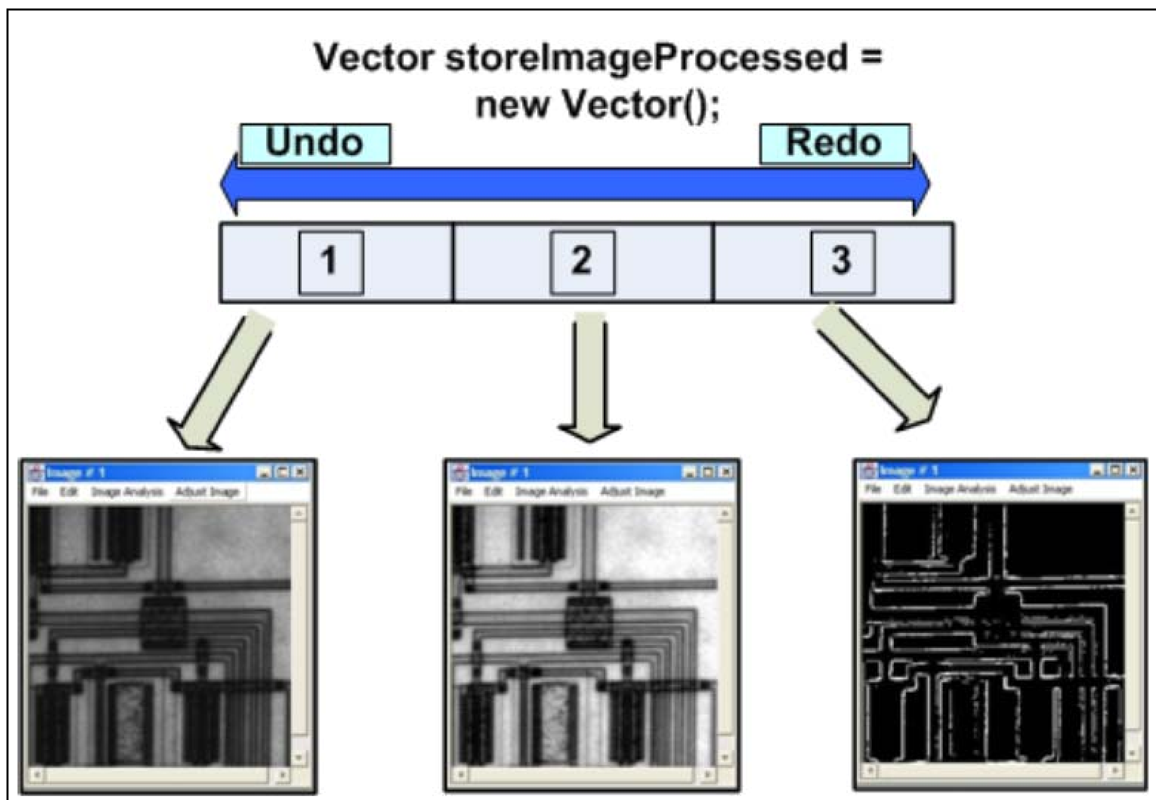


Figura 6.8: As operações de Undo e Redo sob a perspectiva de um padrão idiomático.

Esse objeto Vector também existe em C++, sob a denominação mais usual de contâiner. Assim como no Java ele também serve para guardar objetos e, também implementa operações que facilitam o seu uso como os métodos de armazenamento e recuperação dos objetos nele contidos.

O outro padrão utilizado foi o Factory Method associado com o Strategy. Ambos foram utilizados para implementar as operações de `Save` e `Save As`. Essas operações são fundamentais pois através delas é conseguido persistir a informação por algum mecanismo de armazenamento. No StoneAge essa informação é guardada em arquivos gráficos. Dentre esses há a possibilidade de manipular arquivos com extensão JPEG/JPG [6], TIFF[6], BMP[6] PBM[6] e PNG[6]. No entanto, mesmo a biblioteca JAI oferecendo os "codecs" para a abertura e construção desses arquivos eles devem ser utilizados de forma a seguirem as normas da legibilidade. Essas duas operações

[6]JPEG/JPG: Joint Photographic Experts Group (`http://www.jpeg.org/`); TIFF: Tag Image File Format (`http://partners.adobe.com/public/developer/tiff/`); BMP: Windows Bitmap Format(`http://www.microsoft.com/`) ; PBM: Portable Bit Map; PNG: Portable Network Graphics.(`www.w3.org/Graphics/PNG/`).

tem por finalidade salvar o arquivo aberto utilizando uma mesma extensão e salvar uma imagem inserida em um arquivo gráfico em uma extensão diferente da original. Por exemplo, se a imagem carregada possui a extensão PNG a operação `Save As` permite que ela seja salva, por exemplo, com extensão JPEG.

Para solucionar esse problema foi projetado a solução apresentada pela figura 6.9.

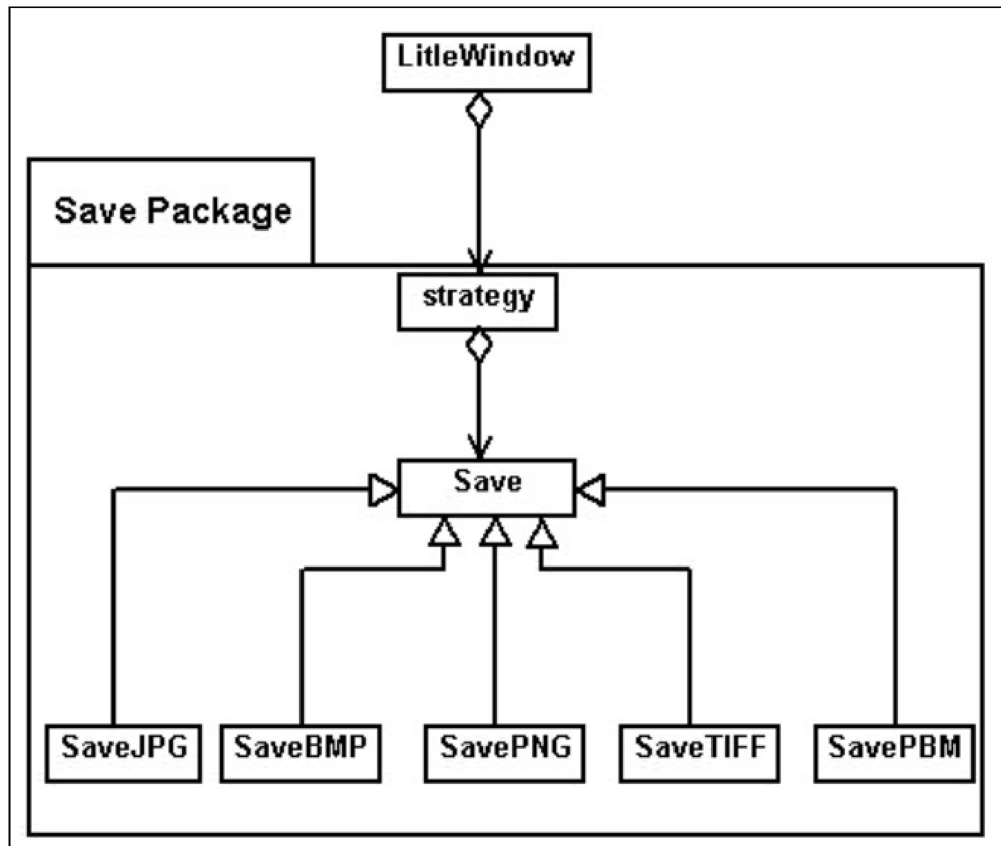


Figura 6.9: A modelagem UML das operações `Save` e `Save As` .

Na figura 6.9, o padrão Factory Method é representado pela classe base `Save` e pelas classes concretas `SaveJPG`, `SaveBMP`, `SavePNG`, `SaveTIFF` e `SavePBM` . Como descrito no capítulo 5, esse padrão Gamma é utilizado quando há a necessidade de retornar algum objeto concreto mediante determinada situação. No caso da operação de armazenamento o seu uso é bastante claro pois o sistema retorna apenas o objeto relativo ao arquivo gráfico filtrado pelo usuário. Não há lógica em retornar o objeto `SaveTIFF` e `SavePNG` se o usuário optou salvar em formato TIFF. Obviamente isso ocorre porque não há lógica em salvar um arquivo gráfico com duas extensões uma vez que os sistemas operacionais mais usuais não o reconheceriam. Assim, no interior de cada classe concreta de `Save` está escondido a complexidade referente aos “codecs” para fins de interpretação dos arquivos gráficos. Dessa forma, qualquer necessidade de mudança, seja por algum “bug” ou até mesmo a agregação de um novo formato basta reportar-se a essa hierarquia de sub-classes. Essa é uma grande vantagem dessa modelagem com classes abstratas, pois cada uma das classes apresentadas na figura 6.9 está dentro de um arquivo próprio. Assim, na necessidade de codificação de um novo “codec” basta copiar a estrutura de qualquer um

dos já existentes alterando apenas a sua lógica interna e seu escopo (nome da classe). Não há necessidade de mudar nada na classe mãe e nada nas classes irmãs.

No entanto, só a implementação do padrão Factory Method não garante a funcionalidade da operação em questão. Além da necessidade de criar uma organização de objetos falta especificar a fábrica propriamente dita, isto é, quem vai executar a chamada para determinada classe concreta. Para isso se deve ter em mente que cada classe filha implementa um algoritmo distinto mediante um mesmo escopo de operação. Nesse contexto, para gerenciar qual algoritmo deve ser invocado, foi utilizado o padrão de projeto comportamental Strategy. A intenção desse padrão pode ser deduzida do seu próprio nome, isto é, escolher qual estratégia(que é o algoritmo) deve-se utilizar. No entanto, essa estratégia deve ser exclusiva, isto é, ou invoca o algoritmo de `SaveTIFF` ou de `SaveBMP`. Para tal finalidade, o padrão Strategy encapsula em seu interior a definição de uma família de algoritmos de forma que sejam intercambiáveis [GAM 2000]. Segundo James W. Cooper, [COO 1998], esse padrão é também muito utilizado para comprimir arquivos utilizando diferentes algoritmos, capturar video utilizando diferentes esquemas de compressão e outros. Ele também é muito útil para abstrair a estrutura de classes do padrão Factory Method, isto é, a classe `LitleWindow` (que controla a escolha do usuário por salvar em determinado arquivo gráfico) se preocupa apenas em passar os argumentos necessários para o objeto Strategy que por sua vez se responsabiliza por invocar a implementação de determinado algoritmo situado nas classes mais "baixas". Esses dois padrões estão armazenados em um pacote chamado `save`, o que é muito positivo pois as classes pertencentes a essa operação de armazenamento físico ficam separadas das classes básicas do sistema. Mais uma vez, a modularidade oferecida pelos padrões acaba oferecendo uma certa organização ao sistema como um todo. Salienta-se também que, tanto a operação de `Save` quanto `Save As` é implementada utilizando a mesma estrutura apresentada na figura 6.9. Tudo vai depender do argumento passado para o objeto Strategy que gerencia os algoritmos. Se o usuário abrir uma imagem de um arquivo JPG e invocar o método `Save As` do menu de opções e após isso decidir que deseja salvar a imagem com outra extensão, o objeto Strategy em conjunto com a classe abstrata `Save` gerenciará essa operação.

Assim, a aplicação dos padrões na construção do *StoneAge* possibilita gerenciar melhor a complexidade dos futuros componentes que serão agregados. No próximo capítulo, é apresentado a construção de um componente de segmentação, que também se baseia em padrões, e como ele é preparado para ser inserido no *StoneAge*.

# 7 APLICAÇÃO: UM COMPONENTE PARA SEGMENTAÇÃO DE IMAGENS BASEADO EM FILTROS DE CONVOLUÇÃO

O objetivo desse capítulo[1] é demonstrar a aplicação dos padrões de projeto na construção de um componente para segmentação de imagens digitais baseado em contornos. Para isso será explicado primeiramente o processo de segmentação utilizado e depois a visão de projeto estrutural do componente.

## 7.1 O processo de segmentação

Segmentação de imagens é o processo pelo qual se particiona uma imagem em um conjunto de regiões uniformes colocando-as em primeiro ou segundo plano conforme o objeto de interesse [RIT 2000], [RUS 1998]. Esse processo, típico de sistemas de visão computacional, baseia-se na propriedades dos pixels que formam a imagem, isto é, a intensidade que cada um apresenta em sua respectiva banda. Dessa forma, em uma imagem com valores de pixels no intervalo entre 0 e 255 a borda é detectada quando há mudanças bruscas nos níveis de cinza, ou seja, em sua magnitude. Segundo Gonzalez e Woods, [GON 2000], essa é uma abordagem baseada na descontinuidade que, por sua vez, precisa ser complementada por um processo também de segmentação conhecido como binarização. Esse último passo, também conhecido como limiarização ou thresholding, faz-se necessário para identificar de forma mais consistente as bordas aproximando-se ao máximo do contorno da imagem.

Nesse trabalho, a detecção de bordas foi aplicado às imagens de banda simples, necessitando assim, que o aplicativo desenvolvido execute a conversão da imagem captada, que por sua vez possui um modelo de cores baseado em três bandas, isto é, RGB para o modelo monocromático, ou seja, cinza.

Segundo Ritter & Wilson [RIT 2000], uma grande variedade de técnicas são usadas para computar as bordas de uma imagem. Dentre elas, utilizou-se a técnica da transformada que aproxima o gradiente que, segundo Gonzalez e Woods, [GON 2000] é o método mais comum de diferenciação em aplicações de processamento de imagens. Para isso, são utilizados máscaras ou *kernels* de convolução que recebem essa denominação porque operam unicamente no domínio

[1] Esse capítulo baseia-se no trabalho publicado no XXIV Encontro Nacional de Engenharia de Produção, [WEL 2004]

do espaço. Para a detecção dessas bordas duas convoluções são necessárias na imagem original. A primeira operação de convolução detecta as bordas na direção horizontal e a segunda na direção vertical, formando assim, as bases do subespaço de bordas. Dentre as máscaras que existem foi implementado a de Frei e Chen pois foi a que apresentou melhor resultado. Na figura 7.1, é demonstrado essa máscara de convolução na forma de arranjo bidimensional.

| | | |
|---|---|---|
| –1.0 | –1.414 | –1.0 |
| 0.0 | 0.0 | 0.0 |
| 1.0 | 1.414 | 1.0 |

**a)**

| | | |
|---|---|---|
| 1.0 | 0.0 | –1.0 |
| 1.414 | 0.0 | –1.414 |
| 1.0 | 0.0 | –1.0 |

**b)**

Figura 7.1: O *kernel* de convolução utilizado: a ) filtro vertical e b ) filtro horizontal de Frei e Chen.

Observe na figura 7.1 que todos os coeficientes apresentam um somatório nulo, o que significa dizer que em áreas constantes a resposta será nula ocorrendo assim a diferenciação entre as regiões [RIT 2000]. Após a aplicação dessas máscaras, aplica-se um limiar ou *thresholding* global sob a imagem segmentada para definir claramente o que é borda do que é plano de fundo. Esse é um passo bastante conveniente uma vez que o algoritmo de Frei e Chen não consegue definir valores constantes e exclusivos para essas duas ocorrências [RIT 2000]. Dessa forma, com a limiarização foi conseguido uma imagem binária cujo plano de fundo apresentava valor 0 e as bordas valor 1. Assim, para a imagem f(x,y), seu limiar T(x,y) foi encontrado como mostra a equação da figura 7.2:

$$\text{T(x,y)} = \begin{cases} 0 & se \quad f(x,y) < k \\ 1 & se \quad f(x,y) \geq k \end{cases}$$

Figura 7.2: Modelo Algébrico para encontrar o Thresholding, onde k é o limite especificado pelo usuário.

Esse processo de segmentação foi utilizado em imagens de componentes industriais. A idéia era detectar os contornos dessas peças e gerar um arquivo texto contendo as coordenadas espaciais dessas bordas. Através desse arquivo a peça pode ser reconstruída em ambiente assistido por computador Autocad[2]. A figura 7.3 demonstra a parte de processamento de imagens desse trabalho. Sublinha-se que, a imagem original antes de ser convertida para tons de cinza sofreu o

[2] ©2004 Autodesk, Inc. All rights reserved. (`http://www.autodesk.com/`).

processo de equalização para fins de melhorar a distribuição da sua luminosidade e, conseqüentemente tornar mais nítida suas bordas.

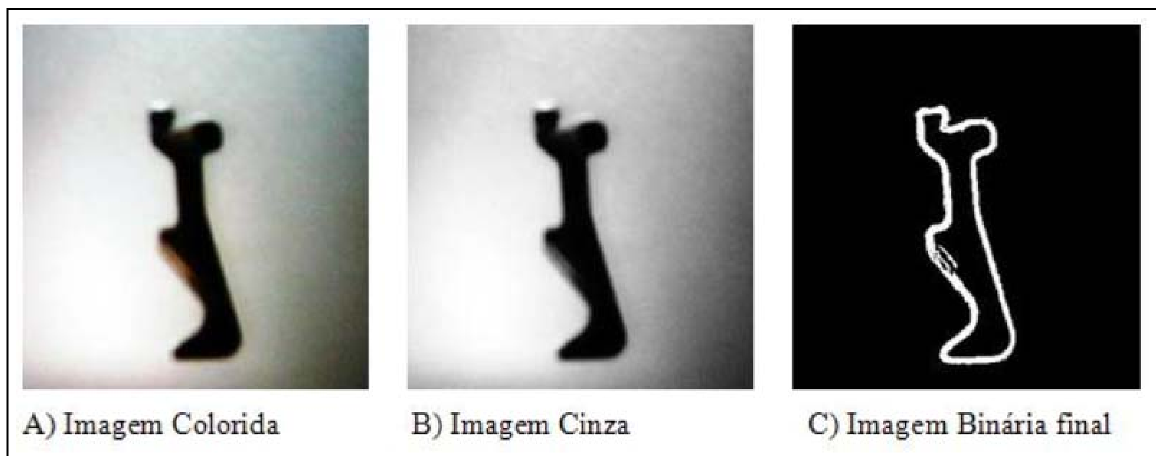


Figura 7.3: As várias etapas necessárias para detectar a borda.

## 7.2 Os padrões utilizados para projetar o componente

De conhecimento da técnica no que diz respeito ao processamento de imagens necessário para esse tipo de segmentação, foi feito então a modelagem de software para fins de codificação posterior. Para isso, foram utilizados quatro padrões de construção Gamma: Builder, Template Method, Strategy e Command.

O padrão Command mais uma vez foi utilizado para prover o controle necessário para gerenciar o componente. Porém ele não teve que ser implementado, isto é, o componente se utiliza da estrutura de controle já implementada nas classes bases do sistema. Em outras palavras, a mesma interface de comando `CommandInterFace` serviu para gerenciar as requisições dos usuários desse componente via interface gráfica.

O padrão Strategy foi utilizado de forma análoga ao problema do armazenamento de imagens descrito no capítulo anterior. Porém agora, em vez de gerenciar os algoritmos responsáveis por implementar determinado tipo de arquivo gráfico, ele implementa as várias opções de máscaras de realce de bordas anteriormente citadas na seção anterior como Prewit, Frei and Chen, Sobel e Roberts. Isso é, esse padrão está fazendo o papel da fábrica onde ocorre a manufatura de determinado objeto e que, por sua vez, apresentam responsabilidades distintas. Porém, nesse componente foi utilizado uma variante do padrão Factory Method para manufaturar os objetos. Esse padrão é o Builder.

O padrão Builder, assim como o Factory Method, também retorna um objeto a partir de uma dada requisição. Porém ele possui a vantagem de construir esse objeto a ser retornado de forma mais modular, isto é, passo-a-passo. Outra vantagem é que, o objeto complexo que está sendo construído fica separado do seu processo de composição.

Antes de usar esse padrão de projeto é necessário enfatizar três aspectos essenciais no que

diz respeito a eficácia desse componente:

1. Há diferentes filtros de realce. Assim, o projeto do componente deve permitir que o usuário possa escolher dinamicamente qual dessas máscaras será utilizada;

2. Só escolher qual o filtro utilizar não basta. O Componente também deve oferecer diferentes tamanhos para essas máscaras. Normalmente, na literatura, elas são 3x3 ou 8 conectado (como a figura 7.1), mas também a literatura descreve como possibilidade de uso máscaras de tamanho 5x5 e 7x7. Assim, é possível ter um mesmo algoritmo operando sob diferentes tamanhos de máscaras.

3. Após escolhida a máscara faz-se necessário ainda executar a operação de convolução propriamente dita. Para isso já se deve ter decidido o nome do filtro e seu tamanho. Nessa etapa de convolução são utilizadas as máscaras horizontais e verticais escolhidas anteriormente.

Nesse contexto, a principal vantagem do padrão de criação Builder na implementação desse componente é, justamente gerenciar essas várias partes necessárias para compor um objeto. Sua configuração estrutural é mostrada na figura 7.4.

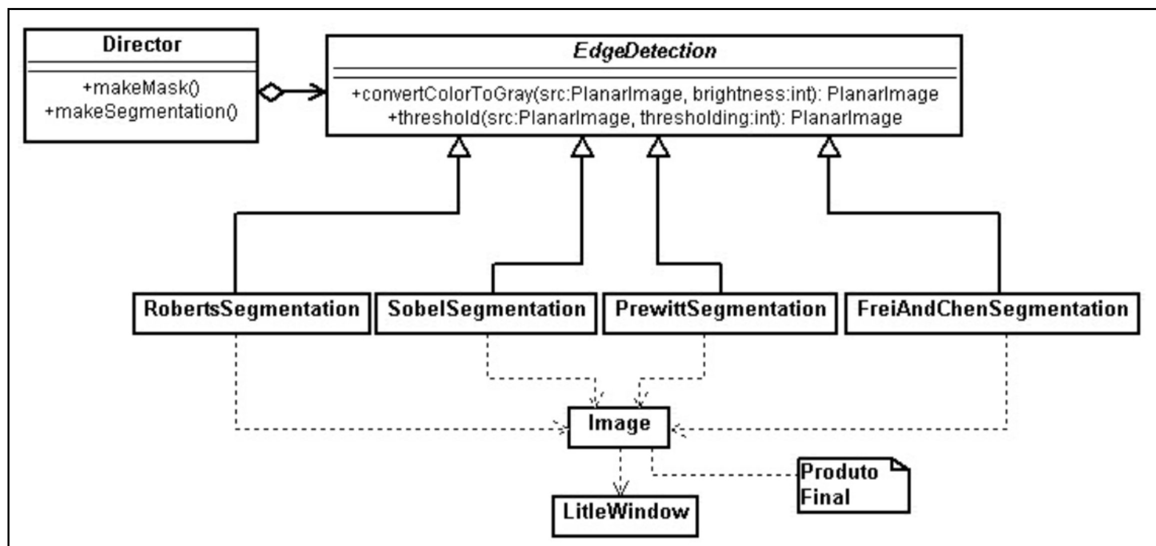


Figura 7.4: O modelo estrutural do padrão Builder. O objeto Image é o produto que está sendo construído.

O padrão Builder representado pela figura 7.4 possui um nível a mais que o Factory Method. A classe `EdgeDetection` é uma classe abstrata que tem como função abrigar diferentes implementações de filtros segundo um mesmo escopo de operação. Suas classes bases são `RobertsSegmentation`(que implementa o filtro de Roberts), `SobelSegmentation`(que implementa o filtro de Sobel), `PrewittSegmentation` e `FreiAndChenSegmentation`. Cada uma dessas classes possui o método `makeMask()` que tem como parâmetro de entrada o tamanho da máscara que o usuário optou em utilizar e o método `makeSegmentation()` que

tem como parâmetro de entrada o valor do limiar escolhido e como valor de retorno a imagem já segmentada. Ambas operações são abstratas e surgem nas sub-classes porém com valores diferentes no que se refere a sua máscara. A primeira operação constrói a máscara tomando como base seu tamanho e a segunda realiza a operação de convolução e depois de binarização. A classe `Image` representa o produto complexo que se está querendo produzir (que no caso é a imagem segmentada). A classe `Director` faz parte da especificação do padrão Builder e é a responsável por invocar todos os métodos das classes filhas em um único método chamado `Constructor`.

A classe `Image` recebe a imagem a ser segmentada do sistema base. É essa imagem original que se transformará no produto final. Por isso que ela possui uma relação de dependência entre as classes filhas e com uma das classes base do StoneAge chamada `LitleWindow`. As classes filhas dependem do produto final pois, este, em seu estado inicial, carrega a imagem a ser processada pela lógica algorítmica contida no interior dessas classes concretas. Já o produto final depende da classe base do sistema porque esse último é que provê a imagem propriamente dita. A figura 7.5 mostra o projeto completo do componente e como ele se integra com a classe base do sistema via interface gráfica provida pelo objeto `SegmentationGUI` e com a classe que gerencia a chamada dos algoritmos denominada de `Strategy`.

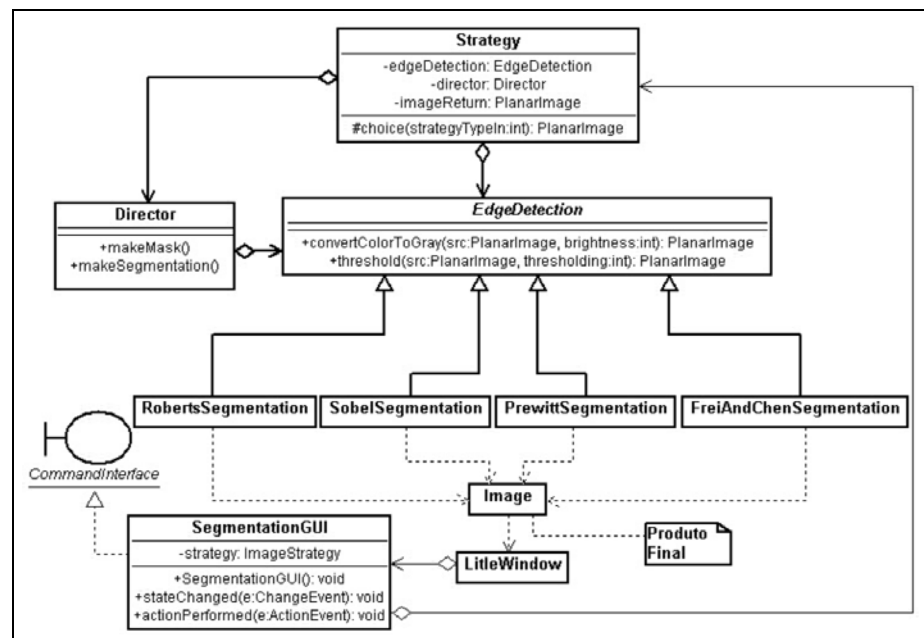


Figura 7.5: A visão de projeto completa do componente de segmentação baseado em máscaras.

O último padrão utilizado no desenvolvimento desse componente foi o Template Method. Como descrito no capítulo 5 esse padrão tem a intenção de prover um método modelo para vários objetos. Para o processo de segmentação ocorrer não basta apenas aplicar filtros de convolução. É necessário ainda tratar a imagem de entrada e também complementar o processo de realce dos filtros pelo processo de binarização. Se a imagem que se está intencionando segmentar for RGB (usualmente conhecidas como multiespectral ou colorida) o componente de

segmentação deve convertê-la para tons de cinza, uma vez que é mais simples operar sob esse tipo de imagem além de ser pré-requisito para o processo de limiarização futura. De outro lado, após a aplicação de determinada máscara é utilizado o processo de binarização (thresholding) para garantir a exclusão dos pixels que pertencem a borda daqueles que não pertencem, isto é, o fundo da imagem. Assim, cada classe concreta de `EdgeDetection` deve implementar ambas funcionalidades para garantir a eficácia do componente. No entanto, como a lógica de ambas operações se mantém igual independente da classe concreta onde está implementada, é possível tratá-las de uma forma mais elegante. Para isso, basta implementar esses métodos na própria classe abstrata (usualmente chamada de mãe) e apenas chamá-los quando for conveniente nas classes filhas. Assim, evita-se replicação de código nas classes concretas (comumente chamada de filhas). Assim, esses métodos se tornam modelos justificando assim o seu nome pela literatura.

A figura 7.6 mostra a interface gráfica desse componente. Essa interface corresponde a classe `SegmentationGUI` da figura 7.5.

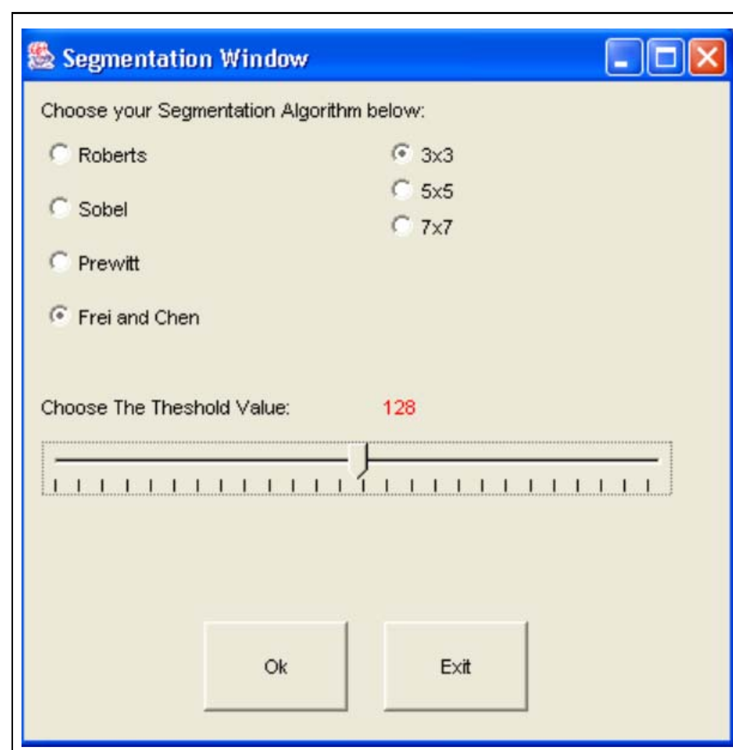


Figura 7.6: A interface gráfica do componente de segmentação baseado em contornos.

Muitas operações em processamento de imagens se baseiam na aplicação de filtros. Dessa forma, a utilização desses quatro padrões, especialmente pelo Builder que dá o suporte central ao componente, podem ser visto com muita utilidade no desenvolvimento de software para imagens. Um bom exemplo é a utilização desses padrões para construir um componente para morfologia binária[3], onde o elemento estruturante apresenta diferentes formatos e tamanhos. Assim

[3]Compreende o estudo da estrutura geométrica de imagens binárias, isto é, aquelas compostas por pixels de valores zeros e uns.

pode-se utilizar o padrão Builder para construir passo-a-passo toda a representação necessária para implementar um objeto que representa o elemento estruturante para, posteriormente utilizá-lo em operações de dilatação, erosão, abertura, fechamento e outras. Em nível de aplicação esse componente se segmentação baseado em contornos pode ser visto como um pré-requisito para um componente de morfologia binária pois, o alvo de estudo dessa parte de morfologia são imagens binárias que por sua vez são o resultado final desse componente de segmentação. Assim, também a nível de funcionalidade um componente acaba ajudando o outro na resolução de algum problema.

# 8 CONCLUSÃO

Os objetivos de construir um software independente de plataforma, de código-livre e de componentes adaptáveis para o processamento e análise de imagens foram cumpridos. Assim, o *StoneAge*, não é considerado uma "bola de lama" pois na sua construção foram aplicados preceitos de engenharia de software avançada, ou seja, os padrões de construção. No entanto, apenas com o decorrer dos anos será possível avaliar se o ciclo de vida do *StoneAge* foi capaz de dar suporte para as várias aplicações do grupo de pesquisa, isto é, se ele realmente apresentou características como fácil manutenção, legibilidade de código e reuso.

Porém, na implementação do sistema e dos componentes desse trabalho a experiência mostrou que os padrões propiciaram formas comprovadamente ótimas de gerenciar a complexidade de um componente afim de torná-lo modular e flexível para fins de reuso. Por exemplo, a aplicação do padrão Factory Method descrito no capítulo 5 para gerenciar as várias formas de armazenamento de imagens em arquivos gráficos. Esse padrão possibilita uma maneira modular de codificação porque cada classe concreta da sua estrutura encapsula uma lógica diferente sob um mesmo nome de operação. Assim, todas essas classes filhas possuem o escopo de métodos comum o que assegura maior facilidade de interpretação pelos novos desenvolvedores, uma vez que, mediante a necessidade de alguma alteração ou criação de nova classe concreta basta basear-se fielmente nas que já existem, isto é, nas suas irmãs. O padrão Strategy também apresenta bons resultados pois não apenas serviu para encapsular os detalhes da invocação de determinado algoritmo, mas como também é o responsável pela gerência de todo o componente. Isso é muito positivo porque para o sistema base invocar um determinado componente basta instanciar o objeto que representa o padrão Strategy. Toda a complexidade do componente fica escondida. Esse processo pode ser visto na figura 7.5 e 6.9. Em outras palavras, a comunicação entre um determinado componente e as classes bases do sistema é realizada quase que totalmente através do padrão Strategy. Isso ocorre quando se utiliza o padrão Builder onde também é necessário o sistema base passar a imagem a ser construída para a classe que representa o produto final. Porém, não há necessidade de agregação pois essa passagem é feita diretamente para o atributo que representa a imagem dessa classe de forma estática.

Outro resultado observado diz respeito a escalabilidade do sistema. Nas primeiras versões do sistema não havia muitos componentes agregados e sua funcionalidade ainda era bastante restrita. Mesmo assim, nesses estágios iniciais já eram utilizados padrões de construção. No

entanto, eles só começaram a realmente demonstrar os seus benefícios a partir do momento em que o sistema foi crescendo no que diz respeito as suas funcionalidades. Assim, a avaliação dessa dissertação não se encerra aqui, mas se dará ao longo do desenvolvimento dos projetos de pesquisa. O padrão Command propiciou regras de controle para o gerenciamento de toda a interface gráfica do sistema base e dos componentes. Qualquer modificação ou agregação de novos componentes gráficos basta seguir um modelo onde todos esses componentes são considerados como classes isoladas que contém operações comuns porém com diferentes implementações. Assim, quando o sistema foi se tornando grande e, conseqüentemente a sua interface gráfica também, a principal vantagem oferecida pelo padrão Command foi a forma com que ele gerenciou a complexidade resultante da associação entre componentes gráficos e requisição de componentes. Essa associação ocorre porque cada componente gráfico executa uma ação que invoca determinada lógica que está presente em um componente e que, por sua vez, está alocado em um pacote distinto.

O padrão arquitetural Layer permitiu construir um código bastante enxuto, porém funcional. Isso foi muito visível na implementação da camada de aplicação. Muitas operações não precisaram ser reimplementadas, isto é, elas apenas foram usadas. Assim, não houve uma mistura entre serviços básicos com a lógica de montagem de um componente. Por exemplo, para construir o componente de segmentação não foi necessário criar uma estrutura para realizar a convolução da máscara na imagem pois o núcleo já a implementa. Assim também é possível criar aplicações com um número reduzido de linhas de código.

A utilização dos padrões idiomáticos podem ser vistos como um mecanismo para resolver um determinado problema utilizando exclusivamente características da linguagem de programação. No entanto, é necessário ter um certo grau de conhecimento sobre a linguagem utilizada para conseguir explorá-la melhor. O uso da coleção `Vector` do Java, garantiu de forma bastante simples as operações de `Undo` e `Redo` para o sistema base. Essas operações são de extrema importância para o StoneAge pois na ocorrência de alguma modificação não satisfatória na imagem, o usuário pode retroceder ou avançar para uma determinada imagem sem a necessidade de efetuar a abertura novamente de seu arquivo gráfico.

A modelagem do sistema e de seus componentes através da notação UML e suas ferramentas, propicia ao desenvolvedor a visão necessária para trabalhar a parte mais difícil na concepção de um sistema. Essa parte é o processo de reconhecer, enxergar a aplicação de um determinado padrão para um problema de imagens. Esse problema no início é muito grande, principalmente no momento de conjugar os vários padrões identificados que compõem o componente. E para esse processo de reconhecimento, é necessário ter conhecimento de todos os padrões apontados pela literatura e, também, conhecimento sobre o problema na área de imagens. Se esses dois aspectos estão dissociados não é possível efetuar uma boa implementação.

Por fim, o trabalho de construir componentes para processamento de imagens requer tanto conhecimentos sobre engenharia de software no que se refere aos padrões, como o conhecimento necessário sobre o domínio da aplicação. De posse disso, o último quesito necessário é optar por uma linguagem de programação que dê suporte às estruturas dos padrões de construção.

# REFERÊNCIAS

[AKR 2003] AKRE, A.; TABIRCA, S. **Imaging technologies in java**. Kilkenny City, Ireland: Proceedings of the 2nd international conference on Principles and practice of programming in Java, 2003. 159-161p.

[ALE 1977] ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. **A pattern language**. New York: Oxford University, 1977.

[ALE 1979] ALEXANDER, C. **The timeless way of building**. [S.l.]: Oxford University Press, 1979.

[ARM 2003] ARMOUR, P. G. **The laws of software process**: a new model for the production and management of software. [S.l.]: CRC Press LLC, 2003.

[BOO 1999] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The unified modeling language user guide**. [S.l.]: Addison Wesley, 1999.

[BUS 1996] BUSCHMANN, F. et al. **Pattern - oriented software architecture**: a system of patterns. New York: John Wiley & Sons Ltd, 1996. v.1.

[CAI 2000] CAI, X. et al. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. **7th Asia-Pacific Software Engineering Conference (APSEC 2000)**, p.372 –, 2000.

[COA 1993] COAD, P.; NICOLA, J. **Object-oriented programming**. [S.l.]: Prentice Hall, 1993.

[COO 1998] COOPER, J. W. **The design patterns - java companion**. [S.l.]: Addison-Wesley, 1998.

[CRN 2002] CRNKOVIC, I.; LARSSON, M. **Building reliable component-based software systems**. [S.l.]: Artech House, 2002.

[DEI 2001] DEITEL, H. M.; DEITEL, P. J. **C++ como programar**. Porto Alegre: Bookman, 2001.

[DEI 2001a] DEITEL, H. M.; DEITEL, P. J. **Java™ como programar**. Porto Alegre: Bookman, 2001.

[DON 2001] DONALDSON, S. E.; SIEGEL, S. G. **Successful software development**. [S.l.]: Prentice-Hall, Inc, 2001.

[D'OR 2004] D'ORNELLAS, M. C.; WELFER, D. **A generic programming approach for automatic color image segmentation using cluster homogeneity**. Florianópolis, Santa Catarina - Brasil: X International Conference on Industrial Engineering Management, 2004.

[D´OR 2001] D´ORNELLAS, M. C. **Algorithmic patterns for morphological image processing**. 2001. Tese (Doutorado em Ciência da Computação) — University van Amsterdam.

[D'OR 2003] D'ORNELLAS, M. C. **A Parallel Algorithmic Pattern**. The Third Latin American Conference on Patterns Languages of Programmig - SugarLoafPlop-Porto de Galinhas, PE, 12-15 agosto: [s.n.], 2003. 119-134p. Disponível em <http://www.cin.ufpe.br/ sugarloaf/>. Acesso em maio de 2004.

[ECK 1998] ECKSTEIN, R.; LOY, M.; WOOD, D. **Java swing**. [S.l.]: O'Reilly, 1998.

[ENG 2000] ENGELS, G.; GROENEWEGEN, L. Object-oriented modeling: a roadmap. **Communications of the ACM**, v.43, n.5, p.103–116, May 2000.

[FAY 2003] FAYAD, M. E.; HAMZA, H.; RAJAGOPALAN, J. **The recovery design pattern**. [S.l.]: The 2003 IEEE International Conference on Information Reuse and Integration, 2003. 594-600p. Disponível em: <`http://dblp.uni-trier.de`>. Acesso em: dez. 2004.

[FOW 1999] FOWLER, M. **Refactoring**: improving the design of existing code. [S.l.]: Addison-Wesley, 1999.

[GAL 2002] GALITZ, W. O. **The essential guide to user interface design**: an in introduction to GUI design principles and techniques. second.ed. [S.l.]: John Wiley & Sons, 2002.

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

[GON 2000] GONZALEZ, R. C.; WOODS, R. E. **Processamento de imagens digitais**. São Paulo - Brasil: Edgard Blücher, 2000.

[GOS 2004] GOSLING, J. **The mars mission continues**. Sun Microsystems, Inc, Disponível em: <`http://www.sun.com/mars`>. Acesso em: abr. 2004.

[JOS 1999] JOSUTTIS, N. M. **The c++ standard library**: a tutorial and reference. [S.l.]: Addison Wesley, 1999.

[KNU 1999] KNUDSEN, J. **JAVA 2d graphics**. [S.l.]: O'Reilly, 1999.

[KOB 1999] KOBRYN, K. UML 2001: a standardization odyssey. **Communications of the ACM**, v.42, n.10, p.29–37, October 1999.

[LAR 2000] LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos. [S.l.]: Bookman, 2000.

[LEA 1993] LEA, D. **Christopher alexander**: an introduction for object-oriented designers. Disponível em: <`http://gee.cs.oswego.edu/dl/ca/ca/ca.html`>. Acesso em: abr. 2003.

[MAY 2003] MAY, D.; TAYLOR, P. Knowledge management with patterns. **Communications of the ACM**, v.46, n.7, p.94–99, 2003.

[MET 2002] METSKER, S. J. **Design patterns java workbook**. [S.l.]: Addison Wesley, 2002.

[MOH 2002] MOHAMED, N. et al. JOPI: a java object-passing interface. **Communications of the ACM**, v.45, n.11, p.37 – 45, November 2002.

[PET 2001] PETERS, J. F.; PEDRYCZ, W. **Engenharia de software**: teoria e prática. Rio de Janeiro: Campus, 2001.

[POP 2003] POPPENDIECK, M.; POPPENDIECK, T. **Lean software development**: an agile toolkit. [S.l.]: Addison Wesley, 2003.

[RIT 2000] RITTER, G. X.; WILSON, J. N. **Handbook of computer vision algorithms in image algebra**. New York.: CRC Press, 2000.

[RUS 1998] RUSS, J. C. **The image processing handbook**. [S.l.]: CRC Press, 1998.

[SAL 1997] SALVIANO, C. F. Introdução à software patterns. **XI SBES, Fortaleza, Ceará - Brasil**, p.22, 1997.

[SAN 1995] SANTOS, L. P. P. D. **Visão por computador**. [S.l.]: Universidade do Minho - Departamento de Informática, 1995.

[SHA 2000] SHAPIRO, L.; STOCKMAN, G. **TextBook**: computer vision. Disponível em: < `http://web.cps.msu.edu/~stockman/book/`>. Acesso em: ago. 2004.

[Sun 1999] Sun Microsystems. **Programming in java ™advanced imaging - release 1.0.1**. [S.l.]: A Sun Microsystems, Inc. Business, 1999. Disponível em: <http://java.sun.com/products/java-media/jai/>. Acesso em: abr. 2004.

[Sun 2004] Sun Microsystems. **Java$^{TM}$APIs for imaging**: enterprise-scale, distributed 2d applications. Disponível em: <http://java.sun.com/products/java-media/jai/collateral/datasheet.html>. Acesso em: mar. 2004.

[TOR 2001] TORRES, G. **Redes de computadores**: curso completo. [S.l.]: Axcel Books, 2001.

[VIT 2003] VITHARANA, P. Risks and challenges of component-based software development. **Communication of the ACM**, v.8, p.67–73, August 2003.

[VÖL 2002] VÖLTER, M.; SCHMID, A.; WOLFF, E. **Server component patterns**: component infrastructures illustrated with EJB. [S.l.]: John Wiley & Sons, 2002.

[WAT 2004] WATT, D. A.; FINDLAY, W. **Programming language design concepts**. [S.l.]: John Wiley & Sons, Ltd, 2004.

[WEI 1999] WEIHE, K. **A software engineering perspective on algorithmics**. [S.l.]: Fakultät für Mathematik und Informatik of Konstanz, Germany, 1999. Research report. (ISSN 1430-3558).

[WEL 2003] WELFER, D.; BASSO, F. P.; D'ORNELLAS, M. C. **Framework colaborativo para processamento de imagens utilizando a tecnologia jini**. Ouro Preto, Minas Gerais: ABEPRO - Associação Brasileira de Engenharia de Produção, 2003. 251p.

[WEL 2004] WELFER, D.; SILVA, A. D. D.; D'ORNELLAS, M. C. **Programação de equipamentos CNC através da análise de imagens por segmentação**. Florianópolis, Santa Catarina: [s.n.], 2004. XXIV Encontro Nacional de Engenharia de Produção e X International Conference on Industrial Engineering Management.