UFSM

Trabalho de Graduação

Um Componente Genérico para Entrada, Saída e Acesso a Imagens

Fábio Natanael Kepler

Curso de Ciência da Computação

Santa Maria, RS, Brasil

2003

Um Componente Genérico para Entrada, Saída e Acesso a Imagens

por

Fábio Natanael Kepler

Trabalho de Graduação apresentado ao Curso de Ciência da Computação – Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação.

Curso de Ciência da Computação

Trabalho de Graduação n° 154

Santa Maria, RS, Brasil

2003

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Graduação

Um Componente Genérico para Entrada, Saída e Acesso a Imagens

elaborado por

Fábio Natanael Kepler

como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Comissão Examinadora:

____
Marcos Cordeiro D'Ornellas
(Orientador)

____
Felipe Martins Müller

____
Rafael Port da Rocha

Santa Maria, 4 de fevereiro de 2003.

Algo que aprendi em uma longa vida: toda nossa ciência, medida contra a realidade, é primitiva e infantil - e ainda assim, é a coisa mais preciosa que temos.

(Albert Einstein)

Num momento onde tantos estudantes no mundo estão calculando, não é desejável que alguns, que podem, sonhem?

(René Thom, 1972)

Ao meu pai.

Agradecimentos

Quero agradecer a todos os meus professores, que foram pessoas vivas na minha realidade além da sala de aula, e que me ajudaram, do seu jeito particular, a enxergar um horizonte cada vez maior, todos os dias. Inclusive ao ponto de, mesmo sem perceberem, me motivarem a perseguir seus passos 'profissionais', ainda que, e no fim das contas, "ganha menos, mas faz o que gosta" :-). Obrigado pelas aulas fora da sala – fora do CT, fora da UFSM... e dentro do campo de futebol, comendo churrasco, bebendo cerveja (ou coca-cola).

Um agradecimento especial ao meu orientador, Marcos d'Ornellas, e ao meu co-orientador, Marcos Carrard. Sem a ajuda destes dois Marcos, talvez eu tivesse deixado escapar o princípio que tentei manter durante o curso: não fazer um trabalho só por fazer. Obrigado por me motivarem e me mostrarem a saída quando eu já não enxergava solução.

Quero, também, agradecer aos meus diversos colegas, tanto de turma quanto de curso. Aos de turma, inclusive os que vão se formar mais tarde, por exporem seus problemas, não só algorítmicos, perguntando por uma opinião, por compartilharem horas(!) de estudo para as angustiantes provas, para os trabalhos no turno da madrugada, e por compartilharem outras tantas horas de futebol, "estudos aplicado de redes" e churrascos (mesmo quando compravam só linguicinha pra comprar mais cerveja).

Um agradecimento, também, àqueles colegas de curso que, mesmo sendo por um período menor do que estes quatro anos (por que alguns já foram e outros recém vieram), trocaram idéias, almoços no RU, conversas quando faltava luz, voltas de ônibus de madrugada, e bolachas no fim da tarde. Às gurias do curso(!) Grasi, Mônica, Rosana, Gabi e Adriana, que não deixam o laboratório ficar vazio de manhã cedo, e às ex-(gurias do curso) Paty, Márcia, Dany e Zane, que já tinham um chimarrão bem cedo e durante o dia todo, obrigado a vocês todas pela companhia e

pelas conversas.

Um muito obrigado em especial aos meus amigos, das minhas duas cidades, Panambi ou Santa Maria. Sempre foi difícil ter que sair de uma delas, mas também era muito bom voltar para elas. Valeu por terem trocado um pedaço de suas vidas com a minha.

Tenho que citar os nomes, para agradecer aos que juntaram os três parágrafos acima, sendo meus colegas e meus amigos. O Tapera, que na verdade é Tiago mas ninguém sabe :-), que filosofa em inglês sobre a vida, o Ninja, que também tem um nome secreto (Daniel...:-), que teima e filosofa ao mesmo tempo, o Lucas, sócio empreendedor, de caronas e de desabafos, que filosofa sobre a vida e sobre as mulheres, e o Diego, que chamam misteriosamente de alemão, que cria comigo as idéias mais mirabolantes ainda não existentes (’dupla dinâmica KK’), e que filosofa sobre a vida, o ensino, e o ’sistema’. Muito obrigado pela convivência filosofada que tivemos. :-) Diego e Lucas, uma empresa nossa de verdade iria dar certo.

Um obrigado também aos meus amigos de Panamba-city, que mesmo com a maioria morando fora, ainda aparecem pra gente jogar bola na chuva. Aos grandes ’opas’, Eduardo, Fábio Kopp, Michel e Ricardo, ’mestres’ na filosofia feminina; aos ’heróis’ Frede, Gian e Lucas, guerreiros zulus; e aos outros mais que encheriam essa página, valeu mesmo, por tudo. E, óbvio, um muito obrigado às minhas queridas amigas Cátia, Franci, Nika, Jo, Leti, Dani, e vocês mais, pelas conversas sinceras e abertas; vocês tornam a vida mais divertida.

Obrigado, enfim, a todos meus amigos ”craques”, que conseguem ser ”habilidosos” fora do campo também.

Por último, e principalmente, um muito obrigado à minha família toda. Aos meus primos, Jonas, Éllen e Ricardo; as aventuras nas casas de nossas mães eram muito legais. Meus tios parceiros, cada um em uma cidade, obrigado pelas hospedagens, caronas, jantas e risadas. Aos meus grandes irmãos, Diogo e Marcos, que são meus amigos de coração e de sangue, aventureiros e ’arteiros’ em todos os tempos; obrigado pelo amor de vocês.

A vocês, pai e mãe, pelo amor e apoio incondicional sempre. E a Deus, pela força e direção na estrada da vida.

Sumário

Lista de Figuras ix

Resumo x

1 Introdução 1

2 Revisão de Literatura 4

2.1 Programação Genérica e Processamento de Imagens 4

2.1.1 Programação genérica 8

2.1.2 A linguagem C++ e a Biblioteca STL 9

2.2 Formatos de arquivos de imagens 10

2.3 Componentes Genéricos em Processamento de Imagens 11

2.4 Estado da Arte 12

2.4.1 A biblioteca `ImageIO` 12

2.4.2 A biblioteca `VIGRA` 14

2.5 O Núcleo de uma Ferramenta para Processamento de Imagens 15

3 Metodologia 17

3.1 Desenvolvimento do Componente de Entrada e Saída 17

3.2 Definição da Estrutura 19

3.3 Detalhes de Implementação 24

3.3.1 Formatos de Arquivo de Imagem Suportados 24

3.3.2 Implementação das Classes 26

3.3.3 Estrutura Interna em Memória 28

3.3.4 O Problema da Especialização de Métodos com *Templates* . . 30

3.3.5 Estrutura Atual 34

3.3.6 Ferramentas Utilizadas 35

4 Resultados **37**

5 Conclusões e Trabalhos Futuros **39**

Referências **41**

Anexo A Manual de Referência **44**

Lista de Figuras

2.1 Exemplo de código em C++ que utiliza funções virtuais. 7

2.2 Modelo lógico do padrão de projeto de fábrica 13

2.3 Proposta de arquitetura da ferramenta para processamento de imagens 16

2.4 Estrutura do núcleo da ferramenta 16

3.1 Organização interna previamente proposta do componente 20

3.2 Estrutura prévia do componente com o problema de modularidade e flexibilidade 21

3.3 Estrutura com a classe de armazenamento. 22

3.4 Estrutura definida para o componente. 23

3.5 Respectivas classes na estrutura do componente 25

3.6 Diagrama de sequência na leitura de um arquivo 26

3.7 Diagrama de sequência na gravação de um arquivo 27

3.8 Exemplo de método utilizando o mecanismo de *templates*. 28

3.9 Exemplo de especialização de método que utiliza o mecanismo de *templates*. 29

3.10 Exemplo de utilização do contêiner `vector` e alguns de seus métodos 30

3.11 Exemplo de implementação do método de uma classe sem *templates*. . 31

3.12 Exemplo de implementação do método de uma classe utilizando *templates*. 31

3.13 Exemplo de implementação de um método gabaritado de uma classe também gabaritada. 32

3.14 Exemplo de especialização de um método gabaritado. 33

3.15 Exemplo de especialização de um método gabaritado e de uma classe gabaritada. 34

3.16 Estrutura atual do componente de entrada e saída 35

RESUMO

Trabalho de Graduação
Ciência da Computação
Universidade Federal de Santa Maria

Um Componente Genérico para Entrada, Saída e Acesso a Imagens

Autor: Fábio Natanael Kepler
Orientador: Marcos Cordeiro D'Ornellas
Data e Local da Defesa: Santa Maria, 4 de fevereiro de 2003.

Neste trabalho, apresentamos o projeto e a implementação de um componente de *software* capaz de realizar operações de entrada e saída de imagens, entre um meio de armazenamento em massa e uma biblioteca externa, de maneira transparente e eficiente. Para justificar seu uso, mostramos os principais problemas existentes atualmente no desenvolvimento de aplicações para processamento de imagens e também as principais deficiências nas abordagens usadas para implementá-las. Propomos então, para solucionar estes problemas, o uso do paradigma de programação genérica aliado ao paradigma de componentes e ao de programação orientada a objetos, balanceando, desta forma, as vantagens e desvantagens presentes em cada um deles. Esta nova abordagem nos permite alcançar transparência, flexibilidade e eficiência, permitindo que novos formatos de arquivos de imagem possam ser facilmente adicionados ao componente, sem que qualquer ambiente externo que o utilize seja influenciado por causa disto. Conseguimos, também, abstrair da implementação o tipo de representação de uma imagem e os detalhes de acesso a ela, através do uso de estruturas genéricas para armazená-la em memória e do uso de *templates* para generalizar os algoritmos que acessam estas estruturas. Finalmente, alcançamos a generalidade proposta sem, contudo, perder eficiência no desempenho do componente.

Capítulo 1

Introdução

Técnicas de processamento de imagens têm gerado várias aplicações úteis em diversas áreas como visão computacional e imageamento médico. Estas aplicações oferecem aos pesquisadores novos meios de visualizar os dados e suas estruturas. O desenvolvimento destas aplicações, entretanto, implica em desafios aos engenheiros de software. Um destes desafios se refere à entrada e saída de arquivos: Como estes softwares podem ser desenvolvidos de maneira a funcionar com dados armazenados em uma grande variedade de diferentes formatos de arquivos guardados em disco [CHA 2001]? Um segundo desafio, quanto à representação genérica de uma imagem, é: Como armazenar internamente e como dar acesso aos dados de uma imagem independentemente do tipo de imagem?[1] E finalmente, como resolver tudo isso sem perda de eficiência?

À primeira vista, uma solução parece simples e direta: decidir para quais formatos oferecer suporte e então codificar algumas funções que entendam cada um destes formatos. O problema é que isto torna a aplicação muito específica a um só problema. E isto é inapropriado quando se trata de uma comunidade de pesquisa com times espalhados por várias instituições e países, que possuem, cada um deles, seu próprio formato de arquivo. Quando necessitássemos usar um novo formato, ou compartilhar arquivos com outros times de uma comunidade global, precisaríamos escrever várias rotinas de conversão de arquivos.

Isto é exatamente o quê acontece atualmente com aplicações que utilizam ar-

[1] Os termos "tipo de imagem" e "formato de arquivo" possuem significados distintos. O primeiro significa o tipo dos dados que compõem uma imagem, isto é, os valores dos pixels, que podem ser inteiros, reais, complexos, ou estruturas mais complicadas. O último se refere à maneira como uma imagem será gravada em um arquivo dentro do disco, dependendo das informações conhecidas da imagem.

quivos de imagens. Como a maior ênfase é dada ao desenvolvimento do núcleo destas aplicações, a camada responsável pela entrada e saída de arquivos é implementada de forma básica e apenas a fim de que funcione. O suporte aos formatos de arquivos de imagem é feito através das suas conversões para um formato padrão que a aplicação define e sobre o qual trabalha com a imagem em memória. Como os formatos de arquivos de imagem existentes atualmente não são compatíveis entre si, pois cada um enfatiza características distintas – como melhor qualidade da imagem ou menor espaço ocupado em disco – e representa a imagem com tipos de valores de pixel diferentes, as conversões diretas de um formato para outro acabam por inserir problemas, como perda de qualidade da imagem, que precisam ser então resolvidos pelas camadas mais altas da aplicação.

Procurando mudar esta abordagem atual, dando um passo adiante no desenvolvimento de aplicações que utilizam arquivos de imagens, buscamos o projeto e a implementação de uma camada de entrada e saída de maneira diferenciada e inovadora. Queremos retirar das camadas mais altas de uma aplicação a responsabilidade por resolver os problemas advindos da conversão direta de arquivos, de modo que os dados originais da imagem possam ser mantidos para a aplicação genérica e transparentemente.

Para isto, é necessário o desenvolvimento de um componente de *software* para entrada e saída de imagens que seja implementado de forma modular e expansível [VIL 2001], de maneira que o suporte a novos formatos de arquivo possa ser facilmente incrementado. Também é necessário que este componente implemente uma estrutura interna genérica e que forneça métodos de acesso a esta estrutura de forma transparente, de modo a melhorar o código escrito e contribuir para a modularidade do componente, permitindo a fácil inserção de novos tipos de imagem.

A idéia principal consiste, então, na implementação deste componente para entrada e saída de dados de imagens. Estes dados, para efeitos de discussão, existem sob duas formas: externamente como arquivos em disco, e internamente como estruturas em memória. Quando necessitamos efetuar um processamento sobre uma imagem, utilizamos sua forma em memória e, para posteriormente visualizarmos o resultado, o salvamos em um arquivo em disco.

O componente deve reconhecer diversos formatos de arquivos de imagens, tanto para leitura quanto para escrita, e permitir a fácil inserção do suporte a novos deles. Deve, também, possuir uma estrutura interna bem definida, que contenha todas as

informações e dados necessários para a imagem. Esta estrutura deve ser genérica e precisa fornecer acesso transparente às informações contidas nela, independentemente do formato de arquivo carregado ou a ser salvo e do tipo dos dados da imagem.

Assim, temos três desafios principais a serem vencidos que exigirão bastante atenção. Primeiro, não importa qual formato de arquivo de imagem que queremos acessar, uma vez definidas as funções que reconhecem cada formato, todas elas devem necessariamente guardar os dados lidos em um mesmo local e da mesma maneira; a estrutura que conterá estes dados não pode depender do tipo de representação da imagem e nem do formato do arquivo lido do disco.

O segundo desafio é de que maneira uma biblioteca externa obterá acesso aos dados armazenados nesta estrutura. Toda questão referente aos formatos de arquivo de imagem e à estrutura interna deve ser transparente à biblioteca. Para isto, precisamos fornecer a ela todos os métodos de acesso aos dados que ela necessitar. Com isto, continuamos garantindo a modularidade que queremos. E por terceiro e último, queremos resolver estes dois desafios sem perder eficiência.

No capítulo 2, fornecemos o contexto histórico e técnico necessários para que este trabalho seja bem entendido, explicando os principais conceitos utilizados e os maiores desafios enfrentados. Em seguida, no capítulo 3, discutimos a proposta, a estruturação e a implementação do componente de entrada e saída. Por fim, no capítulo 4 explicamos os resultados obtidos e no capítulo 5 fazemos as considerações finais e propomos os trabalhos futuros.

Capítulo 2

Revisão de Literatura

Neste capítulo iremos apresentar os principais conceitos por trás da proposta de desenvolvimento de um componente genérico para entrada e saída de imagens. Primeiramente, é explicada a técnica de programação genérica contextualizada com a área de estudo em processamento de imagens, de maneira a justificar os benefícios de sua utilização e demonstrar os meios necessários para alcançá-los. Em seguida, na seção 2.2, é mostrado o estado atual do armazenamento de imagens, juntamente com os problemas advindos da expansão na utilização de imagens no computador sem se tomar o cuidado necessário na padronização dos formatos dos arquivos.

Na seção 2.3 são explicados os conceitos do paradigma de programação baseado em componentes bem como sua importância no desenvolvimento de aplicações para processamento de imagens. Na seção 2.4 são apresentadas as atuais ferramentas encontradas na Internet que propõem abordagens similares às características que enfatizamos para este trabalho. E por fim, na seção 2.5, é mostrada a proposta do núcleo de uma ferramenta para processamento de imagens que fará uso das funcionalidades de nosso componente.

2.1 Programação Genérica e Processamento de Imagens

Nos últimos vinte anos os Computadores Pessoais alcançaram níveis de desempenho que os tornam cada vez mais interessantes para o processamento de imagens. As soluções em *software* possuem muitas vantagens importantes: são facilmente transferidas para a próxima geração de hardware, podem ser rapidamente adaptadas a novas necessidades, e podem ser reusadas em vários diferentes contextos e

sistemas.

Entretanto, não é muito simples alcançar estes objetivos na prática. O desenvolvimento atual de *software* em processamento de imagens está levando a programas que não são tão flexíveis e reusáveis quanto se gostaria que fossem. Köthe [KöTH 99] acredita que estes problemas se devem a duas dificuldades principais:

- **A Flexibilidade vs. Problema de Desempenho**
  Em muitos casos, a flexibilidade introduz uma sobrecarga em tempo de execução que degrada o desempenho. Dada a enorme quantidade de dados que precisa ser processada na maioria das aplicações de processamento de imagens, só existe um espaço muito limitado onde se pode trocar desempenho por flexibilidade. Assim, deve-se procurar por técnicas de flexibilidade que não afetem o desempenho.

- **O Problema do Acoplamento de *Frameworks***
  A maioria dos componentes reusáveis são atualmente fornecidos como partes de grandes *frameworks*. Já que componentes individuais nestes *frameworks* estão fortemente "amarrados" uns aos outros, não é possível tirar apenas aquelas peças que realmente precisamos: o *framework* só funciona como um todo. Isto essencialmente impede o uso de componentes de vários *frameworks* em um mesmo projeto. Assim, precisa-se fornecer *software* reusável na forma de blocos de construção independentes que possam ser reusados individualmente.

Por exemplo, suponha que tivéssemos construído uma aplicação de processamento de imagens à qual gostaríamos de adaptar as necessidades especiais de um importante cliente. O corpo funcional existente é grande, tal que não podemos nos dar ao luxo de jogá-lo fora e começar tudo do zero de novo. Para preencher rapidamente as peças faltantes, gostaríamos de reusar algumas funcionalidades que foram desenvolvidas em outro contexto, talvez até de outro projeto ou domínio público.

Suponha, agora, que queiramos incorporar ao nosso ambiente um filtro morfológico que é parte de outro sistema. Infelizmente, este algoritmo só funciona com uma variedade específica de formato de imagem deste sistema. Se acontecer de não usarmos este formato de imagem em nosso ambiente, teremos um problema de adaptação. Este problema é tradicionalmente resolvido das seguintes maneiras:

- Podemos converter nossa representação de imagem original para o formato exigido pelo outro sistema antes de toda aplicação do algoritmo, e converter os

resultados de novo quando completados. Esta é a solução geralmente aplicada dentro do paradigma de software baseado em componentes. Visivelmente, converter é relativamente ineficiente, porque rouba tempo e memória. Assim, queremos antes evitar isto nos níveis mais baixos de um sistema.

- Para minimizar conversão, podemos optar por construir um módulo inteiro usando a funcionalidade do outro sistema, isto é, além dos filtros morfológicos também usamos as funções deste sistema para as operações de pré e pós-processamento necessárias. Embora isto possa funcionar em alguns casos, não é uma boa idéia usar vários *frameworks* diferentes dentro do mesmo projeto: isto leva a um crescimento explosivo no tamanho do código, e exige que se entenda e mantenha todos estes *frameworks*, o que pode facilmente anular o efeito positivo da reutilização de software.

- Se nós tivermos acesso ao código fonte, também podemos tentar modificar o código de modo a adaptá-lo às necessidades do nosso ambiente. Esta abordagem é muito problemática, já que ela facilmente introduz *bugs* discretos que são muito difíceis de serem detectados. Para evitar enganos, um entendimento muito completo do código existente é necessário, o que é considerado ainda mais difícil do que reimplementar a solução do zero.

Nenhuma destas opções é realmente satisfatória. Seria muito melhor se componentes reusáveis fossem fornecidos como **blocos de construção independentes** que não dependessem de um ambiente específico e pudessem ser adaptados sem o conhecimento de seu funcionamento interno (sem modificação do código fonte) e sem perda de desempenho.

Uma abordagem de projeto alternativa que fornece um suporte muito bom para definição e integração flexível de blocos de construção independentes é chamada de "Programação Genérica". Esta abordagem complementa os outros métodos de maneira que nos permite projetar soluções combinadas (como *frameworks* orientados a objeto delegando suas implementações para componentes genéricos) que enfatizam os pontos fortes de cada método.

A dependência mais importante que devemos remover é a dependência de algoritmos sobre representações de dados específicas. De certa maneira, a programação orientada a objetos fornece uma solução para este problema se os dados forem acessa-

dos através de classes abstratas ou funções virtuais. A figura 2.1 mostra um exemplo de código na linguagem de programação C++ que utiliza esta técnica.

```
class AbstractImage
{
    public:
        virtual unsigned char getPixel(int x, int y) const = 0;
        virtual void setPixel(unsigned char value, int x, int y) = 0;
        virtual int width() const = 0;
        virtual int height() const = 0;
        // ...
};
```

Figura 2.1: Exemplo de código em C++ que utiliza funções virtuais.

Algoritmos que usam esta classe abstrata de base não sabem o que está por trás das funções virtuais. Assim, podemos colocar formatos de imagem arbitrários nas subclasses desta imagem abstrata.

Entretanto, há dois problemas fundamentais com esta abordagem. Primeiro, a chamada à função virtual introduz uma sobrecarga em tempo de execução. Köthe [KöTH 99] mostra que o uso de funções virtuais, em detrimento do acesso direto aos dados através de ponteiros, pode levar a perdas de desempenho de até 500%. Claro, a sobrecarga poderia ser evitada se implementássemos algoritmos como funções-membro das classes concretas de imagem, dando a elas acesso direto aos dados. Mas isto iria contrariar nosso desejo por reusabilidade, já que estas funções-membro iriam novamente depender do formato concreto de imagem e não seriam independentemente reusáveis.

O segundo, e mais sério problema, é o tipo de retorno da função `getPixel()` no exemplo da figura 2.1. Em uma linguagem estaticamente tipada, este tipo de retorno precisa ser fixo ('`unsigned char`' no exemplo) para que imagens '`float`' e RGB não sejam subclasses de `AbstractImage`. Isto poderia ser superado introduzindo outra classe abstrata como `AbstractPixel`, mas isto exigiria ainda mais chamadas virtuais, de modo que esta solução não se apresenta como uma possibilidade prática. Assim, para tipos diferentes de retorno de `getPixel()`, ainda precisamos de diferentes versões de cada algoritmo.

Embora organizar objetos em hierarquias de classe tenha suas vantagens, em uma linguagem estaticamente tipada isto também introduz dependências estáticas,

isto é, todos tipos (classes de base, argumentos de funções, e dados-membro) e funções-membro que são mencionadas em uma classe devem também ser transferidas para o novo ambiente se quisermos reusar esta classe. Isto geralmente leva a uma cadeia de interdependências que é a principal causa do problema de acoplamento de *frameworks* mencionado anteriormente, e evita a definição de blocos de construção realmente independentes. Desta maneira, uma técnica adicional de projeto é necessária.

Tradicionalmente, os problemas mencionados acima têm sido verificados por geradores de código. Estes geradores produzem o código fonte para uma aplicação concreta a partir da implementação abstrata de um bloco de construção e de uma descrição do contexto (contendo, entre outras coisas, os tipos de dados concretos envolvidos).

Ao mesmo tempo, estas facilidades na geração de código estão sendo montadas diretamente dentro da maioria das linguages orientadas a objeto (como C++ e `ADA`) com o nome de generalidade e tipos paramétricos. O mais extensivo suporte a generalidade é dado pelo mecanismo de *templates* em C++. O suporte de generalidade difundido permitiu o desenvolvimento de um novo paradigma – programação genérica – que generaliza e formaliza as idéias de geração de código e assim resolve os problemas mencionados acima sem introduzir uma grande sobrecarga em tempo de execução.

Programação genérica tem sido introduzida como um paradigma de programação independente por Musser e Stepanov [MUS 93]. Ela se tornou popular quando foi escolhida como o paradigma implícito da Biblioteca Padrão C++, em particular a parte da biblioteca que é conhecida como *Standard Template Library* [1] [STE 95]. Ela é especialmente ajustada para implementar estruturas de dados e algoritmos reusáveis nos níveis mais baixos de um sistema. A seção 2.1.1 traz uma introdução geral sobre programação genérica.

2.1.1 Programação genérica

Usualmente, em nível de programação orientada a objetos, a chamada a métodos requer que os eventuais dados passados como argumento sejam de tipos compatíveis com os especificados na definição do método. Da mesma maneira, geralmente o desenvolvedor do programa de computador se vê obrigado a duplicar o código

[1] Em português, algo como *Biblioteca Gabaritada Padrão* (tradução livre).

do método e alterar os tipos de dados acessados, simplesmente para executar uma mesma rotina (implementada por um mesmo algoritmo) sobre tipos de dados diferentes. Por exemplo, um método que retornasse o maior entre dois números inteiros (tipo `int`) passados como parâmetro precisaria ter seu código alterado para que passasse a funcionar com números reais (`float`).

Neste ponto, a linguagem de programação C++ oferece uma técnica de auxílio ao programador. Ela permite que uma classe possa ter dois ou mais métodos sobrecarregados, ou seja, com o mesmo nome mas com tipos de parâmetros e de retorno diferentes [STR 97]. Assim, usando o exemplo acima citado, suponha que temos uma classe `Número`, que implementa o método chamado `Maior`, com a mesma funcionalidade do exemplo. Para usarmos números inteiros ou números reais sem nos preocuparmos com o tipo dos parâmetros quando chamarmos o método, implementamos na classe `Número` o método `Maior`, que recebe dois inteiros como argumento e retorna o inteiro de maior valor, e também outro método `Maior`, que recebe dois números reais e retorna o maior deles. Assim, em tempo de compilação, esta sobrecarga do método `Maior` será detectada e o compilador referenciará cada um deles corretamente, dependendo dos tipos dos argumentos nas suas chamadas. Esta técnica permite que o programador não se preocupe com qual método chamar se os tipos dos dados forem alterados, mas não evita que ele tenha que reescrever um método diferente para cada tipo de dado sobre o qual ele deseja operar.

Desta maneira, a noção de generalidade aparece na possibilidade de reuso de um determinado algoritmo, especificado para uma certa estrutura de dados. Entretanto, neste paradigma de programação, em nenhum momento desejamos desconsiderar aspectos presentes em outros paradigmas. Na verdade, desejamos assimilar as vantagens destes paradigmas anteriores e unificá-las a uma implementação genérica. Assim, aspectos presentes no paradigma orientado a objetos, como abstração, reuso e adaptabilidade, não são desconsiderados, e sim agregados a uma implementação genérica, enquanto que problemas, como a sobrecarga pelo uso de funções virtuais, são minimizados.

2.1.2 A linguagem C++ e a Biblioteca STL

A programação genérica oferece uma grande flexibilidade durante o desenvolvimento de uma aplicação, e um código eficiente em tempo de execução de um programa. Utilizando a linguagem de programação C++, podemos instanciar algo-

ritmos genéricos construindo classes que definem os tipos e as operações de acesso necessárias. Através do uso desta técnica e da cuidadosa atenção às questões de algoritmos, é possível construir componentes de *software* de larga utilização sem sacrificar eficiência [MUS 93].

O poder da programação genérica se tornou aparente com a STL (*Standard Template Library*), que agora é parte do padrão ISO C++, distribuída com todos compiladores C++ [FAB 2001]. A biblioteca STL é estruturada em termos de iteradores, contêiners e algoritmos [JOS 99, VIL 2001]. Os iteradores são utilizados para referenciar os elementos presentes nos contêiners, que são estruturas de armazenamento, de forma que um algoritmo possa manipulá-los. Deste modo, evitamos o uso de herança e funções virtuais, comuns na programação orientada a objetos [LIP 98], e optamos por uma programação genérica baseada em *templates* (gabaritos), objetivando um melhor desempenho de execução sem perda de generalidade [STE 95].

2.2 Formatos de arquivos de imagens

Além dos diversos tipos de representação de imagens, existem inúmeras maneiras destas representações serem armazenadas em um dispositivo de memória em massa. Como não existe uma padronização e nem um consenso quanto à melhor forma de se armazenar uma imagem em disco, cada grupo de interesse cria seu próprio formato. Isto dificulta o reconhecimento do tipo de formato do arquivo, uma vez que os dados são gravados das mais diversas maneiras, e também prejudica a conversão de uma formato para outro, já que os parâmetros e a representação utilizada variam muito, fazendo com que a qualidade da imagem seja degradada.

Uma das melhores formas hoje – embora delicada e sujeita a erros – de descobrir de que maneira um arquivo deve ser lido é através da análise da extensão do seu nome. Por exemplo, um arquivo com o nome `foto.ppm` nos diz que que a imagem está no formato `Portable Pixmap`, por causa da extensão `ppm`. Assim, nós sabemos o que esperar quando lermos este arquivo e quisermos interpretar seus dados.

Cada formato de arquivo de imagem contém suas vantagens e desvantagens [MUR 94], como espaço ocupado em disco e qualidade da imagem armazenada, mas a idéia aqui é fazer com que o acesso (leitura e gravação) a estes arquivos seja totalmente transparente a uma biblioteca externa.

Utilizando uma estrutura genérica, o problema agora consiste em implementar

uma interface responsável pela troca de dados entre o disco e esta estrutura. Tanto para leitura quanto para gravação, esta interface precisa ter procedimentos específicos para cada formato de arquivo, que são responsáveis por reconhecer corretamente a imagem armazenada.

Queremos também que, depois de decidir inicialmente sobre para quais formatos oferecer suporte, o reconhecimento de novos formatos possa ser facilmente adicionado. Partindo deste projeto flexível e expansível, bastaria-nos, então, implementar um novo procedimento que reconhecesse o novo tipo de arquivo. Esta idéia de modularidade é possível porque a estrutura de armazenamento interno será genérica e, portanto, apenas a maneira de reconhecer determinado formato de arquivo precisa ser implementada. Todos os outros métodos que atuam sobre a imagem serão independentes do arquivo que foi lido. Da mesma maneira, o armazenamento da imagem em memória, já processada, em um tipo específico de formato de arquivo, só dependeria da implementação da função que reconhecesse este tipo específico.

2.3 Componentes Genéricos em Processamento de Imagens

A definição de um componente é *"Aquilo que entra na composição de alguma coisa. Parte elementar de um sistema"* (Aurélio). O paradigma de componentes define componente de *software* como sendo uma *"Unidade de* software *com interfaces e dependências claramente especificadas, que pode ser implamantada independentemente e ser utilizada por terceiros para composição"* (C. Szyperski). Dentre as principais vantagens da utilização deste paradigma, temos que ele simplifica o desenvolvimento de sistemas, pois a complexidade necessária para executar uma tarefa é encapsulada por um componente, escondendo os detalhes internos do mundo externo.

Em processamento de imagens, as aplicações desenvolvidas tradicionalmente apresentam um alto grau de dependência quanto à sua finalidade e ambiente na qual foram implementadas. A principal razão para isto é a grande necessidade de eficiência no processamento de grandes quantidades de dados. Existe, ainda, uma complexidade implícita neste processamento quanto à interpretação semântica de uma imagem, ou seja, quanto ao seu conteúdo. Isto faz com que, na tentativa de minimizar esta complexidade, aplicações ainda mais dependentes do tipo de imagem

tratada sejam desenvolvidas [CAR 2001].

Percebemos, então, dois usos diretos da programação genérica no processamento de imagens. Primeiro, a representação das imagens de diversas naturezas pode ser generalizada com a definição de contêiners apropriados, de modo a permitir a representação e manipulação de imagens com tipos de dados diferentes, como valores de pixel inteiros, reais ou complexos, por exemplo. Segundo, a programação genérica permite o desenvolvimento de algoritmos para manipulação destes contêiners capazes de expressar uma lógica computacional de forma independente da imagem a ser tratada (ver [KöTH 99]).

Podemos vislumbrar, então, a estruturação de um conjunto de operandos (contêiners) e de operadores (algoritmos e iteradores) capazes de representar genericamente os mecanismos de manipulação, tratamento e análise de imagens. Como extensão natural destas aplicações conjugada ao pontencial de extensão e reuso fornecido pela programação genérica, podemos pressupor que estes mecanismos conseguem definir um novo conjunto de funções ou biblioteca que é facilmente adaptável a situações particulares e portável a ambientes variados; temos, então, um componente genérico de *software*.

2.4 Estado da Arte

Existem, atualmente, inúmeras bibliotecas para entrada e saída de imagens, que suportam os mais variadas tipos de formato de arquivo. Entretanto, poucas delas propõem um padrão de projeto diferenciado e justificável: a maioria ainda implementa o reconhecimento de diversos formatos em código fixo, e por este motivo tornam-se específicas a um só problema e têm utilidade apenas na aplicação para a qual foram desenvolvidas. Duas das poucas bibliotecas que abordam um padrão de projeto diferenciado de entrada e saída de imagens são a **ImageIO** [CHA 2001] e a **VIGRA** [KöTH 2002].

2.4.1 A biblioteca `ImageIO`

Chandra e Ibanez [CHA 2001] apresentam o desenvolvimento de um *framework* baseado nos critérios de portabilidade, modularidade, extensibilidade e transparência. A ênfase principal da biblioteca é dada ao critério de extensibilidade. Para alcançá-la, os autores usam o padrão de projeto de *software* conhecido

como *Pluggable Object Factory*[2].

Segundo este padrão [PUR 2002], os atores principais são um **cliente**, uma **fábrica** e um **produto**. O cliente é um objeto que requer uma instância de um outro objeto (o produto) por alguma razão. Ao invés de criar a instância do produto diretamente, o cliente delega esta responsabilidade para a fábrica. Uma vez invocada, a fábrica cria uma nova instância do produto, passando-o de volta para o cliente. Simplificando, o cliente usa a fábrica para criar uma instância do produto. A figura 2.2 mostra a relação lógica entre os elementos deste padrão.

Figura 2.2: Modelo lógico do padrão de projeto de fábrica

A fábrica abstrai completamente a criação e inicialização do produto pelo cliente. Esta indireção permite que o cliente se foque em seu papel discreto sem se preocupar com os detalhes de como o produto é criado. Assim, à medida que a implementação do produto muda com o tempo, o cliente permanece inalterável.

Desta forma, usando este padrão de projeto, a biblioteca citada instancia dinamicamente a subclasse apropriada que reconhece um determinado formato de arquivo. Toda vez que o suporte a um novo formato de arquivo é necessário, uma subclasse de `ImageIO` é derivada para reconhecer os detalhes de implementação deste formato. Assim, o suporte a novos formatos de arquivo pode ser adicionado por terceiros, e até mesmo integrado nas aplicações durante a execução através de ligação dinâmica, usando DLL's (*Dynamic Link Libraries*[3]).

Entretanto, apesar da grande extensibilidade e facilidade de manutenção oferecidas, um dos pontos fracos desta biblioteca se refere à eficiência. O uso de ligação dinâmica para a instanciação das subclasses penaliza seriamente o desempenho da aplicação durante sua execução, pois esta precisa carregar na memória as bibliotecas que forem sendo necessárias em tempo de execução. Entretanto, é verdade que, no caso de várias destas aplicações estiverem sendo executadas ao mesmo tempo, apenas uma cópia de cada biblioteca dinâmica será colocada na memória. Mesmo assim, queremos principalmente ganhar desempenho de processamento, e não quantidade

[2] Do inglês, "Fábrica de Objetos Ligáveis" (tradução livre).
[3] Do inglês, *Biblioteca de ligação dinâmica.*

de memória.

Além disto, a eficiência também é prejudicada por que três cópias dos mesmos dados são usadas antes que eles estejam finalmente prontos para uso pela aplicação. Isto pode se tornar um problema quando tratarmos com imagens muito grandes. Mas dependendo da aplicação na qual a biblioteca é usada, as operações de entrada e saída podem ser relativamente infrequentes, já que geralmente a imagem é carregada na inicialização do programa e então salva no disco quando o processamento termina. Assim, conforme seus autores [CHA 2001], esta penalidade é amenizada.

Embora este argumento seja bastante plausível, e apesar de desejarmos possibilitar o uso de nosso componente nos mais diversos contextos e ambientes, estamos especialmente interessados na sua utilização em aplicações de processamento de imagens e visão computacional. Estas aplicações, em particular, usualmente fazem um grande uso de operações de entrada e saída de imagens como, por exemplo, as na área de robótica, onde diversas imagens precisam ser carregadas em questão de segundos. Nosso foco, então, também abrange o critério da eficiência.

2.4.2 A biblioteca `VIGRA`

Köthe apresenta "The VIGRA Computer Vision Library"[4], bastante documentada em [KöTH 2002]. Segundo ele,

> "`VIGRA` deriva de 'Visão através de Algoritmos Genéricos'. É uma biblioteca de visão computacional que coloca sua ênfase principal em algoritmos e estruturas de dados personalizáveis. Usando técnicas de *template* similares àquelas da STL do C++, pode-se facilmente adaptar qualquer componente `VIGRA` às necessidades de uma aplicação, sem, entretanto, desistir da velocidade de execução." [5]

Sua tese de doutorado [KöTH 2000] contém a descrição detalhada do projeto de `VIGRA`. As principais idéias por trás dela também são descritas em [KöTH 99, KöTH 2000a].

[4] Do inglês, *A Biblioteca de Visão Computacional VIGRA* (tradução livre).

[5] No original,

> *VIGRA stands for "Vision with Generic Algorithms". It's a novel computer vision library that puts its main emphasize on customizable algorithms and data structures. By using template techniques similar to those in the C++ Standard Template Library, you can easily adapt any VIGRA component to the needs of your application, without thereby giving up execution speed.*

Basicamente, apesar de `VIGRA` se tratar de uma biblioteca, e não de um componente apenas, as características de seus métodos para entrada e saída de imagens se assemelham às características que queremos para o nosso componente. Como sua principal ênfase está em flexibilidade, a biblioteca foi construída utilizando programação genérica [STE 89] e, assim, permite que seus algoritmos sejam usados em cima de outras estruturas de dados, dentro de outros ambientes. Mas esta flexibilidade não é custosa, já que seu projeto usa *templates* (polimorfismo em tempo de compilação) e assim seu desempenho se aproxima da solução não genérica, inflexível. Como o componente que apresentamos também faz uso de *templates* e programação genérica, esperamos obter esta mesma flexibilidade conjungada com eficiência.

Entretanto, além de desejarmos possibilitar o uso deste componente em diversificados sistemas, principalmente da área de visão computacional, queremos especialmente fornecer suporte a ferramentas de processamento de imagens que utilizam os conceitos da álgebra de imagens em suas implementações. Estas ferramentas, segundo proposto por esta álgebra, usam o conceito de **reticulados**. Assim, para permitir a utilização deste conceito, nossa comunicação com o ambiente externo se dará principalmente por meio de contêiners genéricos. Maiores detalhes sobre esta especificação são apresentados no capítulo da metodologia deste trabalho, na seção Detalhes de Implementação (página 24).

2.5 O Núcleo de uma Ferramenta para Processamento de Imagens

Através da utilização de programação genérica, e enfatizando os conceitos de flexibilidade e reusabilidade, D'Ornellas e Carrard [CAR 2001] propõem o desenvolvimento de uma ferramenta de *software* para processamento de imagens projetada conforme a figura 2.3. Nesta figura está representada a necessidade de desenvolvimento de um núcleo base sobre o qual serão implementadas as bibliotecas, interfaces e aplicações epecíficas. Eles propõem também, como objetivo principal, o desenvolvimento deste núcleo da ferramenta, estruturando-o em três níveis operacionais distintos e construindo-o em camadas, conforme a figura 2.4.

A camada mais interna do núcleo é chamada de **base**, e tem por objetivo definir as estruturas de dados para as informações, seus mecanismos de armazenamento e recuperação, e as garantias de consistências que o desenvolvimento possa necessitar.

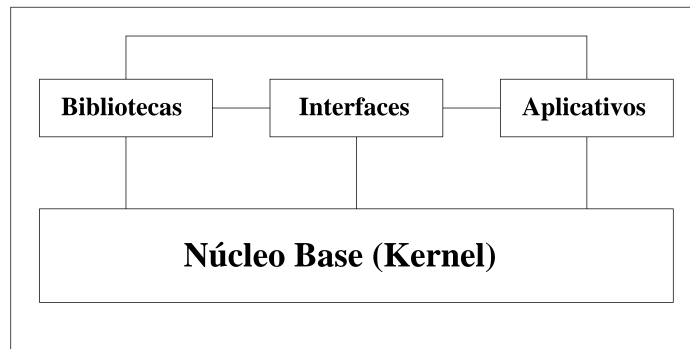

Figura 2.3: Proposta de arquitetura da ferramenta para processamento de imagens

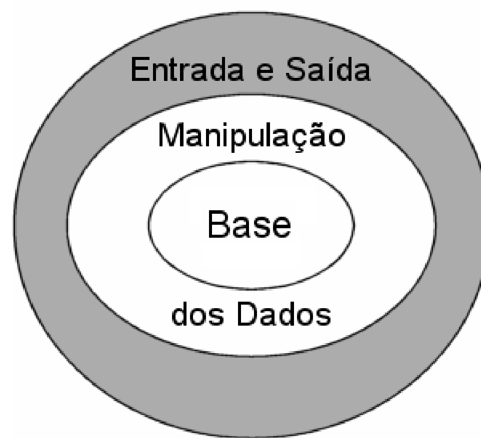

Figura 2.4: Estrutura do núcleo da ferramenta

A camada do meio, chamada de **manipulação dos dados**, tem por finalidade fornecer o conjunto de operações elementares com os dados armazenados, de forma a criar uma base teórica e prática para expressar um grande número de algoritmos presentes em processamento e análise de imagens. Por fim, a camada mais externa, de **entrada e saída**, é projetada como a interface entre as operações elementares com os dados e as aplicações, bibliotecas e interfaces que possam ser desenvolvidas em torno do núcleo.

Enquanto D'Ornellas e Carrard [CAR 2001] se concentram na especificação do projeto e no desenvolvimento das duas camadas mais internas, o objetivo principal deste trabalho é constituído pelo projeto e desenvolvimento da camada mais externo do núcleo. No próximo capítulo apresentamos os detalhes destas definições.

Capítulo 3

Metodologia

Neste capítulo discutimos as características essenciais para o desenvolvimento de nosso componente e também de que maneira conseguimos alcançá-las durante a implementação do mesmo. Na seção 3.1, além de expormos as principais dificuldades enfrentadas no desenvolvimento de bibliotecas para processamento de imagem, mostramos as principais propriedades que desejamos incorporar ao nosso componente para que estes problemas sejam solucionados. Na seção 3.2 apresentamos a definição estrutural do componente, e na seção 3.3 debatemos os principais detalhes de implementação das classes, os problemas encontrados durante seu desenvolvimento, o estado atual do componente e as ferramentas utilizadas.

3.1 Desenvolvimento do Componente de Entrada e Saída

O problema enfrentado ao implementarmos uma biblioteca para processamento de imagens é a quantidade expressiva de formas de representação de imagens, por que os seus tipos de valores dos pixels podem ser os mais diversos possíveis, indo desde números inteiros ou reais até números complexos ou estruturas RGB. Desta maneira, a maioria destas bibliotecas acaba se especializando em um determinado tipo de representação, enquanto deixa os outros de lado [CAR 2001]. Isto faz com que tenhamos que utilizar uma biblioteca específica para cada tipo de imagem que desejamos [D'OR 2001], o que pode ser um grande inconveniente quando queremos trabalhar sobre diversos deles.

A idéia da generalidade se faz presente na medida em que queremos obter uma maneira de oferecer métodos de acesso à imagem de maneira homogênea, indepen-

dente do tipo de representação dela. Por este motivo, queremos implementar uma estrutura de dados genérica que seja capaz de representar qualquer tipo de imagem da maneira mais transparente possível. A conjunção destas duas características nos define o termo ”componente genérico”, na medida em que queremos obter uma estrutura encapsulada que forneça os métodos de acesso necessários e funcione da mesma maneira nas mais diversas situações.

O objetivo, então, é permitir que um único componente seja necessário para que obtenhamos acesso a quaisquer imagens. Para que isto aconteça, é necessário que ele seja solidamente construído, de modo que possa ser usado para o desenvolvimento de aplicações específicas sem que haja a necessidade de realizar adaptações em seu código, principalmente no que diz respeito aos tipos de dados manipulados, suas estruturas de armazenamento e seus métodos de acesso. Assim, devem ser estabelecidas algumas prioridades básicas no seu projeto. Relacionamos a seguir algumas características desejáveis ao componente, com base nas citadas por [CAR 2001] no desenvolvimento do núcleo da ferramenta:

- *Exatidão*: as descrições das soluções algorítmicas propostas devem ter sólido embasamento teórico e devem se comportar de acordo com a especificação presente nesta teoria;

- *Robustez*: o comportamento do componente deve ser estável, mesmo frente a casos especiais e degenerados de uso;

- *Generalidade*: por causa da diversidade dos tipos de dado e das necessidades nas aplicações de processamento de imagens, as soluções propostas aos problemas devem ser genéricas o suficiente para buscarem se sobrepor a estas especificidades. Para isto, três mecanismos são de fundamental importância: propiciar alto nível de abstração de dados; fornecer mecanismos de intercâmbio e tratamento de informações e; utilizar a programação genérica no desenvolvimento do componente, principalmente como mecanismo de reuso;

- *Flexibilidade*: procuramos o projeto de um componente que, sendo flexível por definição, estenda esta característica às aplicações, bibliotecas e interfaces que o utilizam. Como características desta flexibilidade desejamos:

  - Permitir que o componente trabalhe por partes ou módulos (modularidade);

  - Propiciar fácil adaptação a todo tipo de situação (adaptabilidade);
  - Permitir fácil ampliação do componente e seus recursos (extensibilidade);
  - Ter comportamento e uso uniforme (uniformidade);
  - Se adequar aos mais variados tipos de plataformas de *hardware* e *software* (portabilidade);
  - Ser escrito nos moldes dos *softwares* de código aberto e livre, para propiciar a rápida difusão no meio acadêmico e garantir a continuidade e ampliação do trabalho de pesquisa envolvido (*software* livre).

- *Eficiência*: podemos definir a eficiência de um *software* como o grau de utilização de recursos computacionais que o mesmo faz para solução dos problemas aos quais se propõe. Em processamento de imagens, devido à dependência da solução desejada para com os resultados imediatos dos algoritmos, buscamos algoritmos que sejam eficientes em tempo e espaço, de modo a produzir os resultados desejado em condições factíveis.

Definidas as prioridades básicas do projeto, partimos para a implementação do componente. Usando técnicas de programação genérica da linguagem C++, em conjunto com algoritmos genéricos disponibilizados pela STL, procuramos alcançar os objetivos do projeto de acordo com estas prioridades.

3.2 Definição da Estrutura

De acordo com a hipótese proposta previamente no projeto deste trabalho, o componente é dividido em três camadas, ou níveis, com funções diferentes. Primeiro, o componente precisa ter acesso ao meio de armazenamento em massa do sistema no qual está instalado, ou seja, ele deve possuir uma interface para acessar os arquivos de imagem contidos no disco. Segundo, as informações armazenadas no componente precisam ser disponibilizadas para acesso externo, isto é, o componente precisa fornecer a interface necessária para fazer a comunicação com uma biblioteca externa. E terceiro, a última camada necessária é a que armazenará as informações, e que ficará contida entre as outras duas camadas. Esta camada contém a estrutura de armazenamento em memória da imagem lida e disponibiliza os dados para as duas interfaces. A figura 3.1 mostra esta primeira estrutura considerada na implementação do componente.

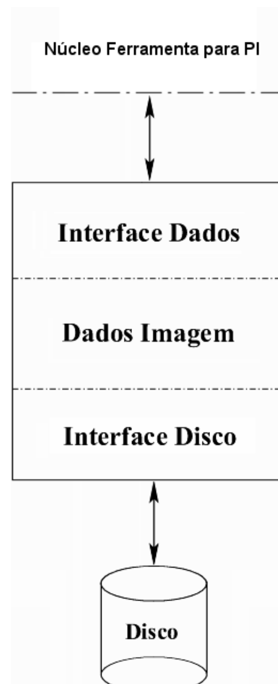

Figura 3.1: Organização interna previamente proposta do componente

Entretanto, à medida que prosseguimos no desenvolvimento do componente, algumas dificuldades foram encontradas, que fizeram com que esta estruturação tivesse que ser repensada e reconsiderada. A classe que implementava a interface de acesso a disco precisava fornecer métodos de reconhecimento para cada formato de arquivo de imagem desejado e, como cada formato representa os dados de maneiras diferentes, também precisava fornecer rotinas necessárias para converter estes dados brutos lidos e armazená-los na estrutura genérica (veja a figura 3.2). Mas isto iria inserir uma penalidade muito grande na flexibilidade do componente, pois a interface inteira precisaria ser modificada para inserir o suporte a um novo formato de arquivo, assim como prejudicaria a modularidade do componente.

Então, para evitar a sobrecarga da classe de acesso a disco, e para conseguirmos melhorar a modularidade, decidimos que, para cada formato de arquivo de imagem que quiséssemos ler ou escrever, uma nova classe deveria ser criada. Desta maneira, a classe de acesso a disco não conversaria diretamente com os arquivos, mas ao invés disto escolheria uma destas novas classes para os acessar, de acordo com o desejado.

Para resolver o problema da flexibilidade, decidimos retirar algumas questões de implementação das classes de disco e externa. Ao invés delas implementarem

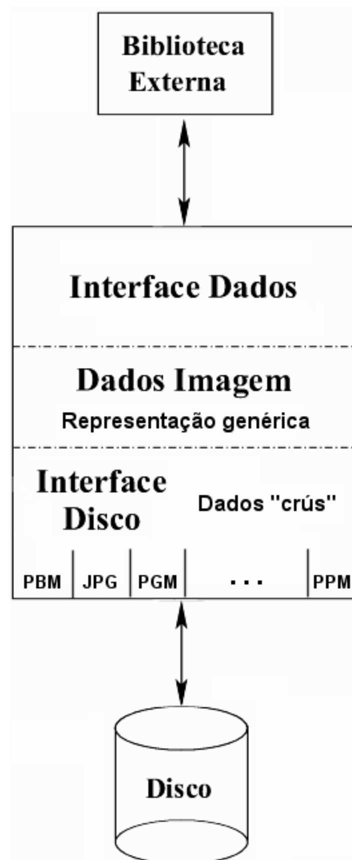

Figura 3.2: Estrutura prévia do componente com o problema de modularidade e flexibilidade

os métodos para acesso à estrutura interna, a própria estrutura forneceria estes métodos, sendo, portanto, implementada na forma de uma classe. Dessa maneira, as classes de disco e externa precisariam apenas invocar os métodos desejados desta nova classe de armazenamento.

Inicialmente, esta solução funcionou bem. Como agora a classe de armazenamento encapsulava as informações da imagem na memória, as classes de acesso externo e a disco não precisavam se preocupar com a forma de implementação da estrutura interna; apenas necessitariam chamar os métodos relativos a cada tipo de informação que necessitassem. Assim, melhoramos o componente nos quesitos flexibilidade e modularidade.

Entretanto, para que esta forma de armazenamento e esta classe interna continuassem transparentes e escondidas de módulos externos, a classe interna foi instanciada dentro destas outras classes. Para que uma única instância da classe interna fosse necessária, teríamos que compartilhá-la através da utilização de herança na

implementação das classes de acesso. Mas esta técnica nos traria perda de eficiência. Desta maneira, a definição da classe interna ficou sendo a mesma para as outras classes, mas cada uma delas continha a sua própria instanciação da classe interna. Para trocar informações bastava, então, passar a classe de armazenamento como parâmetro de uma para outra classe de acesso. A figura 3.3 ilustra esta estrutura.

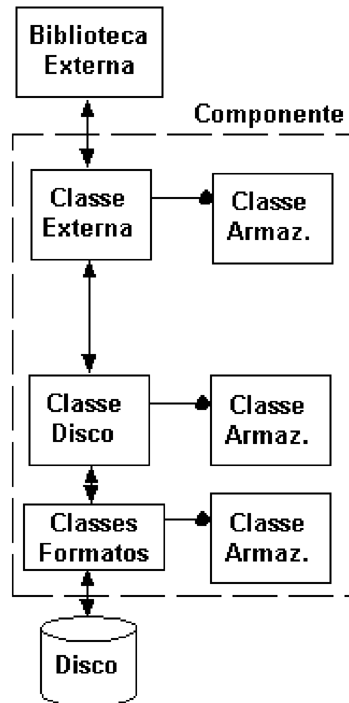

Figura 3.3: Estrutura com a classe de armazenamento.

Entretanto, como a classe de armazenamento era instanciada no momento da instanciação das outras classes, várias cópias da mesma imagem existiam ao mesmo tempo na memória. Quando fôssemos trabalhar com grandes quantidades de dados, com imagens grandes, diversas cópias destas informações seriam proibitivas em termos de desempenho[1]. A eficiência não deveria se limitar apenas a pequenas quantidades de dados. De qualquer maneira, mesmo não desejando usar herança, acabamos por perder eficiência.

Precisando resolver estes problemas da melhor forma possível, decidimos mudar de abordagem. Primeiro, para resolver o problemas das cópias, abandonamos a classe de armazenamento – de maneira que nenhuma outra classe continuou pos-

[1] Um problema semelhante a este é enfrentado por Chandra e Ibanez [CHA 2001] no seu projeto da biblioteca `ImageIO` (veja seção 2.4).

suindo uma instância dela – e mesclamos sua estrutura com a classe de acesso a disco. As informações da imagem e a própria imagem ficaram, então, contidas nela.

Em seguida, para fornecer acesso às informações armazenadas, a classe de acesso externo se transformou em uma interface para a classe de acesso a disco. Os métodos invocados pela biblioteca externa seriam redirecionados para a classe de acesso a disco, que executaria, então, a operação requisitada.

Para acessar os arquivos de imagem, a classe de acesso a disco instanciaria uma classe específica que reconhecesse o formato desejado, e esta classe específica retornaria as informações pedidas. Assim, de fato, a classe de acesso a disco se tornou uma classe interna ao componente, de maneira que não mais era possível acessá-la diretamente por meio externo. Esta estrutura pode ser melhor visualizada na figura 3.4.

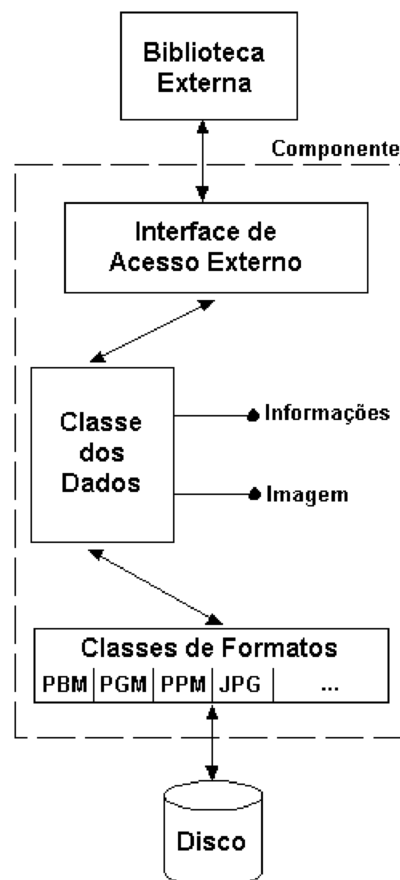

Figura 3.4: Estrutura definida para o componente.

Novas classes de acesso aos formatos de arquivos de imagens podem ser acrescentadas à medida que desejamos, bastando para isto implementar as funções de

entrada e saída específicas e utilizar os métodos da classe dos dados para trocar informações.

Exemplificando, a interface de acesso externo recebe os dados de fora e os passa diretamente para a classe dos dados, que por sua vez os armazena em seus atributos. Quando for exigida a gravação das informações, elas são enviadas para a classe do formato especificado, que depois de gravá-las conforme requisitado é destruída.

A próxima seção explica os detalhes de implementação referentes a esta estruturação. São mostrados e explicados os meios como o componente atinge satisfatoriamente os quesitos de generalidade e flexibilidade através de técnicas da linguagem C++ e uso de algoritmos da biblioteca padrão.

3.3 Detalhes de Implementação

A classe que contém os dados e a de acesso externo, citadas na seção anterior, foram chamadas respectivamente de `LatticeRepresentation` e `LatticeInterface`. Usamos esta nomenclatura em razão da proposta da ferramenta de processamento de imagens (veja seção 2.5) em utilizar o conceito de "reticulados" (*Lattice*, em inglês), que serve essencialmente como uma técnica matemática para representar algoritmos [D'OR 2001]. Este conceito deverá ser amplamente utilizado pela ferramenta citada, e já que nosso componente proposto irá fazer parte da terceira camada do núcleo desta ferramenta, optamos por utilizar uma nomenclatura semelhante. A proposta da ferramenta também utiliza os conceitos de Álgebra de Imagens, que segundo Ritter e Wilson [RIT 2000], é a teoria matemática concentrada na transformação e análise de imagens.

Foram implementadas diversas classes para que pudéssemos reconhecer e obter acesso a diferentes arquivos de imagens. Estas classes foram colocadas entre o disco e a classe `LatticeRepresentation`, conforme mencionado na seção anterior. Isto pode ser visto na figura 3.5, onde estas classes estão representadas pelo retângulo `FileFormats`. Ademais, esta figura nos auxiliará na compreensão do funcionamento das demais classes do componente.

3.3.1 Formatos de Arquivo de Imagem Suportados

No estado atual do componente, quatro formatos de arquivos de imagens são suportados e estão funcionais. O formato Portable Pixmap, de arquivos com ex-

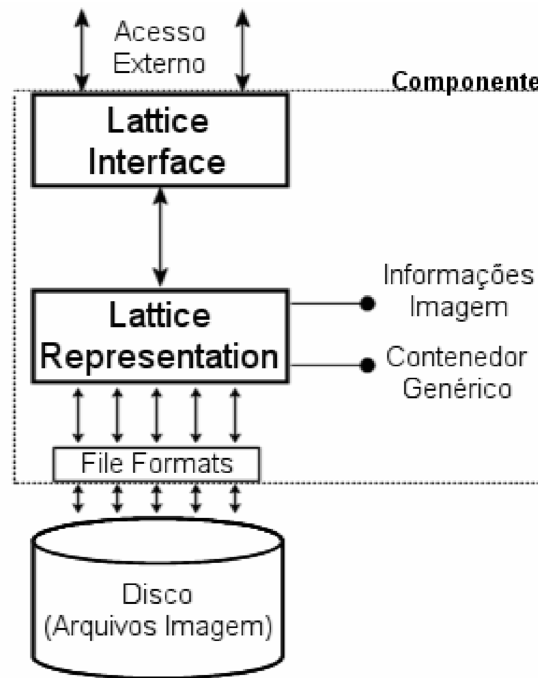

Figura 3.5: Respectivas classes na estrutura do componente

tensão pbm, é usado para armazenar imagens binárias, ou seja, em preto e branco, enquanto que o formato Portable Graymap (extensão pgm) é utilizado para representar imagens em tons de cinza.

Um terceiro formato é o Portable Pixmap (ppm), que pode armazenar imagens coloridas no padrão RGB [2]. Estes três formatos anteriores possuem várias propriedades em comum. A principal delas é um número mágico no cabeçalho do arquivo indicando o modo de armazenamento dos dados da imagem, que pode ser ou binário, onde cada byte contém o valor de um pixel ou de um componente do pixel, ou ASCII, isto é, caracteres literais representando os valores dos pixels.

Um quarto tipo de arquivo de imagem, e um dos mais utilizados atualmente, é o JPEG (extensões jpg e jpeg). Este formato suporta compressão de dados e é muito bom para representar imagens de fotos e de paisagens. Possui vários parâmetros ajustáveis que permitem escolher entre qualidade maior ou compressão melhor.

[2] **RGB**: *Red Green Blue* (Vermelho Verde Azul); a cor de um pixel é definida pela intensidade de cada uma destas três cores.

3.3.2 Implementação das Classes

A classe `LatticeInterface` possui os métodos para acesso externo à imagem armazenada em `LatticeRepresentation`. A requisição de um determinado dado da imagem, realizada por uma ferramenta utilizando este componente, é feita para a `LatticeInterface`, que por sua vez faz esta requisição à `LatticeRepresentation` e retorna o valor recebido.

A classe `LatticeRepresentation` instancia a classe correta quando se desejar acessar algum dos formatos de arquivo em disco. Dependendo da solicitação recebida de `LatticeInterface`, o acesso pode ser ou de leitura ou de escrita. As figuras 3.6 e 3.7 mostram as operações executadas para cada um destes casos. No caso da leitura (figura 3.6), `LatticeRepresentation` analisa a extensão do arquivo solicitado, verifica se esta extensão é reconhecida, instancia a classe correspondente, requisita a leitura dos dados, e os armazena em seus atributos. No caso da escrita (figura 3.7), `LatticeRepresentation` recebe uma requisição de `LatticeInterface` informando, entre outros dados, o formato do arquivo a ser gravado; ela instancia, então, a classe correspondente a este formato de arquivo, abre o arquivo informado, e invoca os métodos de escrita desta classe, enviando os parâmetros necessários.

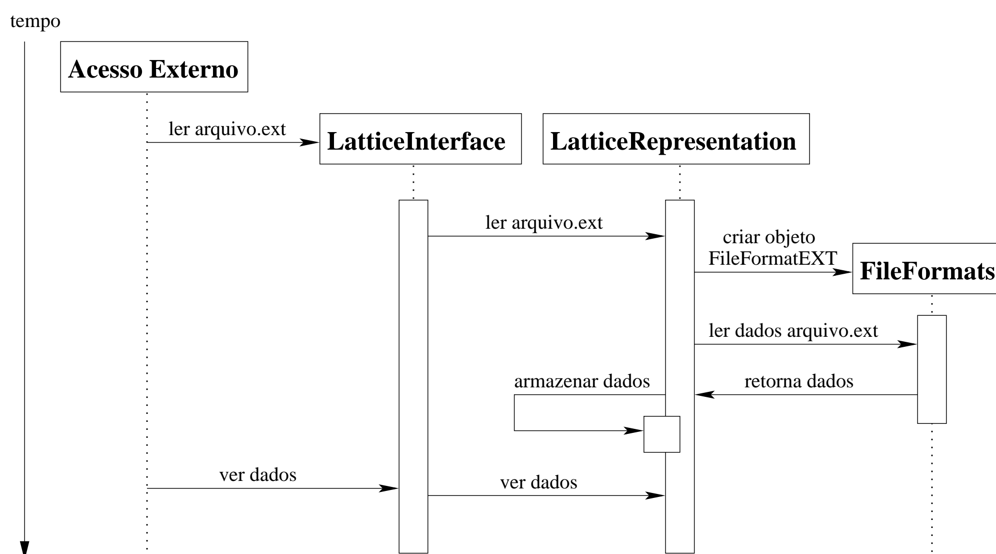

Figura 3.6: Diagrama de sequência na leitura de um arquivo

A instanciação da classe de formatos de arquivo correta se dá através do uso de *templates*, técnica oferecida pela linguagem C++ [STR 97]. Com isto, conseguimos

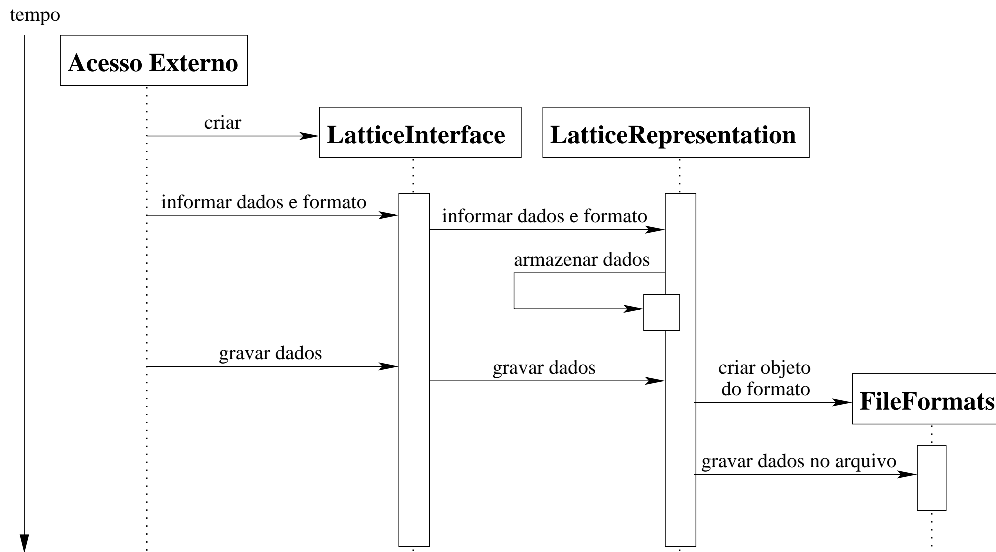

Figura 3.7: Diagrama de sequência na gravação de um arquivo

obter uma importante generalização e flexibilização do componente, de maneira que podemos utilizar apenas um método tanto para instanciar o objeto da classe requisitada quanto para invocar os métodos de leitura e gravação deste objeto.

Por exemplo, a classe `LatticeRepresentation` recebe uma requisição para que faça a leitura de um arquivo pbm. Ela verifica isto e então chama seu método gabaritado[3], fornecendo o tipo da classe a ser instanciada – neste caso, `FileFormatPBM`, responsável pelo reconhecimento do formato PBM. Este método gabaritado instancia a classe `FileFormatPBM` e invoca seus métodos de leitura, armazenando os valores retornados nos atributos correspondentes de sua classe, `LatticeRepresentation`. O código da figura 3.8 mostra um exemplo simples de como funciona este método gabaritado.

Note, no exemplo, que sempre os mesmos métodos são chamados, independentemente da classe que 'T' venha a ser. Por isto todas as classes de formato de arquivo devem possuir métodos com estes mesmos nomes. Mas como cada formato de arquivo de imagem possui alguns parâmetros diferentes dos demais, alguma especialização se torna necessária para que saibamos como acessar este valores específicos.

[3]Em inglês, *templatized*; que utiliza *templates* (tradução livre).

```
// Metodo generico para ler um arquivo de imagem.
// 'T' sera o nome da classe que deve ser instanciada,
template <typename T>
void LatticeRepresentation::readFile()
{
    T fileObj();            // instancia um objeto da classe 'T'
    T.openFile();           // chama os metodos desta classe
    _info = T.getInfo();    // .
    _image = T.getImage(); // .
}
```

Figura 3.8: Exemplo de método utilizando o mecanismo de *templates*.

Felizmente, o uso de *templates* não impede a especialização e a sobrecarga de métodos [D'OR 2002]. Assim, caso um arquivo `pbm`, por exemplo, tenha um dado específico que precise ser lido, podemos criar um novo método para ler dados extras utilizando *templates*, especializá-lo para a classe `FileFormatPBM`, e chamá-lo a partir do método genérico de leitura. Com base no algoritmo da figura 3.8, o código com especialização fica como na figura 3.9, permitindo que pequenas particularidades dos formatos de arquivos sejam tratadas de forma genérica. Entretanto, a especialização nos trouxe alguns problemas de implementação, que serão explicados na seção 3.3.4.

3.3.3 Estrutura Interna em Memória

Para armazenamento da imagem em memória, fizemos uso de um contêiner. Para que alcançássemos o nível de generalidade desejado, utilizamos um contêiner genérico, capaz de armazenar qualquer tipo de dado definido e reagir da mesma maneira em todos os casos. Desta forma, abstraímos qualquer tipo de representação de imagem, de maneira que não é necessário implementarmos um algoritmo diferente para cada tipo específico.

O contêiner utilizado dentro deste componente é fornecido pela STL (veja [STE 95] e seção 2.1.2), e é chamado de `vector`. Este contêiner possui métodos para acesso a qualquer posição do vetor de elementos, métodos para inserir elementos no meio e no final do vetor e métodos para remover um elemento de qualquer posição. O exemplo da figura 3.10 mostra alguns destes métodos.

Exatamente pelo fato do mecanismo de *templates* ser chamado de polimorfismo estático é que o tipo dos dados a ser usado no contêiner precisa ser conhecido em

```
template <typename T>
void readFile()
{
    T fileObj();          // instancia um objeto da classe 'T'
    T.openFile();         // chama os metodos desta classe
    _info = T.getInfo(); // .
    getExtraData(T);    // chama metodo para pegar dados extras,
                        // passa o tipo da classe como parametro
    _image = T.getImage();
}

// Metodo para ler dados adicionais de um arquivo
template <typename ClassName>
void getExtraData(ClassName &fileObj)
{
    cout << "Comandos gerais que valem para qualquer "
         << "classe nao especializada." << endl;
}

// Especializacao do metodo acima para ler um valor extra se
// o arquivo for do formato pbm.
template <>
void getExtraData(FileFormatPBM &fileObjPBM)
{
    // Le valor extra do arquivo
    _extraData = fileObjPBM.getExtraValue();

    cout << "Este comando e' chamado quando o objeto T de"
         << "readFile() for a instanciacao da classe "
         << "FileFormatPBM." << endl;
}
```

Figura 3.9: Exemplo de especialização de método que utiliza o mecanismo de *templates*.

tempo de compilação. Assim, qualquer estrutura para os valores dos pixels da imagem pode ser usada, desde que seja especificada antes da instanciação do contêiner. Como os métodos de acesso aos elementos permanecem os mesmos, conseguimos atingir um alto grau de generalização.

```
void funcao(int elem)
{
    vector<int> vetorInt;
    vetorInt.push_back(elem);
    vetorInt.push_back(elem+2);
    cout << "Primeiro elemento: " << vetorInt[0] << endl;
    cout << "Último elemento: " << vetorInt[1] << endl;
}
```

Figura 3.10: Exemplo de utilização do contêiner `vector` e alguns de seus métodos

3.3.4 O Problema da Especialização de Métodos com *Templates*

Conforme dito anteriormente, é a classe `LatticeRepresentation` quem possui o contêiner genérico que armazena a imagem em memória. Quando um objeto desta classe é criado, o contêiner também é automaticamente instanciado, já que faz parte dos atributos privados da classe. Ora, de acordo com a seção anterior, precisamos saber, antes que o contêiner seja instanciado, qual é o tipo de dado a ser guardado nele. Portanto, precisamos conhecer este tipo de dado no momento da instanciação da classe.

Para isto, a classe `LatticeRepresentation` também precisa ser gabaritada (*templatized*). E para saber previamente qual o tipo de retorno dos métodos de acesso ao contêiner, também a classe `LatticeInterface` e as demais classes de formatos de arquivos precisam utilizar *templates*.

A biblioteca externa que utilizar o componente é quem deve primeiro fornecer o tipo de dados da imagem. Desta maneira, como a primeira classe do componente a ser instanciada é a `LatticeInterface`, já teremos o tipo definido para o restante das classes, pois elas são instanciadas a partir da `LatticeInterface`.

Para implementar as classes com *templates*, tivemos, também, que fazer algumas mudanças na implementação de seus métodos. Usando um exemplo em código C++, as diferenças na implementação são mostradas nas figuras 3.11 e 3.12.

Veja que precisamos acrescentar a diretiva *template* em todos os protótipos dos métodos da classe gabaritada. Isto seria apenas uma questão de escrita de código, se não fosse o fato de que também queremos métodos gabaritados dentro da classe. Ainda assim, basta apenas manter a diretiva template do método gabaritado,

```
// classe que contem um vetor de numeros reais
class vetorDados {
    private:
      vector<float> dados;
    public:
      int tamanho();
}

// retorna a quantidade de numeros reais no vetor
int vetorDados::tamanho()
{
    return dados.size();
}
```

Figura 3.11: Exemplo de implementação do método de uma classe sem *templates*.

```
// classe contendo um vetor de elementos do tipo 'T'
template <typename T>
class vetorDados {
    private:
      vector<T> dados;
    public:
      int tamanho();
}

// retorna o numero de elementos no vetor
template <typename T>
int vetorDados<T>::tamanho()
{
    return dados.size();
}
```

Figura 3.12: Exemplo de implementação do método de uma classe utilizando *templates*.

como se a classe não fosse também gabaritada. No código da figura 3.13 vemos um exemplo de uma classe gabaritada que possui um método também gabaritado.

Se o vetor da classe desta figura contiver números inteiros, por exemplo, e quisermos inserir um número real, não teremos maiores problemas, pois um *cast* [4] de um real para um inteiro é perfeitamente aceitável. O problema, agora, é se

[4] *cast*: palavra do idioma inglês usada em linguagens de programação para designar a mudança de um tipo de dado de uma constante ou variável para outro tipo; é uma técnica bastante utilizada

```
// classe contendo um vetor de elementos do tipo 'T'
template <typename T>
class vetorDados {
    private:
      vector<T> dados;
    public:
      template <typename T2>
      void insereElemento(T2 elem)
}

// implementacao do metodo insereElemento.
// transforma um elemento do tipo 'T2' no tipo 'T'
// e o insere no vetor.
template <typename T>
template <typename T2>
void vetorDados<T>::insereElemento(T2 elem)
{
    dados.push_back((T)elem); // faz um "cast" de 'elem'
}
```

Figura 3.13: Exemplo de implementação de um método gabaritado de uma classe também gabaritada.

quisermos inserir uma cadeia de caracteres em um vetor de inteiros, por exemplo. Um simples *cast* é inválido neste caso. Precisamos especificar uma operação diferente quando quisermos inserir uma cadeia.

Para não perdermos generalidade, podemos utilizar a especialização de *templates*. O método da figura 3.13, especializado, fica como na figura 3.14.

Entretanto, o Padrão C++ não permite que este tipo de especialização seja feito. Não é possível especializar um método gabaritado sem antes especializar a classe gabaritada. Ao tentarmos compilar o código acima, receberemos o erro "**enclosing class templates are not explicitly specialized**". Para resolvermos isto, o código precisaria ser como na figura 3.15. Além desta especialização da classe para números reais, precisamos também especializá-la para quaisquer outros tipos de dados com os quais quisermos que o método especializado funcione.

em linguagens fortemente tipadas, principalmente para transformar números reais em números inteiros (Nota do Autor).

```
// implementacao do metodo insereElemento.
// transforma um elemento do tipo 'T2' no tipo 'T'
// e o insere no vetor.
template <typename T>
template <typename T2>
void vetorDados<T>::insereElemento(T2 elem)
{
    dados.push_back((T)elem); // faz um "cast" de 'elem'
}

// especializacao do metodo insereElemento para
// o caso de um elemento cadeia de caracteres.
template <typename T>
template <>          // tiramos o 'typename'
void vetorDados<T>::insereElemento(char * elem)
  // especificamos o tipo 'char *' para 'elem'
{
    float valor;
    valor = atof(elem);  // supomos que o vetor contenha números;
                         // convertemos a cadeia para um real.
                         // a partir dele é possivel converter
                         // para outro tipo de numero facilmente.
                         // claro, se o vetor nao for de numeros
                         // precisaremos de mais especializacoes,
                         // mas isto é apenas para exemplificar.

    dados.push_back((T)valor); // faz um "cast" de 'valor'
}
```

Figura 3.14: Exemplo de especialização de um método gabaritado.

O problema, agora, é que criar várias especializações, tendo que definir quais serão os tipos dos dados que a classe poderá receber, acaba nos prejudicando exatamente onde menos queríamos: na generalidade do componente. Para solucionar este problema, desistimos de utilizar esta especialização. Passamos a questão dos parâmetros específicos de formatos de arquivo, mostrados na página 27, para as próprias classes responsáveis pelo acesso a disco.

Desta forma, seja a classe responsável pelos arquivos `pbm`, seja a classe que entende o formato `jpg`, todas as classes de formatos de arquivo recebem como argumento todos os parâmetros que `LatticeRepresentation` possa conter para armazenar as informações da imagem em memória. Cada uma delas, então, faz o que

```
// implementacao do metodo insereElemento.
// transforma um elemento do tipo 'T2' no tipo 'T'
// e o insere no vetor.
template <typename T>
template <typename T2>
void vetorDados<T>::insereElemento(T2 elem)
{
    dados.push_back((T)elem); // faz um "cast" de 'elem'
}

// especializacao do metodo insereElemento para
// o caso de um elemento cadeia de caracteres
// e um vetor de numeros reais.
template <float> // especializamos para o caso do vetor ser de reais.
template <>      // tiramos o 'typename'.
void vetorDados<T>::insereElemento(char * elem)
  // especificamos o tipo 'char *' para 'elem'
{
    float valor;
    valor = atof(elem);      // convertemos a cadeia para um real.
    dados.push_back(valor); // simplesmente inserimos o
    // valor convertido.
}
```

Figura 3.15: Exemplo de especialização de um método gabaritado e de uma classe gabaritada.

bem entender com estes parâmetros, ou seja, utiliza apenas os que são necessários para o formato reconhecido.

Uma nova classe de formato de arquivo precisa, assim, apenas fornecer os métodos de leitura e gravação utilizando os protótipos especificados. No manual do componente, encontrado no anexo A, são fornecidas estas especificações das estruturas e dos métodos necessários para a adição destas classes.

3.3.5 Estrutura Atual

A figura 3.16 esquematiza a atual estrutura do componente implementado. A relação entre as classes especificadas anteriormente pode ser melhor observada nesta figura.

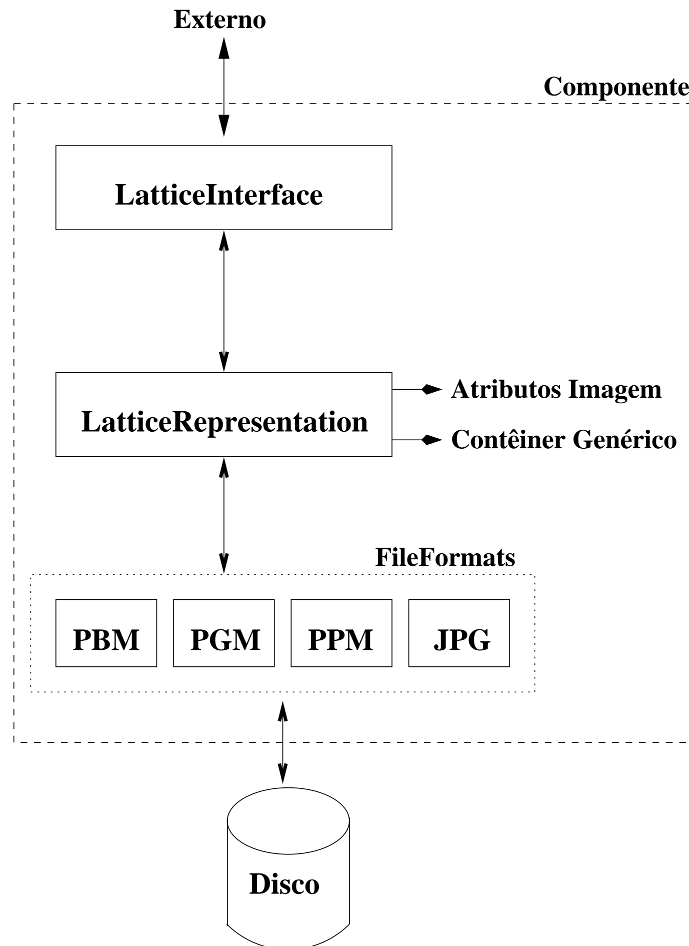

Figura 3.16: Estrutura atual do componente de entrada e saída

3.3.6 Ferramentas Utilizadas

Para escrita do código fonte do componente, fizemos uso da metodologia de programação definida por Carrard em [CAR 2002]. Como parte desta definição, a ferramenta **Doxygen** [HEE 2002] foi usada para a geração automática da documentação do código do componente. Para isto, as diversas classes, estruturas e métodos do código fonte do componente foram comentadas de acordo com demarcações pré-definidas reconhecidas pelo **Doxygen**. A execução desta ferramenta gera um manual detalhado contendo toda a estruturação e documentação do componente. O manual gerado também contém as partes principais do código escrito, sem cheguar ao excesso de exibir todo o código fonte.

Este manual encontra-se no anexo A e deve ser consultado para se obter maiores informações acerca dos detalhes de implementação. Nele também se encontra a

descrição das interfaces utilizadas e os métodos que precisam ser fornecidos caso se deseje criar uma nova classe para o reconhecimento de um novo formato de arquivo de imagem.

Capítulo 4

Resultados

O desenvolvimento do componente introduziu uma nova abordagem sobre a implementação das operações de entrada e saída de arquivos de imagem. Deixamos de visualizar estas operações como sendo a última parte a receber atenção no desenvolvimento de uma aplicação para processamento de imagens, para finalmente resolvermos os conflitos dos diversos formatos de arquivo de imagem em um nível mais baixo destas aplicações. Desta forma, construímos uma estrutura de imagem em memória que é genérica e flexível, de modo que detalhes dos formatos de arquivos são transparentes para a aplicação.

Além de termos conseguido definir uma estrutura de projeto bastante adequada, a utilização de contêiners e de *templates* provou ser eficiente e flexível, melhorando a implementação do componente tanto em velocidade de desenvolvimento quanto em legibilidade e flexibilidade do código. A definição dos métodos de leitura e escrita de arquivos da classe `LatticeRepresentation`, utilizando *templates*, permite que o suporte a novos formatos de arquivos seja facilmente incorporado ao componente. Para isto, basta criar uma classe com os protótipos exigidos pela `LatticeRepresentation` e implementá-los de acordo com o formato de arquivo desejado. As alterações adicionais que precisam ser feitas limitam-se a adicionar o nome do novo formato na estrutura de tipos de formato, e incluir o nome da nova classe no método da `LatticeRepresentation` que seleciona a classe desejada.

A utilização desde o início do trabalho de uma metodologia de desenvolvimento bem definida trouxe diversos benefícios:

- Facilitou a leitura e o entendimento do código escrito;
- Permitiu que mesmo grandes alterações no código fossem realizadas de maneira

simples e livre de erros;

- Padronizou a nomenclatura utilizada e;
- Permitiu que os comentários no código fossem facilmente transformados em um manual de referência.

O manual citado se encontra nos anexos (página 44), e é uma ótima referência para que várias pessoas entendam a estrutura das classes e os protótipos dos métodos utilizados, permitindo que elas iniciem o desenvolvimento de novas classes para o suporte de mais formatos de arquivo de imagem.

Capítulo 5

Conclusões e Trabalhos Futuros

Este trabalho apresentou uma abordagem sobre programação genérica para o desenvolvimento de algoritmos reusáveis no processamento e análise de imagens. Ele está baseado nas idéias que se tornaram populares através da biblioteca padrão STL, da linguagem C++. Esta abordagem supera um número de limitações da maneira tradicional de se projetar *software* reusável, na medida em que seu modelo de programação é mais eficiente do que o de métodos tradicionais – já que uma única implementação de algoritmo pode ser adaptada a diversas situações – e só existe uma pequena perda de desempenho comparada a uma versão otimizada do algoritmo.

A combinação de métodos genéricos com outras abordagens de projeto de *software* – como o desenvolvimento baseado em componentes e o projeto orientado a objeto – se mostrou bastante eficiente. A idéia básica foi utilizar objetos e componentes para conseguir funcionalidade e para representar seus relacionamentos de acordo com as necessidades de uma aplicação em particular, enquanto que a maioria do trabalho em si foi delegada a blocos de construção genéricos residentes em um nível mais baixo da estrutura. Desta maneira, as vantagens e desvantagens dos diferentes métodos de projeto de *software* puderam ser otimamente equilibradas.

Concluindo, desenvolvemos um componente de *software* capaz de ser encaixado como um bloco de construção no desenvolvimento de aplicações mais complexas para o processamento de imagens. A resolução dos problemas advindos dos conflitos de diferentes formatos de arquivo é retirada das aplicações que utilizam entrada e saída de imagens, uma vez que este componente não segue a maneira tradicional de oferecer suporte a diversos formatos, que é simplesmente converter para um formato comum. Ao invés disto, opta por investir, subir um patamar neste assunto e utilizar uma

nova abordagem, criando uma representação genérica para as imagens em memória e oferecendo às aplicações um acesso transparente a elas, independentemente dos formatos de arquivos e suas peculiaridades.

Um aumento na criação e desenvolvimento de blocos de construção genéricos e independentes permitiria que aplicações muito mais complexas e eficientes fossem desenvolvidas, através do encaixe de vários componentes especialistas. Este trabalho abre caminho para que ferramentas que utilizam entrada e saída de imagens se concentrem muito mais em suas funcionalidades específicas sem correrem o risco de encontrar erros gerados por operações de outras camadas.

Os trabalhos futuros a serem feitos se concentram em divulgar estas vantagens obtidas da utilização deste componente na construção de aplicações para processamento de imagens. E também, em sugerir a terceiros a contribuição para o incremento dos arquivos suportados, já que o reconhecimento de novos formatos de arquivos, como GIF, BMP e TIFF, pode ser acrescentado por pessoas não ligadas diretamente a este trabalho, de maneira simples e rápida através do entendimento da interface do componente explicada no manual do mesmo.

Referências Bibliográficas

[CAR 2001] CARRARD, M. C. C.; D'ORNELLAS, M. C. Ferramenta de Software para Processamento de Imagens com Programação Genérica: Projeto do Núcleo. **Academic Project**, 2001.

[CAR 2002] CARRARD, M. C. C. **Linguagem e Metodologia de Programação**. Grupo PIGS – Image Processing Group: UFSM, 2002. Versão 1.0.

[CHA 2001] CHANDRA, P.; IBANEZ, L. ImageIO: design of an extensible image input/output library [*on line*]. **ACM Crossroads**, 2001. [cited 15 January 2003]. Available from www: <`http://www.acm.org/crossroads/xrds7-4/imageIO.html`>.

[D'OR 2002] D'ORNELLAS, M. C. et al. Programação Genérica com C++ e STL. **Escola Regional de Informática 2002**, LACESM/CT/UFSM, CRSPE/INPE, Santa Maria, RS, 2002.

[D'OR 2001] D'ORNELLAS, M. C. **Algorithmic Patterns for Morphological Image Processing**. 2001. Ph.D. Thesis — Univesiteit van Amsterdam.

[FAB 2001] FABRI, A. Cgal – the Computational Geometry Algorithm Library. **Academic Report**, INRIA, France, 2001.

[HEE 2002] HEESCH, D. v. **Doxygen manual [*on line*]**. [S.l.: s.n.], 2002. version 1.2.18. [cited 15 January 2003]. Available from www: <`http://www.doxygen.org`>.

[JOS 99] JOSUTTIS, N. M. **The C++ Standard Library: a tutorial and reference**. London: Addison Wesley Publishing Company, 1999.

[KöTH 99] KöTHE, U. Reusable Software in Computer Vision. In: JäHNE, B.; HAUSSECKER, H.; GEISSLER, P. (Eds.). **Handbook of Computer Vision and Applications**. Fraunhofer-Institut für Graphische Datenverarbeitung, Rostock, Germany: Acadamic Press, 1999. v.3, p.106–134.

[KöTH 2000] KöTHE, U. **Generische Programmierung für die Bildverarbeitung**. 2000. 274p. Ph.D. dissertation — Fachbereich Informatik, Universität Hamburg, Hamburg, Deutschland. ISBN: 3-8311-0239-2, (in German). Available from www: <`http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/DissPrint.ps.%gz`>.

[KöTH 2000a] KöTHE, U. STL-Style Generic Programming with Images. **C++ Report Maganzine**, v.12(1), p.24–30, jan 2000.

[KöTH 2002] KöTHE, U. **The VIGRA Computer Vision Library [*on line*]**. University of Hamburg, Germany: [s.n.], 2002. [cited 15 January 2003]. Available from www: <`http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra`>.

[LIP 98] LIPPMAN, S. B.; LAJOIE, J. **C++ Primer**. 3.ed. [S.l.]: Addison Wesley, 1998.

[MUR 94] MURRAY, J. D.; VANRYPER, W. **Encyclopedia of Graphics File Formats**. [S.l.]: O'Reilly & Associates, Inc., 1994.

[MUS 93] MUSSER, D. R.; STEPANOV, A. Algorithm-oriented generic libraries. **Technical Report**, p.1–9, 1993.

[PUR 2002] PURDY, D. **Exploring the Factory Design Pattern [*on line*]**. [S.l.]: Microsoft Corporation, 2002. [cited 15 January 2003]. Available from www: <`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda%/html/factopattern.asp`>.

[RIT 2000] RITTER, G. X.; WILSON, J. N. **Handbook of Computer Vision Algorithms in Image Algebra**. 2.ed. New York: CRC Press, 2000.

[STE 95] STEPANOV, A.; LEE, M. The Standard Template Library. **Technical Report 94-34**, Hewlett-Packard Laboratories, 1995.

[STE 89] STEPANOV, A.; MUSSER, D. Generic programming. **1st International Joint Conference of ISSAC-88 and AAEC-6**, v.358, p.?–?, 1989.

[STR 97] STROUSTRUP, B. **The C++ Programming Language**. 3.ed. New York: Addison Wesley Publishing Company, 1997.

[VIL 2001] VILLAS-BOAS, S. B. **C/C++ e Orientação a Objetos em Ambiente Multiplataforma [*on line*]**. UFRJ, Rio de Janeiro: [s.n.], 2001. [cited 15 January 2003]. Available from www: `<http://lpi.lps.ufrj.br/~villas/index.chtml?p=livro_cpp.html>`.

Anexo A

Manual de Referência