Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: FREDERICO DE MIRANDA COELHO

e aprovada pela Banca Examinadora.

Campinas, 13 de MARÇO de 03

COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Uso de Componentes de Software no Desenvolvimento de *Frameworks* Orientados a Objetos

Frederico de Miranda Coelho

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uso de Componentes de Software no Desenvolvimento de *Frameworks* Orientados a Objetos

Frederico de Miranda Coelho

Dezembro de 2002

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)

- Prof. Paulo César Masieiro
Instituto de Ciências Matemáticas e de Computação (ICMC – USP – São Carlos)

- Profa. Eliane Martins
Instituto de Computação (IC-UNICAMP)

- Profa. Ariadne Maria Brito Rizzoni Carvalho (Suplente)
Instituto de Computação (IC-UNICAMP)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Coelho, Frederico de Miranda

C65u Uso de componentes de software no desenvolvimento de frameworks orientados a objetos / Frederico de Miranda Coelho -- Campinas, [S.P. :s.n.], 2002.

Orientadora : Cecília Mary Fischer Rubira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Framework (Programação de computador). 2. Software – Reutilização. 3. Análise de componentes principais. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 18 de dezembro de 2002, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Paulo Cesar Masiero
ICMSC – USP

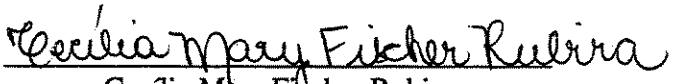
Profa. Dra. Eliane Martins
IC - UNICAMP

Profa. Dra. Cecília Mary Fischer Rubira
IC – UNICAMP

Uso de Componentes de Software no Desenvolvimento de *Frameworks* Orientados a Objetos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Frederico de Miranda Coelho e aprovada pela Banca Examinadora

Campinas, 18 de dezembro de 2002.



Cecília Mary Fischer Rubira
Instituto de Computação
Universidade Estadual de Campinas
(Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Agradecimentos

"Não chores, meu filho;
Não chores, que a vida
É luta renhida:
Viver é lutar.
A vida é combate
Que fracos abate,
Que fortes, os bravos,
Só pode exaltar"

Gonçalves Dias

A Deus, o primeiro a ouvir os meus lamentos e a me consolar com Sua imensa paz.

À minha noiva, Aline Muinhos, que soube ter paciência para com o meu empenho em terminar o trabalho e sempre me apoiou nas horas de dúvida e isolamento.

À minha orientadora, Professora Cecília Mary Fischer Rubira, que me acolheu de forma incondicional nos primeiros dias de meu mestrado, pelos ensinamentos e paciência em todos os momentos, além da confiança em mim, mesmo depois de deixar a dedicação exclusiva ao mestrado.

À Rosana Braga por todo seu apoio, ajuda e ensinamento, além de sua disponibilidade e interesse em me fornecer qualquer informação a qualquer momento.

Ao Paulo Astério pelos ensinamentos e ajuda na obtenção de informações durante o decorrer da dissertação.

À minha família que soube entender a minha ausência nas horas importantes. Em especial às minhas mães Ilda, Dedé e Tia Rita e ao meu pai Renato, que me apoiaram incondicionalmente em todos os momentos durante esse período.

Ao meu irmão, fonte de sabedoria e paciência, que a qualquer momento tem sempre um conhecimento a me transmitir, seja pessoal ou profissional.

A todos que em meus momentos de desespero escutaram os meus lamentos e medos. Em especial, a Deus, Aline, Nalu, Marcelo, Tia Rita, Marize e "Euder", que geralmente eram

as pessoas que estavam mais próximas de mim nos dias em que tinha que voltar para Campinas.

Aos amigos da UNICAMP que me acompanharam e me ajudaram em minha trajetória. À minha família campineira Glauber e Willian, meus amigos-irmãos, que sempre estavam ao meu lado seja para o estudo ou para o "divertimento".

Aos revisores de dissertação, Joziane, Aline, Marcelo e Nalu, que possibilitaram um discernimento mais correto da tese.

Aos KANALHAS por entenderem a minha ausência nesse período e pela lição de vida que cada um me proporciona, a qual sempre foi um grande incentivo para o término do mestrado.

A todos os professores da UNICAMP que tive oportunidade de conhecer e me passaram preciosos conhecimentos que serão úteis durante toda a minha vida.

À UNICAMP que me deu tão nobre oportunidade e à qual agora eu agradeço.

Ao CNPq que me financiou enquanto mantive a dedicação exclusiva ao curso de mestrado.

Finalmente, a todos que de certa forma me ajudaram na escalada de mais esse degrau e que, por motivos de esquecimento momentâneo, não se enquadram em nenhum dos agradecimentos acima e/ou não tiveram seus nomes citados, o meu muito obrigado.

"Vim
Vi
Aprendi
e Venci"

Anônimo

Resumo

Os pesquisadores da área de Engenharia de Software sempre estão buscando formas de conseguir a qualidade de sistemas de software exigida pelo mercado. Inúmeras tecnologias têm sido propostas, sendo os *Frameworks* Orientados a Objetos (OO) uma das tecnologias mais modernas para se obter tal qualidade. Entretanto os *Frameworks* Orientados a Objetos possuem alguns problemas, como de desenvolvimento, composição, uso e manutenção. Para isso, pesquisadores têm unido outras tecnologias a essa, como desenvolvimento baseado em componentes e linguagens de padrões, com o intuito de solucionar os problemas apresentados. Porém não se sabe o quanto efetivamente essas tecnologias auxiliam nos problemas dos *Frameworks* Orientados a Objetos em comparação a um desenvolvido apenas com o paradigma orientado a objetos puro.

Este trabalho apresenta métodos de estudo, desenvolvimento e análise arquitetural e suas aplicações em uma arquitetura de *framework* OO baseada em classes e outras duas constituídas por componentes, sendo todas estas arquiteturas obtidas de um mesmo domínio de aplicação. A finalidade da utilização destes métodos é analisar a aplicabilidade do paradigma de desenvolvimento baseado em componentes na construção de *Frameworks* Orientados a Objetos, em nível arquitetural, em um domínio de aplicação específico, verificando seu impacto no grau de complexidade da arquitetura do *Framework* Orientado a Objetos e, conseqüentemente, na facilidade de sua utilização.

O resultado deste experimento é a diminuição da complexidade da arquitetura do *framework* OO em questão, quando aplicado as técnicas de desenvolvimento baseado em componentes, acarretando, por exemplo, em uma manutenibilidade, reusabilidade, usabilidade e documentação mais fáceis.

Abstract

Software engineering researchers are always investigating for new forms to achieve software quality. Several technologies have been proposed, one of them, the technology for building Object Oriented (OO) Frameworks is a new technology to achieve software quality. However, the OO Frameworks have some limitations, such as complexity in their development, composition, use, and maintenance.

To achieve software quality, researchers have join other technologies, like Component Based development and Pattern Languages to minimize these problems. However, the effectiveness of these technology compared with pure object oriented development isn't known.

The main goal of this research is to analyze, at architectural level, the Component Based Paradigm applicability to built OO Frameworks in a specific application domain.

The application of the component-based development resulted in a reduction of the OO framework architecture complexity and consequently promoted better maintainability, reusability, user-friendly and documentation.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 – O Contexto

Na literatura sobre Engenharia de Software, é comum encontrar o termo "Crise de Software", identificada no final da década de 60, referenciado como uma "Aflição Crônica" (Pressman, 2001). Isto porque, quando foi cunhado o termo "Crise de Software", concluiu-se que os métodos de produção de software não eram adequados às necessidades crescentes da indústria de defesa e de processamento de dados, ou seja, a qualidade de um produto de software não era satisfatória. Esta conclusão ao longo dos anos, manteve-se verdadeira nas suas devidas proporções, apesar do desenvolvimento de várias soluções para superá-la.

Nos últimos dez anos, com o avanço tecnológico e com a globalização mundial, as exigências de qualidade de software se tornaram cada vez maiores. Com isso, novas propostas e soluções antes consideradas não promissoras passaram a ser fonte de estudo para amenizar essa aflição crônica, como o paradigma Orientado a Objetos (OO), *frameworks* OO, componentes de software e linguagens de padrões. Para atingir a qualidade de software tão desejada nos dias atuais, dois pontos relevantes são os atributos de manutenção e reuso de software porque eles têm um custo muito elevado e são muito exigidos na competitividade global. Entretanto, os sistemas de software estão cada vez mais complexos, dificultando a obtenção de tais atributos.

Segundo a literatura especializada, os *frameworks* OO são ferramentas que têm o objetivo de facilitar o desenvolvimento de aplicações de um mesmo domínio, que demonstram ser uma das técnicas mais promissoras para se conseguir a sistematização do processo de criação de sistemas de software complexos e, com isso, a obtenção da qualidade desejada. Isso ocorre porque o *framework* OO é um projeto genérico em um domínio, que pode ser adaptado a aplicações específicas (Fayad *et. al.*, 1999), reduzindo, assim, a distância do conhecimento entre a concepção inicial do sistema e a implementação do sistema executável final.

Entretanto, os *frameworks* OO existentes, possuem ainda alguns problemas em aberto, como: (i) no desenvolvimento do *framework*, existe uma grande dificuldade em assegurar a corretude e a versatilidade de todas as suas funcionalidades; e (ii) a definição de estratégias de manutenção que permitam sincronizar a evolução do *framework* e das aplicações desenvolvidas a partir do mesmo, ocorrem normalmente de forma independente. Porém, um dos principais obstáculos à utilização de *frameworks* OO numa escala maior ainda é a complexidade de sua arquitetura de software, que dificulta o seu entendimento e, conseqüentemente, a instanciação de aplicações derivadas dos mesmos (Bosch *et. al.*, 2000). Entende-se por arquitetura de software a estrutura dos elementos de um programa/sistema, seus inter-relacionamentos, princípios e diretrizes guiando o projeto e evolução ao longo do tempo (Garlan, 1995).

A aplicação de outra tecnologia na estruturação dos *frameworks*, denominada Desenvolvimento Baseado em Componentes (DBC) (Brown *et. al.*, 1998), que também visa a reutilização de software de forma mais sistemática, pode contribuir para reduzir a complexidade de suas arquiteturas, elevar o grau de padronização e permitir a reutilização do software de maneira mais efetiva (Posnak *et. al.*, 1997). A característica principal desse novo paradigma é a construção modular do software, com a separação clara entre a definição das interfaces dos componentes de software (unidades de composição) e suas implementações (Brawn *et. al.*, 1998). Denomina-se como componente de software uma unidade de composição com interface contratualmente especificada e com dependências de contexto (Szyperski *et. al.*1996; Szyperski *et. al.*1997).

Os *frameworks* baseados em componentes são, em geral, construídos a partir de um modelo de componentes padrão, tais como DCOM (*Distributed Component Object Model*) (Box, 1998) ou EJB (*Enterprise Java Beans)* (Johnson, 1998). Essa padronização deixa explícitas algumas decisões de projetos importantes, como o estilo arquitetural básico do *framework*, representando um passo importante para resolver conflitos arquiteturais que podem surgir quando uma mesma aplicação utiliza diferentes *frameworks* (Bosch *et. al.*, 2000), contribuindo também para facilitar o entendimento do *framework* por parte do usuário. Dentre os *frameworks* baseados em componentes já existentes, podemos citar o MultiTEL (Fuentes *et. al.*, 2000), o SanFrancisco (Monday *et. al.*, 1999) e os *frameworks* das chamadas linguagens visuais, como VisualBasic (Basic, 2002) e Delphi (Borland, 2002). Entretanto, não

há um experimento prático que mostre o impacto dessa nova tecnologia na arquitetura dos *frameworks* em comparação com uma arquitetura baseada em classes.

1.2 – Objetivos

Os principais objetivos desta dissertação são:

(1) Averiguar como a utilização de componentes de software pode diminuir a complexidade da arquitetura de um *framework* OO, em comparação a uma arquitetura de software constituída apenas de classes. Para isso, um experimento prático utilizando uma arquitetura baseada em classes e outras duas arquiteturas constituídas por componentes será realizado, com o intuito de evidenciar como essa inserção de componentes de software pode contribuir para a evolução do estado da arte de *frameworks* OO.

(2) Evidenciar e exemplificar os problemas encontrados com os *frameworks* OO baseados em classes descritos na literatura especializada. Para isso, algumas aplicações serão desenvolvidas em um *framework* OO baseado em classes existente, que tem as características mais comuns dos *frameworks* OO encontrados atualmente.

1.3 – O Experimento Prático

A dissertação apresenta um estudo comparativo entre uma arquitetura baseada em classes e outras duas constituídas por componentes para a averiguação da utilização das técnicas de Desenvolvimento Baseado em Componentes na criação e utilização de *frameworks* OO, que (i) não requeira a análise do código fonte; (ii) que seja realizada em um nível de abstração logo acima do código fonte; e (iii) compare as diferentes arquiteturas desenvolvidas com métodos diferentes.

A realização deste estudo envolve um conjunto de técnicas de reutilização de software (Capítulo 2), os quais são de fundamental importância para entendimento do porque unir outras tecnologias à tecnologia de *frameworks* OO e dos termos utilizados durante o experimento. A sistematização do experimento é fornecida por um conjunto de métodos

(Capítulo 3) que descrevem a forma de estudar, desenvolver e analisar tanto a arquitetura baseada em classes quanto as constituídas por componentes.

A análise deste experimento é fundamentada (i) no método de análise de compromissos da arquitetura (ATAM – *Architecture Tradeoff Analysis Method*) (Kazman *et. al.*, 1998; Barbacci *et. al.*, 1998), a fim de realizar a análise qualitativa das arquiteturas; e (ii) na métrica Fator de Acoplamento (CF - *Coupling Factor*) (Harrison *et. al.*, 1998a; Harrison *et. al.*, 1998b), adaptada pelo autor e na diretrizes de modularização (Pressman, 2001) para realizar a análise quantitativa das arquiteturas.

A arquitetura baseada em classes utilizada no experimento foi a arquitetura do *framework* OO GREN (Braga *et. al.*, 2001), que é um *framework* OO para o domínio de gestão de recursos de negócio. A partir de sua arquitetura e de seu domínio foram obtidas as duas arquiteturas constituídas por componentes. Sendo estas três arquiteturas os artefatos utilizados para a realização do experimento, ou seja, na análise arquitetural do *framework* OO em questão, que tem o objetivo de averiguar a diminuição da complexidade de sua arquitetura quando aplicado às técnicas DBC.

Com o estudo comparativo proposto, um *framework* OO constituído por componentes apresenta, na maioria das vezes, várias vantagens em relação a um *framework* OO baseado unicamente em classes, devido à simplificação de sua arquitetura, como, por exemplo, manutenibilidade, reusabilidade, usabilidade e documentação facilitadas.

O *framework* GREN possui uma linguagem de padrões associada ao seu domínio, a qual foi de grande valia para a obtenção das arquiteturas a serem avaliadas, apesar de não ser do interesse inicial do autor utilizar uma linguagem de padrões no decorrer do experimento. Entende-se por linguagem de padrões a representação da seqüência temporal de decisões que levam ao projeto completo de uma aplicação, de forma a tornar-se um método para guiar o processo de desenvolvimento (Braga *et. al.*, 2001). O experimento prático está descrito nos Capítulos 4 e 5 em maiores detalhes.

1.4 – Contribuições

Este trabalho apresenta como principal contribuição:

(1) Solução de alguns problemas encontrados nos *frameworks* desenvolvidos a partir do paradigma orientado a objetos puro, como, por exemplo, o de documentação e desenvolvimento. A solução desses problemas é proporcionada pela introdução de componentes na arquitetura do *framework*, auxiliando na documentação, no desenvolvimento, na composição, no uso e na manutenção do *framework* e, por conseguinte, das aplicações geradas a partir dele. Dessa forma, os métodos de análise arquitetural mostram o quanto os *frameworks* constituídos por componentes podem ajudar amenizando esses problemas.

Como conseqüência dessa contribuição principal pode-se citar algumas sub contribuições:

(1.1) Proposta de um método para o estudo de um *framework* já desenvolvido, com o objetivo de extrair as características, os requisitos, os conceitos e as relações entre os requisitos do domínio a que se aplica. Aplicação desse método em um *framework* do domínio de gestão de recursos de negócio, verificando a sua aplicabilidade.

(1.2) Proposta de um método de reengenharia para a obtenção de uma arquitetura estruturada em componentes a partir de uma arquitetura baseada em classes. Aplicação deste método na arquitetura do *framework* de gestão de recursos de negócio, verificando sua viabilidade.

(1.3) Estudo do processo de especificação de software baseado em componentes, proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001); e proposta do uso de uma linguagem de padrões para auxiliar sua aplicação. Aplicação

desse processo com o uso de linguagens de padrões para o desenvolvimento de uma arquitetura baseada em componentes para o domínio de gestão de recurso de negócio.

(1.4) Estudo sobre o método de análise de compromissos da arquitetura (ATAM) e sua aplicação sobre três arquiteturas distintas de *framework* do domínio de gestão de recursos de negócio.

(1.5) Proposta de adaptação da métrica Fator de Acoplamento (Harrison *et. al.*, 1998a; Harrison *et. al.*, 1998b) para realizar a quantificação do acoplamento arquitetural de uma arquitetura constituída de componentes. Aplicação dessa métrica nas arquiteturas obtidas durante o experimento.

(1.6) Averiguação quantitativa da influência da coesão e do acoplamento na complexidade, manutenibilidade e reusabilidade nas arquiteturas do *framework*.

1.5 – Organização da Dissertação

O Capítulo 2 apresenta o tema da reutilização de software, conceituando diversas técnicas de reutilização de software, discutindo suas relações e como elas podem facilitar o desenvolvimento e a utilização de *frameworks* OO.

No Capítulo 3, são apresentados os métodos de análise e de desenvolvimento arquiteturais propostos pelo autor, necessários para a obtenção das arquiteturas do *framework* a serem utilizadas no experimento. É discutido também como as análises qualitativa e quantitativa devem ser realizadas. O capítulo apresenta o método de análise de compromissos da arquitetura (ATAM), utilizado na análise qualitativa, e o método de análise quantitativa proposto pelo autor, para analisar quantitativamente características de uma arquitetura de *framework* OO.

O Capítulo 4 descreve o experimento a ser realizado e a aplicação de parte dos métodos demonstrados no capítulo anterior, necessários para a obtenção das arquiteturas do *framework* a serem utilizadas no experimento.

O Capítulo 5 utiliza os métodos de análise apresentados no Capítulo 3 para averiguar a influência de componentes de software na arquitetura de um *framework* OO, utilizando as arquiteturas do *framework* obtidas no Capítulo 4. Esse capítulo apresenta os resultados do experimento, discutindo as vantagens e desvantagens dae utilização de componentes de software em uma arquitetura de *framework* OO.

Finalmente, no Capítulo 6, são descritas as conclusões desta dissertação, os trabalhos relacionados e o plano de continuidade deste trabalho.

Capítulo 2

Técnicas de Reutilização de Software

A Engenharia de Software tem como um dos seus principais objetivos a reutilização de software em larga escala. A obtenção deste objetivo implica no emprego de técnicas que permitam a reutilização sistemática de todos os artefatos resultantes do processo de desenvolvimento de software, como idéias, conceitos, requisitos e projetos adquiridos ou construídos.

Várias propostas, como a utilização de linguagens de padrões, desenvolvimento baseado em componentes e *frameworks* OO no desenvolvimento do software, têm surgido com o intuito de propiciar a reutilização de software. Entretanto, tem-se constatado a ineficiência dessas propostas quando aplicadas isoladamente. Desta forma, pesquisas estão sendo realizadas para aplicar algumas dessas propostas em conjunto no desenvolvimento de software com o intuito de obter a reutilização de software em larga escala.

Neste Capítulo serão apresentados os conceitos utilizados nesta dissertação e uma comparação entre eles mostrando como podem ser utilizados em conjunto para a obtenção de um melhor desenvolvimento de software. As Seções 2.1, 2.2, 2.3 e 2.4 conceituam *framework* OO, componentes de software, padrões de software, linguagens de padrões e arquitetura de software respectivamente. A Seção 2.5 faz uma comparação entre essas diversas formas de reuso.

2.1 – Frameworks OO

A meta da tecnologia hoje é a construção de estruturas de sistema de software que possam resolver um certo conjunto de problemas semelhantes, determinando, assim, um caminho de desenvolvimento parecido para aplicações de um domínio específico. *Frameworks* OO são ferramentas para ajudar os desenvolvedores a alcançarem esta meta (Lewandowski, 1998).

No decorrer desta dissertação, a palavra *Framework* estará referindo-se a *Framework* OO. Um *framework* é um projeto reutilizável, de parte ou de todo o sistema, que é representado por um conjunto de classes abstratas, organizadas de maneira que suas instâncias possam interagir. Um *framework* também pode ser definido como um esqueleto de uma aplicação que pode ser particularizado por um desenvolvedor, isto é, um projeto reutilizável de um sistema que descreve o sistema decomposto em um conjunto de objetos e elementos que interagem entre si. Um *framework* descreve a interface de cada objeto e o fluxo de controle entre eles (Fayad *et. al.*, 1999).

Um *framework* captura as decisões de projeto que são comuns ao domínio da aplicação. Assim, uma aplicação baseada em um *framework* faz tanto o reuso do código fonte quanto o reuso do projeto da arquitetura. Isso serve para padronização da estrutura da aplicação e para diminuir significativamente a redução do tempo de construção da aplicação, do tamanho e da complexidade do código fonte. Tem-se ainda a facilidade de manutenção e o aumento da qualidade da aplicação (Foote *et. al.*, 1998).

Para que uma aplicação possa fazer o uso de um *framework* é necessário que ela seja uma extensão deste. A extensibilidade é propiciada por partes do *framework* que servem para ser adaptadas às características específicas da aplicação, denominadas *hot spots*. Tipicamente correspondem a uma classe abstrata na hierarquia de classes do *framework* (Coelho, 1998; Foote *et. al.*, 1998; Pree, 1997).

O *framework* realiza uma inversão de controle entre código específico da aplicação e o software do *framework*. Ao invés de sua aplicação chamar o *framework* (o código que quer reutilizar), é ele quem chama a sua aplicação, Figura 2.1. O resultado dessa inversão é um projeto genérico que pode ser instanciado para a construção de uma aplicação (Gamma *et. al.*, 2000; Fayad *et. al.*, 1999, Lewandowski, 1998).

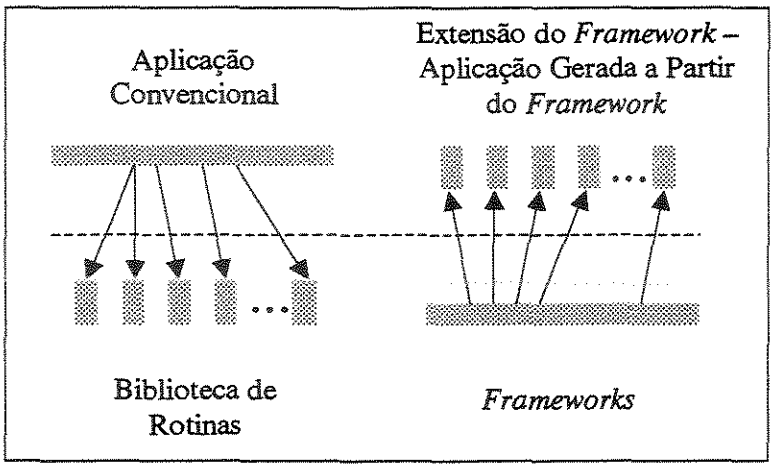


Figura 2.1 – Inversão de Controle

Portanto, os *frameworks* trazem como benefícios: a modularidade, a reusabilidade, a extensibilidade e a inversão de controle, fornecendo um alto índice de reusabilidade para aplicações de um domínio específico (Fayad *et. al.*, 1999; Lewandowski, 1998), caso ele seja bem feito.

A desvantagem é que os *frameworks* requerem um enorme esforço de desenvolvimento. O custo do desenvolvimento de um *framework* é significativamente alto se comparado com os custos de uma aplicação específica. Então, os *framewoks* representam um longo investimento que só dará resultado se forem desenvolvidas várias aplicações similares (Fayad *et. al.*, 1999; Pree, 1997).

Além disso, ferramentas e métodos que auxiliam o desenvolvimento de *frameworks* são quase inexistentes ou incipientes. A tecnologia de *framework* se sustenta por si só, mas não está ainda madura. Mudanças nas classes bases de um *framework* podem causar sérias conseqüências na sua hierarquia de classes (Foote *et. al.*, 1998; Pree, 1997).

Apesar das desvantagens mencionadas, a tecnologia de *frameworks* está sendo amplamente aceita pela sociedade desenvolvedora e pesquisadora de software, já que é um projeto genérico que incorpora soluções típicas de um domínio específico, sendo sua característica mais marcante a busca pelo reuso.

2.1.1 – Classificação dos Frameworks

Os *frameworks* podem ser classificados de acordo com a forma como fornecem os seus *hot spots*, ou seja, a forma como ele pode ser estendido para gerar uma nova aplicação. A primeira forma de classificação são os *frameworks* de herança ou "caixas brancas", que contam com a herança e a ligação dinâmica para conseguir a sua extensibilidade. Em contraste, existem os *frameworks* de interfaces fixas ou "caixas pretas", que são estruturados usando a composição e a delegação de objetos, ao invés de herança. Existe, ainda, um terceiro tipo de *framework*, que é um intermediário entre os dois anteriores, os frameworks desse tipo são denominados "caixas cinzas". Estes possuem flexibilidade e extensibilidade suficientes dos *frameworks* "caixas brancas" e conseguem esconder informações desnecessárias dos desenvolvedores de aplicação, característica dos *frameworks* "caixas pretas". (Fayad, 2000; Fayad *et. al.*, 1999)

Em geral, uma maior flexibilidade implica num maior esforço de programação por parte dos desenvolvedores que fazem uso dos *frameworks*. Por outro lado, implicam num menor esforço para os criadores dos *frameworks*, pois não há necessidade de previsão de todas as alternativas de implementações possíveis, ou seja, definição completa das suas interfaces fixas. Quanto mais os *frameworks* forem constituídos de interfaces fixas, maior será a diminuição do seu nível de flexibilidade e sua reutilização se tornará mais fácil, pois basta que se escolha a interface desejada em vez de fornecer a implementação completa, como é o caso dos "caixas brancas". Porém, isso requer maior esforço dos desenvolvedores dos *frameworks*.

Os *frameworks* podem ser classificados também quanto ao escopo da aplicação a que eles se destinam. Segundo Fayad *et. al.* (Fayad *et. al.*, 1997; Fayad *et. al.*, 1999), os *frameworks* podem ser classificados em: (i) *frameworks* de infra-estrutura do sistema, que simplificam o desenvolvimento da infra-estrutura de sistemas portáteis e eficientes, como, por exemplo, sistemas operacionais e interfaces com o usuário; (ii) *frameworks* de integração *"middleware"*, que são usados, em geral, para integrar aplicações e dar suporte à computação distribuída, são exemplos *o "Object Request Broker"* (ORB) e o *"middleware"*, orientado a mensagens; e, por fim, (iii) *frameworks* de aplicação empresarial, que estão voltados a domínios de aplicação mais amplos, como por exemplo, sistemas de telecomunicações e aviação.

2.1.2 – Problemas Existentes no Desenvolvimento de *Frameworks*

A prática de se desenvolver *frameworks* está se tornando mais comum devido às vantagens que se obtêm ao utilizá-los, entre elas, o crescimento da reusabilidade e a redução de tempo para produzir a aplicação. Mas existem desvantagens e alguns problemas relacionados aos *frameworks*. Os problemas podem ser organizados, segundo Bosch (Bosch *et. al.*, 2000), em quatro categorias: (i) desenvolvimento, (ii) uso, (iii) composição e (iv) manutenção do *framework* (Bosch *et. al.*, 2000).

Problemas no Desenvolvimento de *Frameworks*

Nesta categoria estão relacionados os problemas ocorridos antes da liberação e do uso do *framework* pela primeira aplicação real. Entre eles estão a inviabilidade de se contar com os modelos de negócio já existentes, a verificação de todas as possibilidades do comportamento abstrato (assegurar a corretude[1] e a versatilidade de todas as funcionalidades fornecidas pelo *framework* geralmente é muito difícil) e a obtenção da documentação apropriada do *framework* (Bosch *et. al.*, 2000).

Problemas no Uso de *Frameworks*

Os problemas relacionados a esta categoria dizem respeito à instanciação do *framework*. Tais problemas existem em sua maioria devido às dificuldades de documentação do *framework*, prejudicando assim o seu entendimento. Alguns outros problemas estão relacionados à depuração de programas que usam bibliotecas (Bosch *et. al.*, 2000), já que o *loop* de controle em um *framework* não é geralmente, representado sobre uma entidade e, sim, distribuído ao longo do seu código.

[1] O termo corretude é utilizado no decorrer desta dissertação com o mesmo significado apresentado por Staa (Staa, 2000): "Corretude = Correto + -tude, propriedade de estar correto".

Problemas na Integração de *Frameworks*

Nesta categoria encontra-se a maior variedade de problemas. Tal fato ocorre porque os *frameworks* foram projetados para serem estendidos e não compostos com outros *frameworks*. Segundo Mattsson (Mattsson *et. al.*, 1999), já foram identificados seis problemas comuns no desenvolvimento de *framework* e aplicações, quando se tem dois ou mais *frameworks* integrados. Quatro destes estão relacionados ao projeto da sua arquitetura. Dentre eles, temos (Mattsson *et. al.*, 1999):

inversão de controle: ocorre quando dois ou mais *frameworks* chamam a mesma aplicação simultaneamente, todos acabam assumindo o controle da aplicação;

integração com sistemas legados e ferramentas existentes: ocorre quando há o desejo de integrar ferramentas existentes e componentes legados com um *framework*. Como os *frameworks* possuem uma funcionalidade mais abrangente para possibilitar o desenvolvimento de aplicações voltadas para o usuário final e as ferramentas existentes e os componentes legados já possuem um comportamento específico, ou seja, já são voltadas para o usuário final, acaba havendo uma incompatibilidade de interesses. Um exemplo é a possibilidade das interfaces fornecidas pelos *frameworks* e pelos componentes legados serem incompatíveis, prejudicando a integração;

distâncias entre *frameworks*[2]: ocorre quando dois *frameworks* são compostos para resolver as funcionalidades de uma determinada aplicação e mesmo assim a estrutura resultante não abrange todas as necessidades da aplicação. Há, assim, a necessidade de desenvolver novas funcionalidades e o gerenciamento dessas novas funcionalidades com os *frameworks* integrados não é trivial;

conflitos de arquiteturas[3]: ocorre quando se deseja compor dois ou mais *frameworks*. Um dos motivos desse problema é que a base do estilo de arquitetura em que os *frameworks* são projetados é diferente.

Os outros dois referem-se ao detalhamento do projeto e aos resultados de alguns dos problemas em um nível de projeto de arquitetura. São eles (Mattsson *et. al.*, 1999):

[2] Do inglês *framewok gap*
[3] Do inglês *architectural mismatchs*

sobreposição de composição de entidades[4]: ocorre quando dois *frameworks* se integrarão e possuem alguma entidade do mundo real (por exemplo, uma classe) semelhante, mas modeladas conforme suas perspectivas. Em uma aplicação, entretanto, essa entidade deve ser modelada por um simples objeto e as duas representações devem ser integradas em apenas uma;

integração da funcionalidade de diferentes *frameworks*: ocorre quando uma entidade de mundo real tem que ser modelada por integração de partes das funcionalidades de *frameworks* diferentes.

Todos esses problemas possuem cinco causas comuns (Mattsson *et. al.*, 1999):

comportamento de coesão: para que uma classe de uma biblioteca de classes ou de um outro *framework* substitua a classe de um *framework* qualquer, esta não somente tem que representar apropriadamente o comportamento do domínio, mas também o comportamento coesivo;

cobertura do domínio: quando há a integração de *frameworks,* tem-se a revestimento de domínios. Algumas vezes tal sobreposição pode ser resolvida facilmente, mas também pode ser tão difícil que compense fazer a aplicação do zero;

intenção do projeto: desenvolvedores devem definir explicitamente a intenção do projeto do *framework* para facilitar a "integração" do *framework* com a aplicação específica. Devem documentar se o *framework* pode ser usado somente para extensão ou se pode ser, também, integrado com outro *framework*. Resumindo, deve-se ter uma documentação completa;

carência de acesso ao código fonte: ter este acesso no nível em que se encontra a tecnologia de *frameworks* é importante, já que a integração de *frameworks* pode requerer a edição do código para adicionar um comportamento necessário a um outro *framework*;

carência de normas (padrões) para os *frameworks*: como os *frameworks* orientados a objetos são ainda uma tecnologia nova, não existe uma padronização para modelar, representar e adaptar os *frameworks* existentes.

[4] Do inglês *overlapping framework components*

A Figura 2.2 apresenta a relação entre os problemas apresentados com os *frameworks* e as suas causas, podendo estas serem primárias ou secundárias, dependendo do problema a ser considerado. Na parte superior da Figura 2.2 estão as causas primárias, na parte central os problemas e na parte inferior as causas secundárias (Mattsson et. al., 1999).

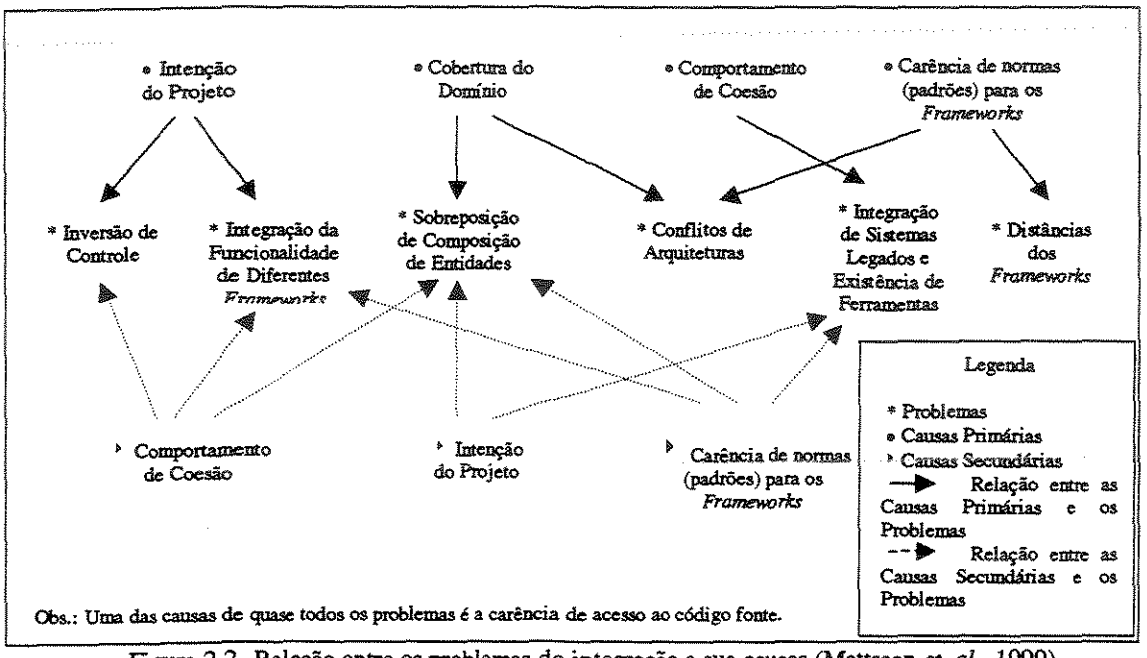


Figura 2.2- Relação entre os problemas de integração e sua causas (Mattsson *et. al.*, 1999)

Em quase todas as soluções propostas por Mattsson *et. al.*, 1999, está envolvida a disponibilidade do código fonte, visto que tendo acesso a tal código, é possível solucionar o problema mesmo que este requeira esforço extra por parte do programador e dificuldade na padronização das versões. Mas ter acesso ao código fonte, na maioria das vezes, não é possível. Já o ponto chave para a maioria dos problemas, é a projeção dos *frameworks* para adaptação e retenção de funcionalidades e não para integração, além da inexistência do desenvolvimento sistemático dos *frameworks*, o que dificulta a união de dois ou mais *frameworks* para o desenvolvimento da aplicação pretendida.

Manutenção de *Frameworks*

O desenvolvimento de um *framework* é um investimento a longo prazo e, como tal, há necessidade de sua manutenção. No início do desenvolvimento de um *framework*, muitos esforços são necessários para as várias interações do projeto. Quando uma mudança é

necessária em uma parte do *framework* ou devido a um erro de projeto ou por se querer adicionar uma nova funcionalidade, surgem os problemas de manutenção. Tais problemas podem ser de dois tipos: (i) de escolha apropriada da estratégia de manutenção, que pode ser uma redefinição do projeto do *framework* ou um trabalho em torno da aplicação específica, e (ii) de incorporação de mudanças no domínio de negócio, que ocorre porque *frameworks* são geralmente desenvolvidos sobre um domínio que é estreitamente relacionado com o domínio de negócio. Entretanto, o domínio é muitas vezes definido resumidamente e evolui com o tempo, havendo, assim, a necessidade de se adaptar o *framework* a essas mudanças (Bosch *et. al.*, 2000).

2.2 – Desenvolvimento Baseado em Componentes

Paralelamente ao surgimento dos *frameworks* e também visando a reutilização de software de forma mais sistemática, o paradigma de desenvolvimento baseado em componentes (DBC) (Brown *et. al.*, 1998) veio propor soluções para diversos problemas encontrados no desenvolvimento orientado a objetos. Uma característica essencial desse novo paradigma é a separação clara entre a definição das interfaces dos componentes de software e suas implementações. O ideal é poder utilizar um componente de software conhecendo-se apenas a definição de sua interface, tornando assim, o sistema construído totalmente independente de qualquer implementação particular de algum componente de software. A reutilização que independe da implementação do componente de software é denominada "Caixa Preta", em oposição à reutilização "Caixa Branca", que depende de como o componente de software é implementado. Essa idéia de classificação a partir do tipo de reutilização fornecida é a mesma descrita na Seção 2.1.1, só que agora aplicada para componentes de software e não *frameworks* (Ye, 2001).

Apesar da nova forma de reutilização de software, denominada de *componentware* (Fuentes, *et. al.*, 2000; Pree, 1997), estar tendo grande ênfase na comunidade especializada, existem muitas questões que ainda estão sendo discutidas; como, por exemplo, o conceito de componente, a criação de metodologias de desenvolvimento de componentes de software e de aplicações baseadas em componentes (Weiss, 2001).

Nos eventos do WCOP (W*workshop on Component-Oriented Programming*), foi definido que um componente de software "é uma unidade de composição com interface

contratualmente especificada e com dependências de contexto explícitas. Os componentes podem ser construídos independentemente e podem estar sujeitos às composições de terceiros". Estipulou-se também que para que algo se torne um componente de software não é necessária a utilização de uma tecnologia específica, desde que se forneça uma interface bem definida (Szyperski *et. al.*, 1996, Szyperski *et. al.*, 1997).

Brown e Wallnau (Brown *et. al.*, 1996) sugerem ainda uma subdivisão da definição do conceito de componente de software:

> **Componente**: uma parte não-trivial de um sistema, praticamente independente, que preenche uma função clara no contexto de uma arquitetura bem-definida;
> **Componente de Software em Execução**: um pacote dinamicamente constituído de um ou mais programas, geridos como uma unidade, ao qual se tem acesso através de interfaces documentadas, que podem ser descobertas durante a execução;
> **Componente de Negócio**: implementação, em software, de um conceito de negócio ou processo de negócio "autômato";
> **Componente de Software**: uma unidade com dependências de contexto apenas explícitas e contratualmente especificadas;

A definição de Componentes dada por Brown e Wallnau assemelha-se à definição dada nos eventos do WCOP. A segunda e a terceira definições de Brown e Wallnau enquadram-se no conceito de Componente Físico dado pela literatura especializada, que se refere aos componentes que possuem um código binário. Já a quarta definição dada, está associada ao conceito de Componente Lógico, que são os componentes que possuem uma interface bem definida e dependências de contexto explícitas através de suas interfaces. Um componente lógico pode ser constituído por um ou vários componentes físicos.

Assim, os pontos centrais para a tecnologia de *componentware* são componentes e suas interações, onde existem vários caminhos para a especificação dos componentes. Conceitualmente um componente é uma unidade independente e coesa de um projeto lógico. Classes, funções e enumerações são entidades lógicas e constituem estes componentes, sendo o encapsulamento das informações, o princípio básico para a construção do seu núcleo (Fuentes , *et. al.*, 2000; Pree, 1997).

Frameworks que oferecem interface de programação bem definida são componentes. Para a obtenção de tais interfaces pode-se utilizar linguagens orientadas a módulos e padrões de componentes, os quais auxiliam, também, a padronização da interoperabilidade entre componentes (Fontoura *et. al.*, 1999, Pree, 1997). Como exemplo de linguagens orientadas a módulos pode-se citar Modula-2 (Woodman, 1990) e ADA (Cohen, 1995) e como exemplo de padrões de componentes CORBA (Sun, 2000), COM (Ewald, 2001) e Java Beans (Sun, 2000).

Neste trabalho, o emprego da palavra componente estará referindo-se à mesma definição dada a componente lógico. A Figura 2.3 mostra um componente lógico e suas interfaces. Essas podem ser de dois tipos: (i) Interface Provida, interface que especifica contratualmente as funcionalidades realizadas pelo componente; e (ii) Interface Requerida, dependência que o componente em questão tem da interface provida por outro componente. O componente descrito (Figura 2.3) realiza as operações contidas nas Interfaces A e B, denominadas de interfaces providas, e depende dos serviços descritos na Interface C, denominada de interface requerida. Nessa representação, como em todas as outras feitas nesta dissertação, foi utilizada a linguagem UML (*The Unified Modeling Language*) (Booch *et. al.*, 2000; Booch *et. al.*, 1997).

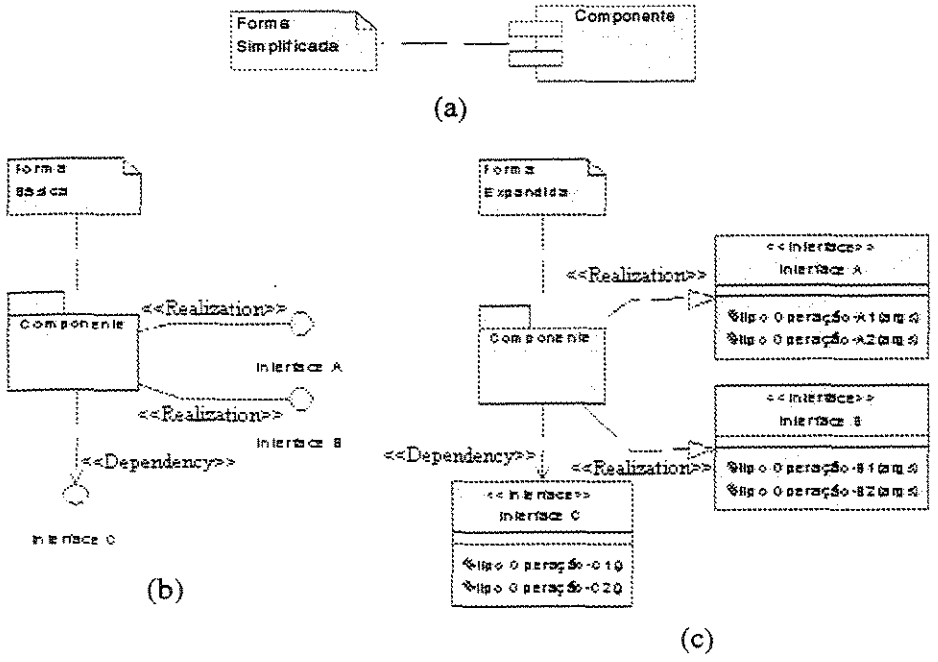


Figura 2.3 – Representação de um Componente Lógico Usando UML

A separação da especificação em diferentes interfaces permite que as dependências entre os componentes sejam restritas às interfaces especificadas e não ao componente como um todo. Com a separação, essa forma de reutilização de software (*componentware)* sustenta a possibilidade da tecnologia *plug* e *play* de software (Pree, 1997). Tal característica permite que muitas adaptações possam estar disponíveis pela simples troca de componentes sem o impacto sobre os clientes desses componentes, desde que esses novos componentes especifiquem, no mínimo, as mesmas interfaces dos componentes originais.

2.3 - Arquitetura de Software

Para lidar com a complexidade e o tamanho de sistemas de software, o aumento do nível de abstração tem sido uma das principais soluções tornando-se necessário, primeiramente, realizar um projeto que descreva a organização do sistema, denominado Arquitetura de Software (Mendes, 2002; Garlan, 1995).

A arquitetura de software de um sistema tem por objetivo representar o maior nível de abstração possível de um sistema, decompondo-o entre os seus elementos computacionais e suas interações. Está relacionada com questões que envolvem a estrutura de controle global de um sistema, os principais elementos de um projeto e a maneira com que eles se comunicam (Shaw, 1996). Garlan e Perry (Garlan, 1995) definiram arquitetura de software como:

> "A estrutura dos elementos de um programa/sistema, seus inter-relacionamentos, princípios e diretrizes guiando o projeto e evolução ao longo do tempo".

A arquitetura de um software constitui um modelo relativamente pequeno, intelectualmente inteligível de como o sistema é estruturado e de como seus elementos trabalham em conjunto, sem se preocupar com o projeto de algoritmos e estrutura de dados (Bass *et. al.*, 1998). Isso permite ao engenheiro de software analisar a efetividade do projeto em satisfazer seus requisitos declarados, considerar alternativas arquiteturais em um estágio em que fazer modificações de projeto é ainda relativamente fácil e reduzir os riscos associados com a construção de um sistema de software (Pressman, 2001).

A definição dada por Brass, Clements e Kazman (Bass *et. al.*, 1998) será empregada para as arquiteturas constituídas de elementos do tipo componente (Arquiteturas Constituídas de Componentes):

> "A arquitetura de software de um programa ou sistema computacional é a arquitetura ou estruturas do sistema que abrangem os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles".

No decorrer desta dissertação, (i) a palavra elemento, mencionada anteriormente nesta seção, refere-se aos vários tipos de elementos arquiteturais de que podem ser constituídas as arquiteturas de software. Por exemplo, Classes, Componentes, etc.; (ii) as palavras arquitetura de software, no presente estudo, quando referidas, serão comprimidas para arquitetura.

2.4 – Padrões de Software e Linguagens de Padrões

O conceito de padrões de software vem do trabalho de Christopher Alexander (Alexander *et. al.*, 1977), que é voltado para as áreas de urbanismo e arquitetura. Segundo Alexander, "cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para esse problema, de forma que você possa utilizar essa solução milhões de vezes sem usá-la do mesmo modo duas vezes". Apesar de Alexander referir-se às soluções relacionadas ao planejamento urbano e à arquitetura de construções, sua definição é perfeitamente aplicável à engenharia de software.

Os padrões de software, como procuram documentar conhecimentos e experiências adquiridos durante anos de prática profissional e ajudar nas soluções de problemas que ocorrem repetidamente em determinados contextos, têm sido amplamente utilizados para auxílio no desenvolvimento de sistemas de software; propiciando assim, desenvolvimento de aplicações de fácil manutenção, elegantes, reutilizáveis e com uma melhor documentação (Gamma *et. al.*, 2000; Buschmann *et. al.*, 1996).

Os padrões de software foram introduzidos inicialmente no paradigma orientado a objetos, oferecendo um vocabulário comum para a comunicação entre os projetistas e criando abstrações em nível superior ao de classes e objetos (Gamma *et. al.*, 2000). Porém, atualmente são utilizados em muitos outros domínios. Desta forma, para facilitar o reuso, pode-se classificá-los em diversas categorias, apesar de não haver muito rigor nessa classificação, devido aos padrões que se encaixam em mais de uma categoria (Braga *et. al.*, 2001). As categorias, segundo Braga (Braga *et. al.*, 2001), são:

Padrões de Processo: definem soluções para os problemas encontrados nos processos envolvidos na engenharia de software: desenvolvimento, testes, etc.;

Padrões Arquiteturais: expressam o esquema ou organização estrutural fundamental de sistemas de software e hardware;

Padrões de Padrões: descrevem o formato em que os padrões devem ser escritos;

Padrões de Análise: descrevem soluções para problemas de análise de sistemas, embutindo conhecimento sobre o domínio específico;

Padrões de Projeto: definem soluções para problemas de projeto de software;

Padrões de Programação: descrevem soluções de programação particulares de uma determinada linguagem ou regras gerais de estilo de programação.

Os padrões podem ser organizados em quatro grupos, sendo um deles denominado linguagem de padrões. Uma linguagem de padrões cobre todos os aspectos importantes de um determinado domínio de aplicação e é completa, apresentando grupos de padrões relacionados que cobrem um domínio específico e representando a seqüência temporal de decisões que levam ao projeto completo de uma aplicação, de forma a tornar-se um método para guiar o processo de desenvolvimento (Braga *et. al.*, 2001, Buschmann *et. al.*, 1996).

Como uma linguagem de padrões possui vários padrões, grande parte da linguagem é formada pelos padrões em si. Outra parte consiste na visão geral do domínio coberto pela linguagem, na sua aplicabilidade e em uma visão geral da interação entre os padrões. Considerando essa segunda parte da linguagem, a literatura especializada sugere um sumário da linguagem, um sumário dos problemas e soluções de cada padrão, um exemplo de aplicação da linguagem é um glossário de termos. O sumário da linguagem deve explicar o porquê de os padrões estarem juntos, as linhas comuns encontradas em mais de um padrão e como os padrões podem trabalhar juntos na execução de algo útil. O sumário dos problemas e soluções consiste em uma tabela contendo um resumo de cada problema e da solução correspondente. Um exemplo deve ser escolhido para ilustrar, de preferência, todos os padrões da linguagem. Um glossário de termos deve explicar a terminologia que não é familiar aos usuários (Braga *et. al.*, 2001).

As outras três formas dos padrões serem organizados são: (i) em coleção, que é um apanhado de padrões que não possuem nenhum vínculo e, geralmente, nenhuma padronização no formato de apresentação; (ii) em catálogo, que apresenta vários padrões, cada um de uma

forma relativamente independente e (iii) em sistemas, que são vistos como uma linguagem de padrões incompleta, pois apresentam um relacionamento entre padrões que cobrem apenas alguns aspectos específicos de um domínio (Braga *et. al.*, 2001, Buschmann *et. al.*, 1996).

No decorrer desta dissertação, a palavra padrão estará se referindo à mesma definição dada a padrões de análise e esses, neste estudo, são desenvolvidos a partir do paradigma orientado a objetos.

2.5 - Comparação Entre Diversas Formas de Reuso

A comparação efetuada nesta seção visa esclarecer as dúvidas mais freqüentes em relação às semelhanças e diferenças entre as várias formas de reuso propostas para a obtenção da reutilização de software de forma mais sistemática. Dentre as formas de reuso, podem-se citar, os *frameworks*, componentes de software, padrões de software, linguagens de padrões, entre outros. Em termos de comparação, este estudo destaca dois trios importantes para o discernimento da dissertação: *Frameworks*/Componentes de Software/Bibliotecas de Classes e *Framewoks*/Linguagem de Padrões/Padrões de Software, abordados nas Seções 2.5.1 e 2.5.2, respectivamente. Comparações para as demais combinações não são aqui apresentadas, já que pouco têm a acrescentar aos conceitos já vistos.

2.5.1 – Frameworks x Componentes de Software x Bibliotecas de Classes

A reutilização de software tem vantagens óbvias, tais como; redução no tempo de desenvolvimento e no custo de manutenção e um impacto positivo sobre a qualidade do sistema (softwares, ou "pedaços" desses, já testados contribuem para a corretude de outros softwares) (Foote *et. al.*, 1998; Pree, 1997).

Há vários meios de se conseguir a reutilização do sistema. Dentre estes, tem-se a utilização de (i) *frameworks* (Seção 2.1), que correspondem a um projeto de alto nível, consistindo em classes abstratas e concretas que especificam uma arquitetura para aplicações e fornecem uma reutilização de alta granularidade; (ii) Componentes de Software (Seção 2.2), que são unidades vindas da decomposição funcional do sistema, que possuem interface e funcionalidade bem definidas, abstraindo um conjunto de classes, fornecendo uma reutilização de baixa granularidade e (iii) Bibliotecas de Classes, que são apenas um conjunto de classes, não necessariamente relacionadas, que fornecem um conjunto de serviços

disponibilizados através da interface pública de suas classes, fornecendo uma reutilização de mínima granularidade. Pode-se considerar que os *frameworks* são uma evolução dos componentes de software e os componentes de software, uma evolução das bibliotecas de classe em termos de reutilização. A Figura 2.4 resume as diferenças entre estas três abordagens (Braga *et. al.*, 2001; Kulesza, 2000).

	Framework	Componente de Software	Biblioteca de Classes
Nível de Abstração	Funcionalidade bem definida	Funcionalidade bem definida	Não tem funcionalidade bem definida
Tamanho	Geralmente é maior que um componente e é sempre maior que uma classe	Geralmente é menor que um *framework* e é maior ou igual a uma classe	É menor que um *framework* e geralmente menor que um componente
Grau de especialização	É específico de um domínio de aplicação	É específico de um domínio de aplicação	Não é específica de um domínio de aplicação
Nível de Implementação	Implementa uma aplicação completa ou semi-completa	Implementa parte de uma aplicação	Geralmente implementa uma parte de uma aplicação menor do que a implementada por um componente
Reuso	Reusa análise, projeto e código	Reusa análise, projeto e código	Reusa código
Auxílio à Geração	Não se Aplica	Pode ser usado na geração de *frameworks*	Pode ser usada na geração de *frameworks* e componentes
Interação entre objetos pré-definida	Sim	Sim	Não
Comportamento *default*	Sim	Sim	Não
Inversão de controle	Sim	Não	Não

Figura 2.4 – Comparação entre *Frameworks*, Componentes e Bibliotecas de Classes

Os componentes de software podem ser usados em mais de um sistema e têm a funcionalidade bem específica, facilitando o desenvolvimento e o entendimento do software. No entanto, os componentes de software proporcionam uma reutilização de baixa granularidade, ao contrário dos *frameworks,* que possibilitam uma reutilização de alta granularidade. Portanto, a prática de desenvolvimento de *frameworks* utilizando projetos

constituídos por componentes de software (Pree, 1997) é melhor sucedida que a reutilização da tecnologia de componentes de software e que o desenvolvimento de *frameworks* caixas brancas, quando estas duas últimas são aplicadas isoladamente.

Um *framework* constituído por componentes é um *framework* com o maior número de componentes possíveis, com predefinição da cooperação entre eles, tendo o propósito de realizar uma determinada tarefa. A utilização desse tipo de *framework* proporciona uma modificação ou incorporação de funcionalidades mais fáceis porque os componentes abstraem um conjunto de classes, simplificando a arquitetura do *framework*, propiciando, então, um nível de abstração mais alto para o entendimento do *framework* e, assim, facilitando o seu uso.

2.5.2 – Frameworks x Linguagem de Padrões x Padrões de Software

Como visto na Seção 2.4, as linguagens de padrões são constituídas de padrões de software interrelacionados, com o objetivo de cobrirem completamente um domínio específico de aplicação. Já as Seções 2.1 e 2.4 apresentam implicitamente o relacionamento bidirecional entre *frameworks* e linguagens de padrões, relatado em Brugali (Brugali *et. al.*, 2000). Por um lado, uma linguagem de padrões oferece as regras para uso dos elementos do *framework* e para sua extensão. Por outro lado, um *framework* permite a implementação de uma aplicação projetada seguindo a linguagem de padrões.

Geralmente, há questionamentos sobre em quais pontos os *frameworks* diferem dos padrões de software e das linguagens de padrões, já que os mesmos possuem algumas similaridades, como, por exemplo, fornecer um nível de abstração que facilite o reuso de práticas já realizadas. Desta forma, a Figura 2.5 apresenta os principais relacionamentos e diferenças entre *frameworks*, linguagens de padrões e padrões de software (Braga *et. al.*, 2001, Brugali *et. al.*, 2000, Johnson, 1997). Alguns resultados apresentados são fontes de estudo para se saber como tirar melhor proveito desses relacionamentos.

	***Framework* (F)**	**Linguagem de Padrões (LP)**	**Padrão de Software (PS)**
Nível de Abstração (na)	F.na< LP.na< P.na	F.na< LP.na< P.na	F.na< LP.na< P.na
Grau de especialização (ge)	É específico de um domínio de aplicação	É específico de um domínio de aplicação	Não é específico de um domínio de aplicação
Auxílio à Documentação (ad)	Não se Aplica	Pode auxiliar a documentação de *frameworks*, já que fornece uma seqüência de decisões que geram o projeto do *framework*.	Pode-se identificar os padrões utilizados no desenvolvimento do *framework*, o que ajuda a entender, de uma só vez, partes maiores do sistema
Reuso (r)	Reusa análise, projeto e código	Reusa o projeto lógico do sistema	Reusa o projeto lógico e partes isoladas de um sistema
Auxílio à Geração de Aplicação (aga)	F.aga> LP.aga> P.aga	F.aga> LP.aga> P.aga	F.aga> LP.aga> P.aga
Auxílio à Extensão	Não se Aplica	Pode oferecer regras para a extensão dos *frameworks*	Pode ser usado nas extensões dos *frameworks*, pois pode resolver problemas encontrados na especificação das extensões
Auxílio à Geração	Pode auxiliar na geração da linguagem de padrões, quando sofrem uma reengenharia, já que possuem em seus elementos de alto nível a concretização das abstrações encontradas no Domínio de aplicação	Pode ser usado na geração dos *frameworks*, já que contém as principais abstrações encontradas no domínio de aplicação, as quais dão origem aos elementos de alto nível do *framework*	É usado na geração de Linguagens de Padrões e pode ser usado no desenvolvimento do *framework*

Figura 2.5 – Comparação entre *Frameworks*, Linguagens de Padrões e Padrões de Software.

O projetista de um *framework* aposta que uma arquitetura (Seção 2.3) funcionará para todas as aplicações do domínio, visto que qualquer mudança substancial no projeto do *framework* reduziria seus benefícios consideravelmente, uma vez que a principal contribuição

de um *framework* para uma aplicação é a arquitetura que ele define (Seção 2.1). Portanto, é imperativo projetar o *framework* de maneira que ele seja tanto flexível e extensível quanto possível (Gamma *et. al.*, 2000) e necessário. Para isso, podem-se utilizar padrões de software (Seção 2.4) que ajudam a tornar a arquitetura do *framework* adequada a muitas aplicações, sem necessidade de reformulações.

Entretanto, geralmente um padrão de projeto não tem um domínio específico de atuação como os *frameworks* obrigatoriamente o têm. Para restringir o uso de determinados padrões dentro da arquitetura de um *framework*, pode-se agrupá-los estruturalmente, apoiando-se uns nos outros, com o intuito de transformar os requisitos e restrições do problema em uma arquitetura (Braga *et. al.*, 2001), criando a linguagem de padrões (Seção 2.4).

Com isso, pode-se citar como alguns dos benefícios da utilização de uma linguagem de padrões no desenvolvimento de um *framework*, a facilidade de tornar a arquitetura modular como guia para o processo de desenvolvimento e a possibilidade de incorporar novos requisitos e restrições sem interferir nos que já existiam.

2.5.3 - Implicações das Comparações Realizadas

Para a obtenção de um nível intermediário de abstração entre a linguagem de padrões e a arquitetura do *framework* OO, deve-se pensar em refinar os padrões da linguagem com o intuito de criar módulos um pouco mais concretos, denominados componentes (Seção 2.2).

Uma das formas de obtenção correta e de maneira facilitada dos elementos desse novo nível de abstração arquitetural está baseada na utilização da linguagem de padrões desenvolvida para o domínio do *framework* que se pretende criar. Para isso, é necessário que se tenha toda a documentação da linguagem (Seção 2.4), pois com ela é possível saber quais são e porquê se dá a relação entre os padrões. Com uma linguagem de padrões devidamente documentada, pode-se agrupar, ou não, os padrões em um único componente, de acordo com a afinidade na solução de problemas. Pode-se, também, determinar a interface de cada um desses "grupos" de padrões através do sumário dos problemas e das soluções, além de verificar essa nova arquitetura, constatando se os exemplos dados para ilustrar cada padrão continuam sendo válidos para exemplificar a arquitetura.

Dessa forma, a independência funcional é conseguida com o desenvolvimento de componentes de "finalidade única" e certa "aversão" à interação excessiva com outros componentes (Pressman, 2001). Como o conceito de independência funcional é uma

decorrência direta da modularidade e dos conceitos de abstração e encapsulamento de informações, consegue-se deixar o número de módulos (componentes) na região ótima (Figura 2.6).

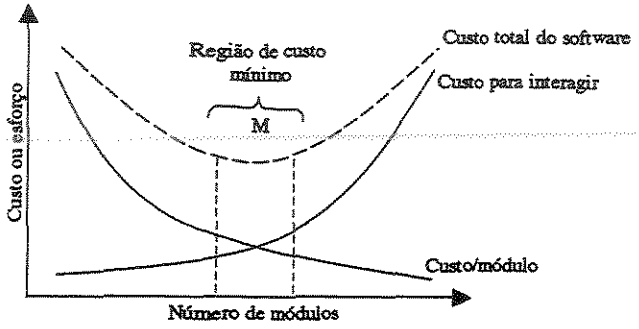


Figura 2.6 – Modularidade e Custo de Software

As curvas mostradas na Figura 2.6 de fato fornecem diretrizes úteis quando a modularização é considerada. Deve-se modularizar, mas tomar o cuidado de se manter próximo a *M*, visto que a submodularização e a supermodularização devem ser evitadas. No primeiro caso, o custo por módulo fica muito alto e, no segundo caso, o custo de integração dos módulos aumenta consideravelmente à medida que aumenta o número de módulos.

Assim, conclui-se que para projetar software constituído por componentes dentro de um custo-benefício favorável, de maneira que cada componente cuide de uma subfunção específica dos requisitos e tenha uma interface simplificada vista de outras partes da estrutura do programa, pode-se utilizar uma linguagem de padrões devidamente documentada.

A Figura 2.7 apresenta graficamente o ponto de vista de Braga (Braga *et. al.*, 2001) sobre o desenvolvimento das aplicações de um *framework* a partir de uma linguagem de padrões, onde segundo Braga (Braga *et. al.*, 2001) "a linguagem de padrões permite documentar e projetar o *framework*. Também ajuda na instanciação de aplicações, que pode ser feita usando o *framework*. O *framework* permite implementar aplicações e dá suporte a linguagens de padrões, implementando os padrões nela contidos. A Figura 2.8 mostra a adaptação feita a esse ponto de vista para incorporar componentes no processo de criação de um *framework* desenvolvido a partir de uma linguagem de padrões. A Figura 2.9 representa a Figura 2.7, agora adaptada para representar o desenvolvimento de um *framework* a partir de uma linguagem de padrões utilizando componentes.

Nas Figuras 2.7 e 2.9 as linhas cheias representam a relação unidirecional entre as terminologias abordadas, representadas pelos retângulos. Já as linhas pontilhadas

unidirecionais representam a possibilidade de uma terminologia auxiliar/facilitar a relação entre outras duas.

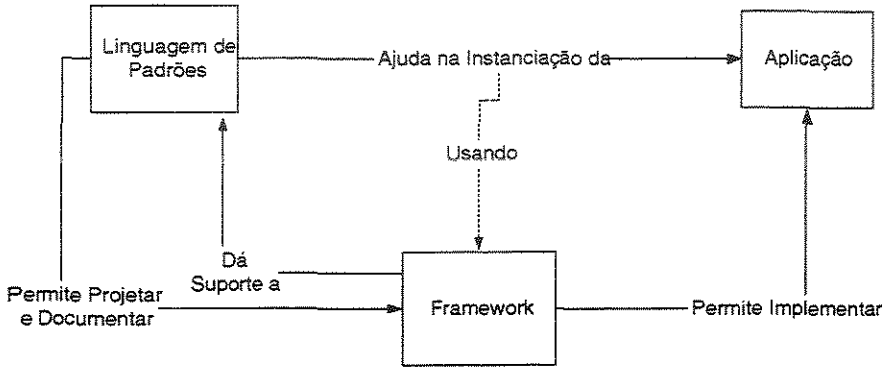


Figura 2.7 – *Frameworks* desenvolvidos a partir de uma Linguagem de Padrões.

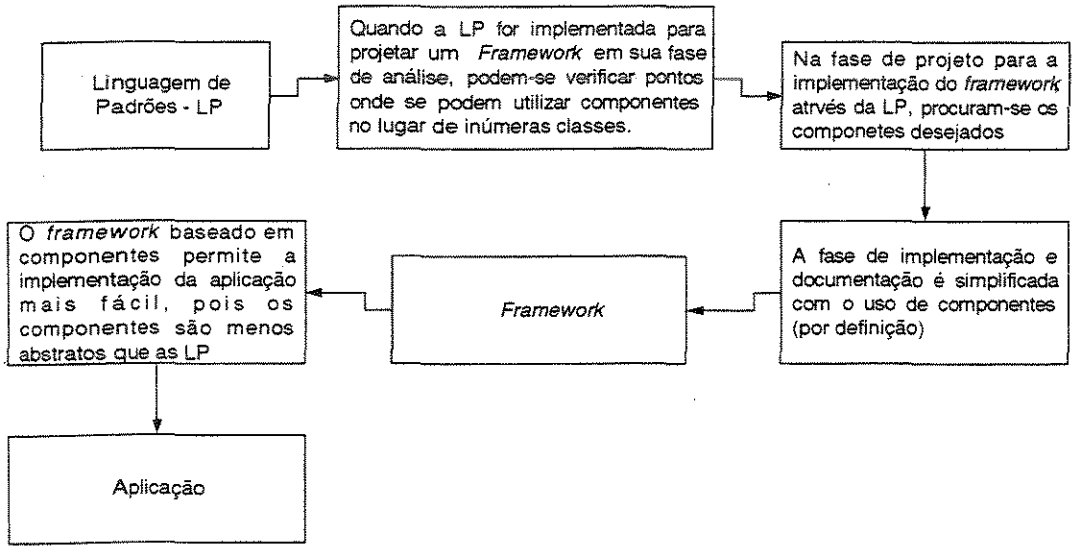


Figura 2.8 – Incorporação de componentes no processo de criação de um *Framework* desenvolvido a partir de uma Linguagem de Padrões.

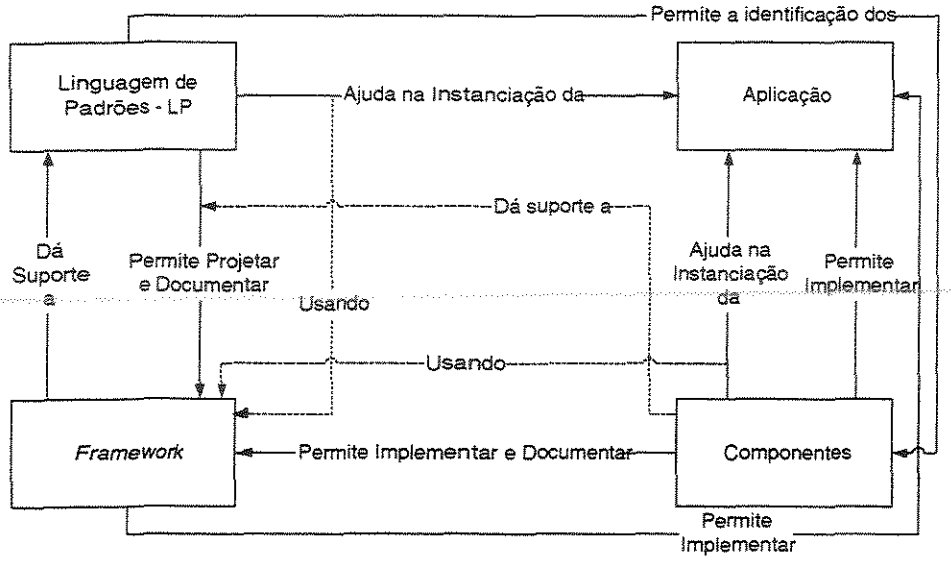


Figura 2.9 – *Frameworks* desenvolvidos a partir de uma Linguagem de Padrões utilizando Componentes.

O presente estudo utiliza a proposta de combinar *frameworks*, componentes e linguagem de padrões para propor algumas contribuições ao estado da arte dos *frameworks* (Seção 2.1), tendo como objetivo, também, sugerir processos para melhor utilização de tal combinação.

2.6 – Considerações Finais

Este capítulo apresentou algumas das várias formas de se reutilizar um software de forma sistemática, dando maior ênfase aos *frameworks*, inclusive mostrando em detalhes o seu estado da arte. A ênfase em *framework* justifica-se por ser este considerado por muitos especialistas a forma mais promissora de se obter a reutilização, reduzindo assim, o tempo para produzir a aplicação.

Quanto às outras formas de reutilização apresentadas (componentes de software, padrões de projeto, linguagem de padrões e arquitetura de software), houve a preocupação de mostrar suas vantagens e desvantagens, expondo como poderiam contribuir para a evolução do *frameworks*.

Em particular, foi demonstrado um relacionamento entre *frameworks* e componentes e outro entre *frameworks* e linguagens de padrões, embora tais relacionamentos ainda sejam fonte de pesquisa para a comunidade especializada. Um terceiro relacionamento, ainda mais novo, foi apresentado: *frameworks*, componentes e linguagens de padrões.

Em suma, este capítulo abordou os conceitos básicos sobre *frameworks*, componentes de software, padrões de projeto, linguagem de padrões e arquitetura de software, comparando-os em grupos, com o intuito de apresentar suas semelhanças e diferenças e demonstrar como podem ser utilizados juntos. Os conceitos e terminologias aqui apresentados, resumidos na Figura 2.10, são de extrema importância para o entendimento da dissertação.

Termo	Definição
Framework	Projeto reutilizável, de parte ou de todo o sistema, que é representado por um conjunto de classes abstratas, organizadas de maneira que suas instâncias possam interagir
Componente	Uma unidade com dependências de contexto apenas explícita e contratualmente especificadas através de suas interfaces
Padrão de Software	Descreve soluções para problemas de análise de sistemas, embutindo conhecimento sobre o domínio específico
Linguagem de Padrões	Conjunto de padrões organizados e interrelacionados que cobrem todos os aspectos importantes de um determinado domínio de aplicação e é completo
Arquitetura de Software	Estrutura dos elementos de um programa/sistema, seus interrelacionamentos, princípios e diretrizes guiando o projeto e evolução ao longo do tempo
Elementos	Vários tipos de elementos arquiteturais pelos quais as arquiteturas podem ser constituídas, por exemplo, Classes, Componentes
Corretude	Correto + -tude, propriedade de estar correto
Componentware	Paradigma de desenvolvimento baseado em componentes
UML	Linguagem de Modelagem Unificada utilizada para a padronização dos modelos OO

Figura 2.10 – Resumo dos Conceitos e Terminologia

No capítulo seguinte, o relacionamento entre *frameworks*, componentes e linguagens de padrões será estudado mais detalhadamente, com o intuito de obter, segundo Pree (Pree *et. al.*, 2000), o suporte tecnológico atual mais promissor de reutilização em larga escala, que é o relacionamento de *frameworks* e componentes; e apresentará, também, os métodos utilizados nesta dissertação para a obtenção e análise das arquiteturas dos *frameworks* a serem estudados.

Capítulo 3

Métodos de Desenvolvimento e Análise de Arquitetura de Software

A utilização de métodos na Engenharia de Software estabelece normas de como construir um sistema de software, baseando-se em uma ação sistemática e não de improvisação. A ação sistemática é obtida através de processos, que são maneiras pelas quais se realiza uma operação, seguindo determinadas regras (Filho, 2001).

Para realizar a obtenção sistemática das arquiteturas dos *frameworks* a serem analisados neste estudo, dois métodos foram desenvolvidos:

1) Método 1: método para o entendimento e levantamento da arquitetura do *framework* a ser estudado (Seção 3.1), através da documentação existente;
2) Método 2: método para o desenvolvimento de uma arquitetura de *framework* estruturada em componentes (Seção 3.2), realizando uma reengenharia na arquitetura analisada no Método 1 (Seção 3.1).

Um terceiro método foi utilizado para a obtenção de uma arquitetura de *framework* baseada em componentes:

3) Método 3: método proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001) e adaptado pelo autor (Seção 3.3).

Outros métodos foram utilizados, também, para analisar as arquiteturas dos *frameworks* tanto quantitativamente, quanto qualitativamente. O método de análise quantitativa foi desenvolvido pelo autor, visto que na literatura especializada não foi encontrado nenhum método que satisfizesse as necessidades do estudo a ser realizado. Já para a análise qualitativa, optou-se por utilizar o ATAM, que consiste em uma técnica de estruturação para compreender os compromissos inerentes nas possíveis arquiteturas de um sistema de software (Kazman *et. al.*, 1998; Barbacci *et. al.*, 1998). Esses métodos são apresentados na Seção 3.4. Todos esses métodos de análise são agrupados em um único método:

4) Método 4: método para avaliação das arquiteturas (Seção 3.4) que utiliza em sua avaliação dois métodos, um de análise qualitativa (Seção 3.4.1) e outro de análise quantitativa (Seção 3.4.2)

3.1 – Método 1: Entendimento e Levantamento da Arquitetura do Framework Original

O entendimento do *framework* a ser estudado, denominado *Framework* Original, é necessário para a obtenção de novas arquiteturas. Para tal entendimento, deve-se levar em consideração os seguintes passos:

- **Análise do Inventário:** Consiste em estudar toda a documentação existente do *framework* e organizar toda a informação de acordo com o que se pretende com a reengenharia. No caso específico deste estudo, deve-se identificar o domínio do *framework*, a que ele se propõe, a ordem em que suas funcionalidades podem ser executadas, a dependência entre essas funcionalidades e qual a relação das partes da arquitetura menos abstrata do *framework* com os itens anteriores. A existência de uma arquitetura de software documentada (representada graficamente) e de uma linguagem de padrões referente ao domínio do *framework* são de grande valia para esta análise.

- **Reestruturação de Documentos:** A pouca e ineficiente documentação dos sistemas de software existentes é um fato. Nesse caso, há três opções (Pressman, 2001):

1) A criação de documentação consome tempo demais; opta-se por fazer a análise com o que se tem.
2) A documentação precisa ser atualizada, mas os recursos são limitados; a opção é a documentação apenas das partes que estão sofrendo modificação.
3) O sistema é crítico para o negócio e precisa ser documentado; opta-se por limitar a documentação ao mínimo necessário.

Para o âmbito deste estudo, a opção dois é a mais indicada, pois será pouco provável encontrar um *framework* que forneça toda a documentação necessária para realizar a análise do inventário e não há a necessidade de documentar o *framework* por completo, já que o interesse deste estudo é avaliar sua arquitetura.

Nesta parte do estudo, a engenharia reversa é o processo de análise de um programa, com o esforço de representá-lo em uma abstração mais alta do que a que ele está atualmente representado. Assim, para a criação apenas da documentação necessária para melhorar o entendimento do software, deve-se efetuar a engenharia reversa no nível imediatamente inferior da documentação do nível que se pretende, para a obtenção da parte da documentação necessária. Caso a documentação de nível imediatamente inferior não seja suficiente, pode-se utilizar toda a documentação para a obtenção da documentação pretendida, Figura 3.1.

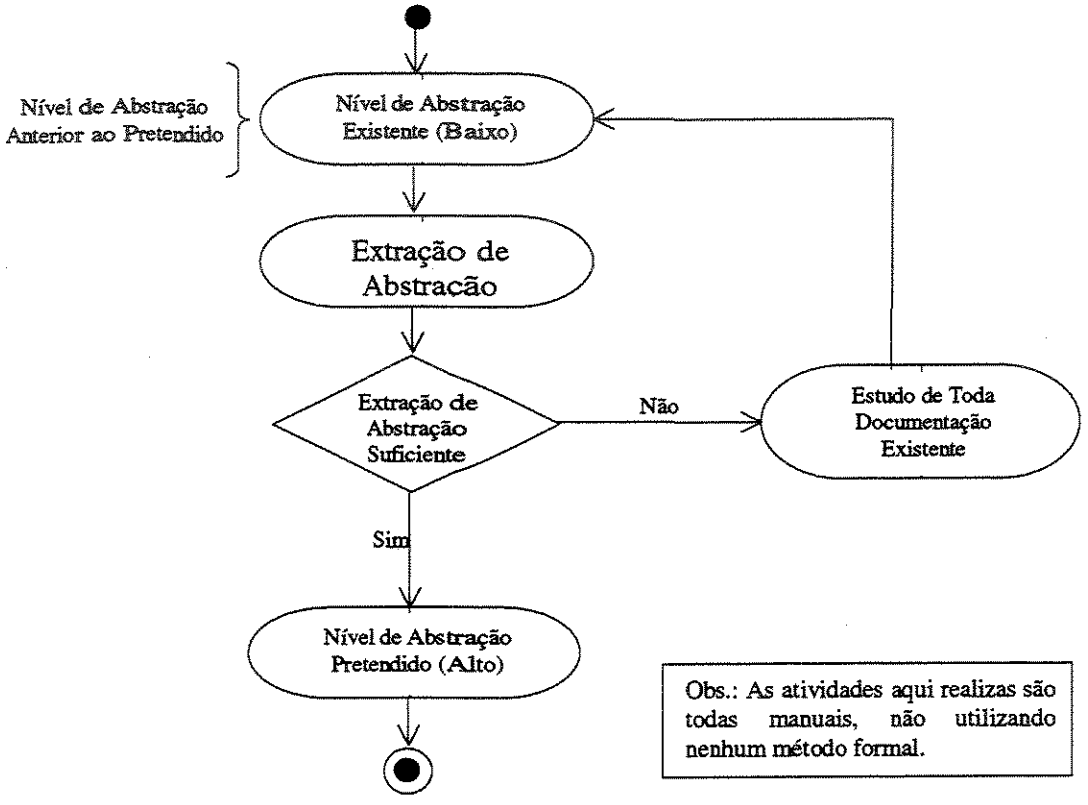


Figura 3.1- Processo de Engenharia Reversa

Este último passo deve ser executado quantas vezes forem necessárias para se obter toda a documentação para a análise do inventário (Figura 3.2), que é fundamental para a análise do *framework* a ser estudado.

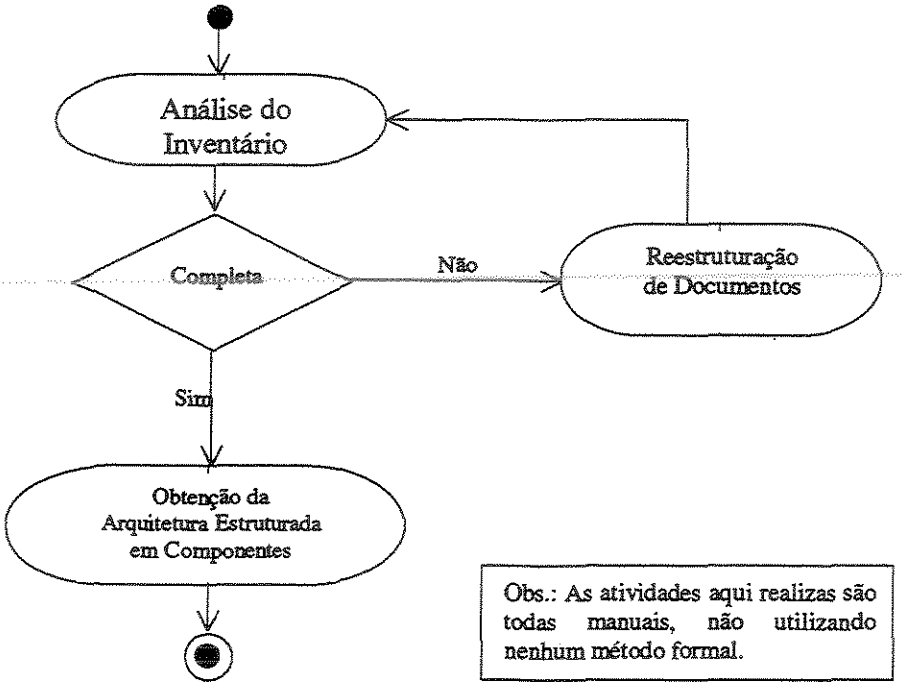


Figura 3.2- Análise do *Framework* Original

A análise do *Framework* Original consiste em conhecer as características do domínio da aplicação e compreender a linguagem de padrões de tal domínio. Caso a linguagem de padrões não esteja escrita, é necessário um levantamento desses padrões e seus relacionamentos a partir da documentação e do código-fonte disponíveis. Como resultado dessa análise, obtêm-se os requisitos, conceitos e relações entre os requisitos do domínio.

3.2 – Método 2: Projeto da Arquitetura Estruturada em Componentes

A arquitetura estruturada em componentes é obtida através de uma reengenharia do *Framework* Original. O método dessa reengenharia (Figura 3.3) foi influenciado por alguns passos propostos por Kulesza (Kulesza, 2000), combinados com algumas fases de desenvolvimento de um framework propostas por Fayad (Fayad *et. al.*, 1999), além de algumas diretrizes para construção de *frameworks* OO propostas por Gurp (Gurp *et. al.*, 2001) (Figura 3.4).

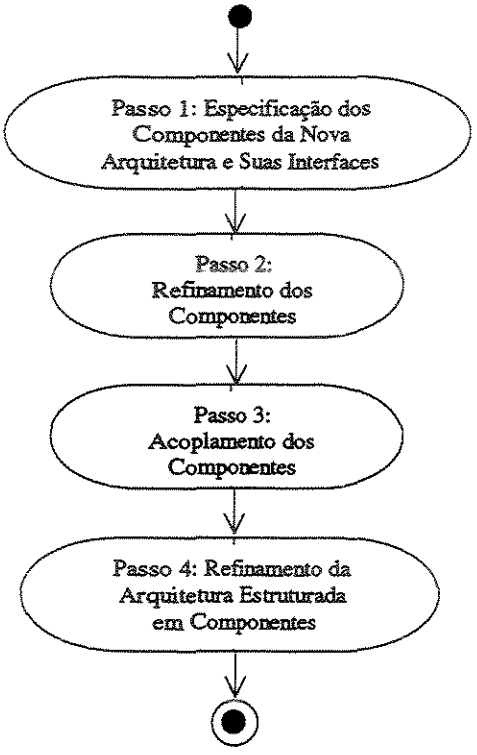


Figura 3.3 – Método de Reengenharia da Arquitetura Original do *Framework*

Passos propostos por Kulesza (Kulesza, 2000) para analisar o impacto da programação orientada a aspecto e a programação adaptativa com relação ao contexto no projeto de sistemas adaptáveis	**Fases de desenvolvimento de um framework, Fayad (Fayad *et. al.*, 1999).**	**Diretrizes para a construção de frameworks OO Gurp (Gurp *et. al.*, 2001), que influenciaram o processo de reengenharia deste estudo**
Compreensão das características de cada sistema	Análise do domínio	A interface de um componente mostra-se separada de sua implementação
Entendimento do código-fonte de cada sistema	Projeto de arquitetura	Uso de componentes pequenos
Engenharia reversa dos sistemas estudados	Projeto do *framework*	
Reengenharia do projeto dos sistemas usando separadamente cada uma das técnicas estudadas	Implementação do *framework*	
Comparação do projeto original com os novos sistemas	Teste do *framework*	
	Documentação	

Figura 3.4 – Influenciadores do Processo de Reengenharia

3.2.1 - Atividades do Método da Reengenharia:

Passo 1: Especificação dos Componentes da Nova Arquitetura e Suas Interfaces

1) A partir da linguagem de padrões, identificar na arquitetura original do *framework* as classes que "materializam" cada um dos padrões (conjunto de classes de um padrão)

2) Definir uma interface para cada um dos papéis de uma classe pertencente a um padrão, contendo apenas as operações previstas naquele padrão. Para isso, utilizam-se os atributos e métodos contidos nas classes do padrão em questão e a definição do padrão feita na linguagem de padrões.

O resultado dessa etapa é um conjunto de componentes isolados, em que praticamente cada componente corresponde a um padrão da linguagem de padrões.

Passo 2: Refinamento dos Componentes

A partir dos componentes da etapa anterior, deve ser feita uma análise da interface de todos os componentes, preocupando-se em identificar possíveis interfaces repetidas e interfaces que dependam em quase sua totalidade de um mesmo conjunto de classes. Após a identificação, um novo conjunto de componentes é gerado, unindo funcionalidades comuns de alguns padrões em um único componente e criando novos componentes que darão suporte aos componentes que "contêm o cerne dos padrões".

Ao determinar cada um desses componentes, deve-se levar em consideração um dos dois fatos descritos a seguir, para se garantir a coesão satisfatória de cada um dos componentes (Pressman, 2001):

1) O componente é constituído por mais de uma tarefa. As tarefas devem estar relacionadas e devem ter a necessidade de serem executadas em uma ordem específica.

2) O componente é constituído por apenas uma tarefa.

A linguagem de padrões e a arquitetura original do *framework* ajudam na determinação (criação) dos novos componentes, já que a linguagem de padrões possui a funcionalidade de cada padrão, a relação (dependência) entre eles e a ordem em que eles têm que ser executados; e a arquitetura original do *framework* possui, entre outras coisas, as classes necessárias para a união desses componentes.

Com a formação do novo conjunto de componentes, a relação entre componentes e padrões, que era praticamente de um para um, não existe mais. Isso se deve ao fato de um componente poder conter o cerne de vários padrões, de as tarefas comuns dos padrões estarem em um único componente e de os componentes realizarem tarefas necessárias ao domínio, mas não pertencerem ao cerne do padrão.

O resultado dessa etapa (Figura 3.5) é um conjunto de componentes que realizam um conjunto mínimo de tarefas necessárias ao domínio do *framework*.

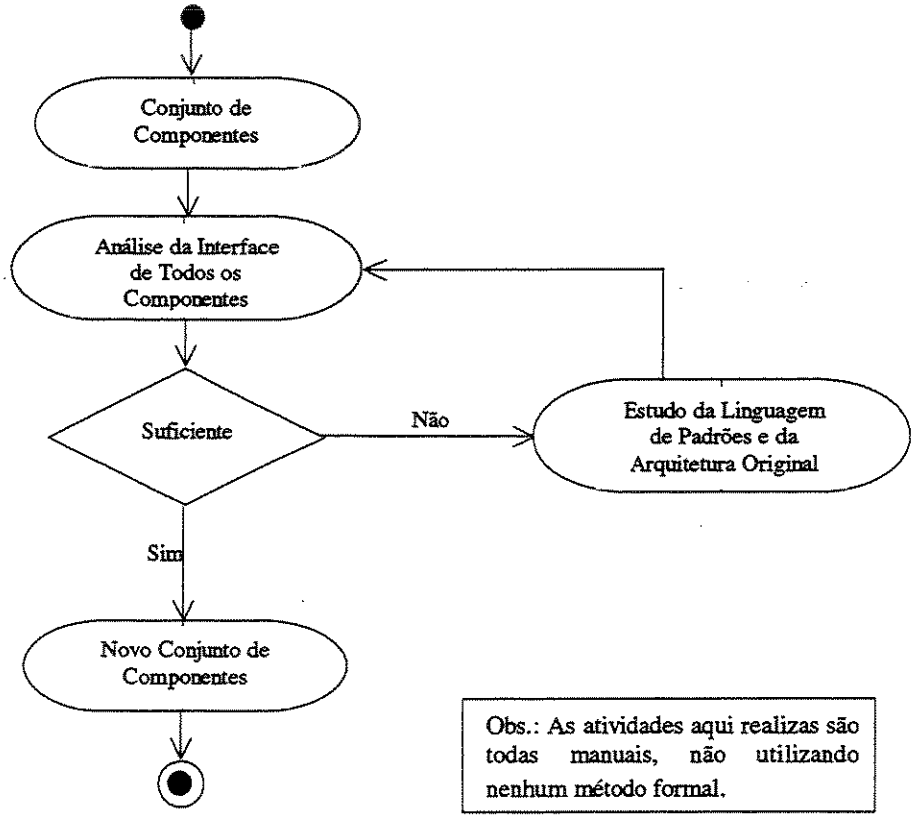


Figura 3.5 – Processo de Refinamento dos Componentes

Passo 3: Acoplamento Dos Componentes

Com os componentes devidamente identificados e com suas respectivas interfaces determinadas, o próximo passo é verificar se existe dependência entre os componentes, ou seja, se os dados de entrada de uma interface de um componente são gerados e fornecidos por uma interface de outro componente.

Para facilitar essa verificação, o seguinte processo é proposto:

1) Gerar uma tabela com duas colunas. Na primeira coluna, em cada uma de suas linhas, encontra-se cada um dos componentes. Na segunda coluna, encontram-se os padrões que têm pelo menos um de seus elementos associados a uma das tarefas do componente em questão.
2) Utilizando a tabela gerada anteriormente, a linguagem de padrões e as assinaturas das interfaces de cada componente, determinar qual interface de um componente depende da interface de outro componente qualquer. Para realizar tal determinação as interfaces de componentes diferentes são organizadas de duas a duas, observando, na tabela, os padrões associados a cada componente; na linguagem de padrões, a dependência entre os padrões e; nas assinaturas das interfaces em questão os dados de entrada necessários e os dados de saída gerados. Esse passo deve ser realizado até que todas as interfaces de componentes distintos tenham sido avaliadas.
3) Com a dependência entre as interfaces dos componentes, consegue-se o primeiro esboço da Arquitetura Estruturada em Componentes fazendo a relação de dependência entre os componentes.
4) Algumas decisões arquiteturais podem ser tomadas, com o objetivo de tornar um componente dependente do menor número possível de outros componentes.

4.1) Para a realização de tais decisões, "componentes-pontes" devem ser identificados para que se consiga transmitir dados de um componente para outro sem haver a necessidade de comunicação entre eles.

4.2) Para a identificação desses componentes deve-se levar em consideração o nível de abstração de cada componente em relação ao domínio da aplicação. Assim, o componente do nível "N" oferece um conjunto de serviços aos componentes de nível superior (isto é, N+1). Para tanto, o componente de nível "N" utiliza suas funções bem como faz uso dos serviços disponíveis no nível inferior (N-1).

5) Uma representação das camadas da arquitetura do *framework* deve ser feita:

5.1) Identificar cada uma das camadas, nomeando-as e conceituando-as.

5.2) Determinar a que camada pertence cada um dos componentes e descrever a funcionalidade de cada um deles.

5.3) Determinar em que camada(s) a aplicação pode ser instanciada.

O resultado desse passo é uma arquitetura do *framework* em camadas estruturada em componentes.

Passo 4: Refinamento da Arquitetura Estruturada em Componentes

A partir da arquitetura obtida no passo anterior, é feito um refinamento na arquitetura, acrescentando componentes ou interfaces necessárias para suprir alguma(s) funcionalidade(s) do domínio que essa nova arquitetura não abrange.

Para a realização dessa etapa propõe-se as seguintes etapas (Figura 3.6):

1) Comparam-se os resultados da Análise do *Framework* a Ser Estudado (Seção 3.1) com o que a arquitetura obtida na seção anterior consegue suprir, identificando os requisitos que faltam ser satisfeitos.
2) Analisa-se a estrutura de cada um dos componentes para saber se o componente tem como suprir algum requisito que falta. Em caso afirmativo, é acrescentado mais um à interface para cada um dos requisitos que esse componente resolve. Logo após, é necessário retornar ao passo Acoplamento de Componentes (Passo 3).
3) Se, após a realização da etapa anterior, algum(ns) requisito(s) não tiver(em) sido satisfeito(s), há a necessidade de se criar novo(s) componente(s) para solucionar tal problema. Após a criação de novos componentes, é necessário retornar ao passo Refinar os Componentes (Passo 2).

O resultado desse passo é o conjunto (i) das funções necessárias do domínio do *framework*, disponibilizadas pelos componentes dessa nova arquitetura a partir de suas interfaces de reutilização e (ii) de regras de projeto que devem ser seguidas por seus usuários. Cada um desses componentes com várias interfaces são os *framelets* definidos por Gurp (Gurp *et. al.*, 2001), que são pequenos *frameworks* com interfaces bem definidas.

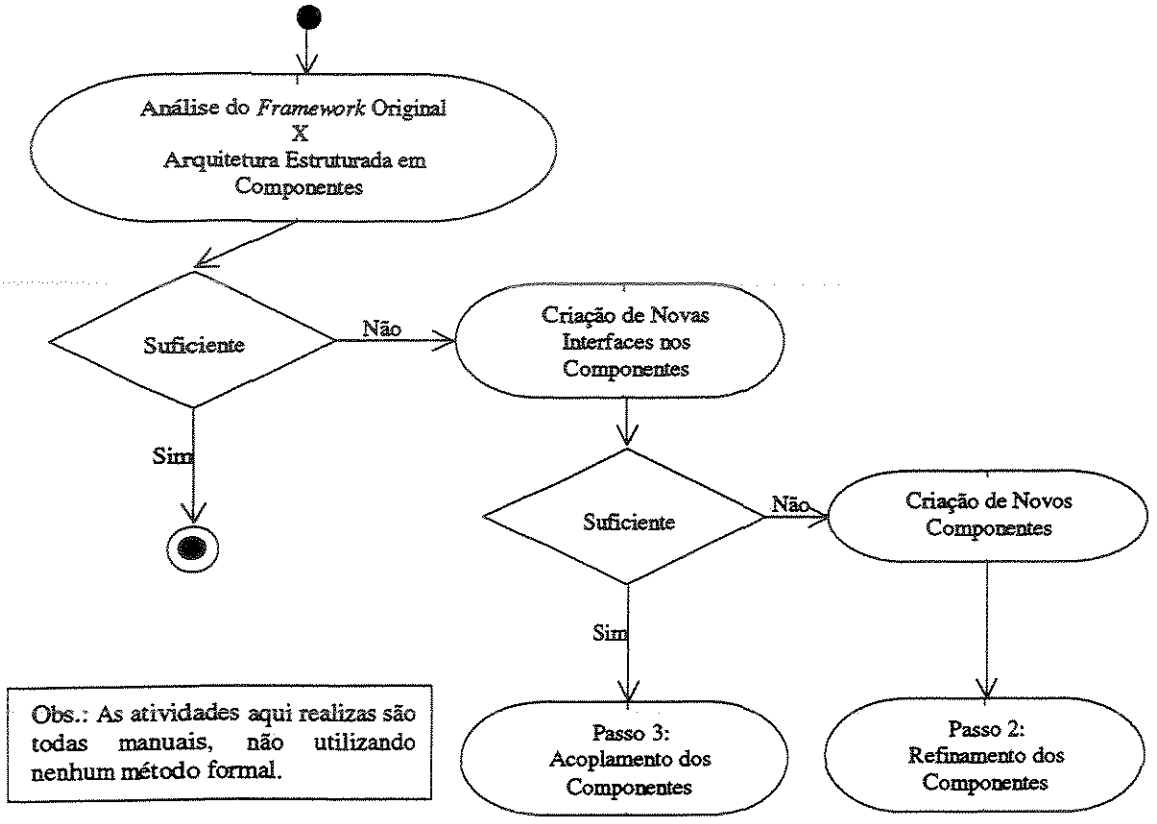


Figura 3.6 – Processo de Refinamento da Arquitetura Estruturada em Componentes

3.3 – Método 3: Projeto da Arquitetura Baseada em Componentes

A Arquitetura Baseada em Componentes é obtida através de parte do processo de especificação de software baseado em componentes, proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001) com a sugestão de utilizar linguagens de padrões para auxiliar a sua aplicação, aproveitando, assim, as vantagens de quando se tem uma linguagem de padrões voltada para o domínio do *framework* que se pretende criar. Nomeou-se essa arquitetura como baseada em componentes devido ao fato de o processo ser direcionado para o uso de componentes desde o início. A arquitetura obtida anteriormente (Seção 3.2) é denominada Estruturada em Componentes devido à influência da arquitetura original do *framework* baseada em classes em seu processo de criação.

A Figura 3.7 apresenta os passos da especificação de software baseado em componentes, proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001), para o desenvolvimento de um software baseado em componentes. Cada um desses passos é representado por um retângulo com

linhas pontilhadas, com o nome do passo no canto superior direto. Os retângulos com linhas cheias são as atividades ocorridas em cada passo e as linhas unidirecionais que chegam e/ou saem destas atividades, são respectivamente, as informações necessárias para a realização da atividade e as informações geradas por esta.

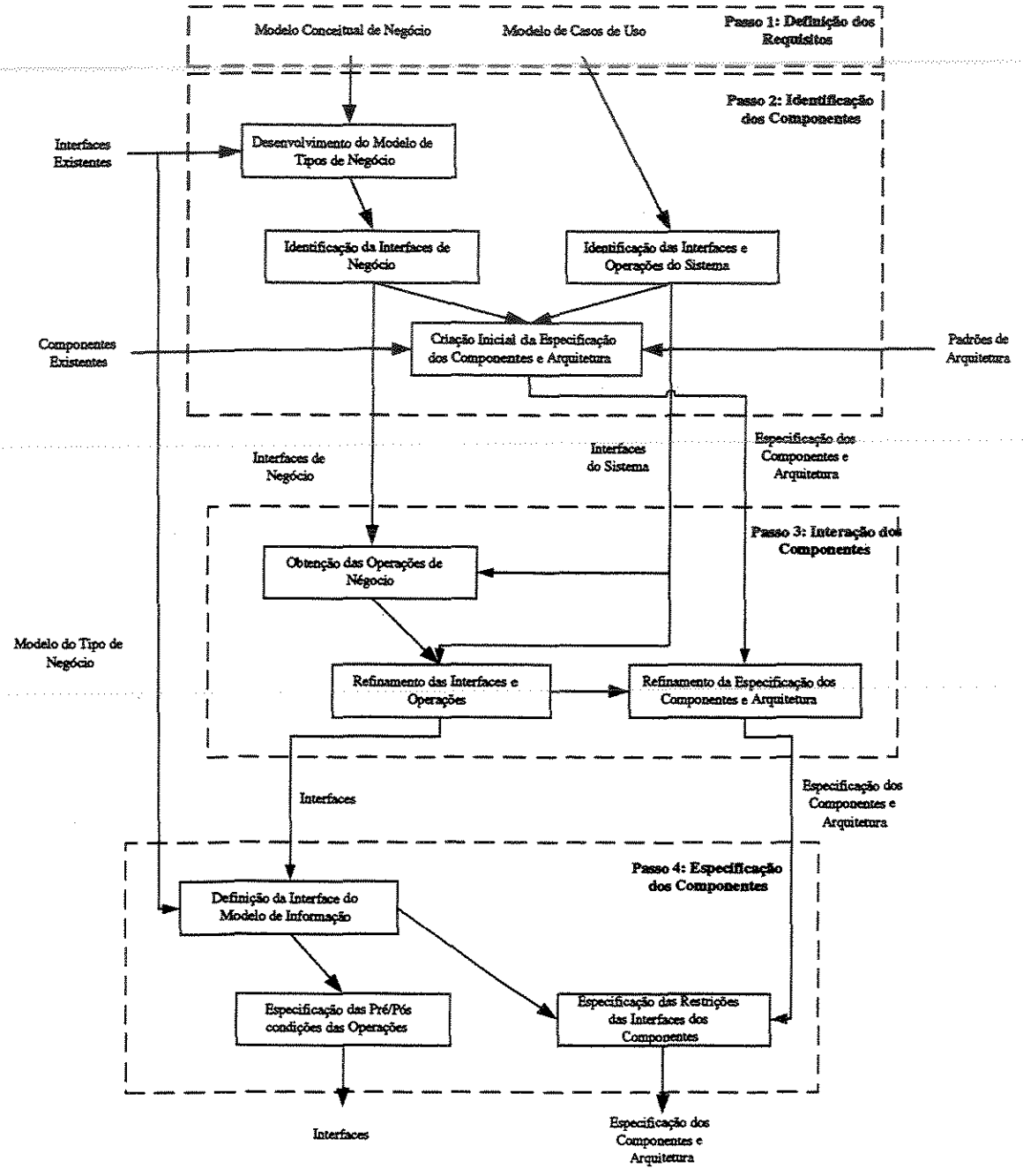


Figura 3.7 – Especificação de Software Baseado em Componentes Proposto Pelo Método UML Components (Cheesman *et. al.*, 2001)

As subseções seguintes descrevem cada um dos passos necessários para a obtenção de uma arquitetura de um *framework* baseada em componentes utilizando o processo proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001) auxiliado por uma linguagem de padrões.

3.3.1 – Passo 1: Definição dos Requisitos

Nesse passo há a preocupação em saber os requisitos, conceitos e relações entre esses requisitos do domínio para o qual se pretende desenvolver uma arquitetura de *framework*. Cheesman e Daniels (Cheesman *et. al.*, 2001) propõem, para tal saber: (i) um mapa com os termos introduzidos no processo de negócio e outros termos importantes, representado a partir de diagramas de classes e a (ii) criação de casos de uso que facilitem o entendimento dos pontos funcionais do sistema, identificando como os atores interagem com o sistema e descrevendo essas interações; denominados respectivamente Modelo Conceitual de Negócio e Modelo de Caso de Uso.

Para a obtenção de tais modelos, a utilização da linguagem de padrões do domínio em que se pretende desenvolver a arquitetura do *framework* é de grande valia, já que ela fornece as funcionalidades que um determinado domínio deve suprir. Pode-se pensar que linguagem de padrões está em um nível de abstração superior aos processos de negócio para os quais se pretende criar um *framework*.

A identificação dos termos do mapa do modelo conceitual de negócio pode ocorrer de duas formas:

1) Para as classes de cada um dos padrões, pode-se criar um processo de negócio, que pode ser representado pelo diagrama de atividades da UML. A partir do processo devidamente representado, faz-se a conceituação de cada um dos termos necessários para o seu entendimento.
2) Tal forma de identificação pode ser usada quando se tem uma linguagem de padrões devidamente documentada (ver Seção 2.4), já que se torna perfeitamente possível identificar todos os papéis que cada classe de um padrão tem e, assim, conceituar cada um dos termos necessários para o entendimento do funcionamento do padrão.

Após a identificação e conceituação de cada termo, fazer a associação entre eles e, por conseguinte, a criação do modelo conceitual de negócio é o próximo passo. Para isso, podem-se utilizar os processos de negócio criados para identificar os termos, como é proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001), ou utilizar o elemento do padrão, denominado por Braga (Braga *et. al.*, 2001) como estrutura, que mostra a relação entre as classes de seu padrão e as aplicações da linguagem de padrões já desenvolvidas, preocupando-se em separar em pacotes os modelos conceituais de negócio criados.

Com o modelo conceitual de negócio desenvolvido, a próxima e última etapa desse passo é a obtenção do modelo de casos de uso, que pode ser realizada de duas formas:

1) Utilizar os processos de negócio criados para identificar os termos, como é proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001), para a obtenção dos atores e papéis de cada um, a identificação dos casos de uso e a descrição dos mesmos (Cheesman *et. al.*, 2001).
2) Utilizar a linguagem de padrões para identificar a necessidade de entrada de cada padrão e o que ele gera de saída e, em seguida, identificar os casos de uso. Utilizar a dependência entre os padrões para determinar a ordem em que os casos de uso devem ocorrer. Utilizando o que foi adquirido anteriormente, descrever cada um dos casos de uso, já determinando a dependência entre eles e separando-os em pacotes, da mesma forma que foi feito na criação do modelo conceitual de negócio.

O resultado desse passo é o Modelo Conceitual de Negócio e o Modelo de Caso de Uso, que são as duas entradas necessárias para estágios da especificação de software baseados em componentes (Cheesman *et. al.*, 2001).

3.3.2 – Passo 2: Identificação dos Componentes

A meta é criar um conjunto de interfaces e especificações de componentes que possam ser agrupados (associadas) dentro de uma arquitetura de componentes inicial.

A ênfase nesse passo é sobre a descoberta – quais informações são necessárias para o gerenciamento do software, quais componentes são necessários para fornecer tais funcionalidades e como eles podem ser agrupados. A pesquisa dessas informações é realizada sob a visão de camadas de arquitetura de uma aplicação, Figura 3.8.

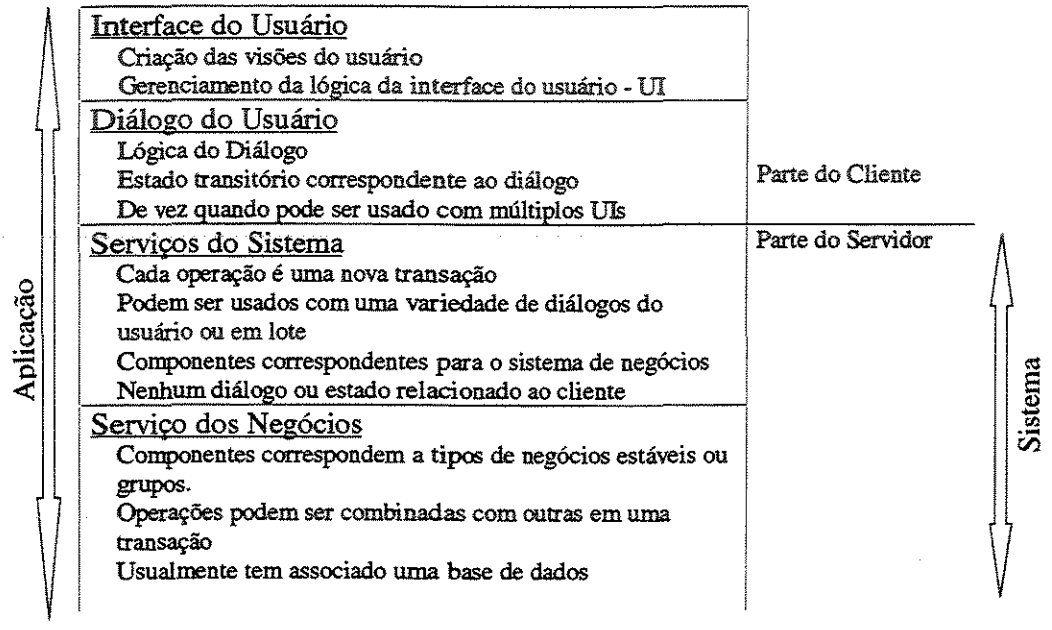


Figura 3.8 – Camadas Arquiteturais (Cheesman *et. al.*, 2001)

O sistema deve ser separado em duas camadas distintas: a camada de sistemas e a camada de serviços de negócio. Tal separação permite a especificação dos *workflows*, que são uma seqüência de atividades com um resultado de valor observável. Nesse passo, serão identificadas as interfaces e os componentes da camada serviço de sistemas e as interfaces de negócio e os componentes de tipo de negócio da camada serviço de negócio.

A identificação das interfaces é feita através da determinação de como o usuário poderá iniciar as operações da aplicação, ou seja, qual "tipo de diálogo" ele pode ter com o sistema. Determinado o tipo de diálogo, faz-se a identificação das interfaces do sistema, utilizando os casos de uso, e das interfaces de negócio, utilizando o modelo conceitual de negócio, Figura 3.9.

Maior detalhe desse processo pode ser encontrado em (Cheesman *et. al.*, 2001). O resultado desse passo é um conjunto de componentes com suas interfaces de sistema e negócio definidas.

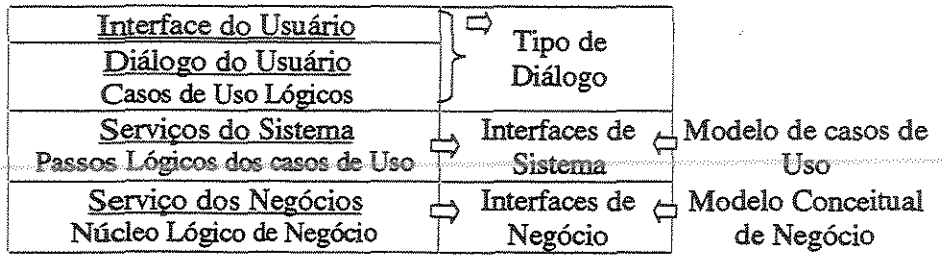


Figura 3.9 – Identificação das Interfaces (Cheesman *et. al.*, 2001)

Com os componentes de sistema e de negócio devidamente representados, uma arquitetura de software inicial é proposta, relacionando os componentes que dependem de uma determinada interface com o componente que oferece aquela interface.

3.3.3 – Passo 3: Interação dos Componentes

A partir das interfaces de negócio e de sistema, é feita a definição das assinaturas de suas operações utilizando o diagrama de colaborações proposto na UML. Na execução do diagrama de colaboração, novas operações podem ser identificadas e, assim, um refinamento na interface de cada um dos componentes é realizado. Após todas as operações de cada interface estarem com suas assinaturas prontas, deve-se ter a preocupação de manter a integridade entre os objetos do sistema. Para identificar possíveis problemas de integridade, utilizam-se os diagramas de colaboração. Para solucionar tais problemas, novas relações de dependência entre as interfaces devem ser criadas e/ou devem-se criar novas operações de interface. Por fim, um refinamento das interfaces deve ser feito, procurando minimizar as chamadas entre interfaces, dependência cíclica entre outros. Maior detalhe desse processo pode ser encontrado em (Cheesman *et. al.*, 2001).

Com as interfaces de cada componente modificadas e prontas, um refinamento (atualização) da arquitetura de componentes inicial deve ser feito.

3.3.4 – Passo 4: Especificação dos Componentes

O objetivo deste passo é definir precisamente os contratos de uso e de realização de cada um dos componentes, utilizando a OCL (*Object Constraint Language*), que é um modelo formal de especificação de software.

Como o objetivo deste estudo é comparar duas arquiteturas e o passo anterior, Interação dos Componentes (Seção 3.3.3), já fornece a arquitetura baseada em componentes com as mesmas especificações da arquitetura estruturada em componentes, não há a necessidade de realizar o restante do processo, incluindo esse passo.

3.4 – Método 4: Avaliação das Arquiteturas

O objetivo inicial era analisar qualitativamente as arquiteturas obtidas nos métodos 1, 2 e 3 e depois expressar os resultados obtidos em números, isto é, realizar a avaliação quantitativa. Entretanto, como será relatado no decorrer desta seção, a literatura disponível para tal análise ainda está em fase inicial, o que possibilitou apenas avaliar quantitativamente cada uma das arquiteturas isoladamente, não permitindo comparar os resultados numéricos obtidos.

O processo de avaliação qualitativa será realizado sobre os requisitos não-funcionais Manutenibilidade e Reusabilidade, utilizando o método de análise de compromissos da arquitetura (ATAM – *Architecture Tradeoff Analysis Method*) (Seção 3.4.1). Já a avaliação quantitativa (Seção 3.4.2) se restringirá à análise do acoplamento e da coesão dos elementos das arquiteturas em questão. Para isso, será utilizada a métrica Fator de Acoplamento, propostas por Harrison, Counsell e Nithi (Harrison *et. al.*, 1998), com uma adaptação em sua equação, e uma discussão sobre a coesão dos elementos das arquiteturas, seguindo a divisão de módulos coesos proposta por Pressman (Pressman, 2001).

Com a análise qualitativa pretende-se compreender as conseqüências de decisões arquiteturais e, assim, direcionar as atividades de aprimoramento de software. Já com a análise quantitativa almeja-se saber se cada uma das arquiteturas avaliadas, em seu paradigma de desenvolvimento, foi bem desenvolvida ou não.

3.4.1 – Método de Análise Qualitativa

Para o processo de avaliação qualitativa, será utilizado o método interativo que avalia os compromissos do projeto desenvolvido pelo SEI (*Software Engineering Institute*), denominado método de análise de compromissos da arquitetura (ATAM – *Architecture Tradeoff Analysis Method*).

O ATAM é um método de análise arquitetural que visa a compreender as conseqüências de decisões arquiteturais em relação aos atributos de qualidade de um sistema. O problema de avaliar uma arquitetura é inverso ao problema de projeto. Os avaliadores precisam entender a arquitetura, reconhecer os parâmetros arquiteturais que estão sendo utilizados, conhecer as implicações desses parâmetros arquiteturais em relação aos atributos de qualidade, comparar as implicações com os requisitos do sistema (Mendes, 2002).

O ATAM pode ser visto como um método de identificação de risco, em que partes com riscos potenciais na arquitetura de um sistema de software são detectadas. Assim, além de elevar a compreensão da arquitetura e melhorar a documentação arquitetural, o ATAM identifica (i) riscos: decisões arquiteturais tomadas para as quais não há entendimento total das conseqüências; (ii) pontos de susceptibilidade: conjunto de pontos da arquitetura que são importantes para a obtenção de algum atributo de qualidade; e (iii) pontos de compromisso: pontos de susceptibilidade que são críticos para a obtenção de múltiplos atributos, afetando-os de forma diferente (Mendes, 2002; Kazman *et. al.*, 1998; Barbacci *et. al.*, 1998).

O ATAM objetiva obter e refinar os requisitos de atributos de qualidade bem como decisões arquiteturais de projeto a fim de que esses elementos possam ser utilizados na avaliação arquitetural. O ATAM possui três aspectos fundamentais sobre os quais o método foi desenvolvido (Mendes, 2002):

1) Caracterização dos atributos de qualidade do sistema de software;
2) Obtenção e análise de cenários a fim de obter um consenso entre os influenciadores do sistema;
3) Análise de decisões arquiteturais utilizando mecanismo de raciocínio, denominado ABAS (*Atribute-Based Architectural Style*), o qual ilustra como decisões arquiteturais podem afetar a operação de atributos de qualidade.

O método de análises de compromissos da arquitetura concentra-se na identificação de metas de negócio as quais conduzem a metas de atributos de qualidade ou (requisitos não-funcionais). O método ATAM faz uso das metas de atributos de qualidade no suporte à análise de como estilos arquiteturais ajudam na obtenção dessas metas. O método de análise ATAM compreende seis passos que são distribuídos em quatro etapas, conforme apresentado a seguir, Figura 3.10 (Mendes, 2002; Kazman *et. al.*, 1998; Barbacci *et. al.*, 1998).

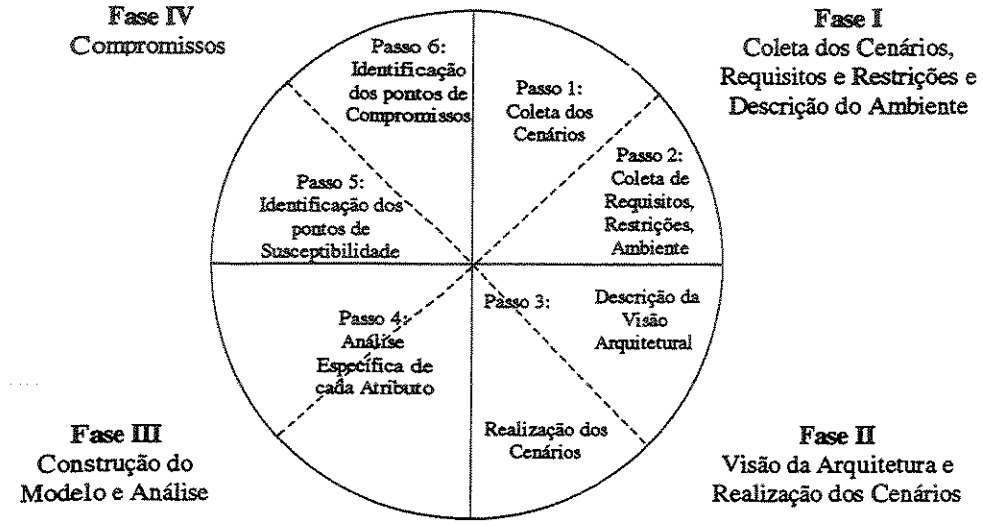


Figura 3.10 – Fases do Método de Análise de Compromissos da Arquitetura (Kazman *et. al.*, 1998)

Fase I - Coleta dos Cenários, Requisitos e Restrições e Descrição do Ambiente

- Passo 1: Coleta dos Cenários: Um conjunto de casos de uso é desenvolvido para representar o sistema do ponto de vista do usuário. Como o sistema a ser avaliado neste estudo é um *framework*, o usuário pode ser um (i) desenvolvedor, que utilizará o sistema para gerar uma nova aplicação (instância) ou (ii) um usuário final, que utilizará o *framework* já instanciado. Desta forma, para a criação dos cenários, deve-se observar se os requisitos não-funcionais são externos ("visíveis" ao usuário final), ou internos("visíveis" ao desenvolvedor).

  Os requisitos não-funcionais a serem avaliados neste estudo são Manutenibilidade e Reusabilidade, visíveis ao desenvolvedor. Por conseguinte,

os cenários a serem desenvolvidos devem relatar a relação desse tipo de usuário com o sistema.

- Passo 2: Obtenção dos Requisitos, Restrições e Descrição do Ambiente: Essa informação é necessária como parte da engenharia de requisitos e é usada para garantir que todas as preocupações do cliente interessado e usuário foram consideradas.

Fase II - Visão da Arquitetura e Realização dos Cenários

- Passo 3: Descrição da Visão Arquitetural: É a descrição dos estilos arquiteturais escolhidos para cuidar dos cenários e dos requisitos. Tais estilos devem ser descritos utilizando perspectivas arquiteturais que representem os requisitos não-funcionais escolhidos para a análise, como:

  Visão de Módulo: para a análise da distribuição de tarefas entre os elementos da arquitetura e o grau em que o encapsulamento de informação foi conseguido.

  Visão de Processo: para a análise do desempenho e da escalabilidade do sistema.

Fase III - Construção do Modelo e Análise

- Passo 4: Análise Específica de cada Atributo: Avaliação dos requisitos não-funcionais (atributos de qualidade) considerando cada um isoladamente. Nesse caso, por exemplo, como a abordagem deste estudo é avaliar a Manutenibilidade do sistema, a arquitetura será submetida a uma análise de manutenção. Como não há modo de medir manutenibilidade de forma direta, medidas indiretas devem ser usadas (Pressman, 2001). Dessa forma, este estudo avaliará, entre outros influenciadores, a independência funcional entre os elementos da arquitetura pois, quanto mais os elementos forem

independentes uns dos outros, mais fácil é a manutenibilidade do sistema com um todo (Pressman, 2001; Mendes, 2002). Essa mesma idéia será utilizada para avaliar a Reusabilidade proporcionada pelas diferentes arquiteturas, já que não há métricas que a avaliem diretamente em nível arquitetural.

Fase IV - Compromissos

- Passo 5: Identificação dos Pontos de Susceptibilidade: Para cada um dos estilos arquiteturais a serem avaliados, deve-se identificar os pontos de susceptibilidade. Isso pode ser conseguido fazendo pequenas modificações na arquitetura e determinando o quanto certo requisito não funcional é sensível a essa modificação. Para isso, o passo quatro é novamente realizado e depois é comparado o resultado anterior com o atual. Quaisquer atributos que são significativamente afetados pela variação da arquitetura são chamados de pontos sensíveis.

- Passo 6: Identificação dos Pontos de Compromisso: Uma vez realizado o passo 5, simplesmente identificam-se os elementos arquiteturais para os quais múltiplos requisitos não-funcionais são suscetíveis (passo 5) para cada um dos estilos arquiteturais a serem avaliados (passo 3). Estes serão os pontos de compromisso.

  Ao fim do sexto passo, um relatório baseado nas informações coletadas com a aplicação do método deve ser elaborado, detalhando os resultados obtidos junto com a apresentação de estratégias a serem adotadas.

3.4.2 - Método de Análise Quantitativa

O trabalho na área de análise quantitativa de projeto arquitetural está ainda nos estágios formativos (Pressman, 2001). Por essa razão, o processo de avaliação quantitativa deste estudo se restringirá à análise do acoplamento (medida de interdependência entre os elementos arquiteturas) e da coesão (medida da robustez funcional relativa a um elemento arquitetural) dos

elementos das arquiteturas em questão, que influenciam nos requisitos não-funcionais a serem avaliados: Manutenção, Reusabilidade e Complexidade. A avaliação indireta de cada um desses requisitos não-funcionais se deve à inexistência de métricas Orientadas a Objetos que realizem tal avaliação.

O método de análise quantitativa tem como propósito discutir os resultados obtidos com a quantificação do acoplamento arquitetural e com a quantificação da coesão arquitetural, com o objetivo de saber se as arquiteturas avaliadas foram bem desenvolvidas ou não.

Problemas Encontrados com o Estado da Arte de Métricas OO

As métricas OO existentes, em sua grande maioria, medem os conceitos relativos ao paradigma OO, como: polimorfismo, herança, coesão, acoplamento, mensagens, métodos e classes. Dessa forma, para a realização da avaliação dos requisitos não-funcionais em um projeto OO, uma ou mais dessas métricas devem ser usadas na avaliação quantitativa, de forma indireta, dos requisitos não-funcionais pretendidos.

A falta de métricas diretas para avaliação de atributos de qualidade foi a causa do primeiro problema encontrado neste estudo para determinar um método de avaliação quantitativa. A seguinte questão veio à tona: Quais seriam as melhores características dos elementos das arquiteturas a serem avaliadas para se poder quantificar indiretamente cada um dos requisitos não-funcionais desejados? Para responder tal pergunta, algumas peculiaridades das arquiteturas a serem avaliadas deveriam ser consideradas como, por exemplo, uma arquitetura ser baseada em classes e as outras serem constituídas por componentes. Após um estudo da literatura disponível sobre arquiteturas baseadas em classes e arquiteturas constituídas por componentes, chegou-se à conclusão de que o acoplamento entre os elementos de cada arquitetura e a coesão de cada um deles seriam as características mais relevantes para se avaliar os requisitos não-funcionais pretendidos. Isso porque são essas duas características que possibilitam a avaliação da independência funcional (ver Seção 2.5.3) de cada elemento (Pressman 2001).

Determinadas as características dos elementos das arquiteturas a serem quantificadas, uma série de problemas surgiu:

1) O estudo das métricas OO está em um estágio inicial e não se encontra tão evoluído como o estudo das métricas para o paradigma procedural. (Newman *et. al.*, 1999)

2) As métricas OO existentes são voltadas para projetos OO sem levar em consideração a possibilidade da existência de componentes de software.

3) Não há consenso de quais são as melhores métricas OO e muitas não têm uma validação muito bem definida (Harrison *et. al.*, 1998b; Abaunader *et. al.*, 1997).

4) O autor desta dissertação desconhece a existência de métricas que avaliam projetos constituídos por componentes e que pudessem ser aplicadas nas arquiteturas a serem analisadas.

Uma das soluções seria utilizar as métricas do paradigma procedural para a avaliação das arquiteturas - já que essas encontram-se em um estágio de aprimoramento bem maior que as métricas voltadas para o paradigma OO - mesmo sabendo que aquelas métricas falham na captura dos conceitos de hierarquia e polimorfismo, que são únicos do paradigma OO (Abaunader *et. al.*, 1997; Li *et. al.*, 1995).

Coppick e Cheatham (Coppick *et. al.*, 1992) já haviam realizado tal experimento obtendo resultados "intuitivamente razoáveis", utilizando a métrica Complexidade Ciclomática, proposta por McCabe (McCabe, 1976). Dessa forma, para a avaliação do atributo de qualidade Complexidade, estudou-se a métrica Complexidade Ciclomática (Pressman, 2001; McCabe *et. al.*, 1994; McCabe, 1976), cuja fundamentação está na teoria dos grafos, que fornece a medida quantitativa da complexidade lógica de um programa.

McCabe e Watson (McCabe *et. al.*, 1994) relataram alguns usos importantes para as métricas de complexidade:

"Métricas de complexidade podem ser usadas para prover informação crítica sobre confiabilidade e manutenibilidade de sistemas de software a partir da análise automática do código fonte (ou informação do projeto procedimental). Métricas de complexidade também fornecem realimentação durante o projeto de software para ajudar a controlar a atividade de projeto. Durante o teste e manutenção, elas fornecem informações detalhadas sobre os módulos de software para ajudar a localizar áreas de instabilidade potencial."

Entretanto, constatou-se que, para a utilização correta da métrica proposta por McCabe, seria necessário um nível de abstração menor que o que se terá com a execução das metodologias para a obtenção das arquiteturas a serem estudadas (Seções 3.1, 3.2 e 3.3).

Esta mesma constatação ocorreu quanto foram estudadas as métricas fundamentadas nas interfaces dos módulos (elementos arquiteturais), como, por exemplo, nas métricas de projeto arquitetural propostas por Card e Glass (Card *et. al.*, 1990), em que para o cálculo da complexidade do sistema é necessário conhecer o número de módulos diretamente invocados pelo módulo e o número de variáveis que são passadas para e do módulo. Para o conhecimento de tais valores na arquitetura baseada em classes, por exemplo, seria necessário uma conscientização do desenvolvedor de não declarar atributos públicos ou protegidos e ter conhecimento do código fonte, já que uma classes filha pode ou não invocar as mesmas classes de sua superclasse.

Já a utilização de métricas do paradigma procedural para a avaliação dos dois outros requisitos de qualidade esbarrou na impossibilidade de realizar a medida direta da manutenibilidade (Pressman, 2001), e nas exigências das métricas existentes para Reusabilidade de que o software já esteja em uso há um bom tempo.

Com a utilização de métricas do paradigma procedural descartada, a solução foi voltar para as características dos elementos das arquiteturas a serem medidas e procurar métricas que pudessem ser utilizadas em um nível arquitetural constituído por classes e/ou componentes.

Tendo o conhecimento da não existência de métricas que avaliem projetos constituídos de componentes, voltou-se o trabalho para a procura de métricas OO que avaliassem um nível arquitetural baseado em classes e que fosse possível adaptá-las para a avaliação de uma arquitetura constituída por componentes.

Após um estudo na literatura especializada, o conjunto de métricas que mais se adequou aos interesses deste estudo foi o proposto por Chidamber e Kemerer (Chidamber, *et. al.*, 1994), denominado métricas CK.

Como foi determinado que as características dos elementos das arquiteturas a serem avaliadas são o Acoplamento e a Coesão, as métricas CK que serviriam para este estudo seriam a CBO (*Coupling between Object Class*) e LCOM (*Lack of Cohesion in Method*), respectivamente. Entretanto, elas são objeto de discussão entre pesquisadores, devido à forma

com que foram validadas e aos resultados obtidos por elas (Harrison *et. al.*, 1998a; Harrison *et. al.*, 1998b; Li *et. al.*, 1995; Hitz, *et. al.*, 1996b).

Como não há um consenso de quais são as melhores métricas OO a serem utilizadas (Abaunader *et. al.*, 1997; Harrison *et. al.*, 1998b), um estudo investigativo foi realizado com o intuito de encontrar uma comparação entre as propostas de métricas existentes e, assim, obter o conhecimento de qual métrica seria mais apropriada.

Trabalhos como o de Etzhorn, Devis e Li (Etzhorn *et. al.*, 1997) e Lakshminarayana e Newman (Newman *et. al.*, 1999), que realizaram uma comparação entre diversas propostas de métricas LCOM, foram estudados, entretanto, obtiveram conclusões diferentes. Diante desse novo inconveniente, optou-se por escolher métricas que satisfizessem o que é pretendido por este estudo: métricas que avaliassem tanto arquiteturas baseadas em classes quanto constituída por componentes, mesmo que tivessem que ser adaptadas, fornecendo, assim, conhecimento útil para a análise quantitativa das arquiteturas.

Quantificação do Acoplamento Arquitetural

Segundo Bansiya (Bansiya, 2000), as classes e as relações entre elas são as duas características mais importantes de um sistema orientado a objetos e, portanto, de *frameworks OO*. Assim, a relação entre os blocos que constituem o *framework* define a sua arquitetura (Bansiya, 2000). Devido a isso, a métrica Fator de Acoplamento, proposta por Harrison, Counsell e Nithi (Harrison *et. al.*, 1998a; Harrison *et. al.*, 1998b) e descrita a seguir, foi escolhida para analisar a manutenibilidade, a reusabilidade e a complexidade de cada uma das arquiteturas, já que o aumento da complexidade, a dificuldade de inteligibilidade, o esforço necessário durante a manutenção e o fraco potencial de reuso estão diretamente relacionados com o aumento do fator de acoplamento (Harrison *et. al.*, 1998b; Chidamber, *et. al.*, 1994).

- **Métrica Fator de Acoplamento:** é definida pela seguinte equação:

$$CF = \sum_i \sum_j é\_cliente\ (C_i, C_j)) / (TC^2 - TC) \qquad \text{(Equação 1)}$$

Onde os somatórios ocorrem de i = 1 até TC e j = 1 até TC e:

- C é a classe
- TC é o total de classes
- é_cliente =1 se, e somente se, existir uma relação entre a classe cliente C_c, e a classe servidora C_s e $C_c \neq C_s$, senão é_cliente = 0.

A métrica Fator de Acoplamento avalia quantitativamente somente o acoplamento entre classes de uma arquitetura. Entretanto, algumas arquiteturas desse estudo possuem componentes em sua estrutura, o que impossibilita a utilização dessa métrica. Isto porque, nessas arquiteturas, tem-se o acoplamento entre as classes de um mesmo componente, além de possíveis acoplamentos entre os componentes, propiciado por suas interfaces. Diante disso, o autor propõe uma adaptação da equação dessa métrica para se avaliar tanto uma arquitetura estruturada apenas com classes quanto uma arquitetura que possui componentes em sua estrutura.

- **Equação Fator de Acoplamento com Componentes (FAC):**

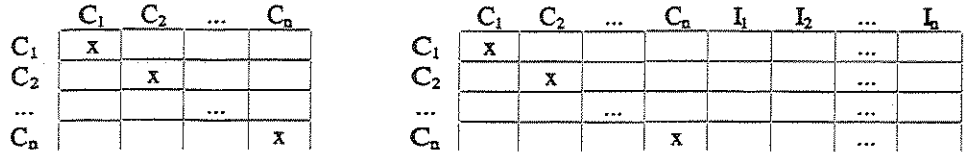
$$FAC = (\sum i \sum j\ é\_cliente\ (Ci, CIj)) / ((TC^2 - TC) + (TC \times TI)) \quad \text{(Equação 2)}$$

Onde os somatórios ocorrem de i = 1 até TC e j = 1 até TC + TI e:

- C é a classe
- CI é classe ou interface
- TC é o total de classes
- TI é o total de interfaces
- é_cliente =1 se, e somente se, existir a relação entre a classe cliente Cc, e a classe ou interface servidora, CIs e Cc ≠ CIs, senão é_cliente = 0.

Tanto na Equação 1 quanto na Equação 2, o fator de acoplamento é, no máximo um, e, no mínimo, zero. Para este estudo, considera-se que, quanto maior for esse número, maior é a complexidade da arquitetura. A manutenção, a compreensão e o potencial para reuso também podem variar proporcionalmente com o Fator de Acoplamento (Pressman, 2001; Chidamber, *et. al.*, 1994), apesar da métrica Fator de Acoplamento não ter sido validada para ser uma medida indireta de tais requisitos (Harrison *et. al.*, 1998a).

Na Equação 1, o fator de acoplamento é o somatório de uma matriz quadrada binária dividido pelo tamanho da matriz menos o número de elementos da diagonal principal. Essa subtração é necessária devido à impossibilidade de uma classe estar acoplada a si mesma. Tomando isso como base, o que se propõe (Equação 2) para o cálculo do fator de acoplamento de uma arquitetura que possui componentes em sua estrutura é realizar as operações da Equação 2 sobre uma matriz gerada a partir da matriz da Equação 1, adicionada de uma coluna para cada interface. Não se adicionou uma linha para cada interface devido à impossibilidade de uma interface ser acoplada a outra interface, mas sim a uma classe. A Figura 3.11 mostra como são essas matrizes, onde: C é a classe, I é a interface e X é onde não há acoplamento de forma alguma.



Matriz da Equação 1 Matriz da Equação 2

Figura 3.11 – Representação Matricial das Equações 1 e 2

Quantificação da Coesão Arquitetural

Segundo Chidamber e Kemerer (Chidamber, *et. al.*, 1994), o aumento da falta de coesão entre os métodos de uma classe indica a pouca relação entre as funcionalidades fornecidas por essa e que a mesma pode ser dividida em duas ou mais classes. Relatam também que a coesão de cada um dos elementos de uma arquitetura indica como melhor pode ser projetado o sistema e qual é a complexidade de cada classe, já que a falta de coesão de cada um dos elementos da arquitetura aumenta a propagação de erro durante o processo de desenvolvimento.

Segundo Pressman (Pressman, 2001), é desnecessário determinar o nível preciso de coesão. Em vez disso, é importante lutar por alta coesão e reconhecer baixa coesão de modo que o projeto de software possa ser modificado para conseguir maior independência funcional. Levando isso em consideração e devido ao autor desta dissertação não ter encontrado uma métrica de coesão que satisfizesse os anseios do estudo a ser realizado, a quantificação da coesão arquitetural será fundamentada na divisão de módulos coesos proposta por Pressman (Pressman, 2001):

1) Módulos fracamente coesos são os módulos que realizam um conjunto de tarefas que se relacionam fracamente umas com as outras, se é que se relacionam de fato (coesão coincidental).
2) Módulos com níveis moderados de coesão são todos os módulos que possuem elementos de processamento relacionados e que devem ser executados em uma ordem específica (coesão procedimental) ou que os elementos de processamento se concentrem em uma área de estrutura de dados (coesão comunicacional).
3) Módulo de alta coesão são os módulos que realizam uma tarefa procedimental distinta.

Para este estudo, cada módulo descrito por Pressman é um elemento de uma arquitetura e as tarefas mencionadas são as funcionalidades fornecidas por um elemento em particular.

Para a realização da quantificação da coesão arquitetural, deve-se categorizar cada um dos elementos de uma determinada arquitetura quanto à divisão de módulos descrita por Pressman (Pressman. 2001). Para isso, pode-se utilizar um quadro onde, na primeira coluna, encontra-se o tipo de módulo em que um determinado elemento se encaixa (1- Fracamente Coeso, 2- Nível moderado de Coesão e 3 - Altamente Coeso) e, na segunda coluna, o nome do elemento. Após o preenchimento desse quadro, uma discussão sobre os resultados obtidos deve ser realizada considerando-se que a coesão do nível inferior é muito "pior" que a do nível intermediário, e que é quase tão "boa" quanto a coesão do nível superior, já que a escala de coesão não é linear (Pressman. 2001).

3.5 – Considerações Finais

Este capítulo apresentou um método que possibilita a realização de um estudo sistemático em um *framework* já existente e dois métodos que possibilitam a obtenção de duas arquiteturas distintas de um mesmo domínio, do qual faz parte o *framework* a ser estudado.

Em particular, apresentou-se como uma linguagem de padrões pode ser de grande valia para a estruturação correta de um *framework*. Os métodos propostos para o desenvolvimento de uma arquitetura estruturada em componentes e baseada em componentes apresentam uma proposta de como a linguagem de padrões pode ser usada para desenvolver uma arquitetura de *framework* constituída por componentes. Também se exemplificou como a linguagem de padrões é útil no desenvolvimento de uma arquitetura.

O ATAM foi descrito como o método a ser aplicado para realizar a análise qualitativa das arquiteturas dos *frameworks* do ponto de vista dos atributos de qualidade manutenibilidade e reusabilidade. Foi apresentado também o estágio inicial em que se encontram os estudos sobre métricas orientadas a objetos, o que dificultou muito a obtenção da proposta de um método de análise quantitativa que pudesse ser utilizado para avaliar os requisitos não-funcionais manutenibilidade, reusabilidade e complexidade de uma arquitetura de *framework*.

O capítulo seguinte apresenta o experimento realizado por este estudo, mostrando, assim, onde e como cada um destes métodos foi aplicado.

Capítulo 4

O Experimento Prático

Para investigar a aplicabilidade da utilização de componentes na arquitetura de um *framework* e o impacto dessa nova tecnologia no estado da arte dos *frameworks*, foi realizado um experimento que avalia os elementos de três arquiteturas, necessários para a instanciação de um sistema específico. A primeira dessas arquiteturas representa a forma como a maioria dos *frameworks* são desenvolvidos hoje, ou seja, baseados em classes. As outras duas arquiteturas são as que utilizam componentes em suas estruturas, que, neste estudo, são de dois tipos: (i) arquitetura estruturada em componentes, desenvolvida a partir da primeira arquitetura e (ii) arquitetura baseada em componentes, que é desenvolvida desde o início pensando-se em componentes. A existência de dois tipos de arquiteturas constituídas por componentes se dá pelo interesse, neste estudo, de se investigar, também, as diferenças e semelhanças entre essas duas abordagens. O presente estudo foi realizado seguindo o que é proposto pelos métodos descritos no Capítulo 2.

A Figura 4.1 ilustra a relação das arquiteturas utilizadas no experimento. Os retângulos com linhas pontilhadas representam o tipo de elemento arquitetural existente nas arquiteturas e os retângulos com linhas cheias representam as arquiteturas envolvidas no experimento e/ou a origem de cada uma das arquiteturas. Já as linhas unidirecionais associam o influenciador da arquitetura com a arquitetura gerada a partir dele.

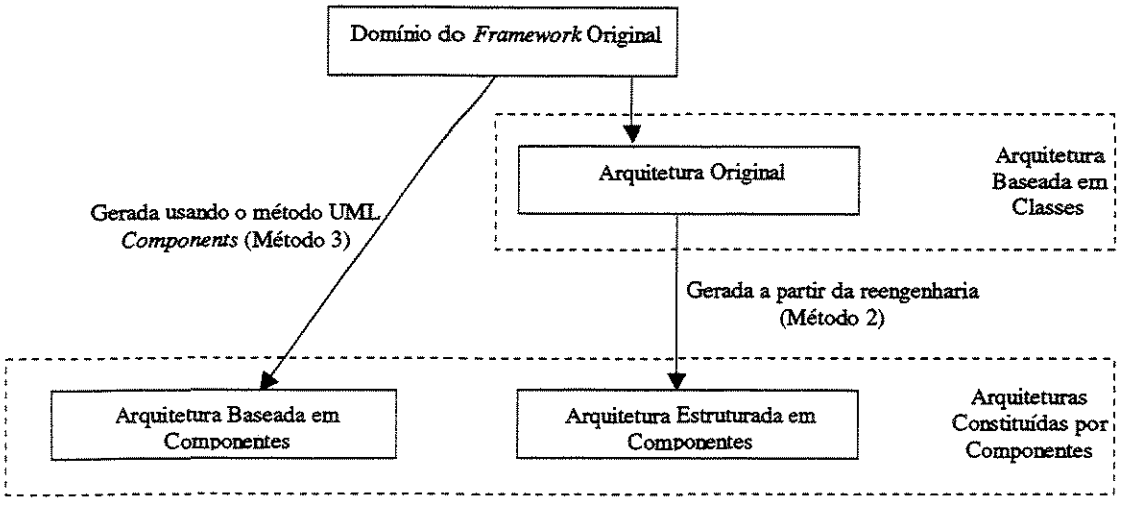


Figura 4.1 – Relação Entre as Arquiteturas Utilizadas no Experimento

As Seções 4.1 e 4.2 mostram, respectivamente, a descrição do experimento e do sistema para o qual as arquiteturas serão instanciadas. O estudo realizado no *framework* existente e sua arquitetura são descritos na Seção 4.3. Já a forma como foram obtidas a arquiteturas Estruturada em Componentes e a Baseada em Componentes, juntamente com a representação gráfica de cada uma delas são mostradas, respectivamente, nas Seções 4.4 e 4.5.

4.1 – Descrição do Experimento

O experimento será realizado usando um *framework* caixa branca, com arquitetura baseada em classes, visando a obter duas novas arquiteturas constituídas por componentes, uma influenciada pela Arquitetura Original do *framework* e outra a partir do domínio à que o *Framework* Original se aplica (Figura 4.1). Após a obtenção das duas novas arquiteturas, será realizada uma comparação qualitativa e quantitativa dos elementos das arquiteturas baseada em classe e constituída por componentes, que são necessários para a instanciação de um sistema de Vídeo Locadora (Seção 4.2). O objetivo do experimento é evidenciar os efetivos ganhos e perdas em desenvolver um *framework* constituído por componentes. Desta forma os passos do experimento são:

Passo 1: Escolha de um *framework* caixa branca, cuja arquitetura seja baseada em classes, e realização de uma análise em sua documentação para a compreensão das características do seu domínio, com o objetivo de obter os requisitos, conceitos e relações entre esses requisitos no domínio em questão, além da instanciação da arquitetura deste *framework* para o sistema de Vídeo Locadora. Este *framework* é denominado Framework Original. Para a realização desse passo é utilizado o método 1 descrito na Seção 3.1.

Passo 2: Desenvolvimento de uma segunda arquitetura, obtida a partir de uma reengenharia da arquitetura de *Framework* Original, que é influenciada pelas propostas de Kulesza (Kulesza, 2000), Fayad (Fayad *et. al.*, 1999)

e Gurp (Gurp *et. al.*, 2001), Seção 3.2. Essa nova arquitetura é denominada Arquitetura Estruturada em Componentes. Sua instanciação será exemplificada a partir da descrição do sistema de Vídeo Locadora (Seção 4.2).

Passo 3: Desenvolvimento de uma terceira arquitetura, denominada Arquitetura Baseada em Componentes, utilizando parte do processo de especificação de sistemas de software baseados em componentes, proposto por Cheesman e Daniels (Cheesman *et.al.*, 2001) (ver Seção 3.3). O sistema de Vídeo Locadora (Seção 4.2) será utilizado para instanciar essa nova arquitetura.

Passo 4: Comparação qualitativa, seguindo os passos propostos pelo ATAM, (Seção 3.4.1), das instanciações obtidas das arquiteturas dos passos 1, 2 e 3.

Passo 5: Comparação quantitativa entre as instanciações obtidas das arquiteturas dos passos 1, 2 e 3, utilizando o método de análise quantitativa proposto na Seção 3.4.2.

4.2 – Descrição do Sistema de Vídeo Locadora

O sistema simplificado de locadora de vídeos, descrito a seguir, será utilizado para a instanciação das arquiteturas dos *frameworks*. Este sistema foi proposto por Braga (Braga *et. al.*, 2001), descrito no manual de instanciação do *framework* GREN (Braga *et. al.*, 2001).

4.2.1 – Visão Geral do Sistema

O sistema para a locadora de vídeos tem por objetivo principal o gerenciamento das locações efetuadas por seus clientes. Algumas de suas principais funções são: acompanhar as locações efetuadas por determinado cliente, que pode consultar se já alugou certa fita de vídeo anteriormente; verificar se uma fita de vídeo devolvida está em atraso e, em caso positivo, calcular a multa devida; emitir alguns relatórios, como, por exemplo, clientes com fitas de vídeo não devolvidas, filmes mais locados no período, gênero de filme preferido dos clientes e faturamento mensal.

Requisitos Funcionais

1) Cadastro e manutenção de fitas, clientes, locações e devoluções

1.1) O sistema deve permitir a inclusão, alteração e remoção de fitas de vídeo em estoque na locadora, com os seguintes atributos: código de barras da fita, nome do filme, atores principais, ano do filme, gênero do filme, preço de locação, diretor, localização na estante e se é dublada ou legendada.

1.2) O sistema deve permitir a inclusão, alteração e remoção de clientes da locadora, com os seguintes atributos: nome, endereço, cidade, estado, telefone, e-mail, documento de identificação e data de nascimento.

1.3) O sistema deve permitir a locação de uma ou mais fitas de vídeo pelo cliente. Cada locação possui os seguintes atributos: data de locação da fita de vídeo, data programada para devolução, nome do cliente e dados de identificação das fitas de vídeo locadas. Para cada fita de vídeo locada deve ser fornecido seu código de barras e confirmado o valor da locação. Durante o processo de locação, o sistema deve modificar, automaticamente, o *status* atual das fitas de vídeo locadas, modificando-o de "disponível" para "locada".

1.4) O sistema deve permitir a devolução de fitas de vídeo locadas. Após a identificação do cliente, o sistema deve efetuar uma busca nas suas locações pendentes e permitir que o operador identifique a locação que está sendo devolvida. Cada devolução possui os seguintes atributos: data de devolução da fita de vídeo, valor da multa (opcional), valor total a ser pago e desconto concedido (opcional). Durante o processo de devolução, o sistema deve modificar, automaticamente, o *status* atual das fitas de vídeo devolvidas, modificando-o de "locada" para "disponível".

1.5) O sistema deve permitir as seguintes opções de pagamento da locação: (i) ao locar as fitas de vídeo; (ii) somente na devolução. O pagamento é sempre efetuado uma única vez.

1.6) Se o cliente optar pelo pagamento ao locar as fitas de vídeo, mas entregá-las com atraso, durante a devolução, será feito o recebimento da multa.

2 – Para a impressão de relatórios e consultas o sistema deve:

2.1) Emitir um comprovante de locação das fitas de vídeo, em duas vias, onde constem os dados do cliente, a data e a hora da locação, a data para devolução, as fitas de vídeo locadas e o total a pagar. Se o cliente optou por pagar ao locar as fitas de vídeo, então deverá ser impressa a palavra "PAGO", juntamente com uma linha reservada para o visto do funcionário da locadora.

2.2) Permitir a impressão de um relatório contendo todas as locações pendentes, com nome e telefone dos clientes, data de locação, data prevista para devolução, total a pagar e discriminação das fitas de vídeo locadas.

2.3) Permitir a impressão de um relatório contendo o faturamento mensal, isto é, as locações recebidas durante o mês, com nome do cliente, data de locação, data de devolução e valor pago.

2.4) Permitir a impressão de um relatório contendo os filmes mais locados no período, contendo o nome do filme, ano, gênero e número de locações no período fornecido.

2.5) Permitir a consulta (na tela) dos filmes já locados por um determinado cliente em ordem descendente de data, até o máximo de 100 filmes, contendo o nome do filme, ano, gênero e atores principais.

2.6) Permitir a impressão de um relatório contendo a classificação dos gêneros de filmes preferidos pelos clientes em um determinado período, com descrição do gênero e percentagem de locação daquele gênero no período fornecido.

2.7) Permitir a consulta (na tela) das fitas de vídeo disponíveis para um determinado filme fornecido.

Requisitos Não-Funcionais

Os requisitos aqui solicitados são do ponto de vista do usuário desenvolvedor:

1) Manutenibilidade

O sistema deve ser fácil de ser corrigido caso alguma funcionalidade solicitada não esteja sendo satisfeita de forma correta. E fácil de ser adaptado a novas funcionalidades antes não previstas.

2) Reusabilidade

Determinada pela facilidade de se conseguir elementos arquiteturais reutilizáveis quando for necessário desenvolver outro sistema que tenha alguma funcionalidade semelhante a este.

3) Complexidade

O sistema deve ser fácil de ser instanciado e ter o seu entendimento interno também facilitado.

4.3 – Entendimento e Levantamento da Arquitetura do *Framework* Original

O *Framework* Original escolhido para ser estudado é o *framework* GREN (Gestão de REcursos de Negócio) que foi desenvolvido com o objetivo de estudar o impacto das linguagens de padrões sobre o desenvolvimento e uso de *frameworks* e, assim, tentar responder algumas questões: Como documentar *frameworks* de forma que os usuários possam entendê-los e os desenvolvedores possam adaptá-los? Como apoiar a evolução de um *framework* em um dado domínio e em direção a novos domínios? Quais são as questões a considerar na construção de *frameworks* para domínios específicos? As sub-seções a seguir descrevem esse *framework* em detalhes.

4.3.1 Escolha do *Framework* GREN e suas Características

Para constatar alguns dos problemas encontrados na maioria dos *frameworks* existentes hoje e mostrar quais são os benefícios e problemas de *frameworks* estruturados/baseados em componentes, foi feita uma pesquisa na literatura especializada para a obtenção de um *framework* que exemplificasse como se encontra a maioria dos *frameworks* disponíveis (Seção 2.1).

Os principais fatores que influenciaram essa escolha foram: (i) o acesso irrestrito à documentação, para que se possa ter um perfeito entendimento e (ii) a possibilidade do diálogo com o(s) autor(es) do *framework* para esclarecimentos de quaisquer dúvidas e verificação do trabalho a ser realizado.

Tomando como base os critérios descritos acima, o *framework* escolhido foi do domínio de Gestão de REcursos de Negócio, denominado GREN (Braga *et, al.,* 2001), que possui aproximadamente 150 (cento e cinqüenta) classes, as quais estão divididas em camadas, descritas mais adiante. O *framework* GREN foi desenvolvido com base em uma linguagem de padrões para gestão de recursos de negócio (Braga *et. al.,* 1999), Seção 4.3.2, visando a apoiar a construção de aplicações para tal domínio, que inclui sistemas para locação, comercialização e manutenção de bens, tais como, automóveis, imóveis, fitas de vídeo, dentre outros.

Sua documentação inclui um diagrama de classes em UML e um Manual de Instanciação que mostra como é feita a instanciação do *framework* GREN para uma aplicação particular, com um exemplo de aplicação. Esse manual apresenta ainda outras informações úteis para o desenvolvedor, tais como o mapeamento dos elementos definidos na linguagem de padrões com as classes implementadas no *framework* e com os métodos correspondentes que devem ser sobrepostos.

Camadas da Arquitetura do Framework GREN

A arquitetura de mais alto nível do *framework* GREN (Figura 4.2) possui 8 (oito) camadas. As duas camadas mais inferiores, MySQL e Smalltalk, são, respectivamente, a base de dados utilizada na persistência dos objetos (MYSQL, 2001) e a linguagem de programação em que o *framework* GREN foi desenvolvido (Cincom, 2001). As camadas localizadas logo acima dessas duas, GREN - Persistence Layer e GREN - Report Writer, são responsáveis, nesta ordem, por isolar a persistência dos dados das demais camadas, que são a camada de aplicação (GREN - Application Layer) e a camada de interface com o usuário (GREN - Graphical User Interface Layer); e por elaborar e imprimir relatórios, já que contêm as classes específicas para essas tarefas. Logo em seguida a essas duas últimas camadas, encontra-se a camada de aplicação, que se comunica com a camada de persistência sempre que precisa guardar um objeto em memória. Dentro da camada de aplicação, existem diversas classes derivadas diretamente dos padrões de análise que compõem a linguagem de padrões Gestão de Recursos de Negócio (Seção 4.3.2). As demais camadas são camada de interface com o usuário, localizada acima da camada de aplicação, que contém as classes responsáveis pela entrada e saída de dados, como formulários de interface, janelas e menus; a camada GREN-

Wizard, que contém as classes que implementam o instanciador do *framework* GREN; e a camada onde se encontram as extensões do *framework*, que podem ser realizadas a partir da camada GREN-Wizard ou da camada de interface com o usuário (Braga *et. al.*, 2001).

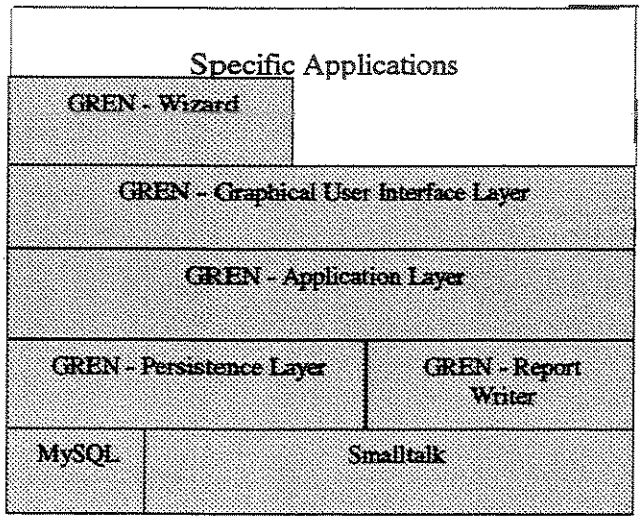


Figura 4.2 – Camadas da Arquitetura do GREN (Braga *et. al.*, 2001)

Essa estrutura arquitetural é toda constituída, unicamente, de classes, que são o principal bloco de construção de software orientado a objetos (Booch *et. al.*, 2000). Para o experimento desta dissertação, serão consideradas as classes encontradas nas camadas GREN - Persistence Layer e de aplicação, que são, em sua totalidade aproximadamente 75 (setenta e cinco).

4.3.2 – A linguagem de Padrões GRN

A linguagem de padrões para Gestão de Recursos de Negócios (GRN) é formada por quinze padrões de análise e tem como objetivo auxiliar os engenheiros de software a desenvolver aplicações que lidem com gestão de recursos de negócios, ou seja, nas quais seja necessário registrar transações de aluguel, comércio ou manutenção de recursos (Braga *et. al.*, 1999).

O aluguel de recursos enfoca principalmente a utilização temporária de um bem ou serviço, como, por exemplo, uma fita de vídeo ou o tempo de um especialista. O comércio de recursos enfoca a transferência de propriedade de um bem, como, por exemplo, uma venda ou leilão de produtos. A manutenção de recursos enfoca a manutenção de um determinado

produto, utilizando mão-de-obra e peças para execução, como é o exemplo de uma oficina de reparo de eletrodomésticos (Braga *et. al.*, 2001, Braga *et. al.*, 1999).

Braga (Braga *et. al.*, 2001) resume da seguinte forma a linguagem de padrões GRN:

A Figura 4.3 "contém um diagrama que mostra as dependências entre os padrões da linguagem de padrões GRN e a ordem na qual eles são geralmente aplicados. Essas dependências são também apresentadas, e eventualmente complementadas, em um elemento específico de cada padrão, o elemento "Próximos Padrões". Os principais padrões da linguagem são indicados por uma linha mais espessa. São eles: LOCAR O RECURSO, COMERCIALIZAR O RECURSO e MANTER O RECURSO.

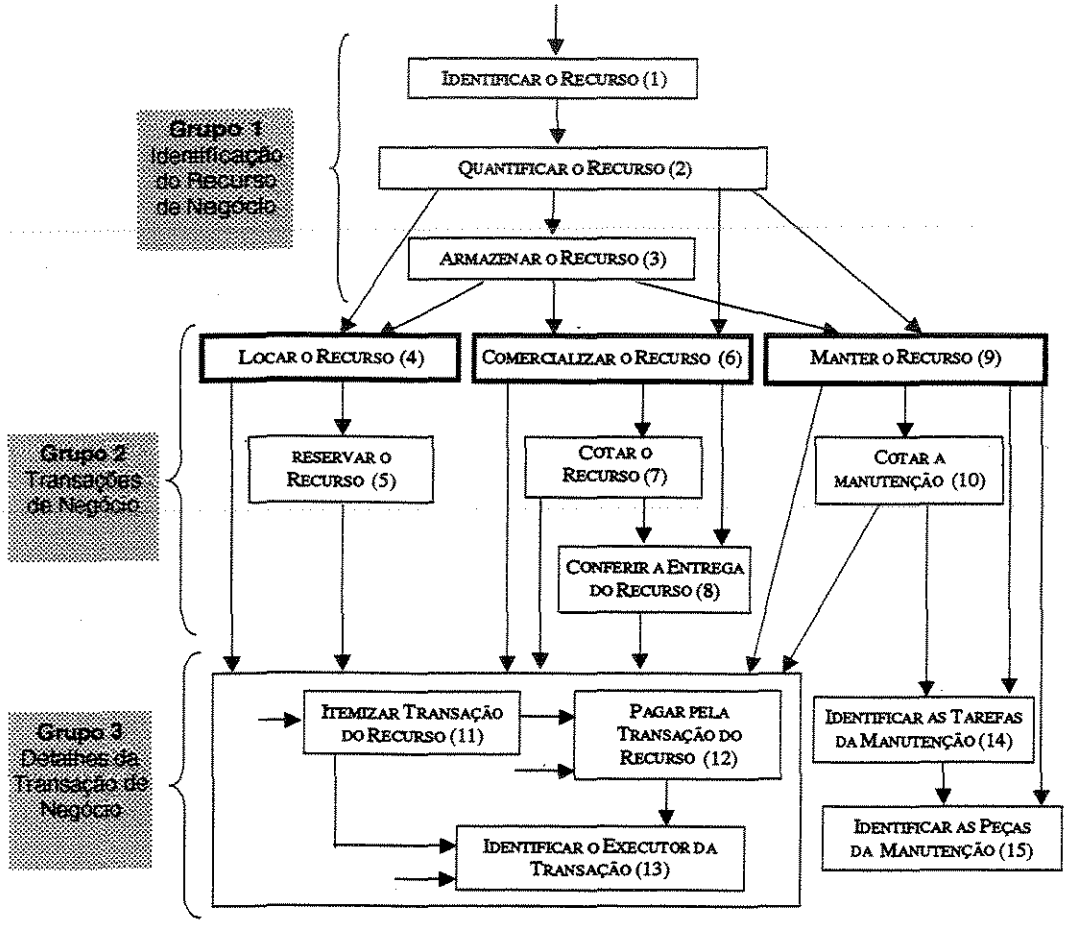


Figura 4.3 – Linguagem de padrões para gestão de recursos de negócios (Braga *et. al.*, 2001)

Os padrões estão agrupados de acordo com seu propósito, como pode ser visto na figura acima. O grupo 1 possui três padrões: IDENTIFICAR O RECURSO, QUANTIFICAR O RECURSO e ARMAZENAR O RECURSO, que dizem respeito à identificação e possível

qualificação, quantificação e armazenagem dos recursos gerenciados pelo negócio. O grupo 2 contém os padrões relacionados à manipulação dos recursos de negócio pelo sistema. Existem sete padrões nesse grupo: LOCAR O RECURSO, RESERVAR O RECURSO, COMERCIALIZAR O RECURSO, COTAR O RECURSO, CONFERIR A ENTREGA DO RECURSO, MANTER O RECURSO e COTAR A MANUTENÇÃO. O grupo 3 contém cinco padrões que cuidam de detalhes das transações efetuadas com o recurso. Os três primeiros, ITEMIZAR TRANSAÇÃO DO RECURSO, PAGAR PELA TRANSAÇÃO DO RECURSO e IDENTIFICAR O EXECUTOR DA TRANSAÇÃO, são aplicáveis a quaisquer das transações do grupo 2. Os dois últimos, IDENTIFICAR AS TAREFAS DA MANUTENÇÃO E IDENTIFICAR AS PEÇAS DA MANUTENÇÃO, são aplicáveis às transações contidas nos padrões MANTER O RECURSO e COTAR A MANUTENÇÃO."

A linguagem de padrões GRN pode ser utilizada tanto para auxiliar na análise de sistemas desse domínio quanto na instanciação do *framework* GREN para atender aos requisitos desses sistemas específicos, pois fornece como resultado um diagrama de classes para uma aplicação específica, composto de classes, relacionamentos, atributos e métodos (Braga *et. al.*, 2001, Braga *et. al.*, 1999).

Resultado da Aplicação da Linguagem de Padrões para o Sistema de Vídeo Locadora.

Os requisitos funcionais solicitados pela análise do sistema de Vídeo Locadora (Seção 4.2) são fornecidos pelas tarefas existentes nos padrões 1, 2 , 4, 11 e 12 da linguagem de padrões (Figura 4.3). A Figura 4.4 representa a linguagem de padrões instanciada para tal sistema. A aplicação completa da linguagem de padrões para o sistema de Vídeo Locadora pode ser encontrado no manual de instanciação do *framework* GREN (Braga *et. al.*, 2001).

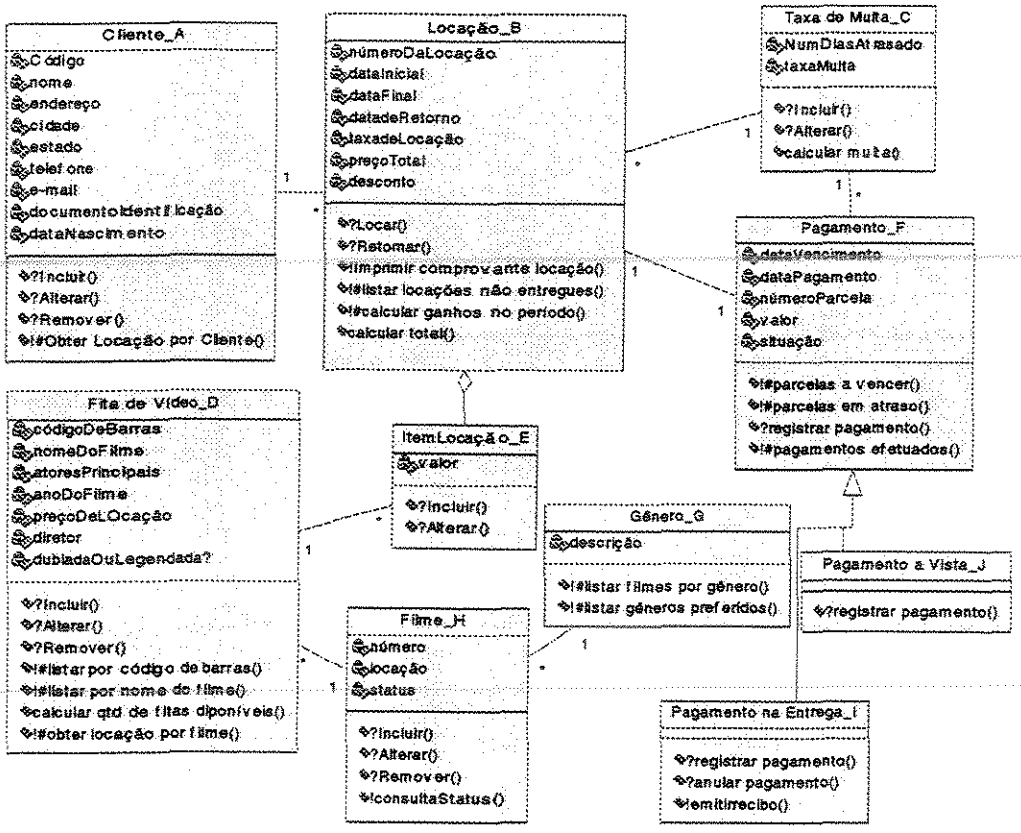


Figura 4.4 – Diagrama de classes após a utilização da linguagem de padrões para o sistema de Vídeo Locadora

A sintaxe do nome da classe é: Nome da Classe_Referência do nome da classe. Essa referência será utilizada para se fazer uma analogia da classe encontrada ao se aplicar a linguagem de padrões para o sistema de Vídeo Locadora e para saber como ela é implementada nas arquiteturas estudadas nesta dissertação.

4.3.3 – Levantamento da Arquitetura do *Framework* Original

Apesar de toda a documentação ter sido fornecida e de haver facilidade de diálogo com os autores do GREN, a obtenção da arquitetura não foi trivial, devido à documentação não estar totalmente atualizada. Com isso, houve a necessidade de se fazer um estudo minucioso do diagrama de classes fornecido, com o que foi implementado em *Smalltalk*, para que fosse possível obter a arquitetura correta do *framework*.

A obtenção da arquitetura do *framework* GREN baseou-se, então, em (i) um documento gerado a partir da implementação em *Smalltalk*, que forneceu: o nome da classe, os atributos, os métodos e a sua superclasse, juntamente com (ii) o diagrama de classes, que estava desatualizado e (iii) a linguagem de padrões GRN (Seção 4.3.2). Com essa documentação em mãos, o método de análise do *framework* a ser estudado (Seção 3.1) foi realizado várias vezes até a obtenção da arquitetura de mais baixo nível do *framework* original, que é mostrada parcialmente na Figura 4.5.

O resultado disso foi um único diagrama de classes contendo aproximadamente 72 (setenta e duas) classes de aplicação e 3 (três) de persistência. A Figura 4.5 representa parcialmente a arquitetura original do *framework* GREN.

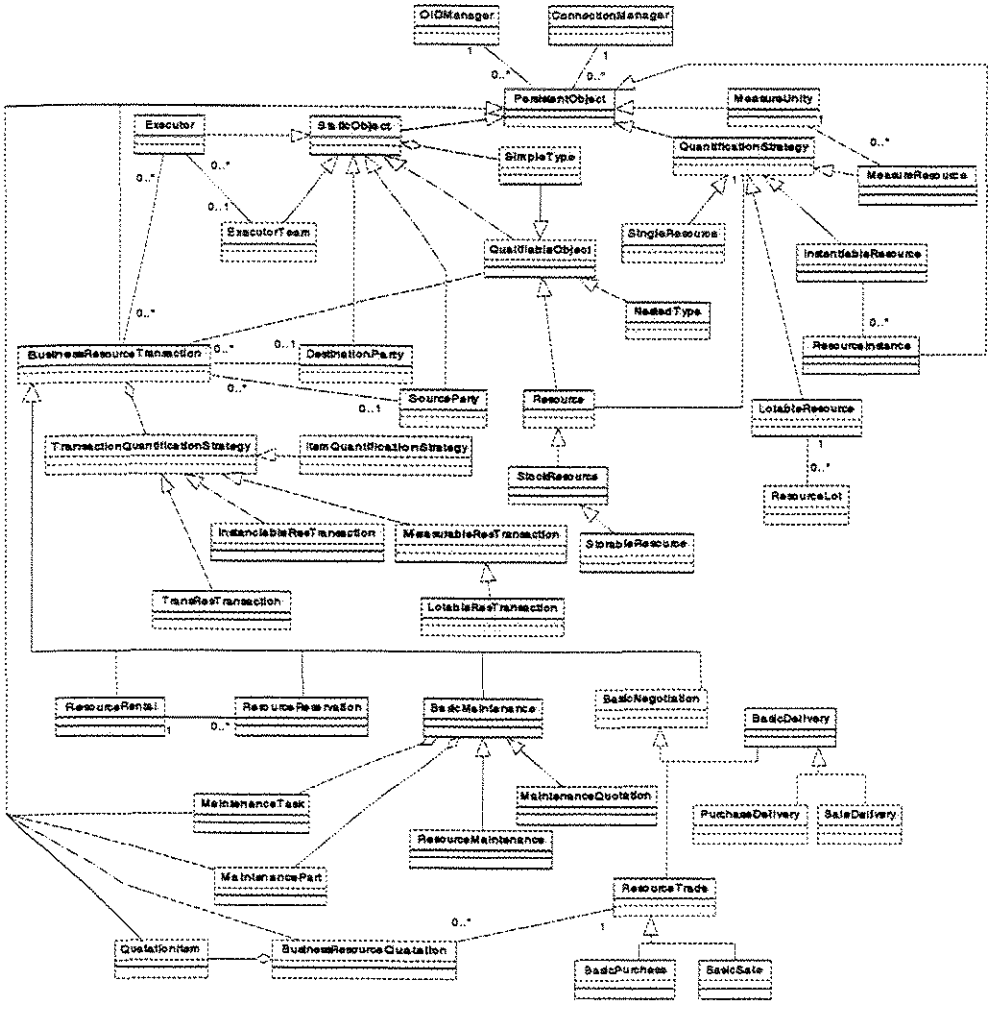


Figura 4.5 – Arquitetura Parcial do *Framework* Original

Elementos da Arquitetura Original Utilizados na Instanciação do Sistema de Vídeo Locadora

Os elementos a serem utilizados para a instanciação do sistema de Vídeo Locadora são os que realizam tarefas existentes nos padrões 1, 2 , 4, 11 e 12 da linguagem de padrões (Figura 4.3) como identificado quando se aplicou a linguagem de padrões para o sistema de Vídeo Locadora.

A Figura 4.6 representa as classes da Arquitetura Original que são utilizadas para a instanciação do sistema em questão.

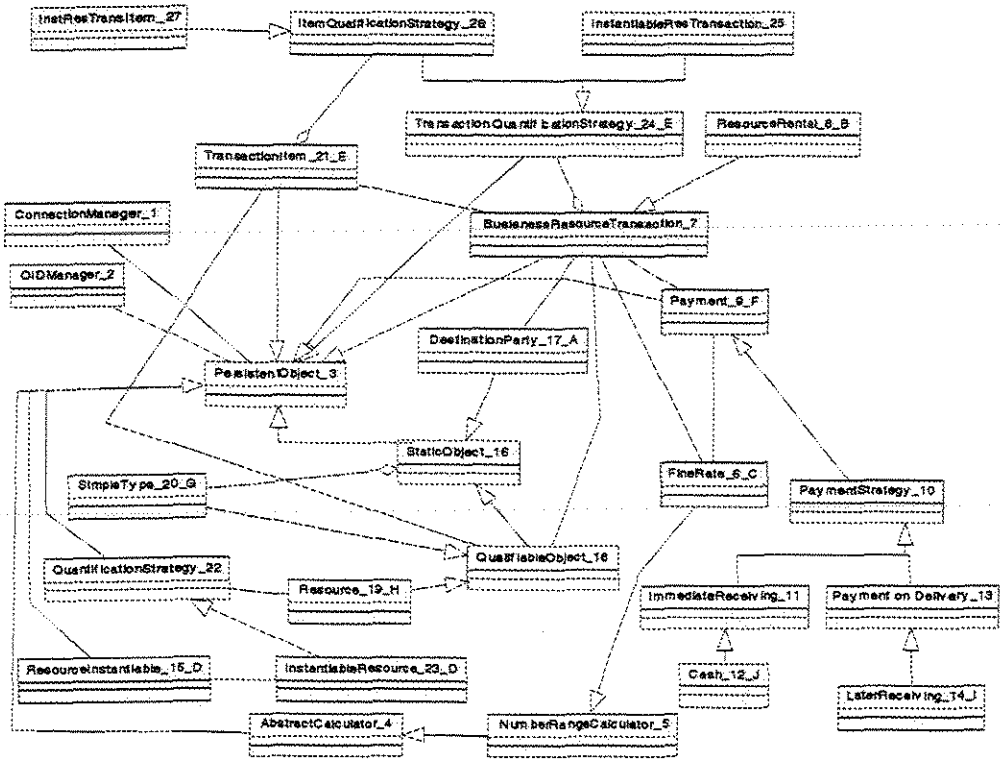


Figura 4.6 - Classes da Arquitetura Original utilizadas no desenvolvimento do Sistema de Vídeo Locadora

A sintaxe de nome da classe é: Nome da Classe do *Framework* GREN_Número da Classe_Referência do nome da classe pertencente à Figura 4.4.

4.3.4 – Considerações Finais

A instanciação adequada da arquitetura original está diretamente ligada à linguagem de padrões GREN. Um bom entendimento dessa linguagem é fundamental para o sucesso do desenvolvimento da aplicação, já que a maior parte do manual faz referência a ela.

O total de páginas a ser estudado, isto é, toda a documentação descrita anteriormente, é em torno de 140 (cento e quarenta). No melhor dos casos, que envolve apenas o diagrama de classes da arquitetura e parte do manual, onde se encontram os métodos que devem ser sobrepostos e quais classes devem ser criadas, o número de páginas é de aproximadamente 60 (sessenta).

A principal dificuldade apresentada na instanciação dessa arquitetura é a necessidade de compreender um grande número de elementos arquiteturais (classes) e seus relacionamentos. Como as classes pertencem a um nível de abstração logo acima de objetos, mesmo que a maioria delas seja altamente coesa, a necessidade de se relacionarem é grande, já que encapsulam características comuns de um conjunto particular de objetos.

A vantagem que mais se destaca é a possibilidade de modelar exatamente o que é pedido pela análise do sistema, já que essa arquitetura é caixa branca e constituída puramente por classes, o que não gera um encapsulamento muito grande entre as distintas tarefas a serem realizadas, possibilitando maior flexibilidade na instanciação do sistema.

4.4 – Obtenção da Arquitetura Estruturada em Componentes

Nessa seção será exemplificado como o método 2 (Seção 3.2) é utilizado para gerar a Arquitetura Estruturada em Componentes. Essa arquitetura é obtida através de uma reengenharia da Arquitetura Original do *Framework* GREN.

Ao se realizar a primeira atividade proposta pelo método, que é a de especificar os componentes da nova arquitetura e as interfaces desses, obteveram-se 14 (quatorze) componentes, sendo que 13 (treze) deles eram a correspondência de um para um em relação aos padrões e 1 (um) continha as funcionalidades dos padrões Pagar pela Transação do Recurso e Identificar o Executor da Transação e algumas outras que davam suporte às funcionalidades contidas nos outros componentes identificados.

Com o término da primeira atividade, as atividades seguintes foram realizadas algumas vezes e, na última execução dessas, foram identificados e tiveram suas respectivas interfaces determinadas os componentes Base, Quantificação de Recurso, Persistência, Endereço, Classificação, Unidade de Medida, Controle de Estoque, Política de Execução, Armazenamento, Recursos Envolvidos na Transação, Pagamento, Compra/Venda, Locar/Reservar Recurso, Manutenção e Executor da Transação. A relação entre os componentes e os padrões da linguagem GRN (Seção 4.3.2) mostrada na Figura 4.7, também foi identificada, além da arquitetura em camadas (Seção 4.4.1), e da arquitetura de mais baixo nível (Figura 4.8).

Nº do Padrão de Projeto	**Componentes da Arquitetura Estruturada em Componentes**
12	Pagamento
13	Executor da Transação
4 e 5	Locar/Reservar Recurso
6, 7 e 8	Compra e Venda
9,10,14 e 15	Manutenção
-	Política de Execução
-	Controle de Estoque
11	Recursos Envolvidos na Transação
3	Armazenamento
3,4,5,6,7,9 e 10	Endereço
1	Classificação
2	Quantificação do Recurso
2	Unidade de Medida
1,3, 4,5,6,7,9,10	Componente Base
-	Persistência
	Total de Componentes = 15

Figura 4.7 – Relação de cada componente da Arquitetura Estruturada em Componentes com os padrões que eles representam total ou parcialmente

As linhas pontilhadas unidirecionais ou bidirecionais existentes entre dois componentes da Figura 4.8 ou qualquer outra figura que mostra a relação entre os componentes de uma arquitetura, representam a dependência que um componente tem com o outro. O componente que tem a linha saindo dele, requer (depende) uma ou várias interfaces que é(são) provida(s) pelo componente que tem a linha chegando nele.

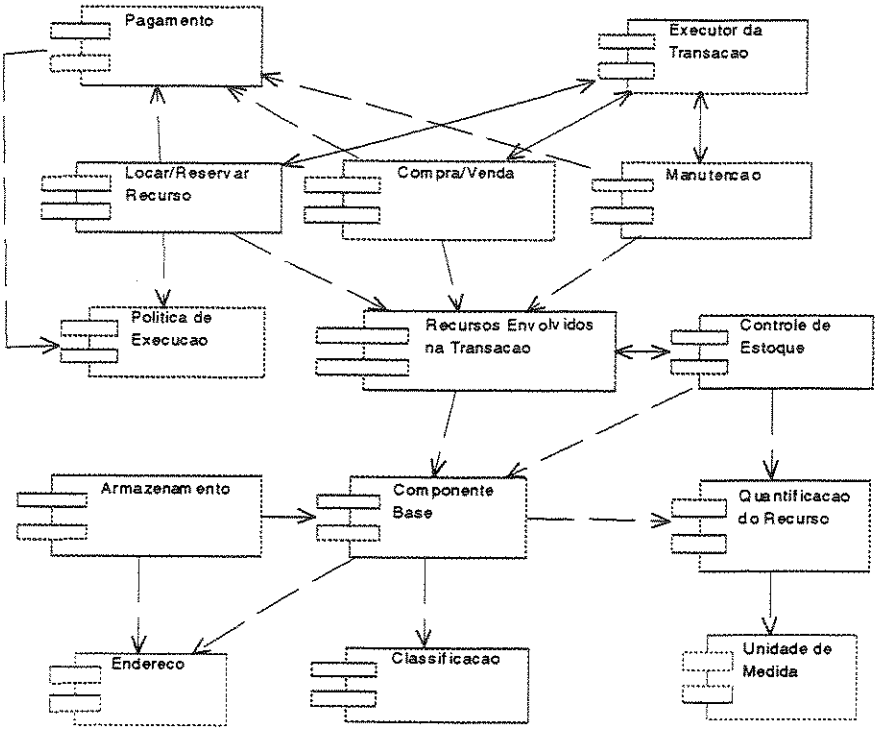


Figura 4.8 – Arquitetura Estruturada em Componentes

Como todos os componentes são persistentes, todos são dependentes do componente Persistência. Porém, essa dependência não é representada na Figura 4.8 para não dificultar sua visualização.

4.4.1 - As Camadas da Arquitetura Estruturada em Componentes

Os componentes da Arquitetura Estruturada em Componentes podem ser divididos em três camadas (Figura 4.9). A camada mais inferior, Camada de Elementos do Domínio, descreve os conceitos fundamentais e essenciais do domínio de gerência de negócio, tais como: recurso, transação e objetos persistentes. Ainda tem a função de prover suporte de software para incremento da aplicação. A camada localizada logo acima dessa, Camada de Regras de Negócio, define as políticas que são implementadas pela aplicação. Fornece algoritmos que realizam funções essenciais da aplicação, de acordo com as características específicas da organização. A terceira camada em ordem crescente é a Camada de Aplicação,

que provê suporte de software para os diferentes contextos onde a aplicação é utilizada e fornece processos e objetos de negócio para o domínio de aplicações de negócios específicos. Além das três camadas onde estão distribuídos os componentes da Arquitetura Estruturada em Componentes têm-se a Camada de Instanciação da Aplicação, onde se encontram as extensões do *framework*, que podem ser realizadas a partir de qualquer uma das camadas descritas anteriormente.

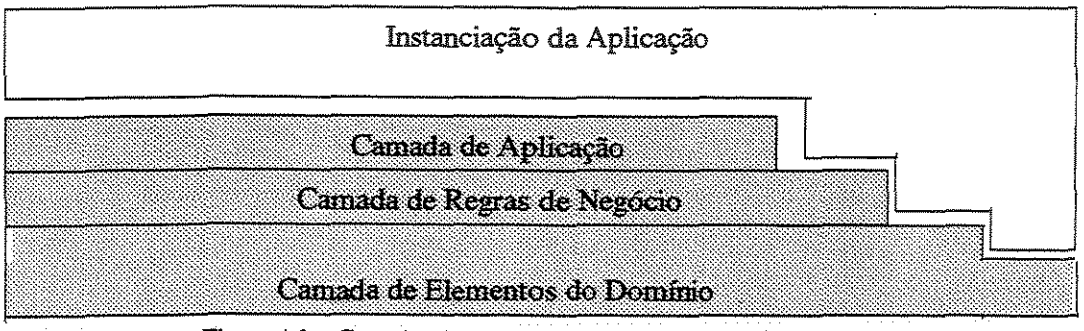


Figura 4.9 – Camadas da Arquitetura Estruturada em Componentes

A Figura 4.10 mostra a distribuição dos componentes da Arquitetura Estruturada em Componentes em cada uma dessas camadas. A Camada 1 é a Camada de Aplicação, a 2 é a de Regras de Negócio e a 3 a de Elementos do Domínio.

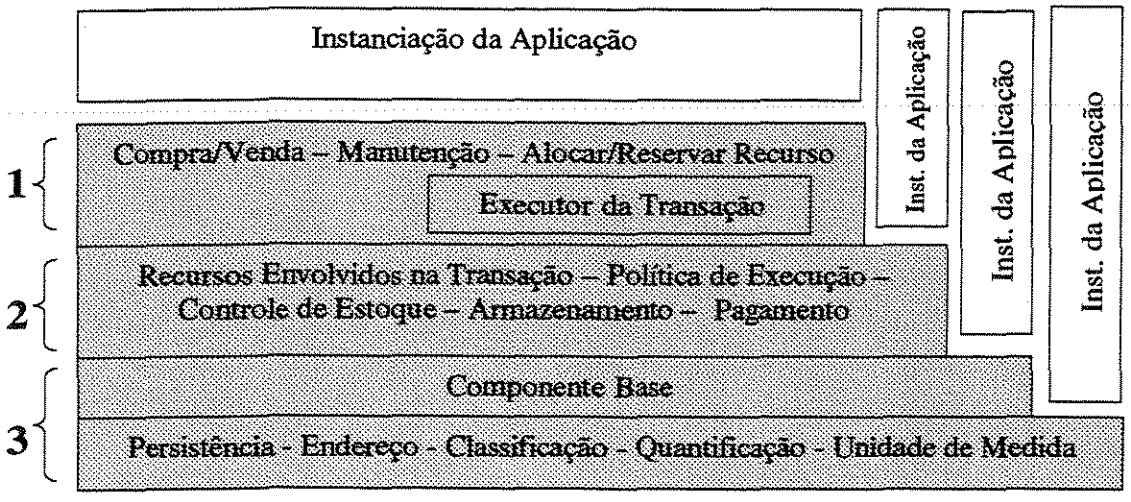


Figura 4.10 – Distribuição dos Componentes nas Camadas da Arquitetura Estruturada em Componentes

A Camada 3, Camada de Elementos do Domínio, é dividida em duas partes apenas para expor para o desenvolvedor que o Componente Base está em um nível de abstração acima dos outros, não havendo distinção desse nível de abstração pelos componentes da Camada 2, Camadas de Regras de Negócio. Na Camada 1, Camada de Aplicação, o Componente

Executor da Transação não tem contato com nenhuma outra camada, significando que ele só pode depender e fornecer funcionalidades para os componentes de sua camada.

4.4.2 - Elementos da Arquitetura Estruturada em Componentes Utilizados na Instanciação do Sistema de Vídeo Locadora

As funcionalidades solicitadas na análise de requisitos do sistema de Vídeo Locadora são realizadas pelas tarefas existentes nos padrões 1, 2 , 4, 11 e 12 da linguagem de padrões (Figura 4.3).

Os componentes da Arquitetura Estruturada em Componentes necessários para a criação do sistema de Vídeo Locadora estão representados na Figura 4.11 e as interfaces utilizadas de cada um deles é apresentada na Figura 4.12.

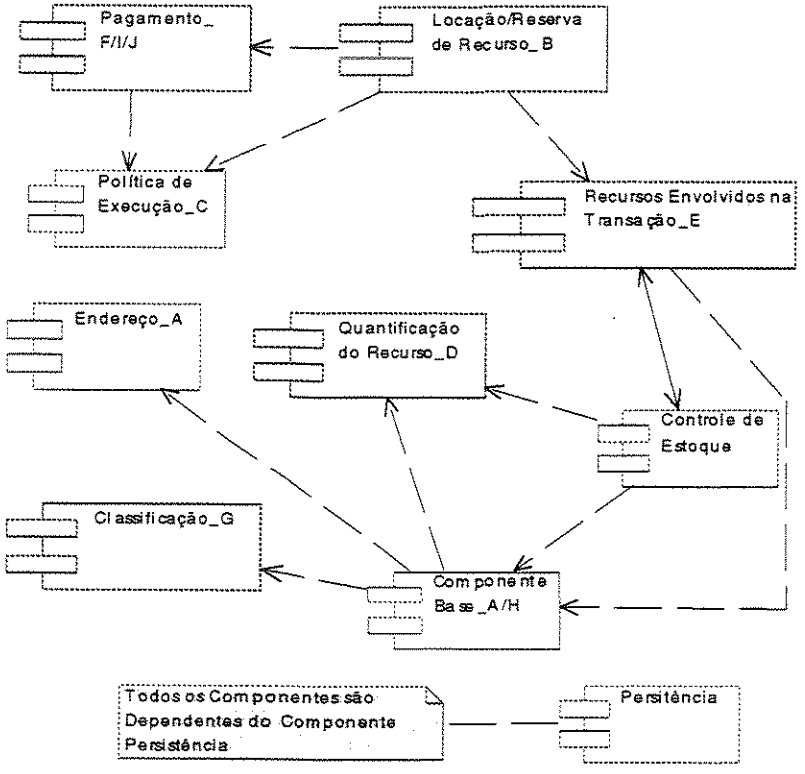


Figura 4.11 - Componentes da Arquitetura Estruturada em Componentes utilizados no desenvolvimento do Sistema de Vídeo Locadora

A sintaxe de nome do componente é: Nome do Componente_Referência do nome da classe pertencente à Figura 4.4.

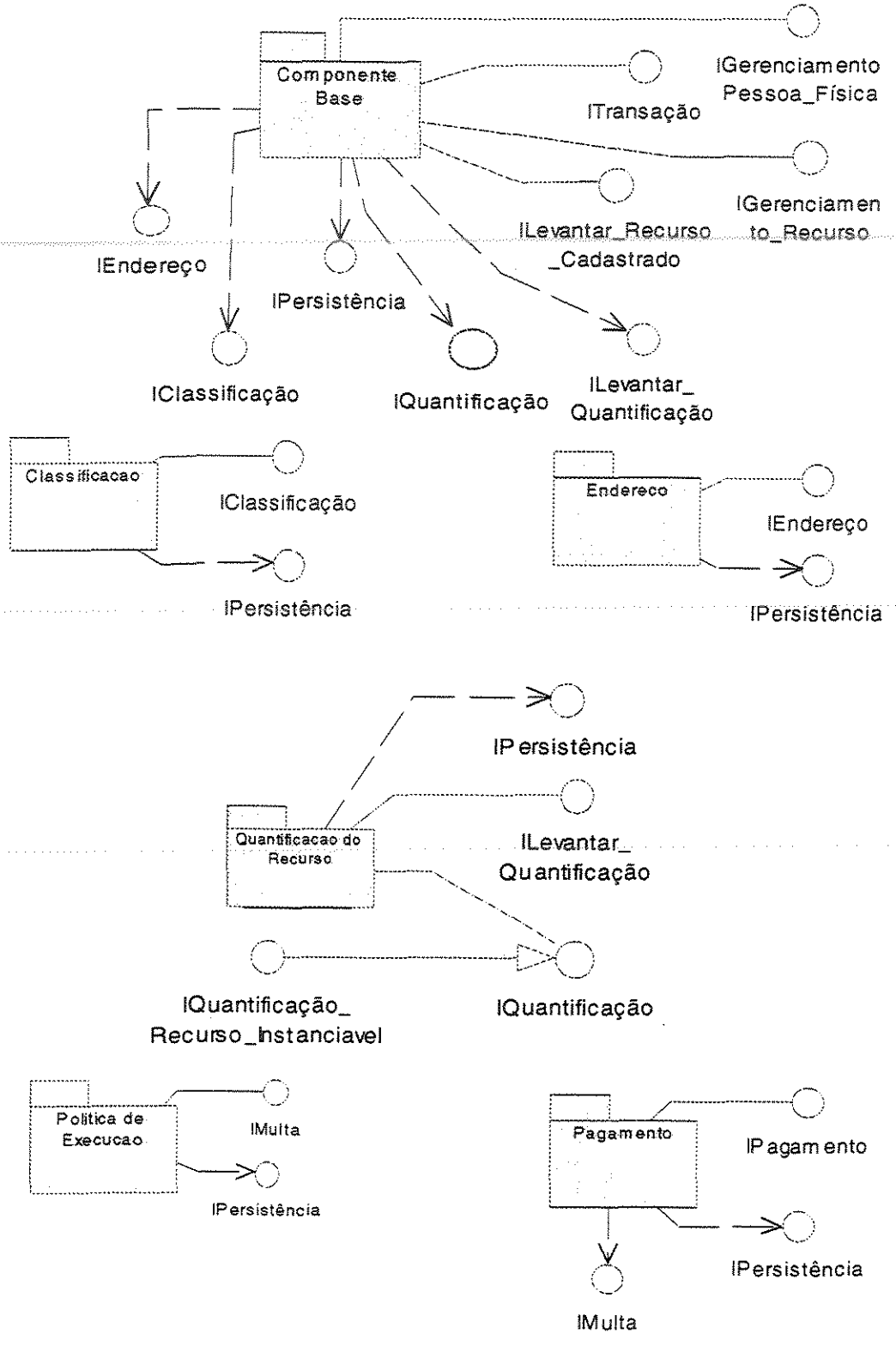


Figura 4.12 – Interface de cada componente da Arquitetura Estruturada em Componentes utilizada na instanciação do Sistema de Vídeo Locadora

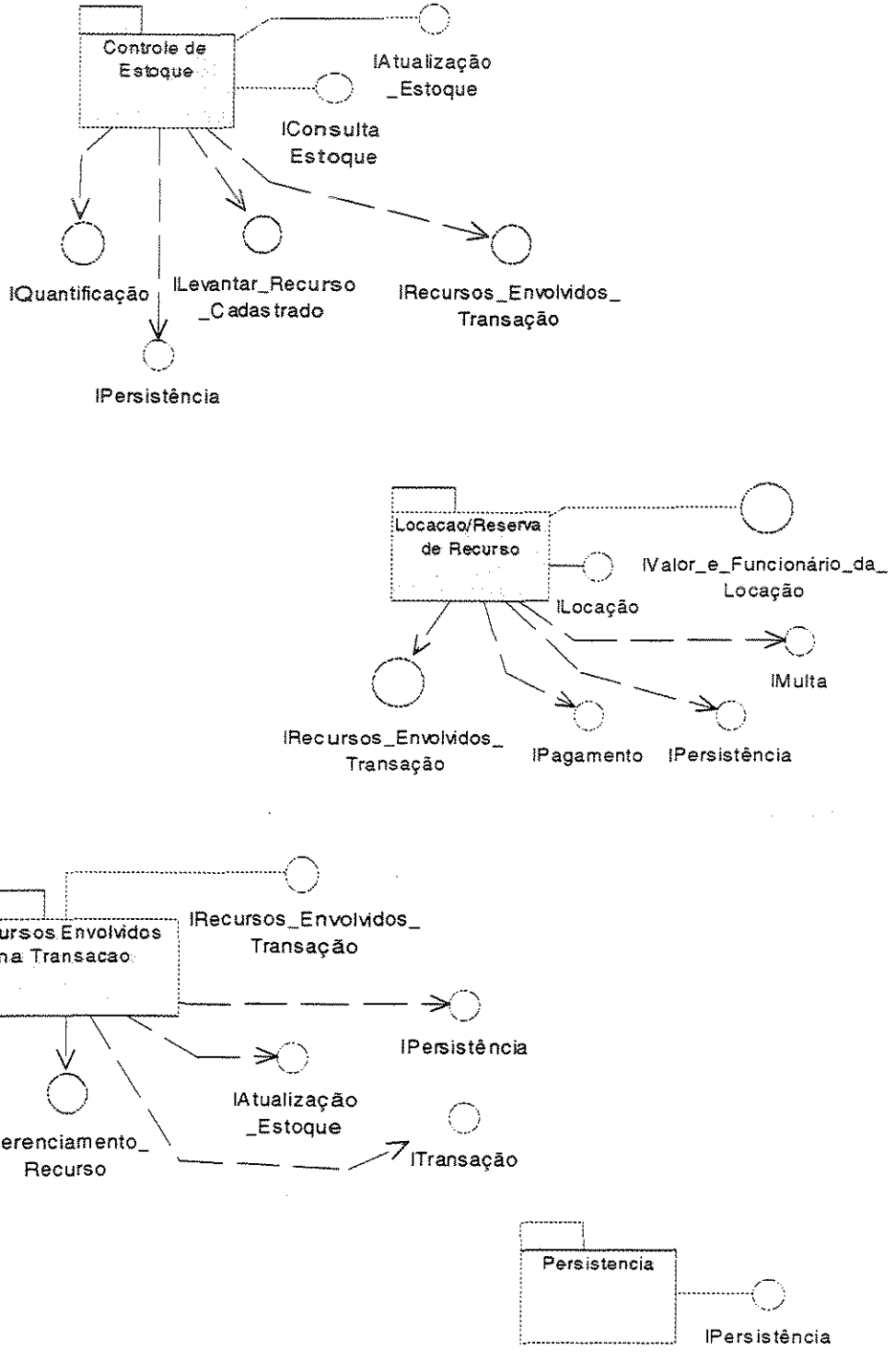


Figura 4.12 – Interface de cada componente da Arquitetura Estruturada em Componentes utilizada na instanciação do Sistema de Vídeo Locadora (Cont.)

4.5 – Obtenção da Arquitetura Baseada em Componentes

Nesta seção será exemplificado como o método 3 (Seção 3.3) é utilizado para gerar a Arquitetura Baseada em Componentes. Essa é obtida através de parte do processo de especificação de software baseado em componentes, proposto por Cheesman e Daniels (Cheesman *et, al.*, 2001) (Seção 3.3).

A primeira etapa, Definição dos Requisitos, tem como resultado o Modelo Conceitual de Negócio e Modelo de Caso de Uso divididos em pacotes, úteis para saber os requisitos, conceitos e relações entre esses requisitos do domínio. No desenvolvimento de ambos os modelos foi utilizada a linguagem de padrões GRN (Seção 4.3.2). Para exemplificação desses modelos obtidos, as Figuras 4.13 e 4.14 mostram, respectivamente, o pacote Locação de Recurso do Modelo Conceitual de Negócio e o Modelo de Casos de Uso.

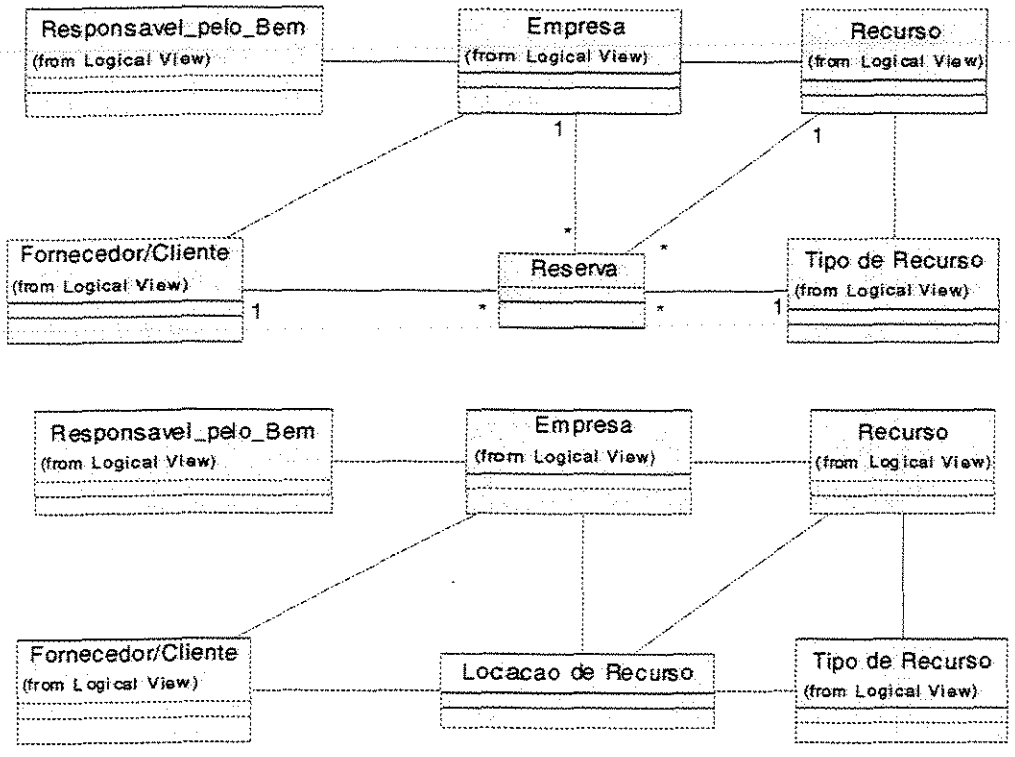


Figura 4.13 - Pacote Locação de Recurso do Modelo Conceitual de Negócio

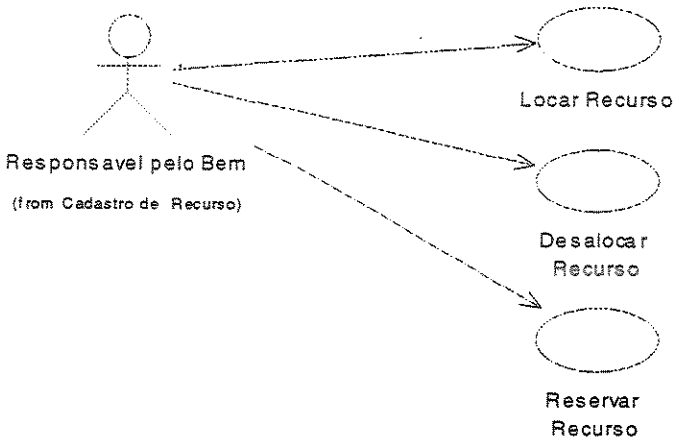


Figura 4.14 - Pacote Locação de Recurso do Modelo Casos de Uso.

Terminada a primeira etapa, deu-se início à segunda, Identificação dos Componentes, determinando como o usuário pode iniciar as operações da aplicação, "tipo de diálogo". Com os "tipos de diálogos" prontos, fez-se a identificação das interfaces do sistema, utilizando os casos de uso. A Figura 4.15 mostra as interfaces do sistema geradas a partir dos casos de uso Locar Recurso, Desalocar Recurso e Reservar Recurso para os "tipos de diálogo" com os mesmos nomes.

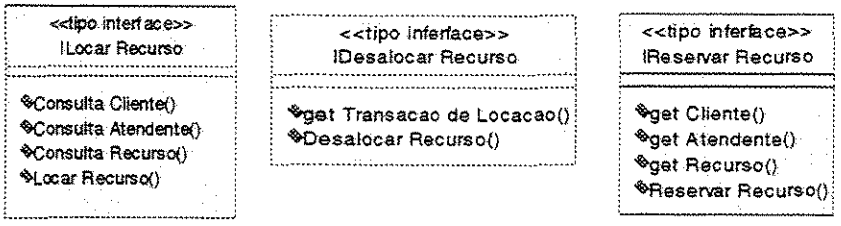


Figura 4.15 – Interfaces de Sistema para Locar, Desalocar e Reservar Recurso

O próximo passo, após a identificação das interfaces de sistema, foi modificar o modelo conceitual de negócio para identificar os componentes que não podem depender de nenhum outro, gerando, assim, o Modelo de Tipo de Negócio do pacote Locar Recurso. A Figura 4.16 apresenta o pacote Locar Recurso do modelo conceitual de negócio modificado para que se possam determinar os componentes de negócio e suas interfaces.

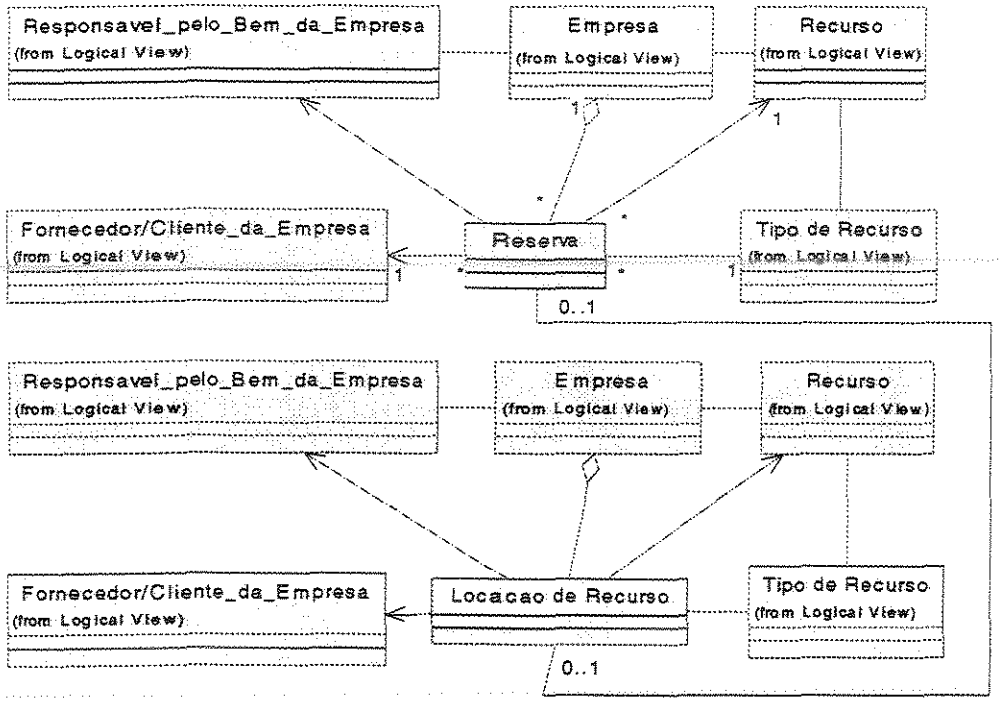


Figura 4.16 - Modelo de Tipo de Negócio do Pacote Locar Recurso

Com os componentes de sistema e de negócio devidamente representados, foi obtido um esboço da Arquitetura Baseada em Componentes, relacionando os componentes que dependem de uma determinada interface com o componente que oferece aquela interface. A Figura 4.17 apresenta os componentes obtidos com os exemplos acima e seus relacionamentos.

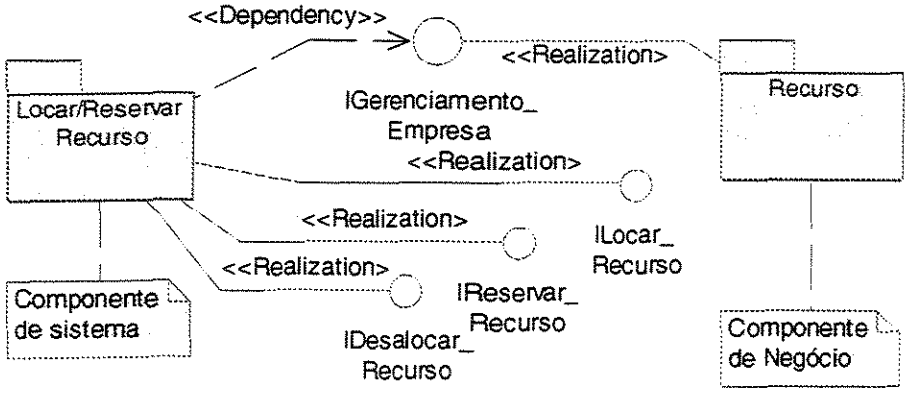


Figura 4.17 – Esboço da Arquitetura Baseada em Componentes

A terceira e última etapa proposta pelo método, Interação dos Componentes, foi realizada e com ela foi possível identificar novas operações para sanar alguns problemas de integridade; como, por exemplo, ao se desalocar um recurso, desvinculava-se a transação desse recurso do cliente, porém não se atualizava o número de recursos disponíveis. Houve, assim, a necessidade de se criar mais uma interface no componente Recursos Envolvidos na Transação para fornecer um método de devolução que, dado o recurso devolvido na transação, possibilite a atualização da disponibilidade desse recurso. Depois de identificados os problemas na arquitetura gerada na etapa anterior, uma atualização na Arquitetura Baseada em Componentes foi realizada. A Figura 4.18 apresenta a arquitetura obtida no final dessa etapa.

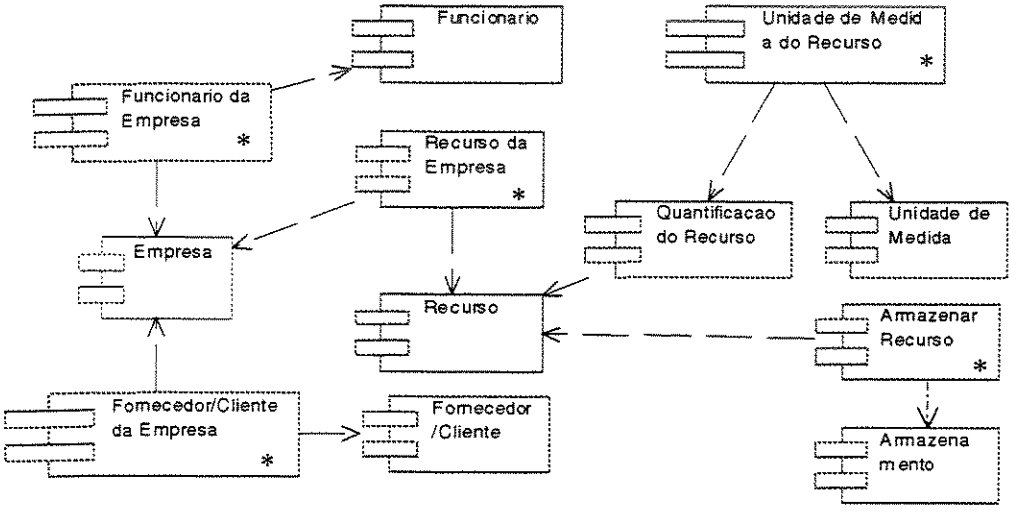


Figura 4.18 – Arquitetura Baseada em Componentes

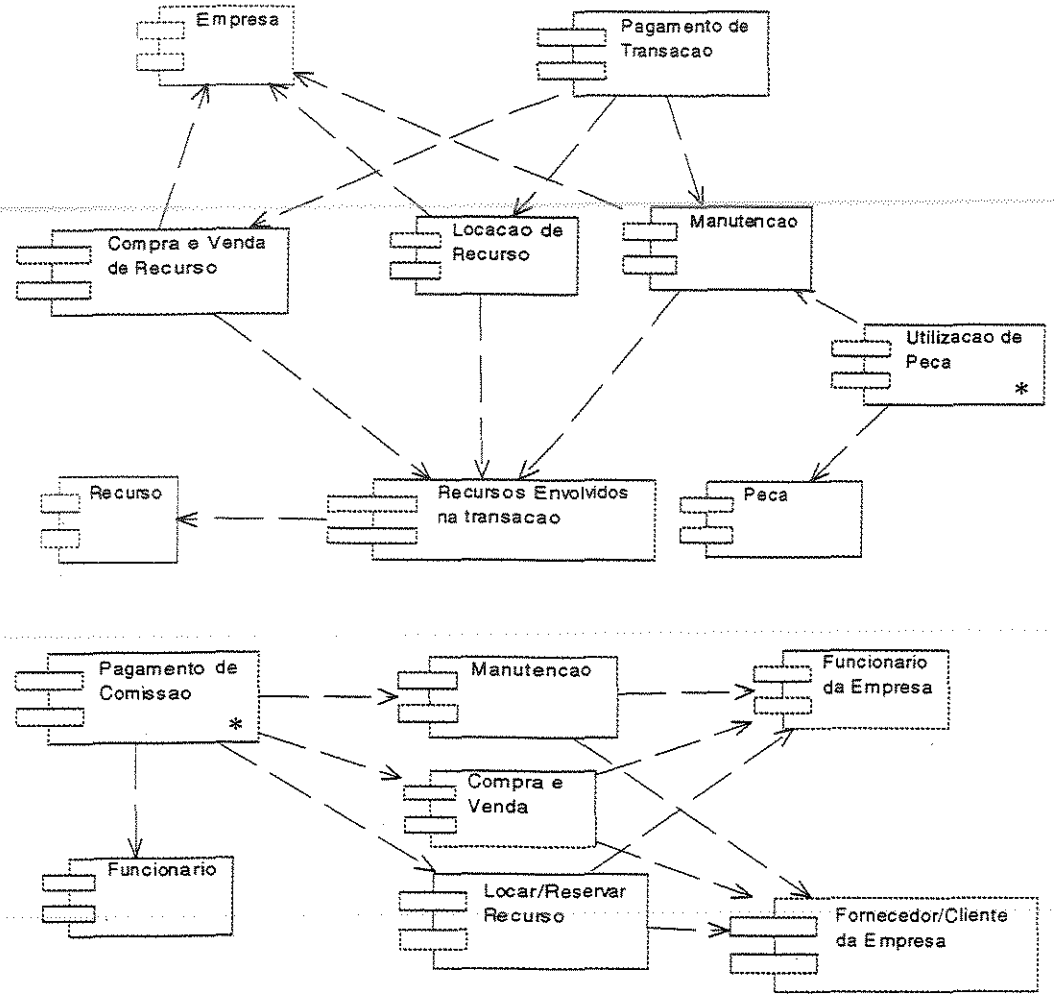


Figura 4.18 – Arquitetura Baseada em Componentes (Cont.)

Devido ao fato de componentes que possuem interface resultante de interface de negócio serem independentes, novos componentes foram desenvolvidos, denominados componentes auxiliares, para que pudesse haver uma associação entre os componentes quando necessário, como, por exemplo, o componente Funcionário da Empresa, que foi criado para propiciar uma relação entre os componentes Funcionário e Empresa. Dessa forma, é possível que o Componente Empresa tenha um código identificador de todos os funcionários pertencentes a uma determinada empresa cadastrada no Componente Empresa. A seguir, são descritos os métodos que possibilitam isso:

Interface Componente Funcionário da Empresa

1.1- Associar Funcionário a Empresa (in Nome Funcionário, in Nome Empresa)

Interface Componente Funcionário

1.2- get Id do Funcionário (in Nome Funcionário, out IdFuncionário)

Interface Componente Funcionário da Empresa

1.3- Associar Funcionário a Empresa (in IdFuncionário, in Nome Empresa)

Caso se queira recuperar a informação de quais são os funcionários pertencentes a uma determinada empresa, as operações das interfaces de cada uma desses componentes são:

Interface Componente Funcionário da Empresa

1.1- Funcionários Pertencentes à Empresa (in Nome Empresa)

Interface Componente Funcionário da Empresa

1.2- get Funcionários da Empresa (in IdEmpresa, out Id´sFuncionários)

Interface Componente Funcionário

1.3- get Nome do Funcionário (in Id´sFuncionários, out Nomes dos Funcionários)

A mesma idéia serve para as demais ocorrências desse fato, marcadas com "*" na Figura 4.18.

4.5.1 – A Arquitetura Baseada em Componentes

Observando a relação existente entre os componentes resultantes da interface de negócio e os componentes desenvolvidos para fazer a associação entre dois daqueles quando necessário, chegou-se à conclusão de que alguns desses componentes poderiam formar um único componente, devido à grande afinidade de dados e operações existentes entre eles. O resultado disso é:

1) O componente Empresa passou a ser constituído pelos componentes: Funcionário, Funcionário da Empresa, Empresa, Fornecedor/Cliente da Empresa, Fornecedor/Cliente.

2) O componente Quantificação do recurso passou a ser constituído pelos componentes: Unidade de Medida, Quantificação do recurso, Unidade de Medida do recurso.

A Figura 4.19 apresenta a Arquitetura Baseada em Componentes após esse novo refinamento.

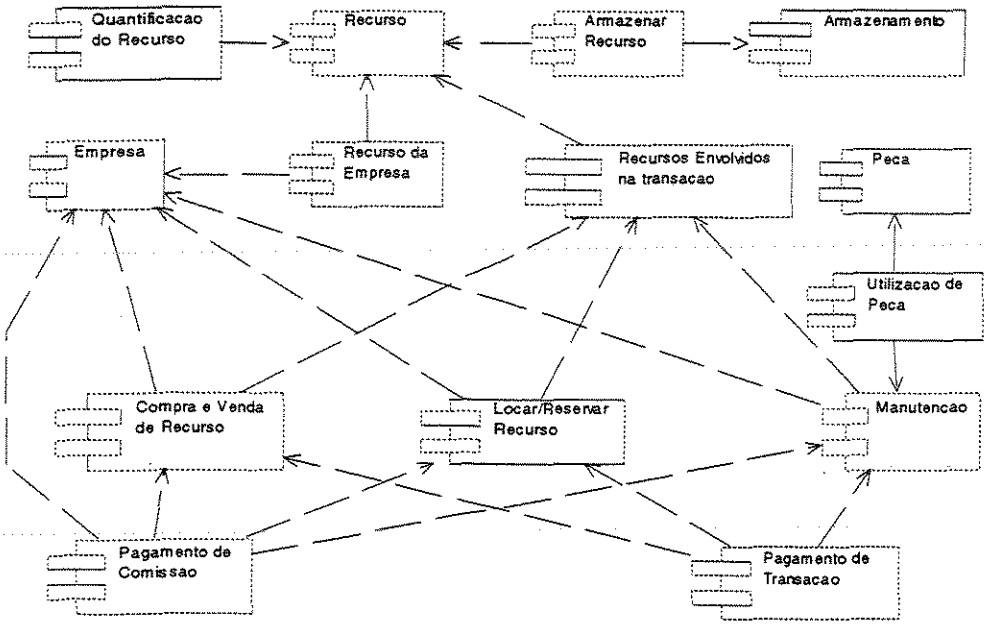


Figura 4.19 – Arquitetura Baseada em Componentes Refinada

4.5.2 - Elementos da Arquitetura Baseada em Componentes Utilizados na Instanciação do Sistema de Vídeo Locadora

Como a linguagem de padrões GRN foi aplicada para o sistema Vídeo Locadora, sabe-se que as funcionalidades que o sistema deve cobrir são as tarefas existentes nos padrões 1, 2 , 4, 11 e 12 (Seção 4.3.2). Então, para facilitar a identificação dos componentes da Arquitetura Estruturada em Componentes necessários para a instanciação deste sistema, foi realizado um estudo para se conhecer a relação dos componentes desta arquitetura com os padrões da linguagem GRN (Figura 4.20).

Nº do Padrão de Projeto	Componentes da Arquitetura Baseada em Componentes
13	Pagamento de Comissão
12	Pagamento de Transação
3,4,5,6,7,9,10,13	Empresa
4 e 5	Locar/Reservar Recurso
6, 7 e 8	Compra e Venda de Recurso
9,10 e 14	Manutenção
15	Utilização de Peça
15	Peça
11	Recursos Envolvidos na Transação
3	Armazenamento
3	Armazenar Recurso
1	Recurso
2	Quantificação do Recurso
3,4,5,6,7,9,10	Recurso da Empresa
	Total de Componentes = 14

Figura 4.20 – Relação de cada componente da Arquitetura Baseada em Componentes com os padrões que eles representam total ou parcialmente

A Figura 4.21 mostra os componentes da Arquitetura Baseada em Componentes necessários para a criação do sistema de Vídeo Locadora e a Figura 4.22, as interfaces utilizadas de cada um desses componentes.

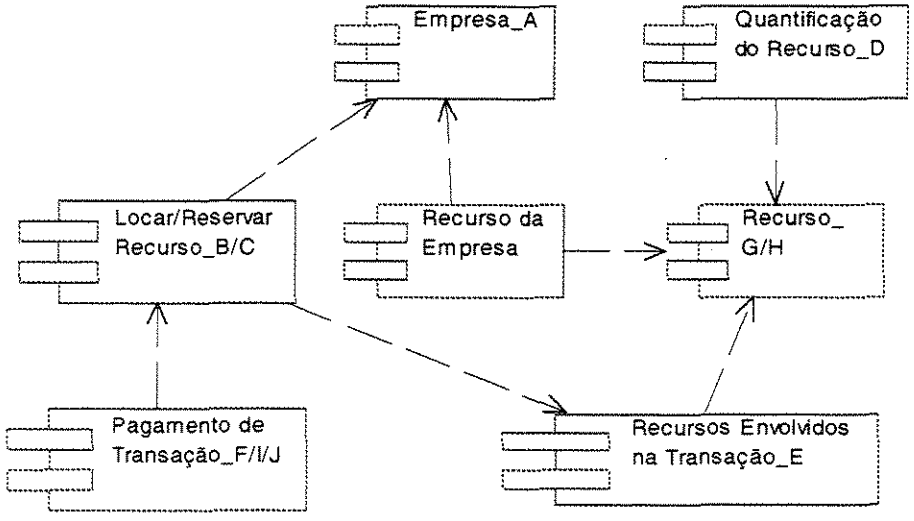


Figura 4.21 - Componentes da Arquitetura Baseada em Componentes utilizados no desenvolvimento do Sistema de Vídeo Locadora

A sintaxe de nome do componente é: Nome do Componente_Referência do nome da classe pertencente à Figura 4.4.

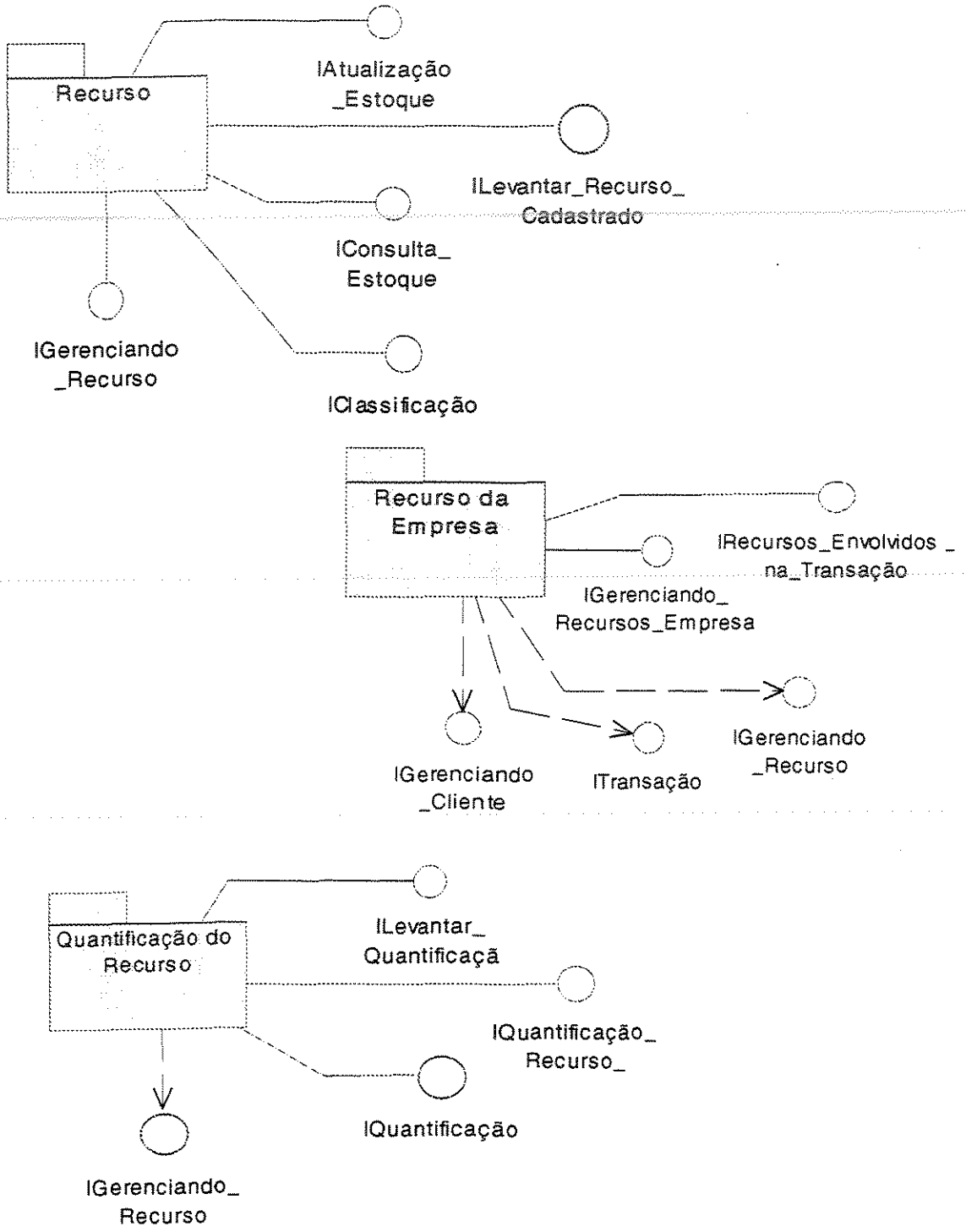


Figura 4.22 – Interface de cada componente da Arquitetura Baseada em Componentes utilizada na instanciação do Sistema de Vídeo Locadora

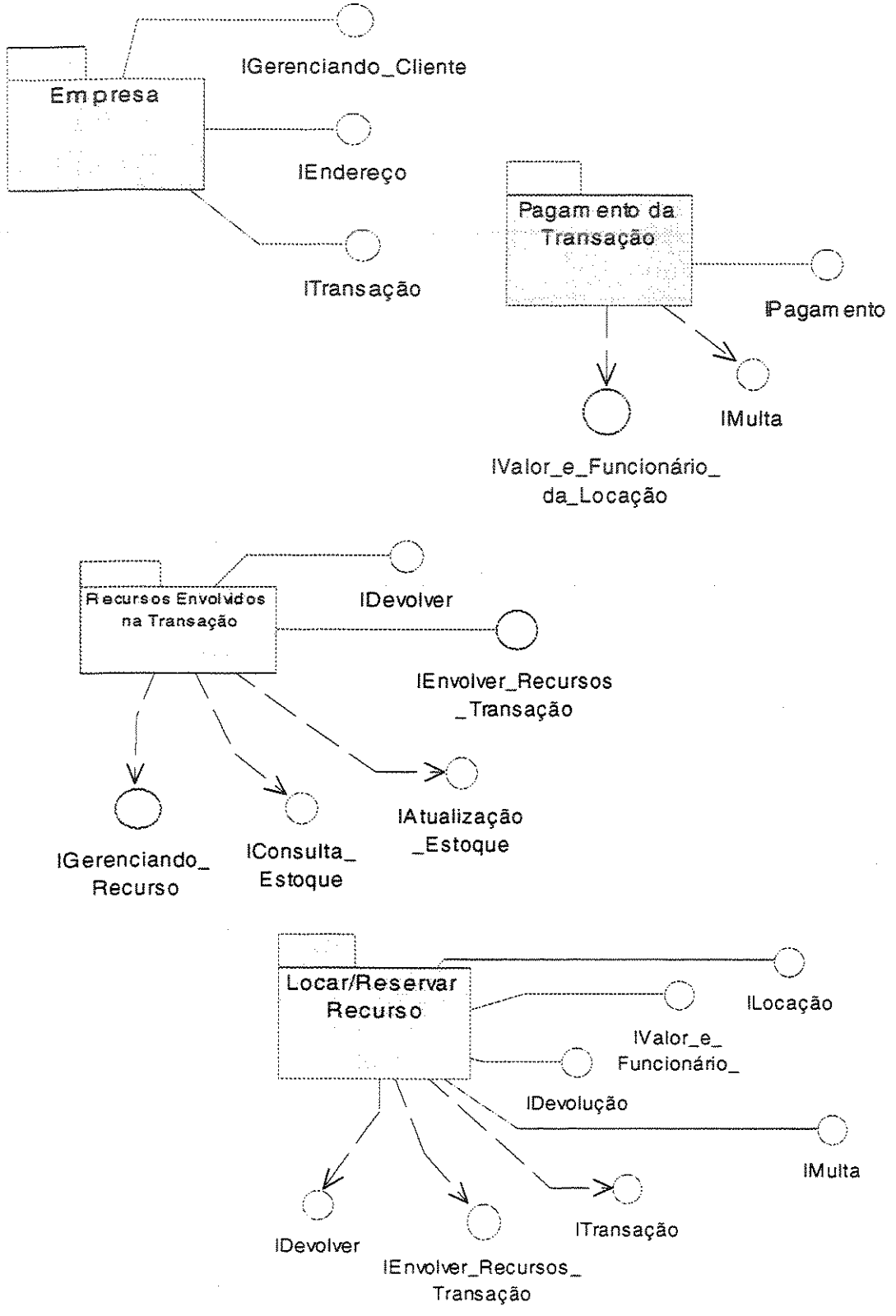


Figura 4.22 – Interface de cada componente da Arquitetura Baseada em Componentes utilizada na instanciação do Sistema de Vídeo Locadora (Cont.)

4.6 – Arquitetura Estruturada em Componentes x Arquitetura Baseada em Componentes

As duas arquiteturas constituídas por componentes desenvolvidas neste estudo, por serem resultados de métodos de desenvolvimento distintos, apresentam algumas decisões de projetos diferentes (ver Figura 4.23).

Componentes da Arquitetura Estruturada em Componentes	**Componentes da Arquitetura Baseada em Componentes**
Pagamento	Pagamento de Comissão Pagamento de Transação
Executor da Transação	Empresa
Locar/Reservar Recurso	Locar/Reservar Recurso
Compra e Venda	Compra e Venda de Recurso
Manutenção	Manutenção Utilização de Peça Peça
Política de Execução	Não se Aplica
Controle de Estoque	Não se Aplica
Recursos Envolvidos na Transação	Recursos Envolvidos na Transação
Armazenamento	Armazenamento Armazenar Recurso
Endereço	Armazenamento Empresa
Classificação	Recurso
Quantificação do Recurso Unidade de Medida	Quantificação do Recurso
Componente Base	Empresa Recurso Recurso da Empresa
Persistência	Não se Aplica
Total de Componentes = 15	Total de Componentes = 14

Figura 4.23 – Relação entre os Componentes das Arquiteturas Estruturada e Baseada em Componentes

A Figura 4.23 apresenta para cada componente da Arquitetura Estruturada em Componentes, Coluna 1, os componentes equivalentes da Arquitetura Baseada em Componentes, Coluna 2. Os componentes Política de Execução e Controle de Estoque existem na Coluna 1 e não existem na Coluna 2, isso porque nos casos de uso e/ou no modelo conceitual de negócio, não se estudou o que tais componentes se propõem a fazer, o que se deu apenas nas etapas seguintes do método utilizado para obtenção da Arquitetura Baseada em Componentes, Coluna 2 (Seção 3.3). Os componentes Endereço e Classificação também existem na Coluna 1 e não existem na Coluna 2, pois, ao se fazer a reengenharia (Seção 3.2)

na Arquitetura Original (Seção 4.3.3) observou-se a independência das classes que implementam as funcionalidades de tais componentes, fato que não foi possível verificar ao se realizar o método de obtenção da Arquitetura Baseada em Componentes, Coluna 2 (Seção 3.3). Mas nada impede que todos os referidos componentes sejam incorporados à Arquitetura Baseada em Componentes após um refinamento. O componente Base da Coluna 1 foi desmembrado em 3 componentes da Coluna 2, sendo os componentes Empresa e Recurso componentes de negócio e o componente Recurso da Empresa um componente auxiliar (Seção 4.5). Tal desmembramento ocorreu devido aos casos de uso terem sido criados separadamente e também à diretriz de modelagem de quando se escolheu quais seriam as interfaces de negócio. É notório que esse desmembramento era maior (Seção 4.5), porém, no segundo refinamento feito na arquitetura, obteve-se o atual estado (Seção 4.5.1). Os motivos apresentados para o desmembramento do Componente Base são válidos também para a existência dos componentes Utilizar Peça, Armazenar Recurso e Pagamento de Comissão na Coluna 2 e a inexistência na Coluna 1.

Com o desmembramento do componente Base da Coluna 1 e a existência de outros componentes por parte Arquitetura Baseada em Componentes, Coluna 2, algumas dependências entre os componentes mudaram, mas a funcionalidade desses continuou a mesma. As únicas alterações significativas em relação à dependência entre os componentes foram:

1) Na Arquitetura Estruturada em Componentes, o recurso que é o responsável por guardar a referência de sua quantificação, já Arquitetura Baseada em Componentes ocorre o inverso.

2) Na Arquitetura Estruturada em Componentes, o componente Recursos Envolvidos na Transação é responsável, entre outra coisas, por obter o ID do cliente /fornecedor da transação e repassá-lo para o componente responsável pela transação. Já na Arquitetura Estruturada em Componentes, o componente Recursos Envolvidos na Transação não tem essa função, pois os componentes responsáveis pelas transações têm dependência direta do componente Empresa.

3) O componente Empresa da Coluna 2 é responsável pelos dados das empresas que utilizam o sistema e também pela persistência de grande parte dos outros componentes (Locar/Reservar recurso, Compra e Venda de recurso, Manutenção, Quantificação do Recurso). Na Coluna 1, esses papéis cabem aos componentes Base e Persistência, respectivamente.

4.7 – Considerações Finais

Para a instanciação de um sistema nas arquitetura constituída por componentes, apenas a utilização de sua documentação é suficiente, não havendo necessidade de estudá-la em sua totalidade, uma vez que grande parte dela pode ser estudada separadamente, dependendo do componente que se pretende utilizar. Nessa documentação, encontra-se a definição de cada componente, suas respectivas interfaces, o que cada um dos métodos das interfaces propicia e suas dependências. A utilização da linguagem de padrões não é obrigatória, mas é fato que, quando se tem uma linguagem de padrões a seguir, a análise e a identificação de componentes e interfaces utilizar ficam facilitadas, como mostrado nas Seções 4.4.2 e 4.5.2.

Uma das vantagens de se utilizar uma arquitetura constituída de componentes é que, ao se escolher um elemento arquitetural (componente), este pode ter, além das interfaces padrões, interfaces específicas para um determinado sistema, o que adiantaria, em muito, a modelagem. Uma outra vantagem é que os elementos dessa arquitetura (componentes) encapsulam características comuns de um determinado assunto, fornecendo um nível bem mais alto que simples classes. Dessa forma, quando bem projetados, possuem um nível de coesão alto, o que garante também uma independência maior de outros elementos.

Uma das desvantagens dessa arquitetura é a rigidez propiciada pelos componentes, pois como eles encapsulam um maior número de tarefas, o que pode acontecer é a necessidade de unir tarefas que, antes foram consideradas independentes, para gerar uma nova tarefa. Para conseguir resolver esse impasse, algumas das soluções são: (i) caso os componentes sejam caixa branca, modificar a sua interface e/ou sua estrutura interna; (ii) utilizar uma interface de forma adaptada para atingir o que é solicitado pelo sistema e (iii) caso os componentes sejam caixa preta, utilizar uma forma de adaptação de componentes, como a de Weis (Weis, 2001).

4.8 – Resumo

Os métodos propostos no Capítulo 2 para o estudo de um *framework* e obtenção de uma arquitetura estruturada e outra baseada em componentes foram utilizados, respectivamente, para o estudo do *framework* GREN, do domínio de gestão de recursos de negócio, e para a obtenção das arquiteturas estruturada e baseada em componentes para este mesmo domínio. Constata-se, assim, a boa aplicabilidade de tais métodos, visto que os resultados obtidos eram os esperados.

Foi apresentada a linguagem de padrões GRN, que serviu como base para o desenvolvimento do *framework* GREN e que foi de grande utilidade para a instanciação das arquiteturas para o sistema de Vídeo Locadora. Isso porque, quando aplicada para o mesmo sistema, apresentou os padrões e, por conseguinte, as tarefas que deveriam ser abordadas por cada uma das arquiteturas.

Discutiram-se as vantagens e desvantagens encontradas na instanciação das arquiteturas baseada em classes e constituída por componentes e abordaram-se as diferenças das arquiteturas Estruturada e Baseada em Componentes, relatando que o processo utilizado para a obtenção da Arquitetura Baseada em Componentes possibilitou a identificação de alguns componentes um pouco mais específicos, o que facilita o entendimento de cada um dos componentes isoladamente e da arquitetura. Entretanto, não foi possível extrair algumas decisões de análise/projeto, o que tornou alguns componentes um pouco mais complexos.

Apresentou-se, também, a arquitetura do *framework* GREN, denominada Arquitetura Original, a Arquitetura Estruturada em Componentes e a Arquitetura Baseada em Componentes, que serão as arquiteturas utilizadas no experimento para a análise arquitetural, tanto quantitativa como qualitativa, apresentada no capítulo seguinte.

Capítulo 5

Análise Arquitetural

Este capítulo tem como objetivo realizar a análise arquitetural qualitativa e quantitativa, a partir dos métodos de análise de arquiteturas propostos no Capítulo 3, utilizando, para a realização do experimento, as arquiteturas Original, Estruturada em Componentes e Baseada em Componentes, obtidas no Capítulo 4. Propõe-se também investigar a aplicabilidade do paradigma de desenvolvimento baseado em componentes na construção de *frameworks* em um domínio de gestão de recursos de negócio.

A Seção 5.1 aborda a análise qualitativa das arquiteturas, onde será utilizado o ATAM, (Seção 3.4.1) e a Seção 5.2 apresenta a análise quantitativa das arquiteturas a partir do método proposto na Seção 3.4.2, Método de Análise Quantitativa.

5.1 – Aplicação do ATAM para a Avaliação Qualitativa

O sistema escolhido para realizar a análise qualitativa das arquiteturas de *framework* utilizando o ATAM (Seção 3.4.1) é o de Vídeo Locadora (Seção 4.2). Os requisitos não-funcionais manutenibilidade e reusabilidade foram os escolhidos por este estudo para serem analisados nas arquiteturas envolvidas.

As seções seguintes descrevem a realização de cada um dos passos do ATAM aplicados para desenvolver essa avaliação.

5.1.1 – 1° Passo: Coleta dos Cenários

Manutenibilidade

A análise da manutenibilidade será guiada por dois cenários: (i) o primeiro representando a necessidade de adaptação (correção), que é a manutenção realizada devido a uma eventual abordagem incorreta de algum requisito previsto na análise de requisitos ou à sua não-contemplação pela modelagem; (ii) o segundo cenário refere-se à necessidade de evolução do sistema, ou seja, acrescentar mais alguma funcionalidade que não havia sido solicitada anteriormente.

Primeiro Cenário – Manutenção de Adaptação: Segundo Cagnin (Cagnin, 2002), apesar do requisito funcional devolução de recurso fazer parte do domínio da linguagem GRN (Seção 4.3.2), ele não é coberto. Como todas as arquiteturas a serem avaliadas sofreram influência desta linguagem de padrões, nenhuma delas, teoricamente, dá suporte para tal requisito. Dessa forma, a descrição do primeiro cenário é:

> O cliente desenvolvedor modifica a arquitetura do *framework* para que esse passe a apoiar o requisito funcional devolução.

Segundo Cenário – Manutenção de Evolução: É referente à evolução do sistema:

> O cliente usuário solicita ao cliente desenvolvedor um controle de acesso ao sistema.

Reusabilidade

A análise da reusabilidade, que determina o quão fácil é conseguir elementos reutilizáveis de uma arquitetura, é também guiada por dois cenários, representando antecipadamente o interesse de reaproveitamento de partes da arquitetura já desenvolvida.

Terceiro Cenário – Reuso do Padrão 1: O cliente desenvolvedor deseja utilizar em um outro sistema o que é proposto pelo Padrão 1 (Identificar o Recurso) da linguagem de padrões GRN (Seção 4.3.2).

Quarto Cenário – Reuso dos Padrões 11-12-13: O cliente desenvolvedor deseja utilizar em um outro sistema o que é proposto pelos padrões 11, 12, 13 (Itemizar Transação do Recurso, Pagar pela Transação do Recurso, Identificar o Executor da Transação) da linguagem de padrões GRN (Seção 4.3.2).

5.1.2 – 2° Passo: Obtenção dos Requisitos, Restrições e Descrição do Ambiente

Manutenibilidade

Na análise da manutenibilidade, os requisitos considerados são tipicamente derivados dos cenários para representar o sistema do ponto de vista do usuário desenvolvedor. Nesse caso, os requisitos são:

RM1 – A alteração do número de elementos da arquitetura para a realização de uma manutenção deve ser o menor possível.

RM2 – Os elementos da arquitetura devem ser específicos para um conjunto de tarefas.

RM3 – Dado que um cliente desenvolvedor tenha que realizar uma determinada manutenção, o número de elementos da arquitetura a serem entendidos deve ser o menor possível.

Somando-se a esses requisitos, deve-se assumir que o padrão comportamental e o ambiente de manutenção são os seguintes:

- A estrutura básica proposta pela linguagem de padrões deve ser mantida.
- Os requisitos solicitados para evolução não devem afetar outras partes da linguagem GRN para as quais eles não foram solicitados, porém a adaptação deve ser fácil, caso isso venha a acontecer.
- Os requisitos de manutenção pedidos devem pertencer ao domínio do *framework*.

Reusabilidade

Para a avaliação da reusabilidade os seguintes requisitos foram identificados:

RR1 - A reutilização deve ocorrer de maneira facilitada, com a menor aquisição possível de requisitos não desejáveis.

RR2 - A reutilização pode ocorrer tanto de um padrão completo como apenas de parte desse.

Devido à possibilidade de possuir pontos de compromisso, cada um desses ítens supracitados é passível de inspeção, validação e questionamento, servindo como um resultado do ATAM.

5.1.3 – 3º Passo: Descrição da Visão Arquitetural

Visto que o ATAM foi criado para ilustrar os compromissos arquiteturais, há a necessidade de algumas arquiteturas para análise. Serão consideradas três opções arquiteturais, sendo que cada uma delas será descrita com o intuito de mostrar como é feita a instanciação do sistema de Vídeo Locadora. As arquiteturas a serem consideradas são:

1) Arquitetura Original (Seção 4.3.3) obtida no primeiro passo do experimento (Seção 4.3).
2) Arquitetura Estruturada em Componentes, gerada no segundo passo do experimento (Seção 4.4).
3) Arquitetura Baseada em Componentes (Seção 4.5.1) desenvolvida no terceiro passo do experimento (Seção 4.5).

As tarefas arquiteturais para o desenvolvimento do sistema de Vídeo Locadora (Seção 4.2) são descritas na Seção 4.3.2, onde foi aplicada a linguagem de padrões GRN (Seção 4.3.2) para esse sistema. Os elementos a serem analisados são os que realizam tais tarefas. Os elementos das Arquiteturas Original, Estruturada em Componentes e Baseada em Componentes estão descritos, respectivamente, nas Seções 4.3.3, 4.4.2 e 4.5.2.

A principal diferença dessas três arquiteturas a serem avaliadas é que a primeira opção arquitetural, a Arquitetura Original, utiliza unicamente o paradigma orientado a objetos; a segunda opção, Arquitetura Estruturada em Componentes, utiliza o paradigma orientado a objetos associado com o termo *componentware*, proposto por Pree (Pree, 1997) (ver Seção 2.2). Já a terceira opção arquitetural difere da segunda opção apenas na forma como foi criada. Enquanto a arquitetura da segunda opção foi desenvolvida a partir de uma reengenharia baseada na Arquitetura Original (Seção 4.4), a arquitetura da terceira opção foi desenvolvida desde o início com o pensamento voltado para componentes (Seção 4.5).

5.1.4 – 4° Passo: Análise Específica de Cada Atributo

A análise isolada em cada um dos cenários (ver Seção 5.1.1) dos requisitos não-funcionais (Manutenibilidade e Reusabilidade) será descrita, agrupando-as pela arquitetura e pelo requisito não funcional a que se referem. Essa análise será realizada argumentando as facilidades e dificuldades encontradas na realização de cada um dos cenários.

A realização e análise dos cenários para os requisitos não-funcionais em questão são apresentadas a seguir.

Realização dos Cenários / Análise da Manutenibilidade

A análise da manutenibilidade, descrita a seguir, apresenta os resultados para o estudo de caso proposto pelo Primeiro e Segundo cenários da coleta dos cenários (Seção 5.1.1), considerando as restrições descritas na Seção 5.1.2.

O primeiro cenário a ser realizado refere-se à necessidade de acrescentar-se mais um requisito funcional ao sistema para o qual foi previsto na análise de requisitos e as arquiteturas não o implementam (Primeiro Cenário - Manutenção de Adaptação, Seção 5.1.1). Para isso, é necessário localizar todos os elementos que serão envolvidos no requisito e modificá-los de acordo com a necessidade. No estudo de caso desta dissertação, o requisito em questão é o de Devolução.

O segundo cenário é o acréscimo de um requisito funcional que anteriormente não pertencia à análise de requisitos (Segundo Cenário - Manutenção de Evolução, Seção 5.1.1). Esse requisito é o acesso do sistema se tornar restrito.

Análise da Manutenibilidade da Arquitetura Original

O primeiro cenário - Manutenção de Adaptação é resolvido com o acréscimo do método Devolução (in Código do Cliente, in Código_Fita, in Data_Entrega) na classe BusisnessResourceTransaction_7 (Figura 4.6). Esse método dispara mensagens em várias outras classes como: Paymente_9, FineRate_6, QualifiableObjetct_18, PersistentObject_3,

DestinationParty_17, etc, para atualizar o sistema na sua parte de estoque, contabilidade e de cliente.

Assim, para que esse novo requisito seja acrescentado ao sistema, o desenvolvedor tem a necessidade de entender a arquitetura em quase sua totalidade, já que, no mínimo, pode haver a necessidade de recalcular o valor a ser pago pelo aluguel, atualizá-lo no sistema, atualizar a disponibilidade da fita, desvincular o cliente da fita e tornar todos esses dados persistentes. Só se consegue isso estudando método por método de cada uma das classes envolvidas nessas tarefas.

O segundo cenário - Manutenção de Evolução pode ser realizado de várias formas, dependendo de como o usuário final deseja esse cenário:

(1) Acrescentar os atributos *login* e senha na classe BusisnessResourceTransaction_7 (Figura 4.6) e os métodos *get* e *set* para cada atributo, além do método Verificar_Permissão (in login, in senha). Com isso, fica garantido que qualquer operação de Locação, Venda ou Manutenção se realizará apenas se o usuário possuir permissão. Entretanto, a restrição de acesso ao cadastro e consulta de clientes e recursos, por exemplo, não estaria garantida.

Nesse caso, não haveria muita dificuldade em realizar a manutenção de adaptação, a não ser pelo desconhecimento da classe que deve ser alterada.

(2) Para se alcançar a restrição total do sistema, uma solução é criar uma nova classe, onde se encontrariam os objetos, com seus *logins* e senhas; e a utilização das outras classes ficaria dependente da existência de tais objetos.

Essa solução seria um pouco mais trabalhosa que a anterior, já que seria necessário tornar qualquer outra classe do sistema dependente de uma única classe, o que pode atrapalhar também manutenções futuras.

(3) Uma outra possibilidade seria permitir acesso restrito a cada um dos objetos do sistema dependendo da permissão do usuário. Para tal, seria necessário, por exemplo, que o usuário informasse um *login* e uma senha a cada objeto novo.

Essa solução seria extremamente difícil de ser realizada, já que para todo objeto criado no sistema haveria um nível de restrição e uma comparação com a permissão do usuário, aumentando em muito a dependência entre os elementos do sistema.

Análise da Manutenibilidade da Arquitetura Estruturada em Componentes

Para que a arquitetura passe a apoiar o requisito funcional devolução (Primeiro Cenário - Manutenção de Adaptação), o método Devolver_Locação (in data de entrega, in IdLocação) deve ser criado na interface ILocação do componente Locação/Reserva de Recurso (Figura 4.11). Além disso, uma dependência entre esse componente e o componente Política de Execução (Figura 4.11) deve existir para que se possa calcular a multa caso haja atraso na entrega. Já no componente Recursos Envolvidos na Transação (Figura 4.11), na interface IRecurso_Envolvidos_Transação, deve ser criado o método Devolução (IdDevolução).

Para a inserção desse novo requisito no sistema, o desenvolvedor deve ter conhecimento da interface dos componentes Política de Execução e Componente Base, bem como conhecimento interno dos componentes Locação/Reserva de Recurso e Recursos Envolvidos na Transação.

Todo o processo descrito anteriormente seria necessário caso essa arquitetura não desse nenhum apoio ao requisito funcional Devolução. Entretanto, ela já o apoiava em parte e faltavam apenas as alterações no componente Recursos Envolvidos na Transação.

O outro requisito a ser inserido, acesso restrito ao sistema (Segundo Cenário - Manutenção de Evolução), pode ser possibilitado por essa

arquitetura, seguindo as intenções do cliente usuário, descritos na Análise da Manutenibilidade da Arquitetura Original, da seguinte forma:

(1) Criação de um componente Acesso, cuja interface possuiria os métodos: Cadastro de Senha, Cadastro de Login, Edição de Senha, Edição de Login, Cancelar Senha e Login, Verificar Restrição. Após a criação desse novo componente, uma dependência com o componente Locação/Reserva de Recurso (Figura 4.11) seria representada, significando que uma locação só pode ser realizada se o usuário possuir permissão. E uma outra dependência entre o Componente Acesso e o Componente Persistência (Figura 4.11) seria criada para que se pudessem armazenar os dados do Componente Acesso.

Nesse caso, haveria a necessidade de alterar a interface do componente Locação/Reserva de Recurso, para que este passasse a utilizar as interfaces fornecidas pelo componente Acesso e saber as interfaces fornecidas pelo componente Persistência para que o componente Acesso se tornasse persistente. Entretanto, os outros componentes não teriam que ser estudados e nem alterados porque as interfaces utilizadas e fornecidas por eles não são necessárias para a realização desse cenário.

(2) O mesmo componente Acesso descrito anteriormente seria criado, só que agora todos os componentes do sistema dependeriam dele.

Nesse caso, o processo descrito para possibilitar o acesso restrito do componente Locação/Reserva de Recurso (Figura 4.11) seria o mesmo para cada um dos outros componentes, sempre envolvendo o mesmo número de componentes, dois, sendo eles o Acesso e o componente em questão. Não há mais necessidade de envolver o componente Persistência, pois uma vez realizada a persistência do componente Acesso, ela fica válida para todas as outras operações.

(3) A terceira forma de possibilitar a implementação do acesso restrito seriam a alteração dos métodos de cada uma das interfaces dos componentes, em que cada um "chamaria" o método Verificar_Restrição (in Senha, in Login), que é implementado dentro de cada componente e a criação dos métodos Cadastro de Senha, Cadastro de Login, Edição de Senha, Edição de Login, Cancelar Senha e Login em uma nova interface de cada componente.

Essa possibilidade acarreta o estudo isolado de cada componente para poder adaptá-los a esse novo requisito.

Análise da Manutenibilidade da Arquitetura Baseada em Componentes

O requisito funcional devolução (Primeiro Cenário - Manutenção de Adaptação) já é apoiado pela arquitetura, devido ao desenvolvimento de diagramas de colaboração no processo de criação da mesma – a realização dos diagramas teve o objetivo de encontrar possíveis problemas de integridade. Porém, caso esse requisito não fosse apoiado por essa arquitetura, seria necessária apenas a modificação dos componentes Locar/Reservar Recurso (Figura 4.21), para que pudesse receber o pedido de devolução e Recursos Envolvidos na Transação (Figura 4.21), o qual liberaria o recurso para uma nova locação propriamente dita.

O outro requisito a ser inserido, acesso restrito ao sistema (Segundo Cenário - Manutenção de Evolução), pode ser possibilitado por essa arquitetura, seguindo as intenções do cliente usuário, descritos na Análise da Manutenibilidade da Arquitetura Original, da seguinte forma:

(1) Criação de um componente Acesso, cuja interface possuiria os métodos: Cadastro de Senha, Cadastro de Login, Edição de Senha, Edição de Login, Cancelar Senha e Login, Verificar Restrição. Após a criação desse novo componente, uma dependência com o componente Locar/Reservar Recurso (Figura 4.21) seria representada, significando que uma locação só pode ser realizada se o usuário possuir permissão.

Nessa arquitetura, a persistência dos novos dados deve ser implementada dentro do novo componente, sendo este um componente de negócio.

(2) O mesmo componente Acesso descrito anteriormente seria criado para possibilitar o acesso restrito a cada um dos componentes de sistema. Já os componentes de negócio, que não dependem de nenhum outro, teriam que ter suas interfaces alteradas para possibilitar esse requisito.O processo para alteração dessa arquitetura é o mesmo descrito na realização deste cenário na Arquitetura Estruturada em Componentes, ítens (2) e (3).

(3) A terceira forma de possibilitar a implementação do acesso restrito seriam a alteração dos métodos de cada uma das interfaces dos componentes, em que cada um "chamaria" o método Verificar_Restrição (in Senha, in Login), que é implementado dentro de cada componente e a criação dos métodos Cadastro de Senha, Cadastro de Login, Edição de Senha, Edição de Login, Cancelar Senha e Login em uma nova interface de cada componente. Também teria que possibilitar a persistência desses novos dados dentro de cada componente de negócio.

Realização dos Cenários / Análise da Reusabilidade

Os cenários a serem realizados, terceiro (Reuso do Padrão 1) e quarto (Reuso dos Padrões 11-12-13) cenários (Seção 5.1.1), e avaliados são, respectivamente, os de reutilização do padrão 1 e dos padrões 11, 12, 13 da linguagem de padrões GRN (Seção 4.3.2).

Análise da Reusabilidade da Arquitetura Original

Para reutilizar a forma como essa arquitetura implementa o padrão 1 (Terceiro Cenário – Reuso do Padrão 1), é necessário inserir no novo sistema as classes QualifiableObject, Resoucer, NestedType e SimpleType (Figura 4.5). Caso queira tornar essas classes reutilizadas persistentes, é necessário inserir também as classes StaticObject, PersistentObject, OIDManager, ConnectionManager (Figura 4.5), totalizando assim 8 (oito) classes.

Já para realizar a reutilização dos padrões 11, 12 e 13 de forma completa, as seguintes classes devem ser reaproveitadas: TransactionItem, ItemQuantificationStrategy, SingleResTransItem, MeasResTransItem, LotResTransItem, InsResTransItem, TransQuantStrategy, TransactionExecutor, ExecutorTeam, Comission, Payment, PaymentStrategy, ImmediateReceiving, Cash, EletronicTransfer, MoneyOrder, LaterReceiving, CreditCArd, Check, Invoice, Payment on Delivery, AbstractCalculator, NumberRangerCalculator, FineRate, ExactNumberCalculator, InterestRate (Figura 4.5). Além dessas 26 (vinte e seis) classes, devem-se reparar os relacionamentos que elas têm com outros padrões para se fazer uma reutilização mais adequada. É bom lembrar que essas classes não são persistentes e, caso as queira tornar persistentes, podem-se reutilizar as classes StaticObject, PersistentObject, OIDManager, ConnectionManager (Figura 4.5); totalizando 30 (trinta) classes a serem reutilizadas.

A reutilização nesse tipo de arquitetura é, de certa forma, inviável, pois há que se agrupar vários elementos arquiteturais (classes) para conseguir reutilizar o que se pretende, além, é claro, de se preocupar com a relação entre esses elementos e os possíveis vínculos que antes eram necessários com outras classes e agora podem ou não ser necessários no novo contexto.

Análise da Reusabilidade da Arquitetura Estruturada em Componentes

A reutilização nesse tipo de arquitetura ocorre em nível de componentes. Desta forma, para se conseguir identificar qual componente reutilizar, pode-se procurar saber, em sua descrição, se ele resolve as tarefas a serem reutilizadas ou não, ou então, procurar, na documentação, se há a relação entre os componentes e os padrões da linguagem de padrões. Após a escolha dos componentes a serem reutilizados, deve-se observar, em sua documentação, a sua dependência com outros componentes, para que essas dependências sejam sanadas no novo sistema.

Os componentes que devem ser aproveitados para possibilitar a reutilização do padrão 1 são Componente Base e Classificação (Figura 4.8). Já os componentes que possibilitam a reutilização dos padrões 11, 12 e 13 são Pagamento, Recursos Envolvidos na Transação e Executor da Transação (Figura 4.8).

O problema nesse tipo de reutilização é a aquisição de algumas partes que não eram necessárias, como, por exemplo, no Componente Base, em que se reutiliza parte do padrão 1, mas também o Gerenciamento de Clientes e Fornecedores.

Análise da Reusabilidade da Arquitetura Baseada em Componentes

A forma de identificação dos componentes a serem reutilizados e as preocupações sobre suas restrições são as mesma descritas para a Arquitetura Estruturada em Componentes.

O componente dessa arquitetura que possibilita a reutilização da forma como o padrão 1 é implementado é o denominado Recurso (Figura 4.19). Já a reutilização dos padrões 11, 12 e 13 é propiciada pelos componentes Pagamento de Transação, Empresa, Recursos Envolvidos na Transação e Pagamento de Comissão (Figura 4.19).

Os problemas encontrados na reutilização da Arquitetura Estruturada em Componentes poderiam ser os mesmos encontrados no tipo de arquitetura em questão. Entretanto, nos cenários escolhidos, nenhum problema ocorreu.

5.1.5 – 5° Passo: Identificação dos Pontos de Susceptibilidade

Para a identificação dos pontos de susceptibilidade, a literatura especializada sugere que sejam realizadas pequenas modificações arquiteturais e, a cada modificação, o 4° passo seja refeito. Uma comparação com os resultados obtidos deve ser efetuada e qualquer mudança significativa nos atributos é um indicativo de sua sensibilidade à alteração arquitetural.

Para a realização deste estudo de caso, duas modificações no sistema de Vídeo Locadora foram feitas. A primeira foi a inserção da possibilidade de realizar compra e venda de fitas e a segunda foi possibilitar, além da compra e venda de fitas, a realização de sua manutenção. Desta forma, os sistemas modelados em cada uma das arquiteturas são:

- SL: **S**istema para **L**ocação (Sistema de Vídeo Locadora, Seção 4.2)
- SLC: **S**istema para **L**ocação e **C**omercialização de compra e venda
- SLCM: **S**istema para **L**ocação, **C**omercialização de compra e venda e **M**anutenção

Análise dos Pontos de Susceptibilidade

A seguir são discutidos os pontos de susceptibilidade de cada uma das arquiteturas, apresentando os resultados obtidos com cada um dos sistemas (SL, SLC, SLCM).

Análise da Arquitetura Original

O número de elementos da arquitetura para a instanciação da parte de aplicação do sistema SL, que é de 27 (vinte sete classes) (Figura 4.6), aumentou para 36 (trinta e seis) no sistema SLC e para 45 (quarenta e cinco) no sistema SLCM.

À medida que se gerencia um conjunto único de classes cada vez maior, a atenção à influência de uma determinada classe sobre as que com ela se relacionam deve aumentar, pois, por exemplo, um erro gerado em uma classe pode afetar as outras classes às quais ela se relaciona e esse erro pode ser propagado para diversas classes. E isso afeta a manutenibilidade do sistema.

A manutenção de evolução, além de ser prejudicada pelo que foi descrito anteriormente, também torna-se difícil de ser realizada pois há que se trabalhar com todos os elementos da arquitetura que estão sendo utilizados na modelagem do sistema, por exemplo, para a inserção do acesso restrito a cada uma dos tipos de objetos do sistema.

A possibilidade de reutilização da arquitetura original em qualquer um dos sistemas é praticamente a mesma. Tal fato pode não se confirmar devido a um aumento das classes relacionadas com o que se quer reutilizar, já que deve-se preocupar com a relação entre esses elementos e os possíveis vínculos com outras classes, que antes eram necessários e agora podem ou não ser pertinentes ao novo contexto.

Análise da Arquitetura Estruturada em Componentes

Para possibilitar a instanciação do sistema SL, foram necessários 10 (dez) (Figura 4.11)dos 15 (quinze) componentes da arquitetura em questão (Figura 4.8). Ao instanciar o sistema SLC, utilizou-se mais um componente da arquitetura, que é relacionado com outros dois que já estavam sendo usados. O mesmo ocorreu com o sistema SLCM. O acréscimo de novos elementos e relacionamentos pouco alterou a integridade entre os elementos das arquiteturas utilizados na instanciação dos sistemas, já que quase toda a arquitetura já estava sendo utilizada no sistema SL.

No sistema SL, houve a necessidade de acrescentar o requisito devolução. Para isso realizaram-se algumas modificações em dois componentes, entretanto, não há que se ter a preocupação se tal modificação afetará os outros componentes. Tal fato se justifica pela inalterabilidade das interfaces já existentes. Se houver a necessidade de modificar um

componente qualquer, deve-se preocupar em não alterar os métodos existentes na interface, evitando-se assim a necessidade de revisar os componentes que utilizam tais métodos, o que aumentaria a complexidade. Sem alteração dos métodos, o acréscimo de elementos nessa arquitetura pouco afetaria qualquer tipo de manutenção.

Considerando o que foi descrito anteriormente, as alterações propiciadas pelo aumento do número de componentes no atributo de manutenibilidade, desta arquitetura, podem ser divididas em três casos:

Melhor Caso: As restrições de uso do componente a ser inserido são satisfeitas por algum(ns) (dos) componente(s) já envolvido(s) na instanciação do sistema. Nesse caso, a manutenibilidade seria prejudicada apenas um pouco, já que a alteração seria no número de interfaces a serem gerenciadas.

Caso Intermediário: As restrições de uso do componente a ser inserido não são satisfeitas por nenhum dos componentes já envolvidos na instanciação do sistema e por nenhum outro que possa vir a satisfazê-las. Nesse caso, a manutenibilidade seria prejudicada medianamente, visto que algum(ns) componente(s) já pertencente(s) à arquitetura teria(m) que ser reestruturado(s) tanto internamente quanto externamente para satisfazer as necessidades do novo requisito; considerando-se que os outros serviços fornecidos pelo(s) componente(s) alterado(s) continuassem inalterados.

Pior Caso: Seria o caso intermediário considerando-se que os outros serviços fornecidos pelo(s) componente(s) alterado(s) não continuassem inalterados, podendo gerar uma propagação de alterações.

A reutilização nesse tipo de arquitetura é independente do número de elementos, já que cada um é desenvolvido para realizar um conjunto de tarefas relacionadas e fornece uma interface que explicita suas dependências e funcionalidade. Um dos problemas na reutilização desse tipo de elemento é a aquisição de funcionalidades não desejáveis.

Análise da Arquitetura Baseada em Componentes

Para possibilitar a instanciação do sistema SL, foram necessários 7 (sete) (Figura 4.21) dos 14 (quatorze) componentes da arquitetura em questão (Figura 4.19). Ao instanciar o sistema SLC, utilizou-se mais um componente da arquitetura, que é relacionado com outros três que já estavam sendo usados. Já com o sistema SLCM, houve o acréscimo de mais três componentes e cinco relacionamentos.

A implicação do aumento do número de elementos nos requisitos não-funcionais analisados é a mesma descrita na análise da Arquitetura Estruturada em Componentes.

5.1.6 – 6° Passo: Identificação dos Pontos de Compromisso

Realizado o 5° passo (Seção 5.1.5), descobriu-se que um ponto de compromisso arquitetural para a Arquitetura Original é o número de elementos de uma arquitetura.

O acréscimo do número de classes na Arquitetura Original prejudica a realização de manutenção. Dependendo do número de classes a serem inseridas, esse requisito não funcional pode chegar a níveis impraticáveis e interfere, também, no requisito reusabilidade, apesar de que, dificilmente, será tão prejudicado como o de manutenibilidade.

O relatório elaborado ao fim deste passo, baseado nas informações coletadas com a aplicação do método, detalhando os resultados obtidos junto com a apresentação de estratégias a serem adotadas, é apresentado na seção seguinte (Seção 5.1.7).

5.1.7 – Análise dos Resultados Obtidos

Com o ATAM realizado por completo verificaram-se várias afirmações/suposições da literatura especializada que, até então, não haviam sido exemplificadas, ao menos, ao conhecimento do autor. Algumas das afirmações/suposições são: Um *framework* caixa branca é muito mais flexível que uma caixa cinza ou preta; a utilização de componentes no desenvolvimento de um sistema acarreta uma aceitação da decisão de projeto tomada no desenvolvimento do componente.

Essa constatação se baseou nos artifícios que tiveram que ser tomados para que, por exemplo, a Arquitetura Estrutura em Componentes (Figura 4.8) pudesse dá suporte aos requisitos desejados. Exemplo: O requisito Obter Locação Por Cliente não era apoiado por tal arquitetura, foi necessário então criar a nova operação Obter Recurso Por Cliente(in IdDestino, out Recursos Participantes por Cliente) na interface IRecursos_Envolvidos_Transação do componente Recursos Envolvidos na Transação e que em seu código ela "chamasse" os métodos: (i) Encontrar Participante_Transação, (ii) Recursos_Transação e (iii) Recursos_Participantes; sendo que o método (i) pertence à interface ITransação do Componente Base e os outros dois métodos à mesma interface do método criado. Isso foi necessário porque, ao se projetar a arquitetura, concluiu-se que quem possuía a relação de recursos locados era a transação e não o cliente.

Com o ATAM também foram obtidas respostas para algumas das questões levantadas no início deste trabalho, questões essas que foram a razão do desenvolvimento deste trabalho: Quais são as efetivas vantagens e desvantagens de utilizar componentes no desenvolvimento da arquitetura de um *framework* OO? Como os componentes podem ajudar na solução dos problemas encontrados nos *frameworks*?

Ao se utilizar a Arquitetura Original (Seção 4.3.3), constatou-se sua flexibilidade ao possibilitar exatamente o que é solicitado pela lista de requisitos, apesar de o "como" realizar essas solicitações não ser muito trivial. A dificuldade de "como" realizar as solicitações ocorre porque as tarefas fornecidas por cada elemento arquitetural são muito restritas, havendo a necessidade de utilizar, muitas vezes, um conjunto desses elementos para realizar um requisito específico. Averiguou-se que a reutilização de certas funcionalidades pertencentes a essa arquitetura envolve um número considerável de elementos apesar de tal arquitetura não adquirir funcionalidades não desejáveis; além de se constatar a dificuldade em

se realizar a manutenção, já que, para tal, um conjunto considerado de elementos tem que ser compreendido, pois os mesmo encapsulam apenas um conjunto restrito de tarefas e, geralmente, os requisitos são satisfeitos envolvendo vários elementos.

O que acaba de ser relatado é a concretização do que a literatura especializada considera serem os problemas encontrados nos *frameworks* OO (Seção 2.1). Essa mesma literatura apresenta várias sugestões para minimizar esses problemas e o próximo relato apresenta a concretização de uma dessas sugestões, que é a de utilizar componentes na arquitetura do *framework* OO (Seções 2.2 e 2.5.1).

Com a inserção de componentes na arquitetura do *framework* OO, o "como" fazer as solicitações da análise de requisitos ficou mais simples, já que os componentes incorporam um conjunto de tarefas relacionadas em um único bloco e as disponibilizam através de interfaces. As mesmas interfaces possibilitaram uma reutilização mais fácil, visto que nelas está explícito o que o componente precisa de entrada e o que ele gera de saída, não havendo a necessidade de entender outros componentes. Entretanto, a aquisição de funcionalidades não desejáveis é mais eminente, já que, ao se utilizar um componente, há o compromisso de aceitar a forma como ele foi desenvolvido para possibilitar a realização de suas interfaces e essa forma pode ser conflitante com outro componente da arquitetura. Há que se mencionar ainda a diminuição da flexibilidade antes existente, devido às decisões de análise/projeto tomadas no desenvolvimento do componente, como descrito anteriormente.

Os mesmos motivos que facilitaram o "como" fazer e a reutilização são também os que possibilitaram uma manutenibilidade menos complexa. Porém, a existência de uma interface a ser seguida se torna um empecilho quando esta não fornece as funcionalidades da forma como se deseja, podendo tornar a manutenção de uma arquitetura constituída por componentes semelhante a uma constituída por classes, no que se refere a entender um conjunto de classes para conseguir extrair dessas uma nova funcionalidade. Ainda assim, a manutenibilidade em uma arquitetura constituída por componentes é melhor, já que a alteração no conjunto dos elementos que constituem um componente não afeta os outros componentes, desde que as interfaces antes existentes continuem as mesmas. Mas, se houver alteração das interfaces existentes, a manutenibilidade pode se tornar tão ruim quanto a de uma arquitetura constituída por classes, já que a alteração de uma interface obriga uma alteração dos outros elementos que a utilizam.

Tais conclusões estão considerando que o código dos componentes são abertos, da mesma forma que o código da arquitetura constituída por classes. Caso essa consideração não seja feita, a adaptação de um componente pode ser extremamente difícil. Não foram considerados os componentes caixa preta nesta comparação entre as arquiteturas, pois a forma mais apropriada de se conseguir a adaptação de um componente caixa preta é ainda fonte de discussão entre os pesquisadores da área.

A Tabela 5.1 apresenta um resumo dos resultados obtidos com a realização do ATAM.

		Arquitetura Original (AO)	**Arquitetura Estruturada em Componentes (AEC)**	**Arquitetura Baseada em Componentes (ABC)**
Manutenibilidade	**RM1 – Alteração do número de elementos (Seção 5.1.2)**	AO $\geq\approx$ AEC ≈ ABC	AO $\geq\approx$ AEC ≈ ABC	AO $\geq\approx$ AEC ≈ ABC
	RM2 – Elementos específicos para um conjunto de tarefas (Seção 5.1.2)	Encapsula um conjunto restrito de tarefas	Encapsula um conjunto maior de tarefas ou tão restrito quanto ao propiciado pela AO	Encapsula um conjunto maior de tarefas ou tão restrito quanto ao propiciado pela AO
	RM3 – Número de elementos a serem entendidos (Seção 5.1.2)	AO $\geq\approx$ AEC ≈ ABC	AO $\geq\approx$ AEC ≈ ABC	AO $\geq\approx$ AEC ≈ ABC
Reusabilidade	**RR1 – Aquisição de requisitos não desejáveis (Seção 5.1.2)**	AO < AEC ≈ ABC	AO < AEC ≈ ABC	AO < AEC ≈ ABC
	RR2 – Reutilização completa ou parcial de um padrão (Seção 5.1.2)	Permite, apesar de não ser trivial	Permite, apesar da possibilidade da aquisição de requisitos não desejáveis	Permite, apesar da possibilidade da aquisição de requisitos não desejáveis
Arquitetura	**Flexibilidade**	AO > AEC ≈ ABC	AO > AEC ≈ ABC, devido a decisões de projeto tomadas no desenvolvimento de um componente	AO > AEC ≈ ABC, devido a decisões de projeto tomadas no desenvolvimento de um componente
	Facilidade de Manutenibilidade	AO $\approx\leq$ AEC ≈ ABC	AO $\approx\leq$ AEC ≈ ABC	AO $\approx\leq$ AEC ≈ ABC
	Facilidade de Reusabilidade	AO $\approx\leq$ AEC ≈ ABC	AO $\approx\leq$ AEC ≈ ABC	AO $\approx\leq$ AEC ≈ ABC

Tabela 5.1 - Resumo dos Resultados Obtidos com a Realização do ATAM

5.2 – Aplicação do Método de Análise Quantitativa

Como os estudos para realizar a análise quantitativa de uma arquitetura estão em um estágio formativo e as métricas OO ainda são fontes de estudo, não existindo métricas para medir diretamente os requisitos não-funcionais Complexidade, Manutenibilidade e

Reusabilidade (Seção 3.4.2), tal análise, realizada nesta dissertação, basear-se-á na independência funcional dos elementos das arquiteturas, já que a independência é um dos fatores que influenciam nos requisitos não-funcionais a serem medidos (Pressman, 2001; Mendes, 2002, Chidamber *et. al.*, 1994).

A independência funcional, segundo Pressman (Pressman, 2001), é medida através do Acoplamento, que mede a interdependência entre os módulos (elementos arquiteturais) e da Coesão, que mede da robustez funcional relativa a um módulo (elemento arquitetural). Para realizar essa medição será utilizado o método de análise quantitativa (Seção 3.4.2).

Na realização do referido método, os elementos arquiteturais avaliados serão os mesmos elementos necessários para a instanciação do sistema de Vídeo Locadora em cada uma das arquiteturas anteriormente descritas (Seções 4.3, 4.4 e 4.5).

As Seções 5.2.1 e 5.2.2 apresentam, respectivamente, os resultados obtidos com a quantificação do acoplamento arquitetural e com a quantificação da coesão arquitetural. As implicações dos resultados obtidos são abordadas na Seção 5.2.3.

5.2.1 – Análise Quantitativa do Acoplamento das Arquiteturas

Como proposto na Seção 3.4.2, a métrica Fator de Acoplamento com Componentes (FAC) foi aplicada aos elementos das arquiteturas envolvidos na instanciação do sistema em questão (Seção 5.2) obtendo-se os seguintes resultados:

(1) FAC = 17,66%, quando aplicado aos elementos da Arquitetura Original envolvidos na instanciação do sistema de Vídeo Locadora (Seção 4.3.3);

(2) FAC = 7,08%, quando aplicado aos elementos da Arquitetura Estruturada em Componentes envolvidos na instanciação do sistema de Vídeo Locadora (Seção 4.4.2)

(3) FAC = 3,86%, quando aplicado aos elementos da Arquitetura Baseada em Componentes envolvidos na instanciação do sistema de Vídeo Locadora (Seção 4.5.2)

Com os resultado obtidos, pode-se concluir, em primeira instância, que os elementos da Arquitetura Baseada em Componentes são praticamente 100% menos acoplados que os elementos da Arquitetura Estruturada em Componentes e estes, por sua vez, são aproximadamente 100% menos acoplados que os elementos da Arquitetura Original. Pode-se pressupor então, levando em consideração o fator de acoplamento dos elementos, que os requisitos de Complexidade, Manutenibilidade e Reusabilidade são melhores obtidos em ordem decrescente na Arquitetura Baseada em Componentes, Arquitetura Estruturada em Componentes e Arquitetura Original, já que quanto menor o valor do FAC melhor é o acoplamento entre os elementos de uma arquitetura.

É válido ressaltar que a discrepância dos valores obtidos entre as arquiteturas constituídas por componentes é devido as arquiteturas serem resultados de métodos distintos, o que ocasiona nos componentes uma distribuição das funcionalidades do domínio diferente e isto influencia na necessidade de um maior acoplamento entre os componentes ou não.

5.2.2 – Análise Quantitativa da Coesão dos Elementos das Arquiteturas

A quantificação da coesão arquitetural está fundamentada na divisão de módulos descrita por Pressman (Pressman, 2001), Seção 3.4.2. Dessa forma, como é proposto pelo método de análise quantitativa (Seção 3.4.2), cada elemento necessário para a instanciação do sistema de Vídeo Locadora foi categorizado em (i) Fracamente Coeso; (ii) Nível Moderado de Coesão; e (iii) Altamente Coeso, obtendo-se os seguintes resultados:

(1) Dos 27 (vinte e sete) elementos envolvidos na instanciação da Arquitetura Original para o sistema de Vídeo Locadora (Figura 4.6), 1 (um), o BusisnessResourceTransaction_7, foi considerado Fracamente Coeso e os outros 26 (vinte e seis) foram classificados como Altamente Coesos;

(2) Na Arquitetura Estruturada em Componentes, os elementos envolvidos na instanciação do sistema de Vídeo Locadora (Figura 4.11) foram assim categorizados: 1 (um), o Componente Base, foi considerado Fracamente Coeso; 3 (três), sendo eles os componentes Locação/Reserva de Recurso,

Política de Execução e Pagamento, de Nível Moderado de Coesão e 6 (seis) Altamente Coesos, que são os outros elemento envolvidos na instanciação da arquitetura para o sistema de Vídeo Locadora.;

(3) O resultado da categorização dos elementos da Arquitetura Baseada em Componentes utilizados na instanciação do sistema de Vídeo Locadora (Figura 4.21) é: 1 (um), o componente Empresa, Fracamente Coeso; 2 (dois), sendo eles os componentes Locar/Reservar Recurso e Quantificação do Recurso, de Nível Moderado de Coesão e 4 (quatro) Altamente Coesos, que são os outros elemento envolvidos na instanciação da arquitetura para o sistema de Vídeo Locadora.

Observando os resultados obtidos, pode-se pensar que a Arquitetura Original disponibiliza melhor os requisitos não-funcionais deste experimento do que as outras duas arquiteturas, e estas são praticamente iguais. Tal conclusão está baseada no impacto causado pelos elementos Fracamente Coesos de cada uma das arquiteturas, já que proporcionalmente o elemento Fracamente Coeso da Arquitetura Original tem uma significância bem menor na arquitetura que os outros componentes Fracamente Coesos das outras arquiteturas. Há que se falar ainda na existência de componentes de Nível Moderado de Coesão nas outras arquiteturas, que, apesar de serem quase tão "bons" quanto os Altamente Coesos, perdem um pouco da facilidade de se conseguir o que os requisitos funcionais em questão propõem.

5.2.3 – Análise dos Resultados Obtidos

Apesar dos resultados obtidos serem válidos (Tabela 5.2), deve-se tomar cuidado com as conclusões tomadas (Seções 5.2.1 e 5.2.2), já que as três arquiteturas analisadas são decorrência de três métodos de desenvolvimento diferentes e ainda, a Arquitetura Original utiliza uma tecnologia unicamente baseada no paradigma orientado a objetos e as outras duas, usam a tecnologia do paradigma orientado a objetos associado com a de componentes de software.

		Arquitetura Original (AO)	**Arquitetura Estruturada em Componentes (AEC)**	**Arquitetura Baseada em Componentes (ABC)**
Acoplamento	**FAC**	17,66%	7,08%	3,86%
Coesão	**Número de elementos fracamente coesos**	1	1	1
	Número de elementos com coesão intermediária	0	3	2
	Número de elementos altamente coesos	26	6	4

Tabela 5.2 – Resultados Obtidos nas Arquiteturas Utilizadas no Experimento Instanciadas para o Sistema de Vídeo Locadora

Os resultados obtidos são válidos para a avaliação de uma arquitetura específica e por isso não são válidos como parâmetros de comparação entre arquiteturas distintas. Como as arquiteturas utilizadas no experimento são resultados de métodos diferentes, a comparação entre os valores obtidos das três arquiteturas, a partir do método de análise quantitativa (Seção 3.4.2), não é válida. Isso ocorre porque cada uma, em sua linha de origem, pode obter resultados tão bons quanto uma outra, significando que, para as linhas de desenvolvimento às quais pertencem as arquiteturas são ótimas para a obtenção dos requisitos não-funcionais em questão.

A seguir são citadas algumas diferenças entre essas três linhas de desenvolvimento de arquitetura, que influenciam nos resultados obtidos:

(1) A coesão é robustez funcional relativa a um elemento, mas a robustez proporcionada pela linha de desenvolvimento da Arquitetura Original é menor que a proporcionada pelas outras arquiteturas, sendo assim mais fácil de conseguir a coesão desejada de cada elemento. Entretanto, o número de acoplamentos entre um certo grupo de elementos desta arquitetura tem que ser maior, para se fornecer as funcionalidades exigidas pelo sistema, fato que não ocorre nas outras arquiteturas.

(2) A forma em que as operações fornecidas são distribuídas dentro das interfaces de cada componente interfere no resultado da métrica FAC (Seção, 3.4.2), e como as duas arquiteturas constituídas por componentes são decorrência de métodos distintos, a forma de criação de interfaces é diferente.

A arquitetura original pode apresentar resultados tão bons ou até melhores que as arquiteturas constituídas por componentes. No entanto, os requisitos não-funcionais de Complexidade, Manutenibilidade e Reusabilidade não serão tão facilmente incorporados quanto nestas últimas arquiteturas. Veja o exemplo:

Realizou-se a inserção de novas funcionalidades em cada uma das arquiteturas e depois aplicou-se a métrica FAC novamente. Essas novas funcionalidades foram necessárias para a realização dos sistemas SLC e SLCM (Seção 5.1.1). O resultado obtido foi que as arquiteturas constituídas por componentes continuaram com o valor praticamente inalterado, já a Arquitetura Original teve o valor da métrica FAC diminuído aproximadamente 2% a cada conjunto de funcionalidades inserido. Essa diminuição ocorreu devido às classes que foram inseridas serem acopladas em um conjunto restrito de classes e totalmente independentes das outras. Pode-se concluir, então que, dependendo das classes a serem inseridas na Arquitetura Original, o resultado da métrica FAC pode chegar a valores tão bons quanto aos das Arquiteturas Constituídas por Componentes.

Verifica-se, entretanto, que a diminuição do acoplamento da Arquitetura Original não implica em uma Manutenibilidade mais fácil, visto que, à medida que se tem que gerenciar um conjunto único de classes cada vez maior, a atenção da influência de uma determinada classe sobre as que com ela se relacionam deve aumentar (Seção 5.1.5), o que prejudica um pouco também a Reusabilidade (Seção 5.1.5). Pelo mesmo motivo, a complexidade dos sistemas foi aumentando gradativamente porque, a cada conjunto de novas classes, a necessidade de criar subclasses para permitir a modelagem do que é pedido na análise de requisitos aumenta, e, por conseqüência, um novo conjunto de métodos deve ser criado e/ou redefinido.

Os acoplamentos dos elementos das Arquiteturas Constituídas também não são o único fator que influencia nos requisitos não-funcionais estudados. Apesar do resultado da métrica FAC ter permanecido praticamente inalterado nas duas arquiteturas, a complexidade do sistema aumentou, e as alterações também podem ser divididas em três casos: Melhor Caso, Caso Intermediário e Pior Caso (Seção 5.1.5).

Com a realização da análise quantitativa das arquiteturas, pode-se constatar que a coesão de cada um dos elementos e o acoplamento das três arquiteturas encontram-se bem próximos do ótimo. Constata-se também que tais formas de medida não são os únicos fatores que influenciam a complexidade, a manutenibilidade e a reusabilidade de uma arquitetura de *framework*, já que a obtenção desses requisitos está atrelada a outras características, como o número de elementos, o nível de abstração desses, a quantidade de funcionalidades fornecidas por cada um e como elas são fornecidas.

A Tabela 5.3 apresenta um resumo dos resultados obtidos com a realização do método de análise quantitativa.

	Arquitetura Original (AO)	**Arquitetura Estruturada em Componentes (AEC)**	**Arquitetura Baseada em Componentes (ABC)**
Robustez de cada elemento	AO ≈< AEC ≈ ABC	AO ≈< AEC ≈ ABC	AO ≈< AEC ≈ ABC
Número de elementos para realizar uma funcionalidade	AO >≈ AEC ≈ ABC	AO >≈ AEC ≈ ABC	AO >≈ AEC ≈ ABC
Coesão dos elementos	Próximo do Ótimo	Próximo do Ótimo	Próximo do Ótimo
Acoplamento da Arquitetura	Próximo do Ótimo	Próximo do Ótimo	Próximo do Ótimo
Complexidade	Aumentou apesar do FAC ter diminuído com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC	Aumenfou apesar do FAC ter permanecido praticamente inalterado com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC	Aumentou apesar do FAC ter permanecido praticamente inalterado com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC
Dificuldade Manutenibilidade	Aumentou apesar do FAC ter diminuído com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC	Aumentou apesar do FAC ter permanecido praticamente inalterado com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC	Aumentou apesar do FAC ter permanecido praticamente inalterado com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC
Dificuldade Reusabilidade	Aumentou apesar do FAC ter diminuído com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC	Não modificou, da mesma forma que o FAC permaneceu praticamente inalterado com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC	Não modificou, da mesma forma que o FAC permaneceu praticamente inalterado com a inserção de novas funcionalidades na instanciação da arquitetura AO >≈ AEC ≈ ABC

Tabela 5.3 - Resumo dos Resultados Obtidos com a Realização do Método de Análise Quantitativa

5.3 – Considerações Finais

Com o experimento realizado por completo (Capítulos 4 e 5) pode-se concluir a partir dos resultados obtidos, que a utilização de componentes na arquitetura do *framework* OO ajuda em vários problemas encontrados no desenvolvimento dos *frameworks* OO (Seção 2.1.2):

Problema de Desenvolvimento e Manutenção:

(1) Pode-se pensar em um nível de abstração acima de classes, encapsulando um conjunto maior de tarefas em um único bloco (Componente).

(2) Cada um desses componentes pode ser desenvolvido isoladamente, havendo apenas a preocupação em fornecer uma interface idêntica à que um outro componente necessite (Independência entre os elementos de uma mesma arquitetura).

(3) Como os componentes são desenvolvidos separadamente, cada um é constituído por sua própria documentação.

Problema de Uso:

(1) Ítens 1 e 3 do problema de desenvolvimento

(2) Cada componente já fornece as interfaces que possibilitam a utilização de suas funcionalidades, sem haver a necessidade de realizar hierarquização de classes e sobreposição de métodos.

Problema de Integração:

(1) As interfaces dos componentes fornecem os "limites" das suas funcionalidades e, por conseguinte, os "limites" das funcionalidades do *framework* OO.

(2) Cada componente já fornece as interfaces que possibilitam a utilização de suas funcionalidades, sem haver a necessidade de realizar hierarquização de classes e sobreposição de métodos.

(3) Ítens 2 e 3 do problema de desenvolvimento

(4) Rigidez de como fornecer as funcionalidades do *framework*, obtida através das interfaces dos componentes (padronização das arquiteturas).

Reusabilidade:

(1) Ítens 2 e 3 do problema de desenvolvimento

(2) Item 2 do problema de uso

Já as desvantagens de se utilizarem componentes em uma arquitetura de *framework* OO estão, em sua maioria, concentradas nas interfaces fornecidas pelos componentes e nos problemas existentes neles, visto que:

(1) Ao desenvolver-se um componente, decisões de análise/projeto são tomadas, enrijecendo a forma como os dados são trabalhados e fornecidos.

(2) A não existência de uma interface adequada para a funcionalidade desejada pode gerar uma correção do componente; ação que pode ser tão difícil quanto a correção de uma arquitetura constituída por classes, isso se o componente for caixa branca.

(3) Componentes caixa preta não são fáceis de serem alterados, o que pode prejudicar, em muito, a manutenção do *framework*.

(4) Dois ou mais componentes podem ser conflitantes e a forma de gerenciar tal situação não é trivial.

Essa experiência mostra que o uso de componentes na arquitetura de um *framework* OO acarreta muitas vantagens, desde que se tenha um amplo conhecimento do domínio do *framework* OO a ser desenvolvido. A obrigatoriedade de tal nível de conhecimento está associada à necessidade de fornecer o maior número de interfaces para cada um dos componentes, evitando-se as desvantagens encontradas nesse tipo de arquitetura.

Levando-se em consideração o estado da arte em que se encontra a adaptação de componentes e os resultados obtidos com este trabalho, o autor concluiu que o desenvolvimento de *frameworks* OO constituídos por componentes é extremamente vantajoso desde que estes sejam caixa branca, pois assim, no pior dos casos, a utilização desse tipo de *framework* seria igual à utilização de um *framework* baseado em classes.

Desenvolver um *framework* OO constituído por componentes caixa preta só seria interessante a médio e longo prazo, já que, no início, o número de adaptações a serem feitas na arquitetura para apoiar uma instanciação provavelmente seria alto e a realização da adaptação necessária para se conseguir uma instanciação neste tipo de arquitetura é mais difícil que na arquitetura baseada em classes. A médio e longo prazo é vantajoso porque o número de interfaces de cada componente já estaria mais condizente com a totalidade de possíveis instanciações desse domínio.

Em toda esta seção usou-se o termo arquitetura constituída de componentes, não diferenciando a Arquitetura Estruturada em Componentes da Baseada em Componentes, pois chegou-se à conclusão de que a forma da obtenção de uma arquitetura constituída por componentes não interfere nas conclusões apresentadas. Outra conclusão foi que a realização de refinamentos em ambas arquiteturas constituídas de componentes forneceria duas arquiteturas bem semelhantes, visto que, por exemplo, o Componente Base da Arquitetura Estruturada em Componentes (Figura 4.8) poderia ser dividido em vários outros componentes menores e mais coesos, como acontece na Arquitetura Baseada em Componentes (Figura 4.19); e o Componente Recurso desta última arquitetura poderia ser dividido nos componentes Recurso e Classificação, como ocorre na Arquitetura Estruturada em Componentes.

Os resultados apresentados não são comprometidos pelo experimento realizado, visto que, ao realizá-lo, houve a preocupação de utilizar aplicações que requeriam o maior número possível de funcionalidades fornecidas pelas arquiteturas e de constatar se estas arquiteturas foram bem estruturadas ou não. Com isso, as conclusões aqui tomadas são baseadas em

arquiteturas bem estruturadas e que tiveram grande parte de seus elementos arquiteturais envolvidas na instanciação das aplicações.

Entretanto, o leitor deve estar ciente que as conclusões aqui apresentadas são válidas para este experimento específico. Para tirar conclusões definitivas e generalizadas, trabalhos futuros ainda devem ser realizados, como apresentado no Capítulo 6.

Capítulo 6

Conclusões

Esta dissertação apresentou uma análise qualitativa e quantitativa de *frameworks* em nível arquitetural, com a finalidade de investigar a aplicabilidade do paradigma de desenvolvimento baseado em componentes na construção de *frameworks*. Para a realização do experimento desta dissertação foram utilizadas três arquiteturas de *frameworks*, sendo uma constituída apenas por classes e as outras duas constituídas por componentes, das quais uma arquitetura é estruturada por componentes e a outra arquitetura é baseada em componentes.

A análise qualitativa utilizou o método de análise de compromissos da arquitetura (ATAM) a fim de compreender as conseqüências de decisões arquiteturais em relação aos requisitos não-funcionais: Manutenibilidade e Reusabilidade. Já a análise quantitativa utilizou a métrica Fator de Acoplamento com Componentes (FAC), adaptada à métrica Fator de Acoplamento (CF) de Harrison, Counsell e Nithi (Harrison *et. al.*, 1998a; Harrison *et. al.*, 1998b); e a divisão de módulos coesos proposta por Pressman (Pressman, 2001) para poder quantificar, respectivamente, os fatores de acoplamento e coesão das arquiteturas e, assim, analisar indiretamente os requisitos não-funcionais: Complexidade, Manutenibilidade e Reusabilidade a fim de compreender, também, as conseqüências de decisões arquiteturais.

Ao realizar os métodos de análise qualitativa e quantitativa, aumentou-se gradativamente o número de funcionalidades fornecidas pelas arquiteturas e pôde-se compreender quando e porquê o número de funcionalidades fornecidas por cada arquitetura interfere nos requisitos não-funcionais avaliados. A experiência realizada sobre o domínio de gestão de recursos de negócio mostra o impacto da utilização de componentes nos requisitos não-funcionais: Complexidade, Manutenibilidade e Reusabilidade de uma arquitetura de *framework*.

Durante o desenvolvimento deste trabalho, vários resultados foram obtidos, dos quais as principais contribuições são:

(1) Estudo sobre as diferentes técnicas de reutilização sistemática de software e seus efeitos no processo de desenvolvimento de sistemas de software. É apresentada a relação da tecnologia de *frameworks* com a de componentes e linguagens de padrões.

(2) Verificação do auxílio da linguagem de padrões, voltada para o domínio do *framework*, no desenvolvimento e na documentação do sistema, através do experimento prático. Tal linguagem de padrões foi utilizada no desenvolvimento das arquiteturas estruturada e baseada em componentes. Dessa forma, os métodos propostos para obtenção das arquiteturas podem servir como veículo para a utilização e disseminação do desenvolvimento de linguagens de padrões para a criação de *frameworks*.

(3) Solução de alguns problemas encontrados nos *frameworks* desenvolvidos a partir do puro paradigma orientado a objetos, possibilitados pela introdução de componentes na arquitetura do *framework* auxiliando na documentação, no desenvolvimento, na composição, no uso e na manutenção do *framework*, e, por conseguinte, do sistema. Dessa forma, os métodos de análise arquitetural mostraram o quanto os *frameworks* constituídos por componentes podem ajudar nesses problemas, podendo os resultados servirem de incentivo para o seu desenvolvimento.

Como conseqüência da contribuição três pode-se citar algumas sub-contribuições:

(3.1) Proposta de um método para análise de um *framework* já desenvolvido, com o objetivo de extrair as características, os requisitos, os conceitos e as relações entre os requisitos do domínio a que se aplica. Aplicação desse método em um *framework* do domínio de gestão de recursos de negócio, verificando a sua aplicabilidade.

(3.2) Proposta de um método de reengenharia para a obtenção de uma arquitetura estruturada em componentes a partir de uma arquitetura baseada em classes. Aplicação desse método na arquitetura do *framework* de gestão de recursos de negócio, verificando sua viabilidade.

(3.3) Estudo do processo de especificação de software baseado em componentes, proposto por Cheesman e Daniels (Cheesman *et. al.*, 2001); e proposta do uso de uma linguagem de padrões para auxiliar sua aplicação. Aplicação desse processo, com o uso de linguagens de padrões para o desenvolvimento de uma arquitetura baseada em componentes para o domínio de gestão de recurso de negócio.

(3.4) Estudo sobre o método de análise de compromissos da arquitetura (ATAM) e sua aplicação sobre três arquiteturas distintas de *framework* do domínio de gestão de recursos de negócio.

(3.5) Proposta de adaptação da métrica Fator de Acoplamento (Harrison *et. al.*, 1998a; Harrison *et. al.*, 1998b) para realizar a quantificação do acoplamento arquitetural de uma arquitetura constituída de componentes. Aplicação dessa métrica nas arquiteturas obtidas durante o experimento.

(3.6) Averiguação quantitativa da influência da coesão e do acoplamento na complexidade, manutenibilidade e reusabilidade nos objetos do experimento.

6.1 – Trabalhos Relacionados

Os trabalhos relacionados ao presente estudo podem ser divididos basicamente em três linhas de pesquisa, sendo que em uma delas encontram-se os trabalhos referentes às soluções propostas para o estado da arte de *frameworks*; em outra, o trabalho comparativo entre

frameworks; e, na última, os trabalhos que unem a tecnologia de *frameworks* com componentes.

Dentre os trabalhos recentes dedicados aos problemas encontrados nos *frameworks*, destacam-se os de Moutinho (Moutinho *et. al.*, 2000), que propõe uma solução para o problema de inversão de controle através do estudo de técnicas de composição de *frameworks* e integração dos fluxos de controle embutidos para a construção de um terceiro *framework*; e o de Fontoura (Fontoura, 1999), que apresenta uma extensão da linguagem UML (Booch *et. al.*, 1997), denominada UML-F, para documentação de *frameworks*. A UML-F utiliza um mecanismo de extensão básico da UML e define novos elementos para modelar os aspectos relevantes de projeto do *framework*, como a documentação de *hot-spots*.

Os trabalhos da linha de pesquisa apresentada acima tratam de um problema específico encontrado nos *frameworks*, propondo soluções a partir de uma adaptação do que já é utilizado para o desenvolvimento de frameworks, como o de Fontoura (Fontoura, 1999) ou baseados em um estudo investigativo em busca da melhor técnica para solucionar o problema abordado, como o de Moutinho (Moutinho *et. al.*, 2000). Já o presente estudo realiza uma investigação para averiguar os efeitos da união de duas tecnologias distintas nos problemas dos *frameworks*, não se preocupando em resolver um determinado problema e sim em como essas tecnologias juntas proporcionam uma reutilização mais sistemática e, por conseguinte, como podem auxiliar em alguns dos vários problemas encontrados nos *frameworks*.

A avaliação de *frameworks* é objeto do trabalho de Bansiya (Bansiya, 2000), que apresenta uma metodologia para avaliar as características estruturais e a estabilidade da arquitetura dos *frameworks*, à medida em que evoluem com o desenvolvimento de novas versões; e compara cinco versões do MFC (*Framework Microsoft Foundation Classes*) e quatro versões do OWL (*Framework Borland Object Windows*). As métricas utilizadas por Bansiya abrangem aspectos estáticos da arquitetura, como o número total de classes e de relacionamentos de herança simples e aspectos dinâmicos, como o número de métodos públicos de uma classe e o acoplamento direto dessas classes (Bansiya, 2000).

O trabalho comparativo de Bansiya é realizado baseado em várias versões de um mesmo *framework*, por conseguinte, com as mesmas características e a mesma tecnologia utilizada, com o objetivo de avaliar a evolução de uma arquitetura. Já no presente trabalho, investigam-se arquiteturas de *frameworks* de um mesmo domínio, porém com métodos de criações

distintos, com o objetivo de analisar a aplicabilidade desses métodos na arquitetura de um *framework*.

Por fim, existem os trabalhos que relacionam *frameworks* e componentes, como o de Lubsen (Lubsen, 2000), que propõe uma ferramenta para ajudar na instanciação das camadas do *framework* SanFrancisco baseadas em componentes (Monday *et. al.*, 1999); e o trabalho de Gurp (Gurp *et. al.*, 2000), que apresenta um modelo conceitual para os *frameworks*, incluindo *frameworks* baseados em componentes e um conjunto de diretrizes para construção de *frameworks* OO aderentes a esse modelo.

Os trabalhos desta linha de pesquisa utilizam os componentes na estruturação de um *framework*, como o de Monday (Monday *et. al.*, 1999), e propõem formas de utilizá-los, como o de Lubsen (Lubsen, 2000); e apresentam técnicas para desenvolver um *framework* constituído por componentes. No entanto, nenhum deles averigua quais são os efetivos ganhos e perdas de se utilizar tais tecnologias em conjunto.

O que o presente estudo faz é realizar um estudo investigativo como o de Moutinho, para obter uma comparação entre diferentes arquiteturas, semelhante ao trabalho de Bansiya, com o intuito de saber o que a união de componentes e *frameworks*, como realizada nos trabalhos de Gurp e Monday, afeta no estado da arte de *frameworks*.

6.2 – Dificuldades Encontradas

As principais dificuldades na realização deste trabalho estão relacionadas ao como proceder para realizar uma análise em nível de abstração tão alto; quais métodos utilizar para realizar esta análise e experimento prático. A seguir, serão descritas as principais dificuldades encontradas:

(1) Como realizar a análise arquitetural, já que cada arquitetura é o resultado de métodos distintos e possuem características diferentes. A análise se limitou aos requisitos não-funcionais Complexidade, Manutenibilidade e Reusabilidade pela inviabilidade de analisar um conjunto maior de requisitos e por o autor os considerar os requisitos mais importantes de uma arquitetura

de *framework*, visto que a obtenção desses, interfere diretamente em outros requisitos não-funcionais, como na usabilidade e na corretude.

(2) A falta de um método para analisar quantitativamente um *framework* a nível arquitetural, levando o autor a procurar formas alternativas para realizar tal análise.

(3) A falta de métricas que pudessem ser aplicadas tanto em uma arquitetura baseada em classes, quanto em uma constituída por componentes, levando o autor a realizar uma pesquisa sobre as possíveis métricas existentes, que pudessem ser aplicadas ao experimento depois de uma adaptação.

(4) O tamanho e a complexidade do *framework* GREN, e o fato de estar sendo desenvolvido em paralelo à esta dissertação. O GREN é completo para o domínio a que se propõe. Na reestruturação de sua arquitetura, houve a necessidade de limitar o número de classes a serem analisadas devido à inviabilidade de reestruturar todo o *framework*. Assim, o experimento foi realizado sobre as classes da camada de aplicação e persistência. O fato de estar ainda em fase de desenvolvimento, sofrendo várias alterações e não possuir uma documentação consistente acabou provocando várias atualizações nas partes já desenvolvidas desta dissertação, já que o interesse sempre foi apresentar os resultados, o mais próximo possível, condizentes com a atual versão do GREN.

6.3 – Trabalhos Futuros

Algumas linhas de pesquisa podem ser destacadas a partir do trabalho apresentado nesta dissertação:

(1) Uma dessas linhas envolve a continuação dos experimentos para a análise da inserção de componentes na arquitetura de *frameworks* de outros domínios. Podem-se realizar experimentos que comprovem a eficiência dos métodos propostos. Experiências na adição de outros requisitos não-funcionais como segurança e eficiência, também podem ser realizados.

(2) Desenvolvimento de um *Cookbook* para o uso das arquiteturas estruturada e baseada em componentes na construção de sistemas de gestão de recursos de negócio.

(3) Extensão das arquiteturas constituídas por componentes para disponibilizarem a camada de interface do usuário e o GREN – Wizard, desenvolvido para o *framework* GREN.

(4) Implementação das arquiteturas propostas. Realização de uma comparação de campo, analisando os efeitos da utilização de componentes nas arquiteturas de *frameworks* através de *feedbacks*.

Capítulo 7

Bibliografia

(Abaunader *et. al.*, 1997) Abaunader, J.R., Lamb, D.A. *A data Model for Object-Oriented Design Metrics*. Department of Computing and Information Science Queen´s University, Outubro 1997 - URL: http:\\www.qucis.queensu.ca/TechReports/Reports/1997-409.ps, visitado em outubro de 2002.

(Alexander *et. al.*, 1977) Alexander, C., Ishikawa, S., Silverstein, M. *A Pattern Language*. Oxford University Press, New York, 1977.

(Bansiya, 2000) Bansiya J. *Evaluating framework architecture structural stability*. Computing Surveys of the ACM, Março 2000 – V. 32, N° 1es. Electronic Edition (ACM DL).

(Barbacci *et. al.*, 1998) Barbacci, M.R., Carriere, S.J., Feiler, P.H., Kazman, R., Klein, M.H., Lipson, H.F., Longstaff, T.A., Weinstock, C.B. *Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis*. Software Engineering Institute, Carnegie Mellon University, Maio de 1998 - URL: www.sei.cmu.edu/pub/documents/97.reports/pdf/97tr029.pdf, visitado em outubro de 2002.

(Basic, 2002) Microsoft Visual Basic - URL: http://msdn.microsoft.com/vbasic/Default.asp, visitado em outubro de 2002.

(Bass *et. al.*, 1998) Bass, L., Clements, P., Kazman, R. *Software Architecture in Parctice*. Addison Wesley, 1998.

(Booch *et. al.*, 1997) Booch, G., Rumbaugh, J., Jacobson, I. *OMG Unified Modeling Language - Version 1.01*. Rational Software Corporation, 1997

(Booch *et. al.*, 2000) Booch, G., Rumbaugh, J., Jacobson, I. *UML, Guia do Usuário*. Campus, 2000.

(Borland, 2002) Borland Delphi - URL: http://www.borland.com/delphi/, visitado em outubro de 2002.

(Bosch *et. al.*, 2000) Bosch J., Molin P., Mattsson M., Bengtsson P. *Object-oriented framework-based software development: problems and experiences*. Computing Surveys of the ACM, March 2000 – V. 32, N° 1es. Electronic Edition (ACM DL).

(Bosch *et. al.*, 2000) Bosch J., Molin P., Mattsson M., Bengtsson P. *Object-oriented framework-based software development: problems and experiences*. Computing Surveys of the ACM, March 2000 – V. 32, N° 1es. Electronic Edition (ACM DL).

(Box, 1998) Box, D. *Essential COM*. Addison-Wesley, 1998.

(Braga *et. al.*, 1999) Braga, R.T.V., Germano, F.S.R., Masiero, P.C. *A Pattern Language for Business Resource Management*. Anais da 6th Pattern Languages of Programs Conference (PLoP'99), Monticello-IL, EUA, v.7, 1999.

(Braga *et. al.*, 2001) Braga, R.T.V. *Um Framework Baseado em uma Linguagem de Padrões para Sistemas de Gestão de Recursos de Negócios.* ICMC-USP, 2001 - URL: http://www.icmc.sc.usp.br/~rtvb/GRENFramework.html, visitado em maio de 2002.

(Brown *et. al.*, 1996) Brown, A.W., Wallnau, K.C. *Engineering of Component Basead Systens*. Component-Basead Software Engineering, IEEE Computer Society Press, 1996.

(Brown *et. al.*, 1998) Brown, A.W., Wallnau, K.C. *The Current State of CBSE*. IEEE Software, September/October 1998.

(Brugali *et. al.*, 2000) Brugali, D., Menga, G., Aarsten, A., *A Case Study for Flexible Manufacturing Systems*. In Fayad *et. al.*, 2000, 2000.

(Buschmann *et. al.*, 1996) Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *A System of Patterns: Patterns-Oriented Software*. John Wiley & Sons, 1996.

(Cagnin, 2002) Cagnin, M.I. *Investigação e Definição de um Processo de reengenharia Orientada a Objetos Visando ao Apoio por Computados*. ICMC-USP-São Carlos, Proposta de Doutorado, 2002.

(Cheesman *et. al.*, 2001) Cheesman, J., Daniels, J. *A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

(Chidamber *et. al.*, 1994) Chidamber, S.R. and Kemerer, C.F. *A Metrics suite for object-oriented design*. IEEE Trans. Software Enginneering, vol. SE-20, N° 6, Junho 1994.

(Cincom, 2001) CINCOM SYSTEMS, INC. *VisualWorks Non-Commercial 5i.3*. Disponível em junho de 2001 para download na URL: http://www.cincom.com/scripts/smalltalk.exe/downloads/index.asp?content=visualworks

(Coelho, 1998) Coelho, M.G. *Uma Abordagem Reflexiva para a Construção de Frameworks para Interfaces Homem-Computador*. DCC-UNICAMP, Tese de Mestrado, 1998.

(Cohen, 1995) Cohen, N. *Ada as A Second Language*. McGraw-Hill, 1995.

(Coppick *et. al.*, 1992) Coppick, J.C., Chesthan. *Softwarre metrics for object-oriented systems*. In Agrawal, J.P., Kumar, V., Wallentine, V., editor, *Proceeding of the 20th Annual Computer Science Conference*, ACM, March, 1992.

(Etzhorn *et. al.*, 1997) Etzhorn, L., Davis, C., Li, W. *A Statistical Comparison of Various Definitions of the LCOM Metric*. University of Alabama, 1997. URL - http://concept.cs.uah.edu/pub/techreports/TR-UAH-CS-1997-02.ps.Z, visitado em outubro de 2002.

(Ewald, 2001) Ewald, T. *Transactional COM+: Building Scalable Applications*. Addison Wesley Professional, 2001.

(Fayad *et. al.*, 1997) Fayad, M.E., Schmidt, D.C. *Object-Oriented – Aplication Frameworks*. Communications of the ACM, Octuber 1997 – V. 40, N° 10.

(Fayad *et. al.*, 1999) Fayad, M.E., Schimidt, D.C., Johnson, R.E. *Building Application Frameworks*. John Wiley & Sons, 1999.

(Fayad *et. al.*, 2000) Fayad, M.E., & Johnson, R.E. *Domain-Specific Aplication Frameworks: Frameworks Experience by Industry*. John Wiley & Son, 2000.

(Fayad, 2000) Fayad, M.E. *Introduction to the computing surveys' electronic symposium on object-oriented application frameworks*. Computing Surveys of the ACM, March 2000 – V. 32, N° 1es. Electronic Edition (ACM DL).

(Filho, 2001) Filho, W.P.P. *Engenharia de Software: Fundamentos, Métodos e Padrões*. LTC, 2001.

(Fontoura, 1999) Fontoura, M. *A Systematic Approach for Framework Development*. Political Catholic University of Rio de Janeiro, Tese de doutorado, Julho 1999.

(Foote *et. al.*, 1998) Foote, B., Ralph, E.J. *Designing to Facilitate Change With Object-Oriented Frameworks*. University of Illinois at Urbana-Champaign, 1988 – URL: http://www.laputan.org/dfc/discussion.html, visitado em março de 2000.

(Fuentes *et. al.*, 2000) Fuentes L., Troya, J. M. *Towards an open multimedia service framework*. Computing Surveys of the ACM, March 2000 – V. 32, N° 1es. Electronic Edition (ACM DL).

(Gamma *et. al.*, 2000) Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Padrões de Projeto*. Bookman, 2000.

(Garlan, 1995) Garlan, D., Perry, D. *Introduction to the Special Issues on Software Architectural*. IEEE Transaction on Software Engeneering, 1995.

(Gurp *et. al.*, 2001) Gurp, J.V., Bosch, J. *Design, Implementation and Evolution of Object Oriented Frameworks: Concepts and Guidelines.* Software - Practice and Experience, 2001 – URL: http:\\ http://www.xs4all.nl/~jgurp/homepage/publications/ visitado em outubro de 2002.

(Harrison *et. al.*, 1998a) Harrison, R., Counsell, S.J., Nithi, R.V. *An Evaluation of the MOOD set of Object-Oriented Software Metric.* IEEE Trans. Software Engineering, vol. Se-24, n° 6, junho 1998.

(Harrison *et. al.*, 1998b) Harrison, R., Counsell, S.J., Nithi, R.V. *An Investigation into Applicability and Validity of Object-Oriented Design Metrics,* 1998 – URL: http:\\www.cs.rdg.ac.uk/~rh/ESECandK.ps, visitado em outubro 2002.

(Hitz *et. al.*, 1995) Hitz, M., Montazeri, B. *Measuring product attributes of object-oriented systems.* In W. Schfer and P. Botella, editors, Proc. ESEC 95 (5th European Software Engineering Conference), September de 1995.

(Hitz *et. al.*, 1996a) Hitz, M., Montazeri, B. *Chidamber and kemerer´s metrics suite: A measurement theory perspective.* IEEE Transactions on Software Engineering, 1996.

(Hitz *et. al.*, 1996b) Hitz, M., Montazeri, B. *Measuring coupling in object-oriented systems.* 1996. URL – http:\\www.pri.univie.ac.at/~hitz/papers visitado em Janeiro 2002

(Johnson, 1997) Johnson, R.E., CS497 Lectures – Lecture 12, 13, 14, 17, disponível em http://st-www.cs.uiuc.edu/users/johnson/cs497, visitado em fevereiro de 1999.

(Johnson, 1998) Johnson, M. *A Beginner's Guide to Enterprise JavaBeans*. Javaworld, October 1998 - URL: http://www.javaworld.com/javaworld/jw-10-1998/jw-10-beans_p.html, visitado em outubro de 2002.

(Kazman *et. al.*, 1998) Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J. *The Architecture Tradeoff Analysis Method, Software Engineering Institute*. CMU/SEI-98-TR-008, ESC-TR-98-008, 1998.

(Kulesza, 2000) Kulesza, U. *Técnicas de Orientação a Objetos para Projeto de Sistemas Adaptáveis*. IME-USP, Tese de Mestrado, 2000.

(Lewandowski, 1998) Lewandowski, S.M. *Frameworks for Componet-Based Client/Server Computing*. ACM Computing Surveys, March 1998, V. 30. N° 1.

(Li *et. al.*, 1995) Li, W., Henry, S., Kafura, D. *Schulman, R. Measuring Object-Oriented Design*. Journal of Oriented Programming, 1995

(Lubsen, 2000) Lubsen, H.V.E.B. *Business Component Prototyper for SanFrancisco: An experiment in architecture for application development tools*. IBM Systems Journal, vol 39, n° 2, 2000.

(Mattsson *et. al.*, 1999) Mattsson, M., Bosch, J., Fayad, M. E. *Framework Integration – Problems, Causes, Solutions*. Comunications of the ACM, Octuber 1999 – V. 42, N° 10.

(McCabe *et. al.*, 1994) McCabe, T.J., Watson, A.H. *Software Complexity*. Crosstalk, vol 7, n° 12, 1994.

(McCabe, 1976) McCabe, T.J. *A Complexity Measure*. IEEE Transactions on Software Engineering, 1976.

(Mendes, 2002) Mendes, A. *Arquitetura de Software – Desenvolvimento Orientado para Arquitetura*. Editora Campus, 2002.

(Monday *et. al.*, 1999) Monday P., Carey J., Dangler M. *SanFrancisco Component Framework - An Introduction*. Addison-Wesley, 1999.

(Moutinho *et. al.*, 2000) Moutinho, B.M., Melo, A.C.V. *Composição de Fluxo de Controle em Frameworks Java*. Workshop de Teses em Engenharia de Software – SBES'2000, Outubro 2000.

(MYSQL, 2001) MySQL REFERENCE MANUAL FOR VERSION 3.23.39, disponível em junho de 2001 para "download" na URL: http://web.mysql.com/.

(Newman *et. al.*, 1999) Lakshminarayana, A., Newman, T.S. *Principal Component Analysis of Lack of Cohesion in Methods (LCOM) metrics*. Technical Report TR-UAH-CS-1999-01, Computer Science, Dept., Univ. Alabama in Huntsville, 1999.

(Posnak *et. al.*, 1997) Posnak, E. J., Lavender, R.G., Vin, H. M. *An Adaptive Framework for Developing Multimedia Software Components*. Communications of the ACM, v. 40, n. 10, october 1997.

(Pree *et. al.*, 2000) Pree, W., Fontoura, M., Bernhard, R. *UML-F: A Modeling Language for Object-Oriented Framework*. ECOOP'2000, LNCS 1850, Sringer-Verlag, 63-83, 2000.

(Pree, 1997) Pree, W. *Component-Based Software Development - A New Paradigm in Software Engineering*. Germany, 1997 – URL:

http// www.softwareresearch.net/publications/J009.pdf, visitado em outubro de 2002.

(Pressman, 2001) Pressman, R.S. *Software Engineering – A Practitioner's Approach*. Fifth Edition, McGraw Hill, 2001.

(Sun, 2000) Sun - http://java.sun.com - visitado em março de 2000.

(Shaw, 1996) Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Didcipline*. Prentice Hall, 1996

(Staa, 2000) Staa, A.V. *Programação Modular: Desenvolvendo programas complexos de forma organizada e segura*. Editora Campus, 2000.

(Szyperski *et. al.*, 1996) Szyperski, C., and *et. al.*. *Sumary of First International Workshop on Component Oriented Programming*. In First International Workshop on Component Oriented Programming (WCOO'96), June 1996.

(Szyperski *et. al.*, 1997) Szyperski, C., Bosch, J., Weck, W. *Sumary of Second International Workshop on Component Oriented Programming*, In Second International Workshop on Component Oriented Programming (WCOO'97), June 1997.

(Szyperski, 1998) Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

(Weiss, 2001) Weiss, G.M. *Adaptação de Componentes de Software para o Desenvolvimento de Sistemas Confiáveis*. DCC-UNICAMP, Tese de Mestrado, 2001.

(Woodman, 1990) Woodman M. *Portable Modula-2 Programming*. McGraw-Hill International Series in Software Engineering, 1990.

(Ye, 2001) Ye, Y. *Supporting Component-Based Software Development with Active Component Repository*. Department of Computer Science – University of Colorado at Boulder, Tese de Doutorado, 2001.