RODRIGO ALVARES DE SOUZA

UM PROCESSO DE TRANSFORMAÇÃO DE ARQUITETURAS DE SISTEMAS LEGADOS BASEADO EM REENGENHARIA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

São Paulo
2004

UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA

RODRIGO ALVARES DE SOUZA

UM PROCESSO DE TRANSFORMAÇÃO DE ARQUITETURAS DE SISTEMAS LEGADOS BASEADO EM REENGENHARIA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

São Paulo
2004

RODRIGO ALVARES DE SOUZA

UM PROCESSO DE TRANSFORMAÇÃO DE ARQUITETURAS DE SISTEMAS LEGADOS BASEADO EM REENGENHARIA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Reginaldo Arakaki

São Paulo
2004

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 10 de julho de 2004

Rodrigo Alvares de Souza

Reginaldo Arakaki

FICHA CATALOGRÁFICA

Souza, Rodrigo Alvares de
Um processo de transformação de arquiteturas de sistemas legados baseado em reengenharia / Rodrigo Alvares de Souza – São Paulo, 2004.
104 p.

Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Padrões de projeto de software 2. Arquitetura de software 3. Reengenharia I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t

Dedico este trabalho aos meus avós Angel Juanes Sevilla (*in memorian*) e Rosa Chiquetto Sevilla.

RESUMO

Os sistemas de software, a partir do momento em que são entregues para utilização, sofrem inúmeras manutenções no decorrer de sua vida útil. Estas manutenções podem ter características corretivas, onde os erros decorrentes da implementação são corrigidos, ou evolutivas/adaptativas, onde a partir de requisitos de usuários, o sistema deve ser transformado, visando obter melhorias estruturais ou funcionais. Em se tratando de manutenções evolutivas ou adaptativas, a reengenharia de sistemas se torna uma grande aliada, pois não são raras as situações onde a equipe responsável pelo projeto de manutenção não possui conhecimento do sistema que será trabalhado. O objetivo desta dissertação de mestrado é a apresentação de um processo que sirva de orientação para manutenções que exijam transformações de arquitetura, estimulando o uso de padrões de projeto, utilizando técnicas de reengenharia. Também será apresentado um estudo de caso em um sistema que ilustra a aplicação do processo, bem como as dificuldades encontradas na elaboração do trabalho. E, por fim, serão apresentadas as conclusões e sugestões para futuros estudos.

ABSTRACT

From the moment software products are delivered, they go through a series of modifications during their life-cycle. The maintenance process can either be corrective or evolutionary/adaptive. A corrective process is used to fix implementation errors and an evolutionary/adaptive process is used to improve performance and structure or other attributes, or to adapt the product to a changed environment, according to requests from users. When it comes to evolutionary maintenance, software reengineering plays a fundamental role, since working teams responsible for maintenance projects frequently do not have enough knowledge about the system they will be working with. The purpose of this dissertation is to present a process that gives directions for maintenance processes that require software architecture modifications, encouraging the usage of design patterns by the means of reengineering technics. This paper will also present a case study where a software system works as an instructive example of how the process can be used, as well as some difficulties found throughout the elaboration of this work. Finally, this paper will present conclusions and suggestions for future studies in the software systems reengineering field.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CORBA	–	Commom Object Request Broker Architecture
OOPSLA	–	Conference on Object-Oriented Programming Systems, Languages, and Applications
RAD	–	Rapid Application Development
UML	–	Unified Modelling Language

1. INTRODUÇÃO E MOTIVAÇÕES

1.1. OBJETIVOS

Os objetivos deste trabalho de pesquisa estão focados em:

- Apresentar um processo para transformação de arquitetura de sistemas legados(que utilizem tecnologia orientada a objeto), utilizando orientação a objetos e técnicas de reengenharia.
- Aplicar o processo proposto em um estudo de caso, baseado em um sistema de automação de propostas comerciais, desenvolvido em 2 camadas.

1.2. MOTIVAÇÃO E JUSTIFICATIVA

A partir dos anos 90, houve um crescente interesse nas tecnologias e metodologias orientadas a objeto, tanto na área acadêmica como na área comercial, que resultou em uma grande quantidade de produtos de software desenvolvidos de acordo com estes paradigmas.

Porém, grandes promessas feitas pelos entusiastas da orientação a objeto não se cumpriram efetivamente (Sanz, 1995). Reutilização de software, sistemas adaptáveis, flexíveis e fáceis de manter foram e são incansavelmente buscados, mas apenas algumas iniciativas podem ser citadas como sucesso (Miah (1997), Steinman (1999)). Em consequência disso, esses sistemas são invariavelmente submetidos à manutenção e, eventualmente, necessita-se que sejam efetuadas transformações na arquitetura dos mesmos, não apenas com a intenção de reestruturação interna, buscando características como reuso, adaptabilidade e facilidade de manutenção, mas

também para aproveitar oportunidades de negócios ou posicionar melhor a empresa do ponto de vista estratégico.

Para que essa transformação de arquitetura possa ocorrer, a reengenharia de software pode ser utilizada no apoio deste processo. A reengenharia de um sistema consiste em avaliar e alterar um sistema existente, e em seguida reconstruí-lo, completamente ou em partes, pelos seguintes motivos:

- Aumentar a qualidade do sistema existente, mantendo os requisitos funcionais;
- Efetuar manutenções (corretivas, perfectivas ou adaptativas);
- Conforme o sistema é alterado, sua estrutura original tende a desaparecer, em virtude de falta de planejamento e projetos da atividade de manutenção;
- Sincronizar sistemas com sua documentação;

Para a condução e gerenciamento de projetos deste tipo, algumas técnicas podem ser utilizadas como apoio a este processo, como padrões de software.

Padrões de software são soluções para problemas recorrentes, com um nome e um propósito bem definido. São soluções genéricas, mas que podem ser traduzidos para áreas específicas. Appleton (2000) descreve a utilidade dos padrões da seguinte maneira:

"*Um padrão é onde a teoria e a prática se encontram para reforçar e complementar uma à outra, mostrando que a estrutura descrita é útil, utilizável e testada*".

Vários problemas arquiteturais podem ser resolvidos através de padrões; então torna-se natural a tentativa de unir as vantagens proporcionadas pelos padrões

ao trabalho de reengenharia. Existem algumas iniciativas em estudos sobre metodologias baseadas em padrões de reengenharia, como a proposta por Demeyer et al. (1999), mas estas se mostram um tanto complexas, e por serem baseadas em padrões, fornecem a solução genérica para o problema, mas não um processo ou método aplicável, nem a integração com padrões de projeto.

O presente trabalho propõe um processo de transformação de arquiteturas, procurando obter os passos básicos para a realização de projetos desta natureza, baseados em reengenharia, integrando a modelagem do sistema com padrões de projeto.

1.3. METODOLOGIA

Como metodologia de pesquisa, efetuou-se um levantamento bibliográfico de assuntos relacionados ao tema. Nesta fase, foi realizado um levantamento de artigos, teses e publicações que seriam utilizados no desenvolvimento do trabalho de mestrado. Os seguintes assuntos foram pesquisados:

- **Manutenção de Software**: Conceitos, definições e tipos de manutenção;
- **Arquitetura de Software**: Definições, estilos de arquitetura, visões de arquitetura;
- **Reengenharia de Software**: Definições, engenharia reversa, processos e técnicas;

Após a fase de levantamento bibliográfico, foi efetuada uma comparação entre modelos de reengenharia existentes, e coletadas algumas informações a respeito das práticas comuns entre estes modelos.

Com base nestas informações, foi elaborado um processo de reengenharia de sistemas legados, com forte orientação em arquitetura, incentivando a utilização de padrões de projeto.

Em seguida, o processo é aplicado em um estudo de caso, onde são discutidas as vantagens e desvantagens da abordagem.

A conclusão do trabalho será a discussão do resultado obtido utilizando a abordagem proposta, e os caminhos futuros para pesquisa na área.

1.4. ESTRUTURA DA DISSERTAÇÃO

O presente trabalho está organizado da seguinte maneira:

1. **Introdução**

   Neste capítulo são apresentados objetivos, motivações e justificativas da dissertação, e uma síntese do conteúdo da mesma.

2. **Fundamentos Conceituais**

   O capítulo 2 apresenta os fundamentos conceituais, levantados através da pesquisa bibliográfica, englobando manutenção de software, arquitetura de software e reengenharia.

3. **Um Processo de Transformação de Arquiteturas de Sistemas Legados Baseado em Reengenharia**

   Apresentação do processo de transformação de arquiteturas. A abordagem tomada consiste na introdução do processo, seu escopo e a descrição de seus macro-processos e respectivas fases.

4. Estudo de Caso

O estudo de caso da aplicação será o foco do capítulo. Será mostrada uma aplicação real do processo.

5. Considerações Finais

Este capítulo resume as considerações finais do trabalho de pesquisa realizado, e a contribuição do trabalho na área de reengenharia e arquitetura. Em seguida, são abordadas as melhorias que podem ser incorporadas em trabalhos futuros.

Referências Bibliográficas

Parte do documento onde serão apresentadas as referências bibliográficas que fazem parte do presente trabalho de pesquisa.

Apêndice e Glossário

Nesta parte, estão detalhados os padrões de projeto utilizados no estudo de caso e são listados os termos técnicos que estão presentes no trabalho.

2. FUNDAMENTOS CONCEITUAIS

2.1. MANUTENÇÃO DE SOFTWARE

2.1.1. Conceitos

A manutenção de software pode ser definida como a atividade de alteração de um produto de software após o momento em que ele torna-se operacional, ou seja, está sendo utilizado. A partir deste momento, é provável que ele venha a sofrer inúmeras manutenções no decorrer de sua vida útil, seja para correção de defeitos, ocasionados na fase de desenvolvimento, inclusão de novas funcionalidades, ou adaptação para utilização em arquiteturas diferentes das quais o mesmo foi projetado. Em termos econômicos, a manutenção de software pode ser considerada a fase do ciclo de vida do sistema que consome mais esforço e recursos, podendo demandar até mais que 70% do esforço de uma empresa de desenvolvimento (Pressman, 1995, p. 876; Nomura e Spínola, 200-?).

Pressman (1995) define 4 tipos de atividades de manutenção. O primeiro tipo de manutenção é a corretiva, que consiste em localizar e consertar erros de programação. Em seguida, a manutenção adaptativa, que leva em consideração alterações de ambiente computacional, como software básico (*e.g.*, sistemas operacionais, sistemas gerenciadores de banco de dados) ou troca de hardware. Em terceiro, a manutenção perfectiva, que é realizada quando o sistema encontra-se funcionando corretamente, porém novas funcionalidades são introduzidas, visando um aumento de satisfação do usuário. E, por último, a manutenção preventiva, que consiste em alterar o software fazendo com que sua confiabilidade e manutenibilidade sejam melhoradas.

Verhoef (200-?) define as mesmas etapas citadas por Pressman, porém acrescenta uma quinta, denominada manutenção de pré-entrega. Este tipo de manutenção leva em consideração decisões e aspectos arquiteturais definidos na etapa de projeto do sistema, visando melhorar a manutenibilidade e o porte futuro do sistema. É uma abordagem interessante, visto que a manutenção normalmente é considerada como uma tarefa em separado do projeto do sistema (Jones, 1998 apud (Voerhef, 200-?))[1], e a maioria dos currículos acadêmicos de Engenharia de Software são inadequados em se tratando de manutenção (Jones, 1994 apud (Voerhef, 200-?))[2].

2.1.2. Tipos de Manutenção

Neste item, serão detalhados os tipos de manutenção e como eles se encaixam no trabalho proposto.

2.1.2.1. Manutenção Corretiva

A manutenção corretiva trata da atividade reativa de consertar erros de execução que não foram encontrados na atividade de teste. Voerhef (200-?) define este tipo de manutenção como uma necessidade de implantação de mudanças devido aos erros encontrados. Estes tipos de erros podem ser os seguintes:

- Falhas de processamento;
- Saídas não esperadas, erros de cálculo;
- Violação de metodologia de programação, inconsistências;
- Término anormal da aplicação;
- Desempenho abaixo do esperado, e outros.

[1] JONES, C. Estimating Software Costs. McGraw-Hill, 1998.
[2] JONES, C. Assessment and Control of Software Risks. McGraw-Hill, 1994.

Nomura e Spínola (200-?) definem a manutenção corretiva como o processo de diagnóstico e correção de defeitos ocorridos e relatados pelo desenvolvedor. Estes erros seriam decorrentes da própria complexidade do sistema.

2.1.2.2. Manutenção Adaptativa

Trocas de ambiente de hardware, atualizações de sistema operacional ou sistema gerenciador de banco de dados, ou atualização de componentes de software obtidos de terceiros podem levar a incompatibilidades do sistema, necessitando a realização de manutenções. A manutenção adaptativa considera as atividades de traduzir um sistema de software à um novo contexto, ou ambiente, conforme os exemplos citados.

Voerhef (200-?) define dois tipos de manutenção adaptativa. O primeiro tipo é a mudança de dados, cujo exemplo mais conhecido foi o problema do ano 2000, que forçou a adaptação de sistemas baseados em data de dois dígitos, e as mudanças de processamento, que lidam justamente com as trocas de ambientes citados no parágrafo anterior.

2.1.2.3. Manutenção Perfectiva

Ao passar dos anos, novas gerações de hardware são introduzidas no mercado, tornando obsoletas as gerações anteriores. Com o software, a situação ocorre de maneira diferente; um sistema de software pode permanecer por até décadas em operação (Pressman, 1995, p. 878). Em conseqüência disso, é de se esperar que novas funcionalidades sejam agregadas ao sistema, visando aumentar sua utilidade. Dessas atividades, surge a manutenção perfectiva. Seu principal objetivo é obter um avanço qualitativo no sistema em questão. Verhoef (200-?) cita como

atividades pertinentes a este tipo de manutenção o aumento de desempenho, melhoria de algoritmos e de documentação. Canfora e Cimitile (2000) definem este processo como a etapa onde as alterações são solicitadas pelos usuários, como inclusão, exclusão, extensão e modificação de funcionalidades, atualização de documentação e melhorias de desempenho. Chapin et al. (2001) definem de maneira similar, apontando este tipo de manutenção como melhorias de desempenho, aumento de eficiência e melhoria de manutenibilidade.

2.1.2.4. Manutenção Preventiva

A manutenção preventiva lida com os problemas antes que eles ocorram. É definida por Nomura e Spínola (200-?) como sendo uma iniciativa que visa melhorar a confiabilidade e a manutenibilidade futura, ou como ferramenta para obter uma base mais confiável para futuras ampliações. Empiricamente, é um tipo de manutenção que é evitado, pois vai de encontro ao princípio popular "Se está funcionando, não mexa".

—

O processo de manutenção de software pode ser considerado estratégico no ciclo de vida do software. É através das manutenções que novas funcionalidades são incluídas, ou migrações de ambiente são realizadas. A maneira como estas alterações são realizadas podem dar vantagens competitivas para as empresas que as realizam. Apesar disso, não é o que ocorre; Canfora e Cimitile (2000) reportam que, como a maioria das empresas relegam a atividade de manutenção a um plano secundário, vários problemas são visíveis dentro do processo, como manutenção realizada por pessoal inexperiente, e por volta de 25% das equipes de manutenção são estudantes e 61% novos contratados.

2.2. ARQUITETURA DE SOFTWARE

2.2.1. Conceitos

A disciplina de arquitetura de software é um conceito recente. Sua formalização aparece pela primeira vez no livro "Software Architecture. Perspectives on an Emerging Discipline", em 1996 (Varoto, 2002). Em conseqüência do curto período de existência dessa formalização, existem várias definições para o termo. A definição clássica é dada por Shaw e Garlan (1996, apud (Varoto, 2002))[3], que conceitua arquitetura de software como a definição de um sistema em termos de componentes computacionais e seus relacionamentos.

De maneira similar, Bass et al. (1998) definem arquitetura de software como sendo constituída por um programa ou sistema de computador, compostos por elementos de software, as propriedades visíveis destes elementos e os seus relacionamentos. Um diferencial desta definição é a idéia de que certas características internas devem ser omitidas, pois faz parte da arquitetura somente aspectos externos, ou externamente visíveis, por onde são medidas características como desempenho, tolerância a falhas de execução, recursos compartilhados, e outros.

Garlan (2001) enfatiza a arquitetura de software como uma ponte entre os requisitos e o código fonte, disponibilizando descrições e modelos com alto nível de abstração, que serão usados como guias de projeto (*design*) do sistema em desenvolvimento. Além disso, a arquitetura pode desempenhar um papel importante nos seguintes aspectos:

---

[3] SHAW,M; GARLAN, D. **Software Architecture.** Perspectives on an Emerging Discipline, Prentice Hall, 1996.

1. **Entendimento**: Facilita a compreensão de sistemas por apresentá-los em um nível de abstração que permite que o projeto dos mesmos seja visualizado com maior clareza.
2. **Reuso**: Reforça o reuso tanto de componentes (ou subsistemas) como *frameworks*[4], onde estes componentes podem ser integrados.
3. **Desenvolvimento**: Fornece um esboço de projeto para o desenvolvimento do sistema, determinando seus componentes principais, e as dependências entre eles.
4. **Evolução**: Permite determinar e visualizar os pontos de extensão e evolução do sistema, através dos componentes de software e suas conexões externas.

Monroe et al. (1996) definem que descrições arquiteturais em engenharia de software são amplamente reconhecidas como elementos essenciais para um sistema bem projetado, e que apesar da maioria destas descrições serem informalmente documentadas, ainda assim são importantes no desenvolvimento do sistema.

O conceito de arquitetura de software, além das definições dadas, também serve para criar um vocabulário comum entre os desenvolvedores, arquitetos e analistas. Este vocabulário pode ser adquirido através de estilos de arquitetura e visões de arquitetura, que serão detalhados a seguir.

2.2.2. Estilos de Arquitetura

Estilos de arquitetura podem ser considerados como soluções organizadas e estruturadas para a resolução de algum tipo de problema.

[4] Ver glossário.

Em Bass et al. (1998) é definido um termo semelhante, chamado padrão arquitetural, que consiste na descrição de como os elementos de software e seus relacionamentos trabalham em conjunto, considerando um conjunto de restrições, além de como estes elementos podem ser utilizados. Também é citado como exemplo desta definição o padrão arquitetural "cliente-servidor", onde os elementos cliente e servidor são os elementos de software, e a coordenação entre eles é definida em termos de um protocolo padrão que o servidor utiliza para comunicar com o cliente. Os autores consideram que este estilo de arquitetura (ou padrão arquitetural) é utilizado incontáveis vezes, porém sempre de uma maneira diferente.

Segundo Monroe et al. (1996), estilos de arquitetura caracterizam famílias de sistemas relacionadas por compartilharem de uma mesma estrutura e propriedades semânticas. Uma característica importante decorrente da aplicação dos estilos é a possibilidade de reuso. Uma vez que estilos de arquitetura são aplicações de um conhecimento comum, pode-se deduzir que este conhecimento já foi aplicado com sucesso. Além disso, são citadas quatro características decorrentes dos estilos:

1. Vocabulário de elementos de projeto, como clientes, servidores, bases de dados;
2. Regras de projeto, ou restrições, utilizadas para determinar como estes elementos estão conectados;
3. Interpretação semântica, por onde as composições dos elementos de projeto, conectados pelas regras, possuem significados bem definidos;
4. Análises pré-definidas que podem ser executadas em sistemas, de acordo com o estilo utilizado.

Buschmann et al. (1996, apud (Varotto, 2002))[5] definem 4 categorias de estilos, que são :

- **"*From mud to structure*"**: Estilo que especifica um meio controlado de decomposição de tarefas, através de sub-tarefas cooperativas.
- **Sistemas distribuídos**: Fornece uma estrutura completa para aplicações distribuídas.
- **Sistemas interativos**: Específicos para estruturação de sistemas onde exista interação homem-máquina.
- **Sistemas adaptáveis**: Provê suporte para extensão e adaptação de aplicações.

Uma das formas de representar um determinado estilo de arquitetura, em um nível menor de abstração, é através dos padrões de software. Padrões permitem que idiomas comuns, encontrados repetidamente nos mais variados sistemas e projetos, sejam tornados explícitos e documentados, para que possam ser aplicados em problemas similares (Monroe et al., 1996).

2.2.3. Padrões de Software

O conceito de padrões de software foi criado pelo arquiteto Christopher Alexander, que sintetizou o conhecimento de séculos de arquitetura em padrões, encadeando soluções previamente utilizadas, de modo que, utilizadas em conjunto, resolvam determinado problema. Em sua obra, Alexander afirma: "*Cada padrão*

[5] BUSCHMANN, Frank et al. **A System of Patterns. Pattern-Oriented Software Architecture**. Nova Iorque : Wiley Computer Publishing, 1996.

descreve um problema em um ambiente e o núcleo de sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira" (Alexander et al., 1977, apud (Gamma et al., 1995))[6]. Apesar de Alexander citar padrões como soluções para construções em geral, sua obra teve um maior impacto na Ciência da Computação do que na arquitetura (Salingaros, 1997). Além do trabalho de Alexander, Fowler (1996) cita algumas outras referências sobre a adoção de padrões dentro da análise orientada a objetos. A primeira delas é a iniciativa de Kent Beck e Ward Cunningham que, através das idéias de Christopher Alexander, criaram uma coleção de padrões arquiteturais. Outra referência importante são os *workshops*[7] liderados por Bruce Anderson nas conferências de OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications) no início dos anos 90, com a intenção de investigar material para o lançamento de um livro para arquitetos de software (Grand, 1999). Apesar das várias iniciativas dentro da comunidade de software para a utilização de padrões, o que realmente elevou os padrões de projeto para estado da arte dentro da computação foi o lançamento do livro *Design Patterns: Elements of Reusable Object Oriented Software* (Gamma et al.,1995). Com essa publicação, os autores classificaram 23 padrões que já eram utilizados no desenvolvimento de sistemas, porém ainda não estavam formalizados. Após a publicação deste trabalho, a bibliografia sobre o tema, que era escassa, cresceu tanto em número como em diversificação. Nos dias de hoje, são publicados trabalhos sobre padrões de projeto, padrões de negócio, padrões de interface com o usuário, padrões de computação distribuída, além de vários outros.

[6] ALEXANDER, C. et al. **A Pattern Language**. Nova Iorque, Oxford University Press, 1977.
[7] Ver glossário.

A característica principal dos padrões de software são os pares problema/solução. Geralmente, os padrões não contêm novas idéias nem soluções originais. Padrões, quando são catalogados, já foram testados de várias maneiras, por várias pessoas; portanto, seu uso já está comprovado para a resolução do problema a que se propõe solucionar. Além disso, o padrão de software abstrai tanto o problema como a sua solução, ou seja, a solução apresentada não é imutável. Ao documentar um padrão, o autor cria uma estrutura mínima para a utilização em um projeto, que pode incluir:

- Diagrama de classes: Mostra um conjunto de classes, interfaces e colaborações e seus relacionamentos (Booch et al., 2000, p. 104) ;
- Diagramas de interação: Mostra uma interação, formada por um conjunto de objetos, e seus relacionamentos, incluindo as mensagens trocadas entre eles (Ibid., p. 241);
- Diagramas de estado: Especifica as seqüências de estados pelos quais um objeto passa durante seu tempo de vida em resposta à eventos, juntamente com suas respostas a estes eventos (Ibid., p. 285);
- E outros, como efeitos colaterais ou principais conseqüências.

Apesar dos diagramas serem extremamente úteis para ilustrar a estrutura e a organização do padrão, eles ainda não são suficientes. Para complementar a documentação, Grand (1999) propõe a seguinte forma de descrição de um padrão:

- Descrição do problema;
- Sumário de motivos que levaram à formulação da solução;
- Solução genérica;

- Conseqüências (prós e contras) da utilização da solução para resolução do problema;
- Lista de padrões relacionados.

Além dessa forma de descrição, um formato de padronização de documentação de um padrão é proposto por Gamma et al. (1995):

- **Nome**: O nome do padrão deve expressar em poucas palavras a sua funcionalidade e utilidade.
- **Intenção e Objetivo**: Este tópico serve para descrever o que deve fazer o padrão e que tipo de problema ele resolve.
- **Também conhecido como**: Geralmente padrões possuem vários nomes. Devem-se especificar quais são estes nomes, se houver.
- **Motivação**: Descreve uma utilização do padrão e como as estruturas de classes que compõem o mesmo solucionam o problema.
- **Aplicabilidade**: Descreve em que tipo de projeto e quais classes de problemas podem ser resolvidos com o padrão descrito.
- **Estrutura**: Diagramas podem ser utilizados para especificar a estrutura do padrão. Os mais usados são os diagramas de classes, de seqüência e de colaboração, na notação mais familiar ao projetista.
- **Participantes**: Classes e objetos que fazem parte do padrão.
- **Colaborações**: Como ocorrem as interações entre os objetos do sistema.
- **Conseqüências**: Aqui são especificados os prós e contras da utilização do padrão, bem como as características que o sistema possuirá a partir do momento que aquele padrão for usado.

- **Implementação**: Observações sobre problemas que podem ocorrer devido a utilização de alguma linguagem específica, ou sugestões para a implementação do padrão.
- **Exemplo de código**: Fragmentos de código que exemplifiquem partes chave do padrão, em alguma linguagem de programação.
- **Usos Conhecidos**: Descrição de estudos de caso que utilizaram o padrão com sucesso.
- **Padrões Relacionados**: Geralmente um padrão se utiliza de outros padrões para resolver o problema proposto. Esta seção serve para especificar quais são os padrões utilizados, para futura referência.

Com as informações contidas na descrição de um padrão, é possível ao analista sintetizar de uma maneira ordenada o conhecimento adquirido em sua vivência, transformando esta experiência em um documento, permitindo que outros desenvolvedores e analistas compartilhem de seus conhecimentos.

Apesar dos primeiros padrões documentados terem como objetivo primário documentar práticas de projeto de software, desde que Gamma publicou seu livro, vários estilos de padrões, visando os mais variados tópicos, começaram a aparecer. Os principais são enumerados abaixo:

- **Padrões de Análise**: O foco principal dos padrões de análise é o negócio que será modelado (Fowler, 1996). Este tipo de padrão cobre problemas de modelagem de negócio, reutilizando idéias de análise.
- **Padrões de Projeto**: Os padrões de projeto são o tipo mais difundido de padrão. Os padrões desta modalidade são específicos para o projeto do sistema, focado na arquitetura do sistema. A partir de um modelo conceitual

de classes de negócio, utilizam-se os padrões de projeto para traduzir a linguagem de análise em uma linguagem de programação, já levando em consideração alguns aspectos técnicos, como interação com base de dados ou com a interface com o usuário. A principal referência no assunto é a obra de Gamma et al. (1995).

- **Padrões de Reengenharia**: Gjörwell et al. (2002) definem padrões de reengenharia como "*padrões que descrevem como mudar um sistema legado em um novo e reconstruído sistema que se encaixe em condições e requisitos correntes*". Demeyer et al. (1999) expõem que um padrão de reengenharia ajuda na diagnose de problemas e identificação de fraquezas que podem atrapalhar o desenvolvimento do sistema no futuro. Outro ponto defendido por Demeyer é a utilidade de padrões de reengenharia como ferramentas que descrevem um processo sem propor uma metodologia. Isso pode ser considerado particularmente importante, pois um padrão de reengenharia mostra o que deve ser feito para remodelar o sistema, mas não diz exatamente como, nem quando os padrões devem ser aplicados, dando certa flexibilidade ao analista para resolver o problema.

2.2.4. Visões de Arquitetura

Visões de arquitetura podem ser definidas como as diferentes maneiras que um participante do projeto pode observar o sistema.

Bass et al. (1998) definem que sistemas modernos são demasiado complexos para que sejam entendidos de uma vez só. Ao invés disso, a atenção deve ser restrita a pequenas partes do sistema por vez, partes com semântica definida, para serem discutidas em cada momento. Estas partes são denominadas visões, que consistem

em uma representação de um conjunto coerente de elementos de arquitetura e seus relacionamentos. Estes conjuntos podem ser divididos em três grupos, dependendo de sua natureza:

- **Estruturas de módulos**: Módulos representam uma maneira baseada em código fonte de representar um sistema, com funcionalidades específicas que permitem responder questões do tipo Qual é a responsabilidade funcional associada para cada módulo ? Quais módulos estão associados entre si através de generalização e especialização ?
- **Estrutura de componentes e conectores**: Elementos de software que determinam questões de tempo de execução, como a interação entre elementos, paralelismo, ou como o sistema muda durante sua execução.
- **Estruturas de alocação**: Mostram como os relacionamentos entre os elementos de software e os elementos externos ao sistema se realizam.

Segundo Albin (2003), visões são aspectos ou dimensões de um modelo arquitetural, baseado em modelos estáticos ou dinâmicos. Visões podem ser identificadas como contendo tanto problemas como soluções dentro do domínio do problema. Algumas visões comuns são:

- **Objetivo/Propósito**: Descreve os objetivos que devem ser atingidos.
- **Comportamento/Funções**: Descreve o que o sistema executa, para alcançar os objetivos.
- **Informações/Dados**: Descreve as informações criadas e armazenadas pelo sistema.
- **Forma/Estrutura**: Descreve a estrutura lógica do software.

- **Desempenho**: Descreve como o sistema executa suas funções.

Em Scopus (2001), é definido que um sistema pode ser avaliado através de atributos associados a 5 tipos de visões. Ou seja, ao avaliar cada uma dessas visões, é desejável que estes atributos estejam presentes no projeto de arquitetura. São elas:

- **Visão de Usuário**
    - Deve possuir interfaces intuitivas e consistentes, que não comprometam a produtividade;
    - Deve permitir que a interface possa ser ajustada de acordo com o nível de experiência do usuário;
    - Deve proporcionar retorno ao usuário quanto às tarefas executadas;
    - Deve permitir integração entre módulos, evitando múltiplos logins ou reentrada de dados, e
    - Deve possuir integração com ajuda interativa ao usuário.
- **Visão de Desenvolvedor**
    - A modelagem do sistema deve abordar tanto negócio como tecnologia;
    - Encapsulamento de funcionalidades enfocando o negócio, criando sistemas estruturados de fácil manutenção;
    - Baixo acoplamento entre os módulos, através de modelagens simples, porém efetivas;
    - Facilidade de automação de testes;
    - Documentação de sistema e código;
    - Interface padronizada entre módulos;

- Reutilização organizada de componentes de software e trechos de códigos;
- Capacidade de integração com aplicações existentes (sistemas legados);
- Gestão de configuração, com controle de versão de código fonte, e
- Possibilidade de paralelismo de desenvolvimento.

- **Visão de Manutenção**
  - Baixo acoplamento entre módulos;
  - Facilidade de depuração do sistema, como logs, e
  - Portabilidade dos módulos e sistemas.

- **Visão de Infra-Estrutura**
  - Utilização de ferramentas de desenvolvimento adequadas às necessidades do sistema, e que permitam velocidade de desenvolvimento;
  - Compatibilidade entre o sistema desenvolvido com o *software* e *hardware* básico das estações de trabalho;
  - Definição de uma forma eficiente de empacotamento e distribuição das versões;
  - Adoção de um padrão de comunicação entre sistemas e
  - Distribuição de processamento balanceado entre servidor e estações.

- **Visão de Segurança**
  - Utilização de mecanismos de segurança para acesso ao sistema;
  - Adoção de mecanismos de criptografia para transmissão de dados;

- Assinatura digital em componentes de software desenvolvidos;
- Definição de planos de contingência;
- Mecanismos de *backup* e recuperação de dados, e
- Geração de registros[8] para auditoria.

Esta abordagem pode ser utilizada como uma espécie de *checklist* para avaliação dos requisitos da arquitetura de um sistema, ou da necessidade de manutenção do mesmo, caso algum item considerado importante sob o ponto de vista estratégico não esteja implementado.

—

Nos dias de hoje, onde os prazos estão cada vez menores e as exigências de qualidade se tornam cada vez maiores, é necessário que os sistemas sejam estruturados de maneira que suportem estas situações. A arquitetura de software fornece subsídios para que estes requisitos sejam alcançados, promovendo o reuso (estilos de software e padrões de software) e fornecendo um vocabulário (visões) para melhorar o entendimento do sistema entre os diferentes tipos de pessoas envolvidas nos projetos, como arquitetos, desenvolvedores e analistas de negócio.

2.3. REENGENHARIA

2.3.1. Introdução

A natureza evolutiva de sistemas de informação requer que estes sistemas possam ser adaptados continuamente, seja para inclusão de novas funções ou para a correção de erros, através de manutenções. A reengenharia é uma ferramenta

[8] Do inglês *Audit Log*.

utilizada no processo de manutenção, com a finalidade de transformar um sistema, após seu entendimento.

Demeyer et al. (1999) definem reengenharia como a modificação de algum sistema, a partir da execução de engenharia reversa, ou seja, a reengenharia requer que, em primeiro lugar seja recuperada uma visão da arquitetura em um nível de abstração mais alto para, em seguida, efetuar alterações nessa visão e, finalmente, implementar estas mudanças em código. Ou, resumidamente, mover o sistema do código para um modelo, efetuar as alterações no modelo e depois implementar o modelo para um código com melhor qualidade.

Segundo Gjörwell et al. (2002), a reengenharia preocupa-se com a análise do projeto e implementação de sistemas legados, e a aplicação de diferentes técnicas e métodos de reprojeto e transformação para que os sistemas possuam uma melhor arquitetura.

A reengenharia não é uma atividade trivial. Por ser uma tarefa que envolve várias fases, dentre elas o reconhecimento ou estudo do sistema legado, a remodelagem e a implementação, seus responsáveis devem ter conhecimentos de quais tarefas envolvem a mesma, além de possuir acesso ao sistema legado. Também deve ser levado em consideração que sistemas legados freqüentemente estão em pleno funcionamento, e a tarefa de reengenharia deve se preocupar em separar as visões que fazem sentido serem transformadas. Estas decisões devem ser feitas com prudência, pois o processo como um todo deve gerar um resultado cuja transição ocorra de maneira mais transparente possível. A seguir, algumas definições de termos corriqueiros no processo de reengenharia.

2.3.1.1. Sistemas Legados

Pode-se considerar um sistema como legado a partir do momento em que ele passa a ser utilizado. Segundo Ducasse (2003), um sistema é considerado legado quando ele é herdado de outra equipe de desenvolvimento e ainda possui valor para quem o utiliza. Se sistemas desenvolvidos em COBOL, Fortran ou *Assembly* já foram considerados legados, nos dias de hoje também podem ser citados aplicativos escritos em C++, Smalltalk ou até mesmo Java. Mesmo no desenvolvimento utilizando linguagens orientadas a objeto, que possuem características como encapsulamento e flexibilidade, aplicações com este tipo de tecnologia requer constante investimento para controlar a entropia intrínseca de qualquer desenvolvimento de software. A lei de entropia de software sentencia que mesmo quando um sistema de software é iniciado com um planejamento e projeto corretos, sua estrutura original tende a desaparecer em decorrência de novos requisitos e manutenções que não foram previstas no projeto (Demeyer et al., 1999).

Uma definição similar é apresentada em “Renaissance of Legacy Systems” (1998), onde um sistema legado é um sistema antigo que permanece em operação em uma organização. Ainda, duas características são encontradas neste tipo de sistema:

- Uso de tecnologias e práticas de desenvolvimento antiquadas.
- Tempo de vida do sistema extenso, incorrendo em extensivas mudanças.

Gold (1998) define um sistema legado como “um sistema sócio-técnico contendo software legado”. Seu ponto de vista afirma que um sistema legado deve ser observado pelo ponto de vista técnico, mas também pelos seguintes pontos de vista:

- Ponto de vista Estratégico: Perspectiva de lucros e prejuízos.

- Ponto de vista de Usuário: Perspectiva de automação e suporte a processos de negócio.
- Ponto de vista Técnico: Perspectiva de sistema de software e manutenções.

No presente trabalho será utilizada a definição de Ducasse (2003).

2.3.1.2. Engenharia Reversa

Entende-se por engenharia reversa o processo de obter um projeto a partir de um produto final, conceito muito utilizado na indústria de hardware. Na engenharia de software consiste em extrair, através de técnicas de análise, suas arquiteturas, componentes e relacionamentos.

Demeyer et al. (1999) define o processo de engenharia reversa com a busca de um modelo com alto nível de abstração, a partir de representações de baixo nível de abstração, como código fonte. Ainda, o seu principal objetivo é a documentação do sistema, além de tentar encontrar componentes e partes de software reutilizáveis. No estudo, alguns padrões de reengenharia são apresentados, com o intuito de facilitar o processo.

Panas et al. (2003) apresentam uma visão mais complexa do processo. Segundo os autores, a engenharia reversa pode ser compreendida se houver a divisão da mesma em três sub-processos. São eles:

- **Análise do programa**: Focado na coleta de informações de arquitetura, implementação e componentes internos. A partir de análises em baixo nível (*e.g.* código fonte, esquemas de base de dados), são geradas abstrações de alto nível (*e.g.* diagramas de classes, visões de arquitetura, diagramas entidade-relacionamento, DFD´s).

- **Foco em informações**: Etapa que visa preparar a informação. Isto quer dizer que, a partir dos artefatos gerados na fase de análise do programa, serão gerados novos artefatos, devidamente filtrados para que a informação gerada seja relevante. Existem três maneiras de realizar esta tarefa:
    - **Abstração da Informação**: A partir das representações do sistema, subtrair informações irrelevantes do modelo, que não irão agregar valor no resultado final.
    - **Compressão da Informação**: Neste tipo de filtro, não há perda de informação. As informações irrelevantes são "escondidas", ou seja, permanecem nos modelos, porém em diagramas separados, e não no modelo principal.
    - **Fusão da Informação**: Leva em consideração o fato dos modelos gerados na análise trazerem informações semelhantes, representadas de maneira diferente. Propõe que estas informações sejam fundidas, com a intenção de se obter um modelo mais completo, possibilitando ainda o encontro de componentes e conexões não encontradas nos modelos originais.
- **Visualização do programa**: Etapa que utiliza ferramentas de visualização gráfica, para exame do programa em três níveis:
    - **Visualização de código fonte**: Através da utilização de editores e ambientes de desenvolvimento integrados. Pode incluir também a utilização de ferramentas de rastreamento, para acompanhar as características do sistema em tempo de execução.

- **Visualização em nível de classes**: Através deste processo, as informações estruturadas em formas de classes podem ser visualizadas através de diferentes perspectivas (ou visões), como dependências, hierarquias e pacotes, no caso do sistema ser orientado a objeto.
- **Visualização em nível de arquitetura**: Abstração do sistema, partindo das visualizações de código e classes. Aqui são obtidas as visões de arquitetura relevantes ao processo.

A engenharia reversa é uma disciplina ou processo, que através de técnicas de modelagem e ferramentas de software, torna possível obter um projeto de alto nível de abstração, através do código fonte e infra-estrutura (e.g. base de dados) de um sistema legado. O intuito desta atividade é que o processo de reengenharia possa ser executado com plena consciência do estado interno do sistema que será reprojetado. Indo mais além, o próprio processo de engenharia reversa pode ser encarado como um processo separado, pois seu resultado final é independente da realização ou não da reengenharia de sistemas.

2.3.2. Motivações para realização de reengenharia

A reengenharia pode ser considerada como uma disciplina decorrente da manutenção de sistemas. A sua execução, em muitos casos, depende da decisão de efetuar a manutenção de um sistema.

Demeyer et al. (1999) citam os seguintes itens como motivações para a realização de reengenharia, levando em conta aspectos tanto arquiteturais como tecnológicos:

- **Desacoplamento**: Separação do software como um todo em partes menores, ou subsistemas, passíveis de serem testadas, entregues e comercializadas em separado.
- **Desempenho**: Melhoria do tempo de resposta da aplicação.
- **Adaptação para outra plataforma**: Transformação da implementação atual para implementação em infra-estrutura diferente.
- **Extração de Arquitetura**: Citada como fase essencial para entendimento do sistema. Podem ocorrer casos onde é a motivação principal da reengenharia.
- **Exploração de novas tecnologias**: Pode envolver desde o aproveitamento de novas características de linguagem até mesmo a adoção de novas padronizações, como CORBA (Commom Object Request Broker Architecture) ou UML (Unified Modelling Language).

Braga (1998) também cita que uma das motivações para a realização de reengenharia é a diminuição dos altos custos de manutenção de sistemas. Conforme os sistemas vão sofrendo manutenção contínua, a implementação fica inconsistente com o projeto inicial, o código fonte torna-se de difícil leitura e sujeito a erros, e a documentação do sistema torna-se desatualizada. Outro ponto interessante abordado no estudo citado é a possibilidade da linguagem utilizada no desenvolvimento do sistema tornar-se ultrapassada, inexistindo suporte técnico por parte dos fabricantes, tampouco programadores especializados. Também deve-se levar em consideração as vantagens ou desvantagens da aplicação da reengenharia em um dado sistema. Sistemas pouco usados ou estáveis, que não sofram alterações ou inclusões de novas funcionalidades não compensam o custo da reengenharia. Sistemas que sofrem

constantes alterações podem ser passíveis da aplicação da reengenharia, como forma de manutenção preventiva, com o intuito de evitar problemas futuros. Além disso, o custo da aplicação da reengenharia em um sistema com estas características pode ser diluído em longo prazo, pela melhoria da manutenibilidade e reuso.

Lehman e Belady (1976, apud ("Renaissance of Legacy Systems", 1998))[9] afirmam que as mudanças continuadas de sistemas são inevitáveis, pois qualquer sistema em utilização deve mudar, caso contrário se tornará menos útil com o passar do tempo. Também é ressaltado que, conforme os sistemas vão evoluindo, suas arquiteturas tendem tornar-se cada vez mais complexas, e recursos adicionais devem ser dedicados para a simplificação da estrutura.

2.3.3. Ciclo de vida do processo de reengenharia

O processo de reengenharia pode ser dividido em fases distintas, para uma melhor organização de desenvolvimento, e uma alocação de recursos mais consciente. Quanto a estas fases, não existe um consenso entre os autores, mesmo nas definições dos seus processos.

Gjörwell et al. (2002) definem o processo de reengenharia como três fases principais, sendo elas:

- **Engenharia Reversa**: Primeiro estágio no processo de reengenharia, consiste na tentativa de documentar os pontos fortes e falhos do sistema, bem como localizar componentes reutilizáveis.
- **Análise e Reprojeto**: Segunda fase do processo, onde são detectados os requisitos de reengenharia e mapeá-los em especificações de projeto,

[9] LEHMAN, M.; BELADY, L. **A model of large program development**. IBM Systems Journal nº 15, p.225–252. 1976.

adicionando novos elementos ou consertando defeitos apresentados (manutenção evolutiva/perfectiva x manutenção corretiva).

- **Engenharia Progressiva (*Forward Engineering*)**: Conhecida como etapa de implementação ou construção no ciclo de vida clássico (Pressman, 1995), não difere no processo de reengenharia, pois a implementação é baseada em um projeto já especificado.

Demeyer et al. (1999), apesar de não definirem a reengenharia como um processo em si, determinam seis etapas que devem ser consideradas:

- **Análise de Requisitos**: Etapa onde se identifica os objetivos da reengenharia.
- **Obtenção da Modelagem**: Documentação e modelagem do sistema legado.
- **Detecção de Problemas**: Identificação de áreas problemáticas dentro do sistema. Pode ser efetuada tanto uma análise estática (código fonte, base de dados ou diagramas de classes) como análise dinâmica (verificar como o software se comporta durante o tempo de execução).
- **Análise de Problemas**: Após a detecção dos problemas, estes devem ser analisados e avaliados com base na decomposição em subsistemas elementares. Estes sistemas elementares, por sua vez, são mapeados para soluções, como padrões de software, por exemplo.
- **Reorganização**: Basicamente é a implementação das estruturas identificadas na etapa anterior em código fonte. Incorpora as atividades de implementação usuais, como compilação, depuração e teste.
- **Propagação de mudanças**: Envolve a homologação do sistema reprojetado em ambiente de uso normal.

No estudo de Demeyer et al. (1999), com o intuito de fornecer subsídios para alcançar o objetivo proposto nas seis etapas sugeridas, são criados o que os autores chamam grupos de padrões (*clusters of patterns*). São definidos dois grupos, um enfocando engenharia reversa, e o outro enfocando reengenharia, onde cada grupo fornece padrões de reengenharia com funcionalidades distintas. Dentre os padrões definidos no trabalho, alguns se mostram particularmente úteis dentro de um roteiro simplificado. Eles estão definidos na Tabela 1:

Tabela 1 - Padrões de reengenharia definidos por Demeyer et al. (1999)

Padrão	Foco	Intenção do Padrão
Entrevista durante demonstração	Engenharia Reversa	Obter um primeiro contato com o software legado, através de uma demonstração e questionamento com a pessoa que apresenta o software.
Verificar a base de dados	Engenharia Reversa	Obter uma percepção do modelo de dados do sistema, através do estudo do esquema da base de dados.
Visualizar a estrutura	Engenharia Reversa	Visualizar a estrutura interna do sistema, inclusive anomalias de projeto, através de determinadas visões estruturais.
Conferência com colegas	Engenharia Reversa/Reengenharia	Compartilhar as informações obtidas durante as etapas de engenharia reversa e reengenharia com os envolvidos no projeto.

continua

continuação

Padrão	Foco	Intenção do Padrão
Detecção de código duplicado	Reengenharia	Detectar código duplicado no sistema, sem

		conhecimento prévio do código analisado.
Reparo de Arquitetura defasada	Reengenharia	Detectar e remover dependências entre pacotes de um sistema, não desejadas dentro da arquitetura proposta.

Kazman et al. (1998) propõem uma linha mestra para o processo de reengenharia, denominado “Ferradura” (*horseshoe*). A idéia é separar o processo de reengenharia em 3 dimensões, similar ao processo proposto por Gjörwell et al. (2002), contendo os processos de engenharia reversa, transformação de arquitetura e desenvolvimento. A proposta do autor é esquematizar cada dimensão em camadas, nas quais e cada uma possui uma camada correspondente nas outras dimensões. Para passar da dimensão de Recuperação de Arquitetura para a dimensão de Desenvolvimento, é aplicada uma transformação, baseada em alguma solução definida na dimensão de Transformação de Arquitetura, desde que esteja na camada correspondente. A Figura 1 ilustra o processo:

Figura 1 - Processo "Ferradura" proposto por Kazman et al. (1998)

A idéia principal é mapear uma etapa da reengenharia (arquitetura de base) para outra do desenvolvimento (arquitetura desejada), utilizando os itens propostos pela etapa de Transformação de Arquitetura. Sendo assim, de acordo com a Figura 1,

a partir da representação da arquitetura, na arquitetura de base é possível alcançar a representação da arquitetura desejada utilizando padrões de software e estilos de arquitetura.

Masuda et al. (2000) propõem que o processo de reengenharia não deve ser realizado no sistema como um todo, mas deve ser orientado a pontos chave (*hot spots*), dando foco em determinado aspecto do sistema, de modo a modularizar seu próprio processo de reengenharia. Barkataki et al. (1998) utilizam uma abordagem semelhante, referindo-se aos tópicos chave como segmentos. Estes segmentos podem ser aspectos de arquitetura (persistência de dados, *workflow*[10]), estéticos (interface com usuário), ou escolhido por algum critério do próprio desenvolvedor.

Cada uma das três abordagens citadas de processo de reengenharia citadas (Gjörwell et al., 2002; Demeyer et al., 1999; Kazman et al., 1998) possui suas próprias características, mas também é possível identificar pontos comuns entre eles. Cada um dos métodos, de diferentes maneiras, determina que o processo de reengenharia consiste de três fases, conforme definido por Gjörwell et al.(2002), a saber:

- Entendimento do sistema que será remodelado, em diferentes níveis de abstração (Engenharia Reversa).
- Transformação de características do sistema (Reprojeto) baseado em requisitos de manutenção (perfectiva, evolutiva, ou corretiva).
- Tradução dos modelos de arquitetura em um sistema funcional, obedecendo às características impostas pelos requisitos (Desenvolvimento).

[10] Ver glossário.

2.4. CONCLUSÃO DO CAPÍTULO

Este capítulo apresentou uma visão em alto nível do processo de manutenção, desde a sua necessidade, abordando temas como os tipos de manutenção, até a o processo de remodelagem de um sistema, visando obter uma arquitetura mais clara, de acordo com o mencionado em 2.2.1, através dos fundamentos da reengenharia.

A manutenção de software foi investigada sob o ponto de vista dos tipos de manutenções que podem ocorrer em um sistema, bem como os problemas inerentes a cada tipo de manutenção. Para apoiar a manutenção evolutiva e perfectiva, foi realizado um levantamento sobre a importância da arquitetura de software, quais os parâmetros que devem ser buscados nessas manutenções, e como os padrões de software apóiam este processo. E, por fim, foi efetuado o estudo de algumas abordagens de reengenharia de software que possuem as características de sustentar o processo desde as etapas iniciais até as finais.

3. UM PROCESSO DE TRANSFORMAÇÃO DE ARQUITETURAS DE SISTEMAS LEGADOS BASEADO EM REENGENHARIA

Neste capítulo é apresentado um processo para transformação de arquiteturas. Serão especificados a definição de escopo, seus macro-processos e fases.

Entende-se transformação de arquitetura como uma espécie de manutenção de caráter perfectivo (adaptativa ou evolutiva), tendo como objetivo alterar a estrutura interna de um sistema, a partir de uma necessidade específica. Exemplos de situações em que uma arquitetura deve ser transformada é a necessidade de migração de um sistema para a Internet, a troca de um gerenciador de banco de dados, ou o desacoplamento de seus objetos da interface gráfica, visando uma arquitetura em três camadas.

Uma das características do processo proposto é a sua representação em alto nível, delineando os passos necessários para a transformação de arquiteturas, sem limitar o processo a algum tipo de ferramenta ou técnica específica.

3.1. DEFINIÇÃO DO ESCOPO

O processo definido neste trabalho tem como escopo de aplicação sistemas desenvolvidos em arquitetura baseada em duas camadas, com acesso a bancos de dados, ou algum meio de persistência. Este tipo de sistema é caracterizado por ser desenvolvido por alguma ferramenta RAD (Rapid Application Development), com acesso direto à base de dados, aproveitando os componentes prontos para este tipo de ação. É assumido também acesso ao código fonte do sistema, bem como ao esquema da base de dados, quando houver.

Outro fator importante a ser considerado é que o processo se limita a sistemas e linguagens orientadas a objeto, para aproveitamento das características dos padrões de software (vide item 2.2.3).

3.2. DESCRIÇÃO DO PROCESSO

O processo de transformação de arquitetura prevê três macro processos (Figura 2). A notação IDEF0 (“IDEF0 Overview”, 2000) será utilizada na descrição dos macro-processos e suas fases.

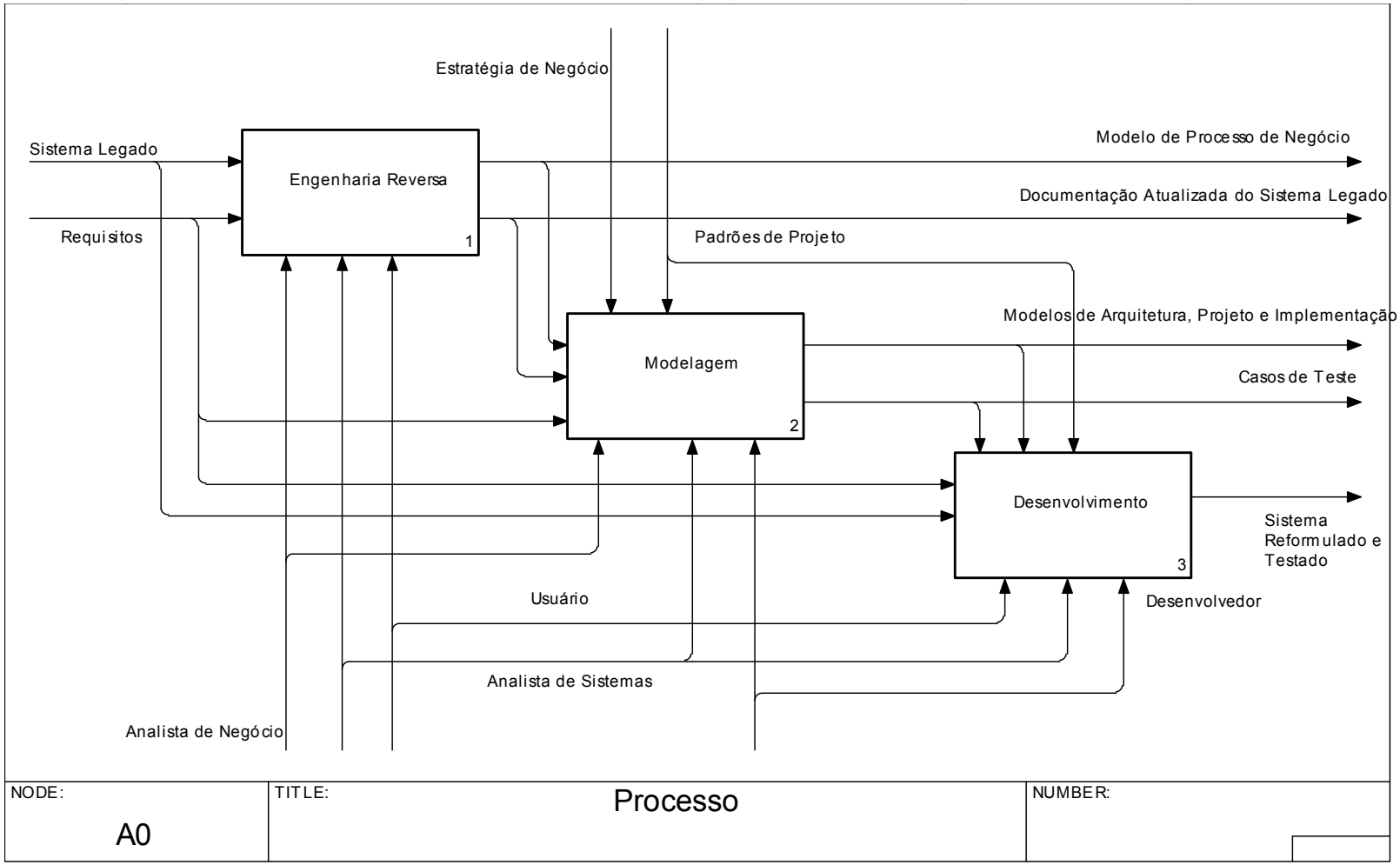

Figura 2 - Diagrama IDEF0 do processo proposto

A etapa de Engenharia Reversa é onde os requisitos do sistema são extraídos através de documentação, código fonte, entrevistas com usuários e desenvolvedores[11].

Na etapa de Modelagem, os requisitos do sistema, funcionais e não funcionais, são enumerados, visando uma melhoria de arquitetura. As novas funcionalidades do sistema (requisitos funcionais), e itens como facilidade de manutenção, escalabilidade e outras melhorias de arquitetura, são levadas em consideração nesta fase.

E, por fim, a etapa de Desenvolvimento. Nesta fase, os artefatos gerados na fase de Modelagem são transformados em um sistema e são realizados os testes para validação.

3.2.1. Perfil da Equipe Responsável pela Execução da Reengenharia

O processo prevê quatro tipos de perfis, que serão responsáveis por liderar ou acompanhar as suas fases. São eles:

- **Analista de Negócios**: Tem conhecimento do processo de negócio apoiado pelo sistema, sabe como traduzir as necessidades do usuário em requisitos de software.
- **Analista de Sistemas**: Possui conhecimento de técnicas de análise, arquitetura e codificação. Lidera o processo como um todo.
- **Usuário**: Conhece do processo e a utilização do sistema. Constituirá uma figura chave no momento em que os testes serão realizados.

[11] As entrevistas podem incluir a utilização do sistema por parte do usuário, para que o analista responsável possa obter dados do próprio sistema em funcionamento.

- **Desenvolvedor**: Possui conhecimentos concretos de codificação e arquitetura bem como de padrões de software.

3.2.2. Engenharia Reversa

As atividades do processo de engenharia reversa tem por objetivo recuperar o estado atual da arquitetura e implementação do sistema. Ela é iniciada a partir da identificação de uma necessidade (requisitos). A Figura 3 mostra os passos desta etapa:

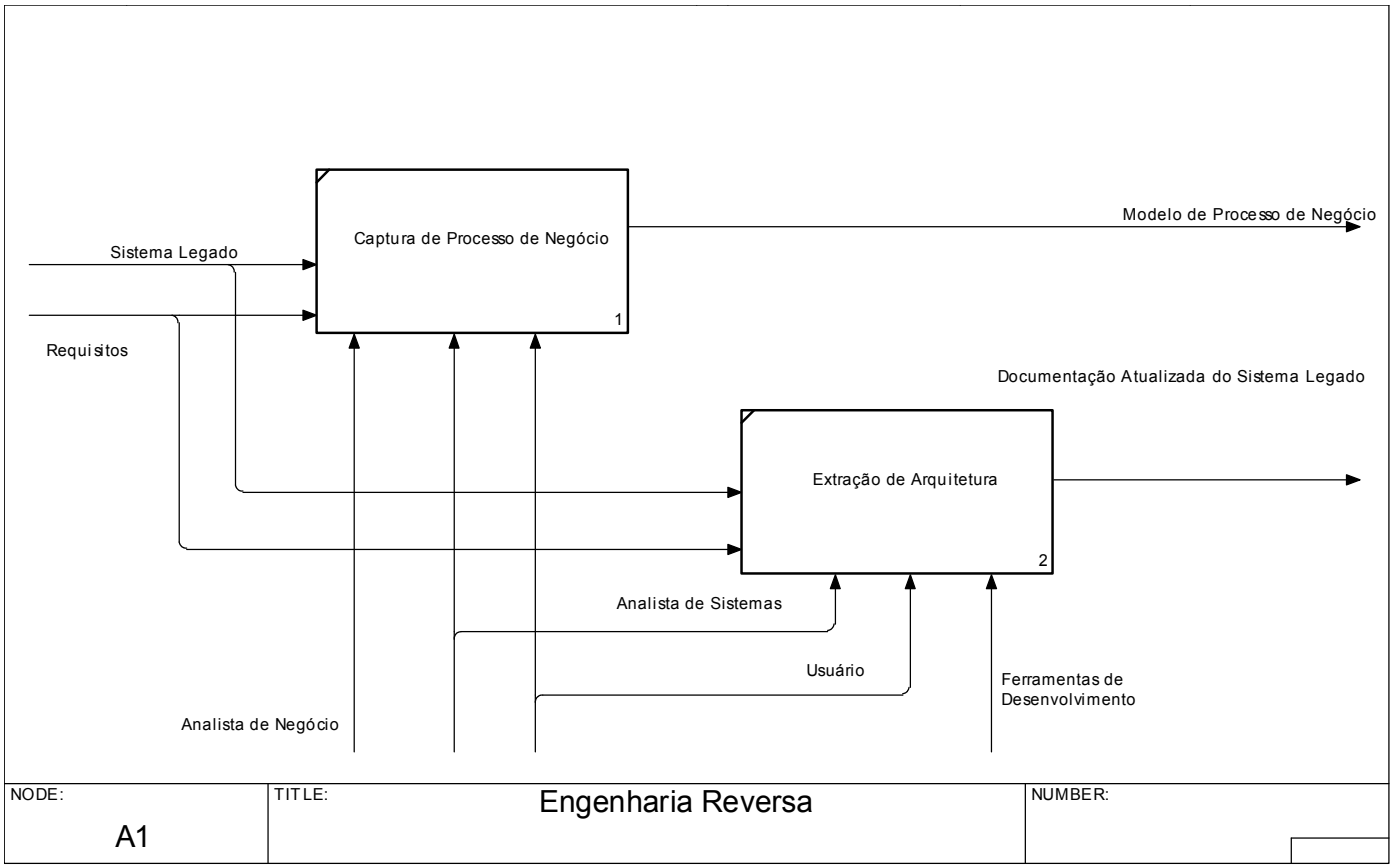

Figura 3 - Detalhamento do processo de Engenharia Reversa

A idéia principal é que, após este passo ser executado, seja possível explicar detalhadamente tanto os aspectos técnicos como as funcionalidades já implementadas do sistema. A proposta é que este item seja realizado através de dois sub-itens, que são:

- **Captura de Processo de Negócio**: Levantamento, através dos meios possíveis, do processo de negócio apoiado pelo software.
- **Extração da Arquitetura**: Obtenção das especificações técnicas e arquiteturais do software.

Cada um destes itens possui suas técnicas e idiossincrasias, que serão detalhadas adiante.

3.2.2.1. Captura de Processo de Negócio

Para um perfeito entendimento de um sistema faz-se necessário conhecer o negócio apoiado por ele. Este conhecimento revela-se útil nos seguintes aspectos:

- **Verificar quais atividades estão automatizadas dentro do processo**: Como é possível que um sistema não automatize completamente um processo, esta técnica torna-se apropriada por tornar possível a avaliação de inclusão de novas funcionalidades no sistema, como a automação de um processo não automatizado, ou a melhoria daqueles processos que já fazem parte do sistema. Uma maneira de realizar esta verificação é através do padrão "Entrevista durante Demonstração"(ver Tabela 1), definido por Demeyer et al. (1999).
- **Avaliação de impacto da reengenharia no funcionamento do sistema**: Através do modelo de processo de negócio é possível verificar quais são os processos que estão conectados e avaliar se problemas na execução de um

processo pelo sistema se propagará pelo restante das funcionalidades implementadas.

Uma técnica particularmente eficaz para modelagem de processos é a notação IDEF0. Esta notação permite capturar o ponto de vista de processos sem manter nenhum vínculo com tecnologia, o que a torna independente de linguagens e metodologias, e ainda permite uma visibilidade tanto em alto como em baixo nível dos processos internos, podendo existir uma clara separação entre o que irá ou não ser automatizado. Além disso, a notação também auxilia o papel do analista com relação ao particionamento do sistema. Outras técnicas de modelagem de negócios também se mostram efetivas nesta questão, como Use Cases (Booch et al., 2000) e fluxogramas.

Além de documentar o modelo de processo de negócio, a principal utilidade do modelo proposto é o fornecimento de subsídios para elaboração dos casos de teste. Através dos processos mapeados no modelo, é possível determinar o que será testado, os parâmetros de teste e os resultados esperados (Almeida, 2003).

Uma vez modelado o processo de negócio, deve-se efetuar a extração da arquitetura legada.

3.2.2.2. Extração de Arquitetura

A extração da arquitetura de um sistema é uma fase eminentemente técnica. Através dela, obtemos conhecimento da implementação do sistema legado, sua arquitetura e seu funcionamento, de modo a obter conhecimento com um grau de abstração mais alto.

Para se conhecer a estrutura interna de um sistema o presente trabalho assume que o seu código fonte esteja disponível (ver item 3.1). Em posse do código fonte, dois passos são sugeridos para apoiar o processo de captura de arquitetura. São eles:

- **Análise de Dados Persistentes**: Visto que grande parte de sistemas armazena seus dados em meios persistentes, uma análise detalhada destes dados pode ser utilizada para complementar a análise do código fonte. Demeyer et al. (1999) define esta atividade dentro do padrão de reengenharia "Verificar Base de Dados" (ver Tabela 1). Ducasse et al. (1999) defendem que a análise dos dados persistentes é viável pelos seguintes motivos:
    - Ferramentas de persistência (bancos de dados) são disponibilizadas com as ferramentas necessárias para sua correta utilização e manipulação de dados e entidades.
    - A própria experiência do analista responsável seria capaz de mapear estruturas estáticas do esquema de persistência para o modelo de classes.

    Nesta atividade são gerados os esquemas das bases de dados, os chamados diagramas entidade-relacionamento.
- **Análise do código fonte**: Através desta análise, é possível obter informações que complementem o modelo de dados obtido na análise dos dados persistentes[12]. Consiste na análise de módulos, componentes e funções, buscando informações que auxiliem o processo da reengenharia, e que

[12] Convém deixar claro que, em sistemas de software, geralmente existem duas versões de código fonte disponíveis: de produção e de pré produção. A versão de produção é a versão do software que está sendo utilizada no cliente, ou seja, uma versão empacotada, e a versão de pré-produção é aquela onde a equipe de desenvolvimento efetua as alterações e correções continuas, sem efeito para os usuários do sistema, até que esta se torne a versão de produção. É recomendado que a análise seja feita da versão de pré-produção, visando considerar as implementações corretivas que possam existir.

proporcionem informações úteis para elucidar a arquitetura do sistema. Além disso, é nesta fase em que podem ser encontrados módulos e partes de código fonte de que podem ser reutilizados tanto no próprio sistema como na transformação de arquitetura, reaproveitando funções, classes e módulos.

A intenção de realizar estas atividades é obter um modelo de representação de classes que consiga espelhar o sistema com um nível mais alto de abstração. Este modelo é definido por Larman (1998) como modelo conceitual de classes. A justificativa para um modelo conceitual é que, dada a natureza de duas camadas do sistema, em princípio não existirão classes de objetos determinadas em seu código fonte.

3.2.3. Modelagem

A fase de modelagem consiste em determinar qual parte da arquitetura será transformada, e como. O enfoque que será dado neste trabalho é o de reengenharia orientada a pontos chave (item 2.3.3) através de visões de arquitetura (item 2.2.4). A Figura 4 detalha o processo:

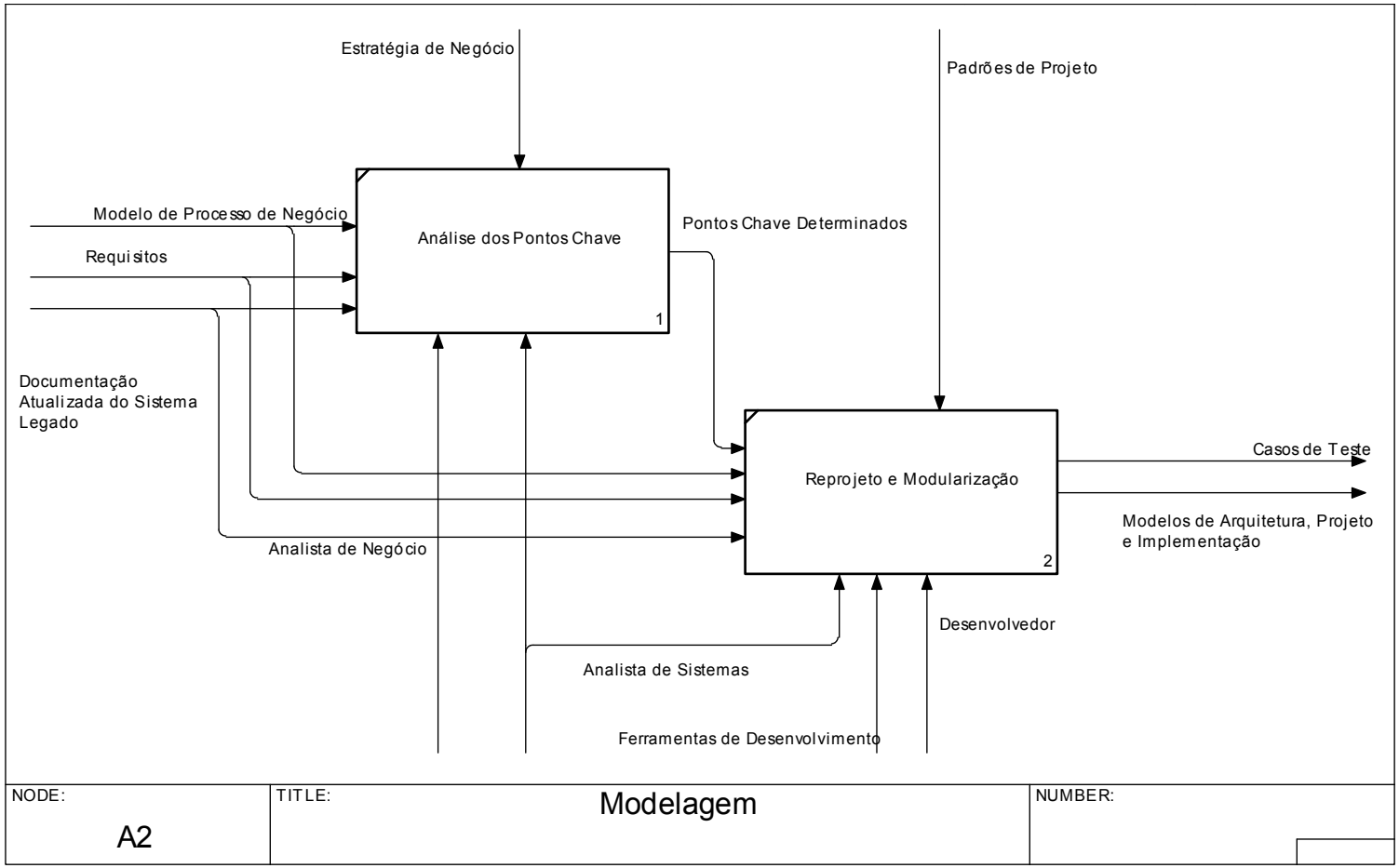

Figura 4 - Detalhamento do processo de Modelagem

3.2.3.1. Análise de Pontos Chave

O objetivo primário desta atividade é, através dos requisitos de transformação, determinar quais partes do sistema serão afetadas pelo processo de reengenharia, e mapear estes requisitos de sistema em requisitos de arquitetura. A proposta é que, através de um consenso entre a equipe de desenvolvimento, sejam catalogados todos os aspectos do sistema que devem ser alterados, baseando estas escolhas nos requisitos iniciais. Como exemplo, são citados os casos da Tabela 2:

Tabela 2 - Mapeamento de requisitos iniciais para requisitos de arquitetura

Requisitos Iniciais	Requisitos de Arquitetura/Pontos Chave
Migração do sistema para Internet	• Promover o desacoplamento de objetos
Alteração de Sistema Gerenciador de Banco de Dados	• Agregar camadas abstraindo o acesso ao gerenciador (*n-tier*[13])
Retirar o acoplamento entre o sistema e aplicativos periféricos, como sistemas de *mainframe*[14]	• Promover o desacoplamento de objetos • Criar objetos intermediários para acesso a sistemas remotos

Para realizar esta tarefa, é necessária a interação entre o analista de negócios e o analista de sistemas para que a escolha dos pontos chave seja realizada de modo que os requisitos iniciais sejam satisfeitos. Não existe uma técnica específica para

[13] Ver glossário
[14] Ver glossário

determinar os pontos chave de um aplicativo. A idéia principal é buscar um incremento qualitativo no sistema legado, baseado nos requisitos iniciais.

3.2.3.2. Reprojeto e Modularização

Na fase de reprojeto e modularização, os pontos chave determinados na fase anterior são reprojetados. Nesta fase é sugerida a utilização de padrões de software, visando reutilização de estilos de arquitetura (item 2.2.2).

Dando continuidade ao processo, devem-se definir os tipos de padrões que se adaptam na remodelagem; como cada ponto chave tem sua funcionalidade específica, deve-se avaliar os padrões existentes, combinando-os se preciso for, para a obtenção de uma nova arquitetura.

Como produto final da fase, dois artefatos são definidos:

- **Modelos de Arquitetura, Projeto e Implementação**: Modelos baseados nos pontos chave escolhidos. Representam graficamente a solução adotada para resolução do problema, detalhando estruturas de dados (diagramas de classes), fluxos de execução (diagramas de seqüências), infra-estrutura (diagramas de implantação) e arquitetura (diagramas de componentes, interfaces e pacotes).
- **Casos de Teste**: Documentos gerados com o propósito de validarem as alterações que o sistema sofreu. Baseiam-se no modelo de processo de negócio e nos modelos de arquitetura (Almeida, 2003).

Após a modelagem e a definição da arquitetura, pode-se passar para a fase de desenvolvimento.

3.2.4. Desenvolvimento

A fase de desenvolvimento compreende duas fases: a implementação e a fase de testes.

Na fase de implementação, a arquitetura definida na fase de reprojeto e modularização (item 3.2.3.2) é convertida em uma linguagem, e por fim, na fase de testes, são validados se os requisitos de arquitetura foram alcançados e se os requisitos funcionais foram mantidos. A Figura 5 ilustra como as fases estão dispostas:

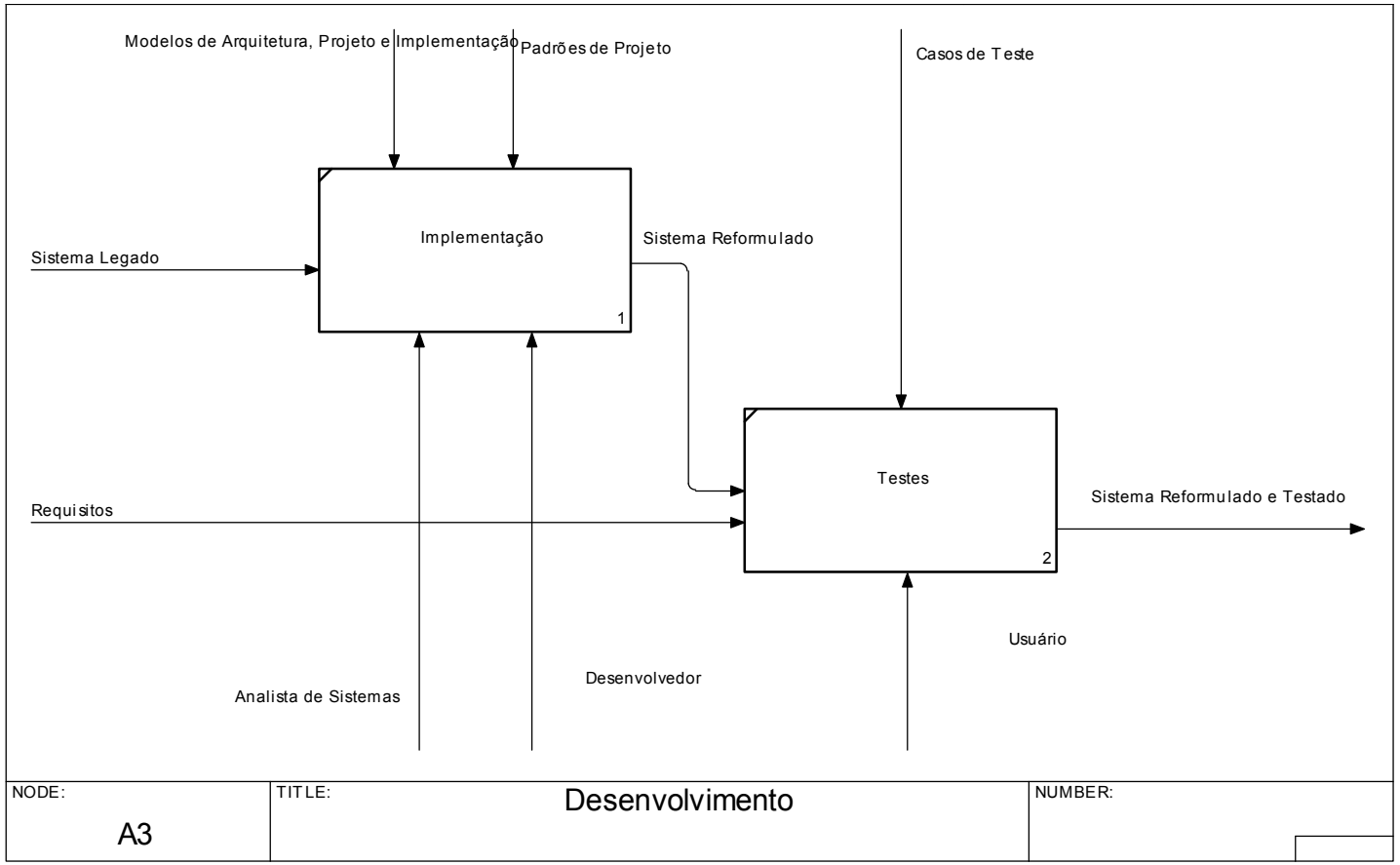

Figura 5 – Detalhamento do processo de Desenvolvimento

3.2.4.1. Implementação

A partir dos artefatos gerados pela fase de Modelagem, são iniciadas as atividades de implementação. Os modelos gerados servem para direcionar o desenvolvimento, fornecendo as diretivas que serão utilizadas para orientar o processo. Essas atividades não diferem do modelo tradicional de implementação definido por Pressman (1995), com exceção da inclusão da utilização de padrões de software.

Na implementação dos padrões que foram utilizados na etapa de reprojeto e modularização, devem ser utilizadas as próprias orientações de implementação da descrição do padrão. Cada padrão tem suas próprias características de implementação (item 2.2.3); então estas características devem ser combinadas, ficando a cargo de um analista ou programador escolher a maneira de desenvolver que esteja mais de acordo com o estilo de programação do sistema em questão.

3.2.4.2. Testes

A atividade de testes é a última fase dentro do processo. Aqui, o usuário deverá validar as alterações de sistema e as inclusões de funcionalidades, para garantir que os requisitos de transformação de arquitetura foram alcançados e os requisitos funcionais anteriores ao processo de manutenção foram mantidos. Neste caso específico, assume-se que os testes convencionais ao desenvolvimento (*e.g.*, teste de unidade, teste de integração) já foram aplicados na etapa de implementação.

3.3. CONCLUSÕES DO CAPÍTULO

No presente capítulo foi apresentado um processo, juntando as partes necessárias de um processo de reengenharia em poucos passos, procurando manter certa flexibilidade para a equipe que irá desenvolvê-lo.

Os principais elementos do processo de Engenharia Reversa se encontram na Tabela 3, os elementos da Modelagem na Tabela 4, e os elementos de Desenvolvimento na Tabela 5:

Tabela 3 - Processo de Engenharia Reversa

Processo	Responsável	Apoiado por	Atividades	Produtos
Captura de Processo de Negócio	Analista de Sistemas	Analista de Negócios Usuário do Sistema	• Identificar o processo de negócio • Identificar os processos apoiados pelo sistema	• Detalhamento dos processos apoiados pelo sistema • Diagrama de processo de negócio
Extração de Arquitetura	Analista de Sistemas	Usuário	• Análise do código fonte (módulos, funções e componentes) • Análise dos dados persistentes	• Modelos Estáticos de Arquitetura • Modelos Dinâmicos de Arquitetura • Esquema da base de dados

Tabela 4 - Processo de Modelagem

Processo	Responsável	Apoiado por	Atividades	Produtos
Análise de Pontos Chave	Analista de Sistemas	Usuário	• Analisar os pontos que devem ser refeitos dentro do sistema, para satisfazer os requisitos	• Documento formalizando as partes ou visões do sistema que serão transformadas
Reprojeto e Modularização	Analista de Sistemas	Desenvolvedor	• Modelar o sistema visando atender os requisitos de transformação, preferencialmente utilizando padrões de projeto	• Modelos estáticos, dinâmicos, e outros, que sejam úteis no processo de desenvolvimento
			• Elaborar os casos de teste	• Casos de Teste

Tabela 5 - Processo de Desenvolvimento

Processo	Responsável	Apoiado por	Atividades	Produtos
Implementação	Desenvolvedor	Analista de Sistemas	• Implementação do sistema, utilizando como base as especificações geradas na modelagem	• Código fonte • Programa executável • Pacotes de instalação
Testes	Usuário		• Testes de validação do sistema	• Relatório de aprovação ou reprovação do sistema

4. ESTUDO DE CASO – SISTEMA DE AUTOMAÇÃO DE PROPOSTAS COMERCIAIS

4.1. CONSIDERAÇÕES INICIAIS

Como estudo de caso foi escolhido um sistema onde o autor participou como analista e desenvolvedor.

O sistema que será analisado neste estudo de caso tem a finalidade de controlar o fluxo de informações das propostas comerciais emitidas pela área comercial e viabilização dos serviços contratados pelas suas filiais. A viabilização de serviços consiste em uma verificação da possibilidade da execução dos mesmos pelas filiais. Um serviço pode ser inviabilizado caso o ponto de execução esteja fora de rotas de abrangência ou os horários disponíveis para a execução não estejam de acordo com a necessidade do cliente.

O sistema foi escolhido como estudo de caso por apresentar duas características:

- Conformidade com as características do modelo proposto, como desenvolvimento em duas camadas, persistência em base de dados e fácil acesso ao código fonte (item 3.1);
- Necessidade de portar o sistema para uma utilização via Internet/intranet (requisito de transformação de arquitetura);

É importante frisar que o estudo de caso se atém à aplicação do processo de transformação de arquiteturas, dentro do escopo definido, não levando em

consideração itens como infra-estrutura de software (*e.g.*, monitores transacionais, sistemas operacionais) ou hardware (*e.g.*, servidores, configurações de rede). Além disso, somente itens relevantes são mostrados para ilustrar a aplicação do processo, pois o detalhamento de itens de arquitetura do estudo de caso não faz parte do escopo do trabalho de pesquisa.

4.2. ETAPAS DE APLICAÇÃO DO PROCESSO

4.2.1. Engenharia Reversa

Como primeiro passo dentro do processo, serão obtidas as informações do sistema em sua implementação atual, passando pelas duas fases pertencentes à engenharia reversa.

4.2.1.1. Captura de Processo de Negócio

Para alcançar o objetivo de obter um modelo de processo de negócio, o mais fiel possível, foram realizadas reuniões com usuários e analistas de negócio da empresa. Através destas reuniões, foram levantados os processos e seus detalhes, descritos na Tabela 6:

Tabela 6 - Processos apoiados pelo sistema

Processo	Descrição
Triagem da Proposta	Nesta fase, os atendentes capturam os dados necessários para a confecção da proposta, como postos de coleta/entrega e periodicidade para prestação do serviço. Terminado o atendimento, a proposta é gerada, e enviada à(s) filial(ais), para aprovação operacional.
Aprovação Operacional	Assim que a(s) filial(ais) são notificadas da nova proposta, é verificada a viabilidade de operação, como disponibilidade de carros e rotas no(s) endereço(s) que devem ser atendidos. No caso de indisponibilidade, a proposta é re-encaminhada à triagem, para alteração da proposta de acordo com a possibilidade da(s) filial(ais). Uma vez sendo aprovada operacionalmente, a proposta passa para o processo seguinte que é a informação de preços pelo departamento comercial.
Informação de Preços	O departamento comercial, de acordo com algumas informações gerenciais do cliente, e a aprovação, informa o preço do serviço, levando em consideração todas as informações necessárias de propostas já efetuadas do cliente.

continua

continuação

Processo	Descrição

Aprovação	Neste processo, a proposta é enviada para aprovação do cliente, para aprovação. Neste caso, existem três possibilidades. A proposta pode ser aprovada, reprovada, e ainda pode voltar para a informação de preços, para ser renegociada.Sendo aprovada, a proposta segue para o processo seguinte, a geração de contrato.
Geração de Contrato	Com a proposta aprovada é gerado um contrato e enviado para o cliente.

Como complemento à descrição dos processos, foi gerado um diagrama em IDEF0, e posteriormente validado com os usuários. Este diagrama é mostrado na Figura 6:

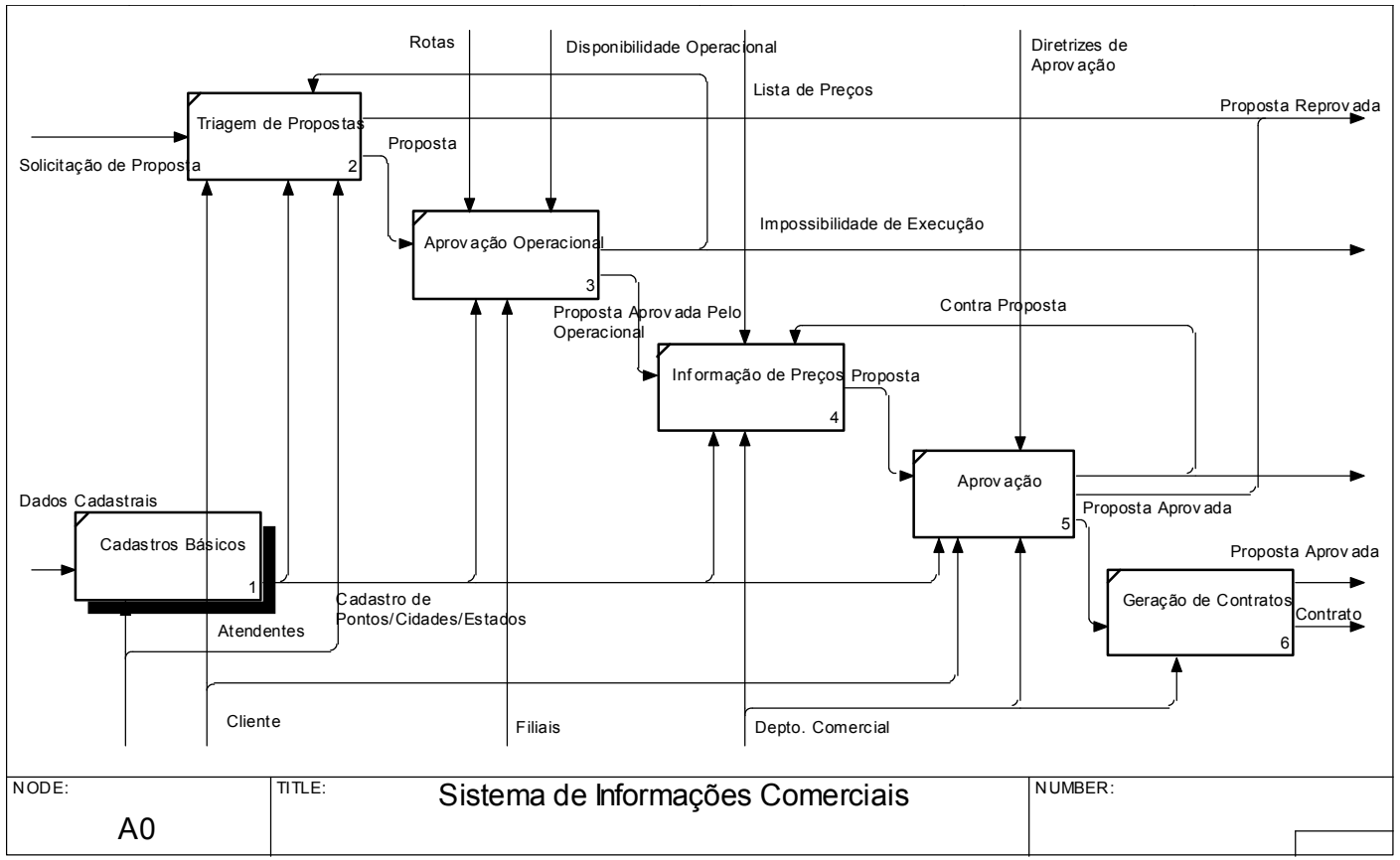

Figura 6 - Diagrama de processo de negócio em IDEF0 do sistema estudado

Além dos processos descritos, ainda existem outras características do sistema que não aparecem nos processos. Estas características são as seguintes:

- Consultas às propostas já efetuadas para comparação de preços de serviços;
- Estatísticas de atendimento da área comercial, de aprovação e confecção de propostas;
- Integração com sistemas legados, em plataforma alta (*mainframe*[15]).

4.2.1.2. Extração da Arquitetura

Dando continuidade à aplicação do processo, as duas atividades sugeridas no item 3.2.2.2 foram realizadas para obter os modelos de extração de arquitetura.

A base de dados encontrava-se em um servidor MS SQL Server 7, que fornece as ferramentas necessárias para a engenharia reversa. O resultado pode ser conferido na Figura 7:

[15] Ver glossário.

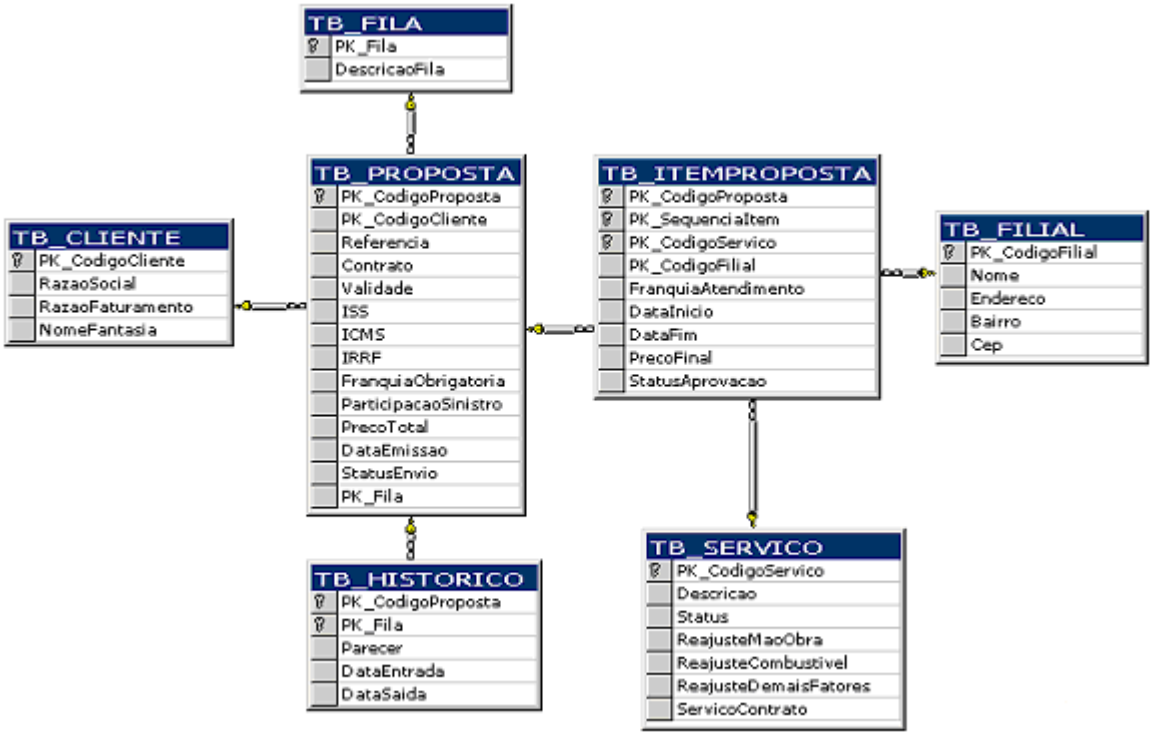

Figura 7 - Base de dados extraída do sistema de automação de propostas comerciais

Através da análise do código fonte foi possível analisar que o software controla o fluxo das propostas (item 4.1) através de atributos na base de dados. Isso significa que, dentro do modelo relacional de dados, existe um atributo na tabela que representa as propostas, que armazena em que estado se encontra a mesma; este atributo é chamado PK_Fila, e também representa uma tabela dentro da base de dados (tabela TB_FILA). Uma proposta só pode estar em uma única fila em determinado instante. Através de alguma ações específicas para cada fila (operações *Aprovar()* e *Reprovar()*), uma proposta pode avançar para a próxima fila ou retroceder para a fila anterior. Uma proposta tem um estado inicial (triagem) e através de operações de Aprovação ou Reprovação, pode ser enviada para um estado posterior ou anterior ao estado corrente. Pode-se perceber que o diagrama de estados é semelhante ao fluxo das informações apresentados no IDEF0 do sistema(Figura 6). Essa semelhança é devido ao processo de negócio ser focado justamente no fluxo da proposta comercial(que deve percorrer vários processos dentro do sistema, até que seja finalmente aprovada). A Figura 8 representa o diagrama de estados possíveis para a proposta:

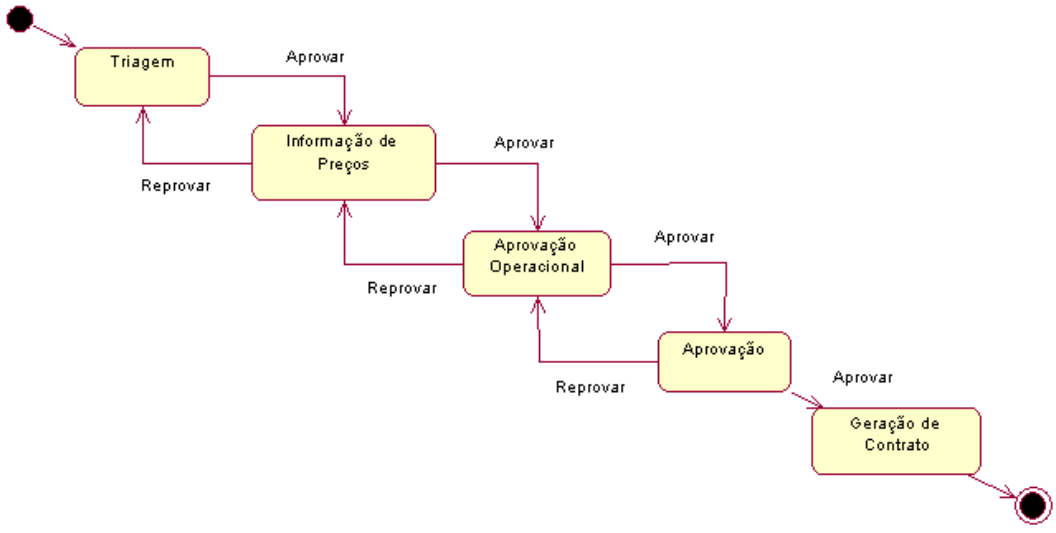

Figura 8 - Diagrama de estados da proposta

Apesar de ser uma abordagem que atende às necessidades do sistema, é possível que se esbarre em problemas de manutenção. Ao ser adicionada ou retirada uma nova fila no sistema, essas alterações incorrem em manutenção do sistema como um todo.

Em virtude de ser uma base de dados com poucas tabelas, e não haver complexidade em seus relacionamentos, o modelo de classes resultante será o simples mapeamento objeto relacional da base de dados, conforme a Figura 9:

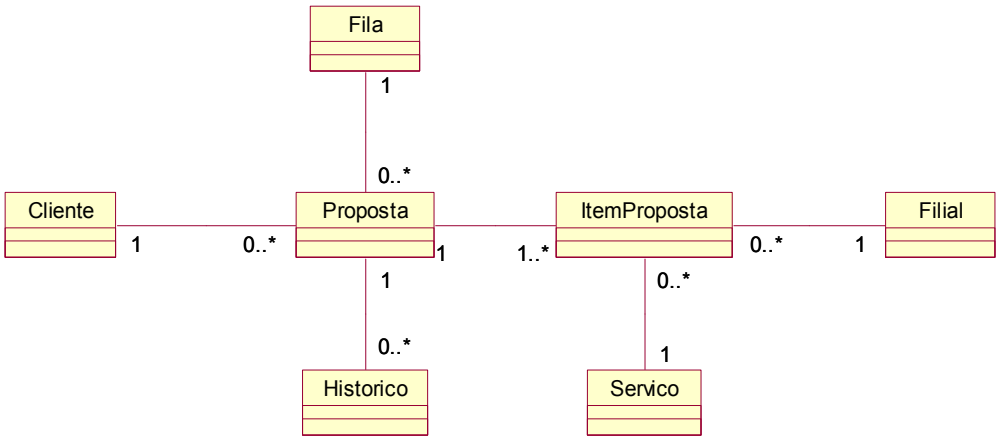

Figura 9 - Modelo conceitual de classes do sistema

Dando continuidade, o próximo passo realizado é a modelagem, onde a transformação da arquitetura será efetivamente realizada.

4.2.2. Modelagem

Utilizando os artefatos gerados na etapa anterior será realizada a transformação da arquitetura do sistema. O primeiro passo é determinar os itens que serão remodelados/otimizados (item 4.2.2.1) e, em seguida, será realizada a modelagem destes itens (item 4.2.2.2).

4.2.2.1. Análise de Pontos Chave

O requisito inicial de transformação de arquitetura consiste em portar o sistema atual para ser utilizado via Internet (item 4.1). Para obter este resultado foi determinado

que o ponto chave para reprojeto deveria ser a implementação do desacoplamento entre os objetos representados nas tabelas da interface gráfica. Desta maneira, os objetos (representados nas tabelas) poderiam ser acessados por várias plataformas (cliente-servidor, Internet), mantendo o princípio do encapsulamento.

4.2.2.2. Reprojeto e Modularização

A solução apresentada para resolver o problema de acoplamento entre os objetos de negócio e a apresentação é incluir uma camada adicional entre a base de dados e a interface com o usuário. Essa abordagem, conhecida como arquitetura em 3 camadas, é detalhada em Fowler (1996) e Sadosky et al. (2000). Outro ponto importante é que, visto que o sistema irá alterar os dados dos objetos dentro de um servidor de banco de dados, através do modelo relacional, devem ser fornecidas operações para manipulação destes dados. Será utilizado o padrão RCRUD (Usaola et al., 2001) para resolver esta questão, por ser o mais adequado, do ponto de vista de simplicidade e funcionalidade. Neste caso, a experiência do analista alocado no projeto foi responsável pela escolha dos referidos padrões. O detalhamento de ambos os padrões encontra-se no Apêndice A.

Para implementar esta arquitetura, os seguintes passos serão executados:

- Mapear cada uma das tabelas da base de dados (Figura 7) para uma classe correspondente. Este mapeamento é o mesmo modelo representado pela Figura 9.

  Incluir nestas classes os métodos correspondentes à instanciação e alteração destes dados, de acordo com o padrão RCRUD. A inserção destes métodos pode ser efetuada através de herança, conforme visto na Figura 10:

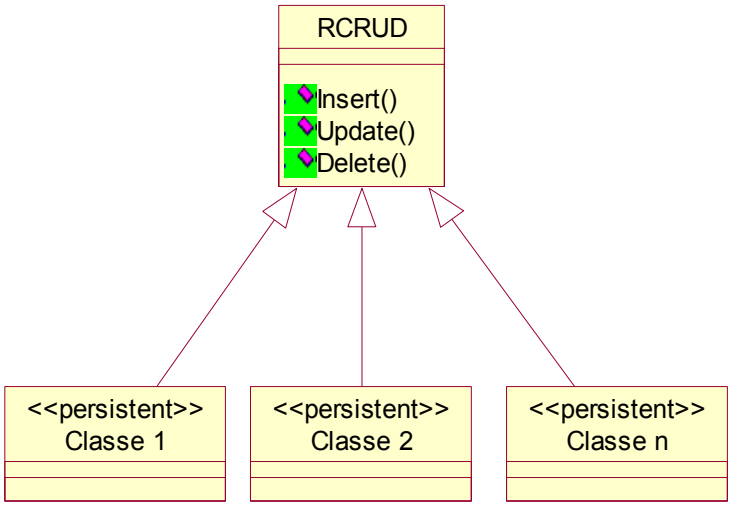

Figura 10 - Padrão RCRUD aplicado ao sistema

Foi criada uma classe adicional, chamada RCRUD, e todas as classes do sistema devem realizar a extensão da mesma, para herdarem a assinatura dos métodos.

- Incluir os métodos relacionados ao negócio (*Aprovar()* e *Reprovar()* na classe Proposta), conforme a Figura 11:

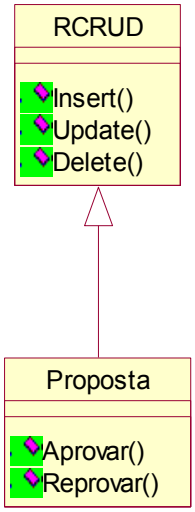

Figura 11 - Objeto Proposta e seus métodos

- Definir a infra-estrutura que habilite o acesso via Internet que trabalhará junto com estes objetos.

Convém enfatizar que este estudo de caso não pretende detalhar o projeto da nova arquitetura, mas sim exemplificar como seriam os artefatos gerados por esta etapa do processo. Da mesma maneira, os casos de teste devem ser elaborados pelo analista de negócios e analista de sistemas e, posteriormente, serem validados pelos usuários.

4.2.3. Desenvolvimento

Nesta etapa, os desenvolvedores responsáveis transformam os modelos definidos em código fonte e são realizados os testes definidos na etapa de Modelagem para aceitação do sistema.

4.2.3.1. Implementação e Testes

A plataforma escolhida para a implementação dos objetos foi a arquitetura de páginas dinâmicas Java Server Pages. Os objetos, implementados em linguagem Java, são acessados pelas páginas, atendendo ao requisito de acesso via Internet.

A implementação pode ser dividida em duas partes distintas: a implementação dos objetos e a implementação das páginas. Esta implementação pode ser seqüencial ou paralela, dependendo da disponibilidade de recursos do projeto. Uma vez terminada a implementação, e realizados os testes de unidade, passa-se para os testes do sistema implementado, realizados pelos usuários.

4.3. RESULTADOS OBTIDOS

A aplicação do processo de transformação de arquiteturas no presente projeto obteve o resultado esperado, apesar de ser prematuro tentar avaliar a validade do processo como um método, dado que ele foi aplicado apenas neste projeto.

Algumas atividades mostraram alguma dificuldade para serem executadas, como o levantamento do processo de negócio e a extração da arquitetura, pois os profissionais que poderiam informações mais detalhadas sobre a arquitetura do sistema ou o processo

de negócio não estavam disponíveis para consulta. Além disso, o mapeamento dos requisitos de alteração do sistema para a nova arquitetura demanda conhecimentos avançados de arquitetura do analista de sistemas responsável, pois uma decisão incorreta neste momento pode levar o projeto ao fracasso.

Uma característica importante da aplicação do processo é o conhecimento antecipado dos processos e seus responsáveis, facilitando ao responsável a correta avaliação dos recursos que serão utilizados. Além disso, a aplicação do processo também recuperou aspectos do sistema desconhecidos até então, como algumas regras de negócio implementadas diretamente no código fonte.

5. CONSIDERAÇÕES FINAIS

5.1. CONCLUSÕES GERAIS E CONTRIBUIÇÕES DO TRABALHO

As pesquisas efetuadas sobre manutenção de software e reengenharia, apresentadas no capítulo 2, mostram que ambos os assuntos estão ligados, sendo a reengenharia uma decorrência direta do processo de manutenção.

A elaboração deste trabalho de pesquisa permitiu juntar em um processo algumas práticas comuns dentro do processo de manutenção e reengenharia em um processo de transformação de arquiteturas, bem como mostrar uma aplicação prática do mesmo.

Dentre os modelos de reengenharia existentes, este trabalho se diferencia pelas seguintes propostas:

- Processo de transformação de arquiteturas dividido em macro-processos, sendo cada um destes dividido em fases, cada uma com responsabilidades bem definidas;
- Encadeamento entre as fases, onde cada uma gera resultados que serão utilizados nas fases seguintes.

Além disso, uma preocupação dentro da proposta é a preocupação com reutilização, que pode ser conseguida tanto através da análise do código fonte do sistema sendo trabalhado como pela adoção de padrões de projeto, que são uma maneira de reutilizar arquiteturas.

5.2. SUGESTÕES PARA PESQUISAS FUTURAS

O processo apresentado não cobre todas as etapas do desenvolvimento de sistemas. Para tornar o processo mais flexível e completo, propõe-se uma adequação do mesmo ao modelo de desenvolvimento espiral, incluindo etapas como análise de riscos e avaliações rotineiras por parte do cliente, bem como a implantação de iterações dentro

do processo, tornando possível que a transformação da arquitetura torne-se mais abrangente.

Outro ponto é a não existência de um mecanismo para medir os resultados do processo. Sem medições adequadas, não é possível avaliar se o processo proposto é eficiente ou não. Outro fator ligado aos resultados é o estudo do paralelismo que pode ser obtido entre as fases do processo. Uma formalização de mecanismos de paralelismo entre as atividades, alterando o processo como um todo de maneira que as atividades posteriores não dependam dos artefatos gerados contribuiria para a diminuição do tempo de desenvolvimento.

Outro tópico que pode ser desenvolvido a partir da aplicação do processo ao estudo de caso proposto neste trabalho seria a elaboração de um processo de transformação de arquitetura específico para sistemas Internet, em 3 camadas. A abordagem adotada, utilizando os padrões *3 tier architecture* e *RCRUD,* é genérica o suficiente para ser aplicada em sistemas que atendam às restrições de aplicação do processo e possuam os mesmos requisitos de transformação de arquitetura, no caso, o porte do sistema para Internet.

REFERÊNCIAS*

ALBIN, Stephen T. **The Art of Software Architecture: Design Methods and Technologies**. Nova Iorque: Wiley Computer Publishing, 2003. 312 p.

ALMEIDA, Edson S. de. Requisitos Funcionais e a Especificação dos Casos de Teste de Homologação do Sistema. **Revista Integração ↔ Pesquisa ↔ Extensão**, São Paulo, nº34, p.172-179, ago. 2003.

BARKATAKI, S.;HARTE, S.;DINH, T. Reengineering a Legacy System Using Design Patterns and Ada-95 OO Features. In: Annual International Conference on ADA, 1., 1998, Japão. **Proceedings of the 1998 annual ACM SIGAda international conference on Ada**. Washington D.C.: ACM Press, 1998. p. 148-152.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture In Practice**. 1998. 2ª ed. Addison-Wesley, 2003. 560 p.

BOOCH, Grady et al. **UML**: Guia do Usuário. Tradução Fábio Freitas. Rio de Janeiro: Campus, 2000. 496 p. Título Original: The unified modeling language user guide.

BRAGA, Rosana T. **Padrões de Software a partir da Engenharia Reversa de Sistemas Legados**. 1998. 144 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, São Carlos. 1998.

CHAPIN, Ned et al. Types of software evolution and software maintenance**. Journal of Software Maintenance**, Nova Iorque, vol. 13, n.1, p.3-30, jan. 2001

* De acordo com:
ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 6023**: Informação e documentação: referências: elaboração. Rio de Janeiro, 2002.

DUCASSE, S. **Reengineering Object-Oriented Applications**. 2003. 157 f. Habilitação para condução de pesquisas – Université Pierre et Marie Curie, Paris, França, 2003.

DUCASSE, S.; DEMEYER, S.; NIERSTRASZ, O. A pattern language for reverse engineering. In: EUROPEAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING AND COMPUTING, 5., 1999, Alemanha. **Proceedings of the 5th European Conference on Pattern Languages of Programming and Computing**. Konstanz, Universitätsverlag, 1999. p. 189-208.

_____; _____; _____. **Object Oriented Reengineering Patterns**. Morgan Kaufmann, 2002. 282 p.

FOWLER, Martin. **Analysis Patterns**: Reusable Object Models. Addison Wesley, 1996. 384 p.

GAMMA, Erich et al. **Padrões de Projeto**: Soluções Reutilizáveis de Software Orientado a Objeto. 1995. Tradução Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000. 368 p.

GJÖRWELL, Daniel; HAGLUND, Staffan; SANDEL, Daniel. **Reengineering and Reengineering Patterns**. Department of Computer Science and Engineering Mälardalens Högskolas, 2002.

GRAND, Mark. **Design Patterns in Java, Volume 2**. Nova Iorque : Wiley Computer Publishing, 1999. 368 p.

LARMAN, Craig. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design. Prentice Hall PTR, 1998.

MASUDA, G.; SAKAMOTO, N.; USHIJIMA, K. Redesigning of an Existing Software using Design Patterns. In: INTERNATIONAL SYMPOSIUM ON PRINCIPLES OF

SOFTWARE EVOLUTION, 1., 2000, Japão. **Proceedings of the International Symposium on Principles of Software Evolution**. Kanazawa, Japão, 2000. p. 165-169.

PANAS, T.; LÖWE, W., AßMANN, U. Towards the Unified Recovery Architecture for Reverse Engineering. **Proceedings of the International Conference on Software Engineering Research and Practice**, Nevada, EUA, vol. 2, p.854-860, jun. 2003.

PRESSMAN, Roger. **Engenharia de Software**. 1995. Tradução José Carlos Barbosa dos Santos. 3ª ed. São Paulo: Makron Books, 1995. 1090 p.

SOUZA, Rodrigo A.; ARAKAKI, Reginaldo. Qualidade de software : O que são design patterns? **Revista Integração ↔ Pesquisa ↔ Extensão**, São Paulo, nº28, p.41-48, fev. 2002.

USAOLA, M.; VELTHUIS, M.; GONZÁLEZ, F. **Reflective CRUD (RCRUD: Reflective Create, Read, Update & Delete)**. In: European Conference on Pattern Languages of Programs, 6., 2001, Irsee :Alemanha. 2001. 15 p.

VAROTO, Ane Cristina. **Visões em Arquitetura de Software**. 2002. 106 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo. 2002.

SCOPUS Tecnologia. **VISÕES de Arquitetura** : Norma de Referência Interna de Desenvolvimento (Adaptado da norma ISO 9126). 2001.

REFERÊNCIAS OBTIDAS NA INTERNET

APPLETON, B. **Patterns and Software**: Essential Concepts and Terminology. 2000. Disponível em <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html#UsefulUseableUsed>, Acesso em 28 jan. 2004

CANFORA, Gerardo; CIMITILE, Aniello. **Software Maintenance**. 2000. Disponível em <http://citeseer.nj.nec.com/rd/59996062%2C495540%2C1%2C0.25%2CDownload/ftp%3AqSqqSqcs.pitt.eduqSqchangqSqhandbookqSq02.pdf>. Acesso em 10 jan. 2004

CARRIÉRE, Jeromy; WOODS, Steven G.; KAZMAN, Rick;. **Software Architectural Transformation**. Software Engineering Institute, 1999. Disponível em <http://www.sei.cmu.edu/reengineering/Wcre99.pdf >. Acesso em 19 jan. 2004

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. **The FAMOOS Object Oriented Reengineering Handbook**, University of Bern. 1999. Disponível em <http://www.iam.unibe.ch/~famoos/handbook/4handbook.pdf>. Acesso em 19 jan. 2004

GARLAN, David. **Software Architecture**. 2001. Disponível em < http://www-2.cs.cmu.edu/afs/cs/project/able/ftp/encycSE2001/encyclopedia-dist.pdf>. Acesso em 16 jan. 2004

GOLD, N. **The Meaning of "Legacy Systems"**. 1998. Department of Computer Science, University of Durham. Disponível em <http://www.dur.ac.uk/CSM/SABA/legacy-wksp1/meaning.html>. Acesso em 21 jan. 2004

IDEF0 Overview. 2000. Disponível em <http://www.idef.com/idef0.html>. Acesso em 09 fev. 2004

KAZMAN, Rick; WOODS, Steven G.; CARRIÉRE, Jeromy. **Requirements for Integrating Software Architecture and Reengineering Models: CORUM II**. Software Engineering Institute, 1998. Disponível em <www.sei.cmu.edu/reengineering/wcre98.pdf>. Acesso em 11 fev. 2004

MIAH, S. **Critique of the Object Oriented Paradigm: Beyond Object-Orientation**. 1997. Disponível em <http://members.aol.com/shaz7862/critique.htm>. Acesso em 28 jan. 2004

MONROE, Robert et al. **Stylized Software Architecture**. 1996. Disponível em <http://people.eecs.ukans.edu/~palexand/teaching/eecs800/files/ObjPatternsArch-ieee97.pdf>. Acesso em 16 jan. 2004

NOMURA, Luzia; SPÍNOLA, Mauro. **Qualidade e Melhoria do Processo de Manutenção de Software**. [200-?]. Disponível em <http://www.unip.br/websites/posgraduacao/engproducao/artigos/doc-pdf/Luzia_Spinola.pdf>. Acesso em 10 jan. 2004

RENAISSANCE of Legacy Systems. 1998. Disponível em <http://www.comp.lancs.ac.uk/projects/renaissance/project/Documents4.html>. Acesso em 22 jan. 2004

SADOSKY, D.; DORNELLA-DORDA, S.; **Three Tier Software Architecture**. Software Engineering Institute, 2000. Disponível em < http://www.sei.cmu.edu/str/descriptions/threetier_body.html>. Acesso em 09 fev. 2004

SALINGAROS, Nikos A. **Some Notes On Christopher Alexander**. 1997. Disponível em <http://www.math.utsa.edu/sphere/salingar/Chris.text.html>. Acesso em 30 dez. 2003

SANZ, C. **Objetos Contra Complejidad**. 1995. Disponível em <http://www.well.com/user/ritchie/cesar.html>. Acesso em 28 jan. 2004

STEINMAN, J. **The Overselling of Object Technology, or How To Fail On Your First Object Project**. 1999. Disponível em <http://www.bytesmiths.com/pubs/9209Overselling.html>. Acesso em 28 jan. 2004

VERHOEF, Chris. **Software Evolution: A Taxonomy**. [200-?]. Disponível em <http://www.swebok.org/stoneman/version_0.1/KA_Description_for_Software_Evolution_and_Maintenance (version_0_1).pdf>. Acesso em 10 jan. 2004

APÊNDICE A – PADRÕES UTILIZADOS NO ESTUDO DE CASO

A seguir, uma descrição dos padrões utilizados no estudo de caso.

Padrão Three Tier Architecture

Intenção e Objetivo

Fornecer um método para desacoplamento entre objetos de negócio, persistência e interfaces externas.

Motivação

- Permite que a aplicação seja descrita em termos do domínio do negócio.
- Suprime referências entre objetos de persistência e interface com o usuário, promovendo o reuso destes objetos.
- Permite o intercâmbio entre diferentes modos de apresentação através de paradigmas diferentes de interfaces com o usuário, como páginas dinâmicas, ferramentas de desenvolvimento rápido ou linhas de comando.

Aplicabilidade

O padrão é aplicável em sistemas onde não existe uma separação clara de lógica de negócio, interface com o usuário e persistência de dados. Este problema pode ser resolvido a partir da aplicação do padrão.

Estrutura

A estrutura do padrão consiste em separar a lógica da aplicação dos componentes de negócio das camadas restantes. A estrutura pode ser visualizada na Figura 1:

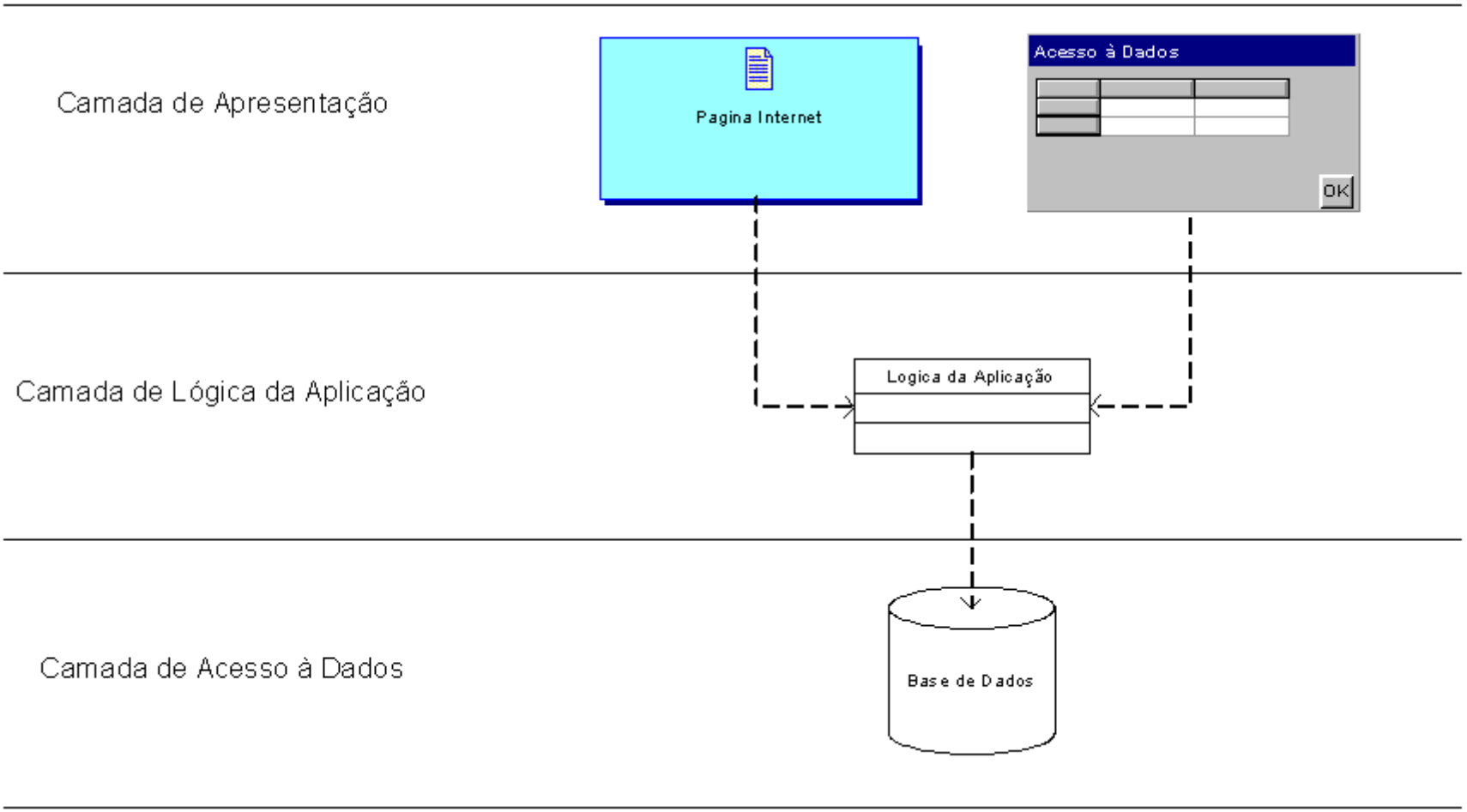

Figura 1 - Estrutura do padrão 3 Tier Architecture

Participantes

- Camada de Apresentação: Representa os objetos que irão acessar os objetos de negócio; são os componentes visíveis ao usuário. Não acessam a camada de acesso a dados.
- Camada de Lógica de Aplicação: Representa objetos dentro do domínio de negócio. Os objetos desta camada são acessados pela camada de apresentação, porém a recíproca não é verdadeira. Acessa somente a camada de acesso a dados.
- Camada de Acesso à Dados: É a camada responsável por materializar e gravar objetos de negócio. É a única camada com permissão para acesso ao meio de persistência.

Conseqüências

- Permite a reutilização de componentes em diversos projetos, devido ao desacoplamento entre as camadas.
- Obriga um projeto mais criterioso dos componentes, pois exige um certo cuidado na separação entre lógica de negócios e camada de apresentação

Padrões Relacionados

Para criar um desacoplamento ainda maior entre os componentes de negócio e a camada de aplicação pode ser utilizado o padrão *Abstract Factory* (Gamma et al., 1995), que possibilita uma abstração sobre a instanciação de classes de negócio, eliminando referências explícitas às classes de negócio. Da mesma maneira, para aumentar o desacoplamento entre os objetos de negócio e a camada de persistência, o padrão *Bridge* pode ser utilizado (Souza e Arakaki, 2002).

Padrão RCRUD

Intenção e Objetivo

Providenciar o acesso entre classes de negócio e suas respectivas representações no meio de persistência.

Motivação

- Diminui o esforço de manutenção visto que o código principal de acesso à base de dados permanece recluso na classe principal.
- Permite a reutilização em outros projetos através de simples "copiar-e-colar" de código fonte.
- Permite que a aplicação seja facilmente adaptada a outros tipos de transformações entre o domínio de negócio e o esquema relacional.
- Não introduz complexidade a ponto de degenerar a desempenho do sistema.

Aplicabilidade

O padrão é aplicável em sistemas de três camadas que acessem bases de dados relacionais, utilizando alguma linguagem que possua o mecanismo de *Reflection* (*e.g.*, Java).

Estrutura

O padrão RCRUD é formado por uma classe denominada RCRUD, que se encarrega de todos os acessos à base de dados. As classes do sistema devem extender suas

funcionalidades e especializá-las caso a funcionalidade *default* não seja suficiente. A Figura 2 ilustra a sua utilização:

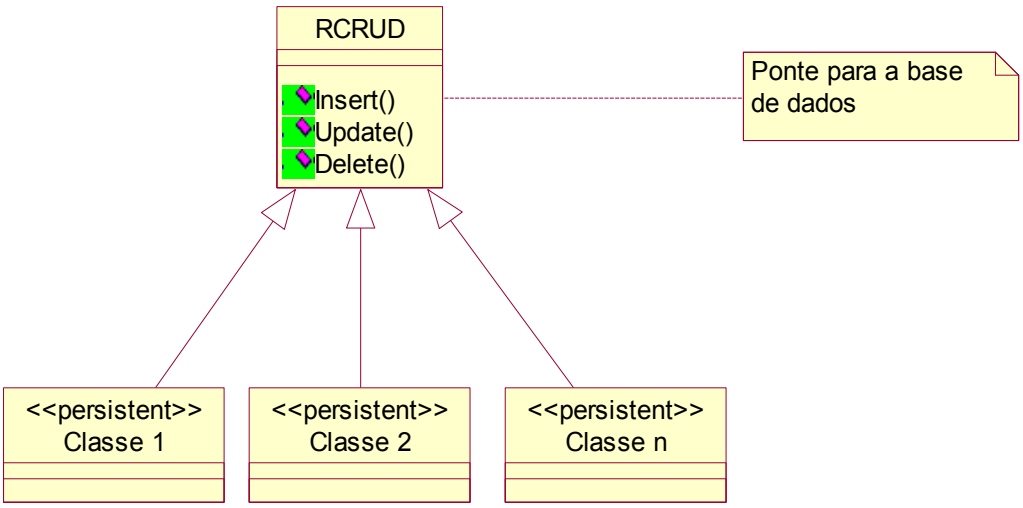

Figura 2 - Estrutura do padrão RCRUD

Participantes

- A classe RCRUD, que através da *Reflection*, fornece métodos necessários em tempo de execução para as classes persistentes;
- Classe 1, Classe 2, Classe n são classes que necessitam do mecanismo de persistência e extendem a classe RCRUD;
- A ponte para a base de dados pode ser algum mecanismo que já seja utilizado no sistema para acesso à base de dados, como JDBC, conexão nativa, etc.

Conseqüências

- Habilita um desenvolvimento rápido de classes que necessitam persistência. Para utilização, basta a extensão da classe RCRUD pela classe cliente;
- Necessita um forte controle sobre nomenclatura das classes. Como está sendo utilizado o mecanismo de *Reflection*, os atributos das classes devem ser idênticos aos atributos das tabelas, acrescentando-se o prefixo “m”.

- Facilita a manutenção, dado que apenas a classe RCRUD possui código de acesso à base de dados.

Implementação

Conforme explicado, o padrão RCRUD utiliza uma característica de linguagem chamada *Reflection*, cujo detalhamento foge do escopo do presente trabalho. Maiores detalhes sobre a implementação podem ser encontrados em Usaola et al. (2001).

Padrões Relacionados

O padrão utiliza os padrões *Template* e *Pure Fabrication* (Gamma et al., 1995) para fornecer os métodos necessários para acesso à base e delegar responsabilidades para cada classe persistente.

APÊNDICE B – DIAGRAMAS DO PROCESSO DE TRANSFORMAÇÃO DE ARQUITETURAS

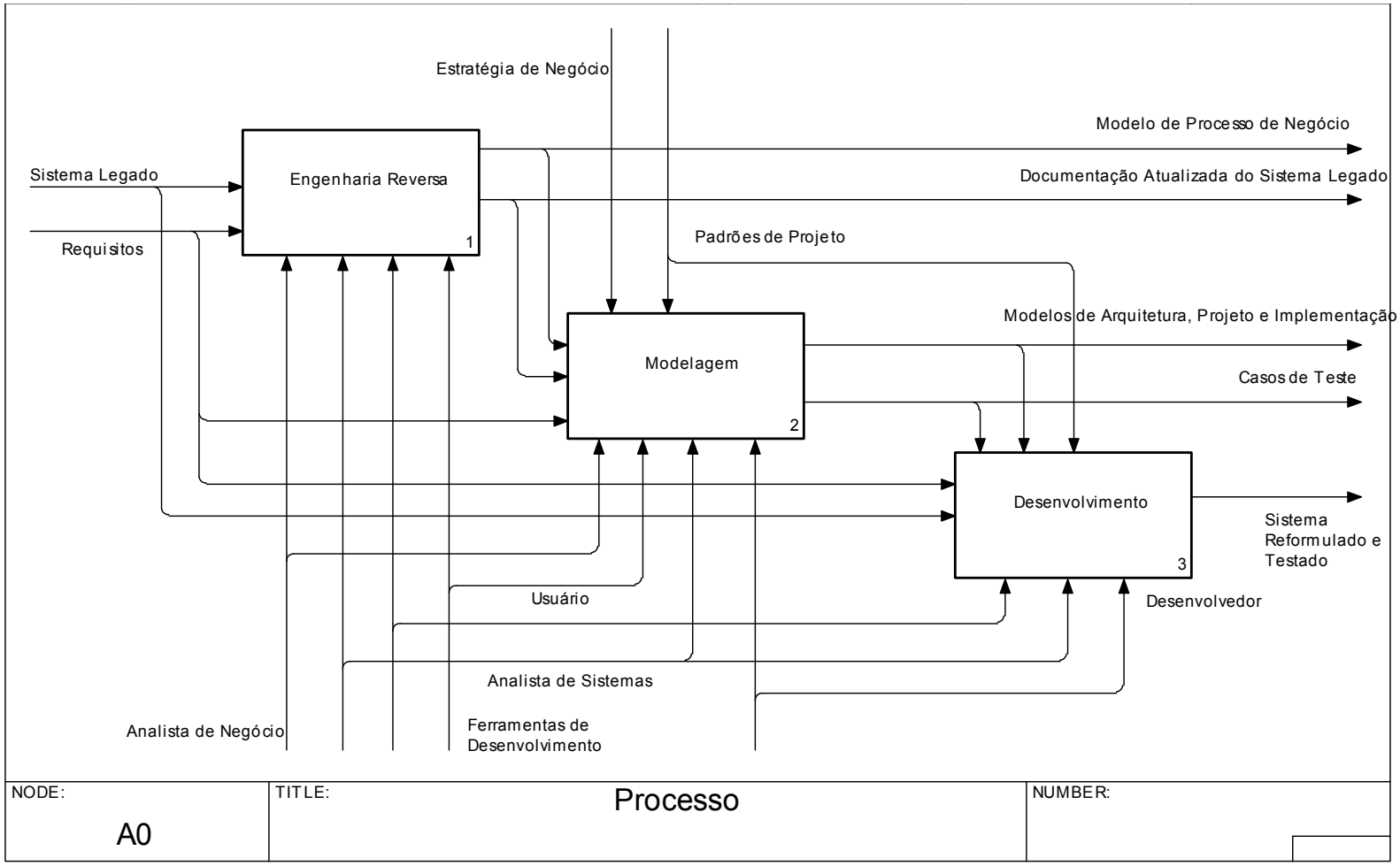

Figura 1 – Diagrama em IDEF0 do processo

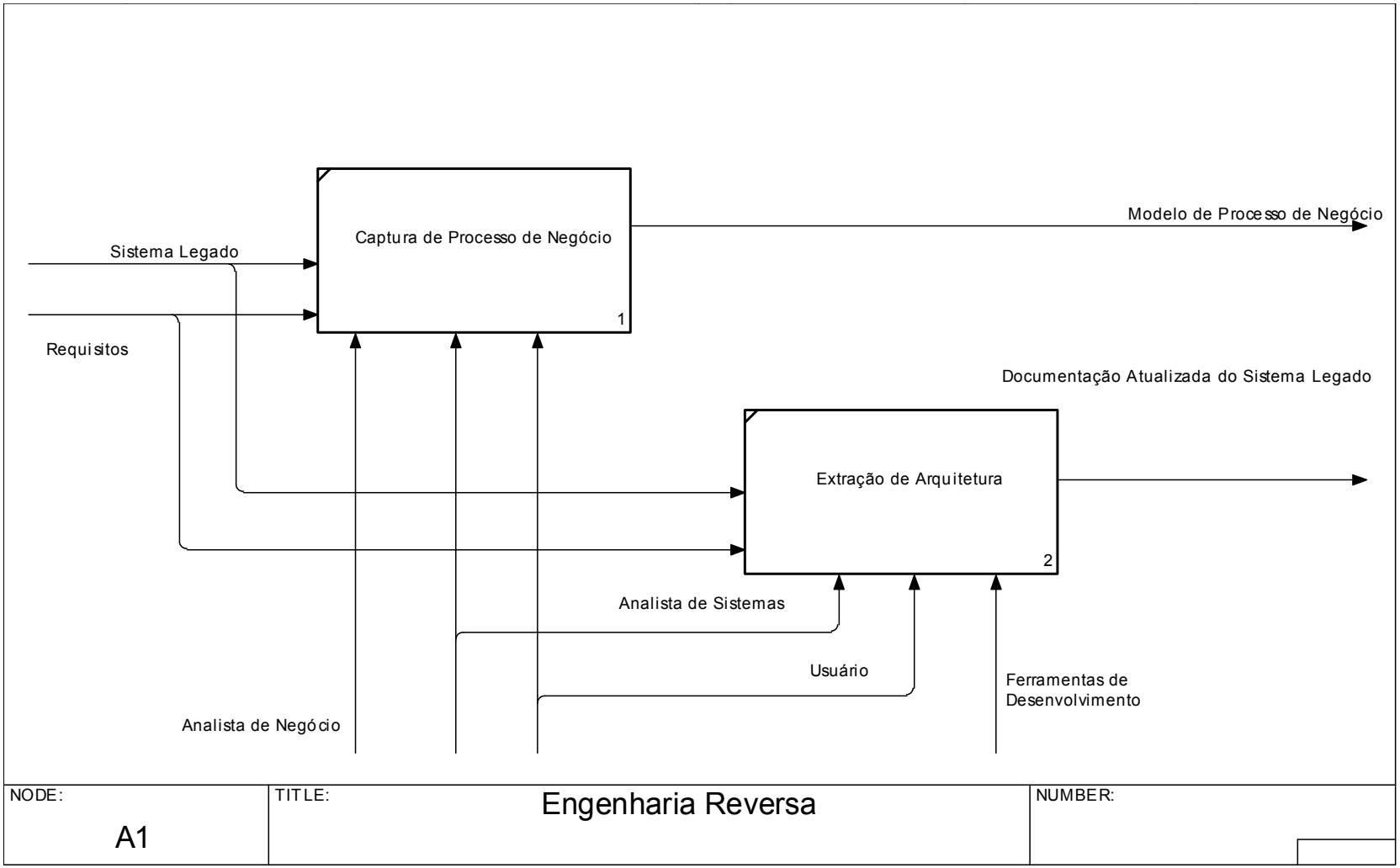

Figura 2 – Diagrama em IDEF0 da fase de Engenharia Reversa

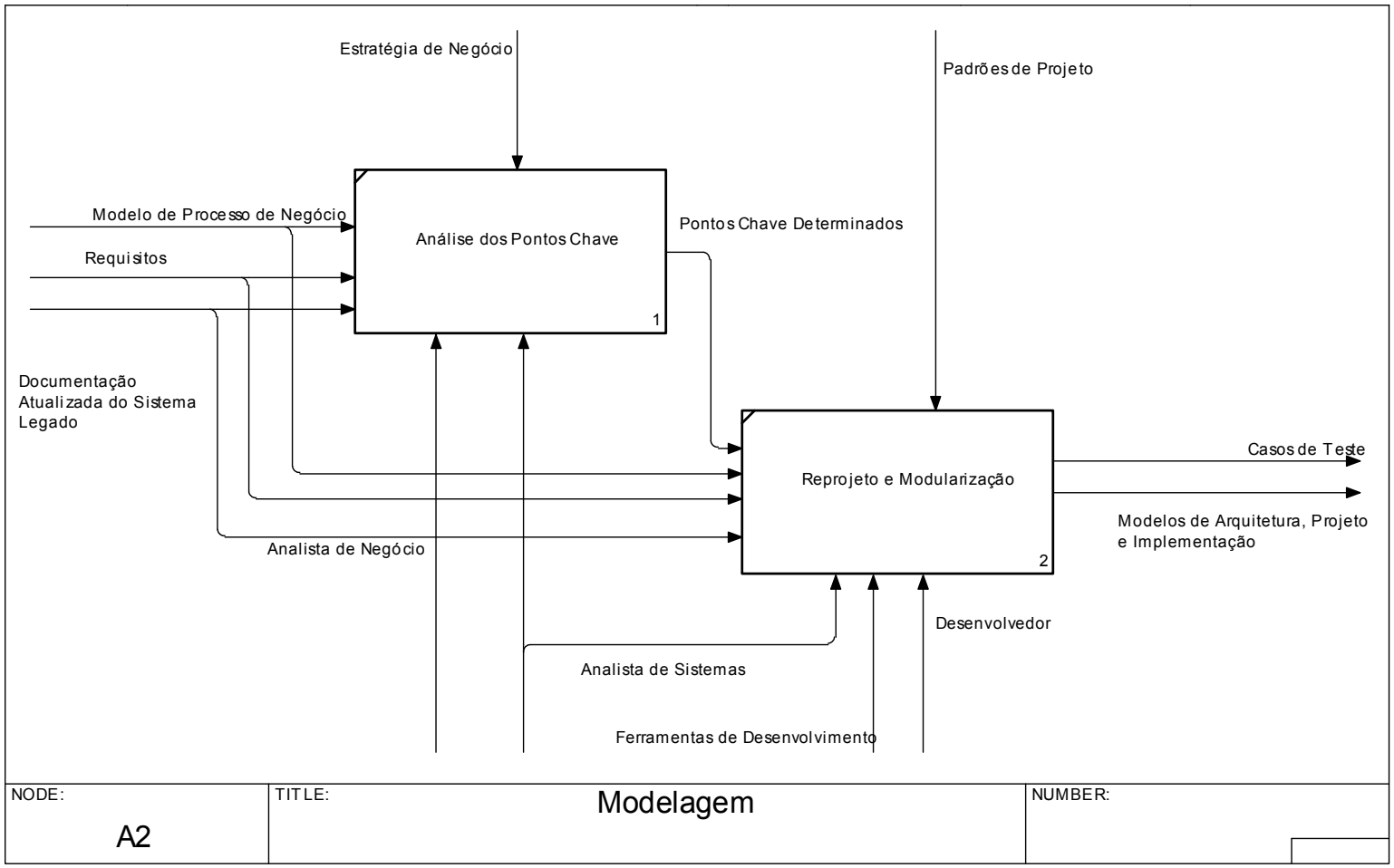

Figura 3 – Diagrama em IDEF0 da fase de Modelagem

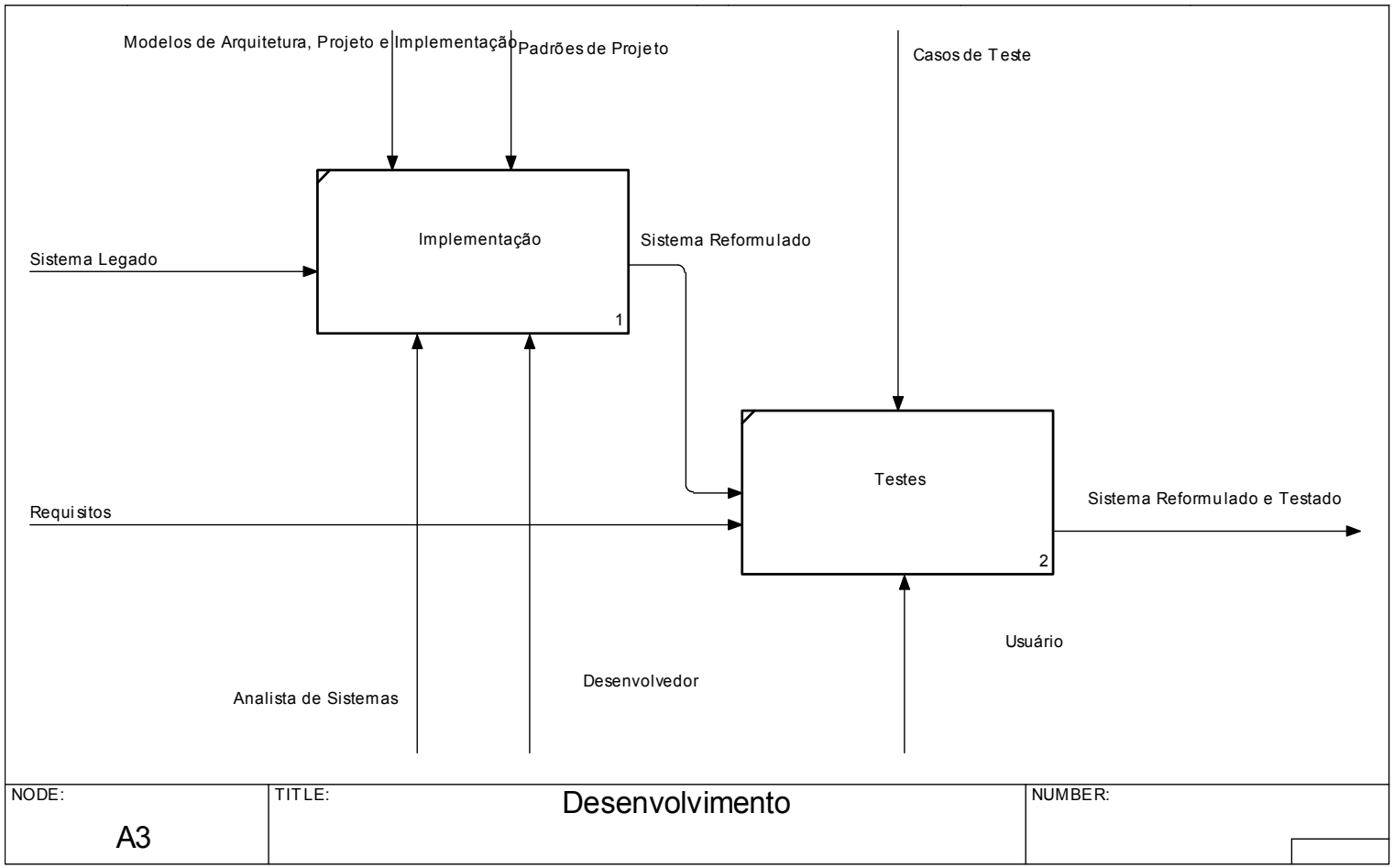

Figura 4 – Diagrama em IDEF0 da fase de Desenvolvimento

GLOSSÁRIO

FRAMEWORK – Biblioteca de classes e interfaces, desenvolvida com o intuito de ser uma base customizável para o desenvolvimento de aplicações. A principal motivação para a utilização de *frameworks* é a eliminação de programação de funções comuns por parte do desenvolvedor ou projetista do sistema[16].

MAINFRAME – Computador de grande porte. Remonta à década de 60, quando todos os computadores eram chamados por este nome. Pode suportar centenas de terminais "burros" e, geralmente, utiliza pequenos computadores para se conectarem com redes externas[17].

N-TIER – Modelo criado com a intenção de flexibilizar e reutilizar aplicações. A aplicação é dividida em camadas e cada camada tem sua responsabilidade definida. Os exemplos mais comuns dessa abordagem são o 2-tier, envolvendo a camada de apresentação e acesso a dados, e a 3-tier, envolvendo as camadas de apresentação, negócio e acesso a dados[18].

WORKFLOW – Representa os aspectos operacionais de um procedimento: sua estrutura de tarefas, a ordem de execução e sincronização destas tarefas e mecanismos de rastreamento para medição e controle das mesmas[19].

WORKSHOP – Seminário, grupo de discussão.

---

[16] LARMAN, Craig. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design. Prentice Hall PTR, 1998.

[17] SADOSKY, Darleen. **Client/Server Software Architecture – An Overview**. Software Engineering Institute, 1997. Disponível em <http://www.sei.cmu.edu/str/descriptions/clientserver_body.html>. Acesso em 08 jul. 2004

[18] FOWLER, Martin. **Analysis Patterns**: Reusable Object Models. Addison Wesley, 1996. 384 p.

[19] MANOLESCU, Dragos-Anton. **Micro-workflow: A workflow architecture supporting compositional Object Oriented Software Development**. 2001. 221 f. Tese (Doutorado em Ciência da Computação) –Universidade de Illinois, Urbana-Champaign. 2001.