MAC0499-Trabalho de Formatura – Monografia

USP - Universidade de São Paulo

Instituto de Matemática e Estatística

Bacharelado em Ciência da Computação

Técnicas de Reuso de Software aplicados na elaboração de Arquiteturas Corporativas

Tipo de Trabalho: Estágio Supervisionado

Aluno: Alexandre Eiki Onishi

Supervisor: Prof. Siang Wun Song

Sumário

Parte I

Introdução

Capítulo 1: Introdução

Capítulo 1

Introdução

1.1 **Visão Geral**

O avanço da internet tem motivado o crescimento de uma nova geração de aplicações baseada na web, que combinam navegação e interatividade num grande espaço de documentos heterogêneos. A web se mostrou como um dos meios mais efetivos e atrativos meios de divulgação, negociação e disponibilização de bens e serviços.

Independente de contexto, o projeto de aplicações ainda não é um processo totalmente definido, como se sabe fazer software é um processo lento e custoso, somado a isso a demanda por software aumentou muito nas ultimas décadas, necessita-se mais, com prazos de entrega cada vez mais curtos e exigentes, ao mesmo tempo em que se exige qualidade e confiabilidade no produto final, e é dentro deste contexto que surge o processo de desenvolvimento baseado em reuso de software.

A reutilização de software é uma das áreas da engenharia de software que propõe um conjunto sistemático de processos, de técnicas e de ferramentas para obter produtos com alta qualidade e que sejam economicamente viáveis. A idéia do reuso é evitar retrabalho no desenvolvimento de um novo projeto, sempre levando em consideração trabalhos anteriores, fazendo com que soluções previamente desenvolvidas sejam aproveitadas e implementadas em novos contextos. Dessa forma, tem-se melhores produtos em um menor intervalo de tempo e um aumento da qualidade, pois muitas dessas soluções já foram testadas e validadas anteriormente. O termo reuso pode ser considerado uma denominação genérica para uma série de técnicas utilizadas, que vão desde a etapa de modelagem de um projeto até a implementação.

Atualmente existem várias técnicas de reuso como *frameworks,* arquiteturas orientadas a serviços (SOA), engenharia de software baseada em componentes, entre outras. Esta dissertação focará em apresentar somente as técnicas de reuso no contexto de *frameworks* e arquiteturas orientadas a serviços.

Ao desenvolver sistemas, o desenvolvedor depara-se com inúmeros desafios. Além de ter que entender sobre regras de negócio do seu domínio de aplicação, ele tem que lidar também com aspectos ligados à infra-estrutura da aplicação como segurança, serviços remotos, persistência de dados, validação entre outros.

Desse modo desenvolver software já é complicado por ter que entender o domínio da aplicação e não deveria ser mais difícil ainda por tais questões de infra-estrutura.

Os Frameworks são uma técnica de reuso que surge como uma solução para este problema, acelerando o desenvolvimento de software. A idéia é permitir que o desenvolvedor se foque mais nos aspectos funcionais do domínio da aplicação, cabendo ao frameworks a tarefa de fornecer a infra-estrutura técnica necessária para o tratamento destas questões técnicas, fornecendo ao desenvolvedor uma maneira de lidar com tais questões de baixo nível como transações, segurança, e serviços remotos, a partir de uma visão de mais alto nível, tornando o processo de desenvolvimento mais fácil e rápido ao desenvolvedor da aplicação.

Arquiteturas orientadas a serviços (SOA) tem como objetivo propor um novo padrão arquitetural para projetar e construir softwares complexos a partir da reutilização de componentes de software pré-construídos e testados, chamados de serviços. Nesta abordagem, as aplicações são vistas como um conjunto de pequenos blocos denominados serviços, desenvolvidos sob um determinado padrão a fim de prover maior reusabilidade, facilitar sua manutenção, e se encontram acessíveis dentro de uma rede de computadores. Devem prover interfaces bem definidas por meio das quais os serviços são integrados uns aos outros, formando um sistema de software baseado em serviços.

1.2 Objetivos

Este trabalho tem como objetivo estruturar um conhecimento básico na área de reuso de software enfatizando técnicas de reuso, mais especificamente *frameworks* e arquiteturas orientada a serviços (SOA), a fim de que fornecesse os conhecimentos necessários para melhorar o processo de desenvolvimento de um software de automação bancaria implementado em Java e em futuras aplicações da empresa. Entende-se que o reuso de software é aplicável no desenvolvimento de sistemas a partir de soluções previamente desenvolvidas e armazenadas, ao invés de desenvolver um projeto desde o início como se costuma fazer em abordagens convencionais.

1.3 Motivações

Quando se trabalha na indústria de software busca-se sempre uma maior qualidade e produtividade no processo de desenvolvimento do produto. Existem muitas soluções existentes a fim de melhorar este processo de desenvolvimento, como métodos orientados a objetos, engenharia reversa, ferramentas *CASE*, qualificação de software, entre outras.

O alto dinamismo com que as tecnologias atuais vêm apresentando, dificulta o desenvolvimento de software onde a rapidez de entrega juntamente com um produto de

qualidade é cada vez mais exigida. Dessa forma, isto tudo faz com que o reuso de software seja uma opção para se manter ágil com qualidade.

Entretanto os ganhos de produção com a prática do reuso não são imediatos e nem em curto prazo, pois desenvolver para reuso requer uma nova maneira de se trabalhar entre os desenvolvedores e requer mudanças técnicas, que implicam em custos e tempo para que os efeitos de qualidade e produtividade almejados apareçam.

1.4 Organização do texto

Os capítulos seguintes fazem uma descrição básica do que é o reuso de software, desafios, como aplicar o reuso em processos de desenvolvimento de software, e mostra técnicas de reuso de software.

As técnicas de reuso de software consistem em basicamente no reaproveitamento de partes previamente desenvolvidas, qualificadas e armazenadas. Existem várias técnicas de reuso como programação orientada a objetos, padrões de projeto, *frameworks*, engenharia de software baseada em componentes, entre outros. Entretanto esta monografia apresentará somente as técnicas de reuso baseada em *frameworks* e arquiteturas orientadas a serviços.

O capítulo sobre atividades realizadas descreve sobre como as técnicas apresentadas nesta dissertação foram usadas no desenvolvimento de uma arquitetura de *software* modular e multicamadas, o sistema *Opus ibank*. A parte final desta monografia contém um relato pessoal de experiências obtidas no ambiente de trabalho do estágio, e sua relação com o curso do bcc.

Parte II

Conceitos, Tecnologias Estudadas e Resultados

Capítulo 2: Conceitos de Reuso de Software

Capítulo 3: Frameworks: Reuso e Aplicações

Capítulo 4: Arquiteturas Orientadas a Serviços (SOA)

Capítulo 5: Atividades realizadas, Resultados obtidos e Conclusões

Capítulo 2

Conceitos de Reuso de Software

2.1 Introdução

A preocupação e os esforços empregados para melhorar os processos de desenvolvimento de software, buscando aumento da produtividade, qualidade e redução de custos e esforços evidenciam novas perspectivas para o processo de desenvolvimento de software.

O reuso é uma solução a ser considerada, pois provoca um impacto na qualidade, produtividade e custo do software.

O reuso de software surgiu inicialmente por uma necessidade de economizar recursos de hardware. Há algumas décadas atrás não havia memória suficiente nos dispositivos para armazenar muitas rotinas e então, foi observado que era possível executar tarefas similares através de uma única sub-rotina parametrizada, e assim não desperdiçar o uso da escassa memória disponível.

Com a evolução dos componentes de hardware e a conseqüente redução de preço, o reuso de software mudou de foco. Desde então o principal objetivo do reuso é economizar recursos humanos no desenvolvimento e não mais de hardware, até porque recursos humanos se tornaram muito mais dispendiosos.

Este capítulo trata então de apresentar os conceitos básicos sobre reuso de software encontrados na literatura. Primeiramente, são revistas algumas definições sobre reuso de software. Logo após são apresentados benefícios e desafios decorrentes da adoção de reuso no processo de desenvolvimento, abordagens de reuso e por fim o capítulo termina com uma descrição de como o reuso pode ser inserido no processo de produção de um software através da análise de dois modelos de processos, o modelo em cascata e o modelo espiral, na perspectiva de reuso de software.

2.2 Definições

Reuso de software é o uso de conceitos, produtos ou soluções previamente elaboradas ou adquiridas para criação de um novo software, visando melhorar significantemente a qualidade e a produtividade do software. Reusar um produto significa poder reusar partes de um sistema desenvolvido anteriormente como: especificações, módulos de um projeto, arquitetura e código fonte.

Reusabilidade é uma medida da facilidade em se utilizar os conceitos e produtos em novas situações.

2.3 Benefícios e Desafios

A construção de soluções flexíveis, ou seja, soluções capazes de se adaptar a novos contextos de utilização, aliada a um processo de desenvolvimento com alta produtividade requer conhecimento e disciplina. Os principais benefícios da reutilização aceita pela maioria dos autores, são destacados abaixo:

- Aumento da produção com a redução no esforço do desenvolvimento.
- Redução dos custos e do prazo de entrega, pois se o esforço de desenvolvimento diminui, logo o tempo de entrega também diminui e a quantidade de homens hora necessária a ser pago também.
- Como as soluções aplicadas foram anteriormente testadas e validadas, a probabilidade de que esteja correta é ainda maior, portanto temos um aumento da qualidade do produto final.
- Padronização dos produtos desenvolvidos pela empresa, pois como as soluções reusáveis foram desenvolvidas segundo uma padronização pré-definida, o reuso em um sistema provoca conseqüente padronização e agilidade na manutenção das aplicações devido a esta padronização da arquitetura.

Embora os benefícios sejam muitas, muitos também são os desafios encontrados na adoção do reuso, como por exemplo:

- A implantação de práticas de reuso requer mudança de mentalidade das pessoas com relação ao processo de desenvolvimento deve-se desenvolver para o reuso e com o reuso. A indiferença com relação a reutilizar tem origem em razões como falta de incentivo, dúvida com relação aos benefícios da reutilização, falta de apoio por parte da gestão, entre outros.
- O fato da não existência de partes previamente elaboradas para a reutilização ou não é um dos principais fatores para o impedimento.
- Dificuldade em compreender uma parte reusável, seja pela falta de documentação ou complexidade do mesmo, é também um fator para desconsideração do reuso.

2.4 Abordagens de Reutilização

No processo de desenvolvimento de um sistema, pode se aplicar o reuso de software em vários momentos. Existe a possibilidade de se reusar idéias, especificações, projetos, códigos-fonte e outros produtos nas diversas fases do processo de desenvolvimento, conforme pode ser visto na figura 1 abaixo:

Figura 1 – Tipos de reuso

2.4.1 Reuso Vertical

Reuso vertical é o que ocorre dentro de um mesmo domínio de aplicação. O objetivo é derivar um modelo genérico para ser usado dentro de um único domínio de aplicação na criação de novos sistemas, este tipo de reuso é o que ocorre em fábricas de software.

2.4.2 Reuso Horizontal

O reuso horizontal tem como meta, a utilização de partes dentro de diferentes domínios de aplicação. Como exemplo temos bibliotecas funções matemáticas e manipulação de string, bibliotecas para construção de interfaces gráficas, entre outras. A característica principal é a sua utilização em diferentes domínios, diferente do que ocorre no reuso vertical.

2.4.3 Reuso planejado

O reuso planejado é a prática sistemática e formal do reuso onde diretrizes e procedimentos são definidos e seguidos Requer um alto grau de investimento e comprometimento gerencial, exigindo uma mudança significativa no processo de desenvolvimento de software. O exemplo que caracteriza este reuso são os modelo de maturidade de software (*Capability Maturity Model* - CMM) que algumas fábricas de software buscam visando uma prova de qualidade de seus softwares produzidos.

2.4.4 Reuso Composicional

O reuso composicional é a utilização de componentes existentes como blocos para a construção de um novo sistema. A característica principal é que um novo sistema é

construído a partir da composição de componentes existentes, como exemplo temos os frameworks de componentes como Java Server Faces que apresentam um conjunto de componentes para construção de interfaces com o usuário e a tecnologia de componentes EJB para sistemas distribuídos.

2.4.5 Reuso Baseado em Geradores de Código

Esta abordagem consiste no reuso no nível de especificação de um sistema, através do emprego de geradores de código ou de aplicações. Como exemplos temos as ferramentas CASE e ferramentas UML.

2.4.6 Reuso Caixa Branca

No reuso caixa branca existe a necessidade que a implementação do componente de software a ser reusado seja exposta de alguma forma. Em linguagens orientadas a objetos como Java e C++, é muito comum o uso de herança para se atingir o reuso, modificando e adaptando o componente. Neste tipo de reuso é preciso conhecer a implementação de algum componente de software que fará parte do reuso.

2.4.7 Reuso Caixa Preta

O reuso caixa-preta visa eliminar a necessidade do desenvolvedor de um conhecimento da implementação de algum componente de software que fará parte do processo de reuso. Em vez disso, o reuso caixa-preta se dá através da descrição de interfaces ou contratos bem definidos que devem ser respeitados pela implementação a ser elaborada. O esforço sempre é usado na nova implementação e nunca ocorre um desperdício tentando entender implementações de terceiros.

2.4.8 Reuso de Códigos Fonte

Este é o tipo de reuso mais utilizado na prática, o reuso de código. A maioria das ferramentas de reuso e métodos são voltados para este tipo de reuso.

2.4.9 Reuso de Projetos

Reuso de projetos oferecem um retorno maior que o reuso de código. Quanto maior o nível do componente, maior ganho se obtém, dado que os subprodutos gerados também serão componentes. Neste sentido, ao se reusar projetos, o reuso de código ou de módulos executáveis é uma conseqüência direta.

O reuso de projetos é realizada com bastante freqüência em orientação a objetos, os vários trabalhos de padrões de projetos refletem a praticidade deste reuso.

2.4.10 Reuso de Especificação

Da mesma maneira que ocorre com projetos, quando se reutiliza uma especificação, tem-se, como conseqüência direta o reuso de projeto e do código fonte.

2.5 Processos de desenvolvimento com reuso de software

Após visto os princípios básicos de reuso de software e os principais tipos de reuso existentes, resta saber como aplicar o reuso em processos de desenvolvimento de software. A seguir dois modelos de processos o modelo cascata e o modelo espiral são analisados com a perspectiva de reuso de software.

2.5.1 Modelo Cascata com Reuso de Software

O modelo cascata tem como característica principal a seqüencialidade das atividades, cada etapa transcorre completamente e seus produtos são vistos como entrada para a etapa seguinte, de forma que uma etapa só poderá ter início quando a anterior tiver terminado e o software é entregue ao final desta seqüência linear.

- Análise e Definição de Requisitos

Nesta etapa, estabelecem-se os requisitos do produto que se deseja desenvolver, quais são suas funcionalidades, limitações e objetivos do software.

A reutilização nesta primeira etapa do modelo por meio da utilização de *patterns* pode facilitar e agilizar o ciclo, partes importantes como documentação e estudos de viabilidade podem ser comparados com os já existentes e utilizados com sucesso em projetos anteriores. Vale ressaltar nesta etapa um tipo de reuso de software que pode auxiliar, o **reuso de projeto**, com foco em utilização de conhecimentos em desenvolvimentos anteriores.

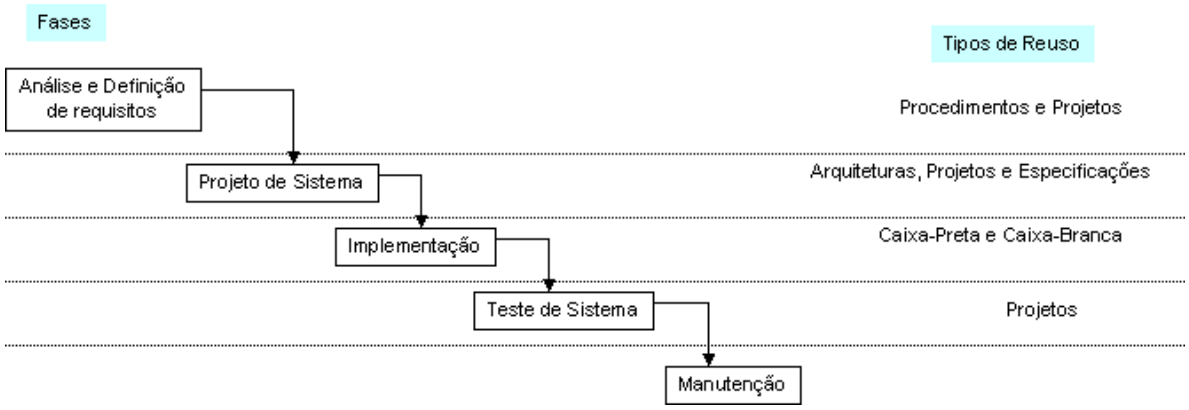

Figura 2 – Mostra o modelo Cascata.

- Projeto de Sistema

O projeto de sistema é a etapa em que se define como vai ser a arquitetura do software e a caracterização das interfaces do sistema. A etapa de projeto representa os requisitos de uma forma que permita a codificação do produto na etapa seguinte.

Nesta etapa podemos utilizar vários tipos de reuso abordados anteriormente neste trabalho, como o **reuso de arquiteturas** relacionado com a arquitetura de software, **reuso de projetos** e **reuso de especificações**, pois nesta etapa de projeto irá ser especificado o requisito de uma forma que permita a codificação do produto.

- Implementação

Esta é a etapa em que são criados os programas. Nesta fase onde a ênfase está na codificação seguida de testes, pode se utilizar o reuso de código.

O reuso de código pode ser em muitos dos casos onde conseguimos a maior produtividade no reuso de software. Testes já utilizados anteriormente são também importantes nesta fase, pois certificam mais rapidamente os módulos gerados pelos componentes. Reuso caixa preta e reuso caixa branca poderão ser aplicados nesta fase também.

- Teste de Sistema

Concluída a etapa de codificação e de testes unitários, começa a fase de teste do sistema. Nesta fase os *patterns* de testes serão muitos importantes nesta fase, pois os *patterns* de testes dentro de um reuso sistemático tornam-se, com o decorrer do tempo, cada vez mais eficazes, pois existe uma reciclagem aumentando a cada projeto o nível de maturidade dos componentes reusáveis.

- Manutenção

Esta etapa consiste na correção de erros que não foram previamente detectados. Vale ressaltar nesta etapa que a revisão e a modificação, aumentam a qualidade de um componente voltado para o reuso, este ciclo é importante, pois assim podemos melhorar os componentes num ciclo de vida evolutivo.

2.5.2 Modelo Espiral com Reuso de Software

O modelo em espiral é um modelo de processo de software que prevê prototipação, desenvolvimento evolutivo e cíclico. Sua principal característica é guiar o processo de desenvolvimento com base em análise de riscos e um planejamento que é realizado durante toda evolução do desenvolvimento. São exemplos de riscos: pessoas que

abandonam a equipe de desenvolvimento, ferramentas que não podem ser utilizadas, entre outras.

O modelo espiral descreve um fluxo de atividades cíclico e evolutivo constituído de quatro quadrantes. Vamos descrevê-los utilizando uma abordagem voltada ao reuso de software.

- Primeiro quadrante

No primeiro quadrante devem ser determinados os objetivos, soluções alternativas e restrições. Temos neste primeiro ciclo a possibilidade de implementação e utilização de reuso de software, a busca de soluções e restrições podem ser obtidas através de experiências de projetos anteriores ou mesmo de projetos em desenvolvimento, portanto podemos aplicar neste primeiro quadrante o reuso de projetos.

- Segundo quadrante

No segundo quadrante devem ser analisados devem ser analisados os riscos de das decisões do estágio anterior. Durante este estágio podem ser construídos protótipos ou realizarem simulações de software. Os Protótipos podem ser construídos a partir de procedimentos realizados anteriores, entretanto a importância deste quadrante é porque podemos alimentar o processo de desenvolvimento com novas e atualizadas informações.

- Terceiro quadrante

O terceiro quadrante consiste nas atividades da fase de desenvolvimento, incluindo especificação, design, codificação e verificação. Nesta fase podemos utilizar vários tipos de reuso de software. O **reuso de idéias** para uma busca de soluções genéricas para uma classe de problemas, **reuso de código** para auxiliar a fase de implementação, **reuso composicional** para desenvolver partes de software a partir da composição de componentes desenvolvidos e validados anteriormente e o **reuso baseado em geradores de código** que consiste basicamente na utilização de geradores de códigos e aplicações.

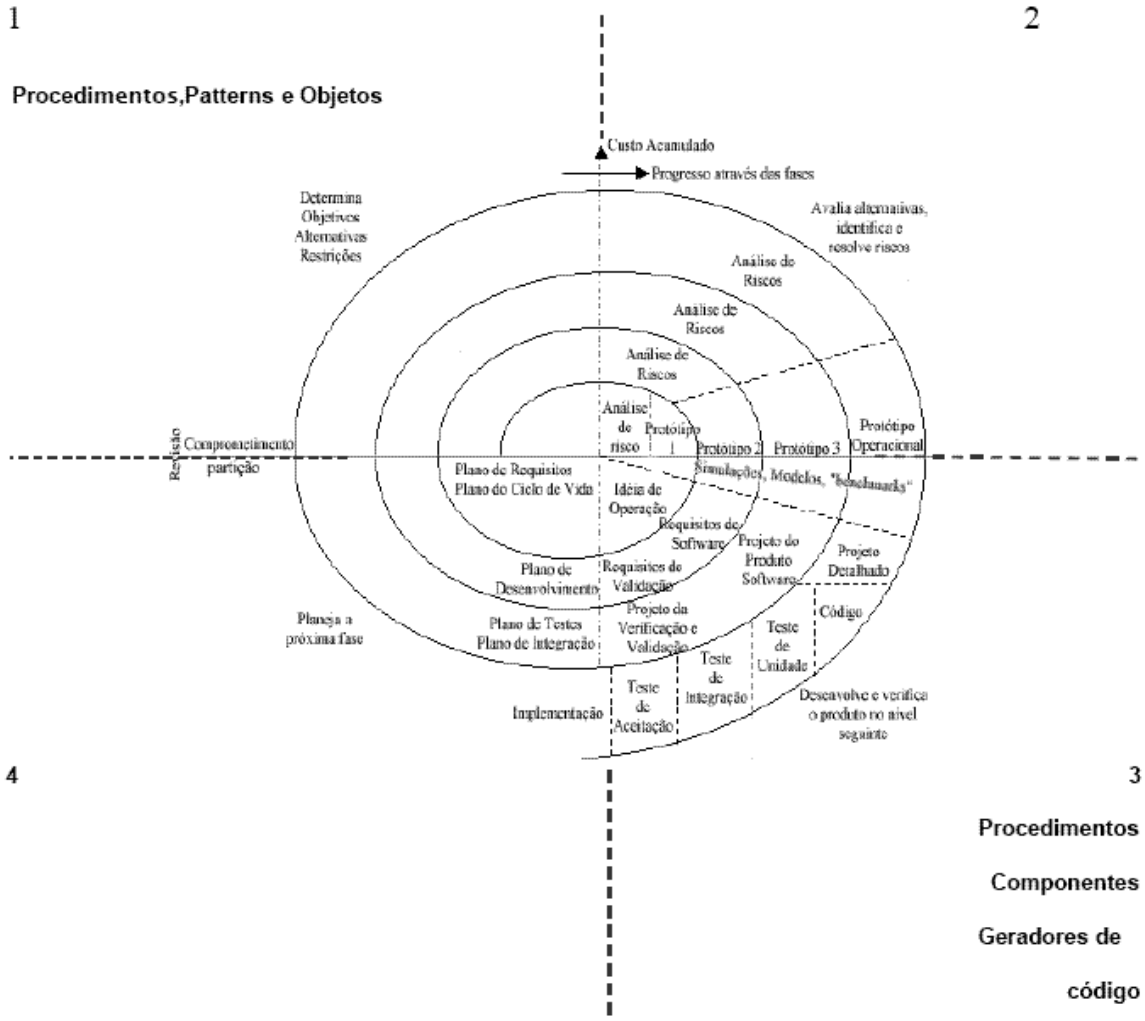

Figura 3 – Modelo espiral

- Quarto quadrante

Por ultimo o quarto quadrante compreende as etapas anteriores e o planejamento da próxima fase. Neste planejamento, dependendo dos resultados obtidos nos estágios anteriores como decisões, análise de riscos e verificação, pode-se optar por seguir o desenvolvimento num modelo cascata. Por exemplo, se já no primeiro quadrante, os requisitos forem completamente especificados e validados pode-se optar por seguir o modelo cascata, caso contrário, pode-se optar pela construção de novos protótipos, incrementando-o, avaliando novos riscos e re-planejando o processo.

Este capítulo abordou a importância do reuso de software, além de apresentar os vários tipos de abordagens de reuso existentes, e apresentou como o reuso pode ser integrado no processo de desenvolvimento de um produto de software. Apesar de apresentados dois modelos de desenvolvimento, não existe um modelo para o qual possamos dizer ser o modelo ideal, porém pelo estudo do tema o reuso de software pode ser associado como

uma maneira complementar ao desenvolvimento de software independente do modelo, provendo qualidade e agilidade.

O próximo capítulo trata de mostrar uma técnica de reuso, que são a utilização de frameworks e padrões para construção de arquiteturas de software. O capítulo sobre *frameworks* tratará então de mostrar em maiores detalhes um pouco de teoria sobre frameworks, além de mostrar uma parte prática também sobre alguns frameworks como o *Java Server Faces*, *Hibernate* e *Spring*.

Capítulo 3

Frameworks: Reuso e Aplicações

3.1 Introdução

Um componente representa reuso de código, um texto representa reuso de design ou reuso de projeto, comparando-se essas duas abordagens de reuso, o reuso de projeto é mais importante que reuso de código, uma vez que pode ser aplicada em um número maior de contextos e é mais comum.

A idéia de reuso tem muitas motivações, as principais são diminuir custos, tempo e esforço de programação, no processo de desenvolvimento de um software. Neste contexto os *frameworks* surgem como uma técnica de reuso intermediária, ela é parte reuso de código e parte reuso de projeto, ou seja, com os *frameworks*, reutilizam-se não somente linhas de código, mas também a arquitetura do domínio de aplicação ao qual ele atende.

Frameworks são utilizados em diversas áreas como em interfaces gráficas com usuário (GUI), editores de texto, sistemas operacionais, sistemas distribuídos, software para aviões, entre outros.

Este capítulo visa, por meio de uma pesquisa bibliográfica, apresentar reuso no contexto de frameworks, conceituando o que são *frameworks,* apresentando suas características gerais, formas de classificação, benefícios e desvantagens envolvendo o uso deles e apresenta também uma comparação com outras formas de reuso. Depois de apresentada um pouco de teoria, este capítulo finaliza mostrando uma parte mais prática, através da apresentação de três exemplos de *frameworks* orientado a objetos *open-source*, que são o *framework* MVC Java Server Faces, o *framework* de persistência *Hibernate* e o framework de inversão de controle Spring, descrevendo os principais conceitos computacionais envolvendo eles, os problemas aos quais pretendem resolver e um pouco de sua funcionalidade através de exemplos.

3.2 O que é um *Framework*?

Um *Framework* orientado a objetos é um conjunto de classes e interfaces que incorpora um projeto abstrato. Ele provê uma infra-estrutura genérica para construção de aplicações dentro de uma família de problemas semelhantes, de forma que esta infra-estrutura genérica deve ser adaptada para a geração de uma aplicação específica. O conjunto de classes que forma o *framework* deve ser flexível e extensível para permitir a construção

de várias aplicações com pouco esforço, especificando apenas as particularidades de cada aplicação.

Por exemplo, um framework para se construir diversos editores de texto poderia ser abstraído. Primeiramente toma-se o que eles têm em comum e produz-se um sistema abstrato. O framework é um conjunto de classes, de uma linguagem específica, que implementa este sistema abstrato de edição de texto. Se admitirmos que todas as classes deste sistema são abstratas, não se pode então instanciar objetos desta classe abstrata, logo este conjunto de classes abstratas, o framework “editor de texto”, não é um programa. Entretanto pode ser transformado em um criando-se subclasses e classes auxiliares. Como o framework é uma abstração de um editor de textos genérico, ele é flexível a mudanças, e, portanto vários editores de texto podem ser derivados dele apenas criando-se classes, subclasses e compondo objetos de classes do framework.

Como se pode ver no exemplo, um framework visa gerar diferentes aplicações dentro de um domínio, e ele contém, portanto, uma descrição dos conceitos deste domínio. As classes abstratas do framework do exemplo servem para representar as partes de um framework que variam, que servem para ser adaptados às necessidades da aplicação, ele poderia ser, por exemplo, um método de uma classe abstrata que foi deixado incompleto para que sua definição seja acabada na geração de uma nova aplicação. Tais partes são chamadas de pontos de especialização ou “*Hot-Spots*”.

Uma característica fundamental dos *frameworks* é quanto ao seu grau de reutilização. Um desenvolvedor que utiliza linguagem C, por exemplo, usa biblioteca de funções para construir aplicações, logo isto poderia ser classificado como reuso de rotinas. Já um programador que usa a linguagem Smaltalk utiliza classes para compor suas aplicações. Reusar classes possui, portanto, um grau de reutilização maior que o reuso de rotinas, uma vez que com classes se reutiliza um conjunto de rotinas, bem como uma estrutura de dados(atributos). Uma característica comum ao reuso de classes e rotinas é que cabe ao desenvolvedor implementar como os elementos do programa se relacionam entre eles, e qual o fluxo de execução da aplicação. Outra característica comum é que a reutilização se dá por reuso de componentes (rotinas ou classes) isolados, cabendo ao desenvolvedor estabelecer relações entre eles no sistema.

Já o grau de reuso promovido pelo uso de *frameworks* é superior ao grau de reuso promovido pelo reuso de rotinas e classes, por reusar um conjunto de classes interligadas ao invés de isoladas, além de que com o uso de *frameworks* os métodos especializados pelo usuário são chamados de dentro do próprio *framework* ao invés de ser chamado de dentro do código da aplicação do usuário. Esta característica presente nos *frameworks* é chamada de “inversão de controle”. As figuras 4, 5 e 6 abaixo representam essas diferenças, e os objetos em cinza representam componentes implementados pelo

desenvolvedor, enquanto que os objetos em branco representam o reuso de um componente.

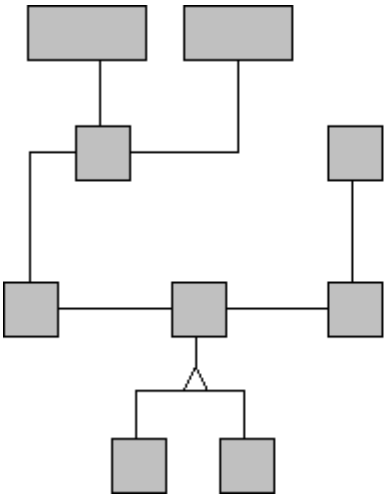

Figura 4 – Aplicação desenvolvida totalmente

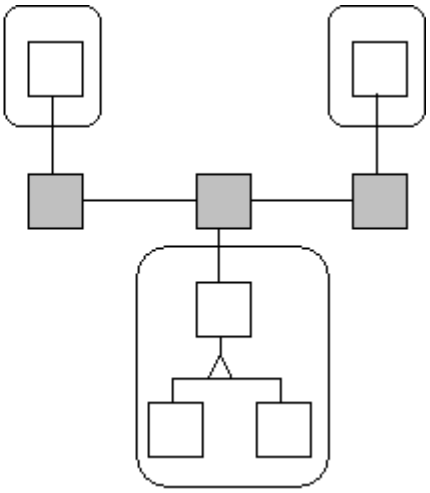

Figura 5 – Aplicação desenvolvida reutilizando classes e rotinas

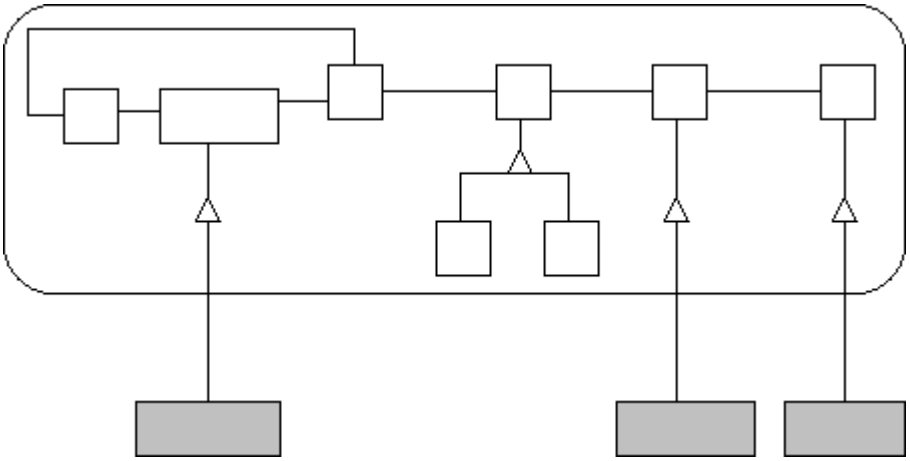

Figura 6 – Aplicação desenvolvida utilizando um *framework.*

Como se pode notar na figura 6, o framework impõe um modelo de colaboração, o resultado de análise e projeto, entre os objetos ao qual deve-se adaptar. Ele provê reuso de código, análise e design, sendo que o reuso de análise e projeto vem do fato de que, um framework representa um projeto de um sistema abstrato, previamente analisado e projetado para capturar todos as características comuns de um domínio de problema.

Um *framework* tem por objetivo ser estendível pela criação de subclasses e parametrizado pela aceitação de objetos de outras classes. Estas características estão presentes nos padrões de projeto "*Template method*" e "*Strategy*".

No padrão de projeto Template Method, um método é redefinido em uma subclasse e automaticamente modifica o comportamento de um outro método herdado e não redefinido. Por exemplo, um método de uma superclasse abstrata poderia ter como parte de sua implementação, uma chamada a um método abstrato a ser definido na subclasse e desse modo esse método definido na superclasse se comportaria de maneira diferente de acordo com a implementação dada a este método abstrato, por cada uma de suas subclasses. Esta técnica é utilizada para produzir sistemas em frameworks denominados caixa-branca (*White-Box*), onde produzimos subclasses redefinindo alguns métodos e com isso criamos um programa concreto que pode ser executado, ou seja, em frameworks caixa-branca o reuso é provido por herança.

O padrão Strategy permite configurar um objeto utilizando-se de um outro objeto que encapsula um algoritmo. Um objeto editor de textos ,por exemplo, pode delegar a tarefa de busca de uma palavra em um texto para outro objeto. O ponto importante é que eu posso passar, em tempo de execução ou compilação, um outro objeto que implementa um algoritmo de busca melhor sem precisar alterar o código do editor de textos, ou seja, eu construo um programa a partir da composição de objetos. Esta característica de composição de objetos é utilizada para criação de frameworks denominados caixa-preta. O instanciador define classes, cria objetos a partir delas e os passa ao framework parametriando-o. Este tipo de framework é mais fácil de ser utilizado do que os de caixa branca, pois não há herança envolvida, somente composição de objetos. O problema da herança é que ela introduz dependências entre métodos herdados que diminui a legibilidade do código.

3.3 Pontos fracos dos Frameworks

Além dos benefícios como reuso, inversão de controle, capacidade de extensão e modularidade, vantajosos como técnica de construção de software, ele possui também algumas desvantagens:

- **Custo de aprendizagem alto** : A aprendizagem de um framework leva um tempo considerável.
- **Não é fácil desenvolver um *framework***: A construção de um *framework* não é simples, e deve ser planejada para que o objetivo de reuso seja alcançado.
- **Difícil depuração**: A remoção de erros em um framework pode ser complicada, pois o uso de classes genéricas na composição de um framework não só auxilia a abstrair os detalhes da aplicação como dificultam o processo de depuração de código, pois não podem ser depurados separadamente da parte especifica da aplicação.

- **Documentação:** Não chega a ser um grave problema desde que o *framework* possua uma boa documentação. Uma documentação deficiente representa um verdadeiro desafio na utilização do framework.

A figura abaixo ilustra um resumo das vantagens e desvantagens. Ainda que se tenha algumas desvantagens, as vantagens são ainda maiores o que justiça o seu uso como técnica de construção de software.

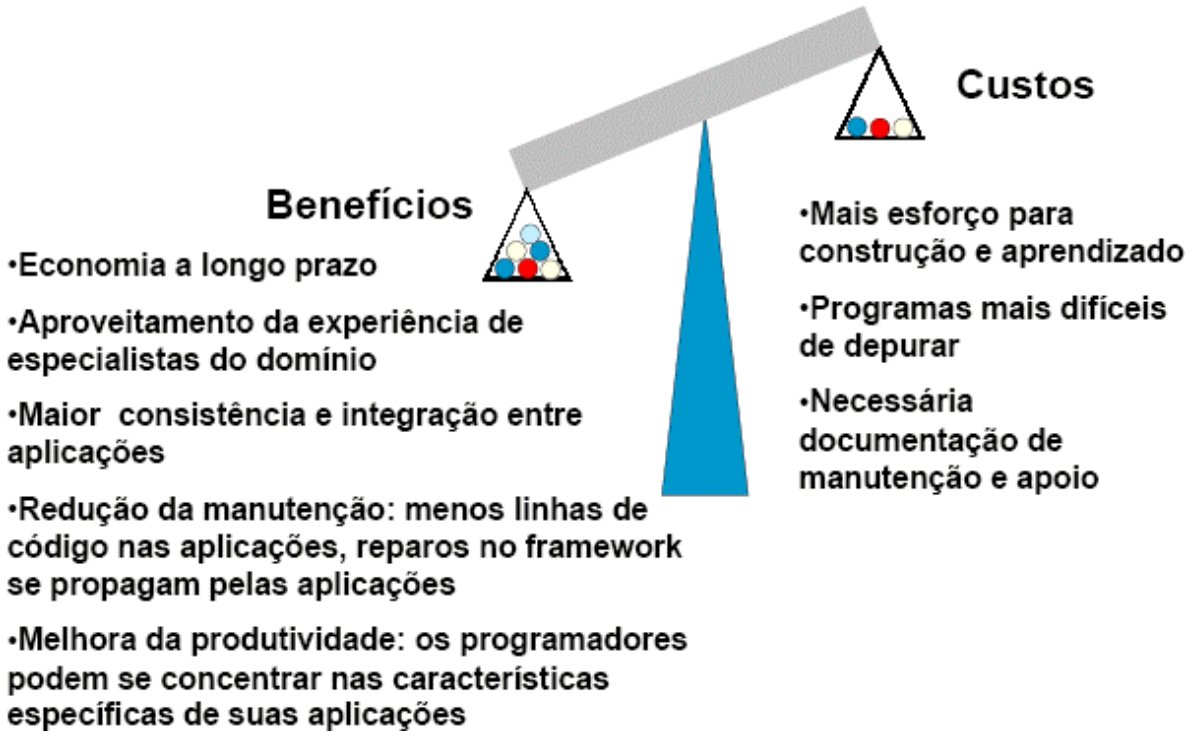

Figura 7 – Custo x Benefícios do uso de frameworks. (Fonte: Sauvê, [15])

3.4 Comparação com outras formas de reuso

Comparando-se frameworks com bibliotecas de rotinas, nas bibliotecas os componentes se apresentam isolados, eles são independentes uns dos outros, enquanto que num framework o conjunto de classes que forma o framework se coopera entre si. Outra diferença é que em uma biblioteca, cabe ao desenvolvedor a chamada de rotinas e determinar com será o fluxo de execução do programa, enquanto que num framework há inversão de papéis, é ele quem determina a estrutura geral do programa e qual o fluxo de execução dele, cabe ao desenvolvedor especializar somente os componentes que contém regras de negócio especificas ao domínio de aplicação do problema.

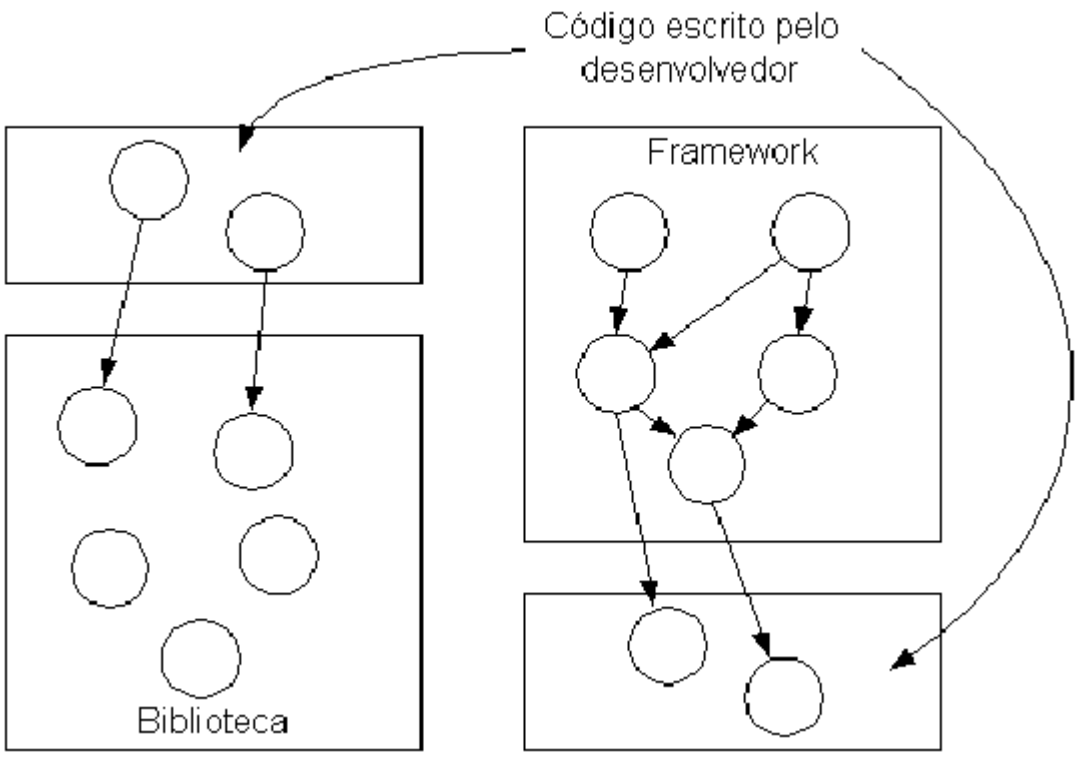

Figura 8 – Comparação entre um framework e biblioteca de rotinas. (Fonte: Sauvê, [15])

Comparando-se padrões de projeto com frameworks, os padrões se caracterizam por serem mais abstratos do que os frameworks, pois um framework é um software, possui uma implementação, enquanto que um padrão representa conhecimento e experiência sobre um projeto (Um framework possui uma forma “física”, já um padrão representa uma forma “abstrata”, uma idéia, uma solução). Outra diferença é que um padrão é “menor” do que um framework, pois geralmente a implementação de um padrão é composta por duas ou três classes, já um framework engloba um conjunto bem maior de classes e é formado por um conjunto de padrões.

Padrões de projeto são menos especializados que frameworks, pois os frameworks são desenvolvidos para um domínio de aplicação específico, já os padrões podem ser usados em diversos domínios.

3.5 Tipos de Frameworks

Frameworks podem ser classificados em dois grupos: *framework* de aplicação orientado a objetos e *framework* de componentes.

Frameworks de aplicação orientado a objetos geram aplicações orientadas a objetos. Seus pontos de extensão são definidos como classes abstratas ou interfaces, que podem ser estendidas ou implementadas para compor uma instancia da aplicação. Além disso, *frameworks* de aplicação orientada a objetos são classificados quanto ao seu escopo em

frameworks de infra-estrutura de sistemas, *frameworks* de integração de *middleware* e *frameworks* de aplicação corporativa.

- **Frameworks de infra-estrutura de sistemas** visam simplificar o processo de desenvolvimento de sistemas dando uma infra-estrutura básica e modular para compor aplicações como sistemas operacionais e ferramentas de processamentos de linguagens, são exemplos de frameworks de infra-estrutura o *Java Server Faces, Spring* e o *Hibernate*. Também são conhecidos como *frameworks* horizontais, a figura 9 ilustra esta categoria onde o enfoque maior ocorre na parte da generalização, isto significa que este tipo de framework visa atender a um número de domínios de aplicações maior, resolvendo apenas uma parte do problema da aplicação.

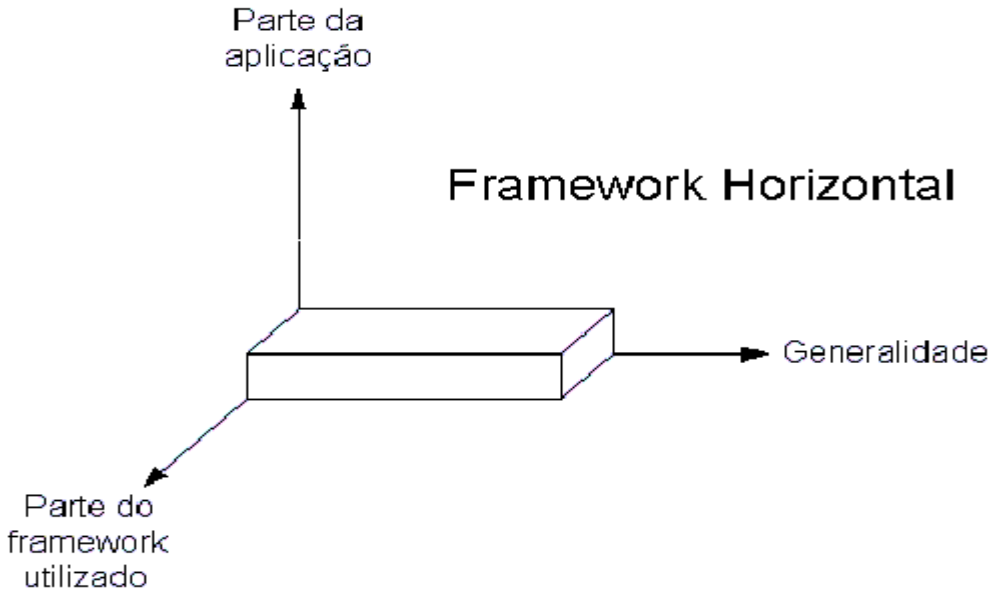

Figura 9 – Framework de infra-estrutura ou Horizontal (Fonte: Sauvê, [15]).

- **Frameworks de integração de *middleware*** são usados para integrar aplicações e componentes distribuídos. Estes *frameworks* escondem o baixo nível da comunicação entre componentes distribuídos, possibilitando que os desenvolvedores trabalhem em um ambiente distribuído de forma semelhante a que trabalham em um ambiente não distribuído. São exemplos de *frameworks* de integração de *middleware* Java RMI e *frameworks* ORB (*Object Request Broker*).

- **Frameworks de aplicações corporativas**, ao contrário dos frameworks de infra-estrutura e de integração de *middleware*, são voltados para um domínio de aplicação mais específico, como por exemplo, os domínios da aviação,

telecomunicações e financeiro. Também são conhecidos como *frameworks* verticais, a figura 10 ilustra esta categoria. Enquanto que nos *frameworks* horizontais se busca a generalidade do domínio, os verticais buscam o detalhe ao nível das aplicações, correspondendo uma parte maior do *framework* utilizado, como pode se analisar na figura 10 abaixo.

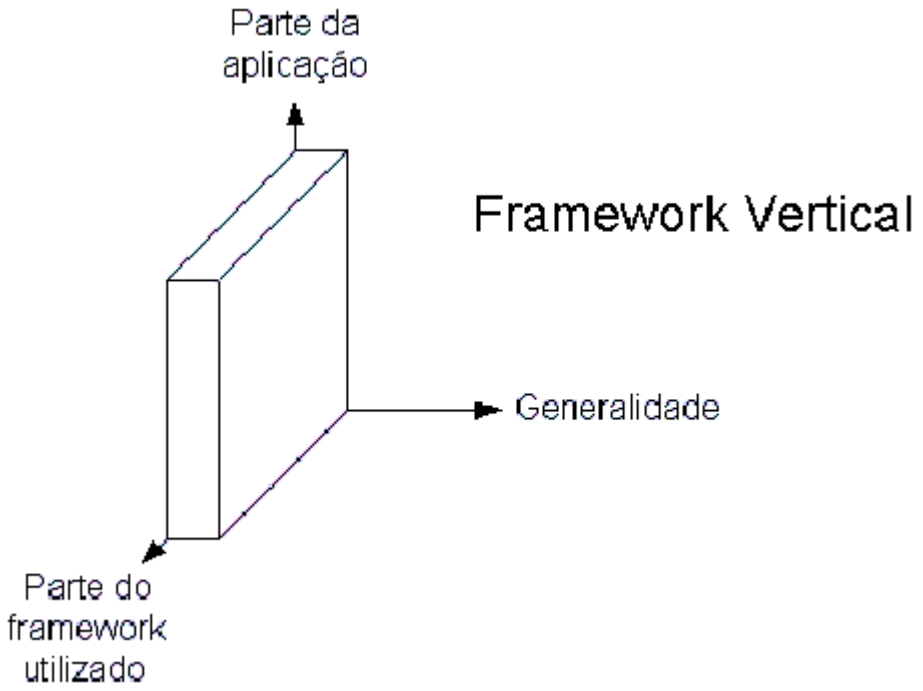

Figura 10 – Framework de aplicação corporativa ou Vertical (Fonte: Sauvê, [15]).

Finalmente, um framework de componentes é uma entidade de software que provê suporte a componentes que seguem um determinando modelo e possibilita que instâncias destes componentes sejam plugadas no framework de componentes. Ele estabelece as condições necessárias para um componente ser executado e regula a interação entre as instâncias destes componentes. São exemplos de *frameworks* de componentes temos o *Tapestry* e o *Java Server Faces*.

A principal diferença entre *frameworks* de aplicação orientado a objetos e *framework* de componentes é que, enquanto *frameworks* de aplicações definem uma solução inacabada que gera uma família de aplicações, um *framework* de componentes estabelece um contrato para plugar componentes.

A figura 11 mostra como pode ser feita a combinação de frameworks para criar uma aplicação, a idéia é que frameworks do tipo infra-estrutura, middleware e componentes sirvam de base sobre a qual se apóiam os verticais, uma vez que os primeiros têm uma preocupação maior na generalidade de domínio visando atender a um número maior de

aplicações, enquanto que os verticais são especialistas em um domínio e usam da infra-estrutura provida pela base.

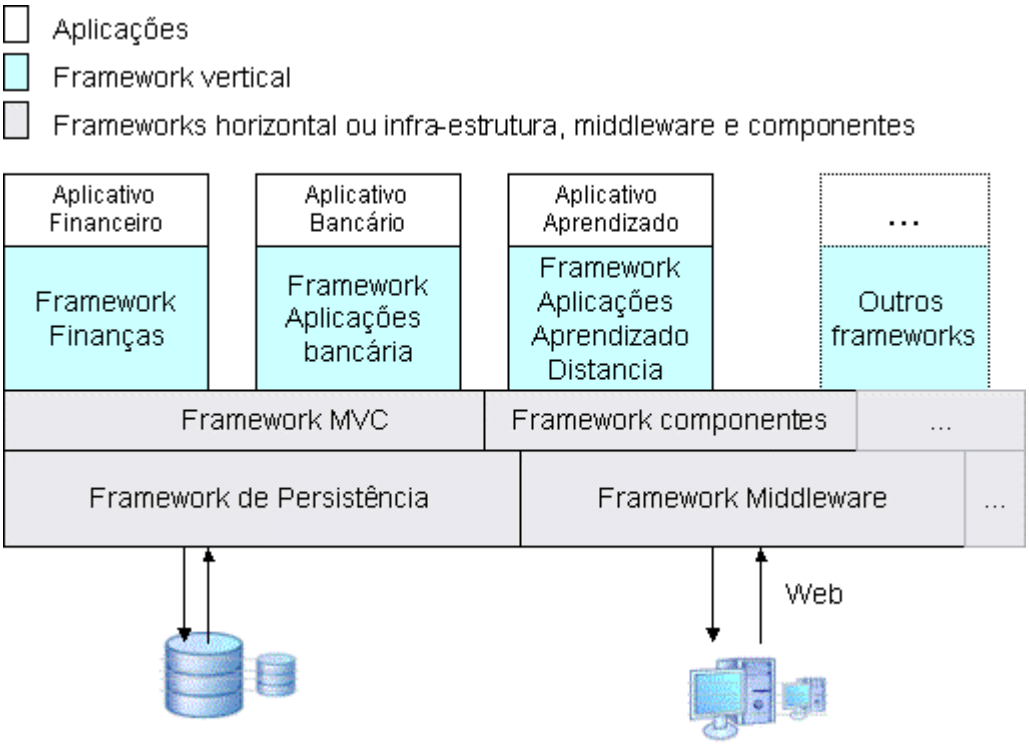

Figura 11 – Combinação de frameworks

3.6 Análise do Framework Horizontal MVC: Java Server Faces

A camada de apresentação em uma aplicação multicamadas tem como objetivo expor a lógica de negócios ao usuário, além de fornecer uma interface para interação do usuário com a aplicação. Em aplicações que utilizam o navegador como interface de interação com o usuário esta camada costuma ser chamada de camada *web*.

Atualmente muitos *frameworks* existentes para a camada web implementam o padrão de arquitetura *Model View Controller* (MVC), um exemplo é o framework Java Server Faces, que é também um framework de componentes e de infra-estrutura. *Framework* web é um bom estudo de caso, pois é um excelente exemplo da aplicação de vários *Design Patterns (como* Command, Template Method, Factory Method, Adapter, Composite), desse modo seu estudo se mostra interessante do ponto de vista computacional.

Para se entender o motivo da adoção do padrão MVC em *frameworks web*, deve-se entender primeiramente as vantagens e desafios decorrentes do uso de interfaces *HTML*.

3.6.1 Interfaces *WEB*

As interfaces *web* utilizam a linguagem HTML no desenvolvimento de interface com o usuário para aplicações *web*. Algumas das principais vantagens do uso de interfaces web são as seguintes:

- As aplicações *web* são instaladas no servidor e o usuário usa o navegador instalado em seu computador e, portanto, não necessita que novas aplicações sejam instaladas na máquina do cliente.
- Na maioria das empresas *firewalls* são configurados para liberar o trafego de rede na porta utilizada pelo servidor de paginas *HTML,* diminuindo o esforço para configuração.
- Impõe restrições de configuração de hardware mais modestas nas maquinas dos clientes já que a maior parte do processamento ocorre no lado servidor.

Contudo também são impostos uma serie de desafios, dentre eles:

- Interfaces com o usuário costumam mudar freqüentemente sem que a lógica de negócios mude necessariamente.
- O modelo de requisição e resposta impede que a camada de apresentação seja notificada de mudanças no modelo, diferentemente do que ocorre em interfaces para aplicações *desktop* onde um componente de interface pode ser imediatamente atualizado assim que ocorra uma mudança no modelo.
- É necessário separar o código de layout estático do código gerado dinamicamente.
- Requisições http carregam parâmetros do tipo String que precisam ser convertidos em tipos mais específicos da aplicação, além de necessitarem de validação.
- HTML oferece um conjunto limitado e não expansível de componentes de interface.
- Questões de desempenho e concorrência devem ser consideradas, pois é impossível prever o número de usuários acessando uma aplicação *web* simultaneamente.

O desenvolvimento de aplicações em camadas contribui para a solução de alguns destes desafios, pois como uma camada só depende de outra imediatamente inferior, com isso a camada de apresentação pode ser alterada sem que a de negócios sofra modificações, alem do que cada camada pode ser testada, depurada e otimizada isoladamente.

Contudo apenas a divisão em camadas não basta para solucionar estes problemas. Para que isto ocorra é necessário que a camada de apresentação seja limpa, isto é, o fluxo de controle e a chamada a métodos de negócios são separados da visão, pois se o código

para a interface gráfica é muito acoplado ao código de negócios da aplicação, a construção de uma nova interface de apresentação implicaria na replicação de código das regras de negócios, uma vez que tais regras estavam acopladas a interface de apresentação anterior.

Para que a camada de apresentação seja separada da camada de negócios, ou seja, para que o fluxo de controle da aplicação e a chamada a métodos de negocio seja separada da visão usa-se o padrão de arquitetura *Model View Controller (MVC)* apresentado na próxima seção.

3.6.2 O padrão *Model View Controller (MVC)*

O padrão de arquitetura MVC divide os elementos da camada de apresentação em visão (*View*), que recebe a entrada do usuário e exibe o resultado da operação, Controlador (*Controller*), que acessa a camada de negócios manipulando o modelo e selecionando a visão apropriada, e o Modelo (*Model*) objeto que contém a lógica de negócios do domínio da aplicação e que provê dados para a visão. A figura 13 abaixo ilustra como esta caracterizada o modelo MVC no *framework Java Server Faces*.

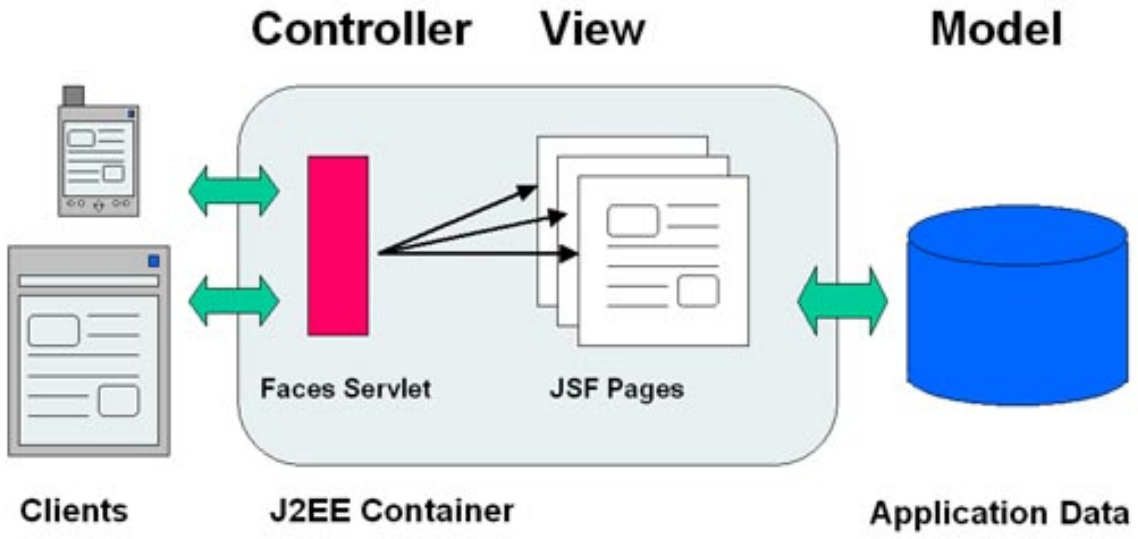

Figura 13 – MVC no Java Server Faces

Embora seja possível implementar uma solução que implante o *MVC*, há vários *frameworks* que trazem esta estrutura além de tratarem de alguns dos desafios da seção 3.6.1, dentre eles o *Java Server Faces* (JSF) utilizado no sistema *Opusibank*, estes *frameworks* que trazem esta estrutura de MVC são comumente chamados de *frameworks MVC* ou *frameworks web*.

3.6.3 *Frameworks Web*

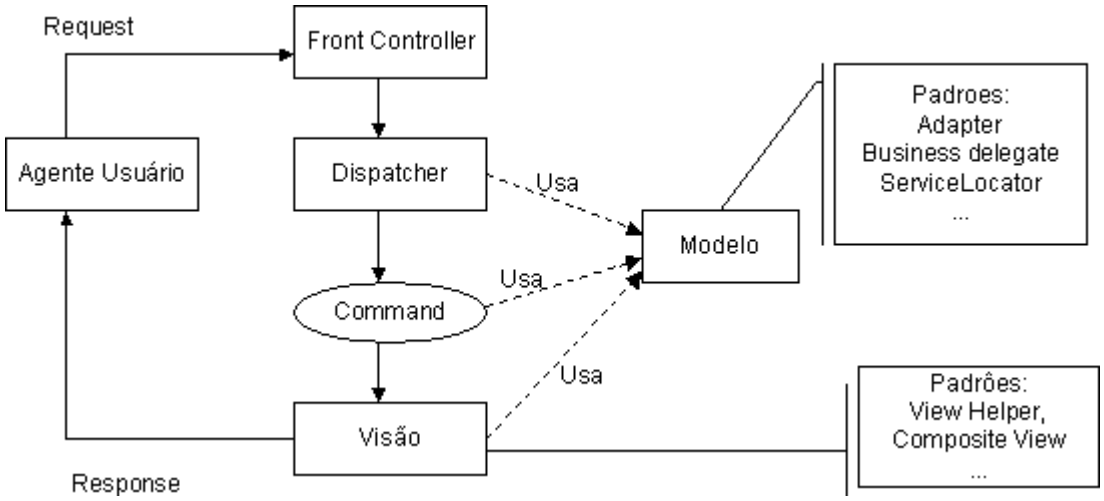

Figura 14 – Característica Geral de Execução dos Frameworks web

Consultando a *web* pode se ver que existem aproximadamente cerca de 54 *frameworks open-source* existentes para o desenvolvimento *web* em plataforma Java atualmente (http://www.manageability.org/blog/stuff/how-many-java-web-frameworks/view), isso sem levar em conta os frameworks comerciais que existem também. As maiorias destes frameworks implementam o modelo MVC de forma muito similar e têm características comuns que visam solucionar alguns dos problemas listados na seção 3.6.1. Este comportamento comum é descrito a seguir.

Estes *frameworks* usualmente trazem o controlador do MVC pronto, implementado através de um *servlet* (*Servlets* são classes/objetos que funcionam como programas CGI) que deve ser configurado ao instanciar o *framework*. O controlador é implementado geralmente através do padrão de projeto *Front Controller(Fowler, 2002)* e sua função é mapear requisições para classes de instancia que implementam o padrão de projeto *Command* (gamma et al., 1994).

Esta instancia do *Command* é responsável pela construção da visão com a ajuda de componentes fornecidos pelo *framework*. Já o modelo é utilizado pela instância e não deve possuir nenhum tipo de dependência com o framework escolhido, pois ele é utilizado em outras camadas, dessa forma aumentamos a modularidade da aplicação e facilitamos a manutenção uma vez que alterações de código estão localizados em um único ponto de extensão.

Cabe ao Controlador extrair os parâmetros da requisição, que são do tipo String e converte-los para tipos mais específicos da aplicação que podem ser desde tipos primitivos como inteiros e boleanos a objetos do modelo, e validá-los. Em caso de erro de validação, o controlador exibe novamente a visão que causou o erro. Quando não ocorre erro de validação, o controlador chama o comando que aciona a camada de negócios

para modificar o modelo. Terminada a operação, o comando sinaliza para o controlador qual a visão deve ser exibida e este finaliza a requisição montando a visão.
A figura 14 ilustra o funcionamento descrito acima, na figura estão presentes muito dos padrões de projeto aplicados na camada de apresentação e na camada de negócios.

3.6.4 O *Framework Java Server Faces (JSF)*

Java Server Faces é um framework MVC que difere dos outros existentes por ser também um framework de components, pois define um modelo de componentes para serem usados na camada de apresentação. Existem três tipos principais de componentes no JSF:

- Componentes de Interface, usados para compor a interface com o usuário, como caixas de texto e campos de seleção de valor.

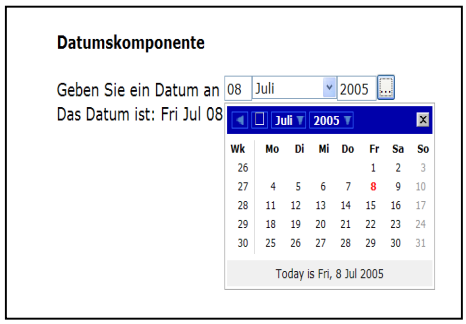

Figura 15 –Componente calendário

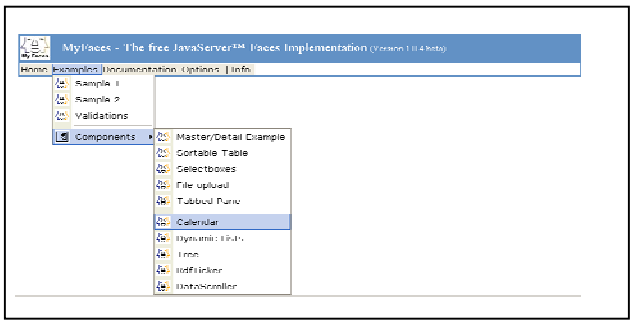

Figura 16 – Componente Menu

- Componentes de conversão de dados, chamados de ***Converter.***
- Componentes para validação de dados, usados para validar a entrada do usuário, chamados de ***Validator***.

Os comandos (padrão Command conforme descrito na figura acima) recebem o nome de backing beans ou managed beans em jsf. Estes não precisam implementar interfaces especificas ou estender classes, bastando que o método do comando seja publico e não receba parâmetros. A cada componente de interface de usuário é associado um backing bean que manipula o estado deste componente e acessa a camada de negócios. Em jsf o controlador do mvc já vem implementado como um servlet conforme descrito na listagem 1 abaixo, o servlet controlador está declarado entre as linhas 2-5:

```
01: <web-app>
02: <servlet>
03: <servlet-name>Faces Servlet</servlet-name>
04: <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
05: </servlet>
06: <servlet-mapping>
07: <servlet-name>Faces Servlet</servlet-name>
08: <url-pattern>*.jsf</url-pattern>
09: </servlet-mapping>
10: </web-app>
```

Listagem 1 – Arquivo web.xml contendo definições do controlador do jsf

A listagem 2 mostra um exemplo de uma pagina jsf.

```
11: <%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
12: <%@taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
13: <f:view>
14: <html>
15: <head>
16: <title>Pagina Envio de Formulário</title>
17: </head>
18: <body>
19: <h:form>
20: <h:messages/><br>
21: Curso:
22: <h:selectOneListbox value="#{CursoBean.idCategoria}"
23:                                    required="true">
24: <f:selectItem itemLabel="BCC" itemValue="1"/>
25: <f:selectItem itemLabel="MAC" itemValue="2"/>
26: <f:selectItem itemLabel="MAT" itemValue="3"/>
27: </h:selectOneListbox><br>
28: Assunto:
29: <h:inputText value="#{CursoBean.message.subject}"
30:                        size="83" maxlength="100" required="true">
31: <f:validateLength minimum="10"/>
32: </h:inputText><br>
33: Mensagem: <br>
34: <h:inputTextarea value="#{CursoBean.message.text}"
                                  rows="15" cols="60" cols="65" rows="16"
35:                             required="true"/>
36: </h:form>
37: </body>
38: </html>
39: </f:view>
```

Listagem 2 – Exemplo Página de Envio De Formulário (JSF)

Na listagem 2 acima expressões delimitadas por “#{“ e “}” demarcam o uso da linguagem de expressões do JSF e servem para criar vínculos de um componente com uma classe Java que trata de manipular o estado deste componente. O item entre as linhas 22 e 27 delimita um componente do tipo lista onde apenas um item pode ser selecionado. O

atributo *value* na linha 22 vincula o valor do item selecionado a propriedade *idCategoria* do *backing bean* CursoBean. Ainda na linha 22, o atributo *required="true"* demarca que o atributo é requerido, gerando regras de validação automaticamente. A linha 31 indica uma validação de tamanho mínimo, indicando que o campo associado a esta só será válido se tiver um tamanho mínimo de 10 caracteres.

É através do arquivo XML de configuração do JSF que são configurados casos de navegação, backing *beans*, componentes de validação e de conversão de tipos. A Listagem 3 exibe um exemplo deste arquivo, que tem como nome padrão faces-config.xml.

```
40: <faces-config>
41: <navigation-rule>
42: <from-view-id>/formulario.jsp</from-view-id>
43: <navigation-case>
44: <from-outcome>success</from-outcome>
45: <to-view-id>/formularioSuccess.jsp</to-view-id>
46: </navigation-case>
47: <navigation-case>
48: <from-outcome>failure</from-outcome>
49: <to-view-id>/formularioFailure.jsp</to-view-id>
50: </navigation-case>
51: </navigation-rule>
52: <managed-bean>
53: <managed-bean-name>CursoBean</managed-bean-name>
54: <managed-bean-class>CursoBean</managed-bean-class>
55: <managed-bean-scope>request</managed-bean-scope>
56: </managed-bean>
57: </faces-config>
```

Listagem 3 – arquivo de configuração faces-config.xml

Entre as linhas 41 e 51 são declarados dois casos de navegação a partir da página /formulario.jsp. O caso entre as linhas 43 e 46 é executado quando a operação é executada com sucesso enquanto que o caso entre as linhas 47 e 51 é executado quando ocorre algum erro durante a execução da operação.

O *backing bean* é declarado entre linhas 52 e 57. Na linha 53 o nome do *backing bean* é definido, enquanto que na linha 54 é configurada a classe. A Listagem 4 exibe o código fonte deste *backing bean*.

```
58: public class CursoBean {
59: private String idCategoria;
60: private Message message;
61:
62: public void setIdCategoria (String newValue) {this.idCategoria =
newValue; }
63: public String getIdCategoria () { return idCategoria; }
64: public void setMessage(Message message) { message = newValue; }
65: public Message getMessage() { return message; }
66:
67: public String postMessage() {
68: try {
69: CursoFacade facade = new CursoFacadeImpl();
70: FacesContext ctx = FacesContext.getCurrentInstance();
71: String curso = ctx.getExternalContext().getRemoteUser();
72: facade.postMessage(curso, idCategoria, message);
73: } catch (Exception e) {
74: return "failure";
75: }
76: return "success";
77: }
78: }
```

Listagem 4 – Backing Bean

3.7 Análise do Framework Horizontal de Persistência: Hibernate

A maior parte dos sistemas existentes hoje requer em alguma fase de implementação do projeto que seus dados sejam persistidos de alguma forma. A forma mais comum de realizar isto é persistir dados através de um sistema de gerenciamento de banco de dados (SGBD) relacional.

Em Java o modo mais baixo nível de se enviar comandos ao SGBD é através da API *Java Database Connectivity* (JDBC). Com o JDBC uma aplicação Java consegue enviar comandos SQL para o SGBD e assim efetuar operações de consulta, atualização, remoção e inserção de dados.

A fim de separar os comandos SQL do código que realiza alguma lógica de negócio com os dados provenientes de uma consulta, o código JDBC (o que realiza operações de

consulta ao SGBD) fica responsável por realizar a conversão de dados provenientes do SGBD em classes orientadas a objeto escritas em Java, fornecendo um modo OO de ver entidades do banco de dados relacional.

Entretanto escrever código de acesso ao banco de dados, realizando as conversões necessárias do paradigma relacional para o paradigma OO é uma tarefa desgastante, repetitiva e algumas vezes propensa a erros devido a diferença entre estes dois paradigmas. É nesse contexto que entra o *framework* de mapeamento objeto/relacional *Hibernate*. O termo mapeamento objeto relacional é um nome dado para *frameworks* capazes de realizar a persistência automática e transparente de classes de uma aplicação em tabelas de banco de dados relacionais, transformando a representação de dados de um paradigma para outro.

Para melhor entendermos o *Hibernate* é preciso entender primeiro o conflito entre os paradigmas OO e relacional, que ele se propõe a mediar.

3.7.1 O Paradigma OO x Relacional

Ao mudar de um paradigma para outro surgem problemas que devem ser resolvidos. Dentre eles, temos: a questão dos relacionamentos e a questão do grafo de navegações.

A questão dos relacionamentos surge a partir da diferença entre estas duas de se representar relacionamentos. Relacionamentos em orientação a objetos são expressos através de referências a objetos ou coleções de objetos. Já no paradigma relacional os relacionamentos são expressos através de chave estrangeira. Outra diferença é que em OO os relacionamentos são direcionais, ou seja, são definidos de uma classe para outra. Para criar um relacionamento bidirecional é preciso definir dois relacionamentos unidirecionais, um em cada uma das classes envolvidas. A noção de direção não existe no paradigma relacional. Além disto, classes em OO podem ser definidas tendo relacionamento entre objetos de 1 para 1, 1 para n e m para n, enquanto que no paradigma relacional relacionamento para n são feitos através de uma tabela extra, que não aparece no modelo OO.

E finalmente a questão do grafo de navegações. Na figura abaixo temos um diagrama UML representando uma associação de unidirecional de 1 para 1 entre a classe Endereço e a classe Usuário.

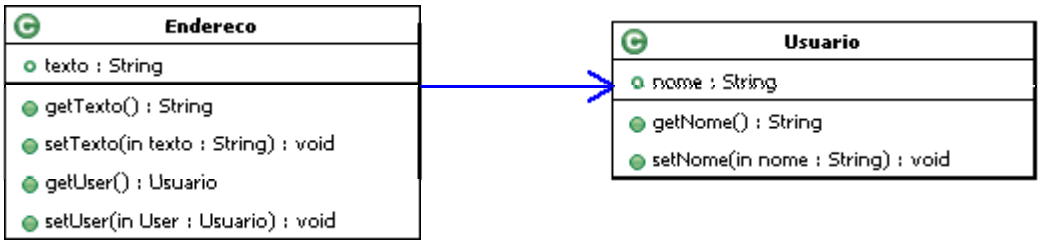

Figura 17 – Diagrama de classes para exemplificar o problema do grafo de navegações

A questão do grafo de navegação vem do fato de que no paradigma OO, a forma natural de acessar dados em objetos relacionados é através da técnica onde se navega de um objeto para outro através de métodos get, por exemplo, para se obter o endereço de um usuário sobre um objeto do tipo Endereço eu tenho que fazer endereço.getUser().getNome(). Apesar de simples esta não é a forma mais eficiente de obter dados em um SGBD relacional e pode levar ao problema do "n + 1 select problem" (King & Bauer, 2005) onde para cada navegação no grafo de objetos é realizada uma consulta na base de dados. À medida que os relacionamentos entre objetos se tornam mais complexos e a quantidade de dados aumenta, recuperar o grafo de objetos de uma vez só implicará em problemas de performance e escalabilidade. A questão do grafo de navegações envolve então determinar qual porção do grafo deve ser recuperada imediatamente e qual deve ser recuperada sob demanda.

Portanto como pode se ver são muitas as questões envolvidas quando se quer converter um objeto em tabelas, esta tarefa é repetitiva e independente do domínio de aplicação, logo favorável à aplicação de frameworks.

Nos itens seguintes então veremos como o *Hibernate* trata destas questões.

3.7.2 Um Exemplo com o *Hibernate*

Para melhor entender o Hibernate esta seção apresenta um exemplo que ilustra o funcionamento deste framework. O modelo de objetos a ser persistido escolhido é o apresentado na figura 18 abaixo.

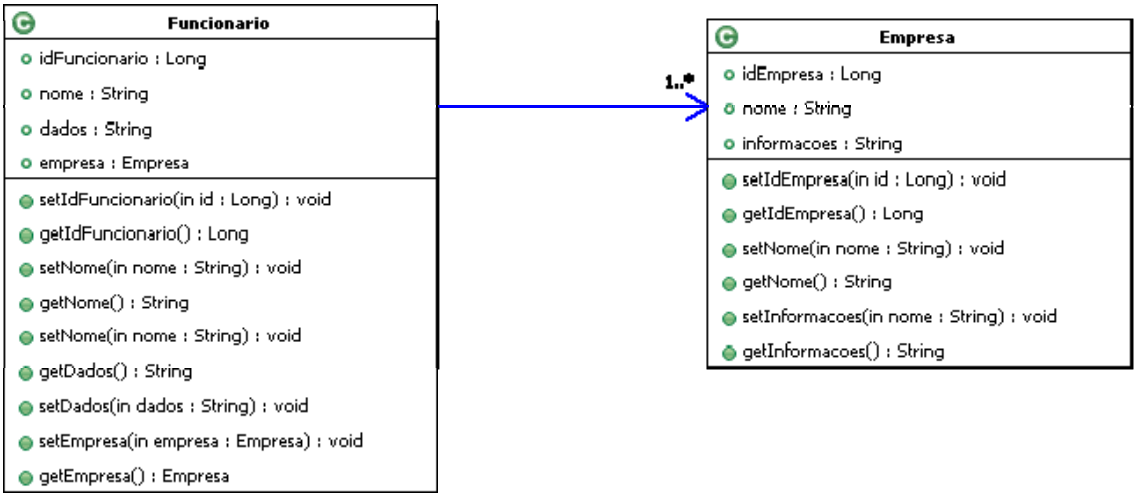

Figura 18 – Diagrama de classes do exemplo entre Funcionário e Empresa.

A implementação das classes Funcionário e Empresa são feitas como Java beans sem quaisquer dependências com classes ou API do Hibernate. As listagens abaixo exibem o código fonte das classes Funcionário e Empresa.

```
01: public class Funcionario {
02:     private Long idFuncionario;
03:     private String nome;
04:     private String dados;
05:     private Empresa empresa;
06:     /* Método get/set */
07:    ( ……. )
08: }
```

Listagem 5 – Classe Funcionário

```
09: public class Empresa {
10:       private Long idEmpresa;
11:       private String nome;
12:       private String informacoes;
13:     /* Métodos get/set */
14:    (…….)
15:     }
```

Listagem 6 – Classe Empresa

A configuração com do Hibernate deve ser feita em um arquivo chamado hibernate.cfg.xml e este arquivo é definido conforme a listagem abaixo.

```
16: (….. )
17: <hibernate-configuration>
18: <session-factory>
19: <property name="connection.datasource"> (….)  </property>
20: <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
21: <mapping resource="Funcionario.hbm.xml"/>
22: <mapping resource="Empresa.hbm.xml"/>
23: </session-factory>
24: </hibernate-configuration>
```

Listagem 7 – Arquivo hibernate.cfg.xml de Configuração do Hibernate

Neste arquivo a linha 18 inicia a seção que configura um *SessionFactory* responsável pela criação de sessão no *Hibernate*, que são objetos do tipo *Session*. A sessão de *Hibernate*, também chamada de gerenciador de persistência, pode ser vista como um gerenciador de objetos relacionados a uma mesma unidade de trabalho. Uma sessão é capaz de inserir, consultar, atualizar e remover objetos assim como detectar mudanças nos objetos pertencentes a unidade, atualizando seus estados persistentes quando a sessão é fechada. A sessão é um objeto leve, ou seja, não ocupa muito espaço em memória e não leva muito tempo para ser criado e destruído.Geralmente objetos sessão possuem um tempo de vida curto.

Na linha 19 é configurado um DataSource, que é um objeto que provê conexões com o banco de dados. A linha 20 define um dialeto, que define para qual banco de dados o *Hibernate* deve gerar comandos SQL, e é justamente esta uma das características principais de *Hibernate* o suporte a diversos tipos de banco de dados relacionais, ou seja, com o Hibernate eu obtenho um sistema independente de banco de dados através deste

framework, e conforme veremos adiante as consultar ao SGBD são feitas via uma sintaxe padrão do *Hibernate*, o *HQL (Hiberntae Query Language*).

Por último, nas linhas 21-22 é definido o nome dos arquivos de metadados de mapeamento para, respectivamente, as classes Funcionário e Empresa. Através destes arquivos o *Hibernate* realiza o mapeamento de objetos em tabelas. As listagens 8 e 9 seguintes mostram estes mapeamentos.

```
25: (……..)
26: <hibernate-mapping>
27: <class name="Empresa" table="EMPRESA">
28: <id name="idEmpresa" column="ID_EMPRESA"  type="long">
29:         <generator class="native"/>
30: </id>
31: <property name="nome"  type="string" column="NOME" length="50"
32:                 not-null="true"/>
33: <property name="informacoes" type="string" column="INFORMACOES"
34:                 length="100"/>
35: </class>
36: </hibernate-mapping>
```

Listagem 8 – Arquivo Empresa.hbm.xml de Mapeamento Objeto – Relacional

A linha 27 indica que este arquivo se refere ao mapeamento da classe Empresa na tabela EMPRESA. Entre as linhas 28-30 são definidas as características do identificador do objeto, de modo similar a propriedade idEmpresa da classe Empresa é associada à chave primaria ID_EMPRESA da tabela EMPRESA.

Entre as linhas 31-34 é continuado o mapeamento entre atributos de uma classe e campos de uma tabela, no caso dos atributos nome e informações da classe Empresa para os campos NOME e INFORMACOES da tabela EMPRESA.

```
37: <hibernate-mapping>
38: <class name=”Funcionario"   table="FUNCIONARIO">
39: <id name="idFuncionario" column="ID_FUNCIONARIO"  type="long">
40:       <generator class="native"/>
41: </id>
42: <property name="nome" type="string" column="NOME"/>
43: <property name="dados" type="string" column="DADOS"/>
44: <many-to-one name="empresa"  class="Empresa"  column="ID_EMPRESA"
45:                     not-null="true"  cascade="save-update"/>
46: </class>
47: </hibernate-mapping>
```

Listagem 9 - Arquivo Funcionario.hbm.xml de Mapeamento Objeto – Relacional

Neste arquivo a classe Funcionário é mapeada para a tabela FUNCIONARIO.
Cada uma das propriedades de Funcionário como idFuncionario, nome e dados são mapeados para, respectivamente, as colunas ID_FUNCIONARIO, NOME e DADOS.
O trecho entre as linhas 44-46 define o relacionamento entre as classes Funcionário e Empresa. O relacionamento entre as duas classes é definido como muitos para um no sentido de Funcionário para Empresa, sendo que o campo ID_EMPRESA definido na linha 44 indica que este é o nome da chave estrangeira armazenada na tabela FUNCIONARIO que referencia a tabela EMPRESA. O atributo *not-null* é configurado como *true* para indicar que o relacionamento é obrigatório e o atributo *cascade* é configurado como "*save-update*" na linha 45 para indicar que as operações de inserção e atualização de entidades devem ser realizadas também para as instancias relacionadas.
Feitas todas as configurações, o próximo passo é usar a API do *Hibernate* para realizar operações de inserção, consulta, atualização e remoção. Na listagem abaixo irei mostrar apenas a operação de inserção.

```
48: public class OperacaoInsercao {
49: public static void main(String[] args) {
50:     SessionFactory sfactory = null;
51:     Session sess = null;
52:     Transaction tx = null;
53:     try {
54:     sfactory = new Configuration().configure().buildSessionFactory();
55:      sess = sfactory.openSession();
56:      tx = sess.beginTransaction();
57:      Empresa empr = new Empresa();
58:      empr.setNome("Empresa X");
59:      Funcionario func = new Funcionário();
60:      func.setNome("Alexandre ");
61:      func.setEmpresa(empr);
62:      sess.save(funcionario);
63:      tx.commit();
64:     sess.close();
65:  } catch (Exception e) {
66:     try {
67:         tx.rollback();
68:        sess.close();
69:    } catch (HibernateException e1) { }
70:        e.printStackTrace();
71:    }
72:  }
73: }
```

Listagem 10 – Operação de Criação com o Hibernate

As linhas 52 e 56 definem e inicializam uma variável do tipo *Transaction*. A interface *Transaction* representa uma abstração de transações, através desta podem ser usadas transações JDBC ou JTA. As transações podem ser configuradas via o arquivo *hibernate.cfg.xml* mostrado anteriormente.

Entre as linhas 57 e 61 um objeto Empresa e um objeto Funcionario são criados e inicializados. Na linha 61, o relacionamento entre os dois objetos é configurado. Na linha 62 o objeto é salvo na sessão. Finalmente, a transação é confirmada na linha 63 e a sessão é fechada na linha 64. Como o atributo *cascade* do relacionamento é definido como *save-update* (linha 45 da listagem 9), o objeto Empresa associado à classe Funcionario também é salvo. O trecho de código entre as linhas 65 e 71 trata eventuais erros que podem ocorrer durante a operação.

Para realizar operações de consulta *Hibernate* possui uma linguagem chamada HQL (*Hibernate Query Language*). O HQL é de certa forma similar ao SQL, mas agrega conceitos de OO, possibilitando a seleção de objetos em vez de tabelas. Um trecho de código que exemplifica essa linguagem de consulta HQL é mostrado abaixo, onde todos os funcionários com o primeiro nome “Alexandre” são selecionados.

```
74: Session sess;
75: //  operacoes de inicializacoes como mostradas na listagem 2.3.6
76: String queryStr = "from Funcionario f where f.nome like ?";
77: Query qry = sess.createQuery(queryStr);
78: qry.setString(0, "Alexandre%");
79: Collection usrs = qry.list();
80: System.out.println(usrs.size() + " funcionarios selecionados!");
81: for (Iterator i = usrs.iterator(); i.hasNext(); {
82:       Funcionario func = (Funcionario) i.next();
83:       System.out.println (func.getName());
84: }
85: sess.close();
```

Listagem 11 – Operação de Consulta com o Hibernate Usando HQL

A string com o comando HQL executado é criado na linha 76. Um parâmetro é usado no lugar para a propriedade *nome*, de modo a tornar a consulta mais genérica. Na linha 77 um objeto *Query* é criado a partir da sessão. O parâmetro é configurado na linha 78 para que apenas os funcionários com nome começando com “Alexandre” sejam selecionados e a consulta é realizada na linha 79. O trecho de código entre as linhas 81 e 84 imprime o resultado da consulta.

3.7.3 *Hibernate* e o problema do Grafo de Navegação

O principal desafio dos mapeadores objeto-relacional é fornecer acesso eficiente a uma base de dados relacional através da representação na forma de um grafo de objetos. A questão é determinar qual porção do grafo deve ser recuperada imediatamente e qual deve ser recuperada sob demanda. Para lidar com esta questão, *Hibernate* define quatro

estratégias de busca que podem ser usadas em quaisquer relacionamentos: busca imediata (*immediate fetching*), busca tardia *(lazy fetching*), busca antecipada (*eager fetching*) e busca em lote (*batch fetching*).

A estratégia de busca imediata(*immediate fetching*) age logo que uma entidade é recuperada do banco de dados, a entidade do relacionamento é recuperada através de uma consulta à base de dados ou então ao cache de entidades. Esta estratégia não costuma ser eficiente a não ser que as entidades relacionadas estejam quase sempre no cache.

A estratégia de busca tardia (*lazy fetching*) permite que a entidade do relacionamento seja recuperada sob demanda, apenas quando for necessária consulta-la.

Com a estratégia de busca antecipada(*eager fetching*), as entidades relacionadas são recuperadas em uma mesma consulta através do uso do comando *SQL OUTER JOIN*. Otimizações em uma aplicação que usa *Hibernate* geralmente envolvem a configuração de relacionamentos, escolhendo a estratégia de busca antecipada para classes que quase sempre são usadas em conjunto.

A estratégia de busca em lote (*batch fetching*) é uma técnica que pode aumentar a performance de relacionamentos com a estratégia de busca tardia ou imediata. Usualmente, ao recuperar um objeto da base de dados, uma consulta SQL é realizada com uma cláusula WHERE especificando o identificador do objeto a ser recuperado. Se a estratégia da busca em lote for usada, sempre que uma consulta for realizada, o *Hibernate* procura por outros objetos na mesma unidade de trabalho da sessão usando a mesma consulta, mas com valores múltiplos para a cláusula WHERE. Quase sempre a estratégia de busca tardia é mais eficiente que esta estratégia, porém a busca em lote é mais adequada para usuários do *Hibernate* que não desejam ou não podem usar seu tempo para ajustar a aplicação com uma combinação de busca tardia e antecipada.

Infelizmente esta seção não mostra um exemplo contendo configurações de busca, porém o objetivo principal era apenas mostrar como o *Hibernate* oferece meios para resolução da questão do grafo de navegação, cabe ao desenvolvedor da aplicação configurar os atributos do relacionamento no arquivo de mapeamento de tabelas do *Hibernate* para obter uma performance satisfatória.

3.7.4 *Hibernate* e o problema dos relacionamentos

O *Hibernate* possibilita o uso de relacionamentos unidirecionais ou bidirecionais, com cardinalidade 1-1, 1-m e m-n. Há várias formas de se mapear relacionamentos com o *Hibernate* e descrever todas seria excessivamente complexo e fugiria ao escopo desta dissertação. Contudo, o exemplo da seção 3.7.2. exemplifica o uso de relacionamentos direcionais 1-1.

O *Hibernate* impõe uma limitação em classes com relacionamentos no lado m da relação: a variável do relacionamento deve ser determinada em função da interface da coleção em vez da classe concreta. Por exemplo, a interface java.util.List deve ser usada em vez da classe java.util.LinkedList. O Hibernate usa sua própria implementação de coleções, que oferece recursos como, por exemplo, busca tardia, vista na seção anterior. Esta limitação, contudo não é um problema já que programar para interfaces é conhecida como sendo uma boa prática de programação.

3.8 Análise do Framework Horizontal: Spring

Spring é um *framework* de infra-estrutura adotado no nosso estudo de caso, o sistema Opus Ibank, para complementar o *Hibernate* fornecendo controle de transações, além de oferecer outros serviços como a exposição de serviços remotos. Este *framework* pode ser considerado como de infra-estrutura uma vez que oferece serviços ao nível de sistema, como gerenciamento de transações, e se integra também com outros *frameworks* de persistência, MVC entre outros. Além disso, o *Spring* é definido como um container leve ("*lightweight container*") de inversão de controle. O termo container é usado no sentido de que ele gerencia o ciclo de vida dos objetos configurados nele, e leve, pois as classes da aplicação tipicamente não possuem dependências com o *framework*, como por exemplo, elas não precisam estender nenhuma classe específica para que sejam inseridas ao *framework*.

O *Spring* é um *framework* dividido em vários módulos funcionais usados de acordo com as necessidades da aplicação. Os módulos de injeção de dependência, serviços remotos e programação orientada a aspectos foram usados no estudo de caso do capítulo 4.

As subseções seguintes descrevem os conceitos de inversão de controle e injeção de dependências, que são os principais conceitos por trás do Spring.

3.8.1 Inversão de Controle

Quando estamos desenvolvendo nossas aplicações seguindo o paradigma OO, dividimos a responsabilidade em diversos objetos. Nessa divisão de responsabilidades, terminamos por ter objetos que delegam um certo serviço a outro objeto. A figura abaixo mostra um diagrama de classes onde mostra a classe ForumFacade, representando o Façade (GOF, 1995) dos serviços de um fórum de mensagens e a classe MensagemDAOImpl e sua interface, que implementam o padrão Data Access Object responsável por efetuar as operações de persistência na base de dados. Note que apesar da classe ForumFacade

referenciar a interface MensagemDAO, ela é dependente da implementação da interface, pois precisa instanciá-la para poder usa-la. Para este exemplo, as coisas parecem simples, mas com o crescimento do Fórum novas dependências de objetos podem ser inseridas futuramente, ele poderia precisar enviar e-mails, acessar serviços remotos, entre outros. Desse modo não seria muito interessante manter todas estas dependências sendo inseridas manualmente no código, e é nesse contexto que entra então a inversão de controle.

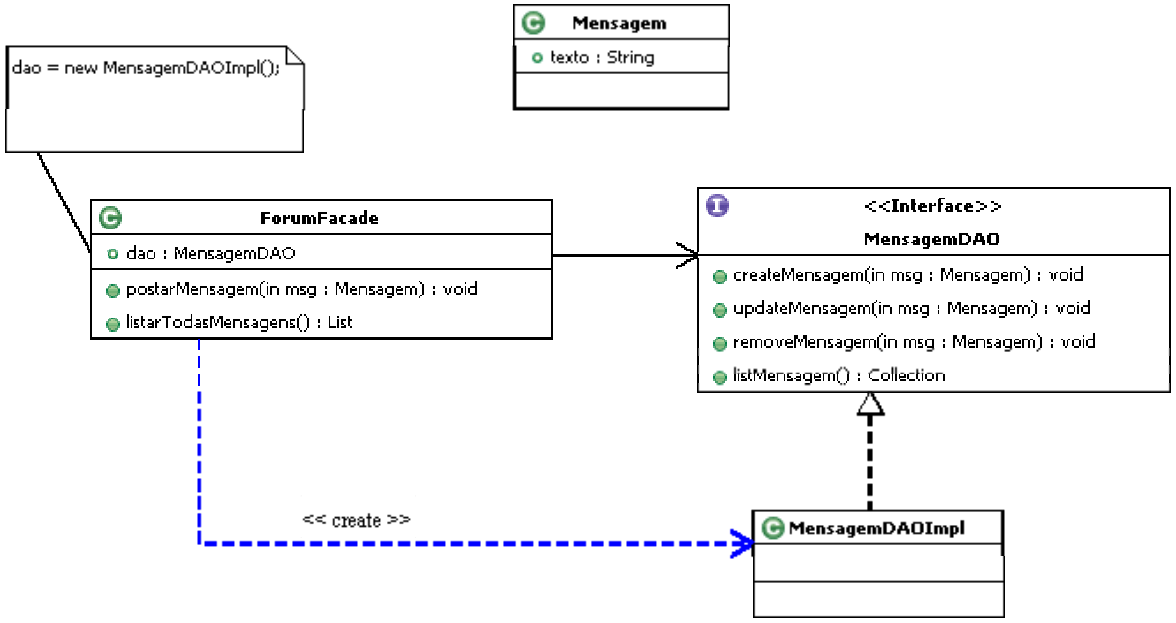

Figura 19 – Dependências configuradas sem injeção de dependências

O princípio de inversão de controle *(Inversion of Control* - Ioc) tem como objetivo oferecer uma maneira simples de prover dependências de objetos em forma de componentes e gerenciar o ciclo de vida dessas dependências. Containeres de IoC, como o *Spring*, servem para fazer a ligação entre dependentes e dependências, fazendo isso de várias maneiras diferentes. A IoC se subdivide em injeção de dependências (*Dependency Injection*) e busca por dependências (*Dependency Lookup*).

A busca por dependências é a maneira mais conhecida de IoC, nela os objetos procuram ativamente por suas dependências, como, por exemplo, quando fazemos busca em um contexto de nomes como o JNDI.

A injeção de dependências é uma nova abordagem recente da IoC, onde os objetos não procuram por suas dependências, elas são inseridas neles pelo container de IoC. Através da injeção de dependência, um terceiro objeto denominando Montador (*Assembler*) esta presente e sua responsabilidade é instanciar e configurar as dependências das classes relacionadas. Este montador injeta dependências de duas maneiras diferentes, através do construtor do objeto, passando dependências como argumentos do construtor, ou através

de métodos “get” e “set” da especificação javabean. A injeção de dependências é considerada a melhor maneira de se trabalhar com IoC, pois ela não polui o código com chamadas para um container, como ocorre na busca por dependências. Os objetos não ficam presos a uma implementação específica porque eles não “sabem” como as dependências foram parar ali, eles apenas as usam. A figura 20 abaixo exemplifica o mesmo modelo da figura 19, mas desta vez usando injeção de dependências.

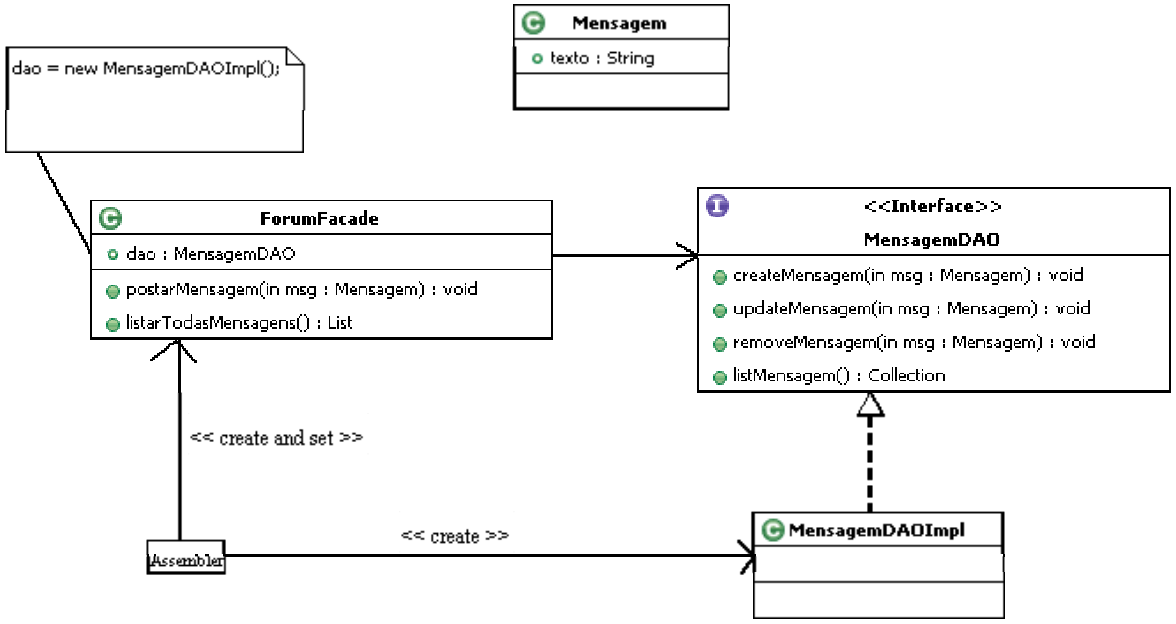

Figura 20 - Dependências configuradas com injeção de dependências

Com a introdução do montador a classe ForumFacade passa a depender apenas da interface MensagemDAO. O montador é responsável por instanciar a implementação adequada da interface e configurar a dependência da classe ForumFacade, desta forma consegue-se um baixo acoplamento entre o componente MensagemDAO e a classe que o usa.

O módulo injeção de dependências do framework Spring entra no lugar do montador, gerenciando a criação de componentes e a configuração de dependências que são descritos em um arquivo em formato XML.

3.8.2 IoC com o Spring

Para entender melhor a descrição feita na seção anterior vejamos um exemplo ilustrativo, utilizando o modelo de classes citado anteriormente.

A classe ForumFacade precisa de uma instancia para enviar mensagens ao fórum ao qual ela gerencia, o modo tradicional de fazer isso seria conforme ilustrado na listagem 12.

```
public class ForumFacade {
  private MensagemDAO dao;

  public void postarMensagem() {
     dao = new MensagemDAOImpl();
    (...)
  }
}
```

Listagem 12 – Classe ForumFacade

Apesar da abordagem acima não ser ruim, nós gostaríamos de abstrair MensagemDAO caso sua implementação mude, nesse caso utilizando talvez um objeto fábrica por exemplo.

```
public class ForumFacade {
  private MensagemDAO dao;

  public void postarMensagem() {
    dao =
      ApplicationFactory.getMensagemDAO();
    (....)
  }
}
```

Listagem 13- Classe ForumFacade refatorada

Em qualquer uma das duas abordagens acima a classe ForumFacade tem que saber como adquirir uma referencia a uma implementação de MensagemDAO.

A IoC segue uma abordagem diferente, com Ioc a classe ForumFacade declararia sua necessidade por um objeto MensagemDAO e o framework de IoC seria o responsável por injetar a dependência nesta classe. Isto significa que a classe ForumFacade não precisa nunca mais saber como adquirir uma referencia a um objeto que implementa MensagemDAO, resultando em um código mais limpo e mais flexível.

A listagem 14 abaixo ilustra uma injeção de dependência via método set. Neste tipo de IoC um arquivo de metadados é usado para resolver as dependências entre os objetos, e conforme dito anteriormente em spring este arquivo de metadados é um arquivo em formato xml de configuração.

```
public class ForumFacade {
  private MensagemDAO dao;

  public void setDAO(MensagemDAo dao) {
    this.dao = dao;
  }
  public void getDao() {
     return dao;
  }

  public void postarMensagem() {
    // faz algo com dão sem precisar instancia ou pega referencia com
    //outro
  }
}
```

Listagem 14 – ForumFacade sem precisar instanciar uma referencia

Note que com o uso de IoC a classe se parece com um bean normal.

Para inserir a referencia primeiro deve-se definir o bean MensagemDAO no arquivo xml de configuração.

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
         "http://www.springframework.org/dtd/spring-beans.dtd">
03: <beans>
04:  <bean id="mensagemDAO"  class="MensageDAOImpl">
05:  </bean>
06:  <bean id="ForumFacade"    class="ConferenceFacade">
07:          <property name=dao">
08:              <ref local="mensageDAO"/>
09:          </property>
10: </bean>
```

Listagem 15 – Arquivo de configuração de beans do spring

O trecho entre as linhas 4 e 5 define o bean mensagemDAO. O *id*, definido na linha 4, identifica unicamente este bean no contexto da aplicação Spring. A propriedade *class*, definida na linha 4 e 6, especifica a classe do bean. O trecho entre as linhas 6 e 10 declaram o bean ForumFacade. A dependência com o bean mensagemDAO é configurada no trecho entre as linhas 7 e 9. Na linha 7 é descrito que a propriedade que armazena a dependência se chama “dao” e na linha 8 é descrito que esta propriedade recebe o bean mensagemDAO, definido anteriormente no mesmo arquivo.

Como se pode ver, o módulo de *Dependency Injection* promove o acoplamento fraco entre componentes. Graças a este fraco acoplamento as classes são mais fáceis de testar unitariamente. As dependências são explicitadas e podem ser substituídos por mock objects, objetos que imitam objetos reais e os substituem em testes, dessa forma promove também um ambiente propicio ao desenvolvimento baseado em testes.

3.8.3 Outros Serviços do Spring

Além de promover o acoplamento fraco entre classes através do módulo de injeção de dependências, o Spring também fornece serviços de infra-estrutura como serviços remotos, e suporte a programação orientada a aspectos.

Eventualmente uma aplicação necessita expor as funcionalidades da camada de negócios de uma aplicação remotamente para outras aplicações e para outros clientes.Spring permite a exposição de seus beans de varias formas dentre elas RMI e Serviços Web.

RMI ou *Remote Method Invocation* é uma tecnologia que possibilita a chamada a métodos de componentes remotos, também usada no Enterprise JavaBeans. Sua principal vantagem é que ela é eficiente quando a comunicação é feita entre duas plataformas

Java. Modelos complexos de objetos podem ser enviados através do mecanismo de serialização da plataforma Java. Sua principal desvantagem é que ela usa portas arbitrárias para se comunicar, o que exige esforço extra de configuração em firewalls, e não possibilita a comunicação de Java com outras plataformas.

A tecnologia de serviços web possibilita a chamada de métodos remotos através da troca de arquivos XML, a descoberta automática de serviços dentre outras vantagens. Esta tecnologia é executada sobre o HTTP então firewalls não costumam ser um empecilho e, como utiliza um padrão bem conhecido, é viável realizar a comunicação entre diversas plataformas. Por outro lado, das duas tecnologias para exposição de serviços remotos citados, é a mais pesada e sua performance pode não ser adequada em alguns cenários.

Spring possibilita que serviços sejam expostos sem que seja necessário alterar o código fonte da aplicação. Para expor um serviço usando RMI, é preciso configurar o *Proxy* adequado no arquivo de configuração. A exposição de serviços como serviços web é mais trabalhosa, pois envolve também a criação do arquivo descritor do serviço WSDL.

Para fins de demonstração, a Listagem 16 apresenta o arquivo de configuração do Spring onde o serviço Conferência é exposto utilizando a tecnologia RMI.

```
01: <beans>
02: <!-- Declaração de Outros Beans -->
03: <bean id="ForumFacade" class="ForumFacadeImpl">
04:            <property name=”dao" ref="mensagemDAO"/>
05: </bean>
06: <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
07:     <property name="serviceName" value="Forum"/>
08:     <property name="service" ref="forumFacade"/>
09:     <property name="serviceInterface" value="ForumFacade"/>
10:     <property name="registryPort" value="1199"/>
11: </bean>
12: </beans>
```

Listagem 16 - Arquivo de Configuração do Spring: Exposição de Serviços Remotos.

O bean do fórum é declarado entre as linhas 03 e 05. Neste caso, considera-se que ForumFacadeImpl é a classe do fórum que implementa a interface ForumFacade. O uso de interfaces é obrigatório quando se lida com *Proxys* do Spring.

O trecho entre as linhas 06 e 11 declara o *Proxy* dinâmico responsável por expor o serviço Conferência remotamente usando RMI. Na linha 07 é associado um nome ao serviço que será usado para cadastrá-lo no registro RMI. Na linha 08 é especificado o bean cujos serviços são expostos pelo *Proxy*, no caso, o bean ForumFacade. Na linha 09 é declarada a interface que este *Proxy* dinâmico implementa e na linha 10, a porta em que o serviço e exposto.

O framework Spring possibilita que serviços sejam expostos declarativamente, sem que seja necessário alterar o código fonte. Como conseqüência, pode-se trocar o mecanismo de comunicação sem alterar o código, aumentando o reuso.

Finalmente, o módulo de programação orientada a aspectos do *Spring* compreende um *framework* de geração de aspectos bem integrado com o módulo de injeção de dependências. A programação orientada a aspectos é um paradigma que propõe uma forma de tratar interesses transversais, os aspectos, que costumam se espalhar pelo código em outros paradigmas como o da orientação a objetos. Um aspecto implementado no estudo de caso do capítulo 4 foi o *log* de atividades, usado para fins de depuração do sistema e registro de atividades de um usuário no sistema.

Capítulo 4

Arquiteturas Orientadas a Serviços (SOA)

4.1 Arquiteturas Orientadas a Serviços (SOA)

SOA é a denominação dada a um novo tipo de arquitetura onde softwares e rotinas são disponibilizadas como serviços numa rede de computadores, e que podem ser utilizados por diferentes aplicações e para vários propósitos. Idealmente, com este tipo de arquitetura, o desenvolvimento de novas aplicações se resumiria em selecionar os serviços disponíveis e encaixá-los numa determinada seqüência de execução, de acordo com as regras de negócio a serem atendidas.

O conceito mais importante em SOA é o conceito de serviços. Uma analogia que se pode fazer para auxiliar a compreensão sobre o conceito de serviços é o brinquedo Lego. Cada peça do Lego representa um serviço, onde a partir da combinação de peças eu consigo montar casas, navios, entre outras coisas.

A montagem do Lego consiste em reunir diversas peças de diferentes formatos, encaixa-los entre si e obter no final uma figura totalmente nova. Existem peças que provavelmente serão utilizadas em qualquer uma das montagens, como as peças retangulares que costumam vir em maior número no brinquedo, outras peças, entretanto, têm funções mais especificas e serão utilizados dentro de sua especialidade, dificilmente se adaptando a outros contextos, como por exemplo, um eixo de automóvel será utilizado na construção de carros e caminhões, mas com certeza não será usado na montagem de casas.

Apesar de simples podem se extrair algumas características básicas do que são serviços: (1) serviços são reutilizáveis, (2) alguns são mais específicos outros de uso mais geral, e (3) serviços podem interagir com outros serviços.

Tendo a idéia de peças de Lego em mente, o principio que rege SOA é de que uma aplicação grande e complexa deve ser evitada e substituída por um conjunto de aplicações pequenas e simples. Ou seja, uma aplicação passa a ser fisicamente composta por vários e pequenos módulos de software especializados, distribuídos, podem ser acessados remotamente, interoperáveis e reutilizáveis, que podem ser reunidos para formar o processo desejado.

Em arquiteturas orientadas a serviços esses pequenos módulos é que são chamados de **Serviços**. Um serviço é um modulo de software que possui uma interface que descreve quais as funções que ele oferece e permite que ele possa invocar um serviço assim como possa ser invocado.

Todas as funções em SOA são, portanto agregadas como serviços reutilizáveis; SOA é o contrato para identificação de serviços contendo regras sobre como acessá-los. Todas as informações sobre requisições e respostas, condições de exceção e funcionalidades são definidas como parte de uma interface. A interface contém informações necessárias para que um serviço possa ser acessado sem a necessidade de conhecer sua implementação, linguagem ou plataforma de implementação. A figura 21 abaixo ilustra como fica a arquitetura de um sistema com a utilização de SOA.

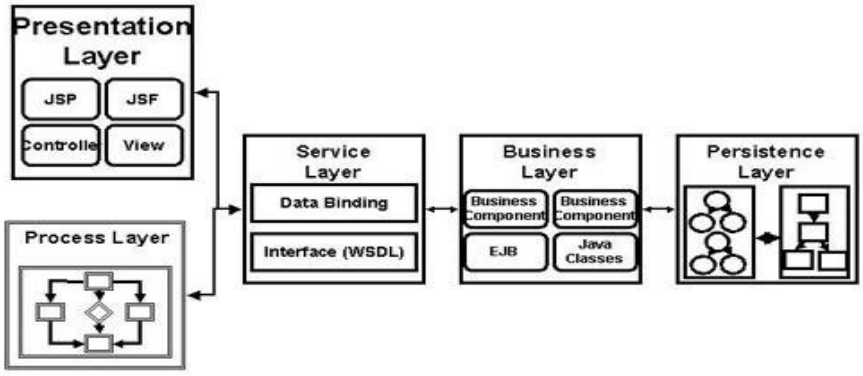

Figura 21 – Arquitetura em camadas com SOA

A idéia de arquiteturas orientadas a serviços não é nova, desde os anos 80 houve várias tentativas de implementar sistemas baseados em SOA. Entretanto nenhuma das implementações realmente foi à frente, principalmente devido a três fatores:

(1) Não havia softwares para integração de aplicativos (*middleware*) padronizados e abertos (não-proprietários);
(2) Não existiam definições de interfaces padronizadas para permitir a conexão entre os módulos;
(3) Não havia compatibilidade e integração entre os produtos dos diferentes fornecedores.

Na busca por novas formas de utilizar a arquitetura orientada a serviços, foi criado, em setembro de 2000, o grupo de trabalho W3C. Formado por membros das maiores empresas de software do mundo, tais como Microsoft, IBM, Oracle e Sun, o grupo definiu uma nova arquitetura computacional, com condições de melhorar o suporte e aprimorar e agilizar a interação entre processos de negócio, e, por conseguinte, entre empresas. Essa arquitetura, denominada Web Services, é baseada no conceito de distribuição e modularidade, adotando protocolos abertos e padronizados para promover a integração de aplicações com baixo acoplamento.

Um *web service* pode ser entendido como um componente que possui suas funcionalidades acessíveis pela rede através de mensagens baseadas em XML. A

disponibilização das operações e a descrição do serviço também ocorrem através do padrão XML. O arquivo descritor do serviço possui todas as informações necessárias para que outros componentes possam interagir com o serviço, incluindo o formato das mensagens (para as chamadas aos métodos do serviço), protocolos de comunicação e as formas de localização do serviço. Um dos maiores benefícios dessa interface é a abstração dos detalhes de implementação do serviço, permitindo que seja acessado independente da plataforma de hardware ou software na qual foi implementado. Como as mensagens trocadas para a comunicação são baseadas no padrão XML, também temos a flexibilidade com relação à linguagem de programação tanto na implementação do serviço quanto no componente que acessará o *web service*. Estas características permitem e motivam a implementação de SOA uma vez que a tecnologia de web services ajuda a superar muitos dos obstáculos necessários para elaboração de arquiteturas orientadas a serviços.

Resumidamente em uma arquitetura SOA, uma camada de serviços agrega componentes e funcionalidades relacionadas a um ou mais processos. Essa camada é publicada em rede e pode ser invocada de forma remota por aplicações clientes ou outros serviços de software. A composição dos serviços oferecidos por essa camada caracteriza uma aplicação SOA. Os principais benefícios de uma arquitetura orientada a serviços implementada através de *web services* são:

1) **Interoperabilidade**: as aplicações clientes de uma aplicação SOA podem estar implementadas em qualquer plataforma de software e hardware, distinta da plataforma onde a camada de serviços foi implementada.
2) **Reusabilidade**: os serviços e funcionalidades oferecidos por uma aplicação SOA são altamente reutilizáveis, pois podem ser usadas em mais de uma aplicação.
3) **Mobilidade**: uma aplicação SOA é acessada remotamente e sua localização deve ser descoberta de forma dinâmica e transparente por aplicações clientes, dessa forma é possível realocar uma aplicação SOA sem comprometer a disponibilidade da aplicação.

Entretanto o uso de SOA não aplicável em casos onde:

- Para aplicações *stand-alone* naturalmente não são distribuídas, como um processador de textos por exemplo.
- Para aplicações que não necessitam fazer uso de outros serviços ou que não têm serviços a oferecer, a serem reutilizados.
- Para aplicações que requerem complexas interfaces gráficas onde o volume de busca por informações seja grande, pois SOA é uma arquitetura para sistemas distribuídos com baixo tempo de respostas.

- Para aplicações cujo fluxo de execução é não estruturado.

4.2 Arquitetura Técnica de Web Services

Uma definição técnica de *web services* poderia ser como um serviço disponibilizado na Internet, descrito via WSDL, registrado via UDDI, acessado utilizando SOAP e com os dados transmitidos sendo representados em XML. A seguir, encontra-se uma breve explicação de algumas tecnologias que compõem serviços web.

SOAP (*Simple Object Access Protocol*) é um protocolo para troca de informações em ambiente distribuído. É baseado em definições XML e utilizado para acessar *web services*. Esse protocolo encapsula as chamadas e retornos aos métodos dos *web services*, sendo utilizado, principalmente, sobre HTTP.

WSDL (*Web Services Description Language*) é a linguagem de descrição de *web services* baseada em XML. Ela permite, através da definição de um vocabulário em XML, a possibilidade de descrever serviços e a troca de mensagens. Mais especificamente é responsável por prover as informações necessárias para a invocação do *web service*, como sua localização, operações disponíveis e suas assinaturas.

UDDI (*Universal Description, Discovery and Integration*) é uma das tecnologias que possibilitam o uso de *web services*. Uma implementação de UDDI corresponde a um *Web Service registry*, que provê um mecanismo para busca e publicação *web services*. Um *UDDI registry* contém informações categorizadas sobre os serviços e as funcionalidades que eles oferecem, e permite a associação desses serviços com suas informações técnicas, geralmente definidas usando WSDL. Como dito anteriormente, o arquivo de descrição em WSDL descreve as funcionalidades do *web service*, a forma de comunicação e sua localização. Devido ao modo de acesso, um *UDDI registry* também pode ser entendido como um *web service*. A especificação UDDI define uma API baseada em mensagens SOAP, com uma descrição em WSDL do próprio *web service* do servidor de registro. A maioria dos servidores de registro UDDI também provê uma interface de navegação por *browser*.

A arquitetura de *web services* se baseia na interação de três entidades: provedor do serviço (*service provider*), cliente do serviço e servidor de registro (*service registry*). De uma forma geral, as interações são para publicação, busca e execução de operações. A figura 22 ilustra estas operações, os componentes envolvidos e suas interações.

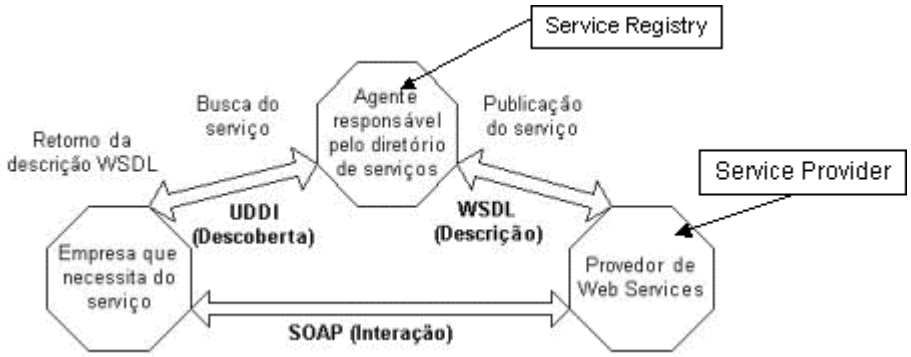

Figura 22 – Arquitetura de web services, operações e interação entre entidades.

O provedor do serviço (*service provider*) representa a plataforma que hospeda o *web service* permitindo que os clientes acessem o serviço. O cliente do serviço é a aplicação que está procurando, invocando ou iniciando uma interação com o *web service*. O cliente do serviço pode ser uma pessoa acessando através de um *browser* ou uma aplicação realizando uma invocação aos métodos descritos na interface do *web service*. O *service registry* representa os servidores de registro e busca de *web services* baseados em arquivos de descrição de serviços que foram publicados pelos *service providers*. Os clientes (*service requestor*) buscam por serviços nos servidores de registro e recuperam informações referentes a interface de comunicação para os *web services* durante a fase de desenvolvimento ou durante a execução do cliente, denominados *static binding* e *dinamic binding*, respectivamente.

4.3 Aplicação de SOA: *EAI – Enterprise Application Integration*

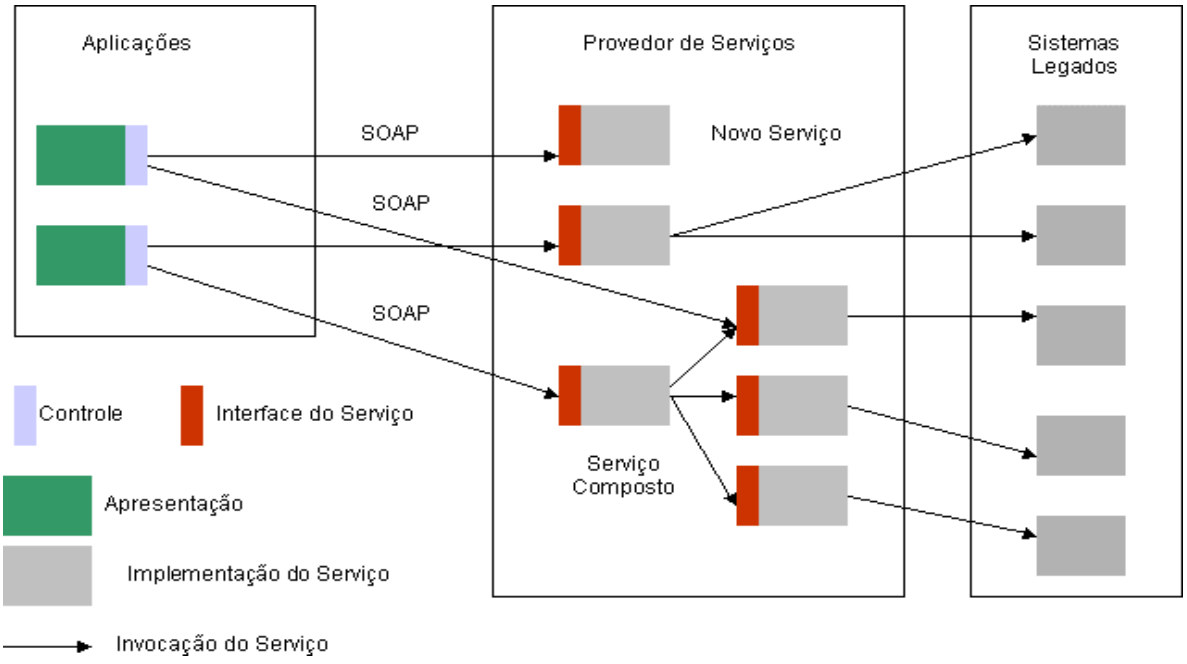

Figura 23 – Arquitetura de acesso a sistemas legados

Uma das áreas onde arquitetura orientada a serviços mais vem sendo empregada é a integração de aplicações corporativas (*EAI – Enterprise Application Integration*), onde as aplicações legadas já constituídas estão executando em diversas plataformas, desde linux em micro computadores até OS/390 em *mainframes*.

Muitas vezes, os sistemas legados são antigas aplicações de software existentes nas organizações que são vitais para o funcionamento do negócio. No entanto, o uso de sistemas legado gera dificuldades para a adaptação da infraestrutura de TI a novas demandas do negócio. Muitas vezes, as regras de negócio embutidas nestes sistemas não são inteligíveis nem para analistas mantenedores do código, devido às inúmeras alterações que o software sofre ao longo de seu ciclo de vida. Assim, uma tecnologia que possa adicionar novas funcionalidades a sistemas legados, sem que seja necessário retirá-los de produção ou decifrar milhares de linhas de código de difícil manutenção, pode aumentar a adaptabilidade e agilidade de uma empresa, e, ao mesmo tempo, reduzir significamente o custo do desenvolvimento e manutenção de sistemas. Para tanto, criam-se adaptadores que acessam as aplicações legadas e se comuniquem através do protocolo definido pelo sistema. A cada nova funcionalidade desejada bastaria adicionar um novo serviço sem ter que alterar as funcionalidades do sistema legado. A figura 23 ilustra a explicação dada nesta seção.

Capítulo 5

Atividades realizadas, Resultados obtidos e Conclusões

5.1 Estudo de caso: Sistema Opus Ibank

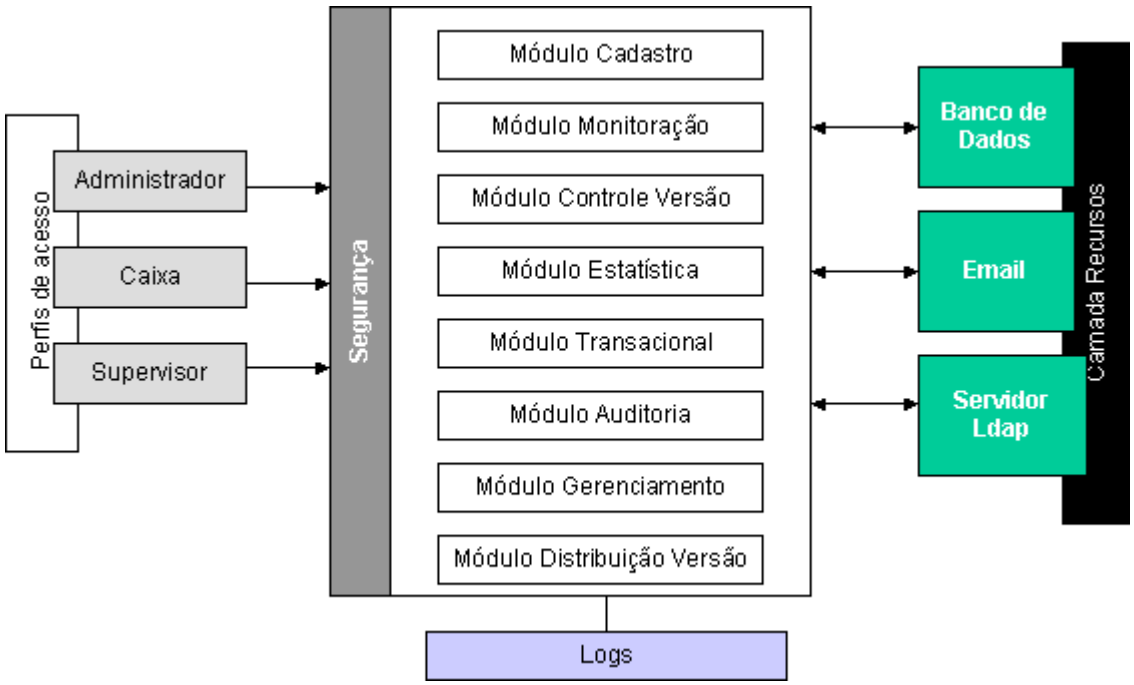

Figura 24 – Visão Macro de processos do Sistema Opus Ibank

O sistema opus ibank é um software de automação bancária que visa o gerenciamento operacional entre todos os terminais da agência e suas funções. Um terminal em um banco é todo dispositivo de atendimento presente em um banco, como terminais de caixa, de auto-atendimento e de atendimento ao cliente, cuja principal função é permitir que um cliente realize operações de movimentação financeira ou consulta, auxiliado por funcionários do banco ou não.

Esta diversidade de dispositivos na rede, torna a integração difícil entre eles, pois cada um deles possui formas particulares de comunicação. E é justamente esta integração entre os diversos dispositivos, além do gerenciamento de todas as informações operacionais de um banco, como transação bancária, cadastro de funcionários e gerenciamento de equipamentos da rede, que o opus ibank se propõe a resolver.

A solução proposta é modular, cuja localização dos componentes que integram o sistema, funciona distribuído em cada um dos equipamentos terminais e servidor da rede. O

terminal ibank, que correspondem a cada um dos terminais da agência, é um dispositivo que tem instalado os módulos transacionais, auditoria, gerenciamento e distribuição de versão. Já o servidor ibank é um equipamento que tem em sua instalação o conjunto formado pelos módulos cadastros, monitoração, estatística e controle de versão.

Foi utilizado na implementação um conjunto amplo de tecnologias e ferramentas open-source Java: *frameworks Hibernate*, *Java Server Faces e Spring,* API *Java reflection*, APIs de criptografia, EJB, *Web Services*, Log4J, apache *Commons Chain* e *Commons BeanUtils*, e programação orientada a aspectos.

Como meu trabalho se deu no lado servidor do ibank desde a fase inicial de elaboração da arquitetura até a fase de implementação, neste capítulo apresentarei como a arquitetura do servidor ibank foi implementada através da agregação dos frameworks citados no capítulo 3, além de descrever outras atividades realizadas durante a fase de desenvolvimento do sistema. A figura 24 ilustra uma visão macro de processos do ibank.

5.2 Desafios e Soluções

Alguns dos desafios com relação ao desenvolvimento deste sistema incluíam:

- Alto desempenho
- Curto período de desenvolvimento e implantação
- Múltiplos canais de atendimento como agências, Atms, internet banking, Pabs entre outros.
- Necessidade de infra-estrutura de segurança incluindo controle de acesso e criptografia de dados.
- Controle de transação da movimentação bancária
- Arquitetura distribuída com comunicação via Terminal-Servidor e Servidor-Servidor.
- Necessidade de um mecanismo de auditoria
- Gerenciamento remoto de equipamentos, por exemplo, eu gostaria de poder desativar um equipamento da rede caso algum funcionário tente fazer uma tentativa ilegal de acesso, ou gostaria de poder fazer testes remotos de funcionamento de equipamentos, tudo isto através de interface web.

Além disso, o desenvolvedor necessitava entender todas as regras do domínio da aplicação, dessa forma os desafios principais com relação ao projeto deste sistema era o entendimento do domínio e como lidar com questões de infra-estrutura. O projetista do sistema deveria se concentrar mais nos aspectos funcionais do domínio da aplicação, utilizando uma infra-estrutura que tratasse de questões técnicas que são independentes de domínio.

A solução foi a elaboração de uma arquitetura técnica multicamada, que faz uso do padrão *model-view-controller* e que integra *frameworks* para fornecer a infra-estrutura necessária para tratar destas questões técnicas. A utilização de frameworks de infra-estrutura proporcionaria uma maneira de lidar com questões de baixo nível como persistência de dados, segurança, serviços remotos, controle de transações, entre outros, a partir de uma visão mais alto nível, além de tornar a aplicação modularizada e dessa forma o desenvolvedor poderia se focar mais nos aspectos funcionais do domínio da aplicação, deixando o tratamento de aspectos de infra-estrutura para outros *frameworks* especialistas.

5.3 Arquitetura Técnica do Servidor ibank

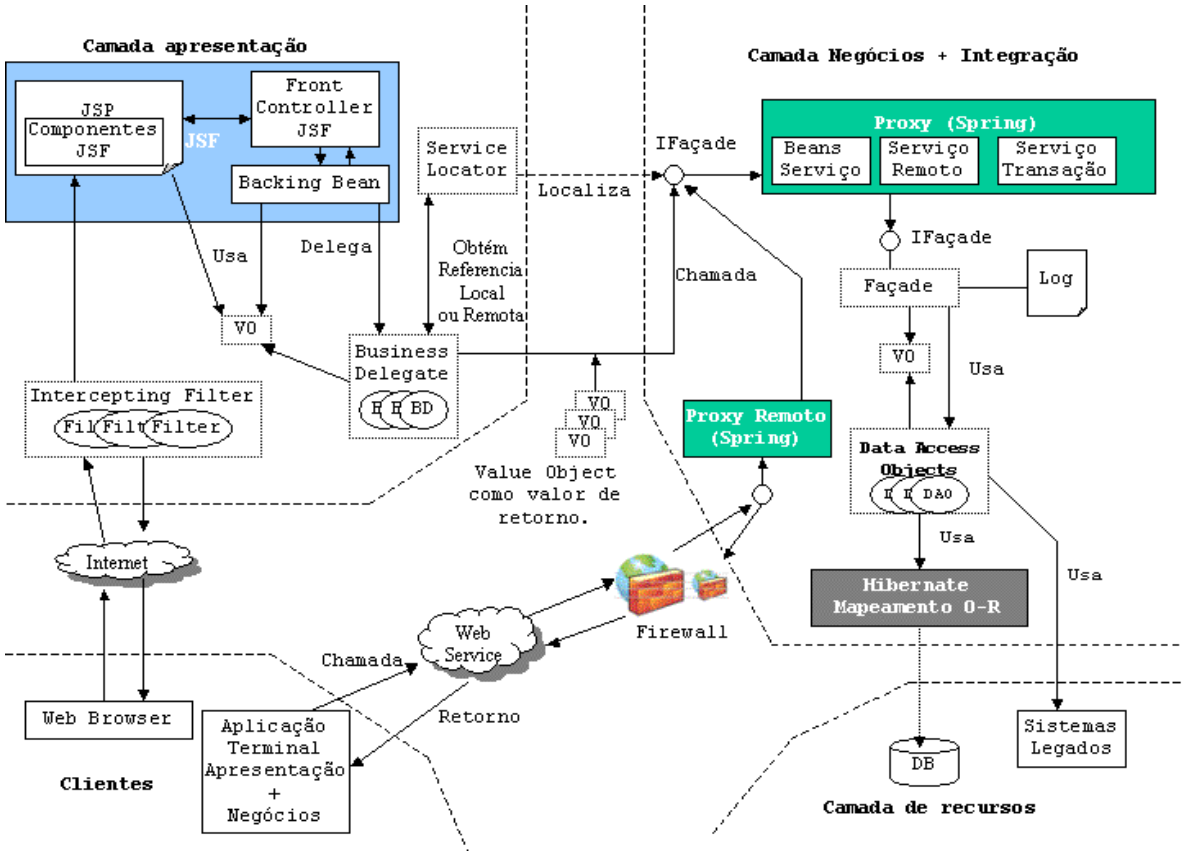

Figura 25 – Arquitetura técnica do Servidor ibank com os frameworks

A arquitetura do servidor ibank foi implementada segundo uma arquitetura técnica que usa uma abordagem em três camadas e também o padrão MVC, sendo ilustrada conforme na figura 25 acima. As setas indicam o fluxo de controle da aplicação, retângulos representam classes e círculos representam interfaces, além disso, linhas pontilhadas representam a divisão entre as camadas.

A camada de recursos relaciona os recursos externos necessários para a execução da aplicação. Na arquitetura do ibank os recursos externos incluíam um banco de dados, um

servidor ldap (não mostrada na figura acima) para se realizar autenticação de usuários e um sistema legado.

O *Spring* e o *Hibernate* formam a base dos *frameworks* de infra-estrutura que oferecem suporte ao desenvolvido de componentes da camada de negócios, fornecendo serviços de infra-estrutura como persistência de dados, controle de transações, serviços remotos, entre outros.

Na camada de apresentação, a visão é construída com paginas JSP compostas por componentes JSF. Estes podem ser componentes de validação, de conversão de dados e de interface com o usuário. O Servlet JSF Controller é usado como controlador do MVC, sendo que cada vez que ao receber uma requisição o controlador chama o backing bean que acessa a camada de negócios.

A camada cliente pode interagir diretamente com os serviços disponibilizados pela camada responsável por implementar a lógica de negócios, entretanto esta interação direta expõe a API destes serviços, tornando a camada de apresentação mais vulnerável a mudanças efetuadas nos métodos de negócio, dessa forma é desejável reduzir o acoplamento entre essas duas camadas, escondendo ao máximo detalhes de implementação. O padrão para resolver este problema é o **Business Delegate** que age como uma abstração, no lado cliente, para métodos de negócio. A utilização deste padrão em conjunto com o padrão **Service Locator**, cuja função é esconder detalhes de busca de beans de serviço no spring, tornava transparente à camada de apresentação detalhes de busca de serviços, além de poder realizar um tratamento de exceções lançadas pelos serviços de negócio, onde caso um problema ocorra em uma chamada a métodos de negócio, uma mensagem mais amigável pode ser produzida e repassada a camada de apresentação.

A entrada na camada de negócios é feita através do *Proxy* do Spring. Como o Proxy implementa a mesma interface do *Façade*(interface IFaçade) o código cliente não precisa ser modificado para usar o *Proxy* em vez da implementação direta do *Façade*. Outra característica do Proxy do spring é a agregação de serviços como controle de transação e serviço de log, em cada classe que implementa o Façade gerada pelo Proxy dinâmico do spring.

Por fim o acesso à base de dados é encapsulado através de classes que realizam o padrão de projetos Data Transfer Object(DAO), desta forma a maneira de persisitir pode variar trocando componentes, sem que seja preciso reescrever código cliente. Os DAOs foram construídos usando o Hibernate, que possibilita o mapeamento de objetos em base dados relacionais, e deixando aberta a possibilidade de se construir daos a partir de uma base de consultar de sistemas legados, essa solução somente ocorreria caso a empresa que comprasse a solução exigisse isto como requisito.

A arquitetura do ibank precisava expor alguns beans de serviço, que continham lógica de negócios, ao Terminal do ibank, como o servidor e o terminal eram aplicações distintas que precisam conversar remotamente, a solução encontrada foi o uso de serviços web como solução para serviços distribuídos.

Conforme se viu no capítulo 3, o Spring possibilita a exposição de serviços de diversas formas. A forma escolhida a princípio é por serviços web(Web Services), pois possibilitam a integração com clientes escritos em diversas linguagens de programação além do Java, além disso, como o spring possibilita que outros proxys remotos sejam acrescentados, dando suporte a outros tipos de protocolos, como o RMI, por exemplo, sem precisar alterar o código do bean que eu quero expor remotamente, a solução provida pelo spring se mostrou muito interessante, aumentando em muito o reuso ao nível de reuso de código, pois a arquitetura não sofre mudanças substrancias com o acréscimo de serviços remotos. A principal desvantagem do uso de serviços web que se pode notar na pratica, foi com relação ao desempenho, o uso de um protocolo como SOAP para trocar dados entre cliente e servidor ser mostrava muito lento mesmo em ambientes de rede intranet, fator que em algumas ocasiões incomodou alguns usuários do sistema.

5.4 Arquitetura orientada a serviços (SOA) e o ibank

O ibank era um sistema no qual o terminal e o servidor eram aplicações distintas que conversavam entre si, como exemplo o terminal precisava realizar operações de movimentação financeira via uma chamada ao servidor ibank, que acessava o sistema legado com as devidas regras de negócio necessárias ao acesso, encapsuladas pelo servidor, de forma que este serviço era apenas um de uma coleção de serviços independente, que em certos casos se comunicava com outros serviços a fim de gerar uma resposta final, e que funcionava em uma arquitetura distribuída. A partir dessa visão imaginou-se então a elaboração de parte da arquitetura do ibank, como uma coleção de serviços publicáveis, que poderiam ser consumidos e combinados para gerar resultados, dessa forma o ibank começou a migrar para um sistema com uma arquitetura parcialmente orientada a serviços (SOA), e a tecnologia escolhida para implementar esta coleção de serviços foi a tecnologia de *web services*, por suas características de fácil integração, interoperabilidade entre sistemas e um protocolo de comunicação independente de especificações proprietárias de um fornecedor de tecnologia específico.

Um *web service* pode ser entendido como um componente que possui suas funcionalidades acessíveis pela rede através de mensagens baseadas em XML (SOAP), cuja disponibilização das operações e a descrição do serviço também ocorrem através do padrão XML. O arquivo descritor do serviço possui todas as informações necessárias para que outros componentes possam interagir com o serviço, incluindo o formato das

mensagens, protocolos de comunicação e as formas de localização do serviço. Como as mensagens trocadas para a comunicação são baseadas em XML, temos flexibilidade com relação à linguagem de programação tanto na implementação do serviço quanto no componente que acessará o *web service*. Dessa forma um *web service* é implementado para disponibilizar uma determinada funcionalidade visando a reusabilidade do serviço e a interoperabilidade com outros sistemas.

Nesse contexto é que se imaginou o ibank com a possibilidade de um sistema que utilizasse um ou mais *web services* para realizar consultas a regras de negócios a sistemas legados, dessa forma como a consulta ao legado era uma operação necessária a ser executada por varias aplicações distintas é que se propôs uma modelagem de acesso a legado nos moldes de SOA.

5.5 Metodologias usadas durante o desenvolvimento da arquitetura do ibank

Durante o período de modelagem do sistema até a fase de implementação foram usadas algumas metodologias de desenvolvimento que merecem destaque, dentre elas a técnica de prototipação durante a fase inicial de projeto de interfaces do sistema, o uso de geradores de código para modelagem relacional do banco de dados do sistema, e o uso de programação orientada a aspectos para lidar com alguns problemas.

5.5.1 Prototipação

Antes da fase de elaboração do design de software, era importante se projetar como seria a interface do sistema, para isso foram realizados protótipos em PowerPoint para fins de discussão e comunicação de idéias. Geralmente se diz que os usuários não conseguem transmitir o que querem, mas quando vêem algo e começam a utiliza-lo, logo sabem o que não querem, foi pensando nisso que foram feitos protótipos da interface do sistema, eles se mostraram muito úteis para esclarecer alguns requisitos vagos, para se verificar se o rumo que a equipe de desenvolvimento do ibank tomou no design da interface é compatível como o resto do desenvolvimento do sistema e ele se mostrou muito útil como ferramenta para realização de estudos de caso.

Esta com certeza foi a fase que mais me surpreendeu na empresa, geralmente desenvolvedores costumam delegar o projeto de interfaces para a fase de implementação, sem levar em consideração um estudo da melhor forma de interação com o usuário, esse aspecto de desenvolvimento no qual se desenvolve primeiro em como vai ser a interação com o usuário, para depois pensar em como vai ser a parte técnica do sistema, facilitou muita a compreensão de como era o domínio da aplicação. A figura 26 abaixo ilustra uma pequena amostra de um dos protótipos feitos.

Tela /OpusiBank/Monitoração/Operação/Diagnóstico/Seleciona Função

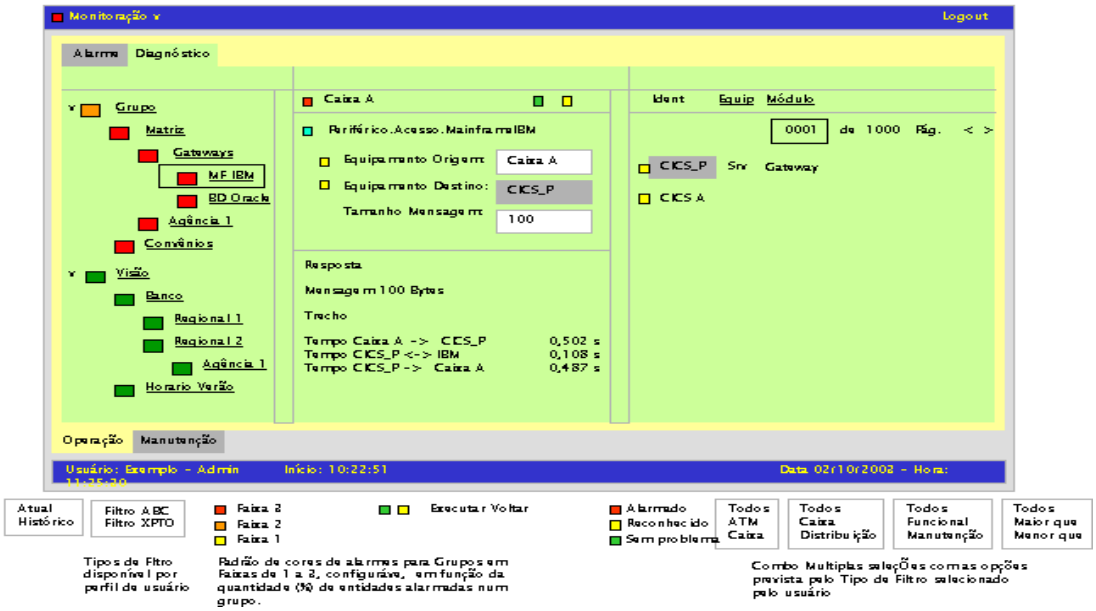

Figura 26- Protótipo de interface do ibank

5.5.2 Programação orientada a aspectos com AspectJ

A programação orientada a aspectos é um paradigma que propõe uma forma de tratar interesses transversais, os aspectos, que costumam se espalhar pelo código em outros paradigmas como o da orientação a objetos.

A ferramenta escolhida para geração dos aspectos no ibank foi o AspectJ do projeto eclipse.

Um aspecto implementado no ibank foi o *log* de atividades, usado para fins de depuração do sistema, registro de atividades de um usuário e auditoria no sistema. Esse é um caso bem comum de uma funcionalidade que se espalha por todos os códigos das classes de uma aplicação, desse modo o uso de programação orientada a aspectos contribui pela facilidade de melhora da legibilidade do código e alterações na política de log. Duas variações de aspectos do log foram implementadas o log de atividades, que nada mais era a exibição no log do servidor de aplicação de cada chamada de método efetuada pelo usuário, e o log de auditoria que era o registro do movimento financeiro efetuado em cada um dos terminais de caixa do ibank. As figuras 27 e 28 ilustram esse caso.

Outro aspecto implementado foi a questão da autorização e autenticação, em determinadas classes da aplicação era necessário que antes de uma chamada a um método de negócios o usuário tivesse feito a autenticação e possuísse as credencias necessárias para efetuar a chamada de método, como por exemplo, em classes que efetuavam operações financeiras era necessário verificar antes de realizar a operação, se

o usuário havia se autenticado e possuía as permissões necessárias para poder efetuar a operação.

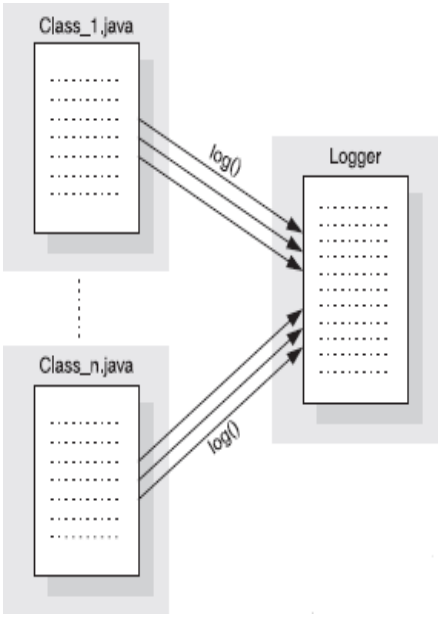

Figura 27 – Registro de log do modo convencional

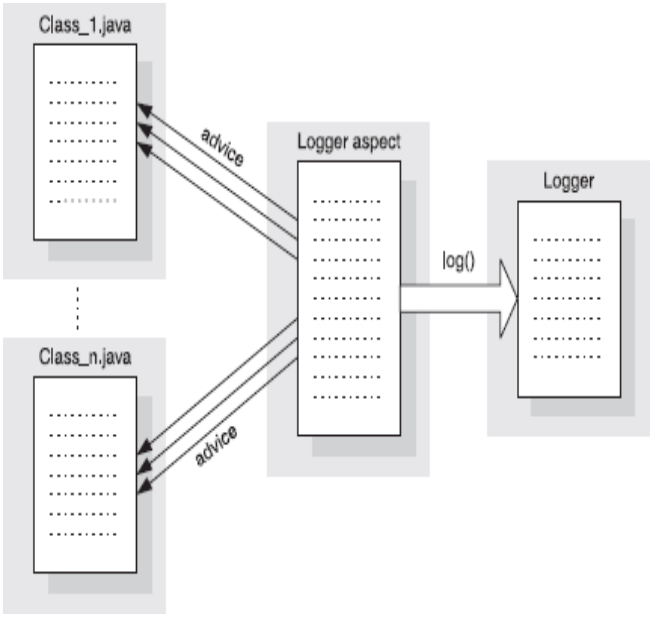

Figura 28 – Registro de log baseada em aspectos

5.5.3 Geradores de código

Os geradores de código foram muito importantes na fase de modelagem relacional da base dados, foi utilizado um gerador de código, o *PowerDesigner* da *Sybase*. A partir da especificação de alto nível do modelo relacional da aplicação, o gerador de código permitia o tratamento de um problema sem a preocupação de para qual base de dados eu precisaria deveria gerar o SQL, pois o gerador faria isso para min, dessa forma o reuso provido pelos geradores de código deu um grande impulso inicial na fase inicial de implementação, permitindo o tratamento de um problema a partir de uma visão mais alto nível.

5.6 Resultados Obtidos

A meta de todo desenvolvedor de sistema é conseguir elaborar uma arquitetura que seja o mais flexível e extensível possível, modular e de fácil manutenção. Tanto frameworks como padrões e outras abordagens de reuso se mostraram muito importantes tanto na fase de implementação, quanto na fase de elaboração e modelagem do sistema, para o alcance das metas almejadas.

O resultado foi um sistema bem flexível, com período de desenvolvimento bem curto por módulo de desenvolvimento e com características funcionais como independência de

banco de dados, flexível com relação aos vários protocolos de serviços remotos, extensível para inserção de novos serviços, fácil manutenção e com uma arquitetura padronizada e com possibilidade de reutilização para futuros projetos da empresa.

5.7 Conclusões

O objetivo deste trabalho, conforme dito anteriormente, é abordar o reuso de software no processo de desenvolvimento do produto, baseado em conceitos fundamentais da área e em técnicas a serem aplicadas no processo de desenvolvimento.

Para se atingir este objetivo, foi feito um estudo das principais abordagens de reuso de software, apresentado técnicas, e finalizado com uma análise de dois modelos de desenvolvimento, os modelos cascata e espiral, aplicando o reuso em cada uma das suas etapas do processo e mostrando que independente do modelo de desenvolvimento o reuso de software pode ser inserido no processo, ajudando desenvolvedores a entregar com mais rapidez e qualidade o produto de desenvolvimento.

O objetivo do reuso de software é o aumento da produtividade e redução no esforço de desenvolvimento de novos produtos, por parte dos desenvolvedores. Entretanto apesar de ser um objetivo de fácil entendimento, fatores como desconhecimento de técnicas de reuso, falta de ferramentas especializadas, infra-estrutura para reuso e fatores humanos são problemas adicionais que tornam complexa a realização de um processo de desenvolvimento com base em reuso. Dessa forma as principais contribuições desse trabalham foram dar um panorama inicial geral do que é o reuso de software e apresentado técnicas que podem ser aplicadas visando atingir os objetivos do reuso, para isso foi apresentada a técnica de reuso com base em *frameworks*.

Framework é uma técnica de reuso que incorpora uma infra-estrutura de um sistema, que é a abstração de vários sistemas relacionados entre si, englobando um conjunto de funcionalidades comuns a este conjunto de sistemas. Com o uso de *frameworks* se obtém não só reuso de código, mas reuso de análise e projeto, além de ser um sistema formado por um conjunto de padrões em sua arquitetura.

O reuso provido pelos *frameworks,* quando comparados a outras técnicas de reuso, como padrões e biblioteca de rotinas, se mostra mais eficiente, pois sua funcionalidade é mais específica para domínio que os padrões(que são estruturas mais genéricas que os *frameworks*) e que as bibliotecas de rotinas(bibliotecas são reutilizadas, mas com todo o controle da aplicação nas mãos do desenvolvedor, o que ocasiona uma maior propensão a erros que no uso de *frameworks*).

Apesar das vantagens, os frameworks apresentam como desvantagens, custo inicial de treinamento e desenvolvimento e sua depuração e correção de erros é mais difícil. Porém todas as desvantagens são superadas pelas vantagens, pois uma vez que se aprende um

framework novos projetos são desenvolvidos com maior produtividade por parte do desenvolvedor.

A arquitetura orientada a serviços é uma técnica de reuso mais atual, que tem como principal objetivo o reuso intenso dos seus componentes (serviços), para que em médio prazo, a tarefa do desenvolvimento de uma aplicação seja primordialmente a tarefa da composição e coordenação dos serviços já implementados, aumentando o reuso e diminuindo o dispêndio de recursos, uma forma de se ver SOA é como uma nova proposta para o desenvolvimento orientado a componentes.

Apesar desta monografia apresentar somente duas técnicas de reuso com base em *frameworks e* SOA, a verdade é que em qualquer projeto o desenvolvimento de sistemas é feito a partir de uma serie de técnicas de reuso que vão desde a fase de modelagem e especificação do sistema até a fase de implementação, como foi o caso do estudo de caso do sistema opus ibank, sempre com o objetivo principal do aumento da produtividade dos desenvolvedores.

Como futuras extensões do trabalho podem ser vistas outras técnicas de reuso como um aprofundamento maior em arquiteturas orientadas a serviços, engenharia de software baseada em componentes entre outras. Podem ser vistas o caso de uma criação de estrutura de banco de dados para catalogação, classificação e recuperação de soluções de reuso previamente estudadas, projetadas e aplicadas melhorando dessa forma o ciclo de desenvolvimento.

Através deste trabalho espero que desenvolvedores possam ter um maior esclarecimento dos conceitos básicos em questão e que sirvam de motivação para estudo e pesquisas sobre o tema de reuso de software.

Parte III

Parte Subjetiva

Capítulo 6: BCC e o Estágio

Capítulo 6

BCC e o Estágio

6.1 Desafios e Frustrações

O estágio realizado na empresa Opus no decorrer deste ano de 2006, me proporcionou um grande amadurecimento profissional. Quando entrei na empresa um dos desafios do trabalho foi com relação à parte tecnológica, foi necessário se aprender um conjunto grande de tecnologias em prazo pequeno de tempo, a plataforma Java J2EE é muito rica com muitas soluções e projetos open-source para o desenvolvimento de aplicativos voltados para servidores.

Outro desafio que tive foi com relação a como modelar um projeto como o opus ibank, para se ter uma idéia foram gastos cerca de três meses somente na modelagem do sistema. Durante a fase de modelagem fazíamos muitas reuniões semanais com nosso chefe, reuniões que duravam cerca de três a quatro horas, além disso, como o projeto foi feito em conjunto com outras duas empresas (sibis software e amatera) era preciso estar em sintonia com o que eles estavam fazendo, apesar de um enorme tempo gasto com reuniões acredito que foi nesse período que desenvolvi melhor minha comunicação verbal e aprendi mais sobre como desenvolver um software real na prática. Vale ressaltar que desenvolver um sistema na prática é bem diferente do que quando fazemos um exercício programa no IME, aspectos como levantamento de requisitos, mudanças de prazo e escopo do projeto, documentação do sistema, entre outros, sem dúvida são desafios que surgem em projetos grandes e que aparecem em maior grau de dificuldade quando comparado a fazer um programa no IME, sem dúvida é muito enriquecedor à formação e complementa o conhecimento acadêmico que adquirimos no IME, além de presenciar como acontece a engenharia de software na prática.

Sinceramente não acho que tenha havido frustrações no decorrer do estágio, meu chefe sempre deu muita liberdade à equipe com relação ao desenvolvimento da arquitetura e estudo de tecnologias, além de sempre me liberar em semanas de prova ou mesmo para poder fazer trabalhos da faculdade.

6.2 O Relacionamento com os Membros da Equipe

O ambiente de trabalho da opus é muito bom, é descontraído, mas sério o suficiente para desenvolver um trabalho de qualidade. A equipe de desenvolvimento da opus era composta por min e mais três membros. Dois membros da equipe inclusive eram ex-alunos do bcc do IME, Thiago Lourençoni e Adriano Saturno, graças a eles consegui me integrar muito bem a empresa e ao projeto.

O que diferencia o trabalho em grupo feita no IME da realizada no estágio é a presença de terceiros. Quando fazemos um trabalho em equipe na faculdade, o objetivo é o aprendizado de algum assunto com a cobrança sendo feita por professores. Já em um ambiente de trabalho há um grau de responsabilidade maior, pois a cobrança é feita pelo cliente, dessa forma atrasos de entrega comprometem a equipe como um todo, todos são responsáveis, além de prejudicar os negócios da empresa, nesse contexto ser ágil é uma necessidade para não prejudicar o grupo e a empresa.

6.3 Disciplinas que foram relevantes para o trabalho

- **MAC 323 Estrutura de dados e MAC 122 Princípios de desenvolvimento de algoritmos:** disciplinas fundamentais a todo programador, é a base para se desenvolver algoritmos.
- **MAC 242 Laboratório de programação II:** foi nessa disciplina que tive meu primeiro contato com a linguagem Java e com a orientação a objetos.
- **MAC 426 Sistemas de Banco de Dados:** o projeto em que trabalhei exigiu a necessidade de se realizar a modelagem relacional do sistema a fim de persistir dados em banco.
- **MAC 332 Engenharia de software:** em qualquer projeto de sistemas conhecer processos de desenvolvimento de software é fundamental saber como projetar sistemas.
- **MAC 441 Programação orientada a objetos:** esta matéria é importante para se aprender como programar orientado a objetos.
- **MAC 440 Sistemas de objetos distribuídos:** o projeto opus ibank exigiu a necessidade de se lidar com serviços remotos, esta matéria foi importante para

meu trabalho, pois da uma base muito boa para se entender como trabalhar com sistemas distribuídos.

- **MAC 446 Princípios de Interação Homem Computador:** de todas as disciplinas esta com certeza merece destaque, pois mostra uma nova abordagem de desenvolvimento de software, a partir do modo como o usuário vai lidar com a interface do sistema. Méritos ao professor Hitoshi, pois a disciplina foi muito bem ministrada quando eu fiz.

6.4 Próximos passos

O sistema opus ibank esta em desenvolvimento ainda, mas com vários módulos já implementados, a meta é que a partir de maio de 2007 esteja concluída e seja instalado no banco ibi do grupo C&A.

Como próximos passos gostaria de realizar um estudo na área de componentização de software a fim de melhorar processos de desenvolvimento, a área de engenharia de software baseada em componentes é muito aplicada na pratica e bem vista com bons olhos pela empresa onde trabalho, com certeza esse será o caminho de estudo que seguirei após a graduação.

6.5 Agradecimentos

Agradeço ao IME pela ótima formação dada aos seus alunos; à empresa Opus pelo apoio financeiro e pela oportunidade de trabalhar como estagiário; ao prof. Siang Wun Song ter me orientado na escrita da monografia.

Finalmente agradeço à minha família e amigos que sem eles com certeza não teria conseguido seguir em frente nesse puxado curso que é o bcc do IME.

Bibliografia

[1] Bauer, C. & King, G. (2004): Hibernate in Action. Manning Publishing Co., 2004.

[2] Fayad, Mohamed C.; Schmidt, Douglas C.; Johnson, Ralph E. *Building Application Frameworks – Object Oriented foundations of framework design.* Wiley , 1999.

[3] Fowler, M. (2002): Patterns of Enterprise Application Architecture. Addison-Wesley, 2002

[4] Fowler, M. (2004): Inversion of Control Containers and the Dependenc Injection pattern: Artigo disponível em http://martinfowler.com/articles/injection.html.

[5] Frakes, W.; Terry, C. Software Reuse and Reusability Metrics and Models, 1996.

[6] Gamma, E., Helm, R., Johnson & R., Vlissides, J. (1995): Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, MA, 1995.

[7] Geary, D. & Horstmann, C. (2005): Core JavaServer Faces, Sun Microsystems, Inc, 2005.

[8] Gustavo Alonso, Fabio Casati, Harumi Kuno e Vijay Machiraju, Web Services Concepts, Architectures and Applications, Springer-Verlag, 2004.

[9] *Hibernate* (2006): Página oficial do *framework Hibernate* disponível em http://www.hibernate.org

[10] Johnson, Ralph. E. *Components, Frameworks, Patterns*. Communications of the ACM, V. 40, nº 10, p. 39-42, 1997.

[11] Monday, Paul. *Web Services Patterns: Java Edition*, Apress-2005

[12] MyFaces (2005): Página oficial do MyFaces disponível em http://myfaces.apache.org/.

[13] Pressman, Roger S. (2002) . *Engenharia de Software*. McGrawHill.

[14] Prieto-Diaz, R. Reuse as a New Paradigm for Software-Development. Proceedings of the International Workshop on Systematic Reuse, 1996

[15] Sauvê, Jacques Philipe. *Frameworks*. Universidade federal da Paraíba. Notas de aula http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm

[16] Walls, C.; Breidenbach, R. Spring em Ação. (2006) Editora Ciência Moderna.