

**UNIVERSIDADE FEDERAL DE SANTA MARIA**  
**CENTRO DE TECNOLOGIA**  
**ESPECIALIZAÇÃO EM SISTEMAS DE COMPUTAÇÃO PARA WEB**

**ESTUDO DE CASO USANDO O SPRING**  
**FRAMEWORK PARA A CRIAÇÃO DE APLICAÇÕES**  
**WEB COM J2EE**

**MONOGRAFIA DE ESPECIALIZAÇÃO**

**José Ronaldo N. Fonseca Júnior**

**Santa Maria, RS, Brasil**

**2007**

**ESTUDO DE CASO USANDO O *SPRING FRAMEWORK* PARA  
A CRIAÇÃO DE APLICAÇÕES *WEB* COM *J2EE***

**por**

**José Ronaldo N. Fonseca Júnior**

Monografia apresentada ao Curso de Especialização em Sistemas de  
Computação para Web, da Universidade Federal de Santa Maria (UFSM, RS),  
como requisito parcial para obtenção do grau de  
**Especialista em Sistemas de Computação para Web.**

**Orientador: Iara Augustin**

**Santa Maria, RS, Brasil**

**2007**

**Universidade Federal de Santa Maria**  
**Centro de Tecnologia**  
**Especialização em Sistemas de Computação para Web**

A Comissão Examinadora, abaixo assinada,  
aprova a Monografia de Especialização

**ESTUDO DE CASO USANDO O SPRING FRAMEWORKS PARA A  
CRIAÇÃO DE APLICAÇÕES COM J2SE**

elaborada por

**José Ronaldo Fonseca Junior**

como requisito parcial para obtenção do grau de  
**Especialista em Sistemas de Computação para Web**

**COMISSÃO EXAMINADORA:**

**Iara Augustin, Dra.**

(Presidente/Orientador)

**João Carlos Damasceno Lima, Msc. (UFSM)**

**Oni Reasilvia Sichonany, Msc. (UFSM)**

Santa Maria, 30 de janeiro de 2007.

## **ABSTRACT**

Monografia de Especialização  
Curso de Especialização em Sistemas de Computação para Web  
Universidade Federal de Santa Maria

### **“CASE STUDY USING THE SPRING FRAMEWORK TO CREATE WEB APPLICATIONS WITH J2EE”**

**AUTHOR: JOSÉ RONALDO NOGUEIRA FONSECA JÚNIOR**

**ADVISOR: IARA AUGUSTIN**

**Defense Date and Local: Santa Maria, January 30, 2007.**

This paper presents a case study about Framework Spring used in the construction of applications to distributed environments or enterprise. Through this study, we searched to stick out the simplification importance of the applications development process based in the Java 2 Enterprise Edition (J2EE) specification. Therefore was necessary to introduce some concepts, as Design Patterns, which the developers would use in the multi-layers architecture creation (n-tier) to solve specific problems found in corporative environments. Nowadays, there are uncountable frameworks solutions liable to utilization in web applications; however, the configuration process of these solutions to team-work is still too expensive. Hence, one of the greatest Framework Spring contributions is the transparent configuration of technological solutions chosen to use by application, releasing the developers to the exclusive focus in questions related with the business or end area of company or organization. The paper is separated in three main chapters respectively about: literature revision (Framework Spring, Definitions; the Enterprise development process; Aspects; Elements); case study (Appfuse and Equinox projects presentation as *framework Spring* effective utilization example); of contributions; and author considerations about approached topics.

**Key-words:** framework; web applications; distributed environments.

## RESUMO

Monografia de Especialização  
Curso de Especialização em Sistemas de Computação para Web  
Universidade Federal de Santa Maria

### ESTUDO DE CASO USANDO O SPRING FRAMEWORK PARA A CRIAÇÃO DE APLICAÇÕES WEB COM J2EE

AUTOR: JOSÉ RONALDO NOGUEIRA FONSECA JÚNIOR

ORIENTADOR: IARA AUGUSTIN

Data e Local da Defesa: Santa Maria, 30 de janeiro de 2007.

Este trabalho apresenta um estudo de caso acerca do *Framework Spring* usado na construção de aplicações para ambientes distribuídos ou *enterprise*. Por meio desse estudo, procurou-se ressaltar a importância da simplificação do processo de desenvolvimento de aplicações baseadas na especificação *Java 2 Enterprise Edition (J2EE)*. Para tanto foi necessário introduzir alguns conceitos, como os padrões de projeto (*Design Patterns*), que embasariam os desenvolvedores na criação de uma arquitetura multi-camadas (*n-tier*) para resolução de problemas específicos encontrados em ambientes corporativos. Atualmente, inúmeras são as soluções de *frameworks* passíveis de utilização em aplicações *web*, porém, o processo de configuração destas soluções para trabalho em conjunto ainda é oneroso demais. Logo, uma das grandes contribuições do *framework Spring* está no fato de configurar de forma transparente as soluções tecnológicas escolhidas para uso pela aplicação, liberando os desenvolvedores para um foco exclusivo em questões relacionadas ao negócio ou área fim da empresa ou organização. O trabalho está dividido em três capítulos principais tratando respectivamente de: revisão de literatura (Framework Spring, Definições; O Processo de Desenvolvimento Enterprise; Aspectos; Elementos); estudo de caso (Apresentação dos projetos *Appfuse* e *Equinox* como exemplo de utilização efetiva do *framework Spring*); das contribuições; e das considerações do autor sobre os assuntos abordados.

Palavras-chave: framework; aplicações web; ambientes distribuídos

## SUMÁRIO

<b>INTRODUÇÃO</b> .....	6
<b>1. O FRAMEWORK SPRING</b> .....	8
<b>1.1 O processo de desenvolvimento de aplicações enterprise</b> .....	9
<b>1.2 Aspectos do framework spring</b> .....	12
1.2.1 Elementos do framework spring .....	13
1.2.1.1 Gerenciamento de transações .....	18
1.2.1.2 Suporte a <i>data access object – dao</i> e <i>java database connectivity</i> .....	20
1.2.1.3 O framework de MVC para web do spring .....	23
1.2.1.4 Integração com outros frameworks .....	26
<b>2. ESTUDO DE CASO</b> .....	28
<b>2.1 Descrição da aplicação</b> .....	29
<b>2.2 Modelagem da aplicação</b> .....	30
<b>CONCLUSÕES E TRABALHOS FUTUROS</b> .....	37
<b>REFERÊNCIAS BIBLIOGRÁFICAS</b> .....	39

## INTRODUÇÃO

Quando o assunto é desenvolvimento de aplicações, a tendência predominante dentre a comunidade de desenvolvedores, no atual estágio de tecnologia, é o uso de *frameworks*, principalmente se as aplicações se destinam a ambientes distribuídos ou *enterprise*, como a *web*. Um *framework* se baseia nas principais características da programação orientada a objetos: abstração de dados, polimorfismo e herança. Estas características são responsáveis por tornar o *framework* possível de ser feito objetivando a reutilização, pois permitem a construção de classes genéricas (FAYAD, 1999).

Diferentemente de qualquer outra linguagem de programação, a linguagem Java tornou possível criar aplicativos complexos, compostos por partes discretas. Já em dezembro de 1996, a *Sun Microsystems* publicou as especificações para *JavaBeans* (CRAIG, 2006). Nessa especificação definiu um modelo de componente de software, com um conjunto de políticas de códigos que capacitam simples objetos java a serem reutilizados e compostos dentro de aplicativos mais complexos, facilitando sua utilização em *frameworks*. Embora o *JavaBeans* tenha sido planejado com a intenção de servir como meio de propósito geral, ele era simples demais para ser capaz de oferecer serviços mais complexos, fazendo com que desenvolvedores que necessitassem criar aplicações mais robustas, como sistemas distribuídos, buscassem algo mais eficiente no mercado.

Logo, a *Sun* publicou a especificação *Enterprise JavaBeans (EJB)*, tendo em vista que aplicativos sofisticados necessitavam oferecer serviços mais complexos como suporte a transações, segurança e computação distribuída (CRAIG, 2006). Essa especificação estendeu a noção de componentes Java para o lado do servidor, oferecendo importantes serviços para ambientes distribuídos. Embora tenha adicionado complexidade ao modelo de desenvolvimento proposto inicialmente, a especificação *EJB* ainda ocupa o papel principal na arquitetura *Java 2 Enterprise Edition (J2EE)*.

Mesmo que muitos aplicativos de sucesso tenham sido desenvolvidos usando essa especificação, o processo de desenvolvimento de aplicações, ainda que para sistemas simples, tornou-se muito complexo, fazendo com que os ganhos de tecnologia fossem deixados em segundo plano. Logo, qual seria a saída para os desenvolvedores que buscam construir aplicações *J2EE* mais simples sem as complexidades adicionadas pela especificação *EJB*? A alternativa que vem mudando os paradigmas de desenvolvimento de aplicações *web* é o uso

de *containers* leves, como o *framework Spring*, que se utilizam de avançadas técnicas de programação como a Programação Orientada a Aspectos (*AOP*) e a Inversão de Controle (*IoC*).

Baseado em um princípio de projeto chamado Inversão de Controle, o *Spring* é um *framework* eficiente, porém leve, que não exige o uso de *EJBs*, reduzindo significativamente a complexidade no uso de interfaces, agilizando e simplificando o desenvolvimento de aplicações. Logo, os usuários podem obter os recursos eficientes e robustos de *EJB*, ao mesmo tempo em que mantêm a simplicidade do *JavaBean*(CRAIG, 2006).

Assim, o presente estudo tem como objetivo principal realizar um estudo de caso acerca do *framework Spring*, usando suas principais características de *container* leve, com o intuito de simplificar o processo de desenvolvimento de aplicações *J2EE*. Para tanto, foi elaborado, também, um estudo sobre aplicações que se utilizam dos conceitos e facilidades do *Spring* para inicializar, de forma rápida e simplificada, projetos de desenvolvimento *J2EE*. Tendo em vista, que embora exista uma ampla gama de tecnologias disponíveis para uso em ambientes distribuídos, seu processo de configuração e montagem inicial de projetos ainda é muito complexo e oneroso à comunidade de desenvolvedores.

O restante do texto está estruturado da seguinte forma: o capítulo 2 aborda as principais características do *framework Spring*, tendo como foco inicial o processo de desenvolvimento de aplicações *J2EE*, buscando mostrar, de forma sucinta, a necessidade da utilização de técnicas e padrões de projeto muito bem documentados para obtenção de sucesso no desenvolvimento de novos projetos de software. Ainda nesse capítulo, os aspectos principais do projeto *Spring* são estudados, levando em consideração assuntos como a disposição em camadas dos módulos principais do *container*, bem como, o uso de técnicas de *AOP* e as funcionalidades oferecidas para as principais camadas em uma aplicação *J2EE*.

No capítulo 3, será mostrada uma aplicação de exemplo de uso do *framework Spring*, mas não apenas isso, esta aplicação monta para os desenvolvedores o projeto inicial do software a ser desenvolvido, configurando as tecnologias escolhidas para que trabalhem juntas, deixando o desenvolvedor livre para se preocupar apenas com os objetos de negócio da aplicação. Com esse projeto, fica fácil estruturar camadas em uma aplicação *J2EE*, pois, alguns padrões de projeto são utilizados para garantir o nível de abstração desejado para um projeto de sucesso para cada camada de uma aplicação distribuída.

## 1 O FRAMEWORK SPRING

*O Spring* é um *framework open-source*, criado por Rod Johnson e descrito em seu livro *Expert One-on-One: J2EE Design e Development*, onde foi originalmente chamado de “*interface21*”, nome batizado em homenagem a empresa responsável por investir e manter o projeto deste *framework*. Este *framework* utiliza conceitos antigos de programação orientada a objetos como a utilização de objetos *POJO*(*Plain Old Java Object*). Objetos java simples que não implementam interface alguma(RICHARDSON, 2006), mas que, utilizados sozinhos, tornam-se ineficientes, pois aplicações *enterprise* necessitam de suporte a serviços mais complexos como gerenciamento de transações, segurança e persistência. Logo, com o *Spring*, essas características se unificaram, possibilitando que as aplicações pudessem usufruir de serviços que só estavam disponíveis em ambientes *EJB*.

A evolução das ferramentas de apoio contribuiu com o processo de desenvolvimento de *software*, mas, por outro lado, o esforço dispendido manteve-se elevado. Com o *framework Spring* temos a possibilidade de testar uma solução de “peso leve” para a construção de aplicações *enterprise* prontas para uso, ao mesmo tempo em que o *framework* se encarrega de prover operações avançadas como gerência declarativa de transações, acesso remoto a objetos de negócio com invocação remota de métodos (*Remote Method Invocation – RMI*) ou *web services*.

Embora as características dos chamados *frameworks* leves se assemelhem é de suma importância que o projeto de *software* esteja aderente a soluções padronizadas para evitar reinvenção de idéias ou retrabalho por parte dos profissionais encarregados de desenvolver o *software*, usando como fonte padrões de projeto alicerçados no mercado de tecnologia *enterprise*.

Padrões de projeto constituem o cerne do projeto do *framework Spring*, buscando resolver problemas com soluções existentes no mercado, facilitando e otimizando a criação de componentes com o objetivo de servir como base para futuras reutilizações bem como inserção em projetos que necessitem de serviços já projetados.

## 1.1 O Processo de Desenvolvimento de Aplicações Enterprise

Nos últimos anos o aumento da procura por soluções de desenvolvimento em aplicações distribuídas, focando em agilidade no processo de desenvolvimento, segurança e voltado para atuação em ambientes heterogêneos, impulsionaram mudanças no modelo de desenvolvimento de *software* existente, forçando os profissionais da área de tecnologia a buscarem novas soluções no mercado para sanar estas demandas.

À medida que as necessidades dos clientes evoluíam para sistemas com um maior grau de complexidade, centrado em processos de desenvolvimento para sistemas corporativos, plataformas que se dispunham a focar seus sistemas em ambientes distribuídos ou *enterprise* ocuparam lugar de respeito no âmbito de tecnologia de informação e no centro dessa mudança está a *Java 2 Platform, Enterprise Edition (J2EE)*, responsável por especificar os principais serviços que uma aplicação *enterprise* deve oferecer.

Desde o nascimento da linguagem Java seu grau de adoção foi intenso na comunidade de desenvolvedores de *software*. À medida que a linguagem foi evoluindo, agrupando novas funcionalidades em sua *API (Application Programming Interface)*, outras tecnologias passaram a fazer parte da plataforma. Logo, passou a existir a necessidade de unificar os padrões existentes na época para desenvolvimento distribuído, bem como, padronizar os serviços que fariam parte da *API* a ser criada especificamente para a plataforma *J2EE*.

Para tanto, a *Sun* e um grupo das principais empresas colaboradoras do projeto, para essa incipiente plataforma, decidiram delegar ao *Java Community Process – JCP*, definir e unificar os padrões e *APIs* disponíveis para a especificação *J2EE* (DEEPAK, 2002). Algumas das vantagens propostas por esta especificação:

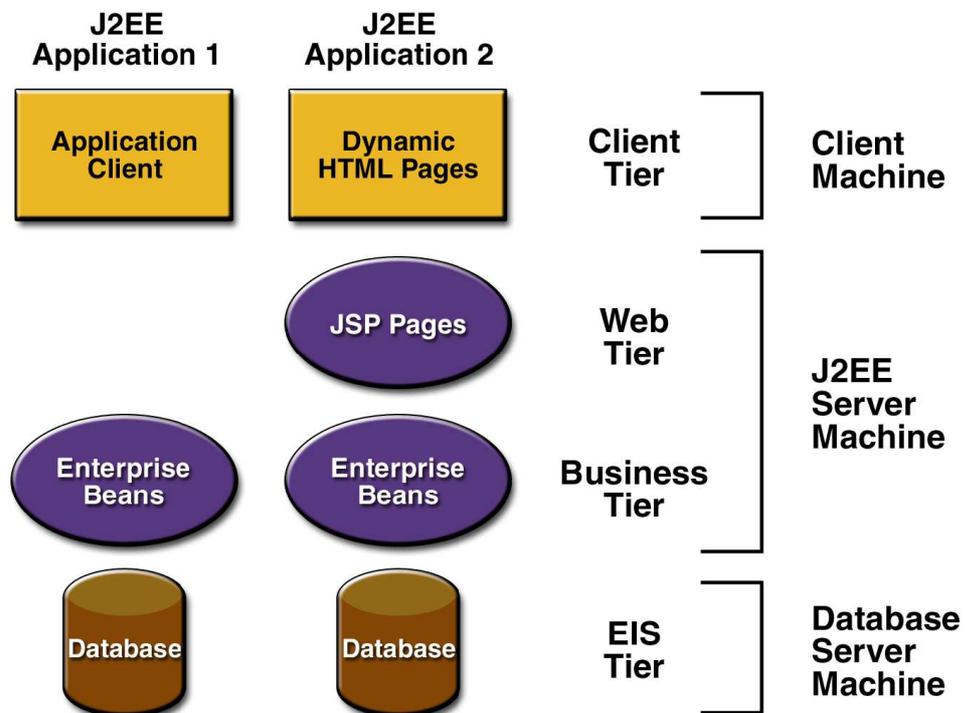
- Estabelecer padrões de computação distribuída para conectividade em banco de dados, componentes de negócios distribuídos, *middleware* orientado a mensagens (*Message Oriented Middleware - MOM*), componentes *web*, protocolos de comunicação e interoperabilidade.
- Promover melhorias no processo de criação de *software* baseado em padrões *open-source*, protegendo o investimento em tecnologia.
- Prover uma plataforma padrão para construção de componentes de *software* portáteis.

- Acelerar o processo de desenvolvimento, pois a infraestrutura é de responsabilidade dos fornecedores das aplicações, que necessitam respeitar os padrões definidos pela especificação *J2EE*. Logo, as empresas de tecnologia não se preocupam mais com problemas referentes ao *middleware*, mas, única e exclusivamente, com a construção de aplicações para solucionarem problemas relacionados ao seu próprio negócio.
- Aumentar a produtividade em programação, pois o programador que já possui familiaridade com a linguagem Java se utiliza destes conceitos. Pois, toda e qualquer aplicação distribuída necessita utilizar como base a linguagem da *API* padrão.
- Promover interoperabilidade entre ambientes heterogêneos.

A criação de arquiteturas corporativas poderosas para aplicativos *J2EE* requer um projeto que permita um rápido crescimento, voltado para aplicações multicamadas, baseadas em componentes que serão executados em um servidor de aplicações. A plataforma atualmente é considerada como padrão de desenvolvimento, tendo em vista que o fornecedor de *software* nesta plataforma deve seguir determinadas regras por motivos de compatibilidade com a especificação *J2EE*.

Ao mesmo tempo em que a plataforma *J2EE* ganhou notoriedade, aprendê-la tornou-se um processo oneroso e confuso para os desenvolvedores de *software*. Embora, exista vasta literatura sobre a especificação, explicando seus aspectos, aplicar seus conceitos tornou-se um processo complicado, forçando os desenvolvedores a buscar um entendimento que estava além do funcionamento das principais *APIs* da arquitetura.

Com o intuito de contribuir com a simplificação do modelo de criação de *software* usado pela especificação *J2EE* a *Sun* tratou de disponibilizar um documento que viesse a auxiliar os profissionais na definição de uma arquitetura de software, focando em projetos de arquitetura de n-camadas (*n-tier*) (Figura 1), documento conhecido como *J2EE Blueprints*.



**Figura 1 – Camadas de um Aplicação definida pelo *J2EE BluePrints* da *Sun*.**

Usando como referência os *Blueprints* criou-se um catálogo com os principais padrões de projeto para a plataforma *J2EE*. Padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema genérico de *design* em um contexto específico (GAMMA, 1995). Os padrões representam soluções de projeto consideradas melhores práticas para implementar vários componentes essenciais em cada uma das camadas identificadas pelos *Blueprints* da *Sun*.

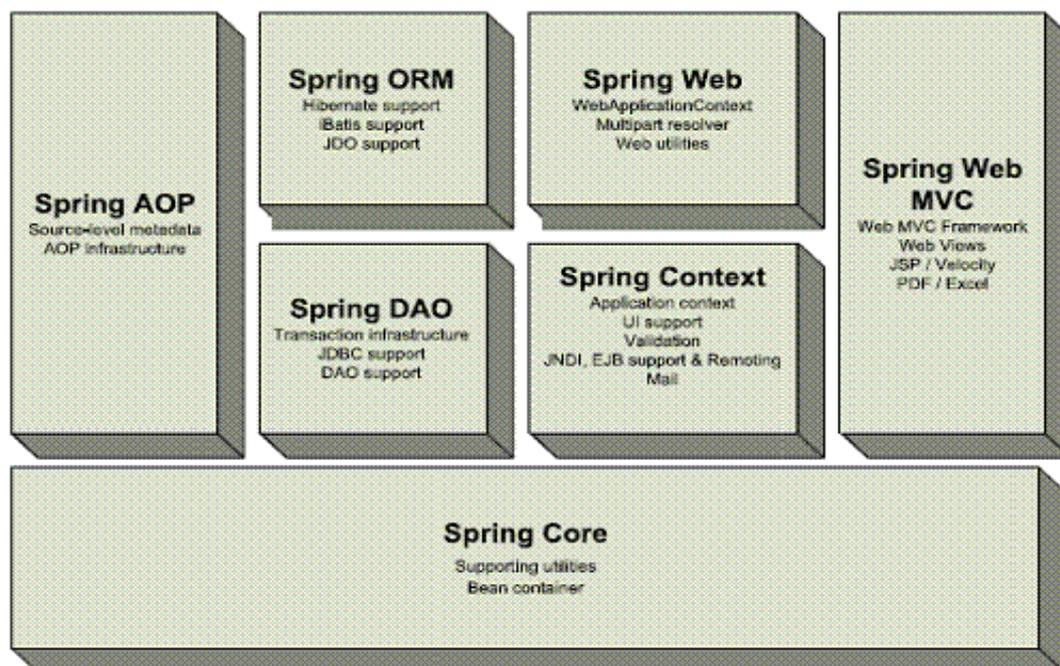
O catálogo de padrões *J2EE* atualmente define 21 padrões, diferentemente da primeira versão deste catálogo datada do ano de 2001 onde foram definidos apenas 15 padrões, focados em macro atividades de projeto e implementação. Baseado em uma arquitetura multicamadas os padrões estão distribuídos em camadas de apresentação, negócio e integração, conforme disposto no documento *J2EE Blueprints*.

Usando este enfoque para construção de sistemas multicamadas o *framework Spring* se encarrega de usar alguns dos padrões definidos pelo catálogo da *Sun* e oferece serviços para cada uma destas camadas. Destacando a camada de negócios onde o *framework* realmente dá um show, pois, define um modelo de programação menos burocrático e

complexo que a especificação de *EJBs*, usando conceitos básicos de objetos *javabeans* e *POJOs*.

## 1.2 Aspectos do Framework Spring

Para possibilitar o ganho de produtividade no desenvolvimento de aplicações *enterprise* o *framework spring* foi projetado de forma modular. Ao todo, o *framework* é composto por sete módulos (Figura 2). Embora os módulos venham a oferecer todos os serviços necessários, o *framework*, por não ser intrusivo, possibilita ao desenvolvedor que opte pelos módulos necessários para resolução de um problema específico, ignorando os demais.



**Figura 2 – Módulos do *Framework Spring*.**

O módulo *Spring Core* representa as principais funcionalidades do *framework*. Nesse módulo encontra-se o *BeanFactory* do *Spring*, que é o coração de qualquer aplicação baseada nesse *container*. O *BeanFactory* trata-se de uma implementação do padrão *factory*, no qual utiliza *IoC* para separar a configuração do aplicativo e as especificações de dependência do código referente a lógica de negócios da aplicação. Uma das principais características que tornam o *spring* um *framework* fica por conta do módulo de contexto da aplicação, pois, este

módulo estende os conceitos do *BeanFactory* adicionando serviços importantes para ambientes *J2EE* como e-mail, acesso a *JNDI*, integração com *EJB*, *remoting*, suporte a internacionalização de mensagens, bem como integração com outros *frameworks*.

O suporte a programação orientada a aspecto fica por conta do módulo *Spring AOP* que por motivos de compatibilidade obedece a padrões definidos pela *AOP Alliance*, projeto responsável por promover a adoção da *AOP* e interoperabilidade entre diferentes implementações de *AOP* definindo um conjunto comum de interfaces e componentes.

Como a razão de existência desse *container* é facilitar o processo de desenvolvimento de aplicações *enterprise* ou *J2EE*, o *Spring* oferece os módulos de *JDBC* e *DAO(Data access Object)* para abstração de *JDBC*, poupando o desenvolvedor da codificação maçante de abertura e fechamento de conexões, controle de exceções de banco de dados, adicionando também, serviços de gerenciamento de transações para objetos via *AOP*.

O módulo *O/R Mapping* é responsável por integrar o *framework* com as ferramentas mais poderosas de mapeamento objeto/relacional; o módulo *Web Context and Utility* baseia-se no módulo de contexto da aplicação, porém o contexto criado por esse módulo é apropriado para uso em aplicações *web* com suporte a integração com outros *frameworks MVC*, como por exemplo, o *Jakarta Struts*.

Se o desenvolvedor já está familiarizado com os *frameworks* de *MVC* disponíveis no mercado e busca usar uma solução menos intrusiva, onde o desenvolvedor não é obrigado a usar determinados componentes da tecnologia para que tenha êxito na sua aplicação. O *framework spring* disponibiliza o seu próprio *framework MVC* para a construção de aplicativos *web*, usando *IoC* para prover a separação necessária entre código referente a lógica de controle e lógica de objetos de negócios.

### 1.2.1 Elementos do Framework Spring

Dentre as tecnologias integrantes do *framework Spring* destacam-se o *container* de inversão de controle(*IoC*) e o *framework* de *AOP* incluso no próprio *container*. A inversão de controle está no coração do *framework Spring*, embora a técnica ainda pareça complexa, a maneira como ela é utilizada facilita sua compreensão.

A inversão de controle, também conhecida como injeção de dependência(*Dependency Injection*), quando aplicada, introduz dependências entre objetos de uma aplicação, através de

alguma entidade externa, normalmente um arquivo de configuração escrito em *XML(Extensible Markup Language)* responsável por coordenar todos os objetos do sistema, diminuindo assim, não só o grau de acoplamento entre os objetos da aplicação, mas, a quantidade de código responsável por inserir dependências entre esses objetos.

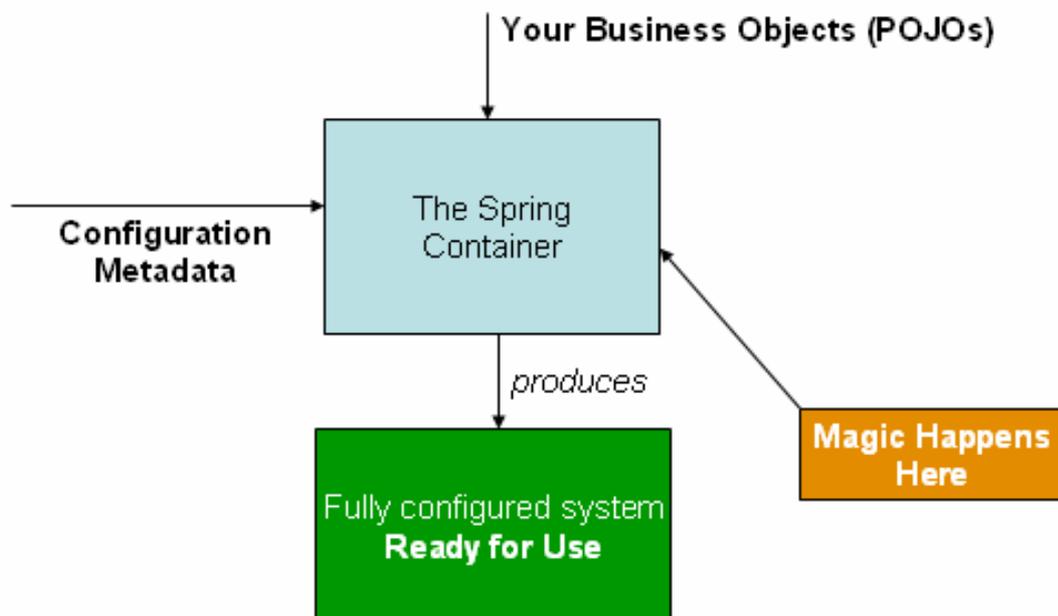
No *Spring* todos objetos que constituem a espinha dorsal de uma aplicação, gerenciados pelo *container* de *IoC* do *framework*, são referenciados como *beans*. Um *bean* é simplesmente um objeto que tipicamente pode ser instanciado, linkado com algum outro objeto, ou qualquer objeto que seja gerido de alguma forma pelo *container* do *Spring*. Estes *beans* e suas dependências estão declaradas em arquivos de configuração usados pelo próprio *container*.

O *container* de *IoC* do *Spring* é representado pela interface *BeanFactory* (`org.springframework.beans.factory.BeanFactory`), a qual é responsável por agrupar e gerenciar todos os *beans* de uma aplicação. Como o próprio nome diz, essa interface representa uma fábrica de *beans*, uma implementação do padrão de projeto *factory*, responsável pela criação e destruição de beans.

Há mais para um *BeanFactory* do que simplesmente instanciar objetos de uma aplicação. Um *BeanFactory* conhece todos objetos que estão rodando dentro de uma determinada aplicação, isso possibilita que crie associações entre objetos no momento que são instanciados. Isto remove o fardo de configuração do próprio *bean* e do *bean* cliente. Como resultado, quando um *BeanFactory* entrega objetos, esses objetos estão totalmente configurados, conhecem seus objetos colaboradores; em outras palavras, estão prontos para uso. Além disso, um *BeanFactory* também participa do ciclo de vida de um *bean*, fazendo chamadas para métodos de inicialização e destruição, se esses métodos estiverem definidos(CRAIG, 2006).

Existem inúmeras implementações para a *interface BeanFactory*, prontas para uso, fornecidas pelo *framework Spring*. A implementação mais usada pela comunidade de desenvolvedores, deste *framework*, é dada pela classe *XmlBeanFactory*. Essa implementação possibilita ao desenvolvedor expressar via um arquivo de metadados, como *XML*, quais objetos irão compor a aplicação, bem como quais os relacionamentos de interdependência entre eles. Logo, através desses arquivos, toda a aplicação, como o sistema em que residem, são configurados declarativamente com o uso de arquivos *XML*.

O *container* de *IoC* do *Spring* consome qualquer forma de configuração de metadados(Figura 3). Tal configuração informa ao *container* como os objetos ou *beans* serão instanciados e configurados dentro do sistema.



**Figura 3 – O Container de Inversão de Controle (IoC) do Framework Spring.**

Em um nível mais elementar, a configuração do *container* de *IoC* do *Spring*, consiste em definir quais *beans* deverão ser gerenciados pelo *container*. Tipicamente, os *beans* disponíveis para uma aplicação podem ser objetos da camada de negócios, objetos para acesso a dados (*DAOs*), objetos da camada de apresentação como instâncias de *Actions* do *framework Struts*(objetos que implementam o padrão de projeto *command*), objetos de infraestrutura como *SessionFactory*s do *Hibernate*(responsável por disponibilizar sessões para uso em banco de dados) e *Java Message Service(JMS)*, etc.

Uma vantagem no uso de *BeanFactory* está na capacidade do *container* carregar o objeto de forma “preguiçosa” ou *lazy*. Ou seja, enquanto o *BeanFactory* carrega as definições do *bean*(descrição do objeto e propriedades), os *beans* não serão instanciados até o momento em que eles sejam necessários. Quando um *bean* torna-se necessário, ele é instanciado pelo *container* e suas propriedades são setadas usando injeção por dependência.

Algumas especializações dessa interface são responsáveis por oferecer serviços mais complexos ao *framework*, como a interface *ApplicationContext* (`org.springframework.context.ApplicationContext`), como capacidades de prover meios para resolver mensagens de texto, incluindo suporte para internacionalização (I18N) das mensagens; prover um modo genérico para carregar recursos de arquivos de imagens, entre outros. Logo, quando se deseja funcionalidades adicionais é preferível usar a interface *ApplicationContext*, por outro lado, se os recursos disponíveis para a aplicação são escassos, como um dispositivo móvel, a solução é continuar utilizando os recursos disponíveis pela interface *BeanFactory*.

Além da funcionalidade adicional oferecida pelos *ApplicationContexts*, outra grande diferença em relação ao *BeanFactory* é como os *beans* carregam os objetos de uma aplicação. Enquanto um *BeanFactory* carrega todos os *beans* de forma “*lazy*”, um *ApplicationContext* é mais inteligente, pois, carrega previamente todos os *beans singleton* (padrão de projeto utilizado quando deseja-se ter apenas uma e somente uma instância de um determinado objeto na aplicação) ao iniciar o contexto da aplicação. Isso implica em ganho de desempenho, tendo em vista que uma única instância de objeto é carregado na memória, não sendo mais necessário esperar pela criação de todos os objetos que venham a ser requisitados pelo sistema.

Teoricamente, os *beans* podem ser configurados virtualmente por qualquer fonte de configuração, incluindo arquivos de propriedades, banco de dados relacional ou até mesmo um diretório *LDAP* (*Lightweight Directory Access Protocol*). Mas na prática *XML* é a fonte de configuração de escolha para a maioria das aplicações que utilizam o *framework Spring*.

Outra grande vantagem do uso de arquivos *XML* de configuração pelo *container* do *Spring* é a possibilidade de definir o escopo de um objeto da aplicação. É possível controlar não apenas as dependências e valores de configuração plugáveis a um *bean*, como a forma de criação, por exemplo, mas, principalmente, o escopo dos objetos criados por essas definições. Dentre os tipos de escopo definidos pelo *framework Spring* para a criação de objetos, destacam-se os tipos *singleton* e *prototype*.

Por padrão, todos os *beans* do *Spring* são *singletons*. Quando um *bean* é configurado como *singleton*, apenas uma instância desse *bean* é compartilhada pela aplicação (Figura 4). Essa única instância passará a ser armazenada pelo *container* em um *cache* para objetos *singleton* e qualquer requisição ou referência subsequente àquele objeto resultará na instância armazenada previamente no *cache* pelo *container*.

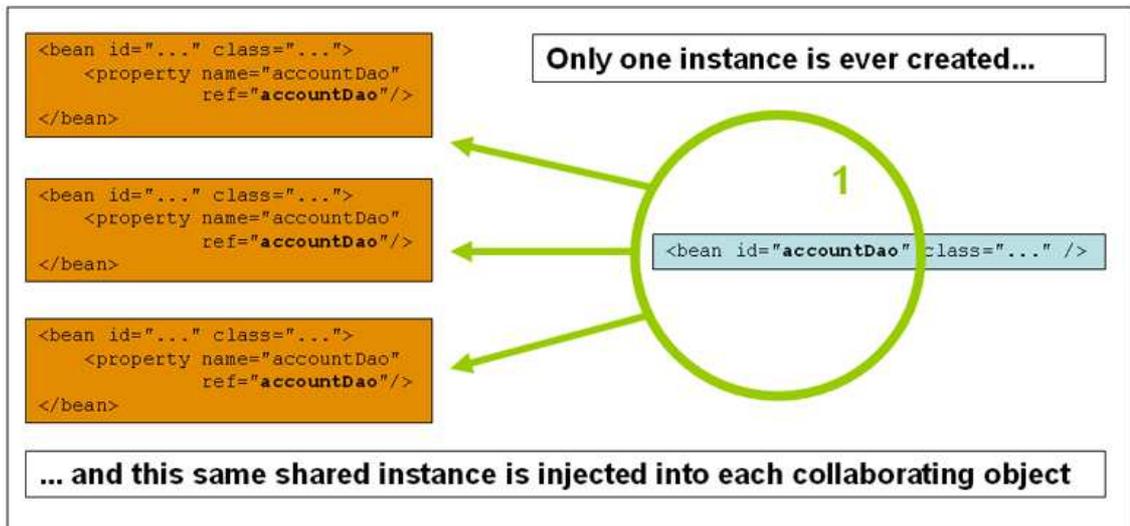


Figura 4 – Processo de criação de um *bean* de escopo *singleton*.

Já o escopo para criação de *beans* do tipo protótipo ou *prototype* fará com que o *container* retorne uma nova instância do *bean* cada vez que este seja solicitado pela aplicação (Figura5).

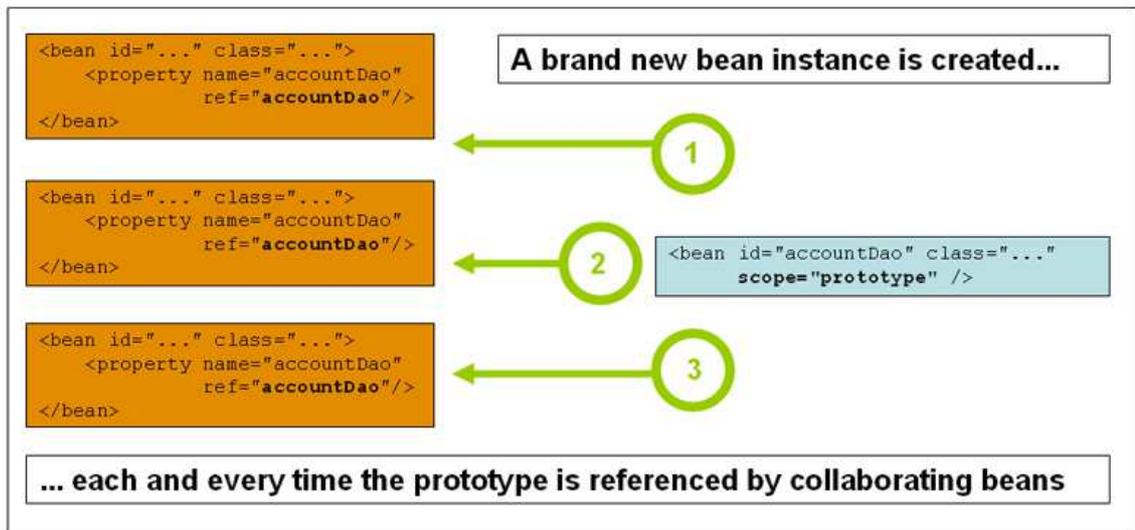


Figura 5 – Processo de criação de um *bean* de escopo protótipo ou *prototype*.

Uma peça de fundamental importância para o *framework Spring*, junto com o *container* de *IoC*, é o seu próprio *framework* de *AOP*. Embora o *container* de *IoC* não dependa do *framework* de *AOP*, é quando trabalham juntos que se obtém a potencialidade máxima deste *framework* como solução de *middleware*.

Os conceitos de *AOP* não dizem respeito a um novo paradigma de programação, mas sim, complementam ou estendem conceitos que embasam o processo de programação orientada a objetos. Ao invés de classes, com *AOP* trabalhamos com aspectos modularizando conceitos como gerenciamento de transações, onde o desenvolvedor passa a focar em uma estrutura de programação totalmente fora dos padrões atuais da orientação a objetos.

A orientação a aspectos permite que as classes desenvolvidas em uma aplicação contenham apenas o código necessário para desempenhar alguma regra de negócio, sem a necessidade de inclusão de blocos auxiliares para outros fins, como auditoria, segurança, *log*, controle de transações, *threads* e tratamento de erros. Com a separação destes aspectos do código básico das classes, é possível obter as seguintes vantagens (RONCONI, 2006):

- Códigos mais simples, uma vez que as classes possuem apenas regras de negócio deixando de lado aspectos de projeto;
- Alterações nos aspectos de projeto não geram impacto nas classes. As classes passam a possuir apenas a regra de negócio;
- Maior produtividade na manutenção, uma vez que um mesmo aspecto pode ser aplicado a diferentes classes de um projeto.

Com a forma de utilização dos conceitos de programação orientada a aspectos do *framework Spring* é possível prover serviços distribuídos de forma declarativa, via *XML*, principalmente como forma de substituição a serviços complexos definidos declarativamente para *EJBs*. Sendo que o serviço de maior importância é prover um gerenciamento de transações de forma declarativa usando um alto grau de abstração definido pelo modelo de programação do *framework Spring*.

#### 1.2.1.1 Gerenciamento de Transações

Uma das principais razões ao optar pelo *framework Spring* está no suporte massivo ao gerenciamento de transações. Dentre os benefícios oferecidos destacam-se:

- Prover um modelo de programação consistente entre diferentes *APIs* como *JTA*, *JDBC*, *Hibernate*, *JPA* e *JDO*;
- Suporte ao gerenciamento declarativo de transações;
- Prove uma *API* de fácil manuseio para o gerenciamento programático de transações;
- Integração via *Spring* com diversos tipos de abstração de acesso a dados.

O *Spring* assim como o *EJB*, oferece um suporte tanto para o gerenciamento de transações programadas quanto declarativas, embora, os modelos de programação aplicados por essas tecnologias venham a diferir consideravelmente. Enquanto o *EJB* utiliza como suporte para o gerenciamento de transações programadas uma implementação da *Java Transaction API (JTA)*, o *Spring* utiliza um mecanismo de *callback*, que abstrai a implementação atual de transação do código transacional. Proporcionando em alguns momentos, quando a aplicação não necessita usar mais que um recurso persistente, usar o suporte transacional oferecido pelo mecanismo de persistência como *Hibernate*, *JDBC*, *Java Data Objects (JDO)*, etc.

As transações programáticas são utilizadas quando a aplicação requer um controle rigoroso sobre os limites das transações, tornando-se na maioria dos casos intrusivas, visto que, é necessário mudar, em alguns casos, a implementação de um objeto de negócio com o intuito de satisfazer alguma condição transacional para a aplicação. Como, na maioria dos casos, as necessidades transacionais de uma aplicação não irão requerer um controle tão preciso sobre os seus limites, a melhor escolha pode ser declarar as transações fora do código da aplicação, usando para isto o modelo declarativo de gerenciamento de transações.

O gerenciamento de transações declarativas suportada pelo *Spring*, antigamente passível de uso somente pela tecnologia de *EJBs*, é implementado através de seu *framework* de *AOP*. Esta é uma característica natural, pois as transações são serviços em nível de sistema, acima da funcionalidade primária de uma aplicação (CRAIG, 2006).

Para empregar transações declarativas em aplicativos *enterprise* o *framework Spring* utiliza conceitos da *AOP* para interceptar transações, através de um objeto com funções de *Proxy* (Figura 6) que tem o controle das transações da aplicação, bem como, os métodos que deverão sofrer controle transacional.

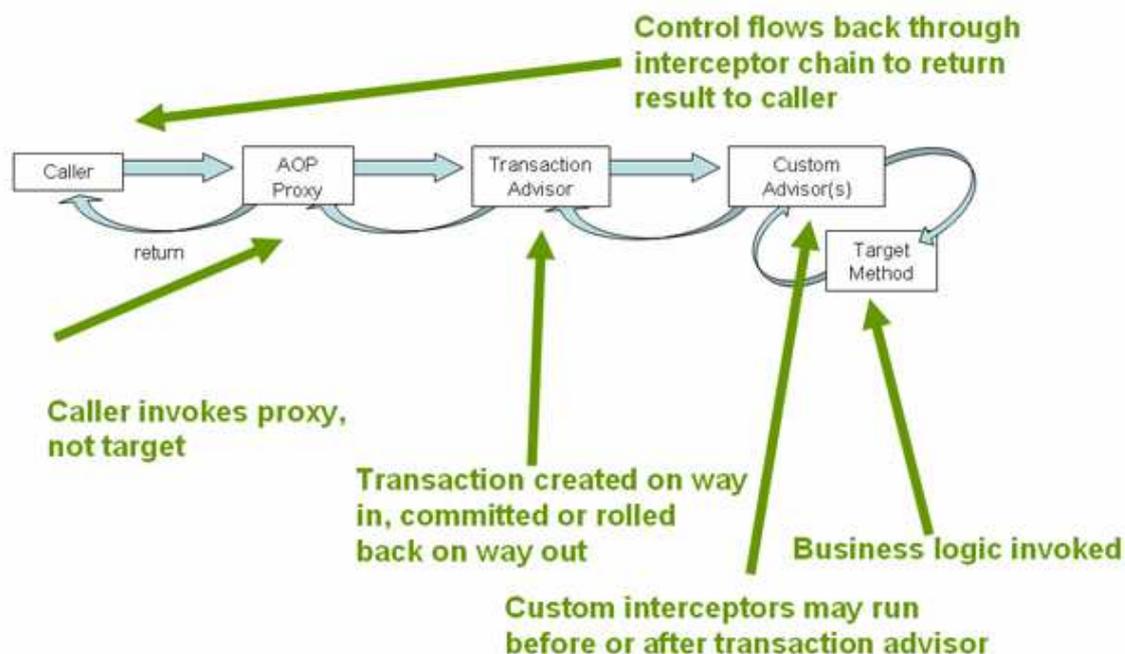
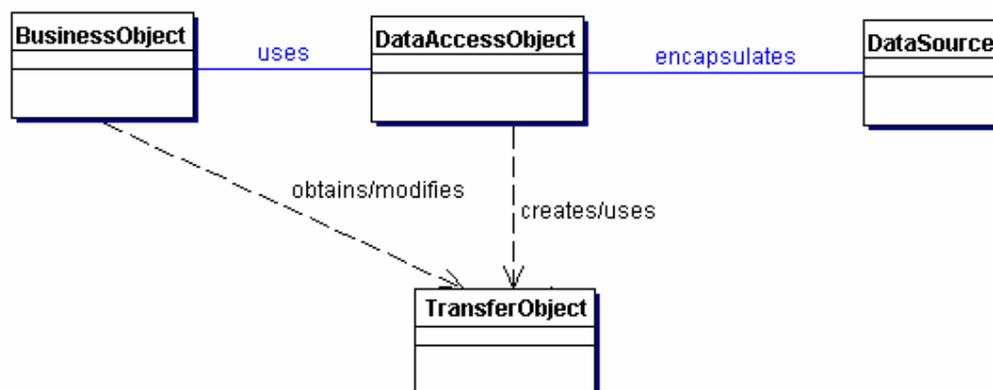


Figura 6 – *Proxy* para transações declarativas do *Spring* (GRAIG, 2006)

Não importa se a forma de programar transações em *beans* ou declará-los como aspectos é a melhor opção, nas duas soluções, a responsabilidade de interagir com implementações de transações específicas de uma plataforma é obrigatoriamente do gerenciador de transações do *Spring*, ficando o desenvolvedor livre pra codificar seus *beans* sem a preocupação com codificação de controle transacional.

#### 1.2.1.2 Suporte a Data Access Object – DAO e Java Database Connectivity

O padrão de *DAO* é responsável por representar Objetos de Acesso a Dados, que descrevem perfeitamente o papel de uma camada de abstração de banco de dados em uma aplicação (figura 7). *DAOs* existem para prover meios de ler e escrever dados no banco de dados. Eles devem expor esta funcionalidade por uma interface, pela qual o resto da aplicação terá acesso (CRAIG, 2006).



**Figura 7 – Data Access Object**

O suporte a *DAO* fornecido pelo *Spring* foi desenvolvido com o objetivo de facilitar a integração com tecnologias de acesso a dados como *hibernate*, *JDBC* e *JDO* de maneira padronizada. Dessa maneira, os usuários devem escolher as melhores ferramentas de acesso a dados baseado em suas vantagens tecnológicas, sem a preocupação com a maçante tarefa de captura de exceções para cada tecnologia a ser usada pela aplicação.

O *framework Spring* traduz exceções de diferentes tecnologias de acesso a dados para sua própria hierarquia de exceções, encapsulando a exceção original. Isso garante ao *Spring* a não necessidade de captura de exceções específicas de uma tecnologia, o que ocasionaria em codificação adicional em outras camadas da aplicação, que por sua vez, necessitariam capturá-las.

Para facilitar o trabalho com diferentes tipos de tecnologia de acesso a dados o *framework* provê um conjunto de classes abstratas para acesso a dados que podem ser utilizadas pela aplicação. Essas classes possuem métodos capazes de obter conexões, bem como gerencia-las independente da tecnologia utilizada.

Como ferramentas de apoio ao padrão *DAO* a *API* de *Java Database Connectivity (JDBC)* se estabeleceu, ao longo do tempo, como a principal e mais poderosa *API* para trabalho com banco de dados relacional em nível de *SQL (Structured Query Language)*. Logo, o *framework JDBC* do *Spring* facilita o uso de *JDBC*, assumindo responsabilidades como gerenciamento de recursos de banco de dados e controle de exceções, deixando o desenvolvedor livre para planejar quais consultas e expressões *SQL* serão necessárias para o sucesso de sua aplicação.

O centro do *framework* de *JDBC* do *Spring* é a classe de *template* para *JDBC* (`org.springframework.jdbc.core.JdbcTemplate`). Ela simplifica o uso de *JDBC* na medida em que manipula a criação e liberação de recursos de banco de dados, ajudando a prevenir a ocorrência de erros banais como o esquecimento do fechamento de conexões. Além destas funcionalidades básicas, é de sua responsabilidade, também, executar consultas, atualizar expressões, fazer chamadas a procedimentos armazenados, capturando, sempre que necessário, as devidas exceções.

Portanto, não importa qual a tecnologia de persistência utilizada, o *Spring* torna esta escolha transparente para o resto da aplicação, fornecendo uma hierarquia consistente de exceções, através dos objetos *DAO*. Interpretando as exceções específicas de cada tecnologia e códigos de erros específicos de cada vendedor o *Spring* proporciona uma aplicação livre quanto à ocorrência de exceções não capturadas entre as camadas da aplicação.

Além do *JDBC* puro, as aplicações podem optar pelo uso de ferramentas de mapeamento objeto-relacional (*Object-Relational Mapping – ORM*) para controlar necessidades de persistência mais complexas. *Spring* provê integração com *Hibernate*, *JDO*, *Oracle TopLink*, *iBATIS SQL Maps* e *JPA* tanto para gerenciamento de recursos, suporte a implementação de *DAOs* como estratégias para gerenciamento de transações.

Há duas formas de obter integração com ferramentas de *ORM*: *templates DAO* ou codificação manual de *DAOs* através das *APIs* de *Hibernate*, *JDO*, *TopLink*, etc. Em ambos os casos, os objetos de acesso a dados, podem ser configurados através de injeção de dependências, o que os possibilita usufruir de alguns serviços fornecidos pelo *framework* como gerenciamento de transações, controle de recursos da aplicação, entre outros. Dentre os benefícios obtidos com o uso do *framework Spring* na criação de *DAOs ORM* destacam-se:

- Facilidades de teste. Através do uso de *IoC* torna possível a troca de implementações e configurações locais entre instâncias de objetos de sessão do *hibernate*, instâncias de fonte de dados *JDBC*, gerenciadores de transação, etc. Isso torna muito mais fácil o processo de teste de forma isolada de cada peça a ser persistida em uma aplicação.
- Provê exceções de acesso a dados padronizadas. O *framework Spring* se encarrega de empacotar as exceções geradas pela ferramenta de *ORM* escolhida para aplicação, convertendo exceções proprietárias em exceções comuns de tempo de execução. Graças a essa facilidade é possível manipular um grande

número de exceções de persistência, nas camadas apropriadas, sem a necessidade de codificação do tipo *throws/catches* ou declaração de exceções.

- Gerenciamento de recursos genérico. O contexto da aplicação fornecido pelo *Spring* pode manipular a localização e a configuração de instâncias de objetos de sessão para *hibernate*, instâncias de fonte de dados *JDBC*, oferecendo uma maneira eficiente, fácil e segura de manipular recursos de persistência.
- Gerenciamento de transações integrado. O *framework Spring* possibilita encapsular o código responsável pelo mapeamento objeto-relacional de forma declarativa, através de interceptadores de métodos via *AOP*, ou empacotadores *template* codificados de forma explícita em Java.

### 1.2.1.3 O Framework de MVC para Web do Spring

Um dos problemas ainda encontrados em aplicações de *software* para *web* diz respeito à predileção, por parte de desenvolvedores, na mistura de codificação entre as camadas de integração, apresentação e negócio. Nestas aplicações, as interdependências entre os componentes causam uma onda de efeitos negativos, devido ao alto grau de acoplamento entre as camadas, dificultando o reuso. Logo, a adição de novas visões da aplicação ou *views* (pontos de acesso com o usuário) requerem reimplementação ou redundância de lógica de negócio, ocasionando elevados níveis de manutenção no sistema.

Como solução para este problema o padrão *Model-View-Controller* busca desacoplar os componentes responsáveis por acesso a dados, lógica de negócios, apresentação de dados (*views*) e interação com usuário. A principal vantagem da utilização deste padrão está em dividir a camada *web* para utilização por três tipos de objetos: controladores (*controllers*), modelos (*models*) e visões (*views*). Os objetos controladores são responsáveis por aceitar entrada de dados via usuário, e invocar lógica de negócio responsável por criar ou atualizar objetos *model*, que representam contratos entre objetos controladores e *views*. Já objetos do tipo *view* são responsáveis por apresentar ao usuário objetos de dados do tipo *model* conforme comando do controlador.

Em aplicações clássicas de uso de *frameworks MVC* para *desktop*, como o *Swing*, objetos *model* são responsáveis por promover atualizações em *views* registradas. Já em aplicações *web*, o padrão *MVC* sofre algumas pequenas variações, devido ao fato de as aplicações *web* estarem restritas ao ciclo requisição/resposta o que inviabiliza atualizações

automáticas em *views*. Em aplicações *web* o controlador expõe a *model* para a *view*, ou seja, ele prepara o objeto *model* para receber a requisição corrente e depois disponibiliza os dados necessários para renderização pela *view*.

Outro fator importante na aplicação do *pattern MVC* é a definição de uma camada *web* “magra” (*thin web tier*), separando a lógica de controle *web*, da lógica de negócio da aplicação. Essa falta de separação entre responsabilidades de cada camada específica da aplicação pode tornar o projeto de *software* oneroso ao extremo.

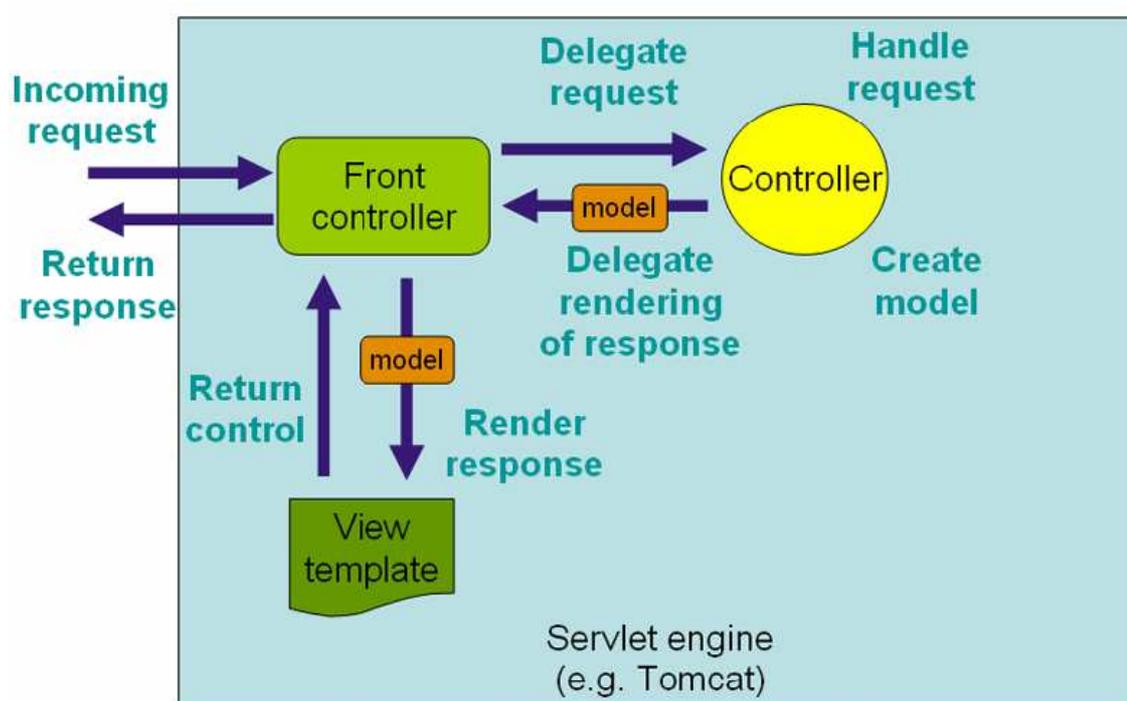
Aplicações *web* são melhores projetadas para utilizarem camadas de negócios que possam ser reutilizadas, mas, na maioria das aplicações isso não é realidade, pois, o controlador *web* não delega de forma apropriada objetos de negócio passíveis de reutilização, mas, ao invés disso, implementam tanto o fluxo de controle quanto lógica de negócio. Enquanto, esse comportamento, pode ser adequado para pequenas aplicações, para aplicações *enterprise*, tipicamente, ocasionará em duplicação de código de controle, dificuldade de teste e reutilização de componentes de negócio.

Decididos a simplificar o uso de *frameworks MVC* para *web*, os criadores do *framework Spring*, desenvolveram uma implementação própria do padrão *MVC*, simplificando o modelo de programação, pois, não é intrusivo, como soluções de mercado baseadas no *framework Jakarta Struts*, onde o desenvolvedor é obrigado a usar classes internas do *framework* para usufruir das vantagens do padrão *MVC*.

Um dos princípios defendidos pelo *framework Spring* em seu projeto de *MVC* para *web*, e nos demais módulos também, foca na possibilidade do uso das principais funcionalidades do *framework* através da extensão de suas classes, mas nunca modificando uma determinada implementação de funcionalidade (“*Open for extension, closed for modification*”). A razão da aplicação deste princípio é que um número muito grande de métodos das classes centrais do *framework* está marcado como *final*, ou seja, a implementação destas funcionalidades não devem ser modificadas, mas apenas utilizadas por algum cliente específico.

O *framework* de *MVC* do *Spring*, assim como outros *frameworks MVC* para *web*, foi projetado em torno de um componente central, um *servlet*, responsável por tratar as requisições e dispará-las aos controladores responsáveis, além de fornecer funcionalidades adicionais que visam facilitar o desenvolvimento de aplicações *web*. Este *servlet*, denominado *DispatcherServlet*, baseia-se em um padrão de projeto para aplicações *J2EE*, chamado *Front Controller* (Figura 8).

Esse padrão tem por objetivo usar um *servlet* como ponto único de acesso à aplicação, responsável por gerenciar o tratamento de requisições, invocação de serviços de segurança como autenticação e autorização, delegação de processos de negócio, gerencia das escolhas apropriadas de *views*, tratamento de erros e gerência sobre a seleção de estratégias de criação de conteúdo (JOHNSON, 2006).



**Figura 8 - Processamento de uma requisição pelo módulo *web MVC* do *Spring***

Basicamente, quando uma requisição é processada pelo *framework MVC* do *Spring*, seis passos são executados (CRAIG, 2006).

1. O processo começa quando um cliente, normalmente um *browser*, envia um pedido. O pedido então é recebido pelo *DispatcherServlet* que irá procurar o controlador adequado para atendê-lo, examinando um ou mais objetos *HandlerMappings*.
2. O *HandlerMapping* entra em ação mapeando modelos de *URL(Uniform Resource Locator)* a objetos *Controller*.
3. Após o objeto *Controller* ser escolhido para uso pelo *DispatcherServlet*, o controle é passado para esse controlador para que venha a executar qualquer regra de negócio

necessária pela aplicação. Na verdade, um *Controller* bem projetado executa pouca ou nenhuma lógica de negócio, ao contrário, delega responsabilidades pela lógica de negócio para um ou mais objetos de serviço.

4. Após a conclusão da regra de negócio o controlador devolve um objeto do tipo *ModelAndView* para o *DispatcherServlet*. O *ModelAndView* pode conter um objeto *View* ou o seu nome lógico.
5. Se o objeto *ModelAndView* contiver o nome lógico de um *View*, o *DispatcherServlet* examinará um *ViewResolver* para determinar o objeto *View* que fará a resposta.
6. Finalmente, o *DispatcherServlet* despacha o pedido ao objeto *View* indicado pela *ModelAndView*. O objeto *View* é responsável por retornar uma resposta ao cliente.

Outro fator importante no processo de um pedido é a ampla seleção de classes de controladores oferecidas pelo *Spring*, com complexidades que variam desde a mais simples interface *Controller* até o mais poderoso controlador *wizard* e diversas camadas complexas entre eles. Isto separa o *framework Spring* dos outros *frameworks web MVC* como o *Struts* e o *WebWork*, onde a escolha dos desenvolvedores se limitam a somente uma ou duas classes controladoras.

Logo, o *MVC* do *Spring* mantém um agrupamento flexível entre a escolha de um determinado controlador para gerência de um pedido de um cliente e a opção de visualização escolhida para exibir informações. Este é um conceito poderoso, pois é possível mesclar diferentes partes do módulo de *MVC* do *Spring*, a fim de construir a camada *web* mais apropriada para aplicações *J2EE*.

#### 1.2.1.4 Integração com outros *Frameworks*

Embora o *framework* de *MVC* do *Spring* seja uma boa escolha para aplicações *web J2EE*, outros *frameworks open source* disponíveis no mercado e que já usufruem do status de “padrão de fato”, pela comunidade *enterprise*, podem ser uma ótima escolha. Tendo em vista que a opção por usar as vantagens do *Spring* no trato com as camadas de serviço e de acesso a dados não inviabilizará o uso de *frameworks* de *MVC* de outros projetos.

O *Spring* integra-se facilmente com outros *frameworks MVC* como o *Jakarta Struts*, o *Java Server Faces (JSF)*, *Tapestry* e *WebWork*. Com o *framework Struts* essa integração é feita de duas maneiras. A primeira opção é declarar os controladores do *Struts* no arquivo de

contexto da aplicação do *Spring*, enquanto que, a segunda opção, passa por configurar o *Struts* para que delegue o controle de seus controladores aos *beans* do *Spring*, possibilitando uma solução de menor acoplamento, embora mais complexa.

Quanto aos *frameworks JSF* e  *Tapestry* o processo de integração é simplificado e semelhante, bastando que seja apenas adicionado um *plug-in* na aplicação para que as configurações de integração sejam satisfeitas.

Já o *framework WebWork*, assim como o *Struts*, possui duas possibilidades de integração, dependendo da versão a ser utilizada. Com a primeira versão do *WebWork* basta adicionar os controladores no arquivo de contexto da aplicação fornecido pelo *Spring*. Com a segunda versão, o *WebWork* tem a capacidade e habilidade de associar os *beans* que foram configurados externamente no arquivo de configuração do *framework Spring*. Ou seja, o *Spring* foi desenvolvido para ser o menos intrusivo possível, deixando a opção das escolhas de tecnologias, a serem usadas nas camadas da aplicação, por conta dos desenvolvedores *J2EE*.

## 2 ESTUDO DE CASO

À medida que aumenta a oferta de soluções de código aberto para uso na plataforma Java, principalmente em ambientes *J2EE*, o uso dessas tecnologias tornou-se um processo de aprendizado complicado e oneroso para os desenvolvedores *enterprise*.

Pensando em resolver estes problemas Matt Raible criou um projeto *open source*, denominado *AppFuse*(fusível), descrito em seu *blog Raible Designs*(RAIBLE, 2006). Este projeto de aplicação foi concebido inicialmente com o intuito de servir como repositório central, onde Matt poderia fazer seus testes com as novas *APIs J2EE* e liberá-las em seu *blog*. Em pouco tempo a *AppFuse* ganhou popularidade, por ser uma excelente fonte de consulta para a integração de diferentes tecnologias de *frameworks J2EE*.

*Appfuse* pode ser definido como o esqueleto do projeto, similar a criação de um projeto passo a passo usando alguma ferramenta para desenvolvimento integrado de aplicações (*Integrated Development Environment - IDE*). Quando um projeto é criado usando a *AppFuse*, as escolhas de quais tecnologias de *framework* utilizar é feita através de um *prompt* de comando pelo usuário, usando como ferramenta de automação de processos de construção, documentação e empacotamento o *Jakarta Ant*(RAIBLE, 2006).

Assim como as ferramentas *IDE*, na criação de um novo projeto, a *AppFuse* cria projetos contendo classes e arquivos necessários para iniciar de forma adequada o desenvolvimento da aplicação. Esses arquivos são usados para implementar algumas funcionalidades necessárias, mas também, servem de exemplo aos desenvolvedores na hora de criar suas próprias aplicações. Outra vantagem é garantir para os desenvolvedores o processo de configuração dos *frameworks open source* usados pela aplicação. Logo, o projeto já é previamente configurado para se integrar com a camada de banco de dados, implantar a aplicação em um *web server*, autenticação de usuários, entre outros. Garantindo a implementação de serviços de segurança de forma integrada sem a necessidade de configuração adicional(RAIBLE, 2006).

As primeiras versões da *AppFuse* suportavam somente o *framework MVC* para *web Jakarta Struts* e *hibernate* para persistência. Atualmente, além de *hibernate*, adicionou suporte ao *iBATIS* como *framework* de persistência(RAIBLE, 2006). E quanto a *web*, o aumento de opções foi grande, sendo possível utilizar hoje *frameworks* como o *Java Server*

*Faces (JSF), Spring MVC, Struts, Tapestry* ou *WebWork*. Os principais serviços oferecidos pela aplicação dizem respeito à autenticação e autorização; gerenciamento de usuário; e-mail; suporte a *implementação de camadas de sockets seguras com Secure Sockets Layer(SSL)*; reescrita de *URL*; decoração de páginas; *templates* para *layout*; *upload* de arquivos, etc.

Logo, a escolha pela aplicação *AppFuse* foi motivada pela possibilidade de uma aplicação *J2EE* usufruir de características como testabilidade, integração, automação, segurança e geração, de forma automatizada, de aplicações *web* prontas para uso pelos desenvolvedores *enterprise*. Citando ainda, a diversificada fonte de documentação para essa aplicação, como também para as tecnologias de *frameworks* a serem integradas no projeto de *software*, sem esquecer da grande comunidade de apoiadores do projeto que sempre buscam inovar nas funcionalidades oferecidas pela *AppFuse*.

## 2.1 Descrição da Aplicação

A aplicação desenvolvida é um exemplo de aplicação usando a solução proposta pela *Appfuse*, porém, será usada uma versão simplificada dessa plataforma, denominada pelos criadores da *AppFuse* de projeto *Equinox*. Esse projeto foi inspirado na criação de uma aplicação na medida em que o usuário vai lendo a documentação do *framework*, como o projeto *struts-blank*, que é uma implementação mínima do *framework Jakarta Struts* disponível no sitio da fundação Apache(APACHE, 2006).

Como o projeto *Equinox* trata-se de uma versão simplificada da *AppFuse* alguns serviços não estão disponíveis para uso pela comunidade de desenvolvedores que optam por começar a desenvolver aplicações *enterprise* usando essa solução. Não estão disponíveis serviços como autenticação e autorização via *framework* de segurança *Acegi Security*, disponível somente na versão completa da *AppFuse*, além de outros como gerenciamento de usuários, geração automática de código para aplicações baseadas em *CRUD(create, retrieve, update and delete)* e *upload* de arquivos.

*Equinox* é uma aplicação para desenvolvimento de soluções *J2EE* baseadas no *frameworks Struts e Spring*. Possui uma estrutura de diretórios pré-definida, via *Jakarta Ant*, todos os arquivos necessários, como os *JARs*, necessários para utilizar os *frameworks Struts, hibernate e Spring* já estão adicionados as bibliotecas do projeto criado.

## 2.2 Modelagem da Aplicação

A aplicação tem por objetivo principal gerenciar usuários, onde será possível realizar operações de *CRUD* pela aplicação. Sua arquitetura é disposta em três camadas, uma camada *web*, uma camada média responsável pela lógica de negócios da aplicação e uma camada de acesso a dados.

Após a instalação inicial do projeto, é possível adicionar novos usuários ao repositório da aplicação, de forma simples, sem grandes codificações por parte dos desenvolvedores. À medida que são desejadas outras funcionalidades, estas devem ser adicionadas nos arquivos de configuração da aplicação. Logo, são duas as páginas principais de interação com o usuário, para este projeto mínimo, conforme as ilustrações abaixo. Tendo em vista, que este projeto tem como foco mostrar como é possível simplificar o processo de configuração inicial de projetos J2EE, e não mostrar sistemas de grande porte e inúmeras funcionalidades complexas de sistemas corporativos, mas sim, como configurá-las inicialmente.

Equinox ~ User List - Mozilla Firefox

Arquivo Editar Exibir Ir Favoritos Ferramentas Ajuda

http://demo.raibledesigns.com/equinox-struts/users.html

SISCOpweb Macroatividade - Gestão de Projetos Equinox ~ User List

**Equinox Powered**

Equinox is a lightweight version of **AppFuse** designed to accelerate starting a webapp with the **Spring Framework**.

Design and CSS donated by **Boér Attila**.

Resources

- Equinox
- Spring Framework
- Spring Live - Book
- Spring Live - Weblog

Add User

31 items found, displaying 1 to 10.  
[First/Prev] 1, 2, 3, 4 [Next/Last]

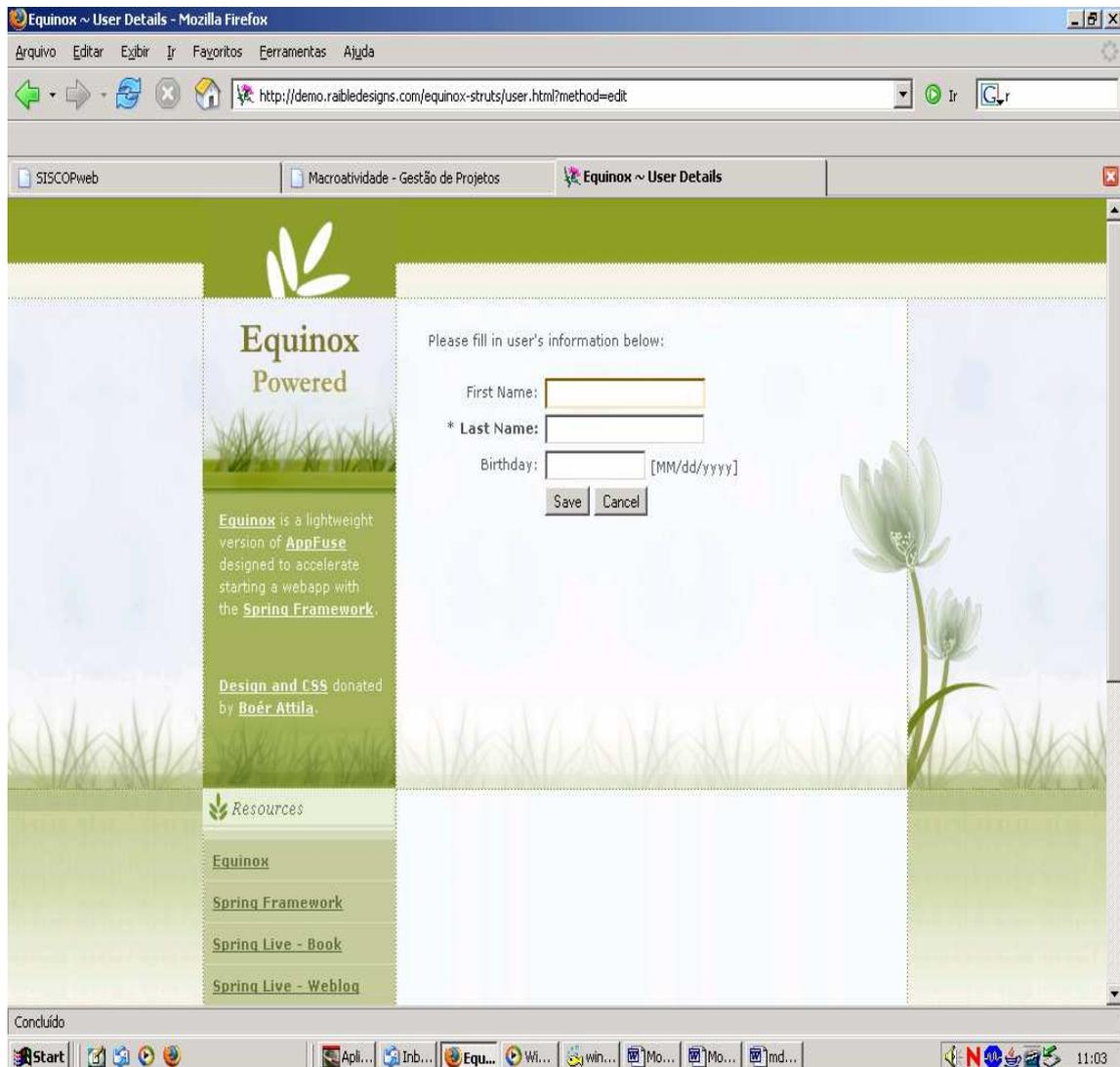
User Id	First Name	Last Name	Birthday
54	asdasda	sadasd	
5	Dion	Almaer	
6	Calvin	Austin	
7	Doug	Bateman	
8	Clinton	Begin	
9	Cedric	Beust	
10	Geert	Bevin	
11	Naren	Chawla	
12	Eugene	Ciurana	
13	Doug	Clark	

Export options: [CSV](#) | [Excel](#) | [XML](#) | [PDF](#)

Concluído

Start Apl... Inb... Equ... Wi... win... Mo... Mo... md... 10:55

**Figura 9 – Página de listagem de usuários do repositório do aplicativo Equinox**



**Figura 10 – Página de cadastro de novos usuários do aplicativo Equinox**

A interação entre objetos das camadas principais da aplicação acontece da seguinte forma: objetos *Action* da camada de apresentação, disponibilizados pelo *framework Struts*, são responsáveis por executar as chamadas a objetos de negócio do tipo *business delegate*, que por sua vez acessam objetos *DAO* (figura 11).

Objetos do tipo *business delegate* implementam a solução proposta pelo *pattern J2EE Business Delegate*. Esse padrão foi proposto para solucionar problemas encontrados na camada de apresentação de uma aplicação *web* onde componentes dessa camada por interagirem diretamente com objetos da camada de negócios estavam expondo detalhes de implementação destes. Logo, os componentes da camada de apresentação tornavam-se

vulneráveis a quaisquer mudanças de implementação da lógica de negócio da aplicação, ou seja, o nível de acoplamento entre estas camadas era alto demais.

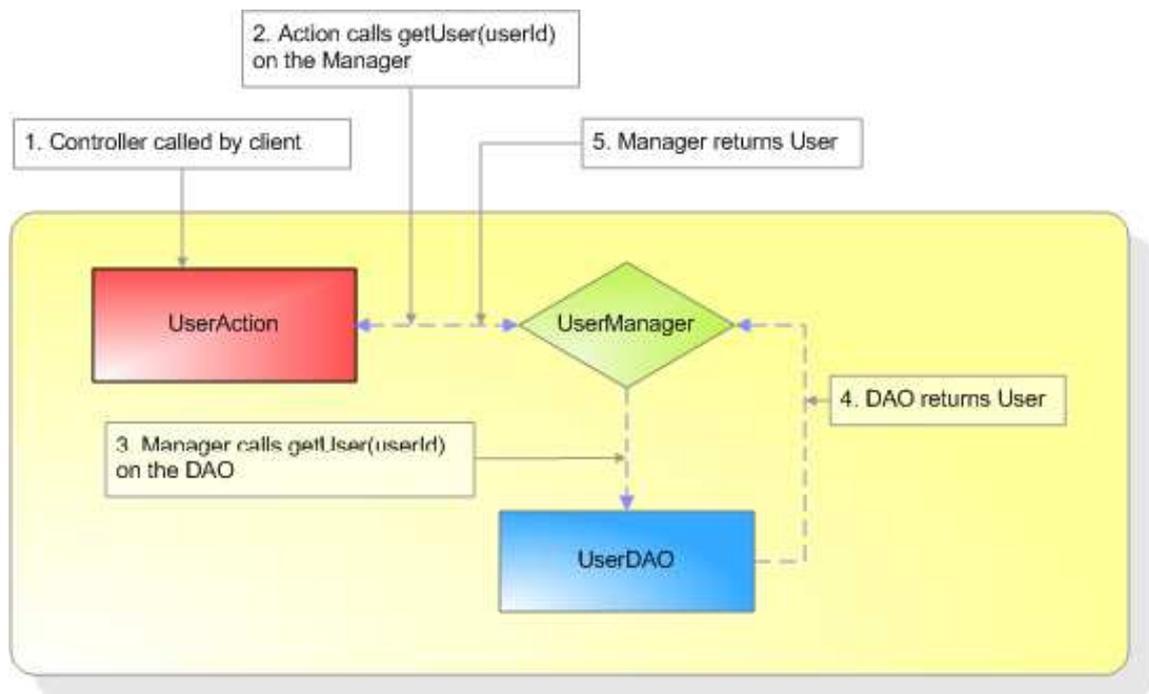


Figura 11 – *Workflow* da aplicação Equinox.

Sendo assim, através do uso do padrão *Business Delegate* (Figura 12), foi possível reduzir o acoplamento entre as camadas de apresentação e de negócio, escondendo os detalhes de implementação entre elas.

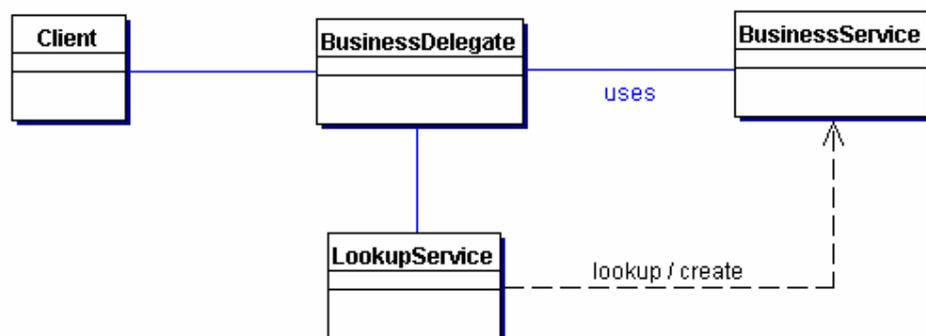


Figura 12 – Padrão *Business Delegate*

A aplicação usa o *framework Jakarta Struts* como *framework* de *MVC* para *web*, devido a grande familiaridade conquistada por este *framework* pela comunidade de desenvolvedores *J2EE*. Tendo em vista, que a grande vantagem no uso do *framework Spring*, está em usar serviços como gerenciamento declarativo de transações, injeção de dependências entre objetos e suporte a persistência através da integração com ferramentas de mapeamento objeto-relacional como o *hibernate* ou *iBATIS*.

Após descarregar o *JAR* do projeto *Equinox*, e fazer as devidas configurações de variáveis de ambiente, inicialização do servidor *web*, etc. Os usuários poderão testar a aplicação, além do teste via *Ant*, visualizando a página inicial, armazenada localmente na máquina de desenvolvimento, conforme ilustração abaixo (figura 13).



**Figura 13 – Página inicial da aplicação Equinox**

Após verificar o sucesso na inicialização do projeto, o usuário passa a dispendar seu tempo criando objetos e adicionando-os aos diretórios criados anteriormente pela ferramenta *Jakarta Ant*. Os passos restantes para criação de uma aplicação mínima, porém funcional, estão assim descritos:

- **Criar testes unitários para a camada de persistência.** Para camada de persistência o projeto *Equinox* usa como *framework* principal o *hibernate*, responsável por relacionar objetos Java com as tabelas relacionais do banco de dados escolhido para a aplicação.
- **Configuração do *hibernate* para trabalho com o *Spring*.** Depois de terem sido criadas as classes básicas da aplicação, ou seja, os *POJOS*, que são representações de tabelas em banco de dados, faz-se necessário criar arquivos de configurações em *XML* para linkar estes objetos com as tabelas relacionais do banco de dados. Além de relacionar as propriedades dos *beans* com os registros do banco, é possível ainda, com o uso do *framework hibernate* usar funcionalidades de sistemas gerenciadores de banco de dados, como relacionamento entre tabelas, restrições de integridade, ajustes de performance, etc.
- **Configuração para uso do *Spring* pela aplicação.** É fácil o processo de configuração de aplicações *J2EE* para trabalharem com o *framework Spring*. Basta adicionar ao arquivo *web.xml*, responsável por carregar o contexto inicial de uma aplicação *web*, o nome de uma classe do *Spring* responsável por carregar suas configurações iniciais. Essa classe é um *servlet* que será inicializado no momento que aplicação *web* for estartada.
- **Configurar as *DAOs* da aplicação.** O *Spring* através de seu arquivo de configuração, *ApplicationContext.xml*, seta as propriedade necessárias para trabalhar com as *DAOs* a serem utilizadas pelo *hibernate*. Graças à utilização de uma classe de suporte para *hibernate*, diferencial em relação a outros *frameworks web*, o *Spring* é capaz de detectar a existência de sessões abertas na aplicação, ou seja, se sessões já foram abertas por outras camadas da aplicação, como a camada *web*, por exemplo. Através desta detecção o *framework* decide se é necessário à criação de uma nova sessão ou se é mantida a sessão existente, procedimento conhecido como “abertura de sessões em *views*” (*Open Session in View*), descrita no padrão *J2EE* para carregamento de coleções de dados de forma “preguiçosa” (*lazy*).
- **Criação de gerentes (*managers*).** Uma prática recomendada na especificação *J2EE*, descrita pela *Sun*, diz respeito a projetar aplicações *enterprise* dispostas em camadas bem divididas. O *framework* usa para prover a separação necessária

entre a camada de *DAOs* e a camada *web* as vantagens oferecidas por objetos *Business Delegate*. Os objetos *Business Delegate* do *Spring* são conhecidos como *managers* e são interfaces que possuem as mesmas assinaturas de métodos que os objetos *DAO*, porém, a principal diferença é que foram projetados para trabalhar de forma amigável com a camada *web* (*web-friendly*), utilizando, por exemplo, objetos do tipo *String* enquanto *DAOS* utilizam tipos primitivos da linguagem Java. Objetos *manager* são utilizados para armazenar toda a lógica de negócio necessária pela aplicação.

- **Declaração das transações.** Para prover o gerenciamento declarativo de transações o *framework Spring* utiliza um objeto do tipo *ProxyFactoryBean*(`org.springframework.aop.framework.ProxyFactoryBean`). Esse objeto é responsável por criar diferentes implementações de uma classe, utilizando a *AOP* para interceptar os métodos necessários para o controle transacional da aplicação. Através do arquivo de metadados, *ApplicationContext.xml*, é possível setar as propriedades e atributos necessários para interceptação de métodos transacionais pela aplicação.
- Configurar os objetos *Action* do *framework Struts*, bem como as configurações necessárias para fazer com que trabalhe com o *Spring*, adicionando ainda, as páginas e arquivos de layout necessários à aplicação web.

## CONCLUSÕES E TRABALHOS FUTUROS

Atualmente, inúmeras são as possibilidades de utilização de soluções de *frameworks* para construção de aplicações *J2EE*, mas o ponto principal a ser procurado pela comunidade de desenvolvedores foca na simplificação e automatização do processo de integração dessas tecnologias, com o objetivo de criar projetos mais robustos e facilmente integráveis, retirando a complexidade maçante de configuração dessas ferramentas.

Os ideais propostos pela solução do projeto *AppFuse*, busca, em primeira instância, simplificar a tarefa do desenvolvedor, pois, definido as escolhas de quais tecnologias utilizar em uma aplicação, os arquivos necessários para desenvolver o projeto *J2EE* já são pré-configurados pela aplicação, bem como toda a estrutura de hierarquia dos principais pacotes do projeto. Ao desenvolvedor fica a possibilidade de trabalhar apenas com os componentes que visam implementar o objetivo fim do projeto, ou seja, as classes de negócio.

Aos iniciantes nos conceitos da especificação *J2EE* a melhor opção é começar usando a aplicação *Equinox*, pois ela simplifica ainda mais a criação do projeto e exclui alguns serviços mais complexos que devem ser adicionados à medida que os usuários absorvem os conceitos da plataforma.

O objetivo central desse projeto foi tentar mostrar uma alternativa ao desenvolvimento de aplicações *web* com *J2EE*, tomando como ferramenta o *framework Spring*. Para tanto, foram necessárias algumas considerações a respeito deste *framework*, mostrando as principais características incorporadas ao seu modelo de programação como a Programação Orientada a Aspecto (*AOP*) e a Inversão de Controle (*IoC*).

A especificação *J2EE* foi construída para suprir, em um primeiro momento, deficiências em sistemas de grande porte, mas acabou por sofrer algumas mudanças na sua última versão a *JEE 1.5*, com o intuito de atender demandas de sistemas menores. Esta versão baseia-se nas soluções *enterprise* propostas pelos chamados “*containers leves*” (*lightweight containers*), simplificando o modelo de programação defendido pela especificação de *EJBs*. O *Spring*, assim como outros *containers*, busca proporcionar aos usuários as características disponibilizadas por ambientes *EJBs*, mas, simplificando a sua implementação.

Além das facilidades adicionadas pelo seu módulo de *AOP*, principalmente, na possibilidade de projetar camadas de serviços contendo apenas lógica de negócio e não serviços complexos que deveriam ser supridos pelos *frameworks J2EE*. O módulo de *MVC*

deste *framework* ainda não possui o estatus de “padrão de fato” como seu co-irmão *Jakarta Struts*, porém, os serviços a serem disponibilizados para a camada *web* são semelhantes.

Logo, para trabalhos futuros, seria interessante explorar de forma mais detalhada a implementação do padrão *MVC* disponibilizado pelo módulo *web* do *Spring*, criando um projeto modular usando todos os recursos disponíveis do *container*, inclusive sua integração com o *framework* de segurança *Acegi Security System*.

## REFERÊNCIAS BIBLIOGRÁFICAS

FAYAD, M. C.; SCHIMIDT, D. C.; JOHNSON, R. E. **Building Application Frameworks – Object-Oriented Foundations of Frameworks Design**. Estados Unidos, Wiley. 1999.

CRAIG W. & RYAN B. **SPRING EM AÇÃO**. Editora Ciência Moderna. Rio de Janeiro. 2006.

RICHARDSON, C. **Developing with POJOs**. Disponível em: <<http://www.developer.com/java/ejb/article.php/3590731>>. Consultado em 05/08/2006.

GAMMA, E.; VLISSIDES J.; JOHNSON R.; HELM R. 1995. **Design Patterns – Elements of Reusable Object-Oriented Software**.

JOHNSON, R. **Expert One-on-One – J2EE Design and Development**. 2004.

DEEPAK A.; CRUPI J.; MALKS D. **Core J2EE Patterns: Best Practices and Design Strategies**. 2002.

RONCONI V. **Orientação a Aspectos – Conceitos Fundamentais**. Disponível em: <<http://www.portaljava.com.br/home//modules.php?name=Content&pa=showpage&pid=111>>. Consultada em 12/10/2006.

RAIBLE, M. **Seven simple reasons to use AppFuse**. Disponível em: <<http://www-128.ibm.com/developerworks/java/library/j-appfuse/>>. Consultado em 05/10/2006.

BLOG **Raible Designs**. Disponível em: <<http://raibledesigns.com>>. Consultado em 08/10/2006.

SITE **Equinox**. Disponível em: <<https://equinox.dev.java.net/>>.

SITE **AppFuse**. Disponível em: <<http://appfuse.org/>>.

JOHNSON, R. et al. **Spring – Java/J2EE Application Framework**. 2006. Disponível em: <<http://www.springframework.org/documentation>>.

**APACHE STRUTS**. Disponível em: <<http://struts.apache.org/>>.

RAIBLE, M. **Spring Live**. Disponível em: <<http://appfuse.org/>>. Consultado em 10/12/2006.