UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO

JACKSON BRAGA

PADRÃO “INVERSÃO DE CONTROLE COM INJEÇÃO DE DEPENDÊNCIA”: APLICAÇÕES EJB “VERSUS” *SPRING FRAMEWORK*

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

JACKSON BRAGA

PADRÃO “INVERSÃO DE CONTROLE COM INJEÇÃO DE DEPENDÊNCIA”: APLICAÇÕES EJB “VERSUS” *SPRING FRAMEWORK*

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação – CSTDSI – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Me. Everton Coimbra de Araújo.

MEDIANEIRA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Coordenação do Curso Superior de Tecnologia em
Desenvolvimento de Sistemas de Informação

TERMO DE APROVAÇÃO

PADRÃO "INVERSÃO DE CONTROLE COM INJEÇÃO DE DEPENDÊNCIA": APLICAÇÕES EJB "VERSUS" SPRING FRAMEWORK

Por

Jackson Braga

Este Trabalho de Diplomação (TD) foi apresentado às 13:50 h do dia 17 de junho de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O candidato foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Everton Coimbra de Araújo UTFPR – *Campus* Medianeira (Orientador)	Prof. Claudio Leones Bazzi UTFPR – *Campus* Medianeira (Convidado)
Prof. Juliano Rodrigo Lamb UTFPR – *Campus* Medianeira (Convidado)	Prof. Juliano Rodrigo Lamb UTFPR – *Campus* Medianeira (Responsável pelas atividades de TCC)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

AGRADECIMENTOS

Em primeiro lugar agradeço o Professor Everton Coimbra de Araújo por sua dedicação, paciência e orientação nesse trabalho, pelas lições de vida adquiridas durante esse período, por grandes experiências e pelo grande aprendizado que tive. Muito obrigado Professor.

Agradeço a minha família que sempre esteve ao meu lado durante todo esse tempo me incentivando a não desistir diante dos obstáculos encontrados nessa caminhada.

Agradeço também aos amigos que estiveram, e estão, presentes em minha vida, compartilhando momentos importantes, descontraídos e engraçados.

Em especial agradeço a Deus por tudo o que me proporcionou até o presente momento.

RESUMO

O presente trabalho tem como objetivo inicial, apresentar o conceito teórico do padrão “Inversão de Controle com Injeção de Dependência” e das tecnologias EJB e *Spring Framework*, focando o desenvolvimento de aplicações corporativas. Posteriormente, o mesmo apresenta um estudo prático sobre o desenvolvimento de aplicações corporativas utilizando o EJB e o *Spring Framework*. O resultado obtido no estudo apresenta às principais características de ambas as tecnologias e suas respectivas vantagens e desvantagens no desenvolvimento de uma aplicação de estudo de caso.

Palavras-chave: Aplicação Corporativa. JAVA. Desenvolvimento de Software.

ABSTRACT

This paper has as objective start, bring the theoretical concepts of the standard "Inversion of Control with Dependency Injection" and the Spring Framework and EJB technologies, focusing on development enterprise applications. Subsequently, it presents a case study on the development of enterprise applications using EJB and Spring Framework. The result obtained in this study presents the main features of both technologies and their advantages and disadvantages in the development of an application case study.

Keywords: Enterprise Application. JAVA. Software Development.

LISTA DE FIGURAS

LISTA DE SIGLAS

AOP	-	*Aspect-Oriented Programming*
API	-	*Application Programming Interface*
CRUD	-	*Create, Retrieve, Update e Delete*
DAO	-	*Data Access Objects*
DI	-	*Dependency Injection*
EJB	-	*Enterprise Java Beans*
HTTP	-	*HyperText Transfer Protocol*
IoC	-	*Inversion of Control*
JMS	-	*Java Message Service*
JPA	-	*Java Persistence API*
MDB	-	*Message Driven Bean*
MVC	-	*Model-View-Controller*
ORM	-	*Object-Relational Mapping*
POJO	-	*Plain Old Java Objects*

SUMÁRIO

1 INTRODUÇÃO

Ao se pensar em como verificar a viabilidade do desenvolvimento de aplicativos corporativos se utilizando do EJB e do *Spring Framework* juntamente com a utilização do padrão de Inversão de Controle com Injeção de Dependência é necessário ter claro qual a funcionalidade de cada um.

Isso se faz necessário devido a uma comum confusão que se tem ao tratar desse tema que é dizer que Inversão de Controle e Injeção de Dependência são sinônimos, enquanto na verdade a Inversão de Controle tem como objetivo oferecer uma maneira simples de prover dependências de objetos em forma de componentes e gerenciar o ciclo de vida dessas dependências enquanto a Injeção de Dependência ao ser aplicada recebe suas dependências em tempo de criação por alguma entidade externa que coordena cada objeto no sistema.

O *Enterprise* Java *Beans* – EJB é uma plataforma para construção de aplicações corporativas de modo a serem portáveis, desacopladas e reutilizáveis, permitindo projetos de aplicações escalonáveis, confiáveis e seguras, sem a necessidade de realizar todo o trabalho de desenvolvimento da infra-estrutura do projeto (PANDA, 2009), o EJB também permite a construção de aplicações corporativas Java sem ter que reinventar serviços necessários para a construção de uma aplicação, como as transações, seguranças, persistência automatizada, entre outras (PANDA, 2009).

O padrão Inversão de Controle com Injeção de Dependência é utilizado pelo *Spring*, que é um *framework* na forma de código aberto e de uso livre para facilitar a complexidade de desenvolvimento de aplicativos corporativos (JAVA FREE, 2009). Ele consiste em um *container*, para gerenciar componentes, e um conjunto de serviços de interfaces de usuário, transações e persistência (LEMOS, 2009). Sua utilidade não é limitada ao desenvolvimento de aplicativos corporativos, qualquer aplicativo Java pode se beneficiar do *Spring*, pois ele promove boas práticas de desenvolvimento, testabilidade, garante a flexibilidade e o desenvolvimento é através de um modelo de componentização baseada em classes Java Simples (POJO) (LEMOS, 2009).

1.1 OBJETIVO GERAL

Realizar um estudo sobre o desenvolvimento de aplicações corporativas utilizando o EJB e o *Spring Framework* juntamente com o padrão de Inversão de Controle com Injeção de Dependência

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

- Realizar um estudo sobre o padrão Inversão de Controle com Injeção de Dependência.
- Realizar um estudo sobre a especificação EJB.
- Realizar um estudo sobre o *Framework Spring*.
- Realizar um estudo de caso apresentando os principais pontos positivos e negativos no desenvolvimento de uma aplicação corporativa utilizando a especificação EJB e o *Framework Spring*.

1.3 JUSTIFICATIVA

O desenvolvimento de aplicações corporativas vem evoluindo constantemente e para acompanhar essa evolução novos recursos tecnológicos e padrões foram desenvolvidos ou aperfeiçoados para melhorar cada vez mais o desenvolvimento com o objetivo de deixá-lo mais prático, produtivo, estruturado, entre outras melhorias.

Devido à demanda da utilização de aplicações corporativas e a necessidade de saber que ferramenta utilizar para atender as especificações do projeto da aplicação, surgiu à necessidade de realizar um estudo sobre as ferramentas mais populares no desenvolvimento de aplicações corporativas.

Para alcançar esse objetivo prático e produtivo é possível se utilizar do *Spring Framework*, criado por Rod Johnson, que lida com a complexidade de desenvolvimento de aplicativos corporativos, e Segundo Panda, 2009 o EJB que é uma plataforma que permite a construção de aplicações corporativas de modo a

serem portáveis, de baixo acoplamento e reutilizáveis. Ambas são ferramentas utilizadas para o desenvolvimento de aplicações corporativas que utilizam o padrão de Inversão de Controle com Injeção de Dependência proporcionando as vantagens de sua utilização.

A utilização dessas ferramentas, digo *Spring Framework* e EJB, se da principalmente para facilitar o desenvolvimento, a manutenção do código e a realização de outros testes que se fizerem necessário.

1.4 METODOLOGIA

O método utilizado para o desenvolvimento desse trabalho inicialmente se deu através de pesquisas bibliográficas sobre o padrão Inversão de Controle com Injeção de Dependência, *Spring Framework* e EJB, essa pesquisa se deu separadamente visando entender e conceituar os mesmos. Posteriormente foi realizada aplicação utilizando o padrão Inversão de Controle com Injeção de Dependência fornecido no EJB, em seguida foi realizada outra aplicação com o mesmo padrão fornecido pelo *Spring Framework*, com objetivo de identificar suas particularidades. Após, foi destacado as vantagens e desvantagens das aplicações tanto das feitas utilizando o EJB como do *Spring Framework*.

1.5 ESTRUTURA DO TRABALHO

O presente trabalho é constituído de mais cinco capítulos. O Capítulo 2 apresenta um estudo sobre o padrão Inversão de Controle com Injeção de Dependência abordando definições, conceitos da utilização do padrão.

No Capítulo 3 é feito um estudo sobre o *Spring Framework*, onde são abordados seus conceitos, características e definições de uso.

No Capítulo 4 é feito um estudo sobre o EJB, onde são abordados seus conceitos, definições e principais características de uso.

No Capítulo 5, são desenvolvidos de uma aplicação de estudo de caso de utilização do *Spring Framework* e do EJB, comparando os seus resultados.

Por fim o Capítulo 6 ressalta as considerações finais sobre o desenvolvimento do trabalho.

2 INVERSÃO DE CONTROLE COM INJEÇÃO DE DEPENDÊNCIA (IOC/DI)

O princípio da 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. *Containers* de IoC servem para fazer a ligação entre dependentes e dependências, fazendo isso de várias maneiras. A IoC ainda se subdivide em Injeção de Dependência (*Dependency Injectioin* – DI) e Busca por Dependência (*Dependency Lookup*)(ARAGÃO JUNIOR, 2007).

2.1 INVERSÃO DE CONTROLE (*INVERSION OF CONTROL* - IOC)

Inversão de controle é referenciada como o princípio de *Hollywood*: "*Don't call us, we will call you"* (Não nos chame, nós chamamos você). Trata-se de transferir o controle da execução da aplicação para o *container* de IoC, que chama a aplicação em determinados momentos como na ocorrência de um evento. Através da IoC o *container* controla quais métodos da aplicação e em que contexto eles serão chamados (DBD, 2010).

O *design pattern Observer/Observable* é um bom exemplo do poder da IoC, um objeto registrado nunca sabe quando vai receber um aviso do *observable*, mas mesmo assim o objeto se registra nele esperando até que ele lhe envie uma notificação, então mais uma vez não é o objeto quem chama, quem o faz é o *observable* (ARAGÃO JUNIOR, 2007).

2.2 INJEÇÃO DE DEPENDÊNCIA (*DEPENDENCY INJECTION* – DI)

Ao aplicar a Injeção de Dependência (*Dependency Injection* – DI), os objetos recebem suas dependências em tempo de criação por alguma entidade externa que coordena cada objeto no sistema. Em outras palavras, as dependências são injetadas nos objetos. Por isso, DI significa uma inversão de responsabilidade com relação a como um objeto obtém referências a objetos colaboradores (WALLS, 2008).

O benefício-chave da DI é o baixo grau de dependência entre dois artefatos[1], ou seja, o baixo acoplamento. Se um objeto conhece apenas suas dependências por sua interface (não por sua implementação, nem como foram instanciadas), então a dependência pode ser trocada por uma implementação diferente sem que o objeto dependente conheça a diferença (WALLS, 2008).

2.3 IDENTIFICANDO O PROBLEMA

Para facilitar o entendimento será visto um exemplo de um gerenciador de uma escola, que faz apenas o controle dos Alunos existentes nela.

Na Figura 1 é definido a classe **GerenciadorAluno**.

```
1  import java.util.Iterator;
2  import java.util.List;
3
4
5  public class GerenciadorAluno{
6      private AlunoDAO alunoDAO;
7
8      public GerenciadorAluno() {
9          this.alunoDAO = new AlunoDAOMemoriaImpl();
10     }
11
12     public List listarAlunosPorTurma(String turma){
13         List alunos = alunoDAO.buscarAlunos();
14
15         for (Iterator it = alunos.iterator(); it.hasNext();) {
16             Aluno aluno = (Aluno) it.next();
17             if(!aluno.getTurma().equals(turma))
18                 it.remove();
19         }
20
21         return alunos;
22     }
23 }
```

Figura 1 - Classe GerenciadorAluno.

A implementação do método **listarAlunosPorTurma** (Linha 12) pede para que o **alunoDao** (objeto de busca) retorne todos os alunos que ele conhece. Então,

[1] Artefatos pode ser qualquer coisa que faz parte do projeto – uma classe, método, componente, pacote, entre outros.

é feita uma iteração na lista de alunos verificando e removendo os alunos que não pertencem à turma passada como parâmetro do método.

Para que o método **listarAlunosPorTurma** seja completamente independente de como os alunos sejam armazenados foi definida uma interface para o **alunoDAO** exibida na Figura 2.

```
1  import java.util.List;
2
3
4  public interface AlunoDAO {
5      public List buscarAlunos();
6  }
```

Figura 2 - Interface AlunoDao.

Com a utilização da interface pode-se ter um baixo acoplamento, mas em algum ponto é preciso obter uma classe concreta para realmente realizar a busca dos alunos.

O ponto principal é o **alunoDAO**, mais particularmente como o **GerenciadorAluno** adquire um **alunoDAO** específico. Neste caso, a criação do objeto **alunoDAO** está no construtor da classe **GerenciadorAluno** (Figura 1).

A classe **AlunoDAOMemoriaImpl** (Figura 3) implementa a interface **AlunoDAO**, ela mantém as informações dos alunos em memória.

```
1  import java.util.List;
2  import java.util.Vector;
3
4
5  public class AlunoDAOMemoriaImpl implements AlunoDAO{
6
7      private List<Aluno> alunos;
8
9      public AlunoDAOMemoriaImpl() {
10         alunos = new Vector<Aluno>();
11         alunos.add(new Aluno("José da Silva", "I21", "Informática"));
12         alunos.add(new Aluno("Maria dos Santos", "I21", "Informática"));
13         alunos.add(new Aluno("Joaquim Pereira", "A52", "Ambiental"));
14         alunos.add(new Aluno("José da Silva", "A52", "Ambiental"));
15     }
16
17     @Override
18     public List buscarAlunos() {
19         return alunos;
20     }
21
22     public List<Aluno> getAlunos() {
23         return alunos;
24     }
25
26     public void setAlunos(List<Aluno> alunos) {
27         this.alunos = alunos;
28     }
29 }
```

Figura 3 - Classe AlunoDAOMemoriaImpl.

Se for preciso utilizar o método **listarAlunosPorTurma** em um ambiente em que as informações dos aluno estiverem em uma base de dados, ou em um arquivo texto, ou em um arquivo XML será preciso desenvolver uma nova classe que recupere os dados do aluno. Como foi definida uma interface, não será preciso alterar a implementação do método **listarAlunosPorTurma**, mas ainda é preciso de um meio para obter uma instância do **alunoDAO** correto.

A Figura 4 ilustra a dependência para essa situação.

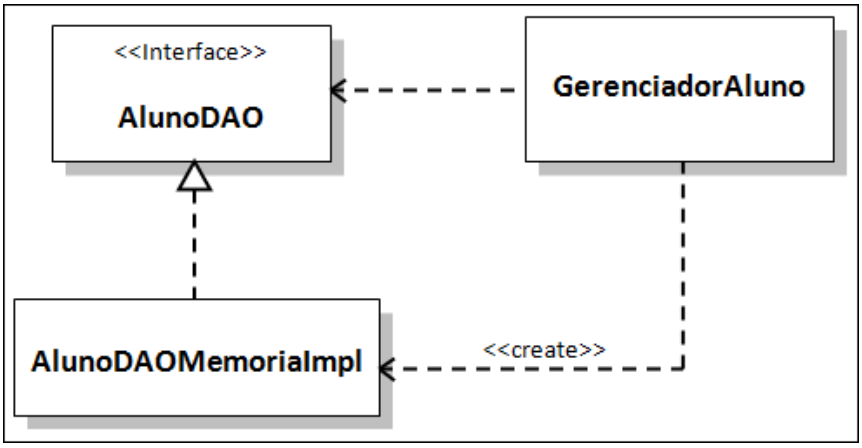

Figura 4 - Dependências da classe GerenciadorAluno.

A classe **GerenciadorAluno**, é dependente tanto da interface **AlunoDAO** quanto de sua implementação. Seria melhor se a classe **GerenciadorAluno** fosse dependesse somente a interface.

O problema é como realizar a ligação de modo que a classe **GerenciadorAluno** não conheça a implementação, e mesmo assim conseguir uma instância que mantenha o funcionamento. Para resolver esse problema é preciso inverter o controle de como adquirir a implementação injetando a dependência, ou seja, injetando a implementação da interface **AlunoDAO**.

Existem várias formas para a realização de injeção de dependência, mas não é a única maneira de resolver esse tipo de dependência.

2.4 FORMAS DE INJEÇÃO DE DEPENDÊNCIA

A forma básica de DI é ter um objeto separado, o montador (*assembler*), que preenche um campo em um objeto **GerenciadorAluno** com uma

implementação apropriada para a interface **AlunoDAO**, resultando em um diagrama de dependência parecido com o da Figura 5.

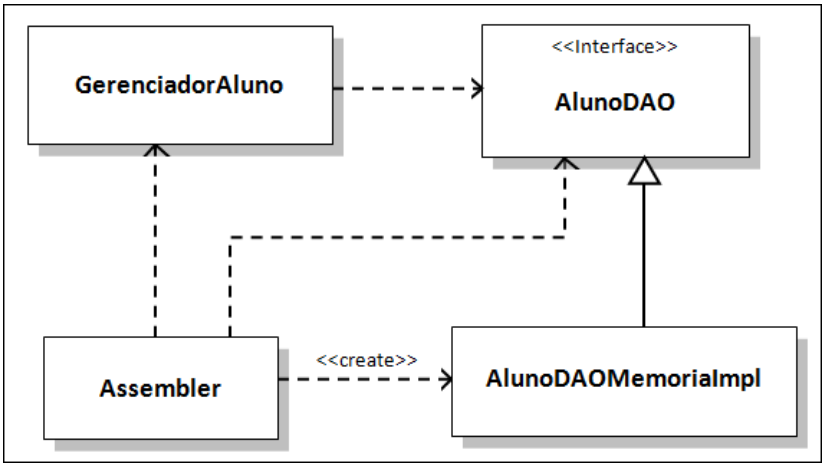

Figura 5 - Dependências para um Injetor de Dependência.

A DI pode ser feita de três maneiras diferentes, através de interfaces, *Interface Injection* (Injeção por Interfaces) ou IoC Tipo 1, através de métodos "*setters*" da especificação *JavaBeans*, *Setter Injection* (Injeção por Métodos Set) ou IoC Tipo 2 e através do construtor do objeto, *Constructor Injection* (Injeção por Construtores) ou IoC Tipo 3 (FOWLER, 2010). Na DI o código não fica poluído por 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 (ARAGÃO JUNIOR, 2007).

2.4.1 Injeção por Interface (*Interface Injection*)

Injeção por Interface é o processo pelo qual todas as dependências são injetadas em um objeto através de uma interface. No exemplo citado anteriormente a classe **GerenciadorAluno** poderia implementar uma interface para definir as operações necessárias para administrar a aplicação de configuração apropriada.

2.4.2 Injeção por Métodos *Set* (*Setter Injection*)

Injeção por Métodos *Set* é o processo pelo qual todas as dependências são injetadas em um objeto através dos métodos *setters*, que são definidos pela

especificação *JavaBeans* (FOWLER, 2010). Por exemplo, a classe **GerenciadorAluno** poderia implementar o método **setAlunoDAO** (Linha 10 da Figura 6) recebendo como parâmetro um objeto que implemente a interface **AlunoDAO**, que nada mais é do que o método *set* do atributo **alunoDAO**.

A Figura 6 exibe a implementação da classe **GerenciadorAluno** utilizando a Injeção por Métodos *Set*.

```
1  import java.util.Iterator;
2  import java.util.List;
3
4
5  public class GerenciadorAluno{
6      private AlunoDAO alunoDAO;
7
8      public GerenciadorAluno() { }
9
10     public void setAlunoDAO(AlunoDAO alunoDAO) {
11         this.alunoDAO = alunoDAO;
12     }
13
14     public List listarAlunosPorTurma(String turma){
15         List alunos = alunoDAO.buscarAlunos();
16
17         for (Iterator it = alunos.iterator(); it.hasNext();) {
18             Aluno aluno = (Aluno) it.next();
19             if(!aluno.getTurma().equals(turma))
20                 it.remove();
21         }
22
23         return alunos;
24     }
25 }
```

Figura 6 - Classe GerenciadorAluno utilizando Injeção por Métodos *Set*.

Como a classe **GerenciadorAluno** está adquirindo o objeto **alunoDAO** através de seu método *set*, ela não precisa mais se preocupar com a criação do objeto como era realizando anteriormente. Assim, removendo a dependência que a classe **GerenciadorAluno** tinha com a classe **AlunoDAOMemoriaImpl**, que implementa a interface **AlunoDAO** (FOWLER, 2010).

Uma das desvantagens é que existe a possibilidade de chamar o método **listarAlunosPorTurma** da classe **GerenciadorDAO** antes de passar as dependências, mas, considerando que foi esta a estratégia escolhida, o desenvolvedor deverá estar ciente desta desvantagem.

2.4.3 Injeção por Construtores (*Constructor Injection*)

Injeção por Construtores é o processo pelo qual todas as dependências são injetadas em um objeto através do construtor da classe (FOWLER, 2010). Por exemplo, a classe **GerenciadorAluno** recebe como parâmetro em seu construtor um objeto que implemente a interface **AlunoDAO**.

A Figura 7 exibe a implementação da classe **GerenciadorAluno** utilizando a Injeção por Construtores.

```
1  import java.util.Iterator;
2  import java.util.List;
3
4
5  public class GerenciadorAluno{
6      private AlunoDAO alunoDAO;
7
8      public GerenciadorAluno(AlunoDAO alunoDAO) {
9          this.alunoDAO = alunoDAO;
10     }
11
12     public List listarAlunosPorTurma(String turma){
13         List alunos = alunoDAO.buscarAlunos();
14
15         for (Iterator it = alunos.iterator(); it.hasNext();) {
16             Aluno aluno = (Aluno) it.next();
17             if(!aluno.getTurma().equals(turma))
18                 it.remove();
19         }
20
21         return alunos;
22     }
23 }
```

Figura 7 - Classe GerenciadorAluno utilizando Injeção por Construtor.

Ao utilizar a Injeção por Construtor, a classe **GerenciadorAluno**, também, não precisa se preocupar com a implementação do objeto **alunoDAO**, pois ela recebe o objeto que implementa a interface **AlunoDAO** em sua criação (FOWLER, 2010).

Esta estratégia evita o problema que pode acontecer quando se utiliza a Injeção por Métodos *Set*, porque, a dependência é injetada na criação do objeto **GerenciadorAluno** através de seu construtor, assim, ao chamar o método **listarAlunosPorTurma** a classe já terá uma instância para o objeto **alunoDAO**.

3 *SPRING*

Spring é um *framework* de código aberto que foi criado por Rod Johnson para lidar com a complexidade de desenvolvimento de aplicativos corporativos. O *Spring* torna possível usar simples *JavaBeans* para conseguir coisas que antes só eram possíveis com EJBs. Porém, a utilidade do *Spring* não é limitada ao desenvolvimento do lado do servidor (WALLS, 2008). Qualquer aplicativo pode se beneficiar do *Spring* provendo simplicidade, aumentando a produtividade de desenvolvimento, o desempenho em tempo de execução e ao mesmo tempo prove uma cobertura para testes e baixo acoplamento (PACHECO, 2010).

O *Spring* é um *framework* e um *container* leve orientado a aspectos, com injeção de dependência, que ajuda no desenvolvimento de código de aplicativo com baixo acoplamento (WALLS, 2008). O *Spring* é extremamente modular, assim é possível usar somente as partes necessárias, os módulos são construídos a partir da base de injeção de dependência e Programação Orientada a Apectos (*Aspect-Oriented Programming* – AOP), criando uma plataforma cheia de recursos, sobre a qual se constroem os aplicativos (PACHECO, 2010).

O Spring é composto de vários módulos bem definidos (Figura 8).

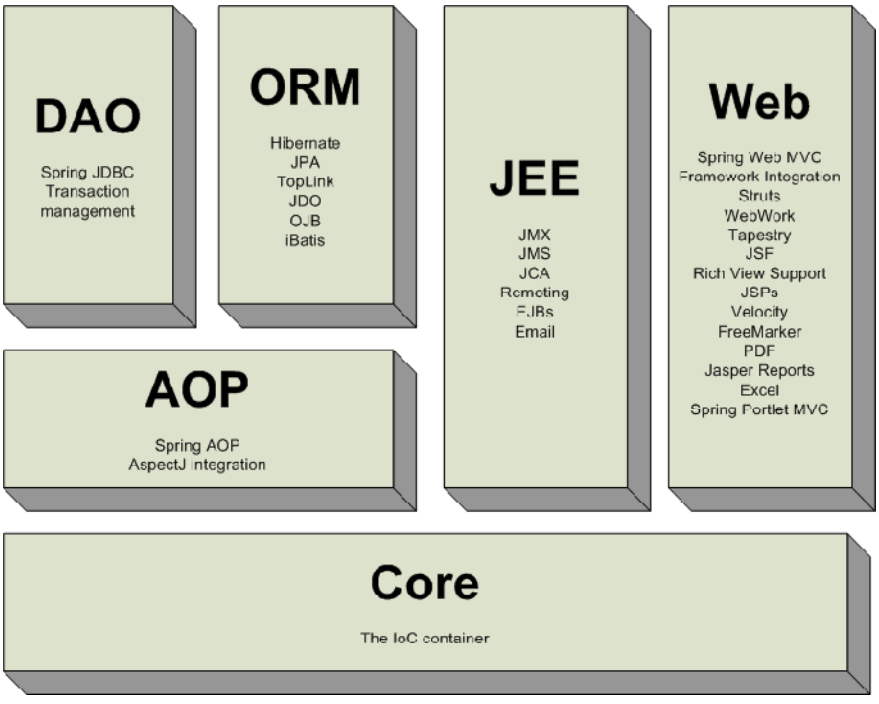

Figura 8 - Estrutura do *Spring Framework* na versão 2.5.x
Fonte: PACHECO (2010).

Ao serem tomados como um todo, estes módulos fornecem tudo o que é necessário para desenvolver aplicativos para soluções corporativas. Mas não é necessário ter o *Spring* como base para seus aplicativos (WALLS, 2008).

O *Spring* oferece pontos de integração com diversos outros *frameworks* e bibliotecas, de maneira que não tenha que desenvolvê-los por si próprio (WALLS, 2008).

3.1 MÓDULO *CORE*

Como se pode observar, todos os módulos do *Spring* são construídos em cima do módulo *Core*. O módulo define como os *beans* são criados, configurados e gerenciados (WALLS, 2008).

Em um aplicativo baseado no *Spring*, seus objetos viverão dentro do *container*. O *container* do *Spring* é um *software* que provê o ambiente necessário para a utilização dos componentes do *Spring* e também é responsável por gerenciar todo o ciclo de vida e a configuração dos objetos. Como ilustrado na Figura 9, o *container* criará e configurará os objetos, estabelecerá uma associação entre eles e gerenciará seu ciclo de vida completo, desde o nascimento à morte (WALLS, 2008).

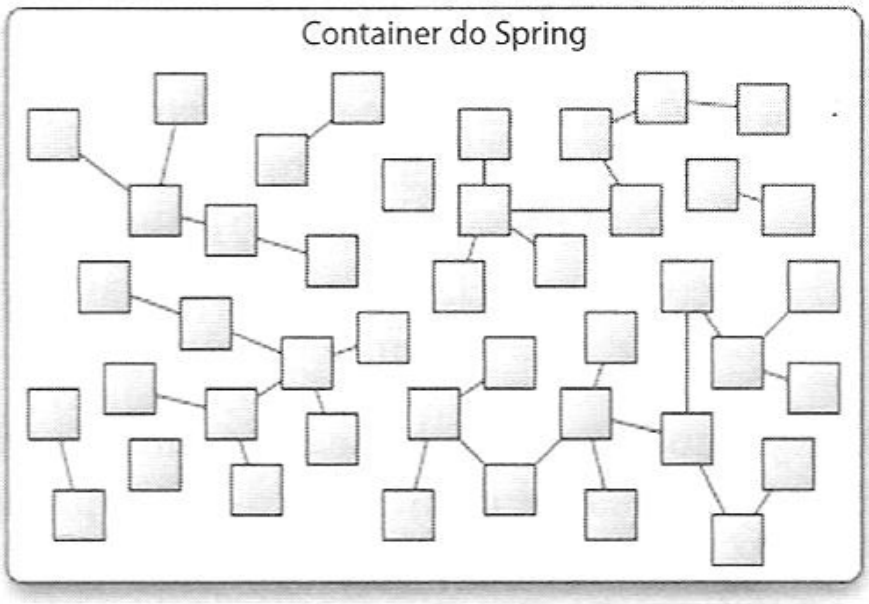

Figura 9 - *Container* do *Spring*.
Fonte: WALLS (2008).

O *container* é o núcleo do *Spring*. O *container* do *Spring* usa DI para gerenciar os componentes que formam um aplicativo (WALLS, 2008). O *container*, também, provê a centralização, automação de configuração e escrita para seus objetos de negócio. É um *container* não intrusivo, capaz de suportar sistemas complexos de um conjunto de componentes (POJO) de baixo acoplamento, consistência e transparência. O *container* traz agilidade ao desenvolvimento, testabilidade e uma alta escalabilidade, permitindo que os componentes de *software* possam ser desenvolvidos e testados isoladamente (PACHECO, 2010).

3.2 MÓDULO AOP

Totalmente integrado com o gerenciamento configuracional do *Spring*, é possível utilizar AOP com qualquer objeto gerenciado pelo *Spring* Por exemplo, adicionando aspectos em gerenciamento transacional (PACHECO, 2010).

Em *Spring*, os aspectos são entrelaçados nos *beans* gerenciados pelo *Spring* em tempo de execução, agrupando-os com uma classe *Proxy*. Como ilustrado na Figura 10, a classe *Proxy* é definida como o *bean* destino, interceptando as chamadas do método, realiza uma operação como, por exemplo, a validação de acesso e encaminhando estas chamadas para este *bean* (WALLS, 2008).

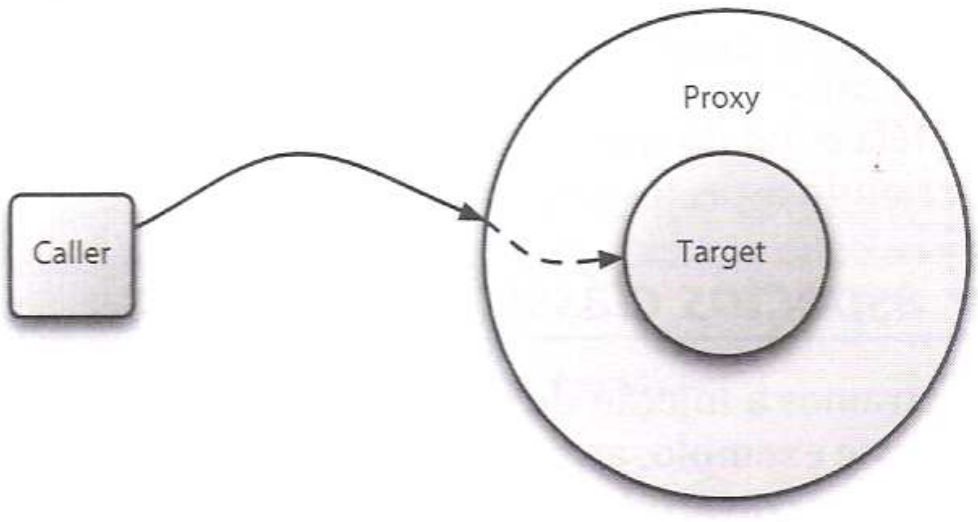

Figura 10 - Os aspectos do *Spring* são implementados como *proxies* que agrupam o objeto destino.
Fonte: WALLS (2008).

Entre o tempo que o *Proxy* intercepta a chamada de método e o tempo que invoca o método do *bean* destino, o *Proxy* desempenha a lógica do aspecto (WALLS, 2008).

O *Spring* não cria um objeto com *Proxy* até que o *bean* com *Proxy* seja necessário pela aplicação. Como o *Spring* cria *proxies* em tempo de execução, não é preciso de um compilador especial para entrelaçar aspectos AOP no *Spring* (WALLS, 2008).

3.3 MÓDULOS DAO E ORM

A estratégia que o *Spring* utiliza para os módulos DAO (*Data Access Objects*) e ORM (*Object-Relational Mapping* – Mapeamento Objeto-Relacional) é muito parecida, a diferença é que o módulo DAO fornece uma camada de abstração para ao código JDBC (Java *Database Connectivity*) e o módulo ORM provê a integração do *Spring* com outros *frameworks* de persistências de mapeamento objeto-relacional, como o *Hibernate*, JPA e iBatis, entre outros (CHHATPAR, 2010).

O *Spring* fornece uma maneira fácil de gerenciar a manipulação de exceção e os códigos de erro emitidos por diferentes fornecedores de base de dados. Além disso, o pacote JDBC fornece uma maneira de realizar gerenciamento de transações programáticas e declarativas, não apenas para classes implementando interfaces especiais, mas também para todos seus objetos Java simples (POJOs) (CHHATPAR, 2010)

O *Spring* ajuda a isolar a camada de acesso a dados do restante da aplicação fornecendo uma hierarquia de exceção consistente que é usada por todos os *frameworks* (WALLS, 2008).

Utilizando JDBC puro não é possível fazer nada sem ser forçado a capturar a exceção *java.sql.SQLException*. A grande questão em torno do *SQLException* é como deve ser tratado quando obtido. Em vez de ter um tipo de exceção diferente para cada problema possível, a *SQLException* é tratada como uma exceção única para todos os problemas relacionados a acesso a dados (WALLS, 2008).

Alguns *frameworks* de persistência fornecem uma rica hierarquia de exceções. Por exemplo, o *Hibernate* fornece uma grande quantidade de exceções,

cada uma direcionada para um problema em específico de acesso a dados. Mas, as exceções do *Hibernate* são proprietárias do *Hibernate* (WALLS, 2008).

O *Spring* fornece uma hierarquia de exceções de acesso a dados disponibilizando diversas exceções de acesso a dados para resolver o problema com a hierarquia de exceções genéricas do JDBC e a hierarquia proprietária dos *frameworks* de persistência. Cada exceção é descritiva do problema pelo qual é lançada (WALLS, 2008). A hierarquia de exceções não é associada a nenhuma solução de persistência. Assim, é possível utilizar o *Spring* para emitir um conjunto considerável de exceções independente do mecanismo de persistência escolhido (CHHATPAR, 2010).

A principal meta da estrutura do *Spring* é padronizar e simplificar o trabalho com tecnologias de acesso de dados, como JDBC, Hibernate, JPA, entre outros. Portanto, utilizando a estrutura do *Spring*, fica bem fácil alternar de uma tecnologia de acesso a dados para outra (CHHATPAR, 2010).

3.4 MÓDULO JEE

O *Spring* provê a integração com uma série de tecnologias Java EE, como *Remoting*, integração com EJB, JMS, JMX, entre outros, dos quais serão abordados *Remoting*, JMS, JMX (JOHNSON, 2010).

3.4.1 *Remoting*

O suporte para *remoting* do *Spring* permite a exposição da funcionalidade de seus objetos Java como objetos remotos, ou se for necessário acessar os objetos remotamente, o *remoting* também facilita o trabalho de associação de objetos remotos em seus aplicativos como se fossem POJOs locais (WALLS, 2008).

Em uma aplicação, é comum a necessidade de realizar o acesso remoto a uma outra aplicação ou serviço. O *Spring* facilita o acesso remoto abstraindo muito a complexidade de acesso, é possível usar os componentes que já estão sendo gerenciados pelo *Spring*, assim, obtendo uma máxima integração entre seus componentes de negócio e os componentes remotos (JOHNSON, 2010).

O *Spring* disponibiliza várias opções de *remoting*, incluindo *Remote Method Invocation* (RMI), *Hessian*, *Burlap* e o próprio invocador HTTP do *Spring* (WALLS, 2008).

3.4.2 JMS

A desvantagem do *remoting* é que o mesmo depende da confiabilidade da rede e que as extremidades da comunicação estejam disponíveis. A comunicação orientada a mensagem por outro lado, é mais confiável e garante a entrega das mensagens, mesmo se a rede e os pontos finais não sejam confiáveis (WALLS, 2008).

O *Java Message Service* (JMS) do *Spring* é assíncrono e ajuda a enviar mensagens para filas e tópicos de mensagens JMS. Ao mesmo tempo, também auxilia na criação de POJOs dirigidos a mensagens que são capazes de consumir mensagens não-sincronizadas (WALLS, 2008).

3.4.3 JMX

Expor os trabalhos internos de um aplicativo Java para gerenciamento é a parte crítica de pegar uma produção de aplicativo pronta. O *Java Management Extension (*JMX) do *Spring* facilita a exposição dos *beans* de aplicativos como *JMX MBeans*. Isso torna possível o monitoramento e a reconfiguração de um aplicativo em tempo de execução (WALLS, 2008).

3.5 MÓDULO *WEB*

O módulo *WEB* do *Spring* é projetado para ajudar no tratamento do gerenciamento de estado, do fluxo de trabalho e a validação, que são aspectos importantes, devido a natureza tolerante a falhas do protocolo HTTP (WALLS, 2008).

O MVC do *Spring* é baseado no padrão *Model-View-Controller* (MVC) para aplicações da *web*. A implementação de MVC do *Spring* fornece uma separação limpa entre o código do modelo do domínio e os formulários da *web* e

permite que todos os outros recursos do *Spring*, como validação, possam ser utilizados (CHHATPAR, 2010).

O *Spring* encaminha todas as requisições para o único *servlet front controller*. Um *front controller* é um padrão de aplicativo *web* comum no qual um único *servlet* delega a responsabilidade de uma requisição a componentes de outro aplicativo para executar a tarefa atual. No caso do MVC do *Spring*, o *DispatcherServlet* é o *servelt front controller*.

A Figura 11 exibe as etapas do ciclo de vida de uma requisição (WALLS, 2008).

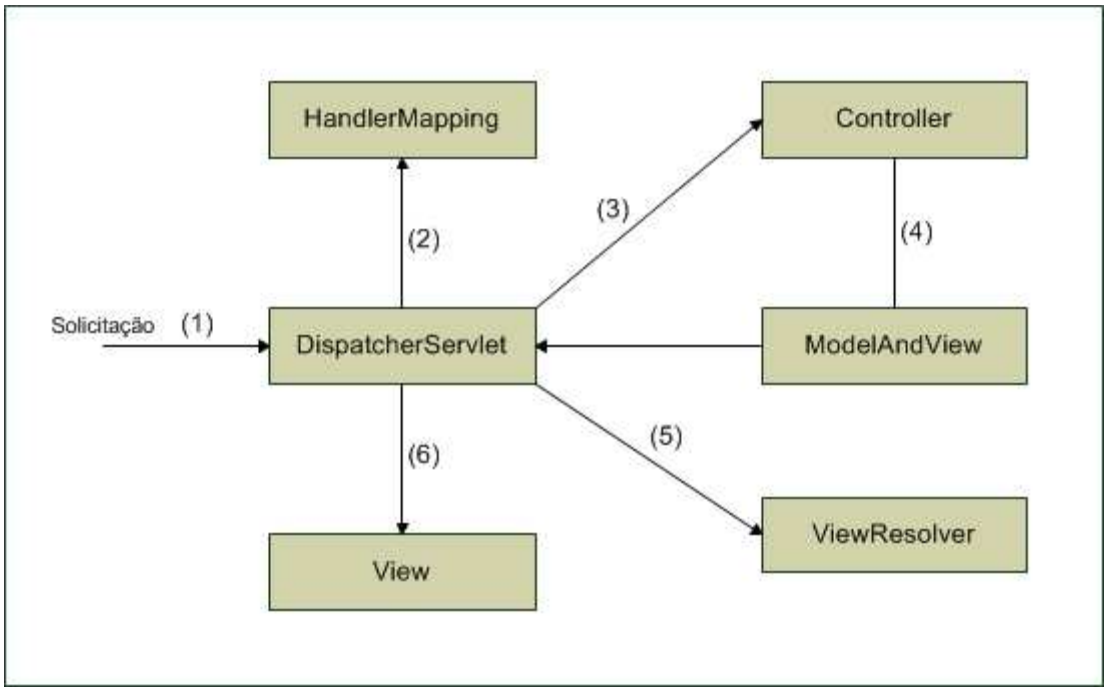

Figura 11 – Etapas do ciclo de vida de uma requisição no *Spring*.
Fonte: WALLS (2008).

Segundo SPRING MVC (2010), as etapas do ciclo de vida de uma requisição no *Spring* são:

1. *DispatcherServlet* – Recebe requisições do meio externo (do navegador, por exemplo) e comanda o fluxo de tarefas no *Spring* MVC;

2. *HandlerMapping* – Dada uma requisição em URL, este componente irá retornar o *Controller* que está associado a ela;

3. *Controller* – Realiza comunicação entre o MVC do *Spring* com a camada de negócio. Retorna um objeto *ModelAndView*;

4. *ModelAndView* – Armazena os dados retornados pela camada de negócio para serem exibidos. Além disso, contêm um nome lógico de uma determinada *View*;

5. *ViewResolver* – A partir do nome lógico contido no objeto *ModelAndView*, este componente determina a *View* que será exibida;

6. *View* – Contém informações de renderização para que o usuário possa ver o que solicitou.

O *Spring* suporta a integração com outros *framewoks* MVC como, por exemplo, o *Struts Framework*.

4 *ENTERPRISE* JAVA *BEANS* (EJB)

A especificação da Plataforma Java EE para os *Enterprise* Java *Beans* (EJB) define uma plataforma que é executada no lado servidor da aplicação, simplificando o desenvolvimento de aplicações escritas em Java, cuja arquitetura se baseia em componentes distribuídos (RICARDO, 2010).

EJB é uma plataforma para construção de aplicações corporativas de modo a serem portáveis, desacopladas e reutilizáveis. Isso permite projetos de aplicações escalonáveis, confiáveis, e seguras, sem a necessidade de realizar todo o trabalho de desenvolvimento a partir do zero (PANDA, 2009).

O EJB permite a construção de aplicações corporativas Java sem ter que reinventar serviços necessários para a construção de uma aplicação, como as transações, seguranças, persistência automatizada, entre outros (PANDA, 2009). Além disso, os EJBs já dispõem de uma infra-estrutura pré-escrita, acelerando o desenvolvimento das aplicações a serem executadas no lado do servidor, permitindo que os desenvolvedores se foquem na construção da lógica de negócio da aplicação corporativa (RICARDO, 2010).

A execução de aplicações distribuídas que utilizam o EJB depende da existência de um *container*. Um *container* para EJB é um *software* que provê o ambiente necessário para a utilização dos componentes básicos da arquitetura EJB. Portanto, um *container* EJB é um pedaço de *software* que implementa a especificação *Enterprise JavaBeans* que é disponibilizada pela Sun (SANTANA, 2010).

No EJB 3 um componente ou *bean* é um POJO com algumas características especiais. A idéia atrás de um *bean* é que ele encapsule com eficácia o comportamento da aplicação. Os usuários de um *bean* não precisam conhecer suas operações internas. Tudo o que precisam saber de um *bean* é o que deve ser informado e o que esperar de volta. No EJB há três tipos de beans (PANDA, 2009).

Os *beans* EJB vivem em um *container*. Juntos, os *beans* e o *container* podem ser vistos como um *framework* que fornece serviços valiosos para o desenvolvimento de aplicação corporativa (PANDA, 2009).

Os *beans* EJB facilitam o desenvolvimento uma vez que eles tratam automaticamente a segurança, persistência, transação, processamento assíncrono,

sistemas de integração, entre outros. Dessa forma o desenvolvedor não precisa se preocupar com a implementação desses serviços, a não ser que realmente deseje (BONI, 2010).

O *container* EJB fornece esses tipos de funcionalidade em comum como serviços funcionais e fáceis de serem usados para que seus *beans* possam usá-los em suas aplicações sem ignorá-las (PANDA, 2009).

A partir do EJB 3, as anotações de metadados (*Annotation*) simplificam o desenvolvimento e teste das aplicações permitindo que os desenvolvedores adicionem serviços aos componentes do EJB quando e onde forem necessários (PANDA, 2009). As *annotations* foram introduzidas no EJB para ser uma alternativa aos arquivos de configuração XML e, consequentemente, aliviar o desenvolvedor da sua complexidade inerente (FIUME, 2010c).

4.1 TIPOS DE *BEANS* DO EJB

Existem três tipos de *beans* no EJB:

- *Session Beans*.
- *Message Drive Beans*.
- *Entitys*.
- *Singleton Beans.*

Cada tipo de *bean* tem um propósito e pode utilizar um subconjunto específico de serviços EJB. O propósito real dos tipos de *beans* é combater a sobrecarga com seus serviços. A classificação do *bean* também ajuda a entender e organizar uma aplicação de uma maneira sensata; por exemplo, os tipos de *beans* ajudam a desenvolver aplicações baseadas em uma arquitetura multi-camada (PANDA, 2009).

Os *Session Bean* e os *Message Drive Beans* (MDBs) são usados para construir lógica de negócios, os *Entitys* são usados para modelar a persistência de uma aplicação (PANDA, 2009).

4.1.1 *Session Bean*

Sessions Bens são responsáveis por armazenar e executar as regras de negócios solicitados diretamente pelos clientes. Sendo que estes *beans* não possuem dados persistentes e seu ciclo de vida está vinculado à sessão criada pelo cliente, ou seja, quando a sessão cliente é fechada o *session bean* é destruído (SANTANA, 2010).

O servidor EJB é responsável por gerenciar o tempo de vida dos *beans*. Ou seja, os clientes não instanciam diretamente os *beans*, o *container* EJB faz isto automaticamente. O *container* EJB destrói os *session beans* no tempo apropriado (PEREZ, 2010).

Existem dois tipos de *Session Bean*:

- *Stateless Session Bean.*
- *Stateful Session Bean.*

4.1.1.1 *Stateless Session Bean*

Destinados a executar automaticamente operações individuais não mantendo o seu estado para cada invocação de um método. Cada cliente pode usar em qualquer momento uma instância de um *Stateless Session Bean*. O servidor mantém uma certa quantidade de instâncias preparadas para receber requisições de seus clientes (TAMIOSSO, 2010). Uma vez que o *bean* concluiu sua tarefa, ele está disponível para servir um cliente diferente sem manter os dados do cliente que o utilizou anteriormente (PEREZ, 2010).

4.1.1.2 *Stateful Session Beans*

Um *Stateful Session Bean* é o *bean* que é designado a servir processos de negócios que podem durar várias requisições ou transação dos clientes (PEREZ, 2010).

Os *session beans* mantém o estado para se trabalhar com cliente individual. O estado é alterado durante a invocação dos métodos, e este mesmo estado pode estar disponível para o mesmo cliente na sua próxima invocação (PEREZ, 2010). Portanto, uma instância não pode ser facilmente devolvida ao seu

pool enquanto uma sessão entre o *bean* e o cliente ainda está ativa (TAMIOSSO, 2010).

4.1.2 *Message Drive Beans*

O *Message Driven Bean* (MDB) é um *bean* que permite o processamento de mensagens assíncronas entre as aplicações Java EE. Esse tipo de *bean* é muito semelhante ao *Session Bean*, porém não pode ser acessado diretamente pelo cliente (SANTANA, 2010).

Os MDBs são consumidores de mensagens JMS. Assim, executando as mensagens quando recebem uma através do serviço de mensagens JMS. As mensagens são tratadas ao serem recebidas (PEREZ, 2010).

4.1.3 *Singleton Beans*

Um *singleton bean* é um tipo de *session bean* que garante que haverá apenas uma instância para uma aplicação na JVM (SAKS, 2010).

Por ser somente outro tipo de *session bean*, um *singleton bean* pode definir a mesma *client view*, local ou remoto, como *stateless* e *stateful bean*.Clientes acessam os *singletons beans* da mesma maneira que acessam *stateless* e *stateful bean*, isto é, através de uma referência EJB (SAKS, 2010).

4.1.4 *Entity* e Java *Persistence* API

Java *Persistence* API (JPA) é a especificação padrão para o gerenciamento de persistência e mapeamento objeto relacional (ORM), surgida na plataforma Java EE 5 (GALHARDO, 2010). O ORM é essencialmente o processo de mapeamento de dados contidos nos objetos Java das tabelas de banco de dados usando configuração (PANDA, 2009).

Na especificação do EJB 3 foi introduzida a utilização da JPA no intuito de substituir os *Entity Beans* (que foram descontinuados) e simplificar o desenvolvimento de aplicações Java EE e Java SE que utilizam persistência de dados. A JPA possui uma completa especificação para realizar mapeamento objeto

relacional, utilizando anotações. Também dá suporte a uma rica linguagem de consulta, semelhante à SQL, permitindo consultas estáticas ou dinâmicas (GALHARDO, 2010).

Um dos principais conceitos relacionados à JPA é o de *Entity*. Uma *Entity* corresponde a um objeto que pode ser gravado na base de dados a partir de um mecanismo de persistência. Os *Entitys* são componentes que representam em memória os dados que estão persistidos em uma base de dados relacional. Neste sentido, os *entitys* sobrevivem às quedas do *container* e paradas temporárias (PANDA, 2009). A classe que implementa o *entity* persistente é um POJO, que basicamente contém um construtor padrão sem argumentos e uma chave primária e também não precisa derivar de nenhuma interface ou classe, nem implementar qualquer método em especial (GALHARDO, 2010).

O *container* EJB é responsável pela sincronização e outras tarefas de manutenção de dados (FIUME, 2010c).

Como mencionado anteriormente, a API é uma especificação para a persistência, existe a necessidade de utilizar um provedor JPA. Por padrão, a implementação de referência é o *Oracle Toplink Essentials*, mas, existem outros provedores, como o *Hibernate*, BEA Kodo, entre outros.

4.2 INJEÇÃO DE DEPENDÊNCIA NO EJB 3

Injeção de Dependência é uma das principais novidades do EJB 3. Com a DI é possível obter o acesso a componentes apenas declarando suas dependências (FIUME, 2010b).

A DI é configurada no cliente, através de anotações. A responsabilidade pela procura, inicialização, instanciação e retorno de referências é do *container* EJB (FIUME, 2010a).

Por exemplo, para acessar os serviços de um componente EJB utilizando a DI, teremos algo semelhando ao mostrado na listagem a baixo:

```
...
@EJB
private ProdutoDAO produtoDAO;
....
```

Isso permite que um cliente simplesmente declare suas dependências para o *container* EJB injetar com transparência o EJB ao cliente, liberando o cliente da responsabilidade de procurar, inicializar e instanciar o componente (FIUME, 2010a).

A injeção de dependência do EJB não removeu os JNDI *lookups* utilizado nas versões anteriores ainda é possível utilizar o JNDI quando for inevitável (PANDA, 2009).

5 APLICAÇÃO DE ESTUDO DE CASO

Para que seja possível realizar análises do *Spring Framework* e a especificação EJB 3, apontando as vantagem e desvantagem de cada um, foi desenvolvida uma aplicação de estudo de caso com objetivo de servir de base para o levantamento das informações necessárias.

A aplicação de estudo de caso tem por objetivo realizar o gerenciamento dos pedidos e dos produtos de uma determinada loja, assim sendo necessária a realização de todas as operações básicas (CRUD - *Create*, *Retrieve*, *Update* e *Delete*).

A base de dados utilizada no desenvolvimento da aplicação de estudo de caso foi o MySQL.

A Figura 12 ilustra o diagrama de classe do modelo de negócio da aplicação de estudo de caso.

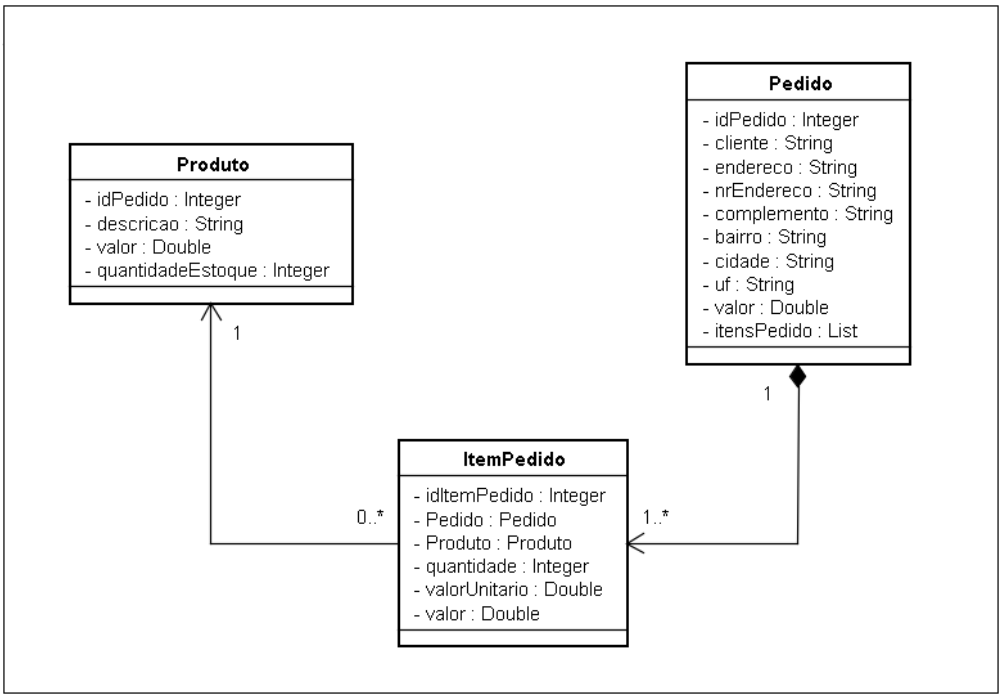

Figura 12 - Diagrama de Classes da aplicação de estudo de caso.

A Figura 13 ilustra o modelo entidade-relacionamento (MER) da aplicação de estudo de caso.

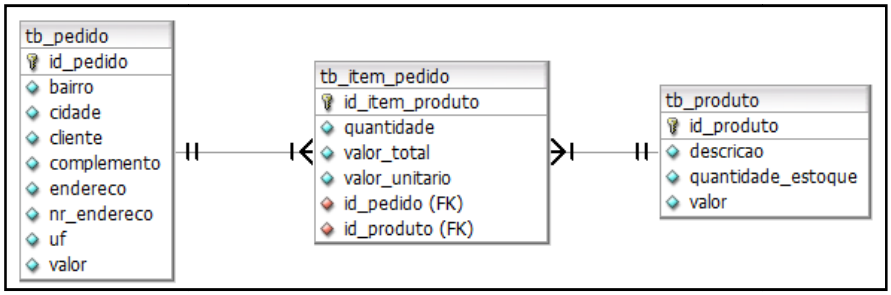

Figura 13 – MER da aplicação de estudo de caso.

Com o objetivo de explorar os recursos de cada tecnologia, existiu a necessidade do desenvolvimento de duas aplicações. Assim, realizando a avaliação da utilização do *Spring Framework* e da especificação EJB 3 separadamente.

O modelo, as regras de negócio e a interface com o usuário são idênticos nas duas aplicações, já a estrutura de cada aplicação leva em consideração as particularidades de cada tecnologia para o seu desenvolvimento, assim as estruturas das aplicações se diferenciam em alguns aspectos.

As aplicações serão desenvolvidas para a ambiente *web*. Portanto, serão divididas em Cliente, responsável pela interação do usuário com o sistema através de um navegador qualquer, e o Servidor, responsável por processar todas as requisições dos usuários.

A Figura 14 exibe a tela de cadastro de produto da aplicação.

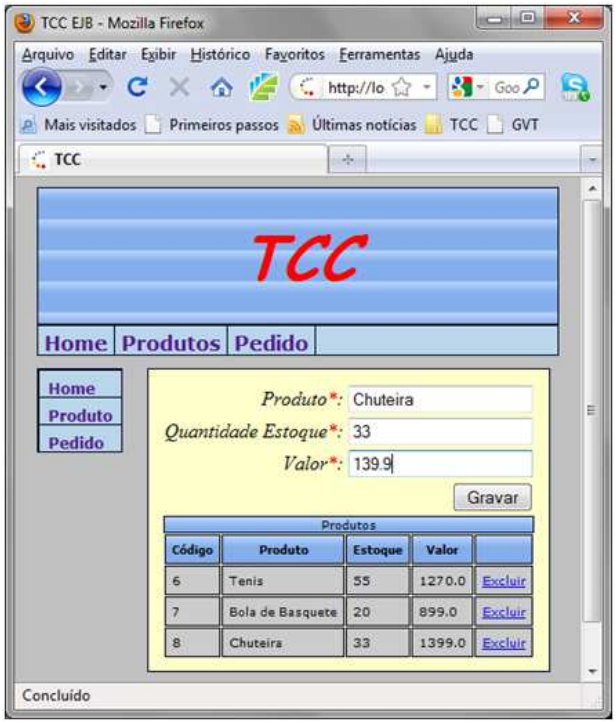

Figura 14 - Cadastro de produtos.

As Figuras 15 e 16 exibem as telas de cadastro de pedido da aplicação.

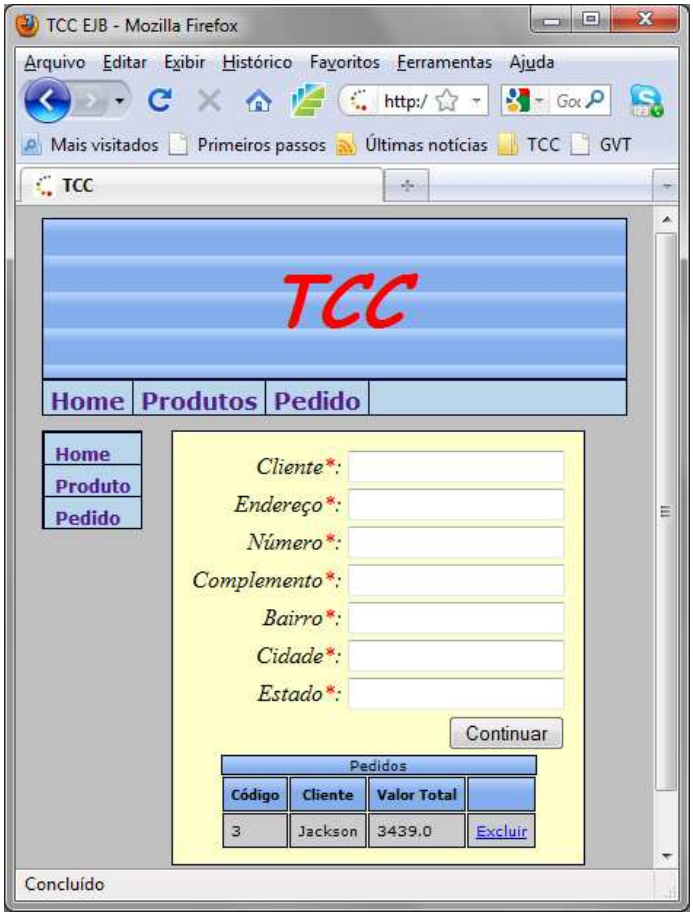

Figura 15 - Cadastro de pedido da aplicação.

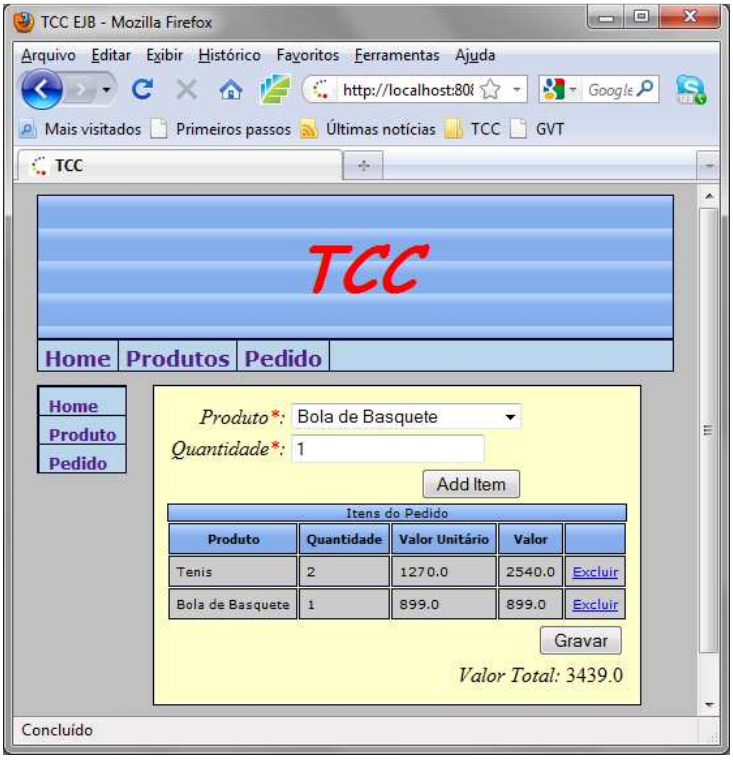

Figura 16 - Cadastro dos itens de um pedido.

O desenvolvimento foi dividido em duas partes, o desenvolvimento da aplicação utilizando o EJB 3 e o desenvolvimento da aplicação utilizando o *Spring Framework,* também foi utilizado alguns *frameworks* para o desenvolvimento das aplicações.

O Diagrama de pacote das duas aplicações pode ser visualizada na Figura 17.

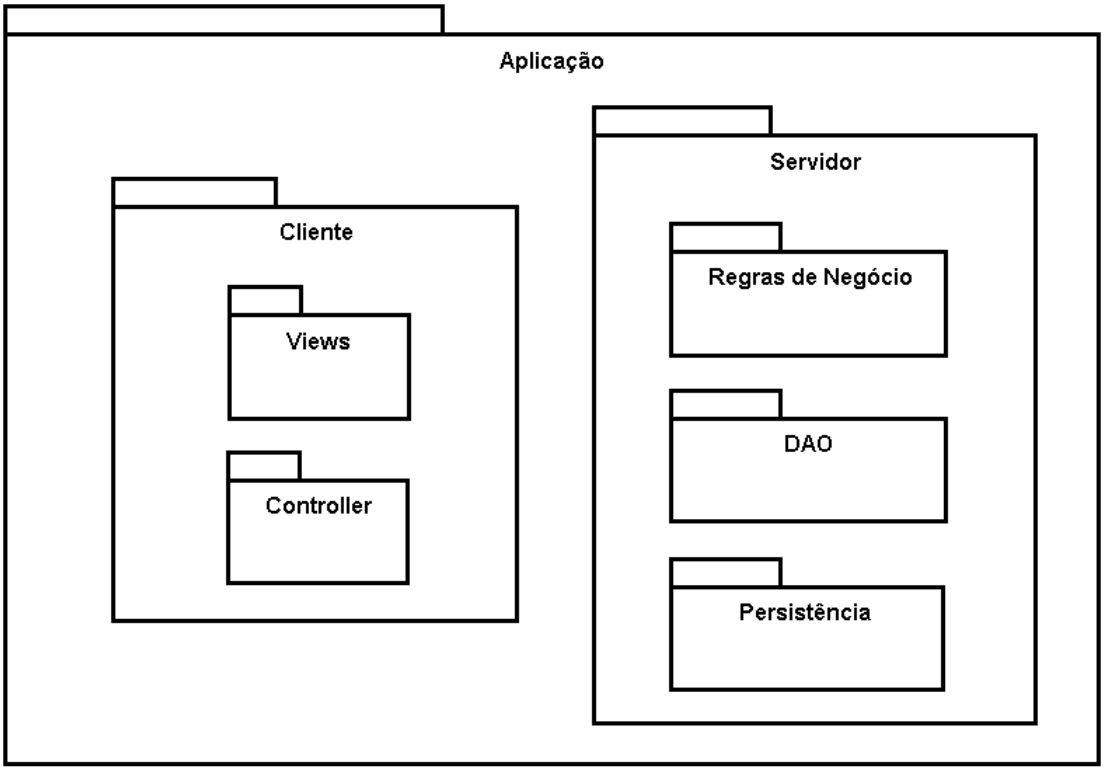

Figura 17 - Diagrama de pacote da aplicação de estudo de caso.

5.1 DESENVOLVIMENTO COM A ESPECIFICAÇÃO EJB 3

A especificação EJB possui várias implementações para que possa ser utilizada pelos desenvolvedores, ou seja, é preciso escolher uma dentre as várias implementações que existe no mercado. Por ser uma especificação, o desenvolvedor não fica “amarrado” a implementação escolhida. Portanto, é possível trocar a implementação utilizada sem precisar adequar a aplicação com a implementação do EJB.

Para o desenvolvimento da aplicação de estudo de caso, foi utilizado a implementação do EJB fornecida pela JBoss *Application Server* na versão 5.1. O

JBoss é um servidor de aplicação Java EE de código aberto, responsável por fornecer toda uma estrutura para o desenvolvimento de aplicações corporativas.

A plataforma EJB é responsável pelo lado servidor da aplicação, por esse motivo, o *framework Struts* 2 foi utilizado para realizar o desenvolvimento da parte *web* da aplicação. O *Struts* 2 é um *framework* de desenvolvimento de aplicações *web* e provê uma implementação do modelo do padrão MVC-2 (*Model*, *View and Controller* – padrão de projeto que define que as camadas de regra de negócio, visualização e controle estejam bem separadas na aplicação).

Para a persistência das informações no banco de dados foi utilizado implementação da JPA disponibilizada pelo *Hibernate Framework*. O *Hibernate* é um *framework* que tem por responsabilidade mapear os objetos (*Beans*) da aplicação para a persistência dos dados em um banco de dados relacional.

Para desenvolvimento da aplicação, existiu a necessidade de dividi-la em dois projetos, um correspondente ao servidor (Figura 18) e o outro a interface *web* (Figura 19) da aplicação.

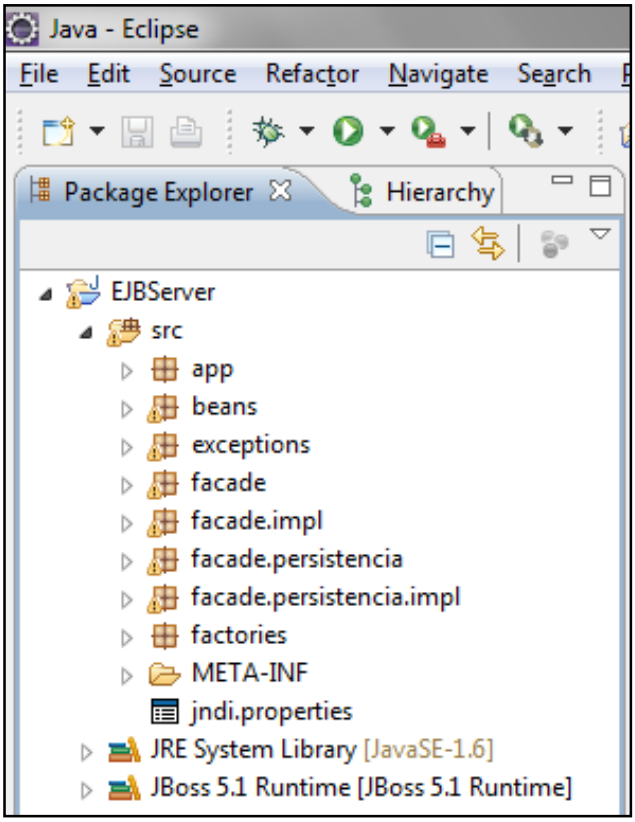

Figura 18 - Estrutura do projeto servidor da aplicação utilizando o EJB.

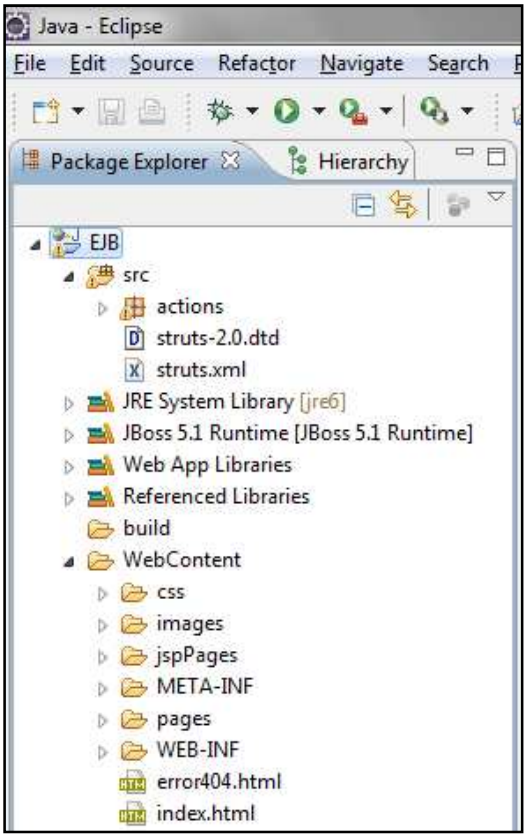

Figura 19 - Estrutura do projeto web aplicação utilizando o EJB.

5.1.1 Vantagens

Por disponibilizar a infra-estrutura, a maior parte do desenvolvimento utilizando o EJB fica focado nas regras de negócio da aplicação, agilizando o seu desenvolvimento.

O desenvolvimento fica focado na utilização dos serviços disponibilizados pelos EJBs uma vez que o *container* do EJB se responsabiliza por todo o gerenciamento da parte servidora da aplicação, consequentemente, o *container* é responsável pelo gerenciamento do ciclo de vida dos EJBs e por disponibilizar as interfaces para que os EJBs possam ser utilizados pela parte cliente da aplicação. Portanto, a instanciação, disponibilização e finalização dos EJBs são de responsabilidade do *container*.

Com a utilização das anotações disponibilizadas, o *container* se responsabiliza por injetar as dependências entre os objetos, ou seja, o *container* verifica as dependências de um determinado EJB e as injeta, assim, esse trabalho fica totalmente transparente para o desenvolvedor.

O *container* EJB realiza todo o controle de persistência da aplicação, operações como a conexão com o bando de dados, as transações, o tratamento de exceções, a garantia da atomicidade das informações, entre outros, são operações gerenciadas totalmente pelo *container*, assim, simplificando a utilização dos serviços de persistência.

5.1.2 Desvantagens

A Injeção de Dependência do EJB é limitada a utilização de anotações, não sendo possível se beneficiar de outras formas de DI. Também, só é possível realizar a Injeção de Dependência somente entre os EJBs, ou seja, os objetos que não forem EJBs, consequentemente, não podem usufruir dos benefícios da DI.

O aplicativo depende do *container* EJB para o seu funcionamento, o que pode acarretar em muito desperdício de recursos dependendo do tamanho da aplicação. O *container* EJB disponibiliza uma quantidade muito grande de recursos, e a aplicação pode acabar utilizando somente alguns de seus recursos disponibilizados, proporcionando um custo muito elevado para a utilização de poucos dos seus beneficio, assim, se tornando um *container* muito pesado para o aplicativo, levando em consideração que ao iniciar o *container* EJB todos os seus recursos são carregados e disponibilizados.

5.2 DESENVOLVIMENTO COM O *SPRING FRAMWORK*

O *Spring Framework* disponibiliza vários módulos para o desenvolvimento de soluções corporativas não se restringindo somente ao lado servidor a aplicação, mas também ao lado cliente. Portanto, o *Spring* realizar o controle da aplicação de estudo de caso por completo.

Para a disponibilização da parte cliente da aplicação, foi utilizado o *TomCat*. O *TomCat* é um servidor *web Java*, considerado como a implementação de referência para as tecnologias *Java Servlet* e *JavaServer Pajes* (JSP).

Com o objetivo de avaliar o *Spring*, a aplicação de estudo de caso foi desenvolvida para aproveitar o máximo dos recursos disponibilizados pelo mesmo.

O *Spring* permite a integração com diversos outros *frameworks*. Com o objetivo de verificar o funcionamento dessas integrações, para a persistência das informações no banco de dados foi utilizada a implementação da JPA disponibilizada pela *Oracle Toplink Essentials* no lugar do módulo de persistência disponibilizada pelo *Spring*.

A Figura 20 exibe a estrutura do projeto da aplicação de estudo de caso.

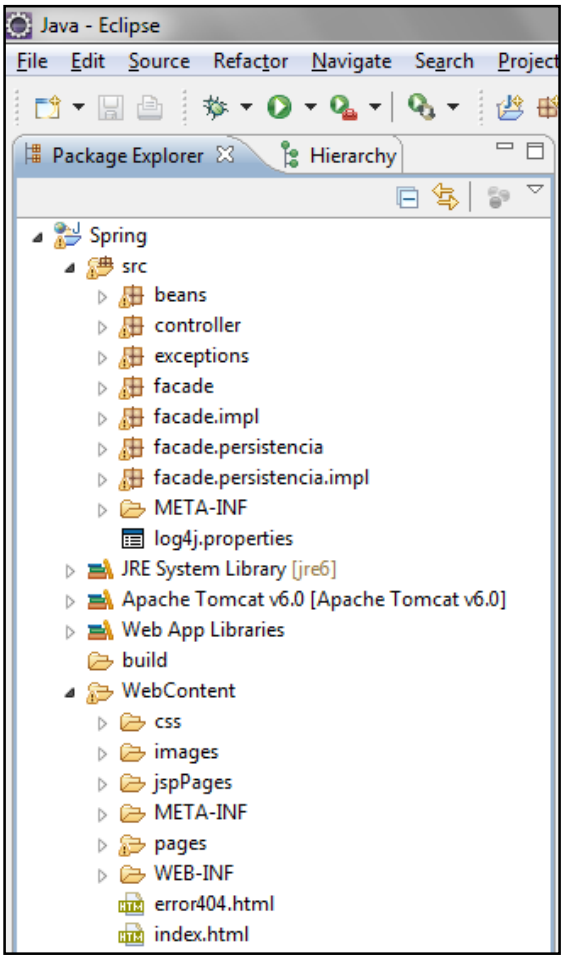

Figura 20 - Estrutura do projeto da aplicação utilizando o *Spring Framework*.

5.2.1 Vantagens

Por ser um *framework* baseado na injeção de dependência, o *Spring* fornece varias o formas de usar a DI de uma forma centralizada, deixando o código da aplicação mais limpo, não é preciso se preocupar em como as dependências são inseridas nos objetos, basta realizar as configurações em arquivos XML, assim, agilizando o desenvolvimento.

O arquivo XML de configuração e utilização do *Spring* pode ser subdividido para separar os módulos da aplicação. Por exemplo, separar a configuração da persistência da configuração das regras de negócio.

O *Spring* não necessita de um servidor de aplicação para a sua utilização e é separado por módulos, sendo possível adicionar e utilizar somente os módulos necessários para a aplicação. Portanto, o *container* do *Spring* carrega somente os módulos utilizados, consequentemente, deixando a aplicação muito mais leve e sem desperdício de recurso, já que os módulos não utilizados não são adicionados a aplicação.

Como o *Spring* realiza o controle de toda a aplicação é possível utilizar os benefícios da injeção de dependência em qualquer classe de qualquer parte da aplicação.

A utilização da injeção de dependência é totalmente centralizada, o que facilita e agiliza o seu desenvolvimento e para realizar a sua manutenção não precisando percorrer vários pontos da aplicação para realizar alterações entre outras coisas.

5.2.2 Desvantagem

A principal desvantagem do *Spring* é a grande quantidade de configurações que devem ser feitas em arquivos XML. A utilização de arquivos XML para a configuração da utilização do *Spring* possui uma margem grande a erros de digitação e também diminui a produtividade para o desenvolvimento do sistema.

Conforme a aplicação evolui, os arquivos XML vão se tornando cada vez maior, assim, aumentando consideravelmente a complexidade da manutenção dos arquivos de configuração e, consequentemente, aumentando a complexidade do gerenciamento da aplicação.

6 CONSIDERAÇÕES FINAIS

6.1 CONCLUSÃO

O EJB é utilizado no desenvolvimento da parte servidora da aplicação, disponibilizando toda uma infra-estrutura e vários serviços prontos promovendo a produtividade e focando o desenvolvimento nas regras de negócio da aplicação. O *container* do EJB se responsabiliza por gerenciar a parte servidora da aplicação e, através das anotações, verifica os objetos EJB realizando a injeção das dependências entre os objetos, mas a injeção de dependência se limita aos objetos do *container*. O *container* EJB carrega todos os seus recursos o que pode acarretar em um desperdício uma vez que a aplicação pode não utilizá-los.

O *Spring* é utilizado no desenvolvimento tanto na parte servidora quanto na parte cliente disponibilizando a utilização da injeção de dependência em qualquer parte da aplicação. Como o *Spring* é separado em módulos, o *container* carrega somente os módulos utilizados na aplicação evitando o desperdício de recursos. O *container* do *Spring* é responsável por realizar o gerenciamento dos objetos sendo configurado através de arquivos XML. Porém, os arquivos XML podem ficar com uma quantidade elevada de informação, acarretando em uma maior complexidade em seu gerenciamento.

Ambas as tecnologias estudadas apresentaram robustez suficiente para o desenvolvimento de aplicações corporativas, porém, é preciso avaliar a aplicação que será desenvolvida para saber escolher qual das duas tecnologias deverá ser utilizada para maximizar a utilização de suas vantagens.

6.2 TRABALHOS FUTUROS

Considerando que o *Spring* é separado em módulos e permite sua utilização em qualquer aplicativo, sugiro a realização de um estudo para verificar os benefícios de sua utilização no desenvolvimento de aplicativos para o ambiente *desktop*.

REFERÊNCIAS BIBLIOGRÁFICAS

ARAGÃO JUNIOR, Maurício Linhares de. **Report Center e Phrame UI: Desenvolvimento da ferramenta de relatórios Report Center e do framework de interface Phrame UI**. 2007. 77. Relatório Final de Estágio – Centro Federal de Educação Tecnológica da Paraíba. João Pessoa, 2007

BONI, Marcel. **Introdução SAP NetWeaver e EJB**. Disponível em <http://javafree.uol.com.br/artigo/864621/Introducao-SAP-NetWeaver-e-EJB.html>. Acessado em 28 mai. 2010.

CHHATPAR, Arun. **Apache Geronimo e o Spring Framework, Parte 1: Metodologia de Desenvolvimento**. Disponível em <http://www.ibm.com/developerworks/br/library/os-ag-springframe1/section6.html>. Acessado em 20 mai. 2010

DBD, PUC-RIO. **Frameworks: Conceitos Gerais**. Disponível em <http://www2.dbd.puc-rio.br/pergamum/tesesabertas/0410823_06_cap_02.pdf>. Acessado em 23 ago. 2010.

FIUME, Rafael. **O Papel da Dependency Injection no EJB 3.0**. Disponível em <http://rfiume.blogspot.com/2007/04/o-papel-da-dependency-injection-no-ejb.html>. Acessado em: 28 mai. 2010a.

FIUME, Rafael. **Service Locator Não Morreu**. Disponível em <http://rfiume.blogspot.com/2007/04/service-locator-no-morreu.html>. Acessado em: 28 mai. 2010b.

FIUME, Rafael. **Sobre Annotations e Deployment Descriptors no EJB 3.0**. Disponível em <http://rfiume.blogspot.com/2007/03/sobre-annotations-e-deployment.html>. Acessado em: 28 mai. 2010c.

FOWLER, Martin. **Inversion Of Control: Containers de Inversão de Controle e o padrão Dependency Injection**. Disponível em <http://javafree.uol.com.br/artigo/871453/Inversion-Of-Control-Containers-de-Inversao-de-Controle-e-o-padrao-Dependency-Injection.html>. Acessado em 26 jan 2010.

GALHARDO, Raphaela. **Introduzindo Java Persistence API**. Disponível em <http://www.jspbrasil.com.br/mostrar/76>. Acessado em: 29 mai. 2010.

JAVA FREE. **Spring**. Disponível em <http://www.javafree.org/wiki/Spring>. Acessado em: 20 dez. 2009.

JOHNSON, Rod; HOELLER, Juergen; ARENDSEN, Alef. **Spring Framework: Reference Documentation**. Disponível em <http://static.springsource.org/spring/docs/2.5.x/spring-reference.pdf>. Acessado em: 23 mai. 2010.

LEMOS, Alberto J. **Spring: Um suíte de novas opções para Java EE**. Disponível em <http://www.slideshare.net/drspockbr/tdcspringframework-presentation>. Acessado em: 20 dez. 2009.

PACHECO, Diego. **Spring Framework (2.0): Framework para Desenvolvimento de Aplicações em Java**. Disponível em <http://www.scribd.com/doc/18517573/Spring-Framework-20-Diego-Pacheco>. Acessado em: 17 mai. 2010.

PANDA, Debu; RAHMAN, Reza; LANE, Derek. **EJB 3 em Ação**. 3. ed. Rio de Janeiro: Alta Books, 2009.

PEREZ, Anderson. **Sistemas Distribuídos Baseados em Objetos EJB**. Disponível em <http://www.univasf.edu.br/~anderson.perez/ensino/sd_i/aulas/Sistemas_Distribuidos_Baseados_Objetos_EJB.pdf>. Acessado em: 28 mai. 2010.

RICARDO, Paulo. **EJB**. Disponível em <http://www.pauloricardo.eng.br/sa/EJB.pdf>. Acessado em: 29 mai. 2010.

SAKS, Ken. **A Sampling of EJB 3.1**. Disponível em <http://blogs.oracle.com/enterprisetechtips/entry/a_sampling_of_ejb_3>. Acessado em 18 out. 2010.

SANTANA, Fabricio; SANTANA, Gabriel; RIBEIRO, Guilherme; FERREIRA, Jocimar. **Persistência com EJB 3.0: Java Persistence API (JPA)**. Disponível em <http://im.ufba.br/pub/MATA60/WebHome/persejb.pdf>. Acessado em: 28 mai. 2010

SPRING MVC. **SYSREQ: Spring MVC**. Disponível em <http://www.scribd.com/doc/23830838/SpringMVC>. Acessado em: 25 mai. 2010.

TAMIOSSO, Daniel. **EJB: Stateless e Stateful Session Beans**. Disponível em <http://danieltamiosso.com/ejb-stateless-e-stateful-session-beans>. Acessado em: 28 mai. 2010.

WALLS, Craig; BREIDENBACH, Ryan. **Spring em Ação**. 2. ed. Rio de Janeiro: Alta Books, 2008.