

Centro Paula Souza
Faculdade de Tecnologia de Taquaritinga
Graduação em Processamento de Dados

Monografia
Spring Framework

AUTOR: Alexandre Wiggert De Nobile Ferreira

ORIENTADOR: Marcus Rogério de Oliveira

Taquaritinga
2011

ALEXANDRE WIGGERT DE NOBILE FERREIRA

SPRING FRAMEWORK

Monografia apresentada a Faculdade de Tecnologia de Taquaritinga, como parte dos requisitos para a obtenção do título de Tecnólogo em Processamento de Dados.

Orientador: Marcus Rogério de Oliveira

Taquaritinga
2011

Dedico,
A todos que me suportaram nessa jornada, especialmente aos meus mentores na FATEC,
UNESP e no Gourmex.

Ferreira, A. W. D. N. **Spring Framework**. Trabalho de Graduação Monografia. Centro Estadual de Educação Tecnológica “Paula Souza”. Faculdade de Tecnologia de Taquaritinga. [59]. 2011

RESUMO

Quando se desenvolve *softwares* empresarias de alta disponibilidade e confiabilidade, *frameworks* são utilizados para facilitar o trabalho e aumentar a produtividade dos programadores. Porém, muitos *frameworks* forçam o desenvolvedor a criar códigos ou até mesmo um módulo inteiro seguindo convenções ou arquiteturas pouco flexíveis que nem seriam necessários para o projeto e que acabam gerando um forte acoplamento entre os módulos do sistema. O *Spring* é um *framework* Java que não segue esse paradigma, não depende de arquitetura (pode ser utilizando tanto em softwares que rodam localmente quanto na *web* ou até mesmo em dispositivos móveis) e que tem como objetivo simplificar o desenvolvimento de *softwares* J2EE através de injeção de dependência, controle transacional declarativo, tudo isso utilizando POJOs de forma simples, rápida e produtiva, gerando assim um produto confiável e previsível. Desse modo os programadores podem dedicar maior parte do tempo desenvolvendo as lógicas de negócio do que reinventando a roda. Para aumentar ainda mais o poder do *Spring*, existe a *Spring Expression Language* que é capaz de realizar operações, manipular *collections* e realizar o *wiring* de *beans* dinamicamente.

Palavras-chave: Framework. Java. Spring. J2EE. SpEL.

Ferreira, A. W. D. N. **Spring Framework**. Trabalho de Graduação Monografia. Centro Estadual de Educação Tecnológica “Paula Souza”. Faculdade de Tecnologia de Taquaritinga. [59]. 2011

ABSTRACT

When it's about developing high availability and reliable enterprise class softwares, frameworks are used to ease the job and raise the programmers productivity. Although, many frameworks forces the developers to create codes or even an entire module following certain low flexibility conventions or architectures that wouldn't be even necessary for the project, creating an strong coupling between the system modules. Spring is a Java framework which does not follow this paradigm, does not require a specific architecture (it can be used for either desktop, web or even mobile applications) and has one goal: to simplify J2EE software development by dependency injection, declarative transactional control using POJOs in a very simple, fast and productive way, therefore, generating a reliable and predictable product. This way, programmers can dedicate more time to develop business logic instead of reinventing the wheel. To increase Spring's power, there is the Spring Expression Language which is capable of make operations, manipulate collections and do the bean's wiring dynamically.

Keywords: Framework. Java. Spring. J2EE. SpEL.

SUMÁRIO

<u>1</u>	<u>INTRODUÇÃO</u>	<u>8</u>
<u>2</u>	<u>J2EE E FRAMEWORKS</u>	<u>9</u>
2.1	Plataforma Java 2 Enterprise Edition (J2EE)	9
2.2	J2EE Container	10
2.3	Frameworks	11
<u>3</u>	<u>SPRING FRAMEWORK</u>	<u>13</u>
3.1	Enterprise Java Beans	13
3.2	O Spring	14
3.3	Injeção De Dependência e Inversão De Controle	15
3.4	Redução De Código Clichê (Boilerplate)	16
3.5	Ciclo de vida dos beans	19
3.6	Application Context	22
<u>4</u>	<u>INJEÇÃO DE BEANS COM O SPRING</u>	<u>23</u>
4.1	Wiring de Beans	23
4.2	Declarando Beans	24
4.3	Injeção através de construtores	25
4.4	Injeção em propriedades do bean	27
4.5	Wiring de propriedades utilizando namespace p	28
4.6	Wiring de collection	28
4.7	Wiring de propriedades com valor nulo	31
4.8	Escopos de beans	32
4.9	Minimizando a configuração XML	32
<u>5</u>	<u>SPRING EXPRESSION LANGUAGE</u>	<u>35</u>
5.1	Fundamentos da SpEL	35
5.2	Referenciando propriedades e métodos	36
5.3	Tipos na SpEL	38
5.4	Realizando operações com SpEL	38
5.5	Utilizando expressões regulares na SpEL	40
5.6	Utilizando collections com SpEL	41
<u>6</u>	<u>CONTROLE DE TRANSACÇÃO COM SPRING</u>	<u>43</u>
6.1	Controle de transações do Spring	43
6.2	Transações JDBC	45
6.3	Transações Hibernate	45
6.4	Transações JPA	46
6.5	Transações JTA	46

6.6 Atributos da transação	46
6.7 Propagação	47
6.8 Níveis de Isolamento	47
6.9 Transações Read-Only	49
6.10 Timeout da transação	49
6.11 Regras de rollback	49
6.12 Controle de transação através de anotações	49
<u>7 EXEMPLO DE IMPLEMENTAÇÃO DO SPRING</u>	<u>51</u>
7.1 Arquivos XML de configuração	51
7.2 A camada de apresentação	57
7.3 As ActionBeans	58
7.4 Context Loader Listener	59
<u>8 CONCLUSÃO</u>	<u>61</u>
<u>REFERÊNCIAS</u>	<u>62</u>

1 INTRODUÇÃO

A utilização de *frameworks* para o desenvolvimento de *softwares* J2EE é constante e comum hoje em dia no mercado, pois tornam o trabalho do programador mais produtivo, fácil sem que tenham que reinventar a roda.

Porém não são todos os *frameworks* que permitem que o programador defina sua própria arquitetura ou necessidade, os forçando a implementar certa classe ou interface, gerando trabalho que nem seria necessário naquele determinado módulo ou projeto. Frameworks também tendem a ser dependentes de certa arquitetura ou container, como é o caso da especificação *Enterprise Java Beans*, gerando assim um forte acoplamento entre o próprio sistema e o ambiente ou entre seus próprios módulos.

O *framework* J2EE *Spring* foge desses problemas e paradigmas impostos por outros frameworks com a missão de transformar o desenvolvimento de *softwares* J2EE simples, sem deixar o programador de mãos atadas ou forçá-lo a seguir determinado padrão ou arquitetura.

Como será mostrado, qualquer programa pode tirar proveito do *Spring*, seja ele para dispositivos móveis, *web* ou *desktop*, dando uma visão abrangente do que o *Spring* pode fazer pela cada de serviço do *software*, que é comum em todas as aplicações, sem depender das outras camadas como de banco ou de apresentação. Através da injeção de dependência, *Spring Expression Language*, controle transacional declarativo, e utilizando POJOs, que são objetos Java simples, o *Spring* consegue criar uma alternativa bem interessante ao EJB e outros *frameworks* existentes no mercado.

Este trabalho foi realizado com base em pesquisas bibliográficas, guias de referência e documentações técnicas das tecnologias abordadas.

2 J2EE E FRAMEWORKS

A cada dia, uma enorme quantidade de dados trafega pelas redes e aplicações interagindo externa e internamente com várias organizações. Softwares de alta disponibilidade e cada vez mais robustos são necessários para garantir a integridade e qualidade desses dados para as organizações.

A plataforma J2EE (Java 2 Enterprise Edition) e os frameworks, de maneira geral, buscam aumentar a produtividade e qualidade desses softwares para que os seus desenvolvedores gastem mais (e melhor) o seu tempo se preocupando nas regras de negócio do que em reinventar a roda, fornecendo um conjunto de recursos já implementados e prontos para usar nas suas aplicações.

2.1 Plataforma Java 2 Enterprise Edition (J2EE)

A *Sun Microsystems* (adquirida pela *Oracle* em abril de 2009) anunciou a plataforma J2EE em meados de 1999, sendo lançada oficialmente ao fim do mesmo ano.

Visando desenvolver aplicações de porte empresarial, foi concebida para ser usada em conjunto com o JDK (*Java Development Kit*) para os tipos de aplicação que são de missão crítica para a empresa e que forneça conectividade *Web*.

Desenvolver uma aplicação empresarial (corporativa) é uma tarefa muito complexa e requer um conhecimento sobre diversas áreas. Por exemplo, uma aplicação empresarial requer que você se familiarize com os problemas de comunicação entre os processos da empresa, questões de segurança, questões de acesso e consultas específicas em um banco de dados, etc. JEE oferece de forma pronta e transparente suporte para essas questões. Como resultado, o desenvolvedor é capaz de focalizar na implementação da lógica de negócio da aplicação. (AHMED; UMRYSH, 2001, p. 15).

De acordo com Ahmed e Umrysh(2001, p.14), um dos maiores benefícios da J2EE é o seu suporte para componentização com as seguintes vantagens:

- Alta produtividade: componentes já testados e consolidados são colocados em conjunto para que os desenvolvedores não precisem criar soluções do zero.
- Desenvolvimento rápido: componentes existentes podem ser colocados em conjunto para criar novas aplicações.
- Maior qualidade: os desenvolvedores podem testar componentes específicos sem interferir nas partes restantes da aplicação.

- Facilidade de manutenção: atualizações, por exemplo, podem ser realizadas em componentes individuais sem afetar os outros, sendo assim mais fácil e mais efetivo.

2.2 J2EE Container

O *container* é o software que irá ser executado em um servidor, fornecendo o ambiente de execução e gerenciando o ciclo de vida dos componentes (*Servlets* e *JSPs*, por exemplo).

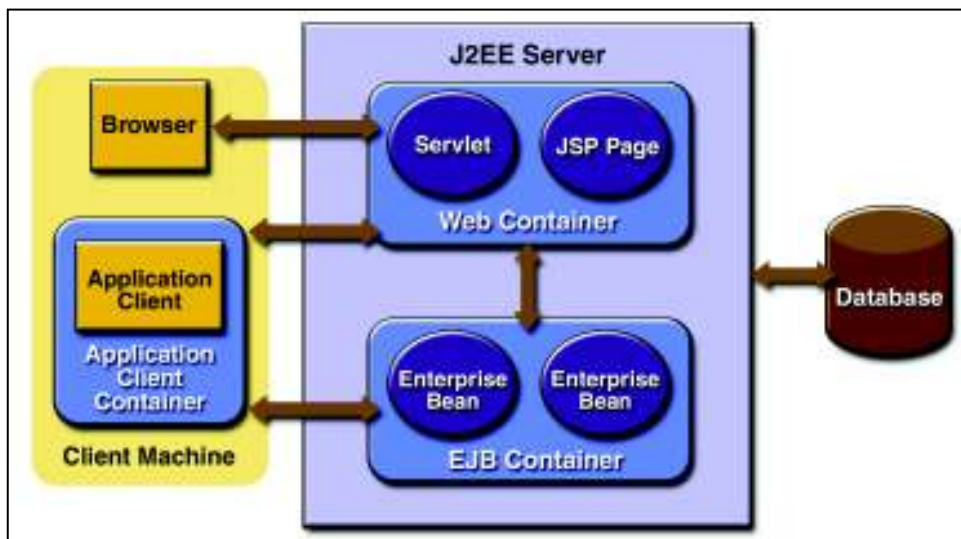


Figura 1 – Container J2EE

Fonte: (<http://download.oracle.com/javaee/1.3/tutorial/doc/Overview4.html>)

Conforme demonstrado na ilustração acima, toda requisição que chega ao servidor *Web*, será encaminhada para o *container* que por sua vez encaminha para o *servlet* designado para atender a requisição, que poderá acessar alguma API (*Application Programming Interface*) para ler ou gravar dados em um banco por exemplo. Após isso, irá fornecer uma resposta a essa requisição, podendo ser uma JSP (*Java Server Page*), por exemplo, que será exibida como HTML (*Hypertext Markup Language*) para o cliente. Todo container, de acordo com a especificação J2EE deve suportar o protocolo HTTP (*Hypertext Transfer Protocol*). Alguns exemplos mais famosos de *containers* são: *Apache Tomcat*, *WebSphere*, *JBoss*, *GlassFish*, entre outros.

2.3 Frameworks

De acordo com Ahmed e Umrysh(2001, p.73), um *framework* pode referenciar qualquer parte de um *software* desenvolvido e testado que é reutilizado em vários outros projetos de desenvolvimento de *software*.

Frameworks visam aumentar a produtividade do programador agrupando e reutilizando seus elementos fornecendo assim um modelo de arquitetura no qual o *software* deve ser adaptado pelo programador, incluindo apenas os requisitos e funcionalidades específicas da sua aplicação sem se preocupar com aspectos de baixo nível do sistema, já implementados e prontos para utilizar pelo *framework*.

Ainda de acordo com os mesmos autores, devem possuir as seguintes características:

- Simples de entender: o desenvolvedor deve aprender e começar a usar o *framework* rapidamente e de forma eficaz.
- Fornecer documentação adequada: para que se possa consultar e tirar dúvidas de maneira objetiva e utilizá-lo da maneira correta.
- Fornecer mecanismos de extensão: para que se possa adaptá-lo ou integrá-lo a novas tecnologias, necessidades ou até mesmo outros *frameworks*.

A maioria dos *frameworks* utilizam o padrão de arquitetura de *software* MVC (*Model View Controller*) que separa as regras (ou lógica) de negócio da camada de apresentação (interface com o usuário) , permitindo assim maior flexibilidade, adaptabilidade e manutenibilidade do *software* podendo ser utilizados em conjunto (mais de um *framework*) de acordo com a necessidade do projeto.

Existem vários *frameworks* Java disponíveis no mercado e várias utilizações para os mesmos. Para citar os principais, por exemplo, existem os *frameworks web* que facilitam a criação de *sites* e *softwares* que são acessados por um navegador, *frameworks* de banco de dados para realizar o mapeamento objeto-relacional e facilitar a configuração, controlar transações e agilizar implementações e utilização de operações CRUD (*Create, Retrieve, Update e Delete* ou Criar, Recuperar, Atualizar e Apagar).

Para citar, alguns dos *frameworks* Java mais famosos: *Swing, Struts, JavaServerFaces, Stripes, Spring, Hibernate, MyBatis, Google Web Toolkit*.

Porém, também existem vários outros frameworks que se baseiam em outras plataformas e tecnologias, como por exemplo o .NET (lê-se “*dot net*”), *Zend*, *CodeIgniter*, entre muitos outros. Algumas empresas desenvolvem seus próprios *frameworks* de acordo com sua necessidade, mas a não ser que seja algo muito específico, muitas vezes, esse processo longo e demorado acaba não compensando.

3 SPRING FRAMEWORK

De acordo com Walls (2011), o *framework Spring* tem ampla utilização, mas as principais são injeção de dependências e programação orientada a aspecto (*Aspect-oriented programming* ou AOP) com uma arquitetura baseada em *interfaces* e POJOs.

Muitas empresas utilizam o *Spring*, como por exemplo: *LinkedIn*, *MTV*, *Samsung*, *AT&T*, *NASA*, *CISCO Systems*, *Johnson & Johnson*, entre outras.

3.1 Enterprise Java Beans

Em dezembro de 1996, a *Sun Microsystems* publicou a especificação *JavaBeans* 1.00-A, que definia um modelo de componentes para Java, permitindo assim que objetos pudessem ser reutilizados e compostos para desenvolver aplicações mais complexas. Porém, parecia muito simples para ser capaz de realizar objetivos difíceis, como por exemplo suporte a transação, segurança e computação distribuída. Os desenvolvedores precisavam de algo mais completo e robusto.

Menos de dois anos depois, em março de 1998, a *Sun* publicou a versão 1.0 da especificação EJB (*Enterprise Java Beans*) que aumentou a noção de componentes Java no lado do servidor, fornecendo serviços empresariais extremamente necessários, porém, a mesma não era simples. De acordo com Walls (2011, p. 4), apesar de muitas aplicações ter sido feitas baseadas na EJB, ela nunca atingiu seu propósito que era simplificar o desenvolvimento de aplicações empresariais, deixando assim muitos desenvolvedores em busca de uma forma mais fácil de alcançar os objetivos propostos pela EJB.

De acordo com Walls (2011, p. 4), novas técnicas de programação, como por exemplo injeção de dependência e AOP estão dando aos *JavaBeans* grande parte do poder até então reservado apenas para EJB. Essas técnicas fornecem aos *plain-old Java Objects (POJOs)* um modelo declarativo de programação que vem da EJB mas sem herdar sua complexidade.

Atualmente, a EJB está na versão 3 que utiliza essas novas técnicas, simplificando (e muito) em relação aos seus antecessores. Porém ao tempo que a nova especificação EJB saiu, já existiam outros *frameworks* de desenvolvimento baseados em POJOs bem estabelecidos no mercado, como por exemplo, *Spring*, *PicoContainer* e *NanningAspects*.

3.2 O Spring

O *Spring* é um *framework* de código fonte aberto, originalmente criado por Rod Johnson e descrito em seu livro *Expert One-on-One: J2EE Design and Development*. (WALLS, 2011). Mesmo sendo concebido para lidar com a complexidade do desenvolvimento de aplicações empresariais, qualquer aplicação Java pode tirar proveito do *Spring* para maior simplicidade, capacidade de realizar teste e baixo nível de acoplamento entre seus diferentes módulos.

O *Spring* não foi feito para ser uma alternativa a EJB, mas provém poderosas, testadas, implementações de recursos, como por exemplo, controle de transação para POJOs, que possibilita desenvolvedores dispensar o uso de EJB em muitos projetos (JOHNSON; HOELLER, 2004). De acordo com Walls (2011), basicamente, o *Spring* foi feito com uma missão fundamental: simplificar o desenvolvimento Java.

Ainda de acordo com o mesmo autor, apesar dos termos *Bean* e *JavaBean* serem muito utilizados ao falar dos componentes da aplicação, isso não significa que um componente *Spring* deve seguir a especificação *JavaBean* a risca. Um componente *Spring* pode ser qualquer tipo de POJO (*Plain Old Java Object*).

Sendo assim, o *Spring* possui quatro principais características e praticamente tudo que ele faz, pode ser atribuído a elas, que são: (WALLS 2011):

- Desenvolvimento minimamente invasivo e leve com POJOs.
- Acoplamento fraco de componentes através de orientação a interface e injeção de dependência.
- Programação declarativa através de aspectos e convenções comuns.
- Redução de clichês através de aspectos e *templates*.

O *Spring* é dividido em vários módulos que fornecem tudo o que o desenvolvedor precisa para criar aplicações J2EE, mas não é obrigado a escolher todos eles, podendo assim escolher o que se encaixa melhor em seu projeto. O *Spring* inclusive oferece integração com vários outros *frameworks* e bibliotecas, como por exemplo: *Stripes*, *Struts*, *Hibernate*, *MyBatis*, entre outros.

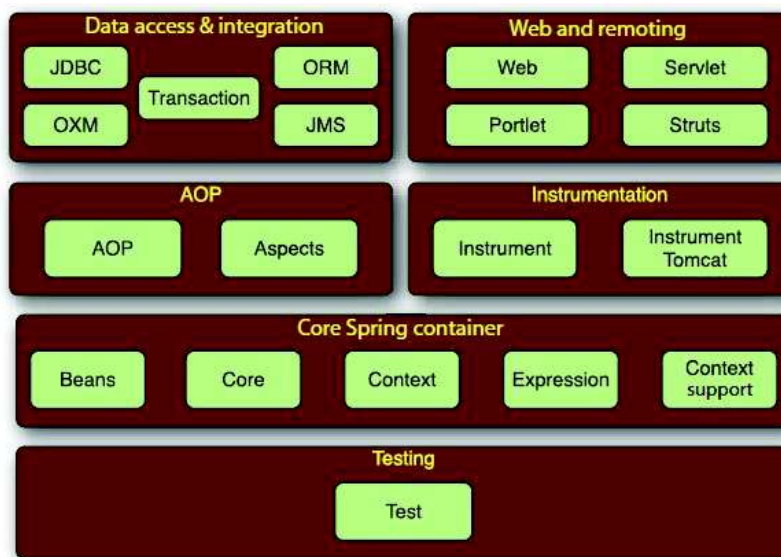


Figura 2 – Módulos do *Spring Framework*

Fonte: WALLS (2011)

3.3 Injeção De Dependência e Inversão De Controle

Injeção de dependência (*Dependency Injection* ou DI) é um padrão de projeto que deixa o código mais simples, fácil de entender e fácil de testar. (WALLS 2011). Uma vez que a maioria das aplicações um pouco mais complexas possuem classes que dependem uma da outra para realizar ações ou regras de negócio, sendo cada objeto responsável para obter suas referências para os objetos que dependem dele, gerando assim um código com alto nível de acoplamento e mais difícil de testar.

Inversão de controle, também é um padrão de projeto, sendo a DI um tipo popularizado deste, na verdade uma estratégia de implementação do IoC, no qual o container irá automaticamente resolver as dependências dos objetos, tirando essa responsabilidade do programador (JOHNSON; HOELLER, 2004).

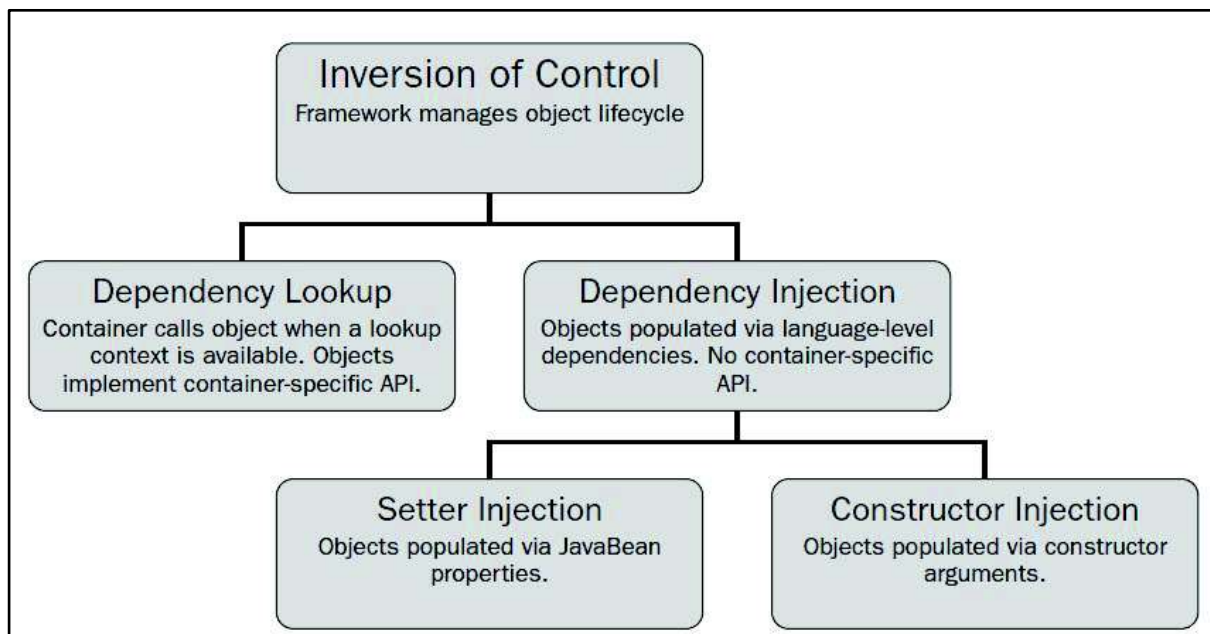


Figura 3 – Inversão de controle e suas implementações.

Fonte: JOHNSON, HOELLER (2004).

Conforme demonstrado na figura 2, será o *framework* que irá controlar o ciclo de vida do objeto e a DI pode ser de dois tipos: *Setter* ou no construtor. De acordo com Johnson e Hoeller (2004), essa abordagem possui as seguintes vantagens:

- Resolução de dependências (*Dependency Lookup*) é completamente removida do código da aplicação.
- Não existe dependência com o container.
- Não é necessária nenhuma *interface* especial.

3.4 Redução De Código Clichê (*Boilerplate*)

Muitos desenvolvedores já se depararam e até mesmo utilizaram *frameworks* que forçavam códigos clichês (ou *boilerplate*) que era obrigatório se estender uma de suas classes ou implementar uma de suas *interfaces*. Por exemplo, um *stateless session bean* na especificação EJB 2:

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class ExemploSessionBean implements SessionBean {
```

```

public void ejbActivate() {
}

public void ejbPassivate() {
}

public void ejbRemove() {
}

public void setSessionContext(SessionContext contexto) {
}

public String retornaDataAtual() {
    DateFormat df = new SimpleDateFormat('dd/MM/yyyy - HH:mm');
    return df.format(new Date());
}

public void ejbCreate() {
}
}

```

Essa classe mostra que a interface *SessionBean* força o desenvolvedor a implementar os métodos de ciclo de vida da EJB (os métodos que começam com “ejb”) mesmo que você não precise deles. Mas não é apenas a EJB que trabalhava de forma invasiva, também estão nessa lista *frameworks* como as primeiras versões do *Struts*, *WebWork* e *Tapestry*, dificultando até mesmo de realizar testes no código.

O *Spring* evita ao máximo encher o código com sua API, nunca forçando a implementação de uma interface ou estender uma classe específica dele. Na verdade, uma aplicação que utiliza *Spring* muitas vezes não possui indicações que está utilizando o mesmo, no máximo pode existir uma ou mais anotações do *Spring*. Dessa forma, tudo o que a classe “ExemploSessionBean” precisaria caso fosse um *bean* gerenciado pelo *Spring*, seria:

```

public class ExemploSessionBean {

    public String retornaDataAtual() {
        DateFormat df = new SimpleDateFormat("dd/MM/yyyy HH:mm");
        return df.format(new Date());
    }
}

```

```
    }
}
```

Dessa forma, o código é mantido limpo, sem nenhum indício do *Spring* ou de métodos do ciclo de vida do *bean* muito menos importar classes do *Spring*. De acordo com Walls (2011), apesar de seu formato simples, POJOs podem ser poderosos. Uma das maneiras de se aumentar o poder de um POJO é através da injeção de dependência.

Por exemplo, considerando a classe *Guitarrista* a seguir:

```
package br.com.tcc.classes;

public class Guitarrista implements Musico {
    private Guitarra guitarra;

    public Guitarrista() {
        this.guitarra = new Guitarra();
    }

    public void tocarGuitarra() throws InstrumentoException {
        this.guitarra.tocar();
    }
}
```

A classe *Guitarrista* está criando seu próprio instrumento (nesse caso *guitarra*), em seu método construtor. Isso cria um forte acoplamento entre a classe *Guitarrista* e a classe *Guitarra*, caso esse guitarrista precise tocar gaita durante um show, ele não conseguirá.

De acordo com Walls (2011), acoplamento é uma faca de dois gumes. De um lado, código fortemente acoplado é difícil de testar, utilizar, entender e tipicamente apresenta um comportamento que consertar um problema pode resultar no surgimento de novos defeitos. Por outro lado, certa quantidade de acoplamento é necessária uma vez que código totalmente desacoplado não cumpre nenhum objetivo, ou seja, para fazer algo útil, classes precisam (de alguma maneira) conhecer umas as outras.

Com a injeção de dependência, as dependências de objetos são fornecidas quando eles são criados por um terceiro elemento que coordena cada objeto no sistema, ou seja, as dependências são injetadas nos objetos que precisam delas. Dessa forma, a classe ficaria da seguinte maneira:

```
package br.com.tcc.classes;
```

```

public class MultiInstrumentista implements Musico {
    private Instrumento instrumento;

    public MultiInstrumentista(Instrumento guitarra) {
        this.instrumento = guitarra
    }

    public void tocarInstrumento() throws InstrumentoException {
        this.instrumento.tocar();
    }

    public Instrumento getInstrumento() {
        return this.instrumento;
    }

    public void setInstrumento(Instrumento instrumento) {
        this.instrumento = instrumento;
    }
}

```

A classe `MultiInstrumentista` não cria seu próprio instrumento, em sua construção, ela recebe um argumento (nesse caso a guitarra). Esse tipo de injeção de dependência é conhecido como injeção de construtor. O instrumento que a classe recebe é de um tipo `Instrumento`, uma interface que todos os instrumentos implementam, dessa forma, a classe pode tocar qualquer instrumento desde que ele implemente essa interface.

Esse é o principal benefício da injeção de dependência: acoplagem fraca. Se um objeto conhece suas dependências apenas através de sua interface e não por sua implementação ou como elas são instanciadas, então sua dependência pode ser trocada por uma diferente implementação sem que o objeto saiba a diferença (WALLS, 2011).

3.5 Ciclo de vida dos *beans*

Considerando uma aplicação Java tradicional, o ciclo de vida de um *bean* é simples: utiliza-se a palavra chave “*new*” para criar uma instância do *bean* que estará pronta para uso. Uma vez que esse *bean* não estiver mais em uso, o *garbage collector* do Java irá destruir essa instância.

Esse ciclo de vida se torna mais complexo quando se trata de um *bean* do *Spring*, uma vez que se pode personalizar como um *bean* é criado. De acordo com Walls (2011), uma *bean factory* realiza vários passos de configuração antes do *bean* estar pronto para o uso, que são:

- O *Spring* instancia o *bean*.
- Valores e referências de outros *beans* são injetados nas propriedades.
- Verificar se o *bean* implementa *BeanNameAware*.
- Verificar se o *bean* implementa *BeanFactoryAware*.
- Verificar se o *bean* implementa *ApplicationContextAware*.
- Verificar se algum *bean* implementa a interface *BeanPostProcessor*. Caso algum implemente, o *Spring* irá chamar o método *postProcessBeforeInitialization* desses *beans*.
- Verificar se algum *bean* implementa a interface *InitializingBean*. Nesse caso, será chamado o método *afterPropertiesSet* desses *beans*.
- Caso algum *bean* implemente a interface *BeanPostProcessor*, nesse caso, agora será chamado o método *postProcessAfterInitialization* desses *beans*.
- Agora, os *beans* estarão prontos para o uso na aplicação e irão permanecer assim até que o contexto da aplicação seja destruído.
- Se qualquer um dos *beans* implementar a interface *DisposableBean*, então o *Spring* irá chamar o método *destroy* dos respectivos *beans*.

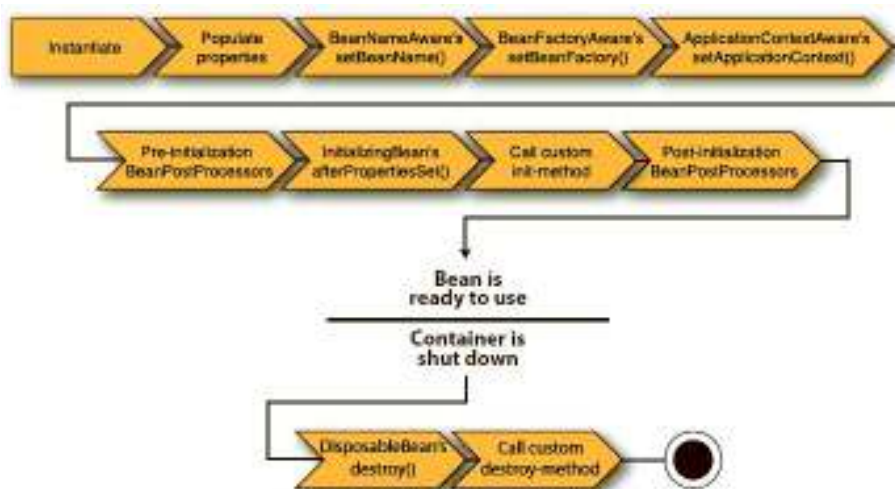


Figura 4 – Ciclo de vida do *bean*.

Fonte: WALLS (2011).

Em uma aplicação que utiliza *Spring*, os objetos terão seus ciclos de vida dentro do *container* do *Spring*, conforme é mostrado na imagem a seguir .

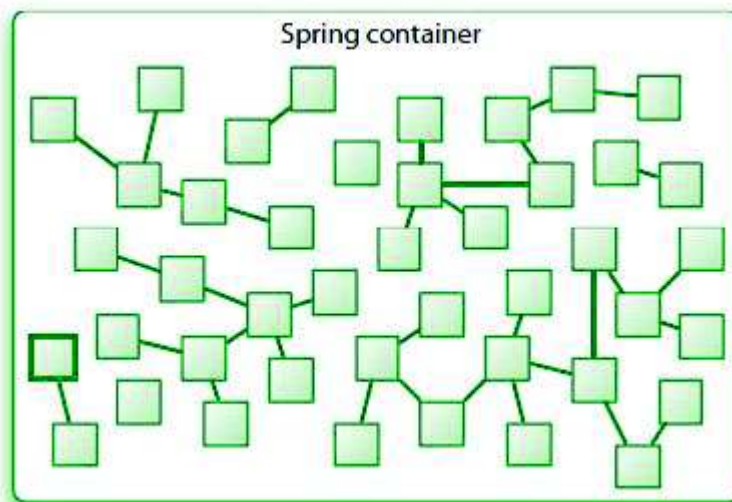


Figura 5 – Container do *Spring*.

Fonte: WALLS (2011).

O *container*, que está no núcleo do *Spring*, irá criar os objetos, realizar o *wiring*, configurá-los e gerenciar todo o ciclo de vida desses objetos (WALLS, 2011). Através da injeção de dependências, o *Spring* irá gerenciar todos os componentes que fazem parte da aplicação, inclusive criar as associações necessárias entre eles.

De acordo com Walls (2011), quando se fala em *container* do *Spring*, é importante ressaltar que o *Spring* vem com várias implementações dele que podem ser categorizadas em dois tipos distintos:

- *Beans factories* (ou fábrica de *beans*) que são os mais simples dos *containers*, provendo suporte básico para injeção de dependências.
- *Applications contexts* (ou contexto de aplicações) construídos sobre a noção de uma *bean factory*, fornecendo serviços de framework da aplicação como por exemplo buscar mensagens de texto de um arquivo *properties* e a habilidade de enviar eventos aos seus respectivos *listeners*.

Apesar de ser possível trabalhar com qualquer um dos dois tipos, as *beans factories* são de baixo nível para a maioria das aplicações. Portanto, *application context* são preferíveis (WALLS, 2011).

3.6 Application Context

O *Spring* possui vários tipos de *application context*. De acordo com Walls (2011) os três mais comuns são:

- *Classpath XML application context*, que carrega uma definições de contexto usando como fonte um arquivo XML localizado no *classpath*.
- *File system XML application context* que carrega definições de contexto usando como fonte um arquivo XML localizado no sistema de arquivos do computador.
- *XML web application context* que carrega definições de contexto usando como fonte um arquivo XML localizado dentro de uma aplicação *web*.

4 INJEÇÃO DE BEANS COM SPRING

O ato de criar essas associações entre objetos da aplicação é a essência da injeção de dependência e é comumente chamado de *wiring* (WALLS, 2011).

4.1 *Wiring de beans*

No *Spring*, os objetos não são responsáveis por encontrar e instanciar outros objetos que eles dependem para realizar operações. Ao invés disso, eles recebem do container, referências para esses objetos.

De acordo com Walls (2011), a ação de criar associações entre componentes da aplicação (beans) é chamada de *wiring*. No Spring, existem algumas maneiras de se realizar o *wiring* de beans, uma delas sendo através de arquivos XML (Extensible Markup Language), que por sua vez possuem várias *tags* (ou elementos) conhecidos como *namespaces* com os quais se pode configurar o *container Spring*, que, de acordo com Walls (2011) são:

- *aop*: fornece os elementos para declarar aspectos e proxy automático para classes anotadas com *@AspectJ* como aspectos no *Spring*.
- *beans*: o núcleo primitivo do *Spring*, que permite a declaração de beans e como deve ser realizado o *wiring* dos mesmos.
- *context*: vem com os elementos para configurar o *application context* do *Spring*, incluindo a habilidade de detectar, realizar o *wiring* e a injeção de objetos que não são diretamente gerenciados pelo *Spring* de forma automática.
- *jee*: oferece integração com APIs J2EE como por exemplo JNDI e EJB.
- *jms*: fornece os elementos de configuração para declarar POJOs *message-driven*.
- *lang*: permite a declaração de *beans* que são implementados como *scripts* *Groovy*, *JRuby* ou *BeanShell*.
- *mvc*: habilita o MVC (*Model-View-Controller*) do *Spring*, como por exemplo *controllers* orientados a anotação, *view-controller* e *interceptors*.
- *oxm*: suporta a configuração de facilidades do mapeamento objeto-XML do *Spring*.
- *tx*: fornece configuração do controle de transação declarativa.
- *util*: uma seleção diversa de elementos úteis, incluindo a capacidade de declarar *collections* (coleções) como *beans* e suporte para elementos *placeholder* de atributos.

Apenas os principais *namespaces* serão abordados nesse trabalho, uma vez que o objetivo é abordar o *Spring* e suas vantagens na camada de serviço de uma aplicação, mas é possível utilizar apenas o *Spring* como framework para construir uma aplicação J2EE inteira, desde a camada de acesso à dados (DAOs) até a camada de apresentação (MVC) juntamente com as *APIs J2EE*.

Para o *Spring* reconhecer os *namespaces* que o programador pretende utilizar e qual versão do *Spring*, além de colocar os *jars* do *Spring* no *classpath* da aplicação, é necessário declará-los no começo do arquivo XML da seguinte maneira:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-
                           3.0.xsd">

    <!--Declaração de beans ocorrem nesse espaço entre as tags beans-->

</beans>
```

4.2 Declarando *beans*

Considerando a classe *MultiInstrumentista* declarada anteriormente, é possível declarar um *bean* dessa classe da seguinte maneira utilizando arquivo XML, que nesse exemplo vamos chamar de *beans.xml*:

```
<bean id="multiInstrumentista"
      class="br.com.tcc.classes.MultiInstrumentista"/>
```

Uma vez declarado, considerando que o tipo de *ApplicationContext* utilizado é o de *ClassPath*, é possível acessar esse *bean* utilizando o seguinte código:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
MultiInstrumentista multiInstrumentista = (MultiInstrumentista)
    ctx.getBean("multiInstrumentista");
multiInstrumentista.tocarInstrumento();
```

Nota-se que não foi preciso instanciar a classe `MultiInstrumentista` utilizando o `new` em Java. Isso porque essa classe já foi instanciada pelo *Spring* no momento que a aplicação foi iniciada. Só foi necessário pegar a referência desse *bean* através do *ApplicationContext*.

Até o momento, a grande vantagem de se utilizar o *Spring* não é tão aparente.

4.3 Injeção através de construtores

A injeção de dependência baseada em construtores é realizada pelo container, invocando um construtor com um número de argumentos, cada um representando uma dependência (JOHNSON et. al., 2011). A resolução de argumentos do construtor ocorre utilizando o tipo (ou classe) do argumento. Se nenhuma potencial ambiguidade existir nos argumentos do método construtor na definição do *bean*, então, a ordem que os argumentos foram fornecidos na declaração do *bean*, será a ordem que esses argumentos serão passados ao método construtor.

Assumindo uma classe possua o seguinte método construtor:

```
package br.com.tcc.classes;

public Livro(String titulo, Integer paginas) {
    this.titulo = titulo;
    this.paginas = paginas;
}
```

É possível declarar o seguinte *bean* para essa classe:

```
<bean id="guerraDosTronos" class="br.com.tcc.classes.Livro">
    <constructor-arg type="java.lang.String" value="A Guerra dos Tronos"/>
    <constructor-arg type="java.lang.Integer" value="591"/>
</bean>
```

Neste cenário, o *Spring* precisou de ajuda para determinar o tipo dos valores através do atributo “*type*” no argumento do construtor.

Considerando agora a classe `MultiInstrumentista` novamente, é possível realizar a seguinte declaração de *beans*:

```
package br.com.tcc.classes;

public class Guitarra implements Instrumento {
```

```

private String cor;
private String marca;

public Guitarra(String cor, String marca) {
    this.cor = cor;
    this.marca = marca;
}

public getCor() {
    return this.cor;
}

public setCor(String cor) {

    this.cor = cor;
}

public getMarca() {
    return this.marca;
}

public setMarca(String marca) {
    this.marca = marca;
}

}
<bean id="guitarra" class="br.com.tcc.classes.Guitarra">
    <constructor-arg type="java.lang.String" value="Vermelha"/>
    <constructor-arg type="java.lang.String" value="Gibson"/>
</bean>
<bean id="guitarrista" class="br.com.tcc.classes.MultiInstrumentista">
    <constructor-arg ref="guitarra"/>
</bean>

```

Nesse caso, um dos argumentos do construtor foi passado como um *bean*, utilizando o atributo “ref” (que aponta para a referência do *bean* que possui o id igual a “guitarra”) ao invés de “value” porque se trata de uma classe que o programador cria e não uma classe do Java.

4.4 Injeção em propriedades do *bean*

Tipicamente, propriedades *JavaBean* são *private* e terão um par de métodos de acesso na forma de “setX” e “getX” (WALLS, 2011). O *Spring* consegue tirar vantagem de um método *setter* para configurar o valor dessa propriedade através de injeção via *setter*. Por exemplo:

```
<bean id="gibsonVermelha" class="br.com.tcc.classes.Guitarra">
    <property name="cor" value="Vermelha"/>
    <property name="marca" value="Gibson"/>
</bean>
<bean id="guitarrista" class="br.com.tcc.classes.MultiInstrumentista">
    <property name="instrumento" ref="gibsonVermelha"/>
</bean>
```

Deste modo, os *beans* instanciados pelo *Spring* irão receber esses atributos (ou propriedades) sem necessitar defini-los no construtor.

De acordo com Walls (2011), o verdadeiro poder de se utilizar o atributo *property*, não está apenas em realizar o *wiring* de valores *hardcoded* (ou estáticos), mas sim em referenciar os outros objetos na aplicação que determinadas classes dependem sem que elas precisem realizar esse trabalho (através de construtores, por exemplo).

Ao longo dos exemplos, foi possível observar que, mesmo não sendo uma funcionalidade do *Spring*, o grande benefício de se programar para interfaces, é que a classe *MultiInstrumentista* pode receber como propriedade qualquer instrumento desde que ele implemente a interface *Instrumento*, fornecendo assim um fraco acoplamento entre as classes da aplicação.

De acordo com Johnson et. al. (2011), o *Spring* recomenda a injeção através de *setters* porque muitos argumentos no método construtor pode se tornar muito pesado, especialmente quando as propriedades são opcionais. Métodos *setters* também fazem com que os objetos de determinada classe fiquem mais sujeitos a reconfiguração ou reinjeção em outros momentos. A desvantagem de se utilizar injeção através de *setters* é que o objeto pode não voltar ao código que o chamou em um estado que foi totalmente inicializado.

Ainda de acordo com os mesmos autores, os programadores devem utilizar o método que mais fizer sentido em cada caso. Uma vez que, ao lidar com classes de terceiros em que não se tem o código fonte, a classe pode não expor nenhum método *getter* e *setter*, portanto, injeção via método construtor é a única opção neste cenário.

4.5 Wiring de propriedades utilizando namespace p

Como alternativa a tag `<property>`, é possível utilizar o *namespace* `p` do *Spring*, bastando adicionar a URI de *schema* no XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

Feito isso, já é possível utilizar o *namespace* `p` da seguinte maneira:

```
<bean id="gibsonVermelha" class="br.com.tcc.classes.Guitarra"
      p:cor = "Vermelha"
      p:marca = "Gibson"/>
<bean id="guitarrista" class="br.com.tcc.classes.MultiInstrumentista"
      p:instrumento-ref = "gibsonVermelha"/>
```

O atributo `p:cor` no *bean* `gibsonVermelha` é definido como “Vermelha”, realizando o *wiring* da propriedade com aquele valor. Entretanto, no *bean* `guitarrista`, foi possível observar o surgimento do sufixo `-ref` que basicamente diz ao *Spring* que para aquela propriedade deve ser feito o *wiring* de receber a referência do *bean* `gibsonVermelha` e não seu valor literal.

De acordo com Walls (2011), fica à escolha do programador qual das duas abordagens utilizar, uma vez que ambos funcionam bem e a principal vantagem do *namespace* `p` é quando se escreve exemplo para um livro com margens fixas.

4.6 Wiring de collection

Até agora, foi possível observar que os atributos *value* e *ref* do *Spring* funcionam bem mas apenas quando as propriedades dos *beans* possuem valores únicos.

O *Spring* oferece quatro tipos de configuração para configurações de *collection* que seguem uma idéia muito parecida com as *collections* do Java, que, de acordo com Walls (2011) são:

- *list*: *wiring* para valores permitindo serem duplicados.
- *set*: *wiring* para valores que garante que não haja duplicidade

- *map*: *wiring* de uma *collection* com pares de chave-valor onde nome e valor podem ser de qualquer tipo.
- *props*: *wiring* de *collection* com pares de chave-valor onde ambos são do tipo *String*.

Ainda de acordo com o mesmo autor, os elementos *list* e *set* são úteis quando se configura propriedades que são vetores ou alguma implementação da interface `java.util.Collection`. Já os elementos *map* e *props* correspondem à *collections* que são do tipo `java.util.Map` e `java.util.Properties`, respectivamente.

Considerando a seguinte classe *Prateleira* e assumindo que os outros *beans* já foram declarados, a declaração do *bean* dessa classe ficaria da seguinte maneira:

```
package br.com.tcc.classes;

public class Prateleira {
    private List<Livro> livros;

    public getLivros() {
        return this.livros;
    }

    public setLivros(List<Livros> livros) {
        this.livros = livros;
    }
}

<bean id="prateleira" class="br.com.tcc.classes.Prateleira">
    <property name="prateleira">
        <list>
            <ref bean="guerraDosTronos"/>
            <ref bean="furiaDosReis"/>
            <ref bean="tormentaDasEspadas"/>
        </list>
    </property>
</bean>
```

Ou ainda uma outra maneira de realizar a declaração:

```
<bean id="prateleira" class="br.com.tcc.classes.Prateleira">
  <property name="prateleira">
    <set>
      <ref bean="guerraDosTronos"/>
      <ref bean="furiaDosReis"/>
      <ref bean="tormentaDasEspadas"/>
    </set>
  </property>
</bean>
```

Nesse caso, o elemento *set* poderia ter sido utilizado para realizar a mesma declaração. Não é porque uma propriedade é do tipo `java.util.Set` que você tem que utilizar o elemento *set*, a única diferença é que seria possível garantir que todos os elementos daquela lista serão únicos.

Agora, será utilizada uma versão levemente modificada de classe `MultiInstrumentista`, para demonstrar o *wiring* de propriedades do tipo `java.util.Map`:

```
package br.com.tcc.classes;

public class MultiInstrumentista {
    private Properties atributos;

    private Map<String, Instrumento> instrumentos;

    public void tocarInstrumentos() {
        for (String chave : instrumentos.keySet()) {
            Instrumento instrumento = instrumentos.get(chave);
            instrumento.tocar();
        }
    }

    public void setInstrumentos(Map<String, Instrumento> instrumentos) {
        this.instrumentos = instrumentos;
    }

    public void setAtributos(Properties atributos) {
        this.atributos = atributos;
    }
}
```

```

    }
}

<bean          id="oneManBand"          class="br.com.tcc.MultiInstrumentista">
  <property name="instrumentos">
    <map>
      <entry key="GUITARRA" value-ref="gibsonVermelha"/>
      <entry key="GAITA" value-ref="gaita"/>
      <entry key="VIOLAO" value-ref="violao"/>
    </map>
  </property>
</bean>

```

O elemento *entry* é feito de uma chave e um valor, que podem ser tanto primitivos (*integer*, *string*) quanto referencias para *beans*. Portanto, existem as seguintes propriedades para o elemento *entry*: *key*, *value*, *key-ref* e *value-ref*, podendo ou não serem combinados entre si, de acordo com os requisitos do projeto.

Por fim, para demonstrar o uso do elemento *props*, ficaria da seguinte maneira:

```

<bean          id="oneManBand"          class="br.com.tcc.MultiInstrumentista">
  <property name="atributos">
    <props>
      <prop key="CABELO">Preto</prop>
      <prop key="ESTATURA">Media</prop>
      <prop key="TIPO_SANGUINEO">B+</prop>
    </props>
  </property>
</bean>

```

4.7 Wiring de propriedades com valor nulo

De acordo com Walls (2011), por mais estranho que pareça, pode surgir a necessidade de realizar o *wiring* de alguma propriedade de um objeto para nulo, como por exemplo algum caso onde o programador desejar forçar um valor nulo ou para sobrescrever uma propriedade que o *wiring* foi feito automaticamente. É possível realizar isso com o *Spring* da seguinte maneira:

```

<!--Restante da declaração omitida-->
<property name="valorNulo"><null/></property>

```

4.8 Escopos de *beans*

Por padrão, todo bean declarado no *Spring* são *singletons*, isto é, na aplicação toda, só existirá uma única instância daquele *bean* (WALLS, 2011), mas em alguns casos, pode ser necessário que o programador precise de uma nova instância do *bean* cada vez que ele é chamado. Nesses casos, a tag “*bean*” possui o atributo “*scope*” para mudar esse comportamento padrão. Por exemplo:

```
<!--Restante da declaracao omitida-->
<bean id="ingresso" class="br.com.tcc.classes.Ingresso" scope="prototype"/>
```

De acordo com Walls (2011), existem os seguintes tipos de opção de escopo para os *beans* do *Spring*:

- *singleton*: apenas uma instância do *bean* por *container Spring*.
- *prototype*: permite que um *bean* seja instanciado quantas vezes for necessário, isso é, cada vez que é chamado ou referenciado.
- *request*: permite que o *bean* seja instanciado por cada requisição HTTP. Só é válido quando se usa um contexto que é capaz de realizar operações na *web*, como por exemplo o *Spring MVC*.
- *session*: mesma limitação do *request* mas o *bean* é instanciado por sessão.
- *global-session*: o escopo do *bean* é ligado à uma sessão HTTP global. É válido apenas quando usado em um contexto *portlet*.

Ainda de acordo com o mesmo autor, a noção de *singleton* do *Spring* que é limitado ao contexto da aplicação, é diferente dos verdadeiros *singletons* que garante uma única classe por *classloader*. Desse modo, nada impede que o programador crie novas instâncias das classes que foram declaradas como *beans* de escopo singleton da maneira tradicional em Java, utilizando “*new*”.

4.9 Minimizando a configuração XML

De acordo com Walls (2011), a configuração em arquivos XML divide opiniões entre os programadores. Alguns não gostam, outros sim, mas com o *Spring* é possível minimizar essa configuração através do *wiring* automático de *beans* que pode ser de quatro tipos:

- *byName*: tenta realizar o *wiring* de propriedades que tenham o mesmo nome dos IDs dos *beans*. Propriedades que não possuem *bean* correspondente continuarão nulas.
- *byType*: Tenta conferir se o tipo do *bean* e da propriedade são iguais para realizar o *wiring*. Assim como o anterior, caso não encontre correspondência, a propriedade continuará nula.
- *constructor*: Verifica quais *beans* podem ser designados para determinado método construtor e realizar o *wiring* dos mesmos.
- *autodetect*: primeiro tenta aplicar o tipo *constructor*. Caso dê errado, tenta com o tipo *byName*.

Existe a possibilidade de realizar o *wiring* por anotação, para isso é necessário adicionar a seguinte entrada no arquivo XML do *Spring*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.0.xsd">
<context:annotation-config/>
<context:component-scan base-package="br.com.tcc.service"/>
<!-- Declaracao de beans omitida -->
</beans>
```

Partindo para um exemplo mais parecido com o encontrado nos *softwares*, supondo que existe uma fachada de negócio chamada *PedidoFacadePojo.java*, que por sua vez, terá como propriedade um DAO (*Data Access Object*) chamado *PedidoDAO.java* para realizar as operações em banco de dados. Para realizar o *wiring* com anotação, bastaria fazer o seguinte:

```
package br.com.tcc.service.dao;

@Repository("pedidoDAO")
```

```

@Scope("singleton")
public class PedidoDAO {
    // Restante da classe omitido
}

package br.com.tcc.service.facade.impl;
@Service("pedidoFacade")
@Scope("singleton")
public class PedidoFacadePojo implements PedidoFacade {

    @Autowired
    @Qualifier("pedidoDAO")
    private PedidoDAO pedidoDAO;

    // restante da classe omitido
}

```

Agora, quando a aplicação iniciar, será possível utilizar uma instância da classe `PedidoDAO` dentro da instância da classe `PedidoFacadePojo`, que já foram ambas instanciadas pelo *Spring*. As anotações `@Service`, `@Repository`, `@Controller` e `@Component` dizem ao *Spring* que ele deve procurar pelas classes anotadas e criar *beans* para elas com o nome (ID) especificado como argumento entre parênteses. A anotação `@Autowired` fala que naquela propriedade, deve ser feito o *wiring* de um *bean* que possui o ID igual ao passado como argumento na anotação `@Qualifier`. Tudo isso, desde que elas estejam dentro do pacote definido no *namespace* `<context:component-scan/>` no arquivo XML, aumento ainda mais a produtividade no *wiring* de *beans*.

5 SPRING EXPRESSION LANGUAGE

Normalmente, o *wiring* de *beans* é definido de forma estática no XML de configuração do *Spring* com valores ou referências estáticas. Mas caso a necessidade de realizar o *wiring* utilizando valores que são desconhecidos até que se execute determinada rotina ou código surja, não será possível utilizar a mesma abordagem demonstrada.

A partir do *Spring* 3, foi introduzida a *Spring Expression Language* ou SpEL, uma poderosa mas sucinta maneira de realizar o *wiring* de valores em uma propriedade de *bean* ou argumentos de métodos construtores utilizando expressões que são avaliadas em tempo de execução.

Usando a SpEL, o desenvolvedor pode realizar façanhas de *wiring* que seriam bem mais difíceis ou até mesmo impossíveis de se alcançar utilizando a abordagem tradicional de *wiring* do *Spring* (WALLS, 2011).

De acordo com Walls (2011), a SpEL possui vários truques como por exemplo:

- Habilidade de referenciar *beans* pelos seus Ids.
- Invocar métodos e acessar propriedades em objetos.
- Operações lógicas, matemáticas e relacionais em valores.
- Checagem com expressões regulares.
- Manipulação de *collections*.

5.1 Fundamentos da SpEL

De acordo com Walls (2011), o objetivo final de uma expressão SpEL é chegar em algum valor após ser avaliada.

Para ser utilizada no arquivo XML do *Spring*, é necessário colocar a expressão dentro dos marcadores `#{}` , como a expressão com valor literal (numérico com *string*) por exemplo:

```
<!--Restante da declaração omitida-->
<property name="anoFimDoMundo" value="#{2012}"/>
<property name="mesFimDoMundo" value="#{'Dezembro}'"/>
```

Também é possível misturar as expressões com valores que não são da SpEL:

```
<!--Restante da declaração omitida-->
<property name="anoFimDoMundo" value="O ano do fim do mundo é #{2012}"/>
```

Números de ponto flutuante ou até mesmo números utilizando notação científica podem ser representados com SpEL, como por exemplo:

```
<!--Restante da declaração omitida-->
<property name="frequencia" value="#{95.7}"/>
<property name="capacidade" value="#{1e4}"/>
```

Valores booleanos podem ser representados da seguinte maneira em SpEL:

```
<!--Restante da declaração omitida-->
<property name="mundoVaiAcabar" value="#{false}"/>
```

Até o momento, não se pôde notar nenhuma grande vantagem em utilizar SpEL, uma vez que não era preciso utilizá-la para realizar a maioria dos *wirings* descritos acima. Mas, de acordo com Walls (2011), expressões SpEL mais interessantes, são compostas de expressões mais simples, portanto é importante a capacidade da SpEL para valores literais.

5.2 Referenciando propriedades e métodos

Para referenciar um *bean* utilizando a SpEL, basta colocar o ID do *bean* entre os marcadores `#{}`, mas isso também é possível fazer sem utilizá-la. Mas o interessante disto, é que torna possível acessar atributos e métodos dos *beans*. Supondo que a classe “Banda” possua o atributo “musica”, que deve ser a mesma de outra classe “Cantor” para trabalharem em conjunto. Dessa forma, a declaração do *bean* ficaria da seguinte maneira:

```
<bean id="oasis" class="br.com.tcc.classes.Banda">
    <property name="musica" value="Wonderwall"/>
</bean>
<bean id="noelGallagher" class="br.com.tcc.classes.Cantor">
    <property name="musica" value="#{multiInstrumentista.musica}"/>
</bean>
```

Conforme demonstrado pelo código acima, a propriedade “musica” do *bean* “oasis” foi acessada e injetada no atributo “musica” do *bean* “noelGallagher” da seguinte maneira: a primeira parte, que vem antes do ponto utilizado como delimitador, faz referência ao *bean* e a segunda parte, após ao ponto, referencia a propriedade do *bean* referenciado.

Acessar propriedades não é a única possibilidade quando se referencia *beans* utilizando SpEL, é possível também acessar métodos dos *beans*. Assumindo que a classe “Cantor” possua um método para gerar escolher frases aleatórias chamado “escolherFrase”, é possível o seguinte uso da SpEL:

```
<!--Restante da declaração omitida-->
<property name="frase" value="#{noelGallagher.escolherFrase()}" />
```

Agora, por qualquer que seja o motivo, é necessário que o retorno desse método (que é uma *String*) em uma ocasião especial, seja totalmente com letras maiúsculas. Um jeito de atingir isso é criando um novo método na classe que retornaria a propriedade “frase” e utilizando a função java “.toUpperCase()”. Isso apenas iria “sujar” o código da classe para atender uma exceção a regra. Porém, é possível resolver este problema de forma mais elegante, da seguinte maneira:

```
<!--Restante da declaração omitida-->
<property name="fraseMaiuscula"
value="#{noelGallagher.escolherFrase().toUpperCase()}" />
```

Isso resolveria o problema, mas caso o método “escolherFrase” retorne um valor nulo, iria ocorrer a exceção *NullPointerException* bastante conhecida pelos programadores Java. É possível contornar esse problema (e até mesmo se tornando uma boa prática) utilizando o acesso *null-safe* da seguinte maneira:

```
<property name="fraseMaiuscula"
value="#{noelGallagher.escolherFrase() ?.toUpperCase()}" />
```

Como foi possível observar, ao invés de utilizar ponto (.) para acessar o método “toUpperCase”, foi utilizado o operador “?.” que garante que o item à sua esquerda não é nulo antes de acessar o item à sua direita. Dessa maneira, se o método escolherFrase retornar um valor nulo, a SpEL não iria tentar acessar o método toUpperCase.

5.3 Tipos na SpEL

De acordo com Walls (2011), a chave para se trabalhar com métodos e constantes dentro do escopo da classe, é usar o operador “T()”. Por exemplo, para se acessar a classe java Math e a constante referente ao número pi em SpEL, é necessário utilizar o operador da seguinte maneira:

```
<property name="pi" value="#{T(java.lang.Math).PI}"/>
```

Também é possível acessar métodos através do operador de tipo:

```
<property name="numeroAleatorio" value="#{T(java.lang.Math).random()}/>
```

5.4 Realizando operações com SpEL

A SpEL oferece diversas operações que é possível aplicar em valores. De acordo com Johnson et. al. (2011), a SpEL suporta todas as operações básicas que o Java suporta com a adição do operador “^” para realizar potências.

Operação	Operadores
Aritmética	+, -, *, /, %, ^
Relacional	<, >, ==, <=, >=, lt, gt, eq, le, ge
Lógica	and, or, not,
Condicional	?:(ternário)
Expressão regular	matches

Tabela 1: tipos de operações e operadores que podem ser utilizados com a SpEL.

Portanto, é possível realizar várias operações e, com o resultado dessas operações, realizar o *wiring* em propriedades de *beans*:

```
<property name="bonus" value="#{empregado.salario + 500}"/>
<property name="ferias" value="#{empregado.salario * 1.3}"/>
<property name="circunferencia" value="#{2 * T(java.lang.Math).PI *
    circulo.raio}"/>
<property name="area" value="#{T(java.lang.Math).PI * circulo.raio ^ 2}"/>
<property name="media" value="#{contador.total / contador.quantidade}"/>
<property name="resto" value="#{contador.total % contador.quantidade}"/>
```

```
<property name="nomeCompleto" value="#{empregado.nome + ' ' +
    empregado.sobrenome}"/>
```

Nota-se que o operador de adição (+) também pode ser usado para a concatenação de *Strings*, literais ou não.

Para comparar valores, a SpEL fornece todos os operadores de comparação que existem no Java e a avaliação da expressão, irá retornar um valor booleano (verdadeiro ou falso).

```
<property name="igual" value="#{contador.total == 100}"/>
<property name="menorIgual" value="#{contador.total <= 100}"/>
<property name="maiorIgual" value="#{contador.total >= 0}"/>
```

Mas, infelizmente, os operadores de menor e maior (< e > respectivamente), apresentam um problema quando são utilizados no XML de configuração do *Spring*, uma vez que possuem significado especial no XML (aberturas e fechamentos de *tags* por exemplo). Portanto, é mais recomendada a utilização dos operadores textuais equivalentes, conforme demonstrado na tabela abaixo.

Operação	Símbolo do operador	Textual equivalente
Igual	==	eq
Menor que	<	lt
Menor ou igual a	<=	le
Maior que	>	gt
Maior ou igual a	>=	ge

Tabela 2: Operadores de comparação e respectivos operadores.

```
<property name="temVaga" value="#{hotel.hospedes lt hotel.capacidade}"/>
```

Caso o programador necessite avaliar expressões baseadas em duas comparações ou negar alguma delas ou até mesmo negar um valor booleano, utiliza-se os operadores lógicos do SpEL.

Operador	Operação
<i>and</i>	Ambos os lados do operador devem ser verdadeiros para a expressão ser verdadeira.
<i>or</i>	Um dos lados deve ser verdadeiro para a expressão ser verdadeira.
<i>not</i> ou <i>!</i>	Retorna o valor inverso da operação ou expressão.

Tabela 3: operadores lógicos e operações da SpEL.

```
<property name="lotado" value="#{!hotel.hospedes lt hotel.capacidade}"/>
<property name="produtoIndisponivel" value="#{not produto.disponivel or
    produto.quantidade == 0}"/>
<property name="circuloGrande" value="#{forma.tipo == 'circulo' and
    forma.perimetro gt 10000}"/>
```

De acordo com Walls (2011), a SpEL não fornece operadores simbólicos equivalentes para os operadores *and* e *or*.

Há também a possibilidade de se utilizar o operador ternário da SpEL, que, de acordo com o mesmo autor, é muito utilizado para verificar se determinado atributo é nulo. Caso seja, atribui-se um valor padrão para a propriedade.

```
<property name="musica" value="#{oasis.musica != null ? oasis.musica :
    'wonderwall'}"/>
```

No exemplo acima, caso a propriedade música do *bean* oasis não for nula, utiliza-se a propriedade para o *wiring*, caso contrário, atribui-se um *wiring* padrão com “wonderwall”.

5.5 Utilizando expressões regulares na SpEL

Quando se trabalha com texto, algumas vezes é útil checar se determinado texto corresponde com determinado padrão (WALLS, 2011). Para isso, a SpEL possui o operador *matches* que tenta aplicar uma expressão regular (dada como argumento ao lado direito) a um valor de uma *String* (dado como argumento ao lado esquerdo). Pode-se utilizar essa técnica, por exemplo, para verificar se um e-mail é válido:

```
<property name="emailValidoBR" value="#{cadastro.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.\com.br}'"/>
```

5.6 Utilizando *collections* com SpEL

De acordo com Walls (2011), um dos truques mais impressionantes da SpEL envolve trabalhar com *collections*. Não é possível referenciar um único item de uma *collection* como no Java, mas também selecionar determinado item baseado nos valores de seus atributos ou até mesmo extrair propriedades de um item da *collection* e coloca-las em uma nova *collection*.

Considerando a classe Cidade, que possui os atributos nome, uf e “populacao”, cuja declaração foi omitida para preservar espaço, pode-se configurar a seguinte lista de objetos Cidade com dados fictícios no *Spring*:

```
<util:list id="cidades">
  <bean class="br.com.tcc.classes.Cidade" p:nome="Araraquara" p:uf="SP"
    p:populacao="250000"/>
  <bean class="br.com.tcc.classes.Cidade" p:nome="Matão" p:uf="SP"
    p:populacao="90000"/>
  <bean class="br.com.tcc.classes.Cidade" p:nome="Taquaritinga" p:uf="SP"
    p:populacao="50000"/>
</util:list>
```

O elemento *util:list* cria um *bean* do tipo `java.util.List` que por sua vez contém uma lista de *beans* do tipo Cidade. É possível acessar um único elemento dessa lista baseando-se na posição do elemento, cuja primeira posição começa em zero e vai até o numero de elementos menos um nesse caso.

```
<property name="cidadeEscolhida" value="#{cidades.[1]}" />
```

No exemplo acima, foi selecionado o segundo elemento da lista de cidades, que é a cidade de Matão e foi feito o *wiring* dessa cidade extraída no atributo cidadeEscolhida. Poderia ser possível também escolher uma cidade de forma aleatória da seguinte maneira:

```
<property name="cidadeEscolhida"
  value="#{cidades[T(java.lang.Math).random() * cidades.size()]}" />
```

Ou até mesmo, caso os objetos da classe Cidade estivessem em um *map* em que as chaves fossem os nomes das cidades, buscar determinado elemento da seguinte maneira:

```
<property name="cidadeEscolhida" value="#{cidades['Araraquara']}" />
```

Agora, supondo que o programador deseje filtrar essa lista e selecionar apenas as cidades do estado de São Paulo (assumindo que essa lista possuísse todas as cidades do Brasil e que não fosse proveniente de um banco de dados) seria um trabalho árduo. Mas com a SpEL é possível realizar isso utilizando o operador “.” da seguinte maneira:

```
<property name="cidadesSP" value="{cidades.[uf eq 'SP']}"/>
```

Essa operação irá criar uma nova *collection* cujos elementos incluem apenas os elementos da *collection* de origem que atenderam ao critério utilizado. Também existem outros dois operadores de seleção: o “.” e o “\$” que retornam o primeiro elemento que corresponder ao critério e o último elemento que corresponder ao critério, respectivamente.

```
<property name="umaCidade" value="{cidades.$[populacao gt 60000]}/>
<property name="umaCidade" value="{cidades.^[populacao gt 60000]}/>
```

Assumindo que a lista de cidades esteja como foi declarada acima, a primeira expressão irá retornar a cidade de Matão e a segunda, a cidade de Araraquara. O último operador de *collection* da SpEL, é o “!” que coleta propriedades de cada elemento de uma *collection* e o coloca em uma nova *collection*. Por exemplo, para se criar uma lista de *Strings* que consiste no nome da cidade concatenado com “ – ” e seu respectivo estado, utiliza-se a SpEL da seguinte maneira:

```
<property name="cidadesUF" value="{cidades.![nome + ' - ' + uf]}/>
```

Essa funcionalidade é chamada de projeção de *collections* (WALLS, 2011).

6 CONTROLE DE TRANSAÇÃO COM SPRING

Quando se está desenvolvendo *softwares*, operações tudo ou nada são chamadas de transações (WALLS, 2011). Transações permitem que o programador agrupe determinadas operações em uma única unidade de trabalho em que ocorre totalmente sem problemas ou não ocorre. Se tudo ocorre sem problemas a transação é sucedida. Mas se algo der errado, tudo que havia acontecido até ocorrer o problema é desfeito e fica como se nada tivesse acontecido.

Para se ilustrar melhor uma transação, pode-se utilizar como exemplo a realização de um DOC no banco, onde o sacado informa o valor e o favorecido do DOC. Esse valor é reduzido da conta corrente do sacado após a verificação que o valor está dentro dos limites de saldo do sacado. E, logo após isso, ocorre algum problema no sistema. O valor já foi debitado da conta corrente do sacado, mas não foi creditado na conta do favorecido, o dinheiro se perdeu e o banco saiu ganhando. O certo seria que o valor fosse devolvido à conta do sacado, que é o que ocorre na maioria das vezes no mundo real.

Transações possuem um papel muito importante em *softwares* uma vez que garantem que os dados e os recursos nunca são deixados em um estado inconsistente (WALLS, 2011).

Ainda de acordo com o mesmo autor, um acrônimo foi criado para descrever transações: ACID, que significa:

- Atômica: transações são feitas de uma ou mais atividades empacotadas. Atomicidade garante que todas as operações dentro da transação ocorrem ou nenhuma delas ocorre.
- Consistente: uma vez que uma transação termina (com sucesso ou não), o sistema é deixado em um estado consistente com a modelagem de negócio. Os dados não podem ser corrompidos em relação à realidade.
- Isolada: transações devem permitir vários usuários trabalharem com o mesmo dado, sem que o dado utilizado por um usuário se misture com o do outro. Ou seja, transações devem ser isoladas umas das outras, prevenindo que ocorram leituras e escritas concorrentes ao mesmo dado. Isolamento muitas vezes significa realizar o *lock* de uma linha ou de uma tabela no banco de dados.

- **Durável:** Uma vez que a transação termina, o resultado dessa transação deve ser persistido para que sobreviva a qualquer tipo de pane no sistema. Como por exemplo, salvar o resultado em um banco de dados.

6.1 Controle de transações do *Spring*

Assim como o EJB, o *Spring* suporta o controle de transação programática ou declarada, mas a capacidade de gerenciamento do *Spring* ultrapassa a da EJB (WALLS, 2011).

O suporte para gerenciamento de transação programática do *Spring* é muito diferente em relação à da EJB. Ao invés de possuir um acoplamento com a implementação da API Java *Transaction* (JTA), o *Spring* utiliza um mecanismo de *callback* que abstrai a implementação da transação do código transacional. Na verdade, o gerenciamento de transações do *Spring* nem mesmo requer uma implementação JTA, suportando um recurso de persistência único ou distribuído. (WALLS, 2011).

Escolher entre um controle de transação programática ou declarativa é uma decisão que considera controle sobre o código contra a conveniência. Quando o programador define as transações dentro do código, ganha controle sobre onde a transação começa e onde ela termina de forma precisa. Normalmente esse nível de controle não é necessário (WALLS, 2011).

Ainda de acordo com o mesmo autor, existem quatro tipos de controle de transação no *Spring* que delegam o controle de transação para implementações específicas de cada plataforma:

- *DataSource*: utiliza-se esse tipo quando a plataforma escolhida é a JDBC ou *Mybatis*.
- *Hibernate*: utiliza-se esse tipo quando a aplicação utiliza o *framework Hibernate*.
- JPA: utilizado quando a aplicação utiliza JPA.
- JTA: utilizado quando a aplicação utiliza uma implementação da JTA.

Para habilitar o controle de transação do *Spring*, é necessário adicionar entradas no arquivo XML de configuração:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
```

```

xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/          spring-beans-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

```

6.2 Transações JDBC

Se a aplicação estiver utilizando JDBC ou *MyBatis* para persistência, deve-se utilizar o controle de transação *DataSource* da seguinte maneira:

```

<bean id="controleTx"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

```

É importante notar que está sendo feito o *wiring* de um *bean* chamado *dataSource*, que também deve ser declarado no XML. Existem diversas maneiras de se definir um *dataSource*, utilizando JNDI por exemplo ou funcionalidades específicas de *frameworks* ou APIs. O importante é que o *dataSource* deve fazer referência à origem dos dados (conexão com o banco de dados).

Por debaixo dos panos, o *Spring* controla as transações fazendo chamadas no objeto `java.sql.Connection` recuperado pelo *DataSource*, chamando o método *commit* ou *rollback* de acordo com o resultado da operação. (WALLS, 2011).

6.3 Transações *Hibernate*

Se a persistência na aplicação for feita pelo *Hibernate*, será necessário declarar o seguinte *bean*:

```

<bean id="controleTx"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

```

O *HibernateTransactionManager* delega a responsabilidade de controle de transação para um objeto *org.hibernate.Transaction* que é recuperado da sessão, chamando os métodos *commit* ou *rollback* de acordo com o resultado da operação (WALLS, 2011).

6.4 Transações JPA

O *JpaTransactionManager* necessita apenas do *wiring* com uma implementação da *entity manager factory*. É possível adicionar o suporte para operações simples de JDBC no mesmo *DataSource*, para isso, deve ser feito o *wiring* com uma implementação do *JpaDialect* desde que essa implementação suporte acesso JPA e JDBC misturados (WALLS, 2011), como segue na declaração abaixo:

```
<bean id="jpaDialect"
class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>

<bean id="controleTx"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="jpaDialect" ref="jpaDialect"/>
</bean>
```

6.5 Transações JTA

O *JtaTransactionManager* delega o controle de transação para uma implementação JTA, que especifica uma API padrão para coordenar transações entre uma aplicação e um ou mais *dataSource* (WALLS, 2011). A propriedade *transactionManagerName* especifica um controle de transação JTA para ser resolvido através de JNDI.

```
<bean id="controleTx"
    class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName"
        value="java:/TransactionManager"/>
</bean>
```

6.6 Atributos da transação

No *Spring*, transações declarativas são definidas com atributos de transação. Um atributo de transação é que define como políticas de transação devem ser aplicadas aos

métodos (WALLS, 2011). Existem cinco atributos de transação: propagação, isolamento, regras de *rollback*, *timeout* e *read-only*.

6.7 Propagação

A propagação é o que define os limites da transação de acordo com o cliente e o método sendo invocado (WALLS, 2011) e existem vários comportamentos para propagação:

- *PROPAGATION_MANDATORY*: indica que o método tem que ser executado dentro de uma transação. Caso não haja uma transação em progresso, ocorrerá um erro.
- *PROPAGATION_NESTED*: indica que o método deve ser executado dentro de uma transação aninhada caso haja uma em andamento. A transação aninhada pode ser confirmada ou cancelada individualmente da transação pai.
- *PROPAGATION_NEVER*: indica que o método não deve ser executado dentro de um contexto transacional. Caso haja uma transação em andamento, ocorrerá um erro.
- *PROPAGATION_NOT_SUPPORTED*: indica que o método não deveria ser executado dentro de uma transação. Caso haja uma transação em andamento, ela será suspensa até o fim da execução do método.
- *PROPAGATION_REQUIRED*: indica que o método tem que ser executado dentro de uma transação. Caso haja uma transação em andamento, o método será executado dentro dela, caso contrário, uma nova transação será iniciada.
- *PROPAGATION_REQUIRES_NEW*: indica que o método tem que ser executado dentro de uma transação própria. Uma nova transação é iniciada e se houver uma transação em andamento, ela será suspensa até o fim da execução do método.
- *PROPAGATION_SUPPORTS*: indica que o método não requer um contexto transacional, mas pode executar dentro de uma transação caso haja uma em andamento.

6.8 Níveis de isolamento

O nível de isolamento é que define o quanto uma transação pode ser afetada pela atividade de outras transações concorrentes (WALLS, 2011). Em uma aplicação típica,

transações podem acontecer concorrentemente, mas, ainda de acordo com o mesmo autor, pode acarretar nos seguintes problemas:

- **Leitura suja:** ocorre quando uma transação lê um dado que foi escrito mas que ainda não foi confirmado (*commit*) por outra transação.
- **Leitura não repetida:** ocorre quando uma transação realiza a mesma consulta duas ou mais vezes e cada vez, o resultado é diferente. Normalmente isso se deve a outra transação atualizando os valores entre as consultas.
- **Leitura fantasma:** ocorrem quando uma transação lê vários registros e outra transação insere registros. Para as próximas consultas, a primeira transação irá encontrar registros a mais que não estavam lá anteriormente.

De acordo com Walls (2011), em uma situação ideal, transações devem ser totalmente isoladas uma das outras para evitar esses problemas, mas isolamento perfeito pode afetar o desempenho porque normalmente envolve o *lock* de registros ou tabelas inteiras do banco de dados. Muitas vezes não é necessário um isolamento perfeito, mas sim, é desejável flexibilidade ao tratar isolamento de transações. Para isso, os seguintes níveis de isolamento são possíveis:

- *ISOLATION_DEFAULT*: usa o nível de isolamento padrão do banco de dados.
- *ISOLATION_READ_UNCOMMITTED*: permite a leitura de dados que ainda não foram confirmados. Pode acarretar em leituras sujas, fantasmas e não repetidas. É o nível de isolamento mais eficiente
- *ISOLATION_READ_COMMITTED*: permite leituras de transações concorrentes que foram confirmadas. Leituras sujas são evitadas mas as outras ainda podem ocorrer.
- *ISOLATION_REPEATABLE_READ*: leituras múltiplas do mesmo campo irão ter o mesmo resultado a não ser que seja modificado pela própria transação. Leituras fantasmas ainda podem ocorrer.
- *ISOLATION_SERIALIZABLE*: esse isolamento totalmente compatível com ACID previne todos os problemas de leitura. É o nível de isolamento mais lento uma vez que realiza o *lock* nas tabelas envolvidas na *transação* e portanto, o menos eficiente.

6.9 Transação *Read-Only*

Se uma transação realiza apenas operações de leitura (*read-only*), é possível aplicar otimizações que tiram vantagem desse aspecto. De acordo com Walls (2011) só faz sentido declarar uma transação como *read-only* em métodos que possuem comportamento de propagação que inicia uma nova transação.

6.10 *Timeout* da transação

Para uma aplicação ter um bom desempenho, suas transações não podem levar muito tempo pois pode acarretar no travamento nos recursos de banco de dados. Supondo que uma transação esteja sendo executada por muito tempo, é possível definir que a transação seja cancelada após um certo número de segundos (WALLS, 2011). Também só faz sentido declarar *timeout* em métodos que criam uma nova transação.

6.11 Regras de *rollback*

De acordo com Walls (2011), as regras de *rollback* são o que define para quais exceções ocorre o *rollback* e para quais não ocorre. Por padrão, transações são canceladas apenas por exceções que ocorrem em tempo de execução (*RuntimeException* em Java). Mas é possível mudar esse comportamento para ocorrer o *rollback* em exceções checadas (*CheckedExceptions*) ou não ocorrer o *rollback* em *RuntimeException*.

6.12 Controle de transação através de anotações

Para utilizar a anotação de controle de transação do *Spring*, é necessário adicionar o seguinte *namespace* ao XML de configuração:

```
<tx:annotation-driven transaction-manager="controleTx"/>
```

Esse elemento diz ao *Spring* para fazer uma busca entre todos os *beans* no contexto da aplicação e procurar quais estão anotados com *@Transactional* (em classes ou métodos) que será configurada devidamente com seus atributos que são argumentos da anotação. Por exemplo, a fachada de negócio para finalizar um pedido:

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class PedidoFacadePojo implements PedidoFacade {
```

```
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void finalizarPedido(Pedido p) {
    //Logica para salvar pedido e dar baixa em estoque omitida
}
}
```

No nível de classe, foi anotado que todos os métodos irão suportar transação e serão apenas leitura. No método `finalizarPedido` foi anotado que é necessária uma transação e não será apenas leitura.

7 IMPLEMENTAÇÃO DO SPRING

Neste exemplo, será usado o *framework web Stripes*. O acesso ao banco de dados será omitido uma vez que foge do escopo deste trabalho, mas é possível utilizar o *Spring* em aplicações *Java desktop*, *mobile* ou em conjunto com outros *frameworks* parecidos como: *Struts*, *Hibernate*, *MyBatis* ou *JPA*. Porém, o *Stripes* é nativamente integrado ao *Spring*, aumentando a produtividade e diminuindo a necessidade de mais arquivos XML de configuração, assim também como o próprio *framework MVC* do *Spring*.

7.1 Arquivos XML de configuração

Uma vez tendo o ambiente configurado com as dependências de todos os *frameworks* a serem utilizados, inclusive as do *Spring* (o módulo *core* basta para este exemplo), iremos criar os arquivos XML de configuração, que irá criar e realizar o *wiring* dos *beans* do *Spring*.

Neste exemplo, o *pool* de conexões com o banco de dados será gerenciado pelo *container (Tomcat)*, utilizando JNDI.

Serão utilizadas duas abordagens: o *wiring* através de XML e o *wiring* através de *annotations* a fim de exemplificação. Pode-se escolher somente uma delas mas o *Spring* requer um mínimo de configuração via XML, não sendo possível abolir totalmente o mesmo.

A distribuição dos *beans* em diferentes arquivos XML de acordo com a camada ou módulo do projeto pode resultar em uma aplicação melhor organizada e fracamente acoplada, melhorando assim a produtividade ao dar manutenção no código ou crescer o projeto. Neste exemplo, iremos assumir que o projeto está dividido em diferentes camadas: a camada de apresentação ou a aplicação *web* em si (provida pelo *Stripes*), a camada de serviço (fachadas, objetos POJOs simples) e a camada de acesso ao banco (*iBatis* por exemplo), resultando assim em três arquivos XMLs distintos: o *applicationContext.xml*, o *servicesBeans.xml* e o *daosBeans.xml*, respectivamente relacionados às camadas. É importante notar que todos os XMLs devem estar no *classpath* do projeto que os utilizam.

Caso fosse utilizado apenas um arquivo XML, o conteúdo de todos os outros deveriam ser adicionados ao *applicationContext.xml*, uma vez que é ele quem vai ditar as regras de *wiring* e declaração dos *beans*.

Começando pelo *applicationContext.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<context:annotation-config/>
<!--Apenas para wiring atraves de annotations - começo -->
<aop:aspectj-autoproxy/>
<context:component-scan base-package="br.com.tcc.service"/>
<bean
class="org.springframework.beans.factory.annotation.AutowiredAnnotationBean
PostProcessor"/>
<!--Apenas para wiring atraves de annotations - fim-->
<tx:annotation-driven transaction-manager="txManager"/>
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
<property name="jndiName" value="java:comp/env/jdbc/conexaoDB"/>
<property name="resourceRef" value="true" />
</bean>

<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
scope="singleton">
<property name="dataSource" ref="dataSource"/>
</bean>

<import resource="classpath:/servicesBeans.xml"/>
<import resource="classpath:/daosBeans.xml"/>

</beans>

```

Agora, é necessário criar as classes e o arquivo XML da camada de serviço (fachada). É uma boa prática sempre programar orientado a interfaces, principalmente quando se utiliza o *Spring*, gerando assim um menor nível de acoplamento entre os módulos do projeto, facilitando migrações posteriores caso haja necessidade, sem ter que reescrever a aplicação inteira, desde a camada de apresentação.

A classe pedido, de forma simples apenas para exemplificar, assumindo que o pedido tenha apenas um código identificador, uma descrição e um valor:

```
package br.com.tcc.services.entidades;
import java.math.BigDecimal;

public class Pedido {
    private Integer idPedido;
    private String descricao;
    private BigDecimal valorFinal;

    public void setIdPedido(Integer idPedido) {
        this.idPedido = idPedido;
    }

    public Integer getIdPedido() {
        return this.idPedido;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public String getDescricao() {
        return this.descricao;
    }

    public void setValorFinal(BigDecimal valorFinal) {
        this.valorFinal = valorFinal;
    }

    public BigDecimal getValorFinal() {
        return this.valorFinal;
    }
}
```

Agora, as interfaces:

```
package br.com.tcc.services.facades;
import br.com.tcc.services.entidades.Pedido;
```

```

public interface PedidoFacade {

    Pedido buscaPedido(Integer idPedido);

    void finalizarPedido(Pedido p);

}

package br.com.tcc.services.facades;
import br.com.tcc.services.entidades.Pedido;

public interface LogFacade {
    void registraPedido(Pedido p);
}

```

As classes que implementarão as interfaces, (a declaração dos DAOs será omitida):

```

package br.com.tcc.services.facades.impl;
import org.springframework.transaction.annotation.Transactional;
import br.com.tcc.daos.PedidoDAO;
import br.com.tcc.services.entidades.Pedido;
import br.com.tcc.services.facades.LogFacade;

@Service("pedidoFacadeBean")
@Scope("singleton")
public class PedidoFacadePojo implements PedidoFacade {

    @Autowired
    @Qualifier("pedidoDAOBean")
    private PedidoDAO pedidoDAO;

    @Autowired
    @Qualifier("logFacadeBean")
    private LogFacade logFacade;

    @Transactional(readOnly=true)
    public Pedido buscaPedido(Integer idPedido) {
        return pedidoDAO.buscaPK(idPedido);
    }

    @Transactional
    public synchronized void finalizaPedido(Pedido p) {

```

```

        this.pedidoDAO.inserere(p);
        this.logFacade.registraPedido(p);
    }

    public PedidoDAO getPedidoDAO() {
        return this.pedidoDAO;
    }

    public void setPedidoDAO(PedidoDAO pedidoDAO) {
        this.pedidoDAO = pedidoDAO;
    }

    public LogFacade getLogFacade() {
        return this.logFacade;
    }

    public void setLogFacade(LogFacade logFacade) {
        this.logFacade = logFacade;
    }
}

package br.com.tcc.services.facades.impl;
import br.com.tcc.services.entidades.Pedido;
import br.com.tcc.services.facades.LogFacade;
import br.com.tcc.daos.LogPedidoDAO;
import org.springframework.transaction.annotation.Transactional;

public class LogFacadePojo implements LogFacade {

    private LogPedidoDAO logPedidoDAO;

    @Transactional
    private synchronized void registraPedido(Pedido p) {
        logPedidoDAO.inserereLogPedido(p);
    }

    public LogPedidoDAO getLogPedidoDAO() {
        return this.logPedidoDAO;
    }

    public void setLogPedidoDAO(LogPedidoDAO logPedidoDAO) {
        this.logPedidoDAO = logPedidoDAO;
    }
}

```

```
}
```

Agora será criado o arquivo `servicesBeans.xml`, que conterà os *beans* das fachadas, exceto da classe `PedidoFacadePojo`, uma vez que esta utiliza *wiring* e declaração de *beans* através de *annotations*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

    <bean id="logFacadeBean"
class="br.com.tcc.services.facades.impl.LogFacadePojo", scope="singleton">
        <property name="logPedidoDAO">
            <ref bean="logPedidoDAOBean"/>
        </property>
    </bean>
</beans>
```

E, por fim, o arquivo `daosBeans.xml`, lembrando que a implementação das classes foi omitida:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

```

<bean          id="pedidoDAOBean"          class="br.com.tcc.daos.GenericoDAO"
scope="singleton">
    <constructor-arg>
        <value>br.com.tcc.daos.PedidoDAO</value>
    </constructor-arg>
</bean>

<bean          id="logPedidoDAOBean"        class="br.com.tcc.daos.GenericoDAO",
scope="singleton">
    <constructor-arg>
        <value>br.com.tcc.daos.LogPedidoDAO</value>
    </constructor-arg>
</bean>

</beans>

```

É possível notar que, onde é necessário utilizar uma fachada, a mesma é acessada e injetada através de sua interface, caso haja necessidade de corrigir algum defeito ou realizar uma migração de tecnologia por exemplo, apenas o módulo necessário é alterado, mantendo o resto do sistema que já funcionava, intacto.

7.2 A camada de apresentação

Como já mencionado, neste exemplo, será utilizado o *Stripes* como o *framework web* para a camada de apresentação. Ele gerencia as páginas *webs* através das *ActionBeans*. A cada nova requisição feita para o servidor, uma nova *ActionBean* é instanciada para responder essa requisição.

Em um projeto que não utiliza o *Spring*, supondo que ele receba mil requisições, serão instanciados mil *ActionBeans*, e caso essa *ActionBean* instanciada precise acessar a camada de negócio (neste exemplo, as fachadas), serão criados também mil fachadas, cada uma com mil vezes o numero de parâmetros que ela recebe, que por sua vez instancia mil DAOs e assim por diante. É fácil notar como o problema escala e foge do controle, perdendo assim confiabilidade e previsibilidade do *software*.

Porém, utilizando o *Spring*, apenas serão instanciadas as mil *ActionBeans*, uma vez que as fachadas e os DAOs estão definidos como *singleton*, ou seja, para o projeto inteiro, existirá apenas uma instância da fachada X e uma instância do DAO Y, que serão responsáveis em responder todas essas mil requisições.

Porém esse comportamento pode gerar acesso concorrente ao banco de dados ou aos métodos, por isso os métodos que não podem sob hipótese alguma serem acessados concorrentemente, é saudável definí-los como *synchronized*, minimizando o problema.

Foi possível observar também que em nenhum momento é necessário instanciar fachadas, DAOs, *beans* do *Spring* em modo geral. O próprio container vai se encarregar de instanciá-los e colocar as referências nos lugares onde são necessários, ou seja, a inversão de controle (injeção de dependências), na prática, simples e fácil.

7.3 As *ActionBeans*

Agora será declarada a classe *CheckoutActionBean*, que basicamente irá finalizar o pedido do cliente, apenas para demonstrar a integração com o *Spring* (implementação da *BaseActionBean* omitida):

```
package br.com.tcc.web.actions;
//Imports omitidos

@UrlBinding("/checkout.do")
public class CheckoutActionBean extends BaseActionBean {

    @SpringBean("pedidoFacadeBean")
    private PedidoFacade pedidoFacade;

    private Pedido pedido;

    public Resolution exhibeCheckout() {
        this.pedido = pedidoFacade.buscaPedido(this.pedido.getIdPedido());
        return new ForwardResolution("/WEB-INF/jsp/checkout.jsp");
    }

    public Resolution finalizaPedido() {
        this.pedidoFacade.finalizaPedido(pedido);
        return new RedirectResolution(AgradecimentoActionBean.class);
    }

    public PedidoFacade getPedidoFacade() {
        return this.pedidoFacade;
    }
}
```

```

public void setPedidoFacade(PedidoFacade pedidoFacade) {
    this.pedidoFacade = pedidoFacade;
}

public Pedido getPedido() {
    return this.pedido;
}

public void setPedido(Pedido pedido) {
    this.pedido = pedido;
}
}

```

Agora, toda vez que uma nova `CheckoutActionBean` for instanciada, o *Spring*, automaticamente irá colocar uma referência para a instância da classe `PedidoFacade` que foi criada na inicialização do container, apenas utilizando a anotação *SpringBean*, que é do *Stripes*, não do *Spring*.

Porém, o *Spring*, só consegue realizar o *wiring* de um *bean* dentro de outro *bean*, através dos *getters* e *setters* ou através de métodos construtores. Isso significa que ele não consegue realizar o *wiring* de um *bean* em uma classe Java que não esteja declarada como um *bean*. Mas como o *Spring* consegue saber que tem que realizar esse *wiring* na inicialização do contexto?

7.4 Context Loader Listener

Neste exemplo, como é uma aplicação *web*, quem vai dizer para o *Spring* qual arquivo XML ele tem que ler para realizar o *wiring* e quando ele tem que fazer isto, é o Context Loader Listener, configurado no `web.xml` da aplicação:

```

<!--Restante do arquivo omitido -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener

```

```

    </listener-class>
</listener>

```

Desta forma, o *Spring* sabe qual arquivo deve ler para realizar o *wiring* do que e onde, assim que o *Container* iniciar o contexto da aplicação. Caso tenha mais de um projeto no container, deve existir mais de um applicationContext.xml, logo, a mesma quantidade de instâncias de *beans* será inicializada.

Uma outra maneira de pegar a referência de um *bean* que o *Spring* inicializa, é a seguinte:

```

// Restante da declaração omitida
private ApplicationContext context;

private PedidoFacade pedidoFacade = (PedidoFacade)
this.context.getBean("pedidoFacadeBean");

@Override
public void setApplicationContext(ApplicationContext arg0) throws
BeansException {
    this.context = arg0;
}

```

8 CONCLUSÃO

Conforme foi demonstrado, o *framework Spring* consegue facilitar muito a implementação de muitas funcionalidades que só poderiam ser alcançadas com EJB ou com grande esforço de código, de forma mais simples, clara, objetiva e produtiva.

Sua grande capacidade de integração permite que vários projetos tirem proveito do *Spring*, o *framework* de desenvolvimento *web Stripes*, por exemplo, vem com a integração nativa com *Spring*, a fim de tirar proveito da injeção de dependências na camada de apresentação. Nas camadas inferiores da aplicação também é possível realizar a integração, seja o *framework* utilizado *Hibernate*, *MyBatis*, *JTA*, *JDBC*, entre outros, com controle de transação declarativa, sem ficar dependente de um *container* EJB.

Dessa forma, na camada de serviço do *software*, seus diversos módulos (fachadas) podem ser declarados como *beans singletons*, isso é, para a aplicação inteira, só existirão uma única instância de cada módulo, permitindo um uso de recurso computacional mais previsível e escalonável, uma vez que esses módulos serão instanciados uma única vez quando o contexto da aplicação for iniciado. Como a instanciação de objetos é altamente custoso em Java, ganha-se no gerenciamento e escalonamento de recursos computacionais. Tudo isso de forma simples e produtiva, sem que o programador precise reinventar a roda.

A utilização do *Spring* e seus diversos módulos complementares pode trazer um grande ganho de produtividade e de qualidade para projetos por ser de fácil aprendizado, implementação e altamente produtivo através de anotações.

REFERÊNCIAS

WALLS, C. **Spring in Action**. 3.ed. Manning Publications, 2011.

JOHNSON, R; HOELLER, J. **Expert one-on-one J2EE Development without EJB**. 1.ed. Wiley Publishing, 2004.

JOHNSON R; et. al. **Spring 3.1 Reference Documentation**. 1.ed. Spring Source, 2011.

AHMED, K. Z.; UMRYSH, C. E. **Developing Enterprise Java Applications with J2EE(TM) and UML**. 1.ed. Addison-Wesley Professional, 2001.

BALL, J. ; EVANS, I. J. **Your First Cup: An Introduction to the Java EE Platform**. 2007.
Disponível em: <<http://download.oracle.com/javaee/5/firstcup/doc/docinfo.html> >. Acesso em: 19 jan. 2011.