Injeção de Dependências com Spring Framework 3

O objetivo deste guia é expor o porquê, como usar e quais as melhores práticas relacionadas ao container de Injeção de Dependências (*Dependency Injection, ou DI*) do Spring Framework 3.0. O assunto pode interessar o leitor pelas seguintes razões:

- Trata-se do núcleo do Spring Framework: sendo assim é conhecimento obrigatório para todos aqueles que desejem usá-lo ou seus derivados.
- Ainda mais importante: é um padrão de projeto que aumenta de forma significativa a qualidade dos sistemas nos quais é aplicado. Mesmo que o leitor não se interesse pelo Spring Framework, poderá usá-lo em seus trabalhos adotando outros containeres como Pico, miocc ou mesmo o seu próprio se for o caso.

Caminhando para a Injeção de Dependências

Um projeto de software ruim costuma apresentar três características: rigidez, fragilidade e imobilidade. Frágil é aquele sistema no qual ao se introduzir uma modificação nos deparamos com efeitos inesperados em outros de seus componentes. A rigidez é conseqüência desta fragilidade: dada a dificuldade em se manter o sistema o desenvolvedor vê suas opções de atuação limitadas (ou mesmo anuladas). Já a imobilidade diz respeito à dificuldade em se extrair partes do sistema que possam ser reaproveitadas em outros projetos.

Estes três problemas possuem a mesma causa: o alto acoplamento. Um sistema é considerado de alto acoplamento quando seus componentes internos são fortemente dependentes. Por componentes devem ser entendidas classes, fontes de dados, sistemas externos ou internos.

Para melhor entender o problema vamos usar um exemplo bastante trivial que será um software que busque títulos de livros escritos por determinado autor. Na **imagem 1** podemos ver sua estrutura inicial.

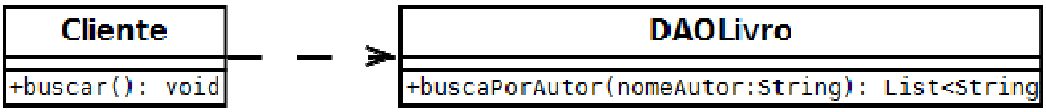

Imagem 1 - Nosso modelo inicial

Na classe cliente temos um único método chamado *buscar()*, que faz uma chamada à função *buscaPorAutor* da classe DAOLivro que encontra títulos de livros escritos pelo nome do autor passado como parâmetro. Imaginemos por um momento que DAOLivro usa como fonte de dados um arquivo texto.

Por anos este sistema poderá funcionar sem problemas. Mas e se no futuro fosse necessário usar alguma outra fonte de dados, digamos, um banco de dados relacional? Podemos imaginar duas soluções que não usem orientação a objetos para resolver o problema:

- Na classe DAOLivro poderíamos criar um novo método chamado *buscaPorAutorNoSGBD*, que faria a consulta usando um banco de dados relacional. Não é uma solução interessante, pois estaríamos aumentando a complexidade da classe incluindo uma nova função que, do ponto de vista das classes cliente, exerceria a mesma função de *buscaPorAutor*.
- Também podemos alterar a função *buscaPorAutor* incluindo um novo parâmetro no qual pudéssemos definir qual a fonte de dados a ser usada. De novo cairíamos no mesmo problema da solução anterior.

Dado que estamos lidando com a plataforma Java e esta é em sua essência orientada a objetos, uma solução mais interessante é adotar o padrão factory. A **imagem 2** expõe um segundo modelo proposto para nosso sistema.

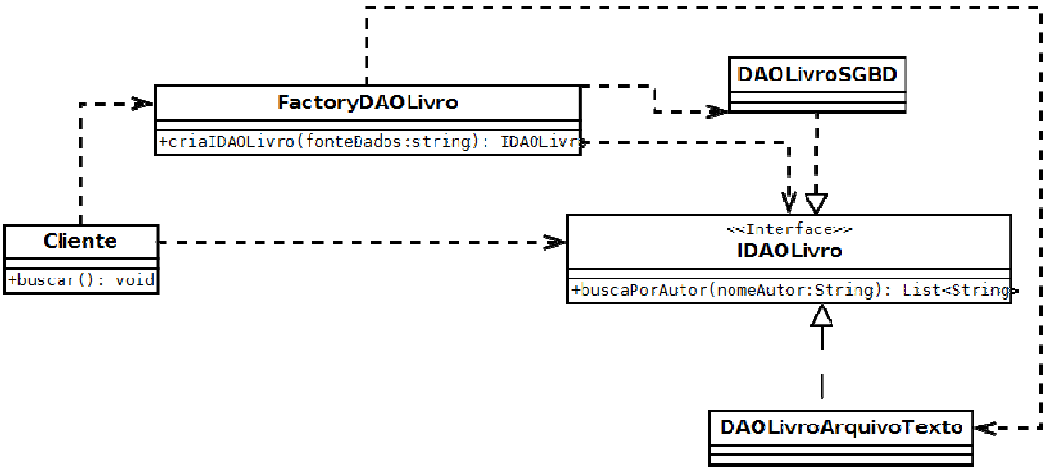

Imagem 2 - Aplicando o padrão Factory

O que é: Padrões de Projeto

O conceito de padrões de projeto foi cunhado por Cristopher Alexander no contexto da arquitetura civil e posteriormente adaptado para o desenvolvimento de software orientado a objeto[1]. Trata-se de soluções para problemas recorrentes na criação de software que possuem soluções padronizadas e fornecem um vocabulário em comum para todos aqueles que os aplicam. Um exemplo de padrão de projeto é o *Factory*, descrito neste artigo.

O que é: Padrão de Projeto Factory

O padrão factory fornece uma interface para criação de famílias de objetos correlatos ou dependentes. Uma possível implementação do padrão é a que apresentamos neste artigo: cria-se uma classe cujo único objetivo é criar novas instâncias de objetos que implementem determinada interface, fazendo com que apenas uma única classe do sistema venha a ter dependências diretas com todas as implementações presentes de determinada classe ou interface.

Nesta nova arquitetura a classe Cliente depende agora da interface IDAOLivro e da classe FactoryDAOLivro, a segunda responsável por instanciar a implementação correta de IDAOLivro a partir do nome da fonte de dados fornecida pela classe Cliente ao chamar a função *criarIDAOLivro*. A solução adotada apresenta vantagens bastante interessantes:

- A classe Cliente é dependente apenas da interface IDAOLivro, e não de uma implementação específica. Sendo assim agora é possível trabalhar com qualquer fonte de dados imaginável, desde que implemente a interface IDAOLivro.
- A responsabilidade pela instanciação de objetos que implementem a interface IDAOLivro fica centralizada em uma única classe, reduzindo significativamente a complexidade do sistema.

[1] O termo se consagrou na disciplina de engenharia de software com o livro "Padrões de Projeto" de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (os quatro autores também são conhecidos como a *Gangue dos Quatro* (Gang of Four ou GoF em inglês)

O que vimos no segundo modelo é na realidade a aplicação do princípio de *inversão de dependências* cunhado por Robert C. Martin em 1996 no seu artigo *The Dependency Invertion Principle*[2]. Este princípio nos diz duas coisas:

- Um módulo de nível mais alto jamais deve depender diretamente de outro pertencente a um nível inferior, mas sim de uma abstração.
- Abstrações não devem depender de detalhes de implementação, mas sim o contrário

Em nosso exemplo a classe *Cliente* é um módulo de nível superior: não lhe interessa como os títulos dos livros são obtidos, mas sim que os mesmos sejam retornados de alguma forma para que possa concluir o seu trabalho. No primeiro modelo, esta dependia diretamente de uma implementação específica (o arquivo no formato textual). Já na segunda, passa a depender apenas de uma abstração que, no caso, é a nossa interface IDAOLivro.

O segundo postulado também foi respeitado no segundo modelo. Observe que as distintas implementações de IDAOLivro não influenciam de forma alguma esta interface, mas sim o contrário. No caso, todas as implementações específicas de IDAOLivro seguem rigidamente as regras definidas pela interface que, no caso, trata-se da obrigatoriedade de se implementar a função *buscaPorAutor*.

Observe que as três características presentes em um design ruim não se aplicam no segundo modelo. Nosso software não é mais frágil, pois caso seja feita qualquer alteração em uma implementação específica da interface *IDAOLivro* não afetará diretamente a classe *Cliente*. Como conseqüência, o sistema agora é mais fácil de ser mantido (não há rigidez) e, ainda mais interessante: obtivemos mobilidade, pois podemos agora transportar com maior facilidade elementos do nosso sistema para outros projetos.

Mas este modelo ainda está longe do ideal, porque aumentamos o grau de acoplamento da classe *Cliente*. Enquanto na primeira versão sua única dependência era a classe *DAOLivro*, no segundo incluímos outra que é a classe *FactoryIDAOLivro*. É hora de apresentar uma terceira versão do sistema, cuja arquitetura é exposta na **imagem 3**.

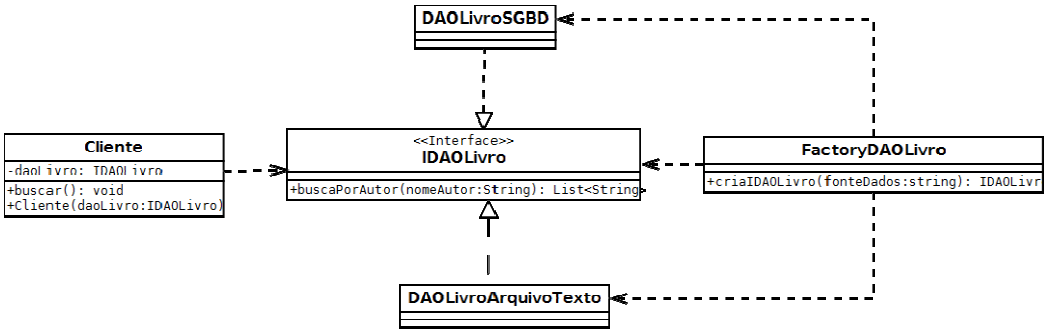

Imagem 3 - Terceira versão do nosso modelo

A mudança foi sutil: apenas explicitamos a presença de um atributo *daoLivro* na classe *Cliente* do tipo *IDAOLivro*, cuja instância é definida pelo construtor da classe *Cliente*[3]. Agora podemos dizer que atingimos o menor grau de acoplamento possível para esta classe.

Porém, por mais que nos esforcemos, se continuarmos seguindo esta linha de raciocínio sempre nos depararemos com problemas de acoplamento. Basta observar a classe *FactoryDAOLivro*, que possui como dependência todas as implementações presentes no sistema da interface *IDAOLivro*. É quase certo que conforme este sistema evolua chegaremos a um momento no qual a classe *FactoryDAOLivro* se tornará um

[2] Este belíssimo artigo encontra-se acessível gratuitamente no endereço http://www.objectmentor.com/publications/dip.pdf

[3] Poderíamos também ter criado um setter para este atributo, porém o efeito seria o mesmo.

monstro de difícil manutenção. A cada nova implementação de IDAOLivro precisaremos alterá-la e compilá-la novamente.

Além deste problema, temos outro: é necessária uma classe que crie uma instância de *Cliente*. Esta, por sua vez, possui o mesmo problema apresentado no modelo 2: duas dependências. Resumindo: chegamos ao limite do que podemos fazer usando apenas o padrão *Factory*.

Entra a Injeção de Dependências

A injeção de dependências (*Dependency Injection* ou *DI* em inglês) é uma forma de inversão de controle na qual o aspecto a ser invertido é, como o próprio nome já diz, a resolução de dependências. Ao invés do desenvolvedor definir manualmente quais as instâncias a serem criadas e injetadas, como fizemos nos três modelos anteriormente descritos, delega-se esta tarefa a um container de inversão de controle (IoC) – veja o quadro *Injeção de Dependências ou Inversão de Controle?* -, evitando assim a necessidade de se implementar o padrão factory.

Grosso modo, podemos pensar no container de IoC como um factory ideal, capaz de instanciar qualquer tipo de classe sem que fiquemos presos a uma família específica de objetos. Porém, como veremos, no caso do Spring este "factory" vai além à medida que também controla o ciclo de vida dos objetos por ele gerenciados. Para introduzirmos o conceito, imagine que este container é uma classe "mágica" que sempre nos retorna um objeto cujo tipo é exatamente aquele que estamos precisando. O "truque" por trás desta "mágica" será exposto no decorrer deste artigo.

O que é uma dependência?

É muito raro encontrarmos um software real e útil composto por uma única classe. Na esmagadora maioria das vezes topamos com classes que possuem atributos cujo tipo são outras classes. Estes atributos que referenciam outros tipos de classe são o que costumamos chamar de dependência.

Injeção de Dependências ou Inversão de Controle?

Se o núcleo do Spring é chamado de Container de Inversão de Controle (*IoC de Inversion of Control* em inglês), porque estamos usando o termo *injeção de dependências*? Porque como bem observou Martin Fowler em seu artigo "*Inversion of Control Containers and the Dependency Injection Pattern*" [4], os containeres de inversão de controle como Spring, Pico e outros aplicam um tipo muito específico de inversão de controle.

E neste momento uma segunda pergunta surge: o que é inversão de controle? Chama-se de inversão de controle a técnica de desenvolvimento na qual um aspecto comportamental do sistema é delegado a uma entidade externa.

Um exemplo de inversão de controle é o EJB. O desenvolvedor implementa o seu EJB e em seguida faz o deploy no servidor de aplicações sem precisar se preocupar com o modo como seu componente será iniciado, finalizado e gerenciado, porque neste caso é tudo feito pelo servidor. Ao invés de nos preocuparmos com estas tarefas as *delegamos* a uma entidade externa, que normalmente recebe o nome de *container*.

A inversão de controle esta no núcleo do conceito de framework. Ao adotarmos um framework, estamos reutilizando a estrutura de uma aplicação semi-pronta que finalizamos ao inserir nossos próprios componentes

[4] O artigo pode ser online neste endereço: http://www.martinfowler.com/articles/injection.html

customizados. No caso de aplicações web este comportamento fica nítido quando nos lembramos do processo de preenchimento de formulários, que é *delegado* ao framework ao invés de ser implementado manualmente pelo desenvolvedor.

Após estes exemplos fica nítido, portanto que o termo *Inversão de Controle* é genérico demais para ser adotado no caso do Spring, aonde o foco é a resolução das dependências das classes de nosso projeto.

Voltando ao nosso exemplo inicial, ao aplicarmos o padrão de injeção de dependências acabaríamos com uma arquitetura similar à exposta na **imagem 4**. Observe que o Container não possui nenhuma dependência direta com nenhuma das classes do nosso sistema. Outro fator interessante a ser observado é a ausência da classe *FactoryDAOLivro*.

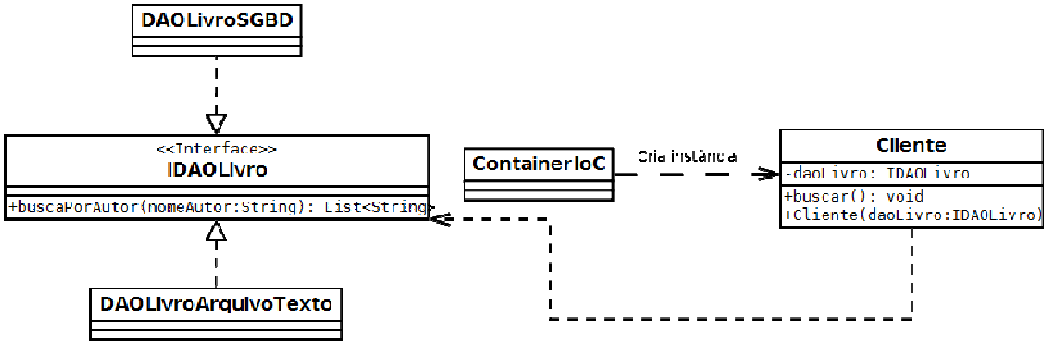

Imagem 4 - Nosso modelo após aplicar Injeção de Dependências

No novo modelo todo o processo de instanciação e preenchimento de dependências é feito pelo container, que é instruído a fazê-lo a partir de configurações que podem estar nos mais variados formatos, desde arquivos no formato XML até anotações em código fonte. A partir destes dados e usando o mecanismo de reflexão provido pelo ambiente computacional o container é capaz de instanciar e injetar todas as dependências que nós o instruamos a fazê-lo.

Uma das grandes vantagens deste modelo é que neste estágio o desenvolvedor se encontra próximo do desacoplamento quase total entre seus componentes. Como conseqüência, temos sistemas com qualidade muito superior, mais fáceis de manter e distribuir nas mais variadas situações.

Caso surja a necessidade de uma nova implementação de alguma das abstrações presentes no sistema, tudo o que o desenvolvedor precisa fazer é implementá-la, distribuí-la como um arquivo. jar (ou .class mesmo), incluir a implementação no classpath do sistema e em seguida alterar o(s) arquivo(s) de configuração do container sem a necessidade de recompilar o sistema para se adequar a esta nova situação. Obtém-se desta maneira mobilidade total do sistema, visto que com o núcleo pronto e estabilizado o processo de recompilação se torna uma tarefa muito mais rara de ser executada.

Spring Framework entra em cena

O objetivo do Spring Framework é simplificar o desenvolvimento de aplicações

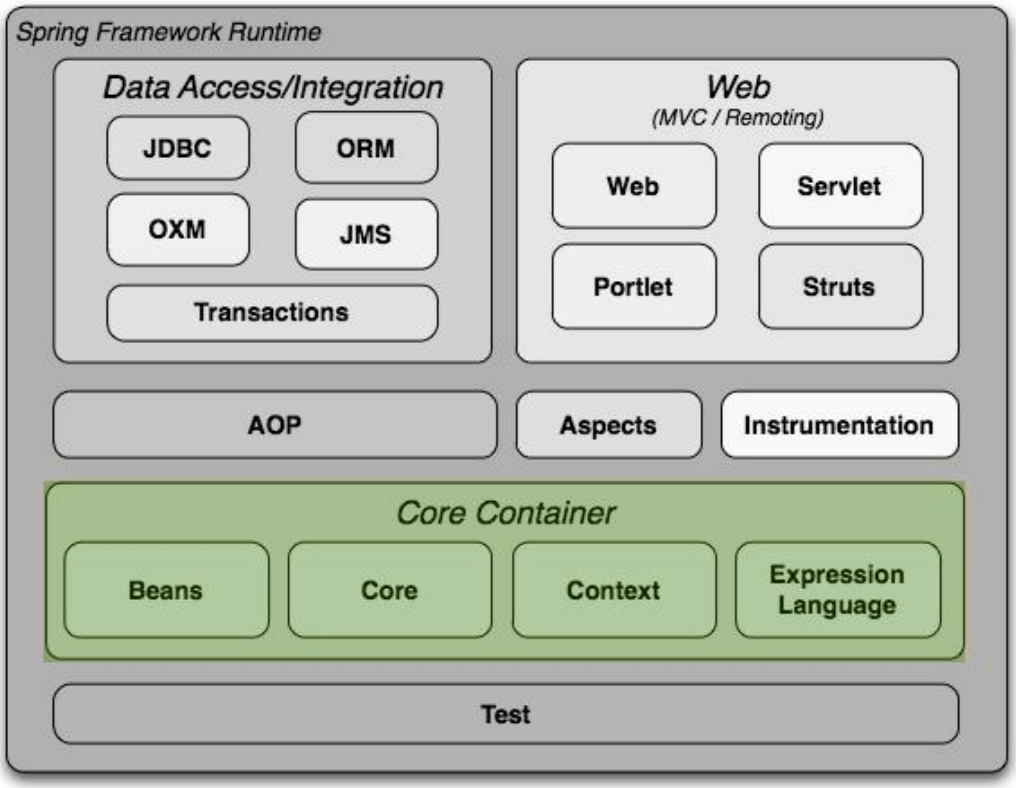

Imagem 5 Componentes do Spring Framework. Nosso foco é no Core Container

corporativas. Na **imagem 5** podemos ver todos os módulos que compõem o framework, que é uma alternativa viável à plataforma Java Enterprise Edition. Este incrível framework possibilitou usando apenas POJO's obter os mesmos resultados que até então só eram possíveis com EJB. E tudo isto graças à aplicação do conceito de injeção de dependências!

Spring pode fazer inúmeras coisas, mas quando nos focamos apenas no seu núcleo, percebemos que estamos lidando na realidade com um container de injeção de dependências e programação orientada a aspectos leve. Para melhor entender a sua definição, convém estudar suas partes.

Por que leve? Spring é não intrusivo. Se sua aplicação é baseada no Spring, é possível que suas classes não venham a ter qualquer dependência direta com o framework. Nada mais natural, visto que o problema do alto acoplamento é justamente o que o conceito de injeção de dependências visa resolver.

Programação orientada a aspectos? O Spring oferece desde sua primeira versão suporte a programação orientada a aspectos. Esta é uma técnica poderosa que permite ao desenvolvedor separar a lógica de negócios de serviços de infra-estrutura presente em nossa aplicação. É importante mencionar esta característica, visto que se trata do outro pilar do framework além do container de injeção de dependências.

Porém nosso foco é o container de injeção de dependências. Sendo assim, iremos tratar aqui apenas do módulo *Core Container* do Spring. Como será visto, é uma ferramenta que consegue ser ao mesmo tempo extremamente poderosa e simples de se usar. E esta é a beleza deste container.

O container de injeção de dependências

Chegou o momento de colocar toda esta teoria em prática. Para usar o container de injeção do Spring é necessário baixar a última versão do framework em seu site oficial[5]. Para este artigo será usada a versão 3.0.2. O Spring é um framework extremamente modular. Isto quer dizer que não precisamos usar todos os arquivos JAR que acompanham a distribuição.

As dependências do Container

Iremos usar apenas os arquivos que compõem o *Core Container* do Spring. Na distribuição oficial estes são nomeados seguindo o seguinte padrão: org.springframework.[nome do módulo].[número da versão].RELEASE.jar. Sendo assim, o arquivo org.springframework. beans.3.0.2.RELEASE.jar por exemplo é na realidade o módulo beans.

Precisaremos, portanto dos módulos *beans, context, context.support, core e expression*. A única dependência externa do container é a biblioteca Commons Logging, que pode ser obtida em seu site oficial[6]. Deverá ser adicionado o arquivo commons-logging-[número da versão].jar em seu classpath.

Como o container funciona

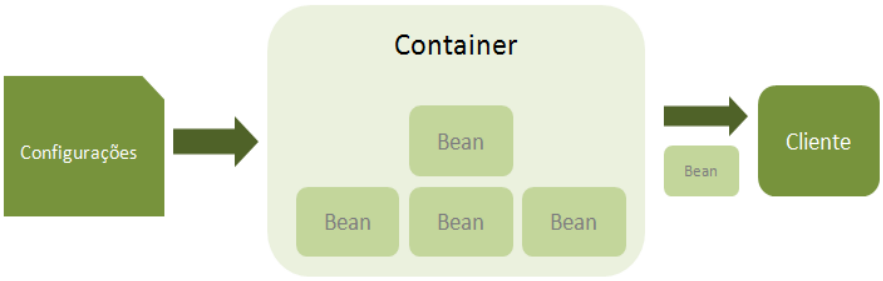

Imagem 6 Funcionamento de um container

Para que o container seja útil, primeiro precisamos configurá-lo. No caso do Spring, suas configurações podem vir de basicamente três fontes: arquivos no formato XML (que é a opção mais comum), anotações em código fonte ou programaticamente[7].

Uma vez alimentado com estas configurações, o container irá armazenar internamente representações dos objetos a serem gerenciados. Estas representações definem quais classes devem ser instanciadas, suas interdependências. Pense nas configurações como uma *receita* a ser seguida pelo Spring quando for necessário instanciar um bean.

Estas "receitas de beans" no entanto dizem mais do que apenas quais classes instanciar e suas interdependências: definem também o escopo de cada bean. Escopos especificam qual deve ser o tempo de vida, ou seja, quando um bean já instanciado deverá ser removido do container. De imediato o Spring nos fornece 5 tipos de escopo: *singleton*, *prototype*, *request*, *session* e *global session*. Destes, apenas trataremos dos dois primeiros, visto que os demais são usados no desenvolvimento de aplicações web, o que foge do foco do nosso artigo. Na **tabela 1** é possível conhecer a descrição destes 5 escopos.

[5] http://www.springframework.org
[6] http://commons.apache.org/logging/ - Na escrita deste artigo a última versão disponível é a 1.1.
[7] O núcleo do Spring é totalmente desacoplado de uma fonte específica de configurações. Sendo assim o desenvolvedor pode criar os seus próprios parsers de configuração caso seja de seu interesse.

Tabela 1 Escopos do Spring

Escopo	Descrição
singleton	Trata-se do escopo default do Spring. O container mantém uma única instância do bean definido.
prototype	O Spring sempre retorna uma nova instância do bean caso o mesmo seja requerido.
request	Usado em aplicações web, define que a instância do bean possuirá o mesmo tempo de vida que uma requisição HTTP
session	Adotado em aplicações web, define que a instância do bean se encontrará armazenada na sessão do usuário, sendo imediatamente destruído junto com esta.
global session	Também usado em aplicações web, armazena a instância do bean em uma sessão global.

Configurado e pronto para o uso, entra a classe cliente, que simplesmente fará requisições ao container do Spring por um bean específico.

Instanciando o Container

O Spring oferece dois tipos de container: *BeanFactory* e *ApplicationContext*. Nossa primeira boa prática portanto diz respeito a qual tipo deverá ser selecionado.

BeanFactory é na realidade uma interface, presente no pacote *org.springframework.beans.factory* que representa o container mais simples fornecido pelo framework, oferecendo apenas suporte a injeção de dependências e gerenciamento de ciclo de vida dos beans.

O framework já vem com algumas implementações desta interface, sendo a mais usada org.springframework.beans.factory.xml.XmlBeanFactory, que lê as configurações do container a partir de um documento XML.

Normalmente opta-se por usar o *BeanFactory* quando o ambiente computacional oferece restrições mais incisivas como por exemplo um Applet.

Já *ApplicationContext*, que também é uma interface presente no pacote *org.springframework.context* é uma extensão de *BeanFactory* e oferece todos os serviços deste e mais alguns como internacionalização, extensibilidade e gerenciamento de eventos. As implementações mais usadas desta interface são *ClassPathXmlApplicationContext* e *FileSystemXmlApplicationContext*. Ambas consomem suas configurações a partir de documentos no formato XML e encontram-se no pacote *org.springframework.context.support*. A diferença entre uma e outra é apenas aonde os documentos XML se encontram. Enquanto a primeira o busca no classpath da aplicação a outra o encontra no sistema de arquivos no qual a aplicação é executada.

Na esmagadora maioria das vezes você acabará usando *ApplicationContext*, a não ser que possua excelentes razões para não fazê-lo. Sendo assim, no transcorrer deste artigo usaremos apenas *ApplicationContext*.

Há uma diferença de comportamento entre *BeanFactory* e *ApplicationContext*. No caso do *ApplicationContext* todos os beans com escopo *singleton* são automaticamente carregados por default, ao contrário de *BeanFactory*, em que beans pertencentes a este escopo só são inicializados na primeira vez em que são requeridos. Esta é uma das razões pelas quais se costuma usar *BeanFactory*.

Oculte seu Container

Uma das principais vantagens do Spring é ser leve, sendo assim devemos proteger esta característica ao máximo possível. De preferência, apenas uma classe deverá

referenciar o container do Spring, ocultando-o do restante do sistema. Na **listagem 1** podemos ver como obter este resultado. É uma classe muito simples mas que nos diz muita coisa.

Listagem 1: Ocultando o Container de Injeção de Dependências

```
public class Container {

    private ApplicationContext contextoSpring;

    private ApplicationContext getContextoSpring() {
        if (contextoSpring == null) {
            contextoSpring = new ClassPathXmlApplicationContext("di/spring.xml");
        }
        return contextoSpring;
    }

    public Object getBean(String nome) {
        ApplicationContext contexto = getContextoSpring();
        if (contexto != null) {
            try {
                return contexto.getBean(nome);
            } catch (NoSuchBeanDefinitionException ex) {
                return null;
            }
        }
        return null;
    }

    public static synchronized Container getInstancia() {return _instancia;}

    private static Container _instancia = new Container();

    private Container() {}

}
```

O processo de instanciação do container é caro do ponto de vista computacional. Sendo assim, é interessante mantermos uma única instância de nosso *ApplicationContext*, o que conseguimos a partir da implementação do padrão *singleton* na classe *Container*. Assim garantimos que uma única instância do container será usada durante todo o processo de execução do nosso sistema.

O que é: Padrão de Projeto Singleton

O padrão de projeto Singleton tem como objetivo garantir que uma única instância de uma classe seja criada, além de também fornecer um único ponto de acesso a mesma. Seu uso é um tanto quanto controverso, sendo normalmente usado apenas em situações como a exposta na listagem 1 deste artigo.

Na **listagem 1** também vemos basicamente tudo o que você precisa saber a respeito de como obter um bean gerenciado pelo container. Basta executar a função *getBean*, que recebe como parâmetro o nome do bean presente no container. Seu valor de retorno é do tipo *Object*. Se no container não houver um bean com este nome, será disparada uma exceção do tipo *org.springframework.beans.factory.NoSuchBeanDefinitionException*, razão pela qual lidamos com esta exceção dentro da função *getBean*.

Observe que a classe *Container* possui apenas um único método público, no caso, *getBean*. Isto nos ajuda a garantir que nenhuma outra classe do projeto venha a ter uma dependência direta com o Spring, garantindo assim a leveza do framework.

Alimentando o Container com XML

Para que o container possa nos retornar um bean, antes ele precisa saber o que e como instanciar. E é neste momento que a configuração entra em cena. Tal como dito anteriormente, há basicamente três maneiras de se configurar o Spring: através de documentos no formato XML, usando anotações ou programaticamente.

Cada uma destas abordagens apresenta suas vantagens e desvantagens, que veremos no transcorrer deste artigo, porém dada sua natureza, é interessante começar pela configuração via XML, pois esta expõe todos os recursos do Spring que serão expostos no restante deste artigo. Na **listagem 2** podemos ver como nossa aplicação inicial pode ser configurada usando este formato.

Listagem 2: Configuração do Spring de nossa aplicação Inicial

```
   <?xml version="1.0" encoding="ISO-8859-1"?>
   <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">
       <bean id="cliente" class="di.Cliente">
          <property name="daoLivro" ref="daoLivro"/>
       </bean>
       <bean id="daoLivro" class="di.DAOLivroTexto" scope="prototype">
          <property name="arquivo" value="/livros.txt"/>
       </bean>
</beans>
```

O elemento raiz do documento é *beans*. Cada bean gerenciado pelo container é definido pela tag bean. Na listagem 2 podemos ver nossa aplicação configurada para trabalhar com um uma implementação de IDAOLivro que usa como fonte de dados um arquivo textual.

O atributo bean possui apenas um atributo obrigatório, que é o *class*, que especifica qual classe deverá ser instanciada. O atributo *id* é usado para nomear os beans gerenciados pelo container. Sendo assim, na **listagem 2** definimos dois beans: *cliente* e *daoLivro*.

No bean *cliente* podemos ver a aplicação da injeção de dependências. No interior da tag *bean* que o representa, há a tag *property*. Esta tag possui o atributo obrigatório *name*, que identifica qual o nome da propriedade que deverá ser atribuída pelo container. Spring utiliza o padrão JavaBeans. Sendo assim a classe *Cliente* deverá possuir um setter para a propriedade *daoLivro*, o que de fato possui, tal como pode ser conferido na **listagem 3**. Na tag *property* observa-se um outro atributo: *ref*. O valor deste atributo é o nome do bean que deverá ser injetado que, no caso, é o bean *daoLivro*.

É interessante observar que a tag *property* também é usada para definir o valor de propriedades do tipo string ou primitivas de um bean. Observe que na **listagem 2** definimos a propriedade *arquivo* do bean *daoLivro* com o valor "/livros.txt" usando o atributo *value*. Caso o atributo seja do tipo numérico o Spring se encarregará de fazer a conversão necessária para o usuário.

Listagem 3 – A classe Cliente

```
   public class Cliente {


       private IDAOLivro daoLivro;

       public IDAOLivro getDaoLivro() {
           return this.daoLivro;
       }

       public void setDaoLivro(IDAOLivro dao) {
           this.daoLivro = dao;
       }

       public void buscar(String valor) {
           List<String> titulos = getDaoLivro().getLivrosAutor(valor);
           for (String titulo : titulos) {
               System.out.println(titulo);
           }
       }

       public Cliente() {}

       public Cliente(IDAOLivro dao) {
           setDaoLivro(dao);
```

```
        }
    }
```

Os tipos de injeção de dependência

No Spring a injeção de dependências pode ser feita de duas formas: por setter ou construtor. Acima acabamos de ver a injeção por setter. Este tipo de injeção é executado em duas etapas. Na primeira o construtor default da classe definida par ao bean é executado, gerando uma nova instância para, em seguida, serem executados os setters definidos pelas tags *property* presentes no interior da tag *bean.* Sendo assim, quando for adotar a injeção de dependências por setter, **é obrigatória a presença do construtor default**.

Já a injeção de dependências por construtor é feita definindo-se qual construtor deverá ser executado ao criarmos uma nova instância do bean. A **listagem 4** expõe como o mesmo efeito que obtivemos usando a injeção por setter pode ser obtido ao adotarmos a injeção por construtor.

Listagem 4 – Injeção de dependências por construtor

```
   <!—Definindo o construtor por tipo →
<bean id="clientePorTipo" class="di.Cliente">
        <constructor-arg type="di.IDAOLivro" ref="daoLivro"/>
  </bean>
<!—Definindo pela ordem do atributo do construtor →
<bean id=”clientePorOrdem” class=”di.Cliente”>
      <constructor-arg index=”0” ref=”daoLivro”/>
</bean>
```

Para fazer a injeção por construtor usamos a tag *constructor-arg*. Há duas opções para se definir quais os valores de cada atributo do construtor. A mais recomendada é por tipo. Neste caso *constructor-arg* deverá receber um atributo chamado *type* (tal como na **listagem 4**) que identifique qual o tipo do argumento. O segundo atributo poderá ser tanto *ref*, caso estejamos referenciando outro bean quanto *value* se estivermos nos referindo a um atributo primitivo ou string.

A segunda opção é definir os valores dos atributos do construtor a partir de sua ordem. Na **listagem 4** podemos ver como isto é feito no bean *clientePorOrdem* . No caso, ao invés de usarmos o atributo *type*, usamos *index*, que define qual atributo a partir da sua ordem (sempre iniciando a partir do zero). Esta opção é útil para se evitar ambigüidade na escolha do construtor caso haja mais de um parâmetro pertencente a um mesmo tipo.

Injeção por setter ou construtor?

Como regra geral, opta-se pela injeção por construtores para as dependências obrigatórias e injeção por setter para as dependências opcionais. Os puristas preferem a injeção por construtores porque assim temos a garantia de que ao obtermos um bean, estamos lidando com uma instância completa, com todas as dependências obrigatórias preenchidas.

No entanto, a equipe de desenvolvimento do Spring incentiva a injeção por setter, pois assim evita-se a criação de construtores com vários atributos, alguns dos quais definindo inclusive dependências opcionais. É uma maneira de influenciar os desenvolvedores a evitarem a escrita destes construtores que acabam dificultando a manutenção.

Outra vantagem da injeção por setter é que ela nos permite reconfigurar a instância após obtida, visto que tudo o que o desenvolvedor precisa fazer é passar uma nova dependência ao setter em questão. Se a pergunta “vou precisar alterar alguma dependência de minha instância após obtida?” for positiva, a injeção por setter deverá ser adotada.

Dando nome aos beans

Não precisamos nomear um bean. Caso não o façamos, o próprio Spring irá criar um nome interno para o bean mapeado e, caso este bean "sem nome" seja o único de seu tipo, isto é, o único bean pertencente a determinada classe, podemos inclusive obtê-lo usando o método sobrescrito getBean(Class classe) do *ApplicationContext*.

Mas esta obviamente não é a melhor alternativa. Afinal de contas, nosso sistema pode crescer, e outros beans pertencentes à mesma classe podem acabar no nosso arquivo de configuração. E neste caso, ao tentar obter um bean pela sua classe ao invés de pelo seu nome, obteríamos uma exceção do tipo *org.springframework.beans.factory.NoSuchBeanDefinitionException* nos informando que há mais de um bean mapeado para aquele tipo específico.

Sendo assim é fundamental saber como dar nome aos nossos beans. A classe *bean* possui dois atributos para que possamos executar esta tarefa. O primeiro deles já vimos nas listagens 2 e 4. Trata-se do atributo *id*.

Usamos este atributo quando queremos definir um nome único para nossos beans. Ao definirmos este atributo para o bean, garantimos que em nosso container só poderá existir um bean como aquele nome. É uma boa prática usá-lo por outra razão: alguns parseadores de XML oferecem otimizações para este atributo, o que pode nos proporcionar ganhos em performance e validação destes documentos, visto que o atributo id é o mesmo definido na especificação do formato XML pelo W3C.

Nossa outra opção é o atributo *name*, que nos permite dar um ou mais nomes para um bean. É também usado quando é necessário nomear um bean usando caracteres especiais, o que não é permitido na especificação XML. No caso de um bean possuir mais de um nome, estes podem ser separados por espaço, ponto e vírgula (;) ou vírgulas (,). Na **listagem 5** podemos ver alguns exemplos.

Listagem 5 – Um bean com mais de um nome

```
<bean name="cliente;exemplo" class="di.Cliente"></bean>
<bean name="cliente exemplo" class="di.Cliente"></bean>
<bean name="cliente,exemplo" class="di.Cliente"></bean>
```

É comum oferecer mais de um nome a um determinado bean em projetos maiores, nos quais possam ocorrer situações nas quais times diferentes trabalhem em subsistemas distintos a serem integrados. Porém, na esmagadora maioria das vezes o ideal é que cada bean possua um único nome a fim de se evitar excessos de nomenclatura que acabem por gerar confusões dentro da equipe.

É uma boa prática adotar padrões de nomenclatura aos beans. Um padrão bastante adotado é o de se aplicar prefixos aos nomes dos beans que exponham sua finalidade, como por exemplo o prefixo *dao* a todos os beans que implementem o padrão de projeto DAO.

Modularizando a configuração

Uma das principais críticas à configuração no formato XML é a extensão que estes documentos possam vir a tomar. Realmente, é um tanto trabalhoso dar manutenção em arquivos gigantescos, porém é possível sanar este problema dividindo a configuração em arquivos distintos. Nestes casos cada arquivo de configuração pode representar uma camada lógica do sistema ou um módulo de sua arquitetura.

Há duas maneiras de se modularizar a configuração. A primeira delas é através da tag <*import*>, que pode ser inserida em seus documentos XML tal como exemplificado na **listagem 6**.

Listagem 6 – Usando a tag <import>

```
<beans>
        <import resource="servicos.xml"/>
        <import resource="dao/daos.xml"/>
      <bean id="cliente" class="di.Cliente"/>
</beans>
```

A tag *import* possui o atributo obrigatório *resource*, que define qual arquivo deverá ser importado para a configuração do Spring. Os caminhos dos arquivos a serem importados sempre são relativos ao do arquivo que está fazendo a importação. Sendo assim, se o arquivo da **listagem 6** estivesse, dentro do classpath do sistema no diretório *di/recursos* o arquivo importado *servicos.xml* também estaria neste mesmo diretório e *daos.xml* se encontraria no path *di/recursos/dao*.

O outro modo de obter o mesmo resultado é usando um construtor especial oferecido tanto por *ClassPathXmlApplicationContext* quanto *FileSystemApplicationContext* que recebe como parâmetros um array de strings fornecendo os caminhos para os arquivos de configuração a serem usados. A **listagem 7** exemplifica este processo usando a classe *ClassPathXmlApplicationContext*.

Listagem 7 – Modularizando a configuração com o construtor especial

```
String[] arquivos = {"di/recursos/spring.xml",
                     "di/recursos/servisos.xml",
                     "di/recursos/dao/daos.xml"};
  ApplicationContext contextoSpring = new ClassPathXmlApplicationContext(arquivos);
```

Dentre as duas opções a princípio é mais interessante a primeira, visto que não é necessário recompilar o código fonte caso seja necessário adicionar algum novo arquivo de configuração ao container. No entanto, a segunda opção se mostra interessante em situações nas quais a escolha dos arquivos de configuração envolvam alguma lógica interna do seu sistema.

Escopo Singleton vs Prototype

Por padrão todo bean gerenciado pelo Spring possui escopo *singleton*. É fundamental compreender a diferença entre os dois escopos básicos oferecidos pelo Spring para evitarmos problemas no futuro.

Tal como exposto na **tabela 1**, quando um bean se encontra no escopo *singleton* o Spring se encarregará de manter uma única instância deste bean sobre seu controle. Sendo assim, toda vez que estivermos chamando o bean *cliente* definido na **listagem 2** estaremos obtendo a mesma instância da classe *di.Cliente*. No entanto, é importante observar que estamos lidando aqui com um singleton *relativo*, visto que nada impede que criemos fora do container uma instância diferente desta mesma classe chamando seu construtor padrão, por exemplo.

Um fator que precisa ser levado em conta é que o *BeanFactory* e o *ApplicationContext* adotam políticas diferentes com relação a beans com escopo *singleton*. Quando o *ApplicationContext* é inicializado, por padrão todos os beans com escopo singleton são instanciados e possuem suas dependências injetadas, ao passo que no *BeanFactory* estes só serão preparados no momento em que forem chamados pela primeira vez.

Já no caso do prototype, sempre que pedirmos ao container um bean que pertença a este escopo obteremos uma nova instância. Isto nos trás um certo impacto na performance, visto que sempre ocorrerá o processo de preparação do bean, que é composto por duas etapas: instanciação do mesmo e injeção das dependências.

Na prática utiliza-se o padrão singleton para beans mais complexos e que não armazenem estado, como por exemplo a classe SessionFactory do Hibernate.

Uma situação para a qual é necessário ficar atento é quando um bean com escopo singleton possui uma dependência com escopo prototype. É preciso ter consciência de

que a injeção de dependências em um bean só é feita no momento em que o bean é instanciado. Sendo assim, a dependência com escopo prototype injetada no bean singleton apenas uma vez.

Se você quiser no entanto que o seu bean com escopo singleton sempre obtenha uma nova instância de sua dependência com escopo prototype, uma solução é tornar este bean consciente da existência do container de injeção de dependências. **Esta não é uma boa prática** pois estamos "adicionando peso" ao Spring, visto que agora uma classe de negócios da nossa aplicação passa a possuir uma dependência direta com o container de injeção de dependências.

Mas caso seja de fato necessário, basta que sua classe implemente a interface *org.springframework.context.ApplicationContextAware* tal como a versão alternativa da classe *Cliente* que encontra-se exposta na **listagem 8**.

Listagem 8: um bean com consciência do container de injeção de dependências

```
public class Cliente implements ApplicationContextAware {
   // o restante da classe é igual
    private ApplicationContext applicationContext;

    public ApplicationContext getContextoSpring() {
        return applicationContext;
    }

    public void setApplicationContext(ApplicationContext ac) throws BeansException      {
        this.applicationContext = ac;
    }

}
```

O container do Spring irá injetar nas classes que implementem a interface *ApplicationContextAware* uma instância de si mesmo chamando o método *setApplicationContext*, que recebe como parâmetro uma instância e *ApplicationContext*. Sendo assim, internamente a classe pode pedir ao container que sempre lhe retorne uma nova instância de uma de suas dependências cujo escopo seja singleton.

Inicialização tardia de beans singleton

Tal como falamos na seção anterior, o *ApplicationContext* por default prepara todos os beans com escopo singleton no momento em que é carregado. Tal comportamento nem sempre é adequado, visto que há situações nas quais alguns beans são usados somente em circunstâncias muito específicas. Sendo assim, porque carregá-los se talvez nem sejam necessários durante a execução do sistema?

A solução para o problema é simples. Basta adicionarmos o atributo *lazy-init* à tag bean com o valor "*true"*. Agora o bean só será processado no momento em que for requerido pela primeira vez. É possível incluir o atributo *lazy-init* na tag *bean* também: assim altera-se o comportamento default de carregamento para todos os beans presentes naquele arquivo de configuração.

No entanto, nem sempre esta é uma boa prática. Enquanto o sistema estiver em desenvolvimento, é interessante que todos os beans sejam carregados, visto que assim será possível encontrar erros de configuração nesta etapa inicial do processo, facilitando assim a correção de erros.

Autowiring

O container de injeção de dependências do Spring pode automaticamente descobrir quais as dependências de um bean a partir de um recurso bastante interessante chamado *autowiring*. Como vantagem, o desenvolvedor pode reduzir significativamente o número de propriedades e construtores que precisa especificar em seus arquivos de configuração.

O funcionamento do autowiring inicia-se quando marcamos um bean para que seja auto injetado[8] pelo container. Quando este bean for ser iniciado, o container irá varrer todos os atributos presentes em sua classe e, para cada um, verificará entre as definições de beans armazenada em seu interior a existência de um bean que possa ser injetado naquele ponto. Encontrados todos os candidatos apropriados, estes são injetados no bean. Caso haja alguma ambigüidade, o container não saberá qual bean deve ser injetado e, neste caso, uma exceção do tipo *org.springframework.beans.factory.NoSuchBeanDefinitionException* informando qual atributo não pode ser injetado.

A melhor maneira de entender este conceito é vê-lo em prática. Na listagem 9 podemos ver como configurar o Spring para tirar proveito deste recurso. Podemos ver três beans definidos: *daoLivro*, *cliente* e *autowired*. O primeiro será injetado nos demais. Enquanto na definição do bean *cliente* precisamos definir todas as dependências manualmente, o mesmo não é verdade com o bean *autowired*. Repare que não foi necessário incluir em sua definição nenhum construtor ou propriedade, apenas o atributo *autowire* com o valor *byType*.

Listagem 9 – Usando autowiring

```
<beans>
   <bean id="daoLivro" class="di.DAOLivroTexto">
      <property name="arquivo" value="/livros.txt"/>
    </bean>
    <bean id="cliente" class="di.Cliente">
      <property name="daoLivro" ref="daoLivro"/>
    </bean>
    <bean name="autowired" class="di.Cliente" autowire="byType">

    </bean>
</beans>
```

Quando lidamos com este recurso, nosso principal oponente é a ambigüidade. Se na **listagem 9** houvesse mais um bean do tipo *di.DAOLivroTexto* nos depararíamos com um erro no momento em que o bean *autowired* fosse iniciado pelo container, pois este não saberia qual bean injetar na propriedade *daoLivro* do bean *autowired*.

Uma das ferramentas que possuímos para evitar estes problemas são os modos fornecidos pelo Spring para encontrar automaticamente as dependências a serem injetadas. Na **tabela 2** podemos ver todos estes modos, cujos nomes correspondem ao valor a ser digitado no atributo *autowire* de um bean automaticamente injetado.

Tabela 2 - Estratégias de auto injeção de dependências

Modo	Descrição
no	Comportamento default. Desabilita a auto injeção de dependências para o bean.
byName	O container irá buscar por dependências que possuam o mesmo nome que a propriedade a ser injetada. Sendo assim, na **listagem 9** poderíamos ter definido a auto injeção usando esta estratégia que funcionaria sem problemas. Caso não hajam beans com o mesmo nome presentes entre as definições do container, a exceção *NoSuchBeanDefinitionException.*
byType	A busca por dependências será por tipo. O container irá buscar por beans que sejam do mesmo tipo que a dependência a ser injetada. Havendo mais de uma possibilidade será disparada uma exceção do tipo *NoSuchBeanDefinitionException*.
constructor	É análoga à estratégia byType. A Diferença é que será feita a injeção de dependências a partir dos construtores do bean.

[8] O termo em inglês que poderia ser adotado ao invés de "auto injetado" seria "autowired"

autodetect	Seguindo esta estratégia o container irá primeiro buscar executar a injeção de dependências por construtor. Se for encontrado um construtor default no bean, o container passará a adotar a estratégia *byType*.

Deve-se usar a auto injeção de dependências apenas em projetos de pequeno e médio porte, visto que trata-se de um processo bem menos preciso que a injeção explícita através da definição de propriedades e construtores. Apesar do Spring se esforçar ao máximo para fazer a escolha correta, podem haver situações nas quais uma injeção incorreta seja feita. Sendo assim seu uso normalmente é desencorajado em projetos maiores, a não ser que sejam adotadas regras rígidas na nomenclatura dos beans e sua tipagem.

Caso queira diminuir ainda mais a quantidade de configuração a ser digitada, é possível definir o comportamento de auto injeção default do container adicionando o atributo *default-autowire* à tag *beans* com um dos valores presentes na **tabela 2**. Assim todos os beans nela contidos seguirão a mesma política.

Para aumentar a precisão da injeção automática, podemos também excluir candidatos à injeção incluindo o atributo *autowire-candidate* com o valor *"false"* na tag *bean* do componente que queremos remover do processo de busca.

Outra possibilidade interessante é dar preferência a uma dependência específica adicionando o atributo *primary* com valo *"true"* à tag *bean* correspondente. Em situações nas quais ocorram mais de uma possibilidade de escolha, este será privilegiado no processo, ajudando assim a diminuir a possibilidade de ambigüidade no processo de auto injeção de dependências.

Porém, estas duas alternativas acabam por tornar mais complexo o processo de criação dos arquivos de configuração. Por esta razão é uma boa prática simplesmente não usar o autowiring em projetos de grande porte.

Callbacks de Ciclo de Vida

Há situações nas quais não basta apenas injetar as dependências de um bean e definir seu escopo. São casos nos quais para que um bean esteja pronto para uso, seja necessário algum pré-processamento interno antes que o mesmo seja fornecido ao objeto que o solicitou ao container. Além disto, há momentos nos quais é necessário que o próprio bean libere os recursos que usou durante sua existência.

Spring nos oferece dois eventos que podemos customizar no ciclo de vida de nossos beans: *inicialização*, que é executado logo após todas as dependências de um bean terem sido injetadas e *destruição*, executado apenas após o container que gerencia o bean ter sido destruído.

O callback de inicialização pode ser adicionado ao bean de duas maneiras. A primeira delas é a implementação da interface *org.springframework.beans.factory.InitializingBean*. Esta contém um único método do tipo chamado *afterPropertiesSet()* que dispara uma exceção genérica. Na **listagem 10** há um exemplo de sua implementação. Este procedimento não é recomendado pois acopla o *bean* ao Spring Framework, violando assim um dos principais objetivos do framework que é justamente sua não intrusão nas classes do sistema.

Listagem 10 – Implementando callbacks de inicialização e destruição no código fonte

```
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class ClienteCallbacks implements InitializingBean, DisposableBean {

    public void afterPropertiesSet() throws Exception {
        System.out.println("Todas as dependências injetadas");
    }
```

```
    public void destroy() throws Exception {
        System.out.println("Container destruido");
    }

}
```

A outra opção é incluir na tag *bean* o atributo *init-method*. O valor do atributo deve ser o nome de um método definido no bean que não possua atributos e que pode ou não disparar alguma exceção. Esta é a abordagem adequada, visto que assim garantimos que nosso bean não esteja diretamente acoplado ao Spring Framework.

Já o callback de destruição também pode ser adicionado usando basicamente os mesmos procedimentos que adotamos par ao callback de inicialização. Podemos implementar a interface *DisposableBean*, que nos obrigará a implementar o método *destroy()*, tal como fizemos na **listagem 10** ou, se preferirmos, podemos também adicionar o atributo *destroy-method* à tag *bean* correspondente. O valor do deste atributo deverá ser um método que não possua parâmetros. Como no caso do callback de inicialização, esta é a abordagem favorita por evitar o acoplamento de nossas classes ao código do Spring Framework.

Há um detalhe que precisa ser mencionado: o callback de destruição não é executado jamais em beans cujo escopo seja prototype, pois uma vez criado, este bean não está mais sobre o poder do container de injeção de dependências, sendo assim é impossível que o container venha a executar os métodos de destruição nestes beans.

Alimentando o Container com Anotações

Uma alternativa ao XML que apareceu pela primeira vez no Spring 2.0 é o uso das anotações. Trata-se de um caminho bastante interessante para os que gostam de ver suas configurações próximas ao código fonte ou sentem-se incomodados com a necessidade de se escrever documentos XML.

O problema com as anotações é que elas aumentam o peso do Spring ao acoplar nossas classes ao Spring Framework. Como veremos, é possível diminuir este acoplamento ao adotarmos anotações baseadas em JSRs, porém mesmo assim estaremos aumentando o acoplamento de nossas classes, o que não ocorreria se estivéssemos adotando configurações puramente baseadas em XML. Outro problema é que a necessidade de se recompilar código ao adicionarmos novos componentes ao nosso sistema aumenta significativamente.

Porém as anotações não excluem completamente o XML. Na realidade, XML e anotações convivem perfeitamente bem em um mesmo sistema. Voltando ao nosso sistema inicial, uma versão alternativa baseada em anotações precisaria de um arquivo de configuração tal como o exposto na **listagem 11**.

Listagem 11 – Configurando o container para trabalhar com anotações

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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"
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="di.anotados"/>

    <bean id="daoLivro" class="di.DAOLivroTexto" scope="prototype" primary="true">
        <property name="arquivo" value="/livros.txt"/>
    </bean>

</beans>
```

Como pode ser observado na **listagem 11**, ainda estou mapeando um bean usando XML. Foi feito propositalmente para expor como as duas abordagens, anotações e XML convencional convivem perfeitamente lado a lado. Comparando a **listagem 11** com a **listagem 2** observa-se a inclusão de um novo namespace: *context*. É fundamental que este seja declarado para que o Spring possa trabalhar com anotações.

Há também duas novas tags: <*context:annotation-config/*> informa o container que serão usadas configurações baseadas em anotações além do XML convencional. Outra tag importante é <*context:component-scan/*>. Esta instrui o container a, no momento em que for inicializado, varrer todas as classes contidas no pacote *di.anotados* em busca dos beans a serem gerenciados pelo container.

O leitor atento também observará a ausência do bean *cliente*. Nesta versão alternativa do nosso sistema este bean será configurado usando apenas anotações. Sendo assim, implementamos uma classe chamada *ClienteComponente* exposta na **listagem 12**. Esta classe expõe praticamente todas as configurações que expusemos na seção dedicada à configuração baseada em XML.

Dado que os conceitos básicos da injeção de dependências foi dado na seção em que tratamos da configuração proveniente de documentos em formato XML, nosso foco agora será em expor os equivalentes daquela configuração ao adotarmos o modelo de configurações baseadas em anotações.

Listagem 12 – A classe ClienteComponente

```
import di.IDAOLivro;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Scope("prototype")
@Component("clienteAnotado")
public class ClienteComponente {


    private IDAOLivro daoLivro;

    public IDAOLivro getDaoLivro() {
        return this.daoLivro;
    }

    @Resource(name="daoLivro")
    public void setDaoLivro(IDAOLivro dao) {
        this.daoLivro = dao;
    }

    @PostConstruct
    public void init() {
        System.out.println("Fui construido. E meu dao é " + getDaoLivro().getClass());
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Meu container foi destruido");
    }

}
```

Identificando beans com @Component

Ao varrer o pacote *di.anotados* o container irá buscar todas as classes que possuam a anotação *@Component*. Esta equivale à tag *bean* que vimos anteriormente. Como boa prática, é interessante passar como parâmetro para esta classe um valor do tipo string que nomeie nosso bean. Caso não seja feito, o container irá criar um nome interno para o bean, porém este seria de pouca valia para nós, pois um *bean* realmente útil é aquele que possamos usar, e para usá-lo, este precisa ser nomeado.

Pense nesta anotação como a "dica" que enviamos ao container para que identifique uma de nossas classes como um bean a ser gerenciado.

Definindo o escopo com @Scope

A anotação *@Scope* é o equivalente ao atributo *scope* da tag *bean*. Se omitido, será adotado o escopo padrão do Spring que, tal como já mencionado, é o Singleton. Esta anotação recebe como parâmetro o nome do escopo que iremos adotar. Na **listagem 12** só para variar adotamos o escopo *prototype*.

Definindo dependências com @Resource

Anotações também possuem um relativo para a tag *property*. No caso, estamos falando da anotação @Resource, que deve ser aplicada sobre atributos ou setters de nossas classes. Este atributo recebe como parâmetro o nome do bean que desejamos seja injetado em nosso bean.

É interessante observar que esta anotação não é nativa do Spring, mas sim da JSR-250, cujo objetivo é definir anotações para conceitos semânticos em comum presentes nas plataformas Java SE e EE. Sendo assim, ao usarmos esta anotação, estamos na realidade evitando mais uma ligação direta com classes do Spring, o que é uma prática interessante, visto que no futuro, caso seja necessário trocar o container de injeção de dependências esta tarefa se tornará menos trabalhosa.

Na **listagem 12** estamos instruindo o container a injetar o **bean** *daoLivro* no bean *clienteAnotado*.

Autowiring com @Autowired ou @Inject

Equivalente ao atributo *autowired* que vimos na configuração baseada em XML possuímos as anotações @Autowired e @Inject. @Autowired é a anotação provida pelo Spring. Sendo assim, ao adotá-la devemos ter em mente que estamos incluindo uma dependência direta ao Spring Framework em nosso código fonte. Já a anotação @Inject faz parte da JSR-330, que especifica o mecanismo de injeção de dependência padrão para a plataforma Java. Sendo assim, o uso de @Inject é preferível ao de @Autowired, pois assim estamos criando uma dependência com um padrão Java, e não algo tão específico quanto o Spring.

Na **listagem 13** podemos ver um exemplo da aplicação de @Autowired em mais uma versão de nosso bean *cliente*, assim é possível expor mais uma diferença entre @AutoWired e @Inject. No caso, @Autowired possui um atributo adicional chamado required, que aceita um valor booleano. Quando definido como true, caso não seja encontrado o bean a ser injetado, uma exceção será disparada pelo container de injeção de dependências.

Esta opção é útil quando a aplicação está em desenvolvimento, pois permite aos desenvolvedores encontrar problemas antes que o sistema entre em produção.

Listagem 13 – Usando @Autowired

```
import di.IDAOLivro;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("clienteAnotadoAutowired")
public class ClienteComponenteAutoWired {

    @Autowired(required=true)
    private IDAOLivro dao;

    public IDAOLivro getDaoLivro() {return dao;}
    public void setDaoLivro(IDAOLivro dao) {this.dao = dao;}

}
```

Callbacks de inicialização e destruição com @PostConstruct e @PreDestroy

Na **listagem 12** podemos ver a aplicação das anotações @PostConstruct e @PreDestroy, que equivalem aos atributos init-method e destroy-method que vimos na configuração baseada em XML.

Seu uso não poderia ser mais simples: inclua @PostConstruct no método sem parâmetros que deverá ser executado após ter sido feita a injeção de dependências em seu bean para identificar o callback de inicialização e @PreDestroy no método sem parâmetros que deverá ser executado no momento em que o container de injeção de dependências for ser destruído.

Quando o XML anula as anotações

Ao ser inicializado, o container do Spring primeiro carrega as configurações provenientes de anotações para em seguida processar as configurações em formato XML. Sendo assim, é importante que o leitor saiba que, caso um bean seja configurado inicialmente usando anotações, e este mesmo bean esteja especificado no formato XML, as configurações no segundo formato anularão a primeira.

Este é um erro muito comum em times que tenham optado por trabalhar com os dois formatos, porém pode ser visto também como uma alternativa viável quando se deseja anular configurações em cima das quais não possuímos controle, como por exemplo as que estão presentes em classes já compiladas presentes no classpath de nosso sistema.

Limitações das anotações

Atualmente configurações baseadas em anotações não permitem que seja mapeado mais de um bean para uma mesma classe tal como fizemos na **listagem 4**. Caso seja necessário que isto seja feito, o desenvolvedor possui apenas duas alternativas: ou mapeia apenas um bean usando anotações para uma classe e o restante usando XML ou faz todo o trabalho usando apenas XML.

Dentre as duas alternativas, a segunda é uma melhor prática, visto que assim expõe para o time todas as variantes do bean para todos os membros da equipe de uma forma mais explícita, evitando assim problemas de comunicação dentro do time.

Outra limitação das anotações é que elas não nos permitem definir o valor de atributos primitivos ou listas, tal como podemos fazer facilmente no caso do XML.

Configuração baseada em Java (100% livre de XML!)

No Spring 3.0 foi introduzido um novo tipo de contexto de aplicação chamado *AnnotationConfigApplicationContext* que, pela primeira vez, possibilita ao desenvolvedor configurar um container de injeção de dependências do Spring sem a presença de qualquer arquivo XML, pois toda configuração é feita via código.

Trata-se de uma alternativa bastante poderosa que pode ser usada em situações nas quais os beans gerenciados pelo container não possam ser definidos apenas declarativamente como fizemos usando apenas anotações ou XML. Agora podemos incluir lógica neste processo de uma maneira bastante simples.

Visto que todos os conceitos referentes ao processo de gerenciamento de beans já foi tratado nas seções anteriores deste artigo, iremos ver aqui apenas como mapear os beans usando este novo recurso de uma maneira rápida comparando-o com as maneiras anteriores.

Apresentando @Configuration e @Bean

Antes de tratarmos da classe *AnnotationConfigApplicationContext* iremos ver o que o alimenta, no caso, as anotações @Configuration e @Bean aplicadas a POJO's. Na **listagem 14** podemos ver a aplicação destas anotações.

Listagem 14 – Aplicando as anotações @Configuration e @Bean

```
import di.Cliente;
import di.DAOLivroSGBD;
import di.DAOLivroTexto;
import di.IDAOLivro;
import org.springframework.beans.factory.annotation.Autowire;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class BeanSource {

    @Bean(name="daoLivro")
    public IDAOLivro getDAOLivro() {
        return new DAOLivroTexto();
    }

    @Bean(name="cliente")
    public Cliente getCliente() {
        Cliente saida = new Cliente();
        saida.setDaoLivro(getDAOLivro());
        return saida;
    }

    @Bean(name="clienteSGBD")
    public Cliente getClienteSGBD() {
        Cliente saida = new Cliente();
        saida.setDaoLivro(new DAOLivroSGBD());
        return saida;
    }

    @Bean(name="clienteAutowired", autowire=Autowire.BY_TYPE)
    public Cliente getClienteAutowired() {
        return new Cliente();
    }

}
```

Tabela 3 - Atributos de @Bean

Atributo	Descrição
name	O nome dado ao bean que será retornado
autowire	Qual política de autowiring que será adotada pelo container. Valores possíveis: Autowire.BY_TYPE Autowire.BY_NAME Autowire.NO > Desabilita o autowiring, e é o valor default
initMethod	Nome do método que deverá ser invocado no bean após este ter suas dependências injetadas
destroyMethod	Nome do método que deverá ser invocado quando o container que gerencia o bean for destruído

A classe presente na **listagem 14** é o que chamamos de *Configuration*, ou seja, uma classe anotada com @Configuration e que possui funções anotadas com a anotação @Bean que produzem beans que serão gerenciados pelo container de injeção de dependências do Spring.

Sendo assim, para criar uma classe de configuração, tudo o que o desenvolvedor precisa fazer é anotá-la com @Configuration e em seguida implementar uma ou mais funções anotadas com @Bean, cujos atributos encontram-se listados na **tabela 3**.

Autowiring

A primeira impressão que temos quando nos deparamos com este novo recurso é que não precisaremos mais da injeção automática de dependências, visto que estamos manualmente criando nossos beans. Felizmente isto não é verdade. Para ativar o *autowiring*, basta definir o valor do parâmetro *autowire* com o valor *Autowire.BY_NAME* ou *Autowire.BY_TYPE* tal como exposto na **tabela 3**.

Observando a **listagem 14** vemos que o autowiring é aplicado no método getClienteAutowired() usando a estratégia por tipo.

Definindo o escopo

Novamente aqui o padrão do Spring se aplica. Se não for definido explicitamente qual o escopo do bean, este será interpretado como pertencente ao escopo *singleton*. Para que o bean possua um escopo

Callbacks de Inicialização e Destruição

Tal como na configuração baseada em XML como na configuração baseada em anotações, também podemos definir os callbacks de inicialização e destruição. A diferença é que os definimos com os atributos *initMethod* e *destroyMethod* presentes na anotação @Bean. Exatamente como nos casos anteriores, eles devem conter o nome de um método que não possua atributos presentes na classe do bean a ser retornado.

Apresentando o container AnnotationConfigApplicationContext

A classe que possibilita a configuração baseada em Java é AnnotationConfigApplicationContext, que se encontra no pacote *org.springframework.context.annotation*. Para usá-la, no entanto, é necessário adicionar duas dependências externas ao seu projeto. No caso as bibliotecas CGLIB e ASM[9], que são usadas para manipular bytecode em tempo de execução.

Há dois modos de se alimentar as configurações de *AnnotationConfigApplicationContext*. O desenvolvedor pode criar uma nova instância desta classe passando uma única classe de configuração ou um array de classes de configuração ou programaticamente. As duas maneiras podem ser vistas na **listagem 15**.

Listagem 15 – Alimentando AnnotationConfigApplicationContext

```
AnnotationConfigApplicationContext contexto = null;
// Alimentando com uma única classe de configuração
contexto = new AnnotationConfigApplicationContext(BeanSource.class);
//Alimentando com um array de classes de configuração
contexto =  new AnnotationConfigApplicationContext({BeanSource.class,
DataSources.class});
// Alimentando o context programáticamente
contexto = new AnnotationConfigApplicationContext();
contexto.register(BeanSource.class);
//após adicionar todas as classes ao contexto, execute obrigatóriamente o método
refresh()
contexto.refresh();
```

Conclusões

Como pudemos ver, o container de injeção do Spring é uma ferramenta bastante rica, que nos permite resolver de maneira elegante o problema do alto acoplamento em nossos sistemas. Tal como dito na introdução, é importante conhecer o conceito de

[9] A biblioteca CGLIB (Code Generation Library) pode ser obtida em seu site oficial: http://cglib.sourceforge.net/ Para este artigo foi usada a versão 2.2. Já a biblioteca ASM pode ser obtida em http://asm.ow2.org/. Usamos neste artigo a versão 3.2.

injeção de dependências mesmo que o Spring jamais seja usado, porque seu conhecimento torna explicita a principal causa de designs ruins de aplicação.

Porém, caso o Spring venha a ser adotado, é importante que a sua não intrusão seja mantida o máximo possível, pois caso contrário podemos acabar voltando ao problema do alto acoplamento. Tal como exposto no artigo, conseguimos manter esta leveza a partir da adoção de configuração no formato XML ou Java e, no caso das anotações, adotando ao máximo possível apenas anotações que sejam parte de alguma JSR, ao contrário das que fazem parte do core do Spring.

O mais interessante, no entanto, é a constatação do poder que o conceito de injeção de dependências possui. Percebemos este poder ao constatar que "apenas" o que foi exposto neste artigo fornece a base para **todos** os demais componentes do Spring Framework e seus derivados, como Grails por exemplo.

Referências

http://www.springframework.org – Site oficial do Spring Framework

http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/ - Documentação de referência do Spring Framework 3.0

http://www.objectmentor.com/publications/dip.pdf - Artigo "The Dependency Injection Principle" de Robert C. Martin

http://www.martinfowler.com/articles/injection.html - Artigo "Inversion of Control Containers and the Dependency Injection pattern" de Martin Fowler que cunhou o termo Injeção de Dependências

Henrique Lobo Weissmann (o Kico) (*kicolobo@itexto.net*) é consultor Groovy/Grails, fundador do Grails Brasil e sócio da itexto Desenvolvimento de Projetos (http://www.itexto.com.br), que atua na criação de projetos adotando software livre e muito Grails. Além disto, em seu blog (http://devkico.itexto.com.br) expõe sua experiência no desenvolvimento de software.