



# lindbergframework

Plug and Use

Versão 1.0

Autor: Victor Lindberg  
Brasil, Distrito Federal, Brasília,  
2011

## Sobre o Autor.

Victor Lindberg, autor do `lindbergframework`, é natural de Maceió – Alagoas e mora atualmente em Brasília – Distrito Federal. Atualmente no Serviço Federal de Processamento de Dados (SERPRO) em Brasília, é Analista Projetista com foco de atuação em Arquitetura JAVA na plataforma Enterprise atuando em projetos como o Novo Sistema Integrado de Administração Financeira (Novo SIAFI), da Secretaria do Tesouro Nacional (STN), e o Novo Sistema de Certificação Digital do SERPRO (Novo SCDS).

E-mail pessoal: [victorlindberg713@gmail.com](mailto:victorlindberg713@gmail.com)

# Sumário

1 - Introdução/Motivação.....	5
2 – Visão Geral.....	6
2.1 – Arquitetura.....	6
2.1.1 – Módulos e responsabilidades.....	7
3 – O Framework.....	8
3.1 – Processo de Configuração.....	8
3.2 - Contextos.....	9
3.3 – Configuração do CORE.....	9
3.3.1 – Configurando o CORE na Prática - Programaticamente.....	10
3.3.2 – Configurando o CORE na prática – Usando XML.....	10
3.3.2.1 – A tag <config-property>.....	12
3.3.2.2 – Inicializando CoreContext usando configuração XML.....	14
3.3.2.3 – Uso do carácter '#' no XML para definir propriedades e métodos getters estáticos.....	15
3.3.3 – A Classe CoreContext.....	16
3.3.4 – Configurando em aplicações WEB.....	16
4 – Injeção de Dependência e Inversão de Controle com o lindbergframework.....	19
4.1 – Inversão de Controle na Prática.....	19
4.2 – Fábrica de Beans (BeanFactory).....	20
4.2.1 – Injeção de Dependências.....	21
4.2.2 – Obtendo beans a partir do Contexto.....	24
4.2.3 – Beans com escopo Singleton.....	25
4.3 – Integrando o Contexto de Beans do lindbergframework com outros frameworks.....	25
4.3.1 – Integrando com JSF (Java Server Faces).....	26
4.3.2 – Integrando com Adobe Flex via Blazeds.....	27
5 – Lindberg Persistence – LINP.....	28
5.1 – Conceitos Básicos.....	29
5.2 – Repositório de Comandos SQL - Introdução.....	30
5.3 – Configuração do LINP.....	30
5.3.1 – Configurando o LINP na prática - Programaticamente.....	32
5.3.2 – Configurando o LINP na prática – Usando XML.....	34
5.3.2.1 – A tag <dataSource>.....	38
5.3.3 – A classe LinpContext.....	40
5.4 – Repositório de comandos SQL – Mais Detalhes.....	40
5.5 – Usando o LINP na prática nos DAO's.....	41
5.6 – População de Beans Multi-Nível.....	42
5.7 – Usando o padrão de nomenclatura do JAVA para definir comandos SQL's.....	44
5.7 – Template de Persistência (PersistenceTemplate).....	44
5.8 – Definindo e trabalhando com repositório de Comandos SQL usando XML.....	51
5.9 – Adicionando os recursos de persistência do LINP aos DAO's.....	55
5.10 – Exemplo de uso do LINP na prática.....	57
5.10.1 - Exemplo DAO acessando repositório de comandos SQL definindo no exemplo:.....	64
5.10.2 - Exemplo DAO sem uso de repositório de comandos SQL:.....	72
5.11 – Gerenciamento de Transações.....	74
5.11.1 – Como definir seu próprio Gerenciador de Transações TransactionManager.....	75
5.11.2 – A annotation @Transax. Como definir um Contexto Transacional.....	76
5.12 – Integrando o LINP com o Springframework.....	80
6 – Importando o lindbergframework a partir do Maven.....	82
6.1 – Importando apenas o módulo padrão do lindbergframework.....	82

6.2 – Importando o módulo padrão do lindbergframework para aplicações WEB.....	82
6.3 – Importando lindbergframework para aplicações WEB integrando com JSF (Java Server Faces).....	83
6.4 – Importando lindbergframework para aplicações WEB integrando com Adobe Flex.....	83
7 – lindbergframework - Links.....	84

# 1 - Introdução/Motivação.

O lindbergframework foi desenvolvido inicialmente para prover soluções simples para problemas que milhares de desenvolvedores se deparam todos os dias de modo a funcionar como algo do tipo “*plug and play*”, onde sem complexidades desnecessárias, sem uma alta curva de aprendizado, se adiciona ao projeto algo que vai prover recursos como várias caixas de ferramentas customizáveis sendo cada uma para determinado estereótipo de problema.

Soluções como implementação de conceito de repositório de comandos SQL de modo a extrair os SQL's dos DAO's provendo um repositório para os mesmos podendo estar esse repositório em qualquer lugar seja um XML, um Arquivo texto, um XML de um Schema específico que o desenvolvedor quiser, seja em um Webservice ou qualquer outro serviço como Rest, ou até mesmo mesclar diversos repositórios, abstração e automatização de população de beans multi-nível, gerenciamento de transações sem complicações e transparente, toda uma solução de persistência, *container* de injeção de dependências para implementação de IOC (Inversion of Control ou Inversão de Controle), integração com JSF (Java Server Faces), Springframework, Adobe Flex, entre outros recursos, compõem essas caixas de ferramentas customizáveis.

Uma das caixas de ferramentas mais importantes providas pelo framework é o LINP (Lindberg Persistence) que fornece diversos recursos de persistência ao projeto de modo eficiente trabalhando diretamente com a API do JDBC mas abstraindo de toda complexidade e trabalho tedioso.

O importante é entender a real motivação de se criar uma solução de persistência visto que hoje já existem framework's ORM (Object Relational Mapping ou Mapeamento Objeto Relacional) consolidados na comunidade como Hibernate e JPA. A questão não é deixar de usar qualquer um destes e usar os recursos de persistência providas pelo lindbergframework. Hoje com todos os padrões e especificações já existentes para persistência em JAVA é cada vez mais comum se adotar um framework ORM, como os citados anteriormente. Mas a questão é que nem todas as aplicações usam esse tipo de framework por diversos motivos.

Imagine uma aplicação que hoje já usa JDBC e tem um conjunto de stored procedures e functions no banco e que não tem um modelo de dados, digamos, muito bem modelado. Mudar tudo para JPA ou Hibernate, uma vez que você já tem métodos prontos com sql direto, fazendo diversos *joins*, chamando essas procedures e functions para efetuar alguma coisa, obter algum resultado, alguma listagem via cursors de banco, enfim, o custo e trabalho para se efetuar uma migração como esta é muito alto. Mesmo um projeto novo pode por algum motivo não usar um framework ORM, seja por decisão arquitetural, seja por algum requisito não funcional que impacte em algum ponto referente a performance que mesmo com os mecanismos de otimização dos mesmos não atenda, pode não usar nenhuma destas soluções ORM e adotar algum componente ou framework mais simples, mas não menos eficiente, que trabalhe diretamente com JDBC ou abstraia o mesmo e todo ou boa parte do trabalho que este requer para operações de persistência. Em alguns casos pode ser uma boa até mesclar as duas soluções, tanto usar um framework ORM como Hibernate e nos casos onde se aplica usar o lindbergframework.

Leia a documentação abaixo, *plug* o lindbergframework e “mãos na massa”.

## 2 – Visão Geral.

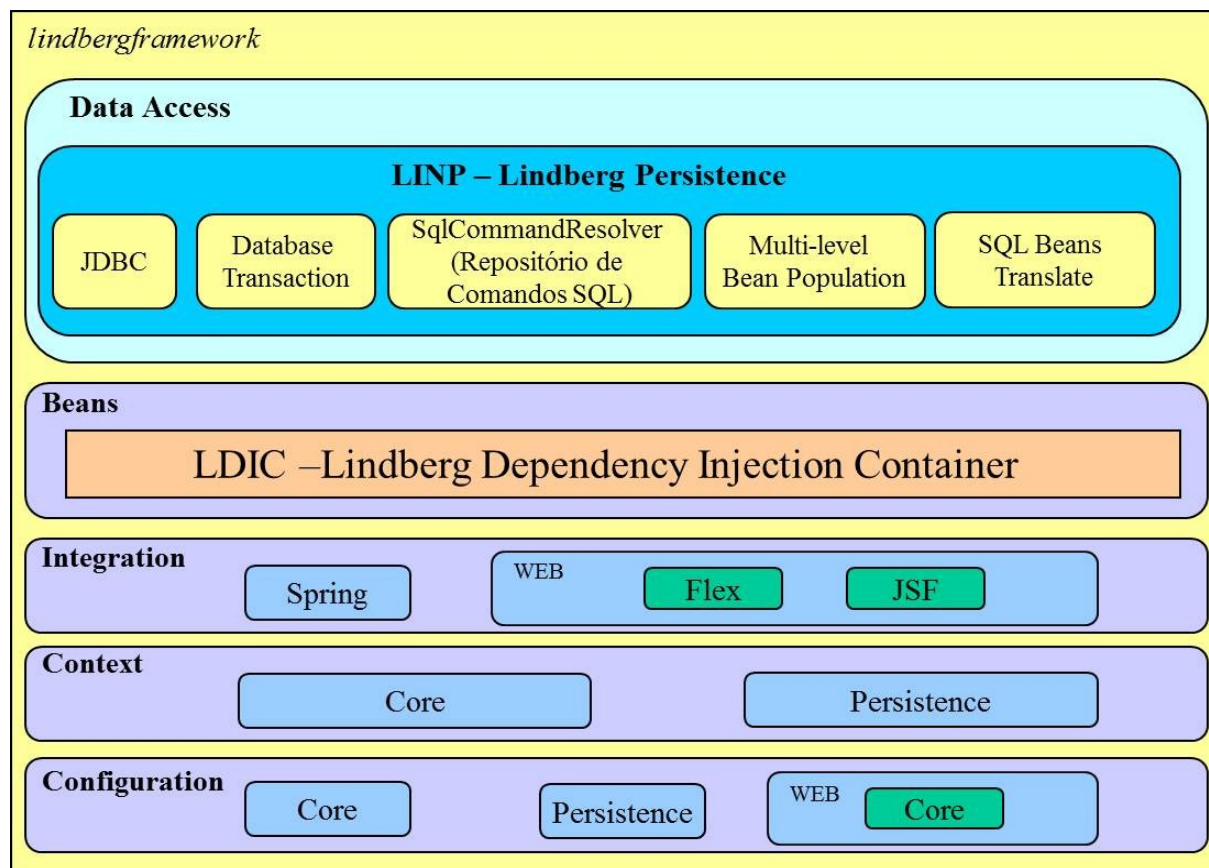
Antes de qualquer coisa, se deseja obter o lindbergframework a partir do maven veja no final deste documento a sessão que trata como importar o framework para projetos maven.

O lindbergframework é um projeto open source e que tem como objetivo prover um conjunto de recursos reutilizáveis para o desenvolvimento de aplicações de modo a sanar ou contribuir na solução de diversos problemas comuns a projetos JAVA.

O conceito do framework é o de “plug and use”, onde seus recursos são facilmente plugados/acoplados aos projetos de forma simples e rápida.

### 2.1 – Arquitetura

O framework é composto em módulos divididos em responsabilidades. Cada módulo contribui no gerenciamento e configuração dos recursos providos. A figura a seguir ilustra a arquitetura atual do lindbergframework na versão 1.x.



## 2.1.1 – Módulos e responsabilidades

- **Beans:** Responsável por todo gerenciamento de inversão de controle (IOC) e injeção de dependências que o framework provê. O **LDIC** (Lindberg Dependency Injection Container) é o mecanismo responsável por resolver as instancias de beans solicitados, bem como suas respectivas dependências, e injeta-los no lugar certo na hora certa de acordo com a necessidade de cada bean.
- **Integration:** Módulo responsável pela integração e operacionalidade entre o lindberg e outros frameworks. Na versão 1.x o framework fornece integração com Spring, Adobe Flex e JSF.
- **Context:** Responsável por toda configuração, beans e gerenciamento em nível de escopo de cada camada: core, persistence, web, etc...
- **Configuration:** Toda a parte de configuração do framework por escopo é responsabilidade deste módulo. Os escopos de configuração estão organizados em Core, Persistence e Web onde o web tem um escopo interno específico para configurações de core que inicializará as configurações nos outros escopos.

## 3 – O Framework.

O lindbergframework como qualquer outro framework é configurável e customizável de acordo com as necessidades de cada projeto. A interface *Configuration* define o contrato de classes que configuram o framework. Existem ainda duas interfaces que estendem desta ultima. Uma para configurações do core, *CoreConfiguration* e outra para configurações específicas do componente de persistencia, *LinpConfiguration*.

São fornecidas junto com a distribuição padrão do framework implementações para configurar o framework usando XML e programaticamente via código java direto. Abaixo são descritas as implementações padrão destas interfaces que não precisam ser conhecidas para usar o framework o próprio framework instancia e usa a implementação padrão de cada uma delas se uma implementação específica não for definida.

- **ClassPathXmlCoreConfiguration:** Implementação de *CoreConfiguration* para definição da configuração do módulo CORE via XML definido no classpath da aplicação, baseado no schema:  
<http://www.lindbergframework.org/schema/lindberg-config.xsd>.
- **SimpleCoreConfiguration:** Implementação de *CoreConfiguration* para definição da configuração do módulo CORE programaticamente.
- **SimpleLinpConfiguration:** Implementação de *LinpConfiguration* para definição da configuração do módulo de persistência (LINP) programaticamente.
- **XmlLinpConfiguration:** Implementação de *LinpConfiguration* para definição da configuração do módulo de persistência (LINP) via XML baseado no schema:  
<http://www.lindbergframework.org/schema/linp-sqlMapping.xsd>.

### 3.1 – Processo de Configuração

O processo de configuração é iniciado e gerenciado a partir do módulo CORE. O módulo core é configurado e este configura os outros módulos dependentes, caso algum esteja sendo usado, como por exemplo o módulo LINP.

A interface *CoreConfiguration* define um método chamado *getModules* que devolve as configurações dos módulos dependentes de modo que no momento da inicialização da configuração do CORE dentro de seu contexto os contextos dependentes que estão sendo usados são automaticamente configurados e inicializados. Este método devolve as implementações de *Configuration* que devem ser inicializadas. Cada implementação desta pode ser configuração via XML, programaticamente, usando arquivos *.properties*, arquivos *.txt*, enfim qualquer implementação que se desejar. Normalmente não haverá necessidade de se escrever implementações de *Configuration* pois o framework já fornece implementações padrão que devem ser utilizadas. Mas se for necessário estender qualquer uma dessas de modo a mudar seu comportamento padrão ou mesmo fornecer uma implementação específica, sintá-se a vontade mas saiba o que está fazendo.



## 3.2 - Contextos

Já foi dito que o framework é composto por módulos. Um contexto é definido dentro do framework como o escopo de beans e configuração de um módulo ou componente.

A interface *ComponentContext* define um contexto de módulo dentro do lindbergframework e duas implementações são definidas: contexto de Core e contexto de Persistência. As implementações de cada um desses contextos são respectivamente *CoreContext* e *LinpContext*. Definidos através de um *singleton* para todo sistema sendo necessário a chamada ao método *getInstance* definido em cada um deles para a obtenção da instância corrente.

Um contexto só pode ser usado se este estiver ativo. Para ativar um contexto o método *initialize* deve ser invocado passando como parâmetro a instancia da configuração sobre o qual o contexto se baseará.

A interface *ComponentContext* também define outros métodos auxiliares, como *isActive* para a checagem do status do contexto quanto a ativo e inativo, *finalize* para efetuar o encerramento do contexto e *verifyContext* para verificação da usabilidade do contexto. Qualquer tentativa de uso de um contexto sem usabilidade resultará em uma *IllegalStateException*.

Não é necessário se aprofundar no estudo dos contextos pois o framework faz todo o trabalho e interage com os contextos quando necessário e não é necessário se preocupar com nada referente a isso, pelo menos para o uso comum.

## 3.3 – Configuração do CORE

A configuração do CORE é composta por informações como pacote base para resolver beans para injeção de dependência, fábrica de beans, módulos dependentes que se deseja utilizar. As configurações de CORE são mais simples, pois dizem respeito apenas as configurações gerais que são definidas para o núcleo do framework.

As duas propriedades que podem ser configuradas no CORE são:

- *lindberg.core.di-basepackage* (*propriedade requerida*) :

Pacote base de beans para injeção.

Constante que define essa propriedade: *CoreConfiguration.CONFIG\_PROPERTY\_DI\_BASEPACKAGE*;

- *lindberg.core.beanfactory* (*se não for definida a fábrica padrão será usada - AnnotationBeanFactory*) :

Fábrica de beans a ser usada.

Constante que define essa propriedade: *CoreConfiguration.CONFIG\_PROPERTY\_BEAN\_FACTORY*;

Vamos definir uma configuração de CORE para os seguintes valores usando as duas implementações de *CoreConfiguration* na prática:

- Pacote base de beans para injeção (*lindberg.core.di-basepackage*) = *org.lindbergframework.exemplo.\**
- Bean factory (*lindberg.core.beanfactory*) = *org.lindbergframework.beans.di.context.AnnotationBeanFactory*

Estas duas propriedades são suficientes para configurarmos o core para usarmos apenas o mecanismo de injeção de dependência. Neste caso os beans para injeção serão procurados dentro do pacote *org.lindbergframework.exemplo* e o ponto asterisco “.” define que o framework também procurará beans para injeção de dependência dentro dos sub-pacotes de *org.lindbergframework.exemplo* e sub-pacotes destes e assim em diante.

O lindbergframework define uma interface chamada *BeanFactory* que define uma fábrica de beans que pode ser usada para obtenção de instâncias de beans dentro do contexto de injeção de dependência corrente. Neste caso foi definida a implementação de uma fábrica de beans que trabalha em conjunto com annotations. Mais a frente serão abordados os detalhes dessa interface *BeanFactory* e de todo o mecanismo de injeção de dependência.

### 3.3.1 – Configurando o CORE na Prática - Programaticamente

Abaixo é mostrado um exemplo da implementação e uso dessas configurações programaticamente direto com código java usando a implementação *SimpleCoreConfiguration*:

```
SimpleCoreConfiguration coreConfiguration = new SimpleCoreConfiguration();
coreConfiguration.setDiBasePackage("org.lindbergframework.exemplo.*");
coreConfiguration.setBeanFactory(new AnnotationBeanFactory());
CoreContext.getInstance().initialize(coreConfiguration);
```

**OBSERVAÇÃO:** O trecho de código

*CoreContext.getInstance().initialize(...)*

poderia ser substituído por

*coreConfiguration.initializeContext();* - Esse método em *CoreConfiguration* faz com que o contexto de CORE seja inicializado com esta configuração.

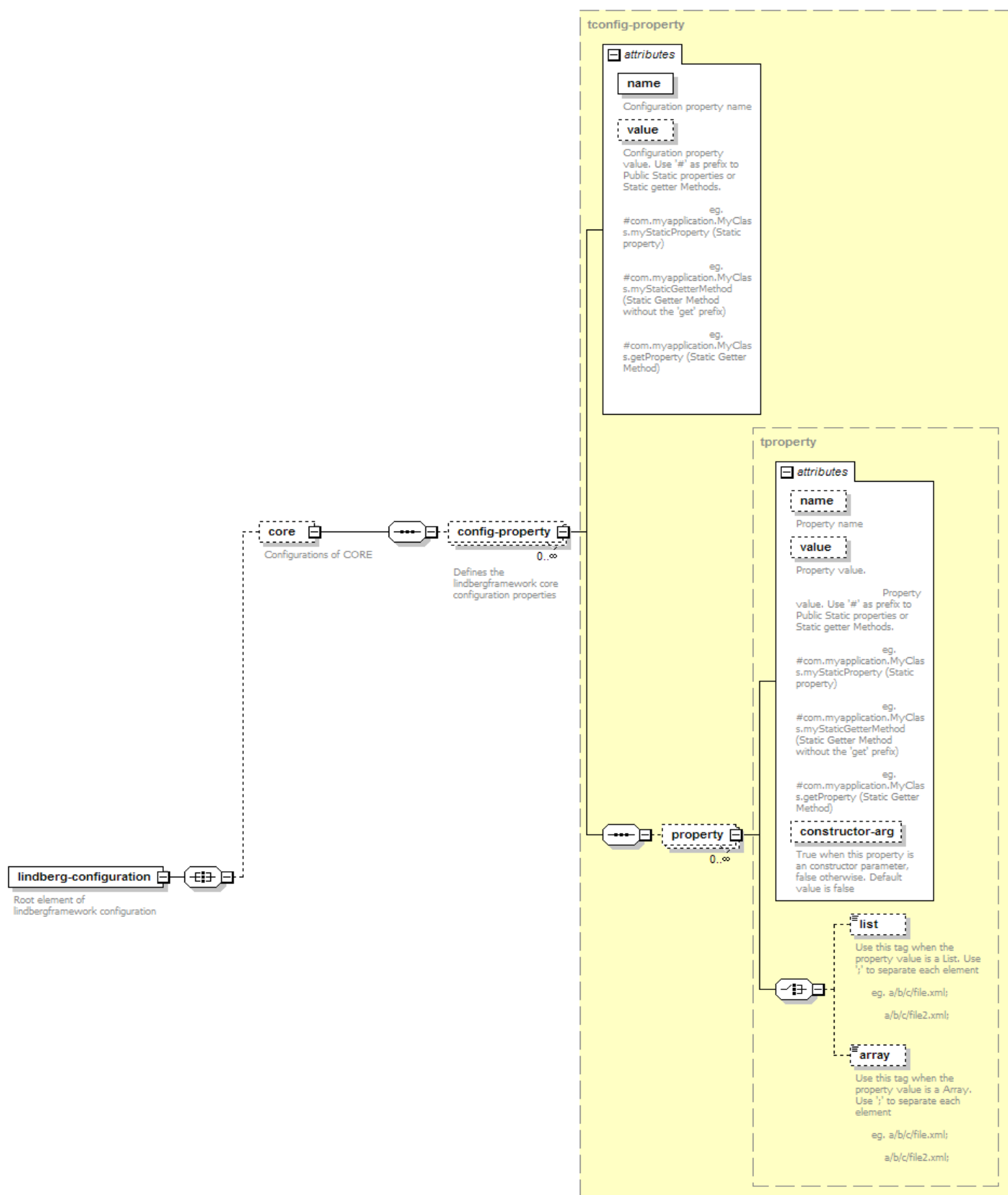
O trecho acima cria uma instância de *SimpleCoreConfiguration*, seta as propriedades pacote base onde encontrar os beans que poderão ser usados dentro do contexto de DI (dependency injection ou injeção de dependência) e a instância da fábrica de beans a ser usada pelo mecanismo de DI. Definidas as propriedades de configuração então o último passo é configurar o módulo core baseado nestas configurações. Para fazer isso é obtida a instância singleton de *CoreContext* via *CoreContext.getInstance()* e o método *initialize* passando a configuração criada é invocado. Pronto, o módulo CORE já está pronto para ser usado e em qualquer lugar você pode invocar *CoreContext.getInstance().getBeanFactory()* para obter a instância configurada da fábrica definida e chamar o método *getBean* da fábrica para obter uma instância de um bean desejado passando o “ID” deste. Um outro modo de se obter uma instância de um bean através da mesma fábrica é usar a classe *UserBeanContext* via *UserBeanContext.getInstance().getBean(id)*. O mecanismo de injeção de dependência será detalhado mais a frente, aqui está apenas sendo demonstrado os conceitos de configuração.

### 3.3.2 – Configurando o CORE na prática – Usando XML

A seguir é demonstrado a mesma configuração só que usando uma implementação de *CoreConfiguration* para configuração baseada em XML. O schema que define as configurações baseadas em XML é detalhado a seguir.

Schema: <http://www.lindbergframework.org/schema/lindberg-config.xsd>

O schema no que diz respeito a tag `<core>` para definições de configurações de CORE é ilustrado de forma detalhada na imagem abaixo:



Abaixo é mostrado um simples arquivo de configuração de core que define as mesmas configurações do exemplo anterior.

```
<?xml version="1.0" encoding="UTF-8"?>
<lindberg-configuration xmlns="http://www.lindbergframework.org/schema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.lindbergframework.org/schema/lindberg-config.xsd">

  <core>
    <config-property name="lindberg.core.di-basepackage"
      value="org.lindbergframework.exemplo.*"/>
    <config-property name="lindberg.core.beanfactory"
      value="org.lindbergframework.beans.di.context.AnnotationBeanFactory"/>
  </core>

</lindberg-configuration>
```

**Obs.:** A beanfactory está sendo definida mas lembre que esta propriedade não é requerida e quando não informada a factory padrão, *AnnotationBeanFactory*, é usada automaticamente pelo framework.

### 3.3.2.1 – A tag *<config-property>*

A tag *<config-property>* é a mais importante de todo o documento XML de configuração. Ela é usada para a definição de propriedades de configuração. Estas propriedades definem todo o comportamento dos componentes fornecidos pelo framework como o pacote base para buscar beans de injeção automática, gerenciador de transações, resolvedor de comandos sql, schema de banco padrão que deve ser usado, entre outros. Estes últimos são abordados mais a frente, quando o componente de persistência (LINP) for apresentado.

Esta é uma tag basicamente de *nome* e *valor*, onde o atributo *nome* define a chave da propriedade de configuração e o atributo *valor* define o valor que será usado na configuração para esta propriedade.

Esta tag fornece a flexibilidade de definição de propriedades que serão setadas na objeto *valor* desta propriedade no momento da inicialização e propriedades que serão usadas como parâmetro para um construtor específico.

Supondo que nós tivéssemos uma propriedade de configuração de CORE cujo a chave é *exemplo* e o valor para esta chave fosse a classe *com.aplicacao.Exemplo* e esta classe tivesse uma propriedade String chamada *propriedade1*, então a definição dessa propriedade seria:

- **Definindo a inicialização simples de propriedades dentro de uma propriedade de configuração**

```
<config-property name="exemplo" value="com.aplicacao.Exemplo">
  <property name="propriedade1" value="valor da propriedade1" />
</config-property>
```

Observe que foi usada uma nova tag dentro da tag *<config-property>*. Ela serve para definir propriedades de inicialização e parâmetros de construtor de propriedades de configuração. A tag *<property>* pode ser usada um número ilimitado de vezes dentro da tag *<config-property>* de acordo com a necessidade. Neste caso a utilizamos uma única vez, para definir o *set* da String '*valor da propriedade1*'

na propriedade de nome *propriedade1* na instância do objeto do tipo *com.aplicacao.Exemplo* para a chave de configuração *Exemplo*. A propriedade *propriedade1* será setada no momento da criação do objeto *com.aplicacao.Exemplo*. Se tivéssemos uma *propriedade2*, bastaria declarar uma outra tag `<property>` dentro de `<config-property>` da mesma forma que a *propriedade1*.

- **Definindo a inicialização de propriedades do tipo List ou Array dentro de uma propriedade de configuração**

Suponha agora que tivéssemos então as *propriedade2* e *propriedade3* e estas fossem respectivamente uma *List* e um *Array* e ambas de String. Esta tag fornece a flexibilidade de inicialização de listas e arrays de Strings diretamente pelo XML de configuração. Abaixo é mostrado como ficaria a tag `<config-property>` com a inicialização destas duas propriedades.

```
<config-property name="exemplo" value="com.aplicacao.Exemplo">
  <property name="propriedade1" value="valor da propriedade1" />
  <property name="propriedade2">
    <list>
      exemplo list string 1;
      exemplo list string 2;
      exemplo list string 3;
      exemplo list string N;
    </list>
  </property>
  <property name="propriedade3">
    <array>
      exemplo array string 1;
      exemplo array string 2;
      exemplo array string 3;
      exemplo array string N;
    </array>
  </property>
</config-property>
```

No trecho acima agora definimos duas novas propriedades *propriedade2* e *propriedade3*. A primeira é uma *List* e a segunda um *Array*. Observe que agora temos duas novas tags para serem usadas dentro da tag `<property>`, as tags `<list>` e `<array>`. A primeira define uma lista e a segunda um array, ambos de Strings. Os valores de cada uma dessas tags são definidos com o conteúdo entre a tag de abertura e fechamento onde cada elemento da *List* ou *Array* é separado por um carácter ';'. Dentro da tag `<property>` se tem a flexibilidade de usar qualquer um dos dois tipos sendo que nunca os dois ao mesmo tempo dentro da mesma `<property>`.

- **Inicialização de propriedades de configuração usando um construtor específico**

Agora suponhamos que a nossa classe *Exemplo* tenha um construtor que recebe uma String e uma coleção de Strings e queiramos usar este construtor para a criação do objeto *Exemplo* transformando a nossa definição da *propriedade1* e *propriedade2* em parâmetros para o construtor. Suponha também que ainda assim queiramos setar a propriedade *propriedade3* da mesma forma que o exemplo anterior.

A nossa nova definição da tag `<config-property>` ficaria da seguinte forma:

```
<config-property name="exemplo" value="com.aplicacao.Exemplo">
  <property value="valor da propriedade1" constructor-arg="true" />
  <property constructor-arg="true">
    <list>
      <exemplo list string 1;
      <exemplo list string 2;
      <exemplo list string 3;
      <exemplo list string N;
    </list>
  </property>
  <property name="propriedade3">
    <array>
      <exemplo list string 1;
      <exemplo list string 2;
      <exemplo list string 3;
      <exemplo list string N;
    </array>
  </property>
</config-property>
```

Observe que o que mudou neste exemplo foi que a tag que definiam as propriedades *propriedade1* e *propriedade2* agora não tem mais definidos seus atributos *nome*, pois agora são parâmetros de construtor e agora ao invés disto estas tags tem a propriedade *constructor-arg* definido para 'true', indicando que estes serão atributos de construtor. Então o construtor que o framework buscaria para criar a instancia de *Exemplo* seria:

```
public Exemplo(String, List<String>)
```

Após a chamada a este construtor e a criação da instancia, a propriedade *propriedade3* seria setada com o *Array de String* definido chamando o método *setPropriedade3*.

### 3.3.2.2 – Inicializando *CoreContext* usando configuração XML

Abaixo é mostrado o mesmo processo de configuração do *CORE* só que usando o XML. Para este exemplo vamos considerar que o XML acima esteja no pacote *org.lindbergframework.configuracao* e que o nome do XML é *lindberg-config.xml* seguindo o nome do schema.

```
XmlCoreConfiguration coreConfiguration =
    new ClassPathXmlCoreConfiguration("org/lindbergframework/configuracao/lindberg-config.xml");
coreConfiguration.initializeContext();
```

Observe que agora inicializamos o contexto de forma diferente do exemplo anterior.

Os dois exemplos resultarão na seguinte saída no console informando o status de configuração do framework:

```
INFO: Initializing Lindberg Core Context
INFO: Lindberg Core Context initialized
```

Neste último exemplo observe que foi utilizada a classe *ClassPathXmlCoreConfiguration* para criar a configuração. Essa classe é uma extensão natural de *XmlCoreConfiguration* para trabalhar com um arquivo de configuração que está dentro do classpath do projeto. Os dois exemplos mostrados ao final configuram o framework respectivamente a partir de:

```
- CoreContext.getInstance().initialize(coreConfiguration);
- coreConfiguration.initializeContext();
```

A partir daí o core do framework está configurado e podemos por exemplo obter uma instância de um bean a partir da fábrica. Supondo que no pacote *org.lindbergframework.exemplo.beans* tenha um bean cujo ID seja “*pessoaDAO*” então podemos obter uma instância desse bean chamando a BeanFactory como demonstrado a seguir, isso só pode ser feito após claro a devida configuração do CORE.

```
IPessoaDAO pessoaDAO = CoreContext.getInstance().getBeanFactory().getBean("pessoaDAO");
```

Uma outra forma de fazer a mesma coisa e obter o mesmo bean seria:

```
IPessoaDAO pessoaDAO = UserBeanContext.getInstance().getBean("pessoaDAO");
```

Todo o funcionamento do mecanismo de injeção de dependência será detalhado mais a frente. Aqui foi demonstrado apenas como exemplo do uso da configuração na prática.

### 3.3.2.3 – Uso do carácter '#' no XML para definir propriedades e métodos getters estáticos

Imagine que você quer por algum motivo que o framework chame um método 'get' seu para obter a instância da implementação de *BeanFactory* a ser usada, imagine que você queira isso pois por algum motivo você precise fazer algum processamento nesse método 'get' para efetuar alguma customização, registro de log, enfim. Ou imagine ainda que por algum motivo você quer que a String com pacote base (onde o framework encontrará os beans que farão parte do contexto de '*inversão de controle*' de modo que a criação, injeção de dependência desses seja responsabilidade do framework) esteja em numa constante em uma de suas classes de modo que você possa usá-la em outros lugares no seu projeto centralizando esse atributo na forma de uma constante.

Para estes casos o framework provê o recurso de usar o carácter coringa '#'. O uso desse carácter antes da definição de um valor de uma propriedade no arquivo XML de configuração informa que o valor real daquela propriedade será obtido através da chamada ao método 'get' estático e público ou da constante também estática e pública definidos na String ao qual o carácter '#' é fixado.

**Isso só é possível para propriedades e métodos que são estáticos e públicos e nos casos do método que seja um método seguindo o padrão de nomenclatura de métodos *getter*, não sendo para este último caso necessário preceder o nome do método no XML com 'get' pois o framework já busca o método adequado baseado no padrão de nomenclatura.**

Por exemplo, se quiséssemos que as duas propriedades mostradas anteriormente para o *CORE* fossem obtidas agora da seguinte forma:

- *di-basepackage*: A partir de uma constante estática e pública chamada *BASEPACKAGE* na classe *com.soft.MyConstants*;
- *BeanFactory*: A partir de um método getter público e estático chamado *getMyBeanFactory* na classe *com.soft.MyFactoryConfiguration*.



Então a nova configuração seria:

```
<core>
  <config-property name="lindberg.core.di-basepackage" value="#com.soft.MyConstants.BASEPACKAGE"/>
  <config-property name="lindberg.core.beanfactory" value="#com.soft.MyFactoryConfiguration.myBeanFactory"/>
</core>
```

Observe que agora os valores das propriedades estão precedidos do carácter '#'. Observe que o valor da propriedade *beanFactory*, que aponta para um método 'get', não contém o prefixo 'get' ou seja o nome completo do método 'getMyBeanFactory' e sim apenas 'myBeanFactory'. Isso pode ser feito pois o framework por padrão vai buscar o método 'get', seguindo os padrões de nomenclatura, fazendo 'get' + 'MyBeanFactory' e invoca o método 'getMyBeanFactory'. Da mesma forma caso deseje informar diretamente o nome do método o framework percebe e o invoca da mesma forma. Seguindo este pensamento a definição da propriedade *beanFactory* poderia ser feita sem nenhuma mudança no processamento da seguinte forma:

```
<config-property name="lindberg.core.beanfactory" value="#com.soft.MyFactoryConfiguration.getMyBeanFactory"/>
```

Agora informamos diretamente o nome completo do método *getMyBeanFactory*. O framework vai detectar que isso foi feito e não fará nenhuma conversão de nomenclatura, invocando diretamente o método informando.

**NOTA:** Apenas métodos e propriedades públicos e estáticos podem ser usados com '#'.

### 3.3.3 – A Classe CoreContext

A classe *CoreContext* é um *singleton* e provê acesso a toda configuração do módulo *CORE* do framework. Para obter a instância *singleton* do contexto de configuração de core basta chamar *CoreContext.getInstance*. As configurações definidas para o core como o pacote base para o contexto de injeção de dependências, a implementação da *BeanFactory*, e qualquer outra configuração pode ser acessada via *CoreContext*.

### 3.3.4 – Configurando em aplicações WEB

Até agora para configurar o framework, através de seu módulo principal, o *CORE*, seja programaticamente, via XML, ou qualquer outra implementação seja em nosso código seja a partir de um método *main*, seja a partir de uma classe que recebe essa responsabilidade ou qualquer outra estrutura temos que chamar sempre `CoreContext.getInstance().initialize(coreConfiguration);` ou `coreConfiguration.initializeContext();` para configurar e inicializar o framework. Em uma aplicação WEB é comum que um framework forneça recursos para que a configuração seja feita de forma automática, sem precisar de nada programaticamente.

Este recurso também é provido pelo *lindbergframework* através de um *ServletContextListener* e da definição de alguns *<context-param>* no *Deployment Descriptor* (*web.xml*) da aplicação. A partir daí quando o servidor for inicializado e a aplicação publicada o *lindbergframework* será automaticamente configurado baseado nas configurações passadas.



Essa configuração é muito simples. Baseia-se em declarar no *web.xml* da aplicação o *ServletContextListener* provido pelo *lindbergframework* e definir os parâmetros (*<context-param>*) informando onde está o arquivo de configuração do framework e qual a implementação do parser de configuração a ser usado, se será Xml, Properties, Txt, seja lá qual for, sendo que para este último é esperado uma implementação de *WebCoreConfiguration*.

Tanto o local onde se encontra o arquivo de configuração quando a implementação de *WebCoreConfiguration* a ser usada não são requeridos pois ambos possuem valores padrão. O path do arquivo de configuração caso não seja definido tem como padrão o valor *lindberg-config.xml* na raiz do classpath do projeto. E a implementação de *WebCoreConfiguration*, caso não seja declarada, é usada como padrão:

*org.lindbergframework.web.core.configuration.WebClassPathXmlCoreConfiguration.*

Abaixo é descrito um exemplo do que precisaria ser adicionado ao nosso *web.xml* se fossemos usar o mesmo arquivo de configuração que usamos nos exemplos anteriores só que agora em um projeto WEB para configurar o framework de forma automática:

```
<context-param><!--(ISTO É OPCIONAL. O caminho padrão para o arquivo de configuração é 'lindberg-  
config.xml' localizado na raiz do classpath)-->  
  <param-name>lindbergConfigLocation</param-name>  
  <param-value>org/lindbergframework/configuracao/lindberg-config.xml</param-value>  
</context-param>  
  
<context-param> <!--(ISTO É OPCIONAL)-->  
  <param-name>lindbergConfigClass</param-name>  
  <param-alue>org.lindbergframework.core.context.WebClassPathXmlCoreConfiguration</param-value>  
</context-param>  
  
<listener>  
  <listener-class>org.lindbergframework.web.LindbergContextLoaderListener</listener-class>  
</listener>
```

No exemplo acima observe que adicionamos o *listener LindbergContextLoaderListerner*, que é o *ServletContextListener* provido pelo *lindbergframework* e que sabe como configurar de forma automatizada o framework. Adicionamos o parâmetro *lindbergConfigLocation* que é onde está o arquivo de configuração. Esta propriedade é opcional e caso não seja declarada, o framework usa o nome e local padrão para o arquivo de configuração, que é *'lindberg-config.xml'* na raiz do *classpath*. Importante dizer que o arquivo não obrigatoriamente tem que ser um XML, isto depende da implementação de *WebCoreConfiguration* que estiver usando. Se, por exemplo, ocorresse a necessidade de usar uma implementação para configurações definidas em um arquivo *.properties* então este parâmetro apontaria para um arquivo *properties*, da mesma forma se eu criar minha própria implementação de *WebCoreConfiguration* que trabalha com um arquivo TXT segundo uma formatação que eu criei então esse parâmetro vai apontar para um arquivo TXT que segue o padrão de formatação definido. Para que haja essa flexibilidade, é que é possível definir a implementação de *WebCoreConfiguration* que deve ser usada através do parâmetro *lindbergConfigClass*.

#### Passos para configuração no **web.xml**:

- Declarar o listener: *org.lindbergframework.web.LindbergContextLoaderListener*
- Declarar os Parâmetros via *<context-param>*:
  - *lindbergConfigLocation*: Local onde está o arquivo de configuração caso o arquivo de configuração não siga o padrão é *'lindberg-config.xml'* na raiz do classpath.
  - *lindbergConfigClass*: Implementação de *CoreConfiguration* que efetuará o parser da configuração. Quando não definido este parâmetro a implementação padrão, *WebClassPathXmlCoreConfiguration*, é usada.

A partir daí é só iniciar o servidor que o a configuração será efetuada automaticamente e você verá no console as mensagens de log informando o processo de inicialização do contexto WEB do framework.

**ATENÇÃO:** Para alguns servidores é necessário adicionar a lib do *xmlbeans* direto nas libs do servidor de modo a sobrescrever a lib padrão provida pelo mesmo. Isso se faz necessário quando ao subir o servidor se obtenha uma mensagem de erro como a descrita abaixo indicando conflito de carregamento de classes do *xmlbeans*.

```
"...Loader constraint violation in interface itable initialization: when resolving method
"org.apache.xmlbeans.impl.store.Xobj$NodeXobj.setUserData(Ljava/lang/String;Ljava/lang/Object;Lorg/w3c/dom/UserDataHa
ndler;)Ljava/lang/Object;" the class loader (instance of org/jboss/classloader/spi/base/BaseClassLoader) of the
current class, org/apache/xmlbeans/impl/store/Xobj$NodeXobj, and the class loader (instance of <bootloader>) for
interface org/w3c/dom/Node have different Class objects for the type org/w3c/dom/UserDataHandler used in the
signature"
```

## 4 – Injeção de Dependência e Inversão de Controle com o lindbergframework.

Um conceito importante abordado por muitos frameworks consolidados na comunidade java, como o spring, é o de inversão de controle para o tratamento de injeção de dependências. Esse conceito propõe a transferência de responsabilidade para a criação de beans no projeto fazendo com que o controle sobre a criação, gerenciamento e injeção de dependências destes beans não fique a cargo do programador e seja invertido para um *container* ou qualquer outro componente que possa tomar controle sobre a execução.

O lindbergframework precisa em muitos casos intervir e customizar a criação de beans para aplicar correta e eficientemente injeção de dependência nos mesmos. Uma outra necessidade de intervenção na criação de beans é para usar diversos tipos de proxies de aspecto para vários fins, como por exemplo no gerenciamento de transações via annotation. Para fazer isso, o *lindbergframework* aplica um proxy sobre o bean que gerenciará todo o processo de transação de forma transparente. Mas para tal, é necessário que o controle de criação dos beans seja invertido para o LDIC (*Lindberg Dependency Injection Container*). Toda a parte do gerenciamento de transações do *framework* será detalhada mais a frente quando a parte de persistência for abordada.

Um framework já consolidado que poderia ter sido adotado é o *spring* por exemplo, mas isso obrigaria que todo projeto que usasse o *lindbergframework* também tivesse que usar o *spring*. Para solucionar o problema um mecanismo de inversão de controle foi criado de forma independente e exclusiva para o *lindbergframework*. Caso você deseje mesmo assim usar spring em seu projeto o *lindbergframework* no módulo integration fornece integração com este framework. Isto será demonstrado mais a frente.

**NOTA: Observe que ao se usar o lindbergframework, principalmente os recursos de persistência, é necessária a dependência para o Spring. Isso não é necessário para efetuar injeção de dependências. O mecanismo de injeção do lindbergframework é independente e implementa outra solução para o problema. A dependência do Spring é requerida pois o lindbergframework usa recursos do SpringDAO.**

### 4.1 – Inversão de Controle na Prática

O mecanismo de inversão de controle do lindbergframework é simples e baseia-se inicialmente no pacote base onde o Lindberg Dependency Injection Container (LDIC - Contêiner Lindberg de Injeção de Dependência), responsável por resolver todas as dependências de cada bean, deve procurar os beans. No exemplo anteriormente mostrado o pacote *org.lindbergframework.exemplo.\** foi definido e isso fará com que o LDIC crie seu contexto de beans visualizando apenas beans que estão dentro do escopo do pacote *org.lindbergframework.exemplo* ou sub pacotes deste já que o “.” foi definido. Abaixo vão algumas regras para a definição do pacote base:

- 1 – O pacote deve ser definido usando o carácter “.”;
- 2 – Pacotes usando o carácter “/” como por exemplo “org/lindbergframework/xxx” são inválidos;

3 – Para informar que o escopo se restrinja a um pacote único e específico e somente este basta definir o pacote sem “.\*”. Por exemplo: *br.empresa.beans* define que apenas os beans dentro deste pacote serão visualizados. Qualquer bean dentro de um sub pacote deste não será encontrado como por exemplo um bean dentro de *br.empresa.beans.exemplo* não será visualizado;

4 – Para definir que o escopo é um pacote e incluir no escopo todo e qualquer bean que esteja em algum sub pacote do pacote base, como no exemplo anterior *br.empresa.beans* como pacote base. Para que um bean dentro de *br.empresa.beans.exemplo* também seja visualizado e incluído no contexto o pacote base deve ser definido usando “.\*” como a seguir:

*br.empresa.beans.\**

5 – Também é possível definir um pacote base usando seus sub pacotes mas excluindo do contexto alguns sub pacotes que não se deseja que façam parte do contexto de beans. Por exemplo, vamos levar em consideração a seguinte estrutura de pacotes:

```
br
br.empresa
br.empresa.beans
br.empresa.beans.exemplo
br.empresa.beans.schemas
```

Se fosse desejável criar um contexto de beans onde o LDIC visualizasse beans dentro do pacote *br.empresa.beans* e seus sub pacotes mas que o pacote *br.empresa.beans.schemas* não fizesse parte do contexto. Para tal, é necessário definir o pacote ou pacotes que se deseja excluir do contexto usando o caráter “:”. Para este exemplo a string do pacote básico seria da seguinte forma:

*br.empresa.beans:schemas.\**

O pacote definido dessa forma diz que o LDIC deve buscar beans no pacote *br.empresa.beans* e todos os seus sub pacotes menos o sub pacote *schemas*. Todo e qualquer bean que estiver dentro de *br.empresa.beans.schemas* os sub pacotes deste não será visualizado pelo LDIC.

## 4.2 – Fábrica de Beans (*BeanFactory*)

A interface *BeanFactory* é fornecida pelo framework para definir as fábricas de beans que poderão trabalhar em conjunto com o LDIC. Uma classe que implementa essa interface deve fornecer instâncias de beans solicitados de acordo com um ID específico entre outras operações. Cada fábrica implementa a sua forma como cada instância de bean é obtida bem como cada dependência desses beans é resolvida e injetada no bean.

Uma outra interface importante mas que não será abordada mais a fundo neste momento e que trabalha em conjunto com as fábricas de beans é a *BeanMapper*. Essa interface define um mapeador de

beans de modo a fornecer meta dados a uma *BeanFactory* para a correta criação e resolução de dependências de um bean com base em um ID específico solicitado.

A classe abstrata *AbstractBeanFactory* que fornece a base para fábricas de beans já contém um atributo *BeanMapper* de modo a fornecer um mapeador de bean para as fábricas concretas.

## 4.2.1 – Injeção de Dependências

Fornecer um mecanismo para obtenção de instâncias de beans é um recurso importante. Imagine que temos uma classe chamada “A” que possui uma dependência para uma classe “B” e esta última por sua vez possui uma dependência para “C”. Se solicitarmos uma instância de “A” é necessário que a instância de “A” venha preenchida com uma instância de “B” e “B” com uma instância de “C” de modo que todas as dependências diretas e indiretas de “A” estejam resolvidas e dessa forma “A” esteja pronto para ser usado. Isso deve ser feito de forma transparente pelo mecanismo de inversão de controle de modo a fornecer recursos para diminuir o acoplamento entre as classes permitindo, por exemplo, que “A” se refira a “B” e “B” se refira a “C” através de uma interface tornando o código independente da implementação final.

Toda e qualquer interação entre “A”, “B” e “C”, poderia ocorrer através de interfaces onde o desenvolvedor não se preocupe com a implementação que em tempo de execução será utilizada. A instância do bean que contém a implementação a ser usada é responsabilidade do mecanismo de injeção de dependência que deve obter a correta instância e injetá-la onde necessário.

**NOTA: Injeção de dependências não está ligado ao uso de interfaces ou não. Mesmo que não se use interfaces nem classes abstratas para prover o baixo acoplamento entra as camadas o uso de inversão de controle para injeção de dependências é algo independente e importante.**

O lindbergframework provê a interface *DependencyManager* que define um gerenciador de dependências que tem como responsabilidade auxiliar uma *BeanFactory* resolvendo as dependências dos beans solicitados. A implementação padrão desta interface é um gerenciador de dependências baseado nas annotations *@Bean* e *@Inject* – *AnnotationDependencyManager*.

Para exemplificar considere o código abaixo que não usa a injeção automática de dependência:

```
public interface ISistemaFacade {  
  
    public void cadastrarPessoa(Pessoa pessoa);  
  
}  
  
public class SistemaFacade implements ISistemaFacade{  
  
    private IPessoaBC pessoaBC = new PessoaBC();  
  
    public void cadastrarPessoa(Pessoa pessoa){  
        pessoaBC.cadastrar(pessoa);  
    }  
  
}  
  
public interface IPessoaBC {  
  
    public void cadastrar(Pessoa pessoa);  
  
}
```

```

}

public class PessoaBC implements IPessoaBC{

    private IPessoaDAO pessoaDAO = new PessoaDAO();

    public void cadastrar(Pessoa pessoa) {
        pessoaDAO.cadastrar(pessoa);
    }

}

public interface IPessoaDAO {

    public void cadastrar(Pessoa pessoa);

}

public class PessoaDAO implements IPessoaDAO{

    public void cadastrar(Pessoa pessoa){
        System.out.println("Pessoa cadastrada: "+pessoa.getNome());
    }

}

```

### Código de teste:

```

Pessoa pessoa = new Pessoa("joão");
ISistemaFacade sistemaFacade = new SistemaFacade();
sistemaFacade.cadastrarPessoa(pessoa);

```

Observe que as classes demonstradas acima não usam recurso de inversão de controle para provimento de injeção de dependências e referem-se diretamente as implementações finais das classes ao qual dependem.

Neste caso observe que a fachada do nosso sistema depende de um *BusinessController* que implementa a interface *IPessoaBC* e a instância é criada diretamente via operador *new* no código tornando a fachada do nosso sistema extremamente dependente da implementação *PessoaBC* da interface *IPessoaBC*. Da mesma forma a implementação *PessoaBC* depende de uma implementação de *IPessoaDAO* e como foi feito na fachada a implementação *PessoaDAO* está diretamente referenciada no código e a instância criada diretamente no código. Neste exemplo os três níveis estão extremamente interdependentes, dificultando qualquer evolução ou manutenção no código pois o código está extremamente amarrado a implementações e não a interfaces mesmo que estejamos usando interfaces pois a implementação final é referida diretamente no código.

Qual o problema nisto? Um exemplo simples seria: Atualmente a maioria das aplicações trabalham com testes (JUnit ou algo similar). Uma pratica comum na implementação de testes é a criação de *Mocks* e *Fakes*. Que a grosso modo são objetos que simulam o real comportamento de objetos provendo para tal o comportamento desejado/esperado para um determinado teste. É comum o uso desse tipo de objetos para a criação de testes desconectados do banco. De modo que um mock seria uma implementação, por exemplo, de um DAO real só que simulando as reais operações de persistência bem como o comportamento desejado para um ou mais testes. Dito isto, no exemplo acima não seria possível normalmente (seria caso usássemos Aspecto) fornecer um *Mock* de *IPessoaDAO* para fins de testes pois o BC *PessoaBC* acessa diretamente a implementação real de *IPessoaDAO* (*new PessoaDAO*) e isso impossibilita a mudança da implementação do DAO a ser usada pois o BC está diretamente acoplado a implementação do DAO.

Se o BC se referisse ao DAO apenas via interface e ao invés de obter a instância da implementação

diretamente via operador *new* usasse um mecanismo de inversão de controle para injetar a implementação de *IPessoaDAO* desejada, poderíamos para os testes usar a implementação *Mock* e para aplicação real a implementação real.

Um mecanismo de inversão de controle faz todo esse trabalho deixando o desenvolvedor livre para tratar do que realmente interessa de modo que a criação e injeção dessas dependências são responsabilidades desse mecanismo, que no *lindbergframework* é chamado de LDIC (Lindberg Dependency Injection Container).

O *lindbergframework* fornece uma implementação padrão de *BeanFactory* que é a classe *AnnotationBeanFactory*. Essa fábrica de beans trabalha com as annotations *@Bean* e *@Inject* que são utilizadas para a definição de beans e suas dependências. Essa *BeanFactory* é a padrão e caso uma não seja definida o framework utilizará esta como padrão.

- **@Bean:** Annotation utilizada para a definir um bean. Os únicos parâmetros que ela tem é o *value* e *singleton*. O *value* é uma String que define o ID do bean. Esse ID deve ser único no contexto e esse ID é o que será usado para referenciar o bean. O ID é requerido nesta annotation. O parâmetro *singleton* é um boolean que recebe “true” quando o bean deve ser um *singleton*, ou seja ter apenas uma instância dentro do contexto retornando sempre a mesma instância deste e “false” caso contrário.

- **@Inject:** Annotation que define um ponto de injeção de dependência em um bean. O único parâmetro que esta annotation contém é o *value* que é o ID do bean que deve ser injetado no atributo de classe onde esta annotation foi declarada.

Abaixo é mostrado o mesmo exemplo só que fazendo uso dos recursos de inversão de controle do *lindbergframework*. No exemplo a seguir considere que no ponto de obtenção da instância do bean o *Core* do framework já foi configurado como mostrado nas sessões anteriores. Como só as implementações foram alteradas as interfaces são omitidas neste exemplo:

```
import org.lindbergframework.beans.di.annotation.Bean;
import org.lindbergframework.beans.di.annotation.Inject;
```

```
@Bean("sistemaFacade")
public class SistemaFacade implements ISistemaFacade{

    @Inject("pessoaBC")
    private IPessoaBC pessoaBC;

    public void cadastrarPessoa(Pessoa pessoa){
        pessoaBC.cadastrar(pessoa);
    }
}
```

```
import org.lindbergframework.beans.di.annotation.Bean;
import org.lindbergframework.beans.di.annotation.Inject;
```

```
@Bean("pessoaBC")
public class PessoaBC implements IPessoaBC{

    @Inject("pessoaDAO")
    private IPessoaDAO pessoaDAO;

    public void cadastrar(Pessoa pessoa) {
        pessoaDAO.cadastrar(pessoa);
    }
}
```



```
import org.lindbergframework.beans.di.annotation.Bean;

@Bean("pessoaDAO")
public class PessoaDAO implements IPessoaDAO{

    public void cadastrar(Pessoa pessoa){
        System.out.println("Pessoa cadastrada: "+pessoa.getNome());
    }
}
```

### Código de teste:

```
Pessoa pessoa = new Pessoa("joão");
ISistemaFacade sistemaFacade = UserBeanContext.getInstance().getBean("sistemaFacade");
sistemaFacade.cadastrarPessoa(pessoa);
```

Observe que neste segundo exemplo as classes não referenciam como dependência nenhum implementação específica. *SistemaFacade* referencia apenas a interface *IPessoaBC* e não é afetada quanto a implementação que será usada. Da mesma forma *PessoaBC* referencia apenas a interface *IPessoaDAO* e seu funcionamento não depende e nem está acoplado a nenhuma implementação desta.

Neste último exemplo a classe *SistemaFacade* é anotada com a annotation *@Bean* onde o ID deste bean é definido como *sistemaFacade*. Este bean possui uma dependência com um *IPessoaBC* e para definir e instruir ao LDIC que faça a injeção da implementação que deve ser usada, este atributo foi anotado com a annotation *@Inject* definindo como parâmetro o ID do bean que deve ser injetado neste ponto. Observe que não há necessidade de definição de um *setter* para *pessoaBC*.

Da mesma forma a nossa implementação de *IPessoaBC* a classe *PessoaBC* é anotada da mesma forma, uma annotation *@Bean* definindo o ID desta e como temos apenas uma dependência, apenas uma annotation *@Inject* foi usada, neste caso para a dependência para *IPessoaDAO*.

## 4.2.2 – Obtendo beans a partir do Contexto

Com as classes devidamente anotadas precisamos obter uma instância de nossa fachada e utilizá-la. É aí que entram as configurações feitas no core e a nossa bean factory. No trecho abaixo obtemos do *CoreContext* a instância da *BeanFacotry* e a partir desta fábrica obtemos a instância do bean da fachada do sistema e para isso passamos o ID do bean da fachada que neste caso é “*sistemaFacade*” para o método *getBean* da *BeanFactory*. A instância do bean da fachada retornada por este método já virá pronta com suas dependências injetadas e prontas para uso.

Para este caso a instância de *sistemaFacade* já virá preenchida com a instância correta de *pessoaBC* e este por sua vez já estará preenchido com a instância correta de *pessoaDAO* tudo de forma automática e transparente feita pelo LDIC.

```
ISistemaFacade sistemaFacade = CoreContext.getInstance().getBeanFactory().getBean("sistemaFacade");
```

Uma outra forma e até mais simples e direta de se obter um bean a partir do contexto usando o mecanismo de inversão e controle, é a partir da classe *UserBeanContext*. Essa classe é um *singleton* e é um atalho para acesso da fábrica de beans padrão definida. O mesmo trecho de código apresentando anteriormente para obtenção de uma instância como exemplo de *IsistemaFacade* usando *CoreContext*



poderi ser feita usando *UserBeanContext* da seguinte forma:

```
ISistemaFacade sistemaFacade = UserBeanContext.getInstance().getBean("sistemaFacade");
```

Observe que no trecho acima apenas a interface é referenciada. Fica a cargo do LDIC retornar a instância da implementação que deve ser usada correspondente ao ID passado.

O LDIC também efetua a injeção em dependências definidas em *superclasse* de modo que se *PessoaBC* estendesse de uma classe qualquer e esta última tivesse dependências definidas via *@Inject* estas dependências herdadas, mesmo *private*, seriam resolvidas e injetadas.

Caso o ID passado para o método *getBean* não corresponda a um bean válido dentro do contexto da bean factory, uma *BeanNotFoundException* é lançada informando que um bean com o ID especificado não foi encontrado.

O *lindbergframework*, já se integra com JSF e Adobe Flex e usando o framework integrado com qualquer um destes últimos não é necessário o uso de *UserBeanContext.getInstance().getBean()* pois no caso do JSF o *managedBean* já é criado e suas dependências resolvidas automaticamente pelo *LDIC* e no caso do Flex o bean invocado para execução de um serviço ou método remoto é resolvido pelo *LDIC*. Toda essa parte de integração será detalhada mais a frente em um capítulo específico sobre integração.

### 4.2.3 – Beans com escopo *Singleton*

O framework também provê a possibilidade de definir um bean de modo que este seja um singleton. Isto é, tenha apenas uma instância dentro do contexto. Quando um bean é definido com o escopo *singleton* o *LDIC* manterá sempre a mesma instância e sempre que o bean for solicitado ao *LDIC* esta mesma instância será retornada. Se fosse desejável, por exemplo, definir que o bean *SistemaFacade* deve ter uma única instância dentro do contexto, então especificaríamos seu escopo como *singleton* na própria annotation *@Bean*. Abaixo é mostrado como ficaria a classe *SistemaFacade* de modo a operar como um *singleton*.

```
@Bean(value = "sistemaFacade", singleton = true)
public class SistemaFacade implements ISistemaFacade{}
```

Observe que a annotation *@Bean* foi alterada adicionando o atributo *singleton* definindo o valor deste como *true* indicando que o bean *SistemaFacade* deve ser trabalhar como um *singleton* dentro do contexto. O LDIC agora retornará sempre a mesma instância de *SistemaFacade* quando este bean for solicitado.

## 4.3 – Integrando o Contexto de Beans do *lindbergframework* com outros frameworks

É possível usar o mecanismo de inversão de controle para gerenciamento e injeção de beans do *lindbergframework* integrado com outros frameworks de modo a possibilitar para estes a obtenção de instâncias de beans a partir do contexto.

### 4.3.1 – Integrando com JSF (Java Server Faces)

A integração do JSF com o contexto de Beans do *lindbergframework* possibilita o uso de qualquer *Bean* declarado com annotation *@Bean* e que faça parte do contexto nas páginas JSF, diretamente via o ID do *Bean*.

**Exemplo:** Considere que temos um *managedBean* chamado *ManterPessoaMB* cujo ID do mesmo, declarado via annotation *@Bean* seja *manterPessoaMB*.

```
@Bean("manterPessoaMB")
public class ManterPessoaMB {

    public String cadastrar(){
        //... implementação da ação de cadastrar
        return null;
    }
}
```

**ATENÇÃO:** Para que o *bean* acima faça parte do contexto de beans o mesmo deve estar direta ou indiretamente abaixo do pacote base, definido na configuração de *CORE*.

Para efetuar a integração e possibilitar que usemos o bean *manterPessoaMB* diretamente em páginas JSF a partir do seu ID, é necessário declarar apenas o *ELResolver* do *lindbergframework*, *org.lindbergframework.integration.web.jsf.beans.LindbergBeanJsResolver* no *faces-config.xml* da aplicação como abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
    version="2.0">

    <application>
        <el-resolver>org.lindbergframework.integration.web.jsf.beans.LindbergBeanJsResolver</el-resolver>
    </application>

</faces-config>
```

Basta isso para que o contexto de beans do *lindbergframework* esteja acessível nas páginas JSF. Abaixo é mostrado um exemplo de um botão que acessa o bean *ManterPessoaMB* a partir do seu ID definido via annotation *@Bean*, *manterPessoaMB*, e chama o action '*cadastrar*' declarado no mesmo.

```
<h:commandButton action="{manterPessoaMB.cadastrar}" value="Cadastrar" />
```

### 4.3.2 – Integrando com Adobe Flex via Blazeds

Da mesma forma como foi apresentada no tópico anterior a integração com JSF, é possível integrar o contexto de beans com o Adobe Flex a partir do BlazeDS. Para tal, basta declarar a implementação de *flex.messaging.FlexFactory* do *lindbergframework*, que promove a integração, no arquivo *services-config.xml* como a seguir. A partir daí os beans contidos no contexto estão acessíveis de forma transparente para a camada de visão/front-end Flex. Da mesma forma como foi apresentada a integração do contexto com o JSF, os beans são acessados via seus respectivos ID's.

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <factories>
    <factory id="lindbergFactory" class="org.lindbergframework.integration.web.flex.LindbergFlexBeanFactory" />
  </factories>
</services-config>
```

## 5 – Lindberg Persistence – LINP.

Persistência é algo muito importante e muitas soluções já foram desenvolvidas provendo suporte de modo a melhorar e facilitar o desenvolvimento no que diz respeito a este conceito. Entre as principais soluções atualmente adotadas pelos desenvolvedores JAVA estão os frameworks ORM – Modelagem Objeto Relacional – que dão suporte a um mapeamento entre o modelo relacional do banco de dados e o modelo orientado a objetos do projeto JAVA de forma mais clara e transparente gerando código SQL e fazendo toda a parte pesada da implementação das operações de persistência.

O problema é: e quando o projeto em questão não adota uma dessas soluções ORM? Neste tipo de caso o que temos de recurso para a camada de persistência? Existem diversos componentes que auxiliam cada um a sua maneira na camada de persistência. O *SpringDAO* por exemplo, é um conjunto de recursos providos pelo framework *Spring* para suporte a persistência mas que não fornece todo o recurso que frameworks como Hibernate fornecem.

O JDBC (*Java Database Connectivity*) é a API padrão java para persistência. O conjunto de classes providas pelo JDBC fornece o recurso de acesso a qualquer banco de dados relacional baseado em um Driver. Os detalhes sobre JDBC fogem do escopo deste documento.

JDBC é poderoso e nos fornece muitos recursos para troca de informações com uma base de dados só que este lhe dá as ferramentas a grosso modo cabe a você usá-las, adaptá-las e/ou evolui-las. Operações como por exemplo, chamar uma *stored procedure* que retorne um ou dois cursores e mapear cada um desses cursores em objetos java, executar uma *query* SQL e popular o resultado uma lista de objetos, chamar uma *stored function* que retorna como resultado um cursor e da mesma forma popular este cursor em objetos java, essas e outras operações ainda são muito custosas em nível de implementação e uma vez feita a mesma solução adotada para um caso é repetida de forma semelhante para outros casos da mesma natureza, se tornando em alguns casos até tediosa.

Considere por exemplo uma classe *ContribuinteDAO* que conteria provavelmente um método chamado *getContribuintes* que retornaria uma lista de objetos *Contribuinte*. Considere também que a implementação da classe *ContribuinteDAO* use JDBC sem nenhum componente ou framework como suporte. Esse método com certeza seria implementado seguindo o seguinte algoritmo:

1. Obter conexão;
2. Obter uma *Statement* a partir da conexão;
3. Executar a query “select \* from tabela\_de\_contribuinte” na *Statement* ;
4. Obter um *ResultSet* a partir da execução da query no *Statement*;
5. Criar uma lista vazia de objetos *Contribuinte*;
6. Fazer um loop percorrendo todo o *ResultSet*;
7. Criar um Objeto *Contribuinte* e populá-lo com os dados do registro atual do cursor do *ResultSet* em cada iteração do loop;
8. Adicionar o objeto *Contribuinte* criado e populado na lista de *Contribuintes* criada;
9. Retornar a lista de *Contribuintes*.

O algoritmo acima poderia ser implementado no seguinte método java e qualquer outra listagem seja

lá qual for a entidade, manteria o mesmo padrão mudando apenas o *sql* da query, o objeto criado e a forma como cada objeto é populado. Isso sem contar a eterna obrigação, quando usando JDBC direto, de tratar as exceções checadas como *SQLException*.

```
public List<Contribuinte> listarContribuintes() {
    Connection conn = ConexaoUtil.getConexao();
    List<Contribuinte> listaContribuintes = new ArrayList<Contribuinte>();
    try {
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery("Select * from tb_contribuinte");
        while (rs.next()) {
            Contribuinte contribuinte = new Contribuinte();
            contribuinte.setNomeFantasia(rs.getString("nome_fantasia"));
            contribuinte.setCnpj(rs.getString("cnpj"));

            Endereco endereco = new Endereco();
            endereco.setBairro(rs.getString("bairro"));
            endereco.setCep(rs.getString("cep"));
            endereco.setRua(rs.getString("rua"));
            endereco.setNumero(rs.getInt("numero"));

            contribuinte.setEndereco(endereco);

            listaContribuintes.add(contribuinte);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return listaContribuintes;
}
```

Na implementação acima usando apenas JDBC direto observe que foi feita uma simples consulta de contribuinte onde no nosso exemplo contém apenas alguns campos e a parte da população do *ResultSet* em objetos *Contribuinte* já demandou várias linhas de código. Considerando que o objeto *Contribuinte* é o primeiro nível, neste caso, temos um segundo nível que é o objeto *Endereco* que está dentro do objeto *Contribuinte* formando assim um segundo nível de atributos. Na população do endereço foi necessário criar um objeto *Endereco* populá-lo para só então setar o endereço no objeto *Contribuinte*. No nosso exemplo foram apenas alguns campos. Imagine uma consulta real de contribuinte quantas dezenas de campos teríamos e em quantos níveis compondo objetos dentro de outros objetos e que cada um destes precisam ser populados de forma correta?

O LINP – Lindberg Persistence provê uma solução que busca simplificar e abstrair o desenvolvedor de todo esse trabalho agregando valor e novos recursos a camada de persistência, mantendo o mesmo focado em outras coisas mais relevantes.

## 5.1 – Conceitos Básicos

O LINP é muito simples e seu funcionamento é baseado em um repositório de comandos sql e um template com vários recursos otimizados para persistência via JDBC. Esse é o conceito básico do componente. A diferença está em como o framework provê esse repositório de comandos e nos recursos providos pelo template. Essas são as bases principais a grosso modo do componente.

Um outro ponto importante que a solução do componente aborda é o tratamento transparente e abstraído de um importante aspecto transversal da parte persistência que é a parte de gerenciamento de

transações que no LINP por padrão é feito a partir de annotations de forma simples, automática e transparente e que será abordada mais a frente.

A ideia básica é ter um repositório de comandos sql seja em arquivos XML, properties, Webservice, etc... fornecendo comandos sql, o que importa é o conceito de ter um local onde hospedar os comandos sql e a partir daí solicitar comandos sql baseados em ID's e estejam esses comandos onde estiver o framework vai lá e o obtém.

O primeiro passo do conceito é a obtenção do comando sql. Após isso é a execução deste a partir de um template. Tudo isso é apenas uma base, tudo será abordado e explicado a medida que esse capítulo for avançando.

A configuração do componente também é importante e imprescindível pois é necessário definir as configurações de acesso a base de dados, como o framework encontrará o repositório de comandos, como é implementado o repositório, configurar a classe responsável por mapear e fazer o intercâmbio de informações entre a aplicação e o repositório de comandos. Os detalhes da configuração do LINP serão abordados e detalhados mais a frente neste capítulo.

## 5.2 – Repositório de Comandos SQL - Introdução

O LINP trabalha com um repositório de comandos sql que nada mais é que uma classe que implementa a interface *SqlCommandResolver*. Um *SqlCommandResolver* é uma classe que é responsável por atender solicitações de comandos sql baseados em um ID e devolver o comando completo e montado de acordo com a solicitação.

A ideia principal é ter um DAO que forneça os recursos do componente e uma vez que seu DAO estende este DAO “turbinado” ele agora tem os recursos principais como obtenção e uso de comandos sql alocados no repositório configurado, população de objetos multi-nível (será abordada mais a frente), chamada fácil a stored procedures e stored functions, obtenção de de cursores populados em objetos prontos originados de parâmetros de saída dessas procedures e functions.

O repositório de comandos sql depende de implementação usada. Este pode ser vários arquivos xml, arquivos properties, arquivos txt, um web service, uma base de dados com comandos armazenados, enfim para o framework não importa como o repositório é implementado e sim a classe responsável por atender as solicitações por estes comandos, resolvê-las e devolver uma resposta a requisição. Esta classe, que deve implementar *SqlCommandResolver*, faz o intercambio entre a aplicação e o repositório de comandos sql.

A próxima sessão mostrará os detalhes da configuração do componente e mais a frente todos esses conceitos serão mostrados na prática.

## 5.3 – Configuração do LINP

Anteriormente foi mostrada a configuração do módulo CORE do framework. O componente LINP

também necessita de configuração e da mesma forma pode ser configurado por padrão usando XML ou direto no código java programaticamente. A implementação do *parser* da configuração funciona baseado em interface, como no CORE, e pode de acordo com a conveniência e necessidade do projeto pode ser implementado por exemplo configuração via arquivo *properties*, arquivo *txt*, enfim. Entenda como *parser* a classe que será responsável por ler os dados do arquivo de configuração seja Xml, txt, properties, etc.. e configurar o contexto do LINP baseado nas propriedades definidas.

A interface que define um *parser* de configuração do componente de persistência LINP é a *LinpConfiguration*. Esta interface provê as constantes das propriedades de configuração que podem ser definidas para o LINP por padrão. Outras propriedades podem ser criadas de acordo com o *parser* usado.

O framework já provê implementações padrão do *parser* para configurar o LINP. Abaixo são descritos estes *parsers* que são encontrados no pacote `org.lindbergframework.persistence.configuration`:

- **SimpleLinpConfiguration:** Deve ser usado para definição da configuração do LINP programaticamente.
- **XmlLinpConfiguration:** Parser padrão para definição da configuração baseado em XML de acordo com o schema: <http://www.lindbergframework.org/schema/lindberg-config.xsd> no que diz respeito ao componente de persistência.

A ideia da configuração é configurar o CORE e o core de encarrega de configurar os outros módulos. Desta forma a configuração do componente de persistência é feita no mesmo XML apresentado anteriormente para configurar o CORE não havendo a necessidade de mais de um arquivo XML para configurar o framework. Uma vez configurado o CORE e dentro dele o LINP é só inicializar a configuração do CORE que este se encarregará das configurações necessárias do componente de persistência, isso se o componente de persistência estiver sendo definido nas configurações do CORE. Se nenhuma configuração for definida para o LINP então apenas o módulo CORE será inicializado.

O schema apresentado anteriormente para configuração do CORE, *lindberg-config.xsd*, provê uma tag denominada *linp*. Esta tag é a tag raiz para toda a configuração do componente LINP via XML usando o parser default *XmlLinpConfiguration*. A tag *linp* engloba as definições de *datasource*, do *transaction manager*, *schema padrão*, da implementação a ser usada para resolver comandos sql, o tipo java a ser usado para tipos de dados sql do tipo *cursor*, entre outros. A maioria dessas configurações já tem valores padrão definidos o que não obriga você a defini-los fazendo com que quando estes não sejam definidos explicitamente o framework use a configuração padrão. Isso faz com que você tenha que definir apenas o que é necessário, específico e foge do padrão já definido pelo framework.

As propriedades que o LINP provê para configuração são apresentadas abaixo e são definidas em *LinpConfiguration*:

- [lindberg.persistence.IntegerCursorType](#):
  - Tipo java inteiro para cursores;
  - Valor padrão é `java.sql.Types.OTHER`.
- [lindberg.persistence.TransactionManager](#):
  - Implementação de *TransactionManager* que será usada para o gerenciamento das transações de banco;



- Valor padrão é *org.lindbergframework.persistence.transaction.LinpTransactionManager*.

- *lindberg.persistence.SqlCommandResolver*:
  - Implementação de *SqlCommandResolver* que fará o intercambio entre a aplicação e o repositório de comandos sql e atenderá a solicitações de obtenção desses comandos;
- *lindberg.persistence.DefaultSchema*:
  - Schema de banco padrão que deve ser usado. Essa propriedade é opcional e não possui o valor padrão.
- *lindberg.persistence.Template*:
  - Implementação de *PersistenceTemplate* que deve ser usado no contexto como padrão para operações de persistência.
  - Usando configuração via XML através de *XmlLinpConfiguration* o valor padrão é *org.lindbergframework.persistence.impl.LinpTemplate*.

**IMPORTANTE:** Além dessas propriedades é importante e necessário definir a parte as configurações de *datasource* que o LINP usará. Isso será abordado mais a frente.

### 5.3.1 – Configurando o LINP na prática - Programaticamente

Supondo que no exemplo usado para apresentar o core agora fosse necessário usar o componente LINP para persistência. Inicialmente vamos usar a configuração via programaticamente usando *SimpleLinpConfiguration*. Essa classe fornece a definição de propriedades diretamente via métodos setters de forma simples. No exemplo a seguir considere que para o CORE estamos usando o mesmo XML apresentado anteriormente para ilustrar a configuração deste:

```
XmlCoreConfiguration coreConfiguration =
    new ClassPathXmlCoreConfiguration("org/lindbergframework/configuracao/lindberg-config.xml");

SimpleLinpConfiguration linpConfiguration = new SimpleLinpConfiguration();

DataSource ds = createDataSource();
DataSourceConfig config = new DataSourceConfig(ds, com.mysql.jdbc.Driver.class);

linpConfiguration.setDataSourceConfig(config);
linpConfiguration.setCursorType(LinpConfiguration.DEFAULT_INTEGER_CURSOR_TYPE);
linpConfiguration.setTransactionManager(new LinpTransactionManager(ds));

XmlSqlCommandResolver sqlCommandResolver =
    new ClassPathXmlSqlCommandResolver("org/lindbergframework/repositorioSql/queries.xml",
                                         "org/lindbergframework/repositorioSql/updates.xml");

linpConfiguration.setSqlCommandResolver(sqlCommandResolver);

coreConfiguration.setLinpConfiguration(linpConfiguration);
coreConfiguration.initializeContext();
```

No trecho de código acima é feita a configuração do LINP programaticamente via *SimpleLinpConfiguration*. Inicialmente criamos uma configuração de CORE que para este caso utilizamos o mesmo XML do exemplo criado para apresentar as configurações do CORE. Como o XML do CORE em nosso exemplo está dentro do classpath de nossa aplicação usamos *ClassPathXmlCoreConfiguration* para configurações do CORE baseado em um XML que está em nosso classpath. Em seguida criamos nosso *datasource* e o encapsulamos na classe que representa as configurações de *datasource* dentro do LINP que



é a *DataSourceConfig*. Para tal passamos a referência do nosso *datasource* criado (ds) e o Driver que o nosso *datasource* está usando para criar um novo *DataSourceConfig*. Isso é mostrado abaixo:

```
DataSource ds = createDataSource();  
DataSourceConfig config = new DataSourceConfig(ds, com.mysql.jdbc.Driver.class);
```

A partir daí definimos as propriedades de acordo com nossa necessidade. Neste caso definimos:

- **DataSourceConfig:**

```
linpConfiguration.setDataSourceConfig(config);
```

- **CursorType:**

```
linpConfiguration.setCursorType(LinpConfiguration.DEFAULT_INTEGER_CURSOR_TYPE);
```

- **TransactionManager:**

```
linpConfiguration.setTransactionManager(new LinpTransactionManager(ds));
```

- **SqlCommandResolver:**

```
XmlSqlCommandResolver sqlCommandResolver =  
    new ClassPathXmlSqlCommandResolver("org/lindbergframework/repositorioSql/queries.xml",  
                                         "org/lindbergframework/repositorioSql/updates.xml");  
linpConfiguration.setSqlCommandResolver(sqlCommandResolver);
```

Aqui usamos uma implementação de *SqlCommandResolver* que trabalha como um repositório de comandos sql em arquivos XML e que será abordada com mais detalhes mais a frente. Esta é a principal implementação de *SqlCommandResolver* que é provida pelo componente. Neste caso usamos uma especialização desse tipo de repositório de sql que trabalha com XML's de comandos sql que estão diretamente no classpath do nosso projeto:

```
org.lindbergframework.persistence.sql.ClassPathXmlSqlCommandResolver
```

Como foi descrito anteriormente, os módulos do framework são configurados de acordo com a necessidade e a partir do módulo principal, o CORE. Após a definição das propriedades agora podemos encapsular a configuração do LINP dentro da configuração do CORE para a partir daí este último, quando inicializado, inicialize e configure o componente LINP:

```
coreConfiguration.setLinpConfiguration(linpConfiguration);
```

O último passo agora é inicializar o CORE. Como encapsulamos a configuração do LINP na configuração do CORE no passo anterior, então o CORE quando inicializado através de seu contexto, *CoreContext*, também inicializará automaticamente o LINP a partir de seu contexto, *LinpContext*. O *CoreContext* já faz isso pra você, você não precisa se preocupar com isso e tudo o que internamente é necessário. Essa ideia seguirá válida para outras configurações e componentes futuros que venham a serem acoplados ao framework. Todos serão inicializados e configurados a partir do CORE:

```
CoreContext.getInstance().initialize(coreConfiguration); OU coreConfiguration.initializeContext();
```

O log no console apresentará, dentre outras coisas, o seguinte trecho após a execução do *initialize* indicando o status da inicialização de cada módulo:

```
INFO: Initializing Lindberg Core Context
INFO: Initializing Lindberg Persistence Context
INFO: Lindberg Persistence Context initialized
INFO: Lindberg Core Context initialized
```

O log aparece exatamente nessa ordem, pois primeiro é disparado o processo de inicialização do CORE, durante este processo o *CoreContext* verifica que precisa inicializar LINP então dispara o processo de inicialização e configuração de *LinpContext* (contexto do LINP). Observe que somente após a inicialização com sucesso do LINP é que a inicialização do contexto do CORE é concluída. Isso ocorre porque a inicialização do CORE é responsável pela inicialização do LINP.

### 5.3.2 – Configurando o LINP na prática – Usando XML

O LINP assim como o CORE pode ser configurado via XML também, que é a forma que deve ser mais comumente utilizada. A configuração do LINP via XML se dá a partir do mesmo schema utilizado para configurar o CORE, ou seja a configuração do componente é feita no mesmo XML onde a configuração do CORE foi definida, só que as configurações deste último estão definidas dentro da tag *<core>* e as do LINP dentro da tag, de mesmo nível hierárquico no XML, *<linp>*. As mesmas configurações que foram usadas no exemplo de configuração programaticamente no exemplo anterior pode ser feita, e com mais flexibilidade ainda, via XML.

Como o XML de configuração do componente *LINP* é o mesmo do de configuração do *CORE* do framework então o schema é o mesmo.

Schema: <http://www.lindbergframework.org/schema/lindberg-config.xsd>

A diferença para as configurações do componente de persistência é que novas tags e atributos são usados para configurar o *LINP*. Estas novas tags estão descritas no schema acima. As propriedades de configuração, seguindo a mesma ideia do *CORE*, são detalhas abaixo. Seu uso é demonstrado na prática mais a frente:

#### - *lindberg.persistence.IntegerCursorType*

(se não for definida o tipo inteiro para cursores de banco padrão será usado – *java.sql.Types.OTHER*) :

Tipo inteiro para cursores de banco, quando o banco em uso provê o recurso de cursores.

Geralmente, quando definida explicitamente é usada alguma constante específica do banco de dados adotado. Por exemplo, quando o estamos usando o banco Oracle e queremos trabalhar com cursores de entrada ou saída em stored procedures e functions então deve-se definir essa propriedade de configuração como abaixo, usando *oracle.jdbc.OracleTypes.CURSOR*, pois o banco oracle fornece junto com seu driver uma constante do tipo inteiro (padrão) que identifica tipos de cursores de banco:

```
<config-property name="lindberg.persistence.IntegerCursorType"
                 value="#oracle.jdbc.OracleTypes.CURSOR" />
```

**NOTA:** Observe que foi usado no valor dessa propriedade o carácter coringa '#' para definir que o valor será obtido através da constante estática e pública *CURSOR* na classe *oracle.jdbc.OracleTypes*. Esse recurso de carácter coringa foi abordado com detalhes na sessão 3.3.2.1 deste documento.

- **lindberg.persistence.TransactionManager**

(se não for definida o *TransactionManager* padrão será usado – *org.lindbergframework.persistence.transaction.LinpTransactionManager*):

Implementação de *TransactionManager* que deve ser usada para o gerenciamento de transações.

**O gerenciamento de transações do LINP é detalhado mais a frente.**

- **lindberg.persistence.SqlCommandResolver** (requerido)

Implementação de *SqlCommandResolver* que será usada para resolver comandos sql requisitados pela aplicação fazendo uma ponte entre os DAO's (*Data Access Objects*) e o repositório comandos sql.

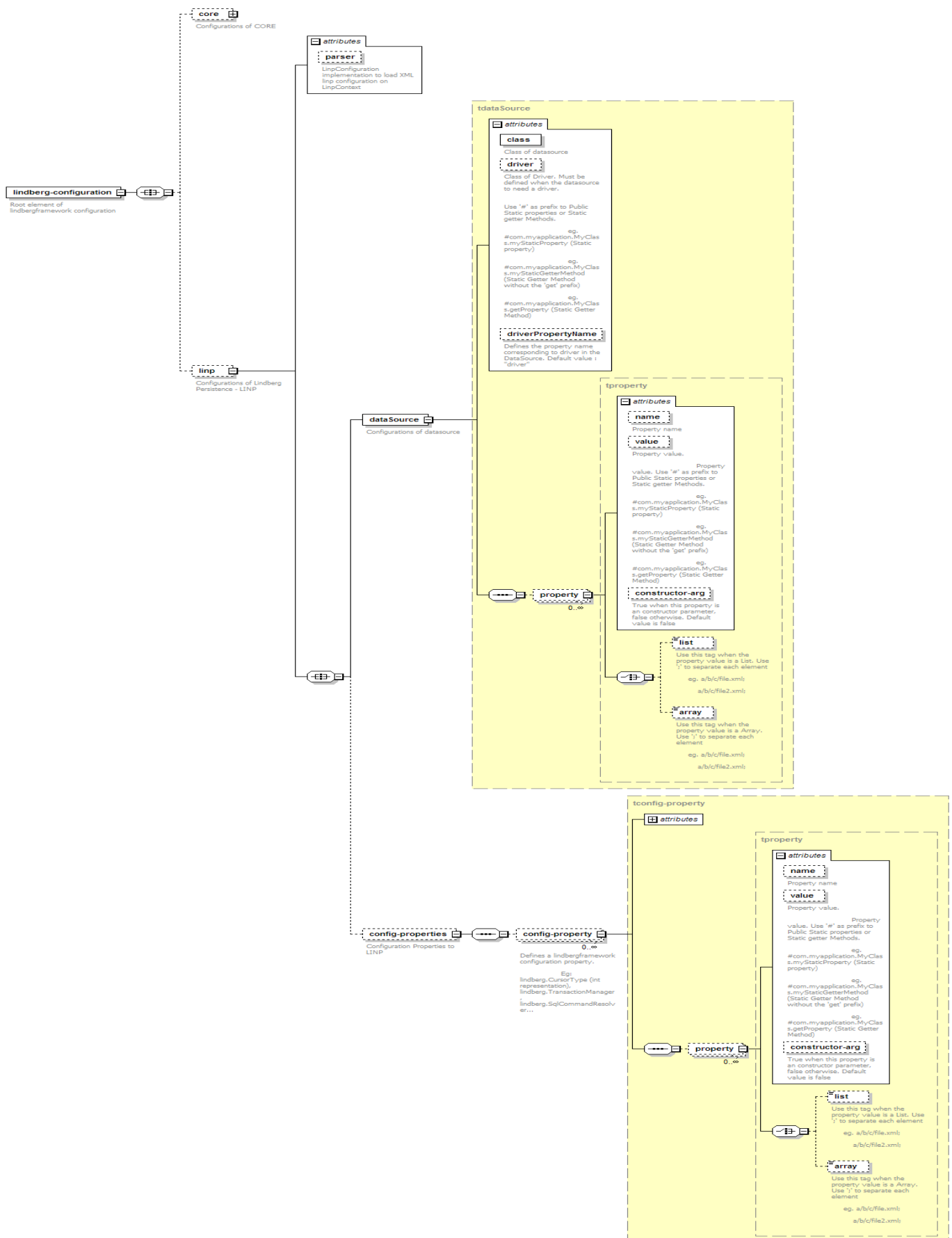
- **lindberg.persistence.DefaultSchema** (opcional)

Schema padrão que deve ser usado.

- **lindberg.persistence.Template** (opcional)

*PersistenceTemplate* que deve ser usado. Quando não definido, a implementação padrão, *LinpTemplate*, é usada.

A seguir é ilustrado de forma detalhada o schema [lindberg-config.xsd](#) no que diz respeito a tag `<linp>` para definição das configurações do LINP. Observe que a tag `<core>` está no mesmo nível e que o schema é o mesmo, fazendo com que tudo referente a configuração, seja do LINP ou CORE fique em um único XML.



**Importante:** qualquer tentativa de uso do componente LINP sem que este seja configurado e seu contexto, *LinpContext*, devidamente inicializado o framework lançará uma *IllegalStateException* com a mensagem descrita abaixo:

```
Linp Context is not active. Call initialize method in LinpContext to initializeContext method in LinpConfiguration implementation to active it
```

Abaixo é descrito o mesmo XML de configuração apresentado quando abordamos as configurações de CORE só que agora adicionando as configurações do LINP usando um datasource configurado para acessar um banco *Oracle*, isso fará com que o CORE configure e inicialize devidamente de forma automática o *LinpContext*.

```
<?xml version="1.0" encoding="UTF-8"?>
<linberg-configuration xmlns="http://www.lindbergframework.org/schema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.lindbergframework.org/schema lindberg-config.xsd">

  <core>
    <config-property name="linberg.core.di-basepackage"
      value="org.lindbergframework.exemplo.*"/>
    <config-property name="linberg.core.beanfactory"
      value="org.lindbergframework.beans.di.context.AnnotationBeanFactory"/>
  </core>

  <linp>
    <dataSource class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      driver="oracle.jdbc.OracleDriver" driverPropertyName="driverClassName">
      <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"/>
      <property name="username" value="SYSTEM" />
      <property name="password" value="admin" />
    </dataSource>

    <config-properties>
      <config-property name="linberg.persistence.IntegerCursorType" value="#oracle.jdbc.OracleTypes.CURSOR" />
      <config-property name="linberg.persistence.TransactionManager"
        value="org.lindbergframework.persistence.transaction.LinpTransactionManager"/>
      <config-property name="linberg.persistence.SqlCommandResolver"
        value="org.lindbergframework.persistence.sql.ClassPathXmlSqlCommandResolver">
        <property constructor-arg="true">
          <array>
            org/lindbergframework/repositorioSql/queries.xml;
            org/lindbergframework/repositorioSql/updates.xml
          </array>
        </property>
      </config-property>
    </config-properties>
  </linp>
</linberg-configuration>
```

O xml acima contém no mesmo arquivo tanto as configurações de CORE quanto, agora, as do componente *LINP*. As configurações do *CORE* não serão abordadas neste momento, visto que isto já foi feito na sessão dedicada ao *CORE* no início deste documento. No que diz respeito ao *LINP* as configurações são definidas a partir da tag *<linp>* e estas defines as propriedades do datasource, para conexão com o banco de dados, o schema padrão a ser usado, o gerenciador de transações (*TransactionManager*), a classe que trabalhará como camada intermediária entre a aplicação e o repositório de comandos sql (*SqlCommandResolver*) atendendo as requisições por comandos sql.

As configurações do LINP definidas no exemplo acima são detalhadas abaixo:

\* **DataSource:** Tag *<dataSource>* é específica, única e define as configurações de conexão com o banco de dados. No nosso exemplo usamos uma conexão com um banco local Oracle. Esta tag será abordada em detalhes mais a frente;

\* ***lindberg.persistence.IntegerCursorType***: Tipo específico inteiro Sql para Cursores de banco. Para este o banco usado no exemplo, Oracle, temos um tipo sql específico para cursores de banco: ***oracle.jdbc.OracleTypes.CURSOR***. Por ser uma constante, então esse valor é definido precedido pelo caracter '#';

\* ***lindberg.persistence.TransactionManager***: Gerenciador de transações que será usado para transações de banco. Para este exemplo é usado a implementação padrão de *TransactionManager*, *LinpTransactionManager*. Esta definição não é necessária pois quando esta propriedade não é definida o framework já usa a implementação padrão mas isto foi explicitado neste exemplo apenas para ilustrar. Use esta propriedade apenas quando for necessário usar um gerenciador de transações diferente e específico. Quando o fizer, certifique-se de estar usando um *TransactionManager* consistente;

\* ***lindberg.persistence.SqlCommandResolver***: Implementação de *SqlCommandResolver* que será responsável por receber e atender requisições de comandos, fazendo a ponte entre a aplicação e o repositório de comandos Sql. Neste exemplo usamos uma implementação que faz a interface para um repositório de comandos sql que estão definidos em arquivos XML que estão dentro do *classpath* da aplicação, *ClassPathXmlSqlCommandResolver*. Observe que esta propriedade de configuração define um parâmetro de construtor para o resolver que é um array de Strings. Esse parâmetro define o caminho onde estão os arquivos XML dentro do *classpath* contendo os comandos SQL que serão usados pela implementação de *SqlCommandResolver* especificada. Neste caso isto foi definido dentro da tag da *<config-property>* da propriedade de configuração do resolver no trecho a seguir:

```
<property constructor-arg="true">
  <array>
    org/lindbergframework/repositorioSql/queries.xml;
    org/lindbergframework/repositorioSql/updates.xml
  </array>
</property>
```

Observe que a *<property>* acima tem o atributo *constructor-arg* definido para 'true' indicando que é um parâmetro do construtor do resolver.

O Schema para esse tipo de repositório de comandos usando XML, *linp-sqlMapping.xsd*, o formato do XML e os detalhes da definição de comandos sql em um repositório dessa natureza a partir desse resolver serão abordados mais a frente.

### 5.3.2.1 – A tag *<dataSource>*

A tag *dataSource* é uma sub-tag dentro da tag *linp* que é requerida quando o componente *linp* é usado e é específica para a definição de configurações de data source. Isso é essencial pois toda a conexão com a base de dados é feita baseada nesses parâmetros são, dependendo da implementação de *DataSource* utilizada, url, driver, usuário, senha, etc..

Atributos da tag *dataSource*:

- \* ***class***: Classe que é a implementação da interface *javax.sql.DataSource* que deve ser usada;
- \* ***driver***: Drive a ser carregado do banco de dados utilizado;

\* ***driverPropertyName***: Nome da propriedade onde será setado o driver na implementação do *datasource* a ser usada. O valor padrão desse atributo é '***driver***'. Exemplo de uso desse atributo. Digamos que a implementação de *DataSource* a ser usada seja a *DriverManagerDataSource*, do spring. Para esta implementação o driver é setado através do método *setDriverClassName*, então esse atributo seria definido como '*driverClassName*'.

Essa tag também aceita o uso da tag *property* do mesmo modo que foi explicado e utilizado nos exemplos anteriores. Essa tag pode ser usada aqui de modo a definir propriedades a serem setadas na instancia da implementação de *dataSource* definida. Por exemplo, é comum quando se define um *DataSource*, configurar a url do banco, o driver, o usuário e a senha. Esses são alguns dos atributos que são comuns na definição e uso da maioria das implementações de *DataSoure*. Então o uso da tag *property* aqui seria necessário para efetuar essas configurações.

Abaixo a tag *DataSource* do exemplo anterior é detalhada:

```
<linp>
  <dataSource class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    driver="oracle.jdbc.OracleDriver" driverPropertyName="driverClassName">
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"/>
    <property name="username" value="SYSTEM" />
    <property name="password" value="admin" />
  </dataSource>
  ...
</linp>
```

O trecho de código acima define:

- \* Um *dataSource* onde a implementação a ser usada é a classe *DriverManagerDataSource* do spring;
- \* O driver para um banco oracle a ser usado, *oracle.jdbc.OracleDriver*;
- \* O nome da propriedade onde será setado o driver no *dataSource*, *driverClassName*;
- \* 3 propriedades que serão setada no *dataSource* quando este for criado:
  - 1 – url: *jdbc:oracle:thin:@localhost:1521:xe*
  - 2 – username: *SYSTEM*
  - 3 – password: *admin*

**Importane:** Um detalhe importante é que dependendo da implementação do *dataSource* a ser usada o *driver* é setado de uma forma diferente. Neste exemplo o driver é setado em forma de uma String que define o nome completamente qualificado da classe do driver. Em uma outra implementação o driver poderia ser setado diretamente como um parâmetro do tipo *class*. Um exemplo de *DataSource* que recebe o driver como um parâmetro do tipo *class* diretamente, é outra implementação do Spring *org.springframework.jdbc.datasource.SimpleDriverDataSource*. Para estes casos basta usar o caráter “#” da mesma forma que nos outros casos apresentados de uso desse caráter especial.

Caso usássemos essa outra implementação então a nova definição da tag ficaria como abaixo. Lembrando que essa implementação usa o nome padrão da propriedade do driver, '*driver*', então o atributo *driverPropertyName* não é necessário pois o valor padrão, como foi dito, já é '*driver*'. Então a tag ficaria da seguinte forma:

```
<linp>
  <dataSource class="org.springframework.jdbc.datasource.SimpleDriverDataSource"
```



```

        driver="#oracle.jdbc.OracleDriver">
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"/>
    <property name="username" value="SYSTEM" />
    <property name="password" value="admin" />
</dataSource>
...
</linp>

```

### 5.3.3 – A classe *LinpContext*

Da mesma forma que a classe *CoreContext*, a classe *LinpContext* também é um *singleton* e fornece as informações referentes ao contexto de persistência, se este estiver sendo usado e estiver devidamente configurado. Para obter a instância de *LinpContext*, o método *LinpContext.getInstance* deve ser chamado. *LinpContext* provê o resolvidor de comandos a partir do repositório configurado, o datasource definido, o gerenciador de transações, o template de persistência que está sendo usado como padrão, e toda a configuração dentro do contexto de persistência.

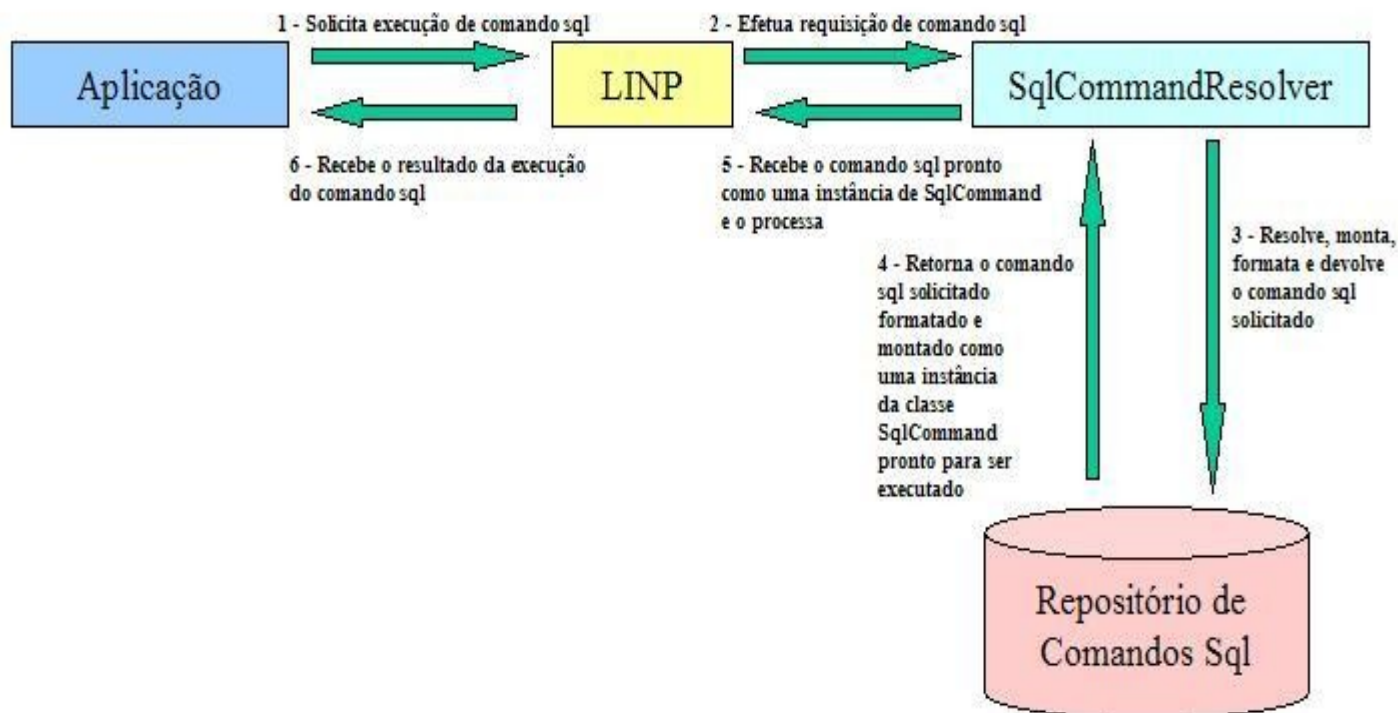
## 5.4 – Repositório de comandos SQL – Mais Detalhes

Como já foi falado no momento em que foi abordado a configuração do *linp*, o componente trabalha com o conceito de repositório de comandos sql. O repositório de comandos sql é o local onde são encontrados os comandos e as definições necessárias para uma chamada sql, seja ela um *select*, *update*, *delete*, uma chamada a uma *stored procedure*, uma *stored function*, etc... O componente se comunica com o repositório e obtém essas definições a partir da interface *SqlCommandResolver*, que faz o papel de intermediário entre o componente e o repositório. Como toda a comunicação entre o componente e o repositório é feita a partir dessa interface, então o repositório pode ser um conjunto de arquivos XML, um Web Service que atende a solicitações a comandos, um arquivo TXT, um outro software responsável por agrupar e atender solicitações a comandos sql, um arquivo XML baseado em um schema criado por você, enfim. Todo o *linp* funciona baseado em interfaces então a implementação a ser usada para o *SqlCommandResolver* quem define é você. Essa implementação é quem será responsável por intermediar toda a comunicação entre o componente e o repositório de comandos sql.

A implementação a ser usada de *SqlCommandResolver* é definida no arquivo de configuração do framework na parte de configuração do *linp*. Considerando que esteja usando a configuração padrão via XML do framework, então isto seria uma propriedade de configuração do componente – *lindberg.persistence.SqlCommandResolver*.

O *linp* já fornece uma implementação padrão de *SqlCommandResolver* que trabalha com um repositório de comandos sql em arquivos XML baseado em um schema padrão do lindbergframework para definição de comandos sql. Essa implementação padrão será detalhada mais a frente.

Abaixo é ilustrado como funciona essa intermediação:



## 5.5 – Usando o LINP na prática nos DAO's

O Linp fornece uma interface que define um *template* que é uma classe que deve fornecer todos os serviços de persistência que o componente provê, como: *selects*, *updates*, chamadas a *procedures* e *functions*, população de objetos retornados por consultas e parâmetros de saída de *procedures* e resultados de *functions*, etc... A interface que define um template é:

***org.lindbergframework.persistence.PersistenceTemplate***

Essa interface fornece métodos para execução de comandos sql diretamente ou acessando o repositório de comandos sql, fornece também a obtenção de uma instancia direta de *Connection*, do *DataSource* definido, do *SqlCommandResolver* definido, entre outros serviços.

A aplicação não precisa se preocupar na criação, obtenção e configuração direta da instancia do template. Isso fica a cargo do framework.

Os DAO's trabalharão fazendo uso de uma instância do *template*. Para tal o DAO deve estender a classe ***org.lindbergframework.persistence.dao.LinpDAO*** que já fornece o método ***getPersistTemplate()*** para a obtenção de uma instância da implementação padrão definida do *template* já configurada e pronta para ser usada. A implementação padrão que o framework provê é:

***org.lindbergframework.persistence.impl.LinpTemplate***

O *template* retornado pelo **LinpDAO** já vem pronto e conectado com o repositório de comandos sql definido de modo que você não precisa se preocupar com nada referente a isso. É só estender e usar. Lembrando que é necessária a correta configuração do componente Linp seja no XML, seja programaticamente, seja lá qual a implementação de *LinpConfiguration* você estiver usando.

## 5.6 – População de Beans Multi-Nível

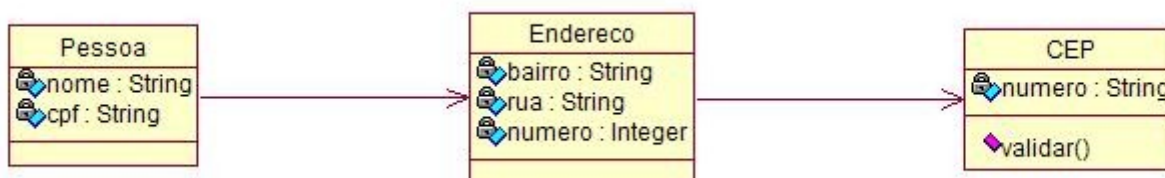
A forma tradicional de se popular uma lista de objetos a partir de uma query ou como resultado de parâmetros de saída de uma chamada a uma *sotred procedure* ou function é percorrendo um *ResultSet* obtendo os dados de cada coluna no formato linear e ir criando os objetos adjacentes e setando cada valor na propriedade correspondente no objeto correspondente. Então você obtém dados no formato linear e os organiza no formato orientado a objetos em diversos níveis.

Para ilustrar isso imagine a seguinte linha retornada de uma query:

Nome	CPF	Bairro	Rua	Numero	CEP
João Da Silva	11111111-11	Farol	Maria das Graças	102	98765-321

Observe que os dados estão todos em um mesmo nível e se nós tivéssemos que popula-os em objetos com certeza teríamos, caso não estejamos usando nenhum framework que nos auxilie nisso, que fazer todo o algoritmo para montar cada objeto, popular cada objeto com esses dados e setar cada objeto em seu lugar correto dentro da estrutura do modelo.

Imagine então que nós tivéssemos o simples modelo apresentado no diagrama de classes abaixo e que nós tivéssemos que popular esses dados em objetos do tipo Pessoa.



Observe que no diagrama acima os dados são organizados em um modelo orientado a objetos. Então nós temos um resultado de uma query com os dados retornados da consulta todos organizados de uma forma de uma tabela e precisamos popula-los em objetos onde cada dado deve ser setado em um objeto diferente dentro do contexto de pessoa.

O *linp* fornece uma interface que define um populador de beans que é:

***org.lindbergframework.persistence.beans.BeanPopulator***

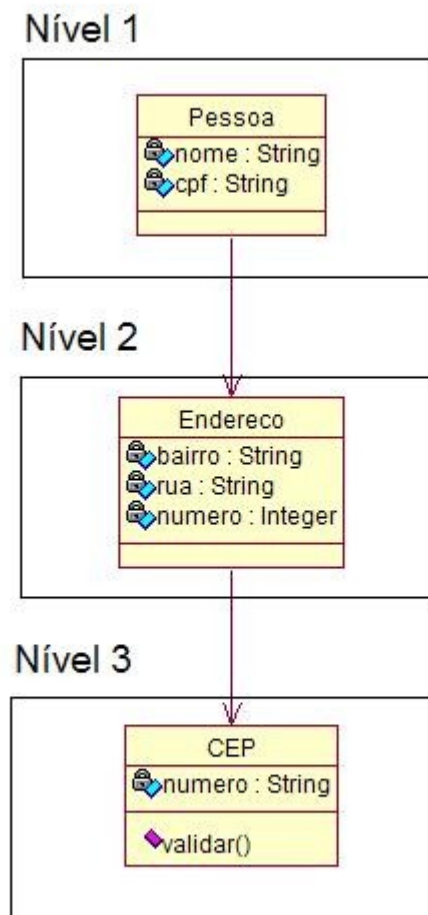
O *linp* já provê também uma implementação padrão dessa interface e que implementa o conceito de

população multi-nível:

***org.lindbergframework.persistence.beans.MultiLevelsBeanPopulator***

Com base nessas classes, imagine que façamos uma query e queremos popular o resultado da query em objetos do tipo *Pessoa*. Observe que o resultado é devolvido em forma linear e não orientado a objetos como em uma hierarquia de objetos – *Pessoa* tem um *Endereco* e o *Endereço* tem um *Cep*.

Para ilustrar isso imagine as mesmas classes agora da seguinte forma, organizadas em níveis.



Cada linha retornada na query será populada em objetos do tipo *Pessoa*, que representa o primeiro nível, ou o nível raiz. A partir do nível 1, *Pessoa*, será então criado um objeto *Endereco*, que dentro da hierarquia está no segundo nível, pois é um atributo direto de um objeto de nível 1, neste caso *Pessoa*. A partir do objeto *Endereco*, precisamos agora criar o objeto *CEP* que é um objeto de nível 3 e será populado como atributo de um objeto de nível 2, neste caso *Endereco*.

Isso é a população de multi-nível, pois é feita em diversos níveis. Como foi dito, O *linp* fornece como implementação padrão da interface ***BeanPopulator*** a classe ***MultiLevelsBeanPopulator*** e esta classe implementa a população em multi-níveis baseada em uma linha específica de um resultado de uma query.

## 5.7 – Usando o padrão de nomenclatura do JAVA para definir comandos SQL's

Agora você deve tá se perguntando como o *linp* sabe onde setar cada valor retornado na consulta nos objetos?

É simples, para definir o local da propriedade correspondente a coluna retornada pelo resultado o *linp* usa o padrão de nomenclatura de JAVA nos sql's. Vamos a prática. O select acima poderia ser:

```
Select nome, cpf, bairro endereco.bairro, rua endereco.rua,  
        numero endereco.numero, cep endereco.cep.numero  
From Pessoa
```

Você agora deve tá pensando... Mas o banco que eu uso não aceita essa sintaxe usando ponto “.”. Não importa, o *linp*, independente do banco faz uma tradução do sql nessa notação para o sql correto e faz a tradução correta de modo a servir como ponte entre o sql nessa notação e o verdadeiro e devido sql a ser executado no banco. Isso é transparente você não precisa se preocupar com isso.

Observe que as propriedades *nome* e *cpf* não usaram a notação. Isso não foi necessário por que isso diz ao *linp* que esse é um valor de uma propriedade localizada no próprio objeto raiz (nível 1), que neste caso é *Pessoa*. De fato nossa classe *Pessoa* tem uma propriedade *nome* e outra *cpf*. E estas serão populadas diretamente no nível 1.

Após a execução desse sql, para a coluna CEP por exemplo, o *linp* vai criar para cada linha do resultado retornado na consulta, um objeto *Pessoa*, um objeto *Endereco*, um objeto *CEP* e vai setar o valor da coluna *cep* na propriedade número do objeto *CEP*, vai setar este objeto *CEP* (nível 3) na propriedade *cep* do objeto *Endereco* (nível 2) e por último seta este objeto *Endereco* na propriedade *endereco* do objeto raiz *Pessoa* (nível 1).

Obs.: Essa notação funciona exatamente como o alias da coluna na consulta. Então se o banco que estiver usando necessite por exemplo usar a palavra reservada 'AS' para isso, então é necessário usa-la. Então neste caso no sql a propriedade bairro, por exemplo, ficaria da seguinte forma:

```
... bairro AS endereco.bairro ...
```

### 5.7 – Template de Persistência (PersistenceTemplate)

Os recursos de persistência do *linp* são providos por um template, que é uma classe que deve fornecer uma ponte entre os DAO's da aplicação e os serviços de persistência disponíveis pelo *linp*. A interface que define um template do *linp* é:

***org.lindbergframework.persistence.PersistenceTemplate***

Essa interface define os seguintes serviços (métodos) que devem ser providos por um template de persistência dentro do lindberg persistence.

**NOTA: O LINP já provê uma implementação padrão e que é a usada como padrão *org.lindbergframework.persistence.impl.LinpTemplate***

**\* Método: execQuery**

**Assinatura:** `public <E> List<E> execQuery(Class<E> clazz, String sqlId, Object... params)`

**Descrição:** Executa uma query a partir de um comando sql do repositório de comandos. Use o carácter '?' para definir parâmetros no comando definindo no repositório.

Exemplo: *Select \* From Person where id = ?*.

**Retorno:** Lista de objetos populados a partir do resultado da consulta.

**Parâmetros:**

- *sqlId*: ID do comando sql no repositório
- *clazz*: Classe (classe raiz, nível 1) que será usada para popular o resultado da consulta
- *params*: Array (varargs) de objetos que são os parâmetros da consulta

**\* Método: execSqlQuery**

**Assinatura:** `public <E> List<E> execSqlQuery(Class<E> clazz, String sql, Object... params);`

**Descrição:** Executa uma query sql. Use o carácter '?' para definir parâmetros no sql da query.

Exemplo: *Select \* From Person where id = ?*.

**Retorno:** Lista de objetos populados a partir do resultado da consulta.

**Parâmetros:**

- *clazz*: Classe (classe raiz, nível 1) que será usada para popular o resultado da consulta
- *sql*: Sql da consulta a ser realizada
- *params*: Array (varargs) de objetos que são os parâmetros da consulta

**\* Método: execQueryForObject**

**Assinatura:** `public <E> E execQueryForObject(Class<E> clazz, String sqlId, Object... params)  
throws NonUniqueRowException;`

**Descrição:** Executa uma query a partir de um comando sql do repositório de comandos onde é esperado no máximo um registro como resultado. Use o carácter '?' para definir parâmetros no comando definindo no repositório. Exemplo: *Select \* From Person where id = ?*.

**Retorno:** Objeto populado a partir do único registro retornado no resultado da consulta ou *null* caso a consulta não tenha retornado nenhum registro.



**Throws:**

- *NonUniqueRowException*: Lançada quando a query retornou mais de um registro.

**Parâmetros:**

- *clazz*: Classe (classe raiz, nível 1) que será usada para popular o resultado da consulta
- *sqlId*: ID do comando sql no repositório
- *params*: Array (varargs) de objetos que são os parâmetros da consulta

**\* Método: execSqlQueryForObject**

**Assinatura:** `public <E> E execSqlQueryForObject(Class<E> clazz, String sql, Object... params) throws NonUniqueRowException;`

**Descrição:** Executa uma query sql onde é esperado no máximo um registro como resultado. Use o carácter '?' para definir parâmetros no sql da query.

Exemplo: *Select \* From Person where id = ?*.

**Retorno:** Objeto populado a partir do único registro retornado no resultado da consulta ou *null* caso a consulta não tenha retornado nenhum registro.

**Throws:**

- *NonUniqueRowException*: Lançada quando a query retornou mais de um registro.

**Parâmetros:**

- *clazz*: Classe (classe raiz, nível 1) que será usada para popular o resultado da consulta
- *sqlId*: ID do comando sql no repositório
- *params*: Array (varargs) de objetos que são os parâmetros da consulta

**\* Método: execUpdate**

**Assinatura:** `public int execUpdate(String sqlId, Object... args);`

**Descrição:** Executa uma atualização a partir de um comando sql do repositório de comandos e retorna o número de registros afetados pela atualização. Usado para qualquer tipo de atualização : *delete*, *update* e *insert*. Use o carácter '?' para definir argumentos no comando definindo no repositório. Exemplo: *insert into person values(?,?,?)*.

**Retorno:** Número de registros afetados pela atualização.

**Parâmetros:**

- *sqlId*: ID do comando sql no repositório
- *args*: Array (varargs) de objetos que são os argumentos para a atualização a ser executada.

**\* Método: execSqlUpdate**

**Assinatura:** `public int execSqlUpdate(String sql, Object... args);`

**Descrição:** Executa uma atualização sql e retorna o número de registros afetados pela atualização. Usado para qualquer tipo de atualização : *delete*, *update* e *insert*. Use o carácter '?' para definir argumentos no sql



da atualização. Exemplo: *insert into person values(?,?,?)*.

**Retorno:** Número de registros afetados pela atualização.

**Parâmetros:**

- *sql*: sql da atualização a ser executada.
- *args*: Array (varargs) de objetos que são os argumentos para a atualização a ser executada.

**\* Método: callProcedure**

**Assinatura:** `public Map callProcedure(String sqlId, SqlArg... args);`

**Descrição:** Chama e executa um comando sql a partir do repositório para uma stored procedure.

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da procedure onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Parâmetros:**

- *sqlId*: ID do comando referente a chamada a stored procedure a partir do repositório de comandos.
- *args*: Array (varargs) de *org.lindbergframework.persistence.sql.SqlArg* que são os argumentos de entrada para a procedure a ser chamada. Essa classe *SqlArg* nada mais é do que uma classe que engloba um par de nome e valor de um argumento de um sql.

**\* Método: callProcedure** (Sobrecarga usando um MAP como entrada de argumentos para procedure)

**Assinatura:** `public Map callProcedure(String sqlId, Map<String, Object> args);`

**Descrição:** Chama e executa um comando sql a partir do repositório para uma stored procedure.

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da procedure onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Parâmetros:**

- *sqlId*: ID do comando referente a chamada a stored procedure a partir do repositório de comandos.
- *args*: Map composto pelos argumentos de entrada da procedure onde em cada elemento a chave é o nome do argumento na assinatura da procedure e o valor é o próprio valor a ser usado.

**\* Método: callProcedure** (Sobrecarga não usando o repositório de comandos, usando a classe *SqlProcedure*)

**Assinatura:** `public Map callProcedure(SqlProcedure procedure, SqlArg... args);`

**Descrição:** Chama e executa uma procedure a partir da classe  
*org.lindbergframework.persistence.sql.SqlProcedure*

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da procedure onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Parâmetros:**

- *procedure*: Instância de *SqlProcedure* composta pela definição da procedure a ser chamada.

- *args*: Array (varargs) de *org.lindbergframework.persistence.sql.SqlArg* que são os argumentos de entrada para a procedure a ser chamada. Essa classe *SqlArg* nada mais é do que uma classe que engloba um par de nome e valor de um argumento de um sql.

\* **Método: callProcedure** (Sobrecarga não usando o repositório de comandos, usando a classe *SqlProcedure* e um *Map* como parâmetros de entrada para a Procedure)

**Assinatura:** `public Map callProcedure(SqlProcedure procedure, Map<String, Object> args);`

**Descrição:** Chama e executa uma procedure a partir da classe *org.lindbergframework.persistence.sql.SqlProcedure*

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da procedure onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Parâmetros:**

- *procedure*: Instância de *SqlProcedure* composta pela definição da procedure a ser chamada.
- *args*: Map composto pelos argumentos de entrada da procedure onde em cada elemento a chave é o nome do argumento na assinatura da procedure e o valor é o próprio valor a ser usado.

\* **Método: callFunction**

**Assinatura:** `public Map callFunction(String sqlId, SqlArg... args);`

**Descrição:** Chama e executa um comando sql a partir do repositório para uma stored function.

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da function onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**NOTA:** O valor padrão para a chave do valor retornado como resultado da function no map é definido pela constante *SqlFunction.DEFAULT\_RESULT\_NAME*. Caso o nome desse parâmetro seja definido na declaração do comando no repositório o padrão é sobrescrito pelo definido explicitamente.

**Parâmetros:**

- *sqlId*: ID do comando referente a chamada a stored function a partir do repositório de comandos.
- *args*: Array (varargs) de *org.lindbergframework.persistence.sql.SqlArg* que são os argumentos de entrada para a function a ser chamada. Essa classe *SqlArg* nada mais é do que uma classe que engloba um par de nome e valor de um argumento de um sql.

\* **Método: callFunction** (Sobrecarga usando um MAP como entrada de argumentos para function)

**Assinatura:** `public Map callFunction(String sqlId, Map<String, Object> args);`

**Descrição:** Chama e executa um comando sql a partir do repositório para uma stored function.

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da function onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Observação:** O valor padrão para a chave do valor retornado como resultado da function no map é definido

pela constante *SqlFunction.DEFAULT\_RESULT\_NAME*. Caso o nome desse parâmetro seja definido na declaração do comando no repositório o padrão é sobrescrito pelo definido explicitamente.

#### Parâmetros:

- *sqlId*: ID do comando referente a chamada a stored function a partir do repositório de comandos.
- *args*: Map composto pelos argumentos de entrada da function onde em cada elemento a chave é o nome do argumento na assinatura da function e o valor é o próprio valor a ser usado.

\* **Método:** *callFunction* (Sobrecarga não usando o repositório de comandos, usando a classe *SqlFunction*)

**Assinatura:** *public Map callFunction(SqlFunction function, SqlArg... args);*

**Descrição:** Chama e executa uma function a partir da classe  
*org.lindbergframework.persistence.sql.SqlFunction*

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da function onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Observação:** O valor padrão para a chave do valor retornado como resultado da function no map é definido pela constante *SqlFunction.DEFAULT\_RESULT\_NAME*. Caso o nome desse parâmetro seja definido na declaração do comando no repositório o padrão é sobrescrito pelo definido explicitamente.

#### Parâmetros:

- *function*: Instância de *SqlFunction* composta pela definição da function a ser chamada.
- *args*: Array (varargs) de *org.lindbergframework.persistence.sql.SqlArg* que são os argumentos de entrada para a function a ser chamada. Essa classe *SqlArg* nada mais é do que uma classe que engloba um par de nome e valor de um argumento de um sql.

\* **Método:** *callFunction* Sobrecarga não usando o repositório de comandos, usando a classe *SqlProcedure* e um Map como parâmetros de entrada para a Procedure)

**Assinatura:** *public Map callFunction(SqlFunction function, Map<String, Object> args);*

**Descrição:** Chama e executa uma function a partir da classe  
*org.lindbergframework.persistence.sql.SqlFunction*

**Retorno:** Map contendo os parâmetros de saída retornados após a execução da function onde a chave de cada elemento é o nome do parâmetro retornado e o valor é o próprio valor retornado para o parâmetro.

**Observação:** O valor padrão para a chave do valor retornado como resultado da function no map é definido pela constante *SqlFunction.DEFAULT\_RESULT\_NAME*. Caso o nome desse parâmetro seja definido na declaração do comando no repositório o padrão é sobrescrito pelo definido explicitamente.

#### Parâmetros:

- *function*: Instância de *SqlFunction* composta pela definição da function a ser chamada.
- *args*: Map composto pelos argumentos de entrada da function onde em cada elemento a chave é o nome do argumento na assinatura da function e o valor é o próprio valor a ser usado.

**\* Método: configureDefaultPopulator**

**Assinatura:** `public void configureDefaultPopulator();`

**Descrição:** Configura a implementação padrão de *BeanPopulator* como responsável por toda a população de beans originados de queries, cursores como parametros de saída de functions e procedures e onde mais for necessário popular objetos a partir de dados obtidos do banco .

*org.lindbergframework.persistence.beans.MultiLevelsBeanPopulator*

**\* Método: configureDefaultDataSource**

**Assinatura:** `public void configureDefaultDataSource();`

**Descrição:** Configura o datasource padrão definido na configuração do *LINP* a partir do contexto (*LinpContext*).

**\* Método: setDataSourceConfig**

**Assinatura:** `public void setDataSourceConfig(DataSourceConfig dataSourceConfig);`

**Descrição:** Configura explicitamente as configurações de DataSource que devem ser usadas pelo template.

**\* Método: getDataSourceConfig**

**Assinatura:** `public DataSourceConfig getDataSourceConfig();`

**Descrição:** Retorna as configurações de datasource definidas que estão sendo usadas pelo template.

**\* Método: getDataSource**

**Assinatura:** `public DataSource getDataSource();`

**Descrição:** Retorna o datasource definido que está sendo usado pelo template.

**\* Método: setBeanPopulator**

**Assinatura:** `public void setBeanPopulator(BeenPopulator beanPopulator);`

**Descrição:** Configura explicitamente qual implementação de *BeanPopulator* deve ser usada pelo template.

**\* Método: getConnection**

**Assinatura:** `public Connection getConnection() throws SQLException;`

**Descrição:** Retorna uma conexão obtida a partir do datasource definido para o template.

**Throws:**

- *SQLException*: Se ocorreu algum problema na obtenção da conexão.

**\* Método: getSqlCommandResolver**

**Assinatura:** `public SqlCommandResolver getSqlCommandResolver();`

**Descrição:** Retorna a implementação de *SqlCommandResolver* que está sendo usada pelo template através do contexto (*LinpContext*).

## 5.8 – Definindo e trabalhando com repositório de Comandos SQL usando XML

A implementação padrão de repositório de comandos SQL provida pelo lindbergframework é a que trabalha com um repositório definido em um ou mais arquivos XML onde o *SqlCommandResolver* vai buscar os comandos SQL solicitados.

Essa implementação padrão que trabalha com XML é a classe:

*org.lindbergframework.persistence.sql.XmlSqlCommandResolver*

Essa classe implementa toda a lógica de trabalhar com repositório de comandos SQL em arquivos XML baseados em um schema definido e que será abordado mais a frente.

Esta classe possui uma extensão:

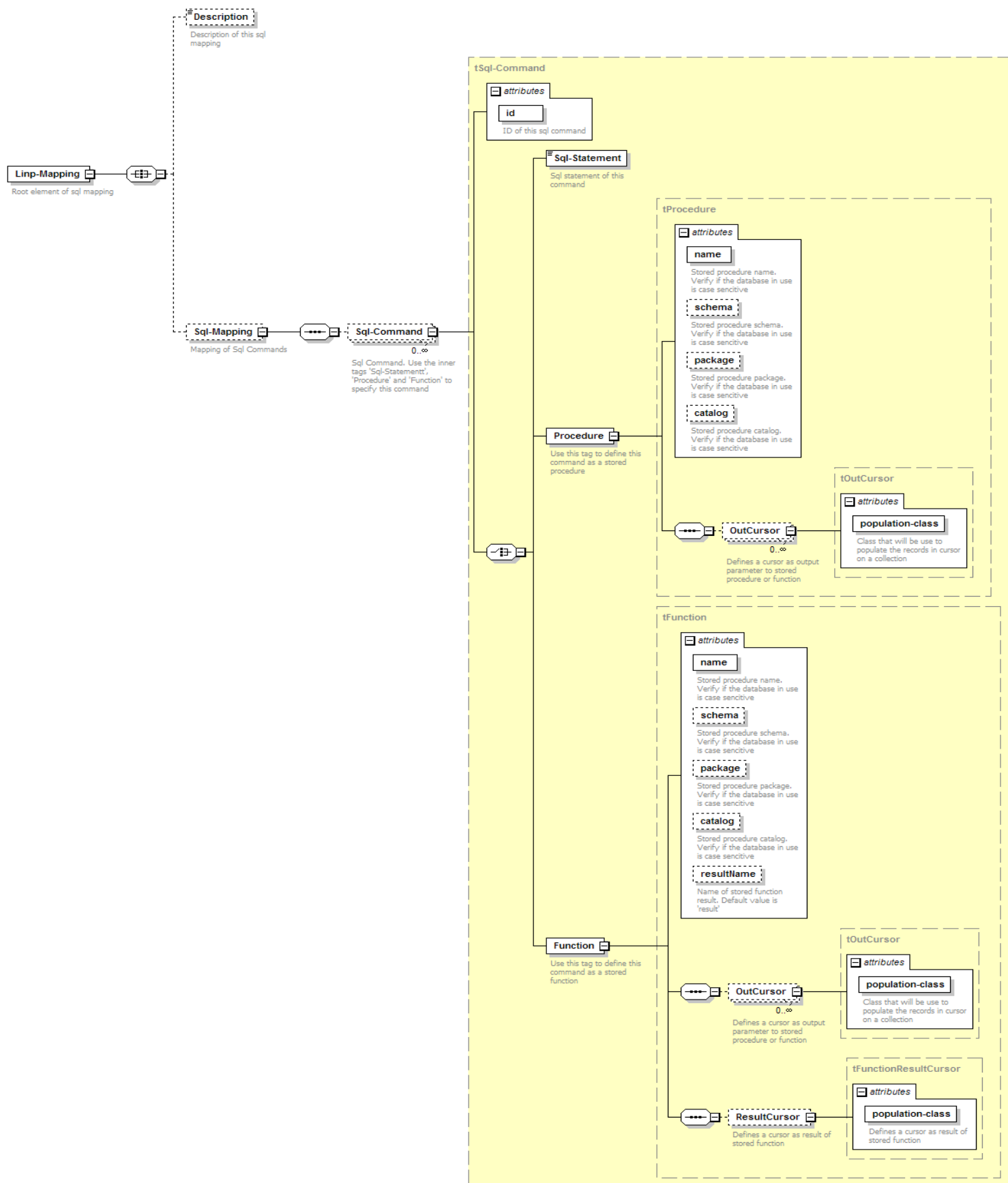
*org.lindbergframework.persistence.sql.ClassPathXmlSqlCommandResolver*

Esta extensão será a normalmente utilizada e é especializada em trabalhar com arquivos XML definidos dentro do classpath do projeto.

O Schema usado para definir os arquivos XML que farão parte do repositório de comandos é o:

<http://www.lindbergframework.org/schema/linp-sqlMapping.xsd>

O schema bem como sua estrutura hierárquica são apresentados na figura a seguir:



A ideia de um XML baseado nesse schema é definir comandos SQL, sejam *selects*, *updates*, *deletes*, chamadas a *procedures* e *functions*. O Schema define simplesmente que cada comando deve possuir um *ID*. Esse *ID* deve ser único em todo o contexto do *LINP* mesmo que o repositório seja composto por vários arquivos XML distintos mas que fazem parte do mesmo contexto. O *ID* não deve se repetir entre estes arquivos. Caso isto ocorra uma *org.lindbergframework.exception.DuplicatedSqlMappingIdException* é lançada.

O Schema define a tag raiz que é a `<Linp-Mapping>`. Abaixo dessa tag na hierarquia contém duas tags `<Description>` e `<Sql-Mapping>` onde a primeira nada mais é que uma descrição livre sobre o XML em questão e a segunda é onde ficaram definidos os comandos SQL.

Dentro da tag `<Sql-Mapping>` os comandos sql são definidos usando a tag `<Sql-Command>`. Cada tag `<Sql-Command>` define um comando sql individual. Essa tag possui um atributo que é "id". Esse atributo é o identificador do comando e é requerido. É através desse *ID* que esse comando será localizado dentro do repositório e partir deste também que o template de persistência solicitará a execução do comando.

Para definir se esse comando é um SQL direto ou uma chamada a uma stored procedure ou function esta tag assume 3 outras tags abaixo dela na hierarquia que definem o estereotipo do comando e que somente uma pode ser usada em cada comando. Elas são apresentadas abaixo:

**<Sql-Statement>**: Define um comando SQL simples como *select*, *update* e *delete*. Não possui nenhum atributo e seu conteúdo define a string do comando sql diretamente.

#### EXEMPLO:

```
<Sql-Command id="consultarPessoasPorBairro">
  <Sql-Statement>select nome, telefone, rua endereco.rua,
                    bairro endereco.bairro, numero endereco.numero,
                    cep endereco.cep from Pessoa where bairro = ?
  </Sql-Statement>
</Sql-Command>

<Sql-Command id="excluirPessoas">
  <Sql-Statement>delete from Pessoa where id = ?</Sql-Statement>
</Sql-Command>
```

**NOTA:** A declaração do alias da coluna usando o carácter '.' (ponto) é seguindo o conceito de população multinível, apresentado na sessão 5.6 – *População de Beans Multi-Nível*. O uso direto desse sql causaria erro de compilação mas como o *LINP* propõe o conceito de população multi-nível então internamente ele faz a devida formatação e tradução desse comando para um *SQL* válido.

**<Procedure>**: Define uma chamada a uma stored procedure. Nesta tag são definidos também os parâmetros de saída que são *cursores*, caso exista algum, o catálogo, o pacote e o schema onde está a procedure, caso seja necessário.

**NOTA:** A interface *PersistenceTemplate* que define um template de persistência provê na assinatura dos métodos de chamada a stored procedures e stored functions o retorno como sendo um Map. Esse map possui é retornado contendo os parâmetros de saída da execução de procedure ou functions, de modo que o próprio nome do parâmetro obtido através da assinatura da procedure ou function é usado como nome chave no map e o seu valor como o valor do map.



## EXEMPLO:

```
<Sql-Command id="minhaProcedure1">
  <Procedure name="procedure1" />
</Sql-Command>

<Sql-Command id="minhaProcedureDefinindoSchema" >
  <Procedure name="procedure1" catalog="meuCatalogo"
    package="meuPacote" schema="meuSchema"/>
</Sql-Command>

<Sql-Command id="listarPessoasUsandoProcedure">
  <Procedure name="listarPessoas" >
    <OutCursor population-class="br.com.meupacote.Pessoa"/>
  </Procedure>
</Sql-Command>

<Sql-Command id="minhaProcedureCom3CursosDeSaida">
  <Procedure name="procedureCom3CursosDeSaida" >
    <OutCursor population-class="br.com.meupacote.Classe1"/>
    <OutCursor population-class="br.com.meupacote.Classe2"/>
    <OutCursor population-class="br.com.meupacote.Classe2"/>
  </Procedure>
</Sql-Command>
```

A tag **<OutCursor>** define um parâmetro de saída na *Procedure* e que é um *cursor*. O atributo *population-class* define qual classe deve ser usada para a criação dos objetos onde serão populados cada registro retornado pelo cursor. Se a procedure possuir mais de um *cursor* de saída a ordem importa pois a ordem em que cada tag **<OutCursor>** foi declarada é a ordem esperada na assinatura da *Procedure* no banco de dados.

**NOTA:** Esses cursores devem fazer uso da nomenclatura padrão de beans para que a população multi-nível possa ser realizada. Só que como o sql da criação dos cursores é definido dentro da própria procedure e o uso direto de por exemplo *endereco.nome* como alias de uma coluna causaria erro de compilação e o *LINP* não tem como efetuar o devido tratamento e a devida tradução para um SQL válido então em stored procedures e functions os cursores de saída podem usar o carácter '\$' ao invés do carácter '.' (ponto), que é um carácter comum e aceito normalmente pelos SGDB's. Então se em um cursor de saída nós tivéssemos 3 colunas com alias como:

*endereco.nome*  
*endereco.cep.numero*  
*campo1.campo2.campo3.campo4.valor*

Esses campos também poderiam ser retornados com o alias sendo, no caso do banco que estiver usando não suportar definição de alias tipo “campo1.campo2.campo3” :

*endereco\$\$nome*  
*endereco\$\$cep.numero*  
*campo1\$\$campo2\$\$campo3\$\$campo4\$\$valor*

**<Function>**: Define uma chamada a uma stored function. Nesta tag são definidos também os parâmetros de saída que são *cursores*, caso exista algum. Esta tag funciona da mesma que a tag **<Procedure>**, não sendo necessário repetir toda a descrição aqui. A diferença é que por ser uma *function* essa tag define também um nome chave (que será usado no map de resultado. Ver o trecho de observação da tag *Procedure*) para o valor retornado como resultado da função e pode definir também uma classe de bean

que será usada para popular o resultado da function retornado como cursor em uma lista de beans. O valor retornado como resultado da função seja ele um cursor ou não tem por padrão o nome chave definido na constante `org.lindbergframework.persistence.sql.SqlFunction.DEFAULT_RESULT_NAME`. O atributo dessa tag `function` que define o nome chave do resultado é o `resultName` e quando este for definido este será usado ao invés da constante padrão. Quando não for definido a constante `DEFAULT_RESULT_NAME` será usada no map de resultados contendo tanto o valor resultado da execução da `function` quanto os parâmetros de saída da `function`, caso exista algum.

## EXEMPLO:

```
<Sql-Command id="minhaFunction1">
  <Function name="function1" />
</Sql-Command>

<Sql-Command id="minhaFunctionDefinindoResultName">
  <Function name="function1" resultName="meuResultado"/>
</Sql-Command>

<Sql-Command id="minhaFunctionDefinindoSchema" >
  <Function name="function1" catalog="meuCatalogo"
    package="meuPacote" schema="meuSchema"/>
</Sql-Command>

<Sql-Command id="listarPessoasUsandoFunctionRetornandoQtdToalCadastrada" >
  <Function name="listarPessoas" resultName="qtdPessoas">
    <OutCursor population-class="br.com.meupacote.Pessoa"/>
  </Function>
</Sql-Command>

<Sql-Command id="minhaFunctionRetornandoCursorComoReultado">
  <Function name="functionRetornandoCursor" >
    <ResultCursor population-class="br.com.meupacote.Pessoa"/>
  </Function>
</Sql-Command>

<Sql-Command id="minhaFunctionCom3CursosDeSaidaRetornandoCursorComoReultado">
  <Function name="functionCom3CursosDeSaidaRetornandoCursor"
    resultName="cursorPessoas">
    <ResultCursor population-class="br.com.meupacote.Pessoa"/>
    <OutCursor population-class="br.com.meupacote.Clasel"/>
    <OutCursor population-class="br.com.meupacote.Classe2"/>
    <OutCursor population-class="br.com.meupacote.Classe2"/>
  </Function>
</Sql-Command>
```

## 5.9 – Adicionando os recursos de persistência do LINP aos DAO's

Como demonstrado previamente, os recursos do LINP são providos por uma implementação de *PersistenceTemplate*. Para que os DAO's obtenham uma implementação do template já pronta e configurada e assim possam fazer uso dos recursos providos por ele, existem duas formas principais. A primeira é estender o DAO provido pelo LINP e que já fornece o template padrão pronto e configurado para usar e a segunda é obter diretamente uma instância do template a partir da fábrica de beans do contexto de persistência.

Os dois modos serão detalhados abaixo:

**1- LinpDAO:** Estender a classe `org.lindbergframework.persistence.dao.LinpDAO` é o modo mais comum de *turbinar* os DAO's com os recursos fornecidos pelo template do LINP. Essa classe provê

um método para obtenção da implementação de *PersistenceTemplate* configurada, funcional e pronta para ser usada baseada nas configurações feitas no contexto *LINP* (datasource, transaction manager, repositório de comandos, etc...).

O método provido em *LinpDAO* para obtenção do template pronto é:

```
public PersistenceTemplate getPersistTemplate()
```

Por padrão o *LinpDAO* retorna como template uma instância do template padrão, que já foi abordado anteriormente, *LinpTemplate*. Esse template já é provido pelo *LinpDAO* com o *BeanPopulator* padrão, *MultiLevelsBeanPopulator* e com o *datasource* definido nas configurações do contexto *LINP*.

*LinpDAO* também fornece a possibilidade de você definir explicitamente, via construtores sobrecarregados, uma outra implementação de *PersistenceTemplate*, *BeanPopulator* e/ou *datasource* devem ser usados para o DAO em questão.

Para tal, os construtores sobrecarregados que podem ser usados são:

```
public LinpDAO(PersistenceTemplate template)

public LinpDAO(BeansPopulator populador)

public LinpDAO(PersistenceTemplate template, BeansPopulator populador)

public LinpDAO(PersistenceTemplate template, BeansPopulator populador,
               DataSourceConfig dsConfig)

public LinpDAO(DataSourceConfig dsConfig)

public LinpDAO(BeansPopulator populador, DataSourceConfig dsConfig)
```

Qualquer um desses só será usado em algum caso excepcional pois apenas estender *LinpDAO* usando o construtor padrão sem argumentos já fornece um template pronto e configurado para uso. Mas caso precise estes construtores acima apresentados podem ser usados. Quando for fazer isto, certifique-se de saber o que está fazendo.

**2- Fábrica de Beans do Contexto de Persistência:** Uma outra forma de obter o template pronto e configurado para uso seja nos DAO's ou em qualquer lugar da aplicação onde for necessário é a partir da fábrica de beans do contexto de persistência:

*org.lindbergframework.persistence.context.LindbergPersistenceSpringBeanFactory*

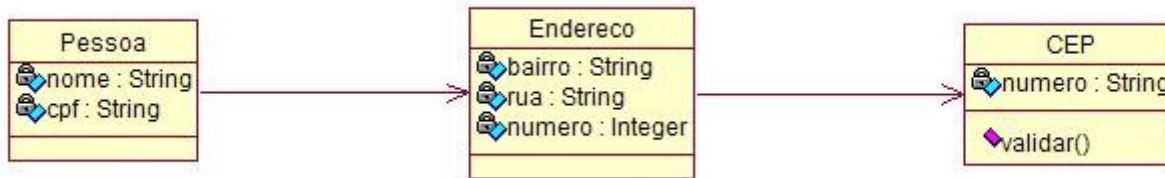
Essa fábrica é um *singleton* e é responsável pelos beans pertencentes ao contexto de persistência como a implementação de *PersistenceTemplate* padrão, o *BeanPopulator* padrão, entre outros.

Abaixo é demonstrado como é feita a obtenção da instância do template via a fábrica descrita:

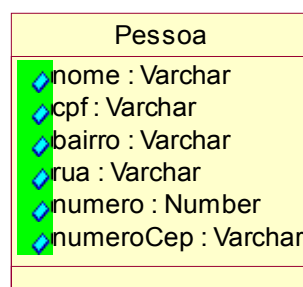
```
PersistenceTemplate template = LindbergPersistenceSpringBeanFactory.getInstance()
                              .getBean(PersistBeans.DEFAULT_PERSISTENCE_TEMPLATE);
```

## 5.10 – Exemplo de uso do LINP na prática

Para demonstrar na prática DAO's usando o *LINP*, considere as mesmas classes que apresentamos anteriormente para abordar a população multinível e que estão novamente apresentadas abaixo:



Considere que no nosso banco de dados nós temos uma única tabela de pessoa como a seguir:



Vamos considerar que estamos usando um banco *oracle* e os dados para conexão sejam:

**URL:** jdbc:oracle:thin:@localhost:1521:xe

**USER:** SYSTEM

**PASSWORD:** admin

Considere que para o exemplo a seguir nós vamos usar uma implementação de *datasource* do spring que é *org.springframework.jdbc.datasource.DriverManagerDataSource* e que o driver que vamos usar seja o já provido pelo oracle *oracle.jdbc.OracleDriver*.

Considere também que nós vamos usar a implementação padrão de repositório de comandos sql, *ClassPathXmlSqlCommandResolver*, que trabalha com arquivos XML que obedecem ao schema <http://www.lindbergframework.org/schema/linp-sqlMapping.xsd> e que em nosso exemplo o nosso repositório de comandos SQL é composto por dois arquivos XML que estão dentro do classpath da nossa aplicação, onde um arquivo é só para *consultas* (queries) e outro apenas para *atualizações* (updates) denominados respectivamente como *exemploQueries.xml* e *exemploUpdates.xml*.

Para a definição dessas configurações no framework considere o seguinte arquivo de configuração dentro do pacote *org.lindbergframework.exemplo.conf*:

\* lindberg-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<lindberg-configuration xmlns="http://www.lindbergframework.org/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.lindbergframework.org/schema lindberg-config.xsd">

    <core>
        <config-property name="lindberg.core.di-basepackage"
            value="org.lindbergframework.exemplo.*"/>
    </core>

    <linp>
        <dataSource
            class="org.springframework.jdbc.datasource.DriverManagerDataSource"
            driver="oracle.jdbc.OracleDriver" driverPropertyName="driverClassName">
            <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
            <property name="username" value="SYSTEM" />
            <property name="password" value="admin" />
        </dataSource>

        <config-properties>
            <config-property name="lindberg.persistence.IntegerCursorType"
                value="#oracle.jdbc.OracleTypes.CURSOR" />
            <config-property name="lindberg.persistence.SqlCommandResolver"
                value="org.lindbergframework.persistence.sql.ClassPathXmlSqlCommandResolver">
                <property constructor-arg="true">
                    <array>
                        org/lindbergframework/exemplo/sql/exemploQueries.xml;
                        org/lindbergframework/exemplo/sql/exemploUpdates.xml
                    </array>
                </property>
            </config-property>
        </config-properties>
    </linp>
</lindberg-configuration>
```

No arquivo xml acima (*lindberg-config.xml*) definimos as configurações necessárias para o funcionamento do *linp*.

A tag *core* define, neste caso, apenas a configuração do pacote raiz onde a partir deste será feito o escaneamento dos beans anotados com a annotation *@Bean* que comporão o contexto de Beans que será acessado via o mecanismo de inversão de controle provido pelo framework. Este mecanismo provê acesso ao contexto e consequentemente aos beans nele contidos através de *UserBeanContext.getInstance()*. Um bean pode ser obtido totalmente pronto e com suas dependências resolvidas a partir do contexto chamando o método *getBean* de *UserBeanContext*.

A tag *linp* define toda a configuração de persistência. Observe que inicialmente temos a tag *dataSouce*. Essa tag define as configurações da conexão propriamente dita: classe da implementação de *DataSource* a ser usada, classe do *driver* a ser usada, url, usuário e senha do banco.

Abaixo dessa tag *dataSource* temos uma tag *config-properties* que agrupa uma série de tags *config-property* onde cada tag dessa última define uma propriedade de configuração específica. Neste caso como usamos o banco Oracle, para este banco nós devemos explicitar o tipo de dado sql para cursores usando uma constante definida pelo próprio drive do oracle:

```
<config-property name="lindberg.persistence.IntegerCursorType"
    value="#oracle.jdbc.OracleTypes.CURSOR" />
```

Para saber mais sobre o uso do carácter '#' veja a sessão ***“Uso do carácter '#' no XML para definir métodos getters estáticos e propriedades estáticas”***.

A próxima propriedade define um repositório de comandos sql a ser usados (*SqlCommandResolver*).

```
<config-property name="lindberg.persistence.SqlCommandResolver"
    value="org.lindbergframework.persistence.sql.ClassPathXmlSqlCommandResolver">
    <property constructor-arg="true">
        <array>
            org/lindbergframework/emplo/sql/emploQueries.xml;
            org/lindbergframework/emplo/sql/emploUpdates.xml
        </array>
    </property>
</config-property>
```

Para este exemplo usamos a implementação comum e mais usada que é o repositório de comandos sql usando XML onde os arquivos de XML estão dentro do classpath da aplicação *ClassPathXmlSqlCommandResolver*. Definimos isso informando um parâmetro de construtor desta classe através de `<property constructor-arg="true">` sobre o qual este é um array de Strings `<array>` que contém elementos com os valores:

```
org/lindbergframework/emplo/sql/emploQueries.xml;
org/lindbergframework/emplo/sql/emploUpdates.xml
```

Esses valores são os arquivos de comandos sql que estão dentro do class path da aplicação e que comporão o repositório de comandos sql. Para saber mais informações sobre a completa configuração usando XML leia a sessão ***“Configurando o LINP na prática – Usando XML”***.

Com o arquivo de configuração do framework já criado falta agora nós criarmos esses arquivos XML que comporão o repositório de comandos sql. Para este exemplo vamos separar os comandos de consulta dos de atualização respectivamente nos arquivos *emploQueries.xml* e *emploUpdates.xml*, ambos no pacote *org/lindbergframework/emplo/sql*.

## \* exemploQueries.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Linp-Mapping xmlns="http://www.lindbergframework.org/schema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.lindbergframework.org/schema linp-sqlMapping.xsd">

<Description>Exemplo de Queries</Description>

  <Sql-Mapping>
    <Sql-Command id="listarPessoas">
      <Sql-Statement>select nome, cpf, bairro as endereco.bairro,
                           rua as endereco.rua, numero as endereco.numero,
                           numeroCep as endereco.cep.numero
                        from pessoa
                        </Sql-Statement>
    </Sql-Command>

    <Sql-Command id="consultarPessoa">
      <Sql-Statement>select nome, cpf, bairro as endereco.bairro,
                           rua as endereco.rua, numero as endereco.numero,
                           numeroCep as endereco.cep.numero
                        from pessoa where cpf = ?
                        </Sql-Statement>
    </Sql-Command>

    <Sql-Command id="listarPessoasPorIniciaisNomeUsandoProcedure">
      <Procedure name="listarPessoasPorNomeProc" >
        <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/>
      </Procedure>
    </Sql-Command>

    <Sql-Command id="listarPessoasPorIniciaisNomeUsandoFunction">
      <Function name="listarPessoasPorNomeFunc">
        <ResultCursor population-class="org.lindbergframework.exemplo.beans.Pessoa" />
      </Function>
    </Sql-Command>

    <Sql-Command id="qtdPessoasUsandoFunction">
      <Function name="qtdPessoas" />
    </Sql-Command>

    <Sql-Command id="qtdPessoas">
      <Sql-Statement>select count(*) from pessoa</Sql-Statement>
    </Sql-Command>

  </Sql-Mapping>
</Linp-Mapping>
```

## \* exemploUpdates.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Linp-Mapping xmlns="http://www.lindbergframework.org/schema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.lindbergframework.org/schema linp-sqlMapping.xsd">

<Description>Exemplo de Updates</Description>

  <Sql-Mapping>
    <Sql-Command id="inserirPessoa">
      <Sql-Statement>insert into pessoa values(?,?,?,?,?,?)</Sql-Statement>
    </Sql-Command>

    <Sql-Command id="excluirPessoa">
      <Sql-Statement>delete from pessoa where cpf = ?</Sql-Statement>
    </Sql-Command>

    <Sql-Command id="atualizarEnderecoPessoa">
      <Sql-Statement>update pessoa set bairro = ?, rua = ?, numero = ?,
                           numeroCep = ? where cpf = ?
                        </Sql-Statement>
    </Sql-Command>

  </Sql-Mapping>
```



```

<Sql-Command id="excluirPessoaUsandoProcedure">
  <Procedure name="excluirPessoa"/>
</Sql-Command>

<Sql-Command id="excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction">
  <Function name="excPessoasRetornaExcluidosFunc" resultName="PESSOASEXCLUIDAS">
    <!-- Não Excluidos São como resultado da função através de um cursor -->
    <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/>
    <!-- Excluidos -->
    <ResultCursor population-class="org.lindbergframework.exemplo.beans.Pessoa" />
  </Function>
</Sql-Command>

<Sql-Command id="excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure">
  <Procedure name="excPessoasRetornaExcluidosProc">
    <!-- Não Excluidos -->
    <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/>
    <!-- Excluidos -->
    <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/>
  </Procedure>
</Sql-Command>
</Sql-Mapping>
</Limp-Mapping>

```

Para nosso exemplo considere o seguinte *script* de banco *ORACLE*:

```

CREATE TABLE "PESSOA"
(
  "NOME" VARCHAR2(4000),
  "CPF" VARCHAR2(4000),
  "BAIRRO" VARCHAR2(4000),
  "RUA" VARCHAR2(4000),
  "NUMERO" NUMBER,
  "NUMEROCEP" VARCHAR2(4000)
);

Create or replace procedure listarPessoasPorNomeProc(nomeExemplo in varchar, pessoas out sys_refcursor) as
begin
  open pessoas for
    select nome, cpf, bairro as "endereco.bairro",
           rua as "endereco.rua", numero as "endereco.numero",
           numeroCep as "endereco.cep.numero"
    from pessoa where nome like nomeExemplo || '%';
end listarPessoasPorNomeProc;

Create or replace procedure excluirPessoa(p_cpf in varchar) as
begin
  delete from pessoa where cpf = p_cpf;
end excluirPessoa;

Create or replace function listarPessoasPorNomeFunc(nomeExemplo in varchar) return sys_refcursor is
  pessoas sys_refcursor;
begin
  open pessoas for
    select nome, cpf, bairro as "endereco.bairro",
           rua as "endereco.rua", numero as "endereco.numero",
           numeroCep as "endereco.cep.numero"
    from pessoa where nome like nomeExemplo || '%';

  return pessoas;
end listarPessoasPorNomeFunc;

Create or replace function excPessoasRetornaExcluidosFunc(nomeExemplo in varchar, pessoasNaoExcluidas out
sys_refcursor) return sys_refcursor is
  pessoasExcluidas sys_refcursor;
begin
  open pessoasNaoExcluidas for
    select nome, cpf, bairro as "endereco.bairro",
           rua as "endereco.rua", numero as "endereco.numero",
           numeroCep as "endereco.cep.numero"
    from pessoa where not nome like nomeExemplo || '%';

  open pessoasExcluidas for
    select nome, cpf, bairro as "endereco.bairro",
           rua as "endereco.rua", numero as "endereco.numero",
           numeroCep as "endereco.cep.numero"

```

```

from pessoa where nome like nomeExemplo || '%';

delete from pessoa where nome like nomeExemplo || '%';

return pessoasExcluidas;
end excPessoasRetornaExcluidosFunc;

Create or replace procedure excPessoasRetornaExcluidosProc(nomeExemplo in varchar, pessoasExcluidas out
sys_refcursor, pessoasNaoExcluidas out sys_refcursor) as
begin
open pessoasNaoExcluidas for
select nome, cpf, bairro as "endereco.bairro",
           rua as "endereco.rua", numero as "endereco.numero",
           numeroCep as "endereco.cep.numero"
from pessoa where not nome like nomeExemplo || '%';

open pessoasExcluidas for
select nome, cpf, bairro as "endereco.bairro",
           rua as "endereco.rua", numero as "endereco.numero",
           numeroCep as "endereco.cep.numero"
from pessoa where nome like nomeExemplo || '%';

delete from pessoa where nome like nomeExemplo || '%';
end excPessoasRetornaExcluidosProc;

Create or replace function qtdPessoas return number is
qtd number;
begin
select count(*) into qtd from pessoa;
return qtd;
end qtdPessoas;

```

Abaixo temos as classes dos beans sobre a qual nosso exemplo efetuará as operações de persistência, a classe *Pessoa*, *Endereco* e *Cep*:

### \* Pessoa

```

public class Pessoa {

    private String nome;

    private String cpf;

    private Endereco endereco;

    public Pessoa() {
        //
    }

    //setters e getters

}

```

### \* Endereco

```

public class Endereco {

    private String bairro;

    private String rua;

    private Integer numero;

    private Cep cep;

    public Endereco() {
        //
    }

    //setters e getters

}

```

## \* Cep

```
public class Cep {  
    private String numero;  
  
    public Cep() {  
        //  
    }  
  
    //setters e getters  
}
```

Uma vez definida e inicializada a nossa configuração do framework podemos obter um comando facilmente a partir do *SqlCommandResolver*, como abaixo:

```
LinpContext.getInstance().getSqlCommandResolver();
```

Obter um comando diretamente a partir do *SqlCommandResolver* definido não é muito comum e não será muito usado, pois isso só me retorna um comando estático, quando na verdade o que queremos é executar e obter o resultado pronto (lista de beans populadas, quantidade de registros afetados por uma atualização, cursores populados, etc...) e para isto precisamos de um *PersistenceTemplate*, que como já foi abordado anteriormente na sessão “**Template de Persistência (PersistenceTemplate)**” é obtido nos DAO's pronto e configurado para uso a partir da extensão da classe *LinpDAO* que já fornece um método para obtenção desse template.

Para o nosso exemplo vamos usar duas implementações de uma interface de DAO de *Pessoa* que fazem as mesmas operações através do template obtido pelo *LinpDAO*, só que um implementa as operações acessando o repositório de comandos sql e o outro usa sql direto e não acessa em nenhum momento o repositório de comandos. Abaixo a interface para nosso exemplo, *IPessoaDAO*.

**Observação:** Desconsidere os nomes extensos e fora da realidade dos métodos do DAO abaixo. Os nomes são apenas para descrever exatamente o que quero mostrar de funcionalidade do template no uso do método.

```
public interface IPessoaDAO {  
  
    public abstract List<Pessoa> listarPessoas();  
  
    public abstract void inserirPessoa(Pessoa pessoa);  
  
    public abstract void excluirPessoa(Pessoa pessoa);  
  
    public abstract Pessoa consultarPessoa(String cpf);  
  
    public abstract void atualizarEnderecoPessoa(Pessoa pessoa);  
  
    public abstract List<Pessoa> listarPessoasPorIniciaisNomeUsandoProcedure(String exemploNome);  
  
    public abstract void excluirPessoaUsandoProcedure(String cpf);  
  
    public abstract List<Pessoa> listarPessoasPorIniciaisNomeUsandoFunction(String exemploNome);  
  
    public abstract Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction(String exemploNome);  
  
    public abstract Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure(String exemploNome);  
  
    public int getNumeroPessoasCadastradasUsandoFunction();  
  
    public int getNumeroPessoasCadastradas();  
  
}
```

## 5.10.1 - Exemplo DAO acessando *repositório de comandos SQL* definindo no exemplo:

```
import java.math.BigDecimal;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.lindbergframework.beans.di.annotation.Bean;
import org.lindbergframework.exemplo.beans.Endereco;
import org.lindbergframework.exemplo.beans.Pessoa;
import org.lindbergframework.persistence.dao.LinpDAO;
import org.lindbergframework.persistence.sql.SqlArg;
import org.lindbergframework.persistence.sql.SqlFunction;

@Bean("pessoaDAOAcessandoRepositorio")
public class PessoaDAOAcessandoRepositorio extends LinpDAO implements IPessoaDAO{

    public PessoaDAOAcessandoRepositorio() {
        //
    }

    public List<Pessoa> listarPessoas(){
        return getPersistTemplate().execQuery(Pessoa.class, "listarPessoas");
    }

    public void inserirPessoa(Pessoa pessoa){
        Endereco endereco = pessoa.getEndereco();
        getPersistTemplate().execUpdate("inserirPessoa", pessoa.getNome(),
                                         pessoa.getCpf(),
                                         endereco.getBairro(),
                                         endereco.getRua(),
                                         endereco.getNumero(),
                                         endereco.getCep().getNumero());
    }

    public void excluirPessoa(Pessoa pessoa){
        getPersistTemplate().execUpdate("excluirPessoa", pessoa.getCpf());
    }

    public Pessoa consultarPessoa(String cpf){
        return getPersistTemplate().execQueryForObject(Pessoa.class, "consultarPessoa", cpf);
    }

    public void atualizarEnderecoPessoa(Pessoa pessoa){
        Endereco endereco = pessoa.getEndereco();
        getPersistTemplate().execUpdate("atualizarEnderecoPessoa", endereco.getBairro(),
                                         endereco.getRua(),
                                         endereco.getNumero(),
                                         endereco.getCep().getNumero(),
                                         pessoa.getCpf());
    }

    public List<Pessoa> listarPessoasPorIniciaisNomeUsandoProcedure(String exemploNome){
        Map parametrosOut = getPersistTemplate()
            .callProcedure("listarPessoasPorIniciaisNomeUsandoProcedure",
                new SqlArg("nomeexemplo", exemploNome));
        return (List<Pessoa>) parametrosOut.get("pessoas");
    }

    public void excluirPessoaUsandoProcedure(String cpf){
        Map mapParam = new HashMap();
        mapParam.put("p_cpf", cpf);
        getPersistTemplate().callProcedure("excluirPessoaUsandoProcedure", mapParam);
    }

    public List<Pessoa> listarPessoasPorIniciaisNomeUsandoFunction(String exemploNome){
        Map parametrosOut = getPersistTemplate()
            .callFunction("listarPessoasPorIniciaisNomeUsandoFunction",
                new SqlArg("nomeexemplo", exemploNome));
        return (List<Pessoa>) parametrosOut.get(SqlFunction.DEFAULT_RESULT_NAME);
    }

    public Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction(String exemploNome){
```

```

        return getPersistTemplate()
            .callFunction("excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction",
                new SqlArg("nomeexemplo", exemploNome));
    }

    public Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure(String exemploNome) {
        return getPersistTemplate()
            .callProcedure("excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure",
                new SqlArg("nomeexemplo", exemploNome));
    }

    public int getNumeroPessoasCadastradas() {
        return getPersistTemplate().execQueryForObject(Integer.class, "qtdPessoas");
    }

    public int getNumeroPessoasCadastradasUsandoFunction() {
        Map<String, BigDecimal> map = getPersistTemplate().callFunction("qtdPessoasUsandoFunction");
        BigDecimal retorno = map.get(SqlFunction.DEFAULT_RESULT_NAME);
        return retorno.intValue();
    }
}

```

### \* Detalhando as operações do DAO *PessoaDAOAcessandoRepositorio*:

Observe o quanto escrevemos pouco em cada um dos métodos. Tem alguns métodos como inserção, listagem, remoção que possuem apenas uma linha de código. Nos casos das listagens e consultas, essa única linha de código já nos dá as instâncias prontas e populadas. Semelhante ao que ocorre com o Hibernate ou JPA. Só que aqui estamos usando JDBC.

- ***listarPessoas***: Faz a listagem de pessoas cadastradas. A lista de pessoas é retornada contendo objetos pessoa completamente populados bem como seus atributos complexos também. Fazemos isso com uma linha apenas de código. Sem sujar nosso DAO com sql nem com aquelas iterações chatas em *ResultSets* para criar cada objeto baseado em cada linha retornada no *ResultSet* para a consulta.

Observe que chamamos o método *execQuery* do template passando com qual classe queremos popular cada linha do resultado da consulta. Passamos também o ID do comando sql no repositório de comandos que será executado para efetuar a consulta. Neste caso o ID é '*listarPessoas*'. Esse ID de comando está definido no nosso XML *exemploQueries.xml*, pois estamos usando um repositório de comandos baseado em XML.

#### \* Código java:

```

public List<Pessoa> listarPessoas() {
    return getPersistTemplate().execQuery(Pessoa.class, "listarPessoas");
}

```

#### \* Declaração do comando '*listarPessoas*' no repositório:

```

<Sql-Command id="listarPessoas">
    <Sql-Statement>select nome, cpf, bairro as endereco.bairro,
                       rua as endereco.rua, numero as endereco.numero,
                       numeroCep as endereco.cep.numero
    from pessoa
    </Sql-Statement>
</Sql-Command>

```

No sql desse comando estamos tirando proveito do conceito de população multinível. Nossa classe Pessoa possui alguns atributos complexos que estes por sua vez possuem outros atributos e por aí vai. Neste caso por exemplo, Pessoa tem *Endereco* e *Endereco* tem *Bairro*. A consulta SQL retorna o valor de bairro apenas como um valor único. Esse valor tem que ser setado em *Endereco* e o *Endereco* em

Pessoa. Para dizer ao *Linp* esse local onde seta o valor nós declaramos no comando '*endereco.bairro*' dizendo que o valor da coluna bairro será setado em *Endereco* e *Endereco* dentro a própria instância onde está sendo populado o resultado. Neste caso, *Pessoa*.

- ***inserirPessoa***: Efetua a inserção de uma nova *Pessoa*. Para tal, é chamado o método *execUpdate* do template passando o ID do comando, '*inserirPessoa*', e os parâmetros do sql na ordem em que são declarados (usando '?') no comando.

\* Código Java:

```
public void inserirPessoa(Pessoa pessoa){
    Endereco endereco = pessoa.getEndereco();
    getPersistTemplate().execUpdate("inserirPessoa", pessoa.getNome(), pessoa.getCpf(),
                                    endereco.getBairro(),
                                    endereco.getRua(),
                                    endereco.getNumero(), endereco.getCep().getNumero());
}
```

\* Declaração do comando '*inserirPessoa*' no repositório:

```
<Sql-Command id="inserirPessoa">
  <Sql-Statement>insert into pessoa values(?,?,?,?,?,?)</Sql-Statement>
</Sql-Command>
```

- ***excluirPessoa***: Efetua a exclusão de uma *Pessoa*. Para tal, é chamado o método *execUpdate* do template passando o ID do comando, '*excluirPessoa*', e os parâmetros do sql na ordem em que são declarados (usando '?') no comando.

\* Código Java:

```
public void excluirPessoa(Pessoa pessoa){
    getPersistTemplate().execUpdate("excluirPessoa", pessoa.getCpf());
}
```

\* Declaração do comando '*excluirPessoa*' no repositório:

```
<Sql-Command id="excluirPessoa">
  <Sql-Statement>delete from pessoa where cpf = ?</Sql-Statement>
</Sql-Command>
```

- ***consultarPessoa***: Efetua a consulta de uma *Pessoa*. Neste caso como queremos consultar uma pessoa pelo identificador único da mesma, usamos o método *execQueryForObject* do template. Esse método faz a consulta onde é esperado o retorno de no máximo uma linha. Se nenhuma linha for retornada o retorno é *null*. Se uma linha for retornada, essa linha é populada em uma instância da classe passada como esperada para o resultado como parâmetro do método. Neste caso a classe *Pessoa*. Se a consulta retornar mais de um registro então uma *NonUniqueRowException* é lançada.

\* Código Java:

```
public Pessoa consultarPessoa(String cpf){
    return getPersistTemplate().execQueryForObject(Pessoa.class, "consultarPessoa", cpf);
}
```

\* Declaração do comando '*consultarPessoa*' no repositório:

```
<Sql-Command id="consultarPessoa">
  <Sql-Statement>select nome, cpf, bairro as endereco.bairro,
                        rua as endereco.rua, numero as endereco.numero,
                        numeroCep as endereco.cep.numero
                    from pessoa where cpf = ?
  </Sql-Statement>
</Sql-Command>
```

- **atualizarEnderecoPessoa:** Efetua a atualização de um endereço de uma pessoa. Neste caso, queremos atualizar o endereço específico de uma pessoa de cpf determinado. Para tal, o template funciona da mesma forma que usamos para fazer a inserção de uma nova pessoa pois o template entende inserção, atualização e remoção tudo como atualização na base. Então funciona da mesma forma. Esse método *execUpdate* retorna um *int* indicando quantos registros foram afetados pela atualização, seja ela qual for.

\* Código Java:

```
public void atualizarEnderecoPessoa(Pessoa pessoa){
    Endereco endereco = pessoa.getEndereco();
    getPersistTemplate().execUpdate("atualizarEnderecoPessoa", endereco.getBairro(),
                                    endereco.getRua(),
                                    endereco.getNumero(),
                                    endereco.getCep().getNumero(),
                                    pessoa.getCpf());
}
```

\* Declaração do comando '*consultarPessoa*' no repositório:

```
<Sql-Command id="atualizarEnderecoPessoa">
  <Sql-Statement>update pessoa set bairro = ?, rua = ?, numero = ?,
                                numeroCep = ? where cpf = ?</Sql-Statement>
</Sql-Command>
```

**NOTA:** Os próximos métodos descritos, mostram o uso fácil de stored procedures e functions. Observe como é fácil, transparente e pouco verboso trabalhar com procedures e functions sejam usando ou não cursores. Desconsidere os nomes do mesmo pois estes não refletem um nome real de método em um sistema real. O intuito é apenas descrever a funcionalidade do template que está sendo abordada no método.

- **listarPessoasPorIniciaisNomeUsandoProcedure:** Efetua a listagem de pessoas cadastradas que tem o nome com as iniciais passadas como parâmetro para o método usando uma stored procedure que tem um *cursor* como parâmetro de saída e este cursor contém as pessoas retornadas pela consulta. Para tal, usamos o método *callProcedure* do template passando o ID do comando referente a stored procedure no repositório de comandos sql, '*listarPessoasPorIniciaisNomeUsandoProcedure*'. Note que precisamos passar as iniciais do nome, sobre qual a busca vai filtrar as pessoas cadastradas, como argumento (parâmetro de entrada) para a procedure. Isso é feito definindo um *SqlArg* onde o nome do parâmetro na assinatura da procedure e o seu respectivo valor são passados via esse *SqlArg*. Passando esse *SqlArg* para o método *callProcedure*, que aceita um varargs de *SqlArg*, ou seja *SqlArg...*), estamos definindo um o valor de argumento para o parâmetro da procedure. Neste caso, o nome do parâmetro é '*NOMEEXEMPLO*' e seu valor é a variável '*exemploNome*'.

Lembra que esta procedure tem um parâmetro de saída e que é um *cursor*? Observe que em nenhum momento aqui no nosso código java nós declaramos isso. Então como o *Linp* sabe que essa



procedure tem um cursor de saída e que este deve ser populado em objetos do tipo *Pessoa*? Simples, isso está definido na declaração do comando la no repositório. Como estamos usando uma repositório de comandos SQL baseado em XML, observe no trecho abaixo do XML do comando a tag `<OutCursor>`. A presença de cada tag dessa na declaração de uma procedure define um cursor de saída e cada tag `<OutCursor>` define qual classe será usada para popular o cursor via o atributo *population-class*.

Uma vez que chamamos a procedure o template nos devolve um *Map* como resultado contendo os parâmetros de saída onde a chave é o nome do parâmetro declarado na assinatura da procedure e o valor é o próprio valor do parâmetro de saída. No caso dos *cursores* o valor é uma lista de objetos do tipo definido via *population-class* já prontos e populados a partir do *cursor*. Neste caso, uma lista de objetos do tipo *Pessoa*.

#### \* Código Java:

```
public List<Pessoa> listarPessoasPorIniciaisNomeUsandoProcedure(String exemploNome) {
    Map parametrosOut = getPersistTemplate()
        .callProcedure("listarPessoasPorIniciaisNomeUsandoProcedure",
            new SqlArg("nomeexemplo", exemploNome));
    return (List<Pessoa>) parametrosOut.get("pessoas");
}
```

#### \* Declaração do comando '*listarPessoasPorIniciaisNomeUsandoProcedure*' no repositório:

```
<Sql-Command id="listarPessoasPorIniciaisNomeUsandoProcedure">
  <Procedure name="listarPessoasPorNomeProc" >
    <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/>
  </Procedure>
</Sql-Command>
```

- ***excluirPessoaUsandoProcedure***: Efetua a exclusão de uma pessoa só que usando uma procedure. Para tal, chamamos novamente o método *callProcedure* do template passando o ID do comando referente a procedure e o parâmetro de entrada, que é o *CPF* da pessoa que se deseja excluir. Observe que neste caso usamos uma forma diferente de passar parâmetros para a procedure. Usamos um *Map*. Essa sobrecarga do método *callProcedure* permite passar um *Map* onde cada item possui como chave o nome do parâmetro exatamente como está definido na assinatura da procedure e o seu valor é o respectivo valor do parâmetro. Observe que neste caso como não temos nenhum *cursor* nem qualquer parâmetro de saída a declaração do comando referente a procedure fica bem simples e se resume apenas ao ID do comando e o nome da procedure.

#### \* Código Java:

```
public void excluirPessoaUsandoProcedure(String cpf) {
    Map mapParam = new HashMap();
    mapParam.put("p_cpf", cpf);
    getPersistTemplate().callProcedure("excluirPessoaUsandoProcedure", mapParam);
}
```

#### \* Declaração do comando '*excluirPessoaUsandoProcedure*' no repositório:

```
<Sql-Command id="excluirPessoaUsandoProcedure">
  <Procedure name="excluirPessoa"/>
</Sql-Command>
```

- ***listarPessoasPorIniciaisNomeUsandoFunction***: Efetua a mesma listagem de pessoas filtrando pelas

iniciais do nome que é feita no método *listarPessoasPorIniciaisNomeUsandoProcedure* só que neste caso é usada uma *stored function* que retorna como resultado um *cursor* contendo as pessoas retornadas pela consulta. Para tal, chamamos o método *callFunction* passando o ID do comando no repositório referente a function e da mesma forma como fizemos quando usamos procedure passamos o parâmetro de entrada '*exemploNome*' via instância de *SqlArg*. Mais uma vez observe que não referenciamos em lugar algum aqui a classe que será usada para popular o *cursor* retornado como resultado, nem muito menos que o resultado dessa função é um *cursor*. Isto mais uma vez é definido na declaração do comando no repositório.

Neste caso, no XML via a tag `<ResultCursor>` disponível hierarquicamente abaixo e exclusivamente para tag `<Function>`. A tag `<ResultCursor>` também possui um atributo *population-class* para a definição de qual classe será usada para popular o *cursor*. A tag `<Function>` também possui um atributo *resultName* que pode ser definido para configurar a *chave/alias/nome* do valor do resultado da função no Map retornado pelo método *callFunction*.

No exemplo usando procedure foi dito que o Map retornado pelo método *callProcedure* contém os parâmetros de saída de *procedure*. O método *callFunction* funciona da mesma forma, só que contém não apenas os parâmetro de saída mas também o resultado/retorno da *function*. Observe que neste caso não definimos no nosso xml do comando o atributo *resultName*. Quando este atributo não é definido o *Linp* usa o nome de chave padrão, que é definido na constante *SqlFunction.DEFAULT\_RESULT\_NAME*. Neste exemplo, como não definimos *resultName* usamos essa constante para obter o valor do resultado da função no Map retornado pelo método *callFunction*.

#### \* Código Java:

```
public List<Pessoa> listarPessoasPorIniciaisNomeUsandoFunction(String exemploNome) {
    Map parametrosOut = getPersistTemplate().callFunction("listarPessoasPorIniciaisNomeUsandoFunction",
                                                         new SqlArg("nomeexemplo", exemploNome));
    return (List<Pessoa>) parametrosOut.get(SqlFunction.DEFAULT_RESULT_NAME);
}
```

#### \* Declaração do comando '*listarPessoasPorIniciaisNomeUsandoFunction*' no repositório:

```
<Sql-Command id="listarPessoasPorIniciaisNomeUsandoFunction">
  <Function name="listarPessoasPorNomeFunc">
    <ResultCursor population-class="org.lindbergframework.exemplo.beans.Pessoa" />
  </Function>
</Sql-Command>
```

- ***excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction***: Efetua a exclusão das pessoas de acordo com as iniciais do nome usando uma function e retorna duas listas de pessoas, uma contendo as pessoas que foram excluídas e outra contendo as não excluídas. As pessoas excluídas são retornadas como resultado da function a partir de um *cursor* e as não excluídas usando um *cursor* também só que a partir de um parâmetro de saída. Para tal, chamamos novamente o método *callFunction*, passando o parâmetro *exemploNome* via *SqlArg*. O Map retornado pelo método contém tanto a lista de pessoas excluídas retornada como resultado da function quanto a lista das pessoas não excluídas retornada via parâmetro de saída. Observe que neste caso declaramos explicitamente o nome que queremos para a chave referente ao resultado da function no Map retornado. Fazemos isso definindo o atributo *resultName*, que neste caso foi '*PESSOASEXCLUIDAS*'. Tanto para o resultado da function quanto para o parâmetro de saída nós definimos que o cursor deve ser populado usando a classe *Pessoa*. Observe no trecho a abaixo onde é mostrado um exemplo de chamada que o Map é obtido e as listas prontas tanto de pessoas excluídas quanto

de não excluídas a partir do Map.

#### \* Código Java:

```
public Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction(String exemploNome) {  
    return getPersistTemplate().callFunction("excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction",  
                                             new SqlArg("nomeexemplo", exemploNome));  
}
```

#### \* Exemplo de chamada:

```
Map map = pessoaDAO.excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction("M");  
List<Pessoa> pessoasExcluidas = (List<Pessoa>) map.get("pessoasexcluidas");  
List<Pessoa> pessoasNaoExcluidas = (List<Pessoa>) map.get("pessoasnaoexcluidas");
```

#### \* Declaração do comando '*excluirPessoasPorIniciaisRetornandoExcluidos...Function*' no repositório:

```
<Sql-Command id="excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction">  
    <Function name="excPessoasRetornaExcluidosFunc" resultName="PESSOASEXCLUIDAS">  
        <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/><!-- Não Excluidos -->  
        <ResultCursor population-class="org.lindbergframework.exemplo.beans.Pessoa" /><!-- Excluidos -->  
    </Function>  
</Sql-Command>
```

- ***excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure***: Efetua a exclusão das pessoas de acordo com as iniciais do nome da mesma forma como apresentada no método anterior só que usando procedure. A chamada é a mesma só que usando *callProcedure*. Neste caso como estamos usando uma procedure as duas listas, tanto os excluídos quanto os não excluídos, são retornadas como parâmetros de saída. Observe que o exemplo de uso é o mesmo do exemplo anterior usando function pois o Map é obtido da mesma forma o que muda é apenas a forma como o cursor de pessoas excluídas foi retornado. No caso da function, como resultado. No caso da procedure via parâmetro de saída.

#### \* Código Java:

```
public Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure(String exemploNome) {  
    return getPersistTemplate().callProcedure("excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure",  
                                             new SqlArg("nomeexemplo", exemploNome));  
}
```

#### \* Exemplo de chamada:

```
Map map = pessoaDAO.excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure("M");  
List<Pessoa> pessoasExcluidas = (List<Pessoa>) map.get("pessoasexcluidas");  
List<Pessoa> pessoasNaoExcluidas = (List<Pessoa>) map.get("pessoasnaoexcluidas");
```

#### \* Declaração do comando '*excluirPessoasPorIniciaisRetornandoExcluidos...Procedure*' no repositório:

```
<Sql-Command id="excPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure">  
    <Procedure name="excPessoasRetornaExcluidosProc">  
        <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/><!-- Não Excluidos -->  
        <OutCursor population-class="org.lindbergframework.exemplo.beans.Pessoa"/><!-- Excluidos -->  
    </Procedure>  
</Sql-Command>
```

- ***getNumeroPessoasCadastradas***: Obtem o número total de pessoas cadastradas usando um sql direto via *select count*. Observe que neste caso é uma consulta simples, não retorna objetos complexos como nos exemplos anteriores. Apenas retorna a quantidade de pessoas cadastradas. Neste caso, como a consulta retorna apenas uma coluna, o valor direto da coluna será convertido para o tipo passado como parâmetro para o método *execQueryForObject*. Neste caso como estamos obtendo uma simples quantidade, usamos a classe *Integer*. E como temos certeza que será retornada uma única linha então usamos a variação do *execQuery* para consultas que retornam no máximo uma linha, o *execQueryForObject*. Da mesma forma se usarmos o método *execQuery* em uma consulta que retorne um conjunto de linhas mas que contém apenas uma coluna, o *LINP* converterá esse único valor dessa coluna diretamente para o tipo passado como parâmetro. Isto será feito para cada linha do resultado da consulta.

Imagine que temos uma listagem que retorna uma série de linhas mas com apenas uma coluna contendo um valor financeiro. Neste caso o método *execQuery* seria chamado passando por exemplo, como tipo a ser usado para popular o resultado, a classe *Double*. O *LINP* devolveria uma lista de *Double* contendo cada valor de cada linha da listagem.

#### \* Código Java:

```
public int getNumeroPessoasCadastradas() {  
    return getPersistTemplate().execQueryForObject(Integer.class, "qtdPessoas");  
}
```

#### \* Declaração do comando 'qtdPessoas' no repositório:

```
<Sql-Command id="qtdPessoas">  
    <Sql-Statement>select count(*) from pessoa</Sql-Statement>  
</Sql-Command>
```

- ***getNumeroPessoasCadastradasUsandoFunction***: Obtem o mesmo número de pessoas cadastradas do exemplo anterior mas dessa vez usando uma function. Neste caso é uma função simples que não retorna cursor nem nada complexo também. Retorna apenas a quantidade de pessoas cadastradas. Da mesma forma como foi feito no exemplo anterior, o *LINP* retornará o valor direto mas neste caso não é possível especificar o tipo que você deseja que o *LINP* converta diretamente como no exemplo anterior. O *LINP* converterá diretamente para o tipo *JAVA* correspondente ao tipo *SQL* retornado. Neste caso o valor foi convertido como um *BigDecimal* então a partir desse tipo é obtido o valor *inteiro* que o retorno do método requer.

#### \* Código Java:

```
public int getNumeroPessoasCadastradasUsandoFunction() {  
    Map m = getPersistTemplate().callFunction("qtdPessoasUsandoFunction");  
    BigDecimal retorno = (BigDecimal) m.get(SqlFunction.DEFAULT_RESULT_NAME);  
    return retorno.intValue();  
}
```

#### \* Declaração do comando 'qtdPessoasUsandoFunction' no repositório:

```
<Sql-Command id="qtdPessoasUsandoFunction">  
    <Function name="qtdPessoas" />  
</Sql-Command>
```

## - OVERVIEW

Fazendo um *overview* sobre a classe detalhada acima, observe que ela não possui nenhum código SQL diretamente, observe que fazemos atualizações e consulta usando sql, usando stored procedures e functions e que tudo isso é feito sem a necessidade de escrever tanto código, sem a necessidade de se preocupar com popular seus objetos baseados em *ResultSet* ou seja lá o que você esteja usando. Veja que o *Linp* abstrai toda a parte de população multinível, abstrai toda a parte de procedure e functions, tudo referente a cursores. Note que basta um *callProcedure* ou *callFunction* para chamarmos facilmente uma procedure ou function, basta um *execQuery* que retorna uma lista de objetos já prontos e populados baseados no resultado da listagem.

### 5.10.2 - Exemplo DAO sem uso de *repositório de comandos SQL*:

Abaixo é mostrado um outro DAO que efetua as mesmas operações do DAO apresentado anteriormente acessando o repositório só que sem fazer uso do repositório. Essa classe faz exatamente as mesmas operações e prover exatamente os mesmos métodos só que usando sql direto e nos casos de procedures e functions as classes *SqlProcedure*, *SqlFunction* e *SqlFunctionForCursor*.

Observe que agora declaramos tudo programaticamente, definimos cursores como parâmetros de saída tanto de procedures quanto de functions via construtor ou via método *registerSqlOutCursorsParam*. Note que nos casos onde temos uma função que seu resultado é um *cursor* usamos uma especialização de *SqlFunction* chamada *SqlFunctionForCursor*.

```
import java.math.BigDecimal;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.lindbergframework.beans.di.annotation.Bean;
import org.lindbergframework.exemplo.beans.Endereco;
import org.lindbergframework.exemplo.beans.Pessoa;
import org.lindbergframework.persistence.dao.LinpDAO;
import org.lindbergframework.persistence.sql.SqlArg;
import org.lindbergframework.persistence.sql.SqlFunction;
import org.lindbergframework.persistence.sql.SqlFunctionForCursor;
import org.lindbergframework.persistence.sql.SqlOutCursorParam;
import org.lindbergframework.persistence.sql.SqlProcedure;

@Bean("pessoaDAOSemAcessarRepositorio")
public class PessoaDAOSemAcessarRepositorio extends LinpDAO implements IPessoaDAO{

    public PessoaDAOSemAcessarRepositorio() {
        //
    }

    @Override
    public void atualizarEnderecoPessoa(Pessoa pessoa) {
        Endereco endereco = pessoa.getEndereco();
        getPersistTemplate().execSqlUpdate("update pessoa set bairro = ?, rua = ?, " +
            "numero = ?, numeroCep = ? where cpf = ?", endereco.getBairro(),
            endereco.getRua(),
            endereco.getNumero(),
            endereco.getCep().getNumero(),
            pessoa.getCpf());
    }

    @Override
    public Pessoa consultarPessoa(String cpf) {
        return getPersistTemplate().execSqlQueryForObject(Pessoa.class,
            "select nome, cpf, bairro as endereco.bairro,"+
            "rua as endereco.rua, numero as endereco.numero,"+
            "numeroCep as endereco.cep.numero"+
            " from pessoa where cpf = ?", cpf);
    }
}
```

```

@Override
public void excluirPessoa(Pessoa pessoa) {
    getPersistTemplate().execSqlUpdate("delete from pessoa where cpf = ?", pessoa.getCpf());
}

@Override
public void excluirPessoaUsandoProcedure(String cpf) {
    Map mapParam = new HashMap();
    mapParam.put("p_cpf", cpf);
    getPersistTemplate().callProcedure(new SqlProcedure("excluirPessoa"), mapParam);
}

@Override
public Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoFunction(
    String exemploNome) {
    SqlFunctionForCursor function = new SqlFunctionForCursor("excPessoasRetornaExcluidosFunc",
        "pessoasexcluidas",
        Pessoa.class,
        new SqlOutCursorParam(Pessoa.class));

    Map map = getPersistTemplate().callFunction(function, new SqlArg("nomeexemplo", exemploNome));
    return map;
}

@Override
public Map excluirPessoasPorIniciaisRetornandoExcluidosENaoExcluidosUsandoProcedure(
    String exemploNome) {
    SqlProcedure procedure = new SqlProcedure("excPessoasRetornaExcluidosProc");
    procedure.registerSqlOutCursorsParam(new SqlOutCursorParam(Pessoa.class),
        new SqlOutCursorParam(Pessoa.class));

    Map map = getPersistTemplate().callProcedure(procedure, new SqlArg("nomeexemplo", exemploNome));
    return map;
}

@Override
public void inserirPessoa(Pessoa pessoa) {
    Endereco endereco = pessoa.getEndereco();
    getPersistTemplate().execSqlUpdate("insert into pessoa values(?,?,?,?,?)",
        pessoa.getNome(), pessoa.getCpf(),
        endereco.getBairro(), endereco.getRua(),
        endereco.getNumero(), endereco.getCep().getNumero());
}

@Override
public List<Pessoa> listarPessoas() {
    return getPersistTemplate().execSqlQuery(Pessoa.class, "select nome, cpf, bairro as endereco.bairro,"+
        "rua as endereco.rua, numero as endereco.numero,"+
        "numeroCep as endereco.cep.numero"+
        " from pessoa");
}

@Override
public List<Pessoa> listarPessoasPorIniciaisNomeUsandoFunction(
    String exemploNome) {
    SqlFunctionForCursor function = new SqlFunctionForCursor("listarPessoasPorNomeFunc",
        SqlFunction.DEFAULT_RESULT_NAME,
        Pessoa.class);

    Map parametrosOut = getPersistTemplate().callFunction(function, new SqlArg("nomeexemplo", exemploNome));
    return (List<Pessoa>) parametrosOut.get(SqlFunction.DEFAULT_RESULT_NAME);
}

@Override
public List<Pessoa> listarPessoasPorIniciaisNomeUsandoProcedure(
    String exemploNome) {
    SqlProcedure procedure = new SqlProcedure("listarPessoasPorNomeProc", new SqlOutCursorParam(Pessoa.class));
    Map parametrosOut = getPersistTemplate().callProcedure(procedure, new SqlArg("nomeexemplo", exemploNome));
    return (List<Pessoa>) parametrosOut.get("pessoas");
}

public int getNumeroPessoasCadastradas() {
    return getPersistTemplate().execSqlQueryForObject(Integer.class, "select count(*) from pessoa");
}

```



```

public int getNumeroPessoasCadastradasUsandoFunction() {
    SqlFunction function = new SqlFunction("qtdPessoas");
    Map<String, BigDecimal> map = getPersistTemplate().callFunction(function);
    return map.get(SqlFunction.DEFAULT_RESULT_NAME).intValue();
}
}

```

O exemplo acima pode ser baixado como projeto do eclipse no link a seguir. O projeto de exemplo contém tanto o exemplo apresentado anteriormente quanto o exemplo referente a gerenciamento de transações e que será apresentado a seguir. Para testar os DAOs apresentados no exemplo anterior, tanto acessando quanto não acessando o repositório de comandos, no projeto de exemplo tem uma classe chamada *LinpTest* que é um simples *main* fazendo uso dos métodos apresentados nos DAOs anteriormente.

**NOTA:** Poderia ter sido usado testes com o *junit* por exemplo mas preferi um simples método *main* com *system.out.println* escrevendo no console cada passo executado de modo a não ser necessário se conhecer nem o *junit*. A classe *LinpTest* aplica um roteiro de operações nas duas implementações da interface *IPessoaDAO*, usando e não usando o repositório de comandos.

**NOTA:** O exemplo abaixo já contém uma pasta *lib* com todas as dependências necessárias para o projeto. Para o exemplo foi usado banco oracle 10g. O script SQL de criação do banco está em `\src\org\lindbergframework\exemplo\sql\create.sql`.

Link do exemplo: <http://www.lindbergframework.org/downloads/ExemploLinp.zip>

As duas implementações da interface *IPessoaDAO* estão anotadas com a annotation *@Bean*, e desta forma ambas as implementações podem ser obtidas a partir do contexto de beans de injeção usando o *singleton UserBeanContext*. A classe *LinpTest* usa *UserBeanContext* para a obtenção dessas implementações.

Observe também que o método *test* da classe *LinpTest* está anotado com *@Transax*. Isso indica que o método deve ser executado dentro de um contexto transacional. Para que isto ocorra note que a própria classe *LinpTest* está anotada também com *@Bean* e que no método *main* da mesma classe nós obtemos a instância de *LinpTest* usando *UserBeanContext*. Isto foi feito para que ao obter a instância a partir do contexto de injeção do framework o recurso de injeção de dependências automática é ativado e desta foram, também o mecanismo transacional a partir da annotation *@Transax*. Se tivéssemos usado o operador *new* para criação da instância de *LinpContext* o método *test* não executaria dentro de um contexto transacional e a annotation *@Transax* de nada adiantaria. O mecanismo de gerenciamento de transações é abordado na próxima sessão.

## 5.11 – Gerenciamento de Transações

O *lindbergframework* fornece um mecanismo de gerenciamento de transações de forma automática e transparente, abstraindo toda a preocupação de gerenciar isto manualmente. O mecanismo trabalha baseado em um gerenciador de transações e contextos transacionais.

Entenda como gerenciador de transações (*TransactionManager*) o objeto que será responsável por executar um determinado contexto dentro de uma transação e de acordo com o resultado dessa execução



decidir o que fazer, *commit* ou *rollback*. Da mesma forma, entenda como um contexto transacional (*TransactionalContext*) um determinado ponto do sistema, como uma classe ou um método específico, que deve ser executado dentro de uma transação onde a partir daí todas as operações de persistência são executadas dentro de uma transação de banco. Esses pontos transacionais são definidos facilmente via annotation e serão abordados mais a frente.

Um gerenciador de transações é definido a partir da interface:

*org.lindbergframework.persistence.transaction.TransactionManager*

Esta interface possui um método chamado *execute* que é chamado pelo contexto de persistência, quando uma transação for requerida, para gerenciar a transação de um contexto transacional específico. O método *execute* possui um parâmetro em sua assinatura que justamente o contexto transacional que deve ser gerenciado. Um contexto transacional é definido a partir da interface:

*org.lindbergframework.persistence.transaction.TransactionalContext*.

Caso uma transação falhe, ou seja, uma exceção dentro contexto transacional seja lançada o *rollback* da transação é efetuado a exceção não checada abaixo é lançada. Caso nenhuma exceção seja lançada e todo o processamento do contexto transacional que deu origem a transação corrente seja concluído, o *commit* é realizado.

*org.lindbergframework.exception.TransactionFailureException*

Você não precisa se preocupar com nada disso. O framework já fornece a implementação padrão de cada uma dessas interfaces. De qualquer forma você pode mudar a implementação de *TransactionManager* que deve ser usada definindo explicitamente na configuração do *LINP*. Isso será mostrado mais a frente.

A implementação padrão provida pelo framework para *TransactionManager* é:

*org.lindbergframework.persistence.transaction.LinpTransactionManager*

A implementação padrão usada para *TransactionalContext* é provida via classe anônima dentro do proxy de transações, *TransactionProxy*. Este proxy é responsável por delegar ao gerenciador de transações definido, o gerenciamento da transação corrente.

Mas você não precisa se preocupar com nada disso. Isso é transparente para aplicação. Você não precisa saber nem manusear nada disso. O *LINP* faz para você. Foi abordado aqui apenas para que saiba de uma forma mais geral a função e responsabilidade das classes que compõem esse importante mecanismo.

### **5.11.1 – Como definir seu próprio Gerenciador de Transações TransactionManager**

Como foi abordado na sessão de configuração *LINP* quando foram apresentadas as propriedades de configuração disponíveis para definições de persistência, é possível definir explicitamente qual gerenciador de transações a ser usado. A interface *LinpConfiguration* define um método chamado *getPersistenceTemplate* e é a partir deste que o contexto de persistência, quando inicializado com uma *LinpConfiguration*, obterá a implementação de *TransactionManager* que deverá ser usada. Abaixo é mostrado como definir um gerenciador de transações usando configuração baseada em XML e programaticamente.

- **Definindo via XML:** Para definir um *TransactionManager* específico usando XML basta definir com propriedade de configuração do *LINP* (dentro da tag *linp*) a propriedade *lindberg.persistence.TransactionManager* e seu respectivo valor como a implementação desejada. Abaixo é mostrado um exemplo desta definição configurando um gerenciador de transações chamado *CustomTransactionManager*.

```
<config-property name="lindberg.persistence.TransactionManager"
    value="com.company.CustomTransactionManager" />
```

- Definindo via programaticamente: A definição de um *TransactionManager* específico, assim como qualquer outra propriedade de configuração, pode ser definido programaticamente. Para tal, basta usar as implementações de *simple* das interfaces de core e persistence. Neste caso como queremos especificar uma propriedade de persistência vamos utilizar uma implementação que fornece métodos de escrita para as propriedades de configuração de persistência, a classe *SimpleLinpConfiguration*. Abaixo é mostrado um exemplo desconsiderando as demais propriedades que são requeridas para a correta configuração do *LINP* e desconsiderando também a configuração do *CORE*, que apesar de não requerida para se usar o *LINP*, será algo que normalmente será feito.

```
SimpleLinpConfiguration linpConfig = new SimpleLinpConfiguration();
//... configurações das demais propriedades de persistência necessárias
linpConfig.setTransactionManager(new CustomTransactionManager());
linpConfig.initializeContext();
```

Abaixo o mesmo exemplo só que configurando o *CORE* e encapsulando o as configurações de persistência no *CORE*. Observe que no exemplo abaixo, apenas chamamos *initializeContext* no *CORE*. Apenas isso é feito porque como as configurações de *LINP* estão encapsuladas no *CORE* então este último, quando inicializado, já inicializará os módulos que ele encapsula. Neste caso, o *LINP*. No log de inicialização aparecerá informando '*inicializando core*' e logo em seguida '*inicializando persistence*' e '*persistence inicializado*'. Só após a inicialização do módulo encapsulado é que a mensagem de inicialização concluída do *CORE* é exibida, '*core inicializado*'.

```
SimpleLinpConfiguration linpConfig = new SimpleLinpConfiguration();
//... configurações das demais propriedades de persistência necessárias
linpConfig.setTransactionManager(new CustomTransactionManager());

SimpleCoreConfiguration simpleCore = new SimpleCoreConfiguration();
//... configurações das demais propriedades de core necessárias
simpleCore.setLinpConfiguration(linpConfig);
simpleCore.initializeContext();
```

### 5.11.2 – A annotation @Transax. Como definir um Contexto Transacional.

Um contexto transacional para o framework, como foi dito anteriormente, é um ponto de entrada onde a partir daí tudo estará sendo efetuado tendo de uma transação. Ou seja, dentro de um contexto transacional. Esse ponto de entrada pode ser um método ou uma classe como um todo.

Para definir estes pontos de entrada que serão processados em um contexto transacional, o framework prover uma anotação simples chama *@Transax*. Essa anotação pode ser usada no escopo de método ou de classe e uma vez utilizada sinaliza ao framework que a partir dali o processamento deve ser realizado em uma transação.

O ponto de entrada definido com a anotação `@Transax` não indica que uma nova transação é requerida. Indica que uma transação é requerida e que sendo assim, se uma transação já existir dentro do contexto sobre o qual o ponto de entrada foi chamado o ponto de entrada em questão participará da mesma transação corrente e uma nova não será criada. Outros frameworks fornecem definições de propagação de transações de modo que seja possível dizer se o ponto de entrada requer uma nova transação ou usa a corrente se existir uma. O *LINP* adota o padrão de participar da transação corrente se esta existir.

**IMPORTANTE:** Para que o mecanismo de gerenciamento de transações seja ativado corretamente e o ponto de entrada realmente seja executado dentro de um contexto transacional, é necessário que a instância da classe onde se encontra o ponto de entrada para o contexto transacional seja obtido a partir do contexto de injeção de dependências do *framework*, a partir de *UserBeanContext* ou da *BeanFactory* definida obtida a partir de *CoreContext.getInstance().getBeanFactory()*. É importante lembrar que para que um bean seja obtido a partir do contexto de beans usando qualquer um desses modos apresentados, a classe do bean deve estar anotada com `@Bean` e esta classe está dentro ou abaixo do pacote base para injeção de dependência (*DIBasePackage*), definido nas configurações do *CORE*. Isso é necessário pois no momento da criação da instância do bean, o LDIC (lindberg dependency injection container) ativa o mecanismo de verificação automática da necessidade da criação ou não do contexto transacional para um ou mais pontos de entrada transacionais que estejam definidos via anotação `@Transax` na classe do bean desejado. Obter a instância via operador *new* não ativará o mecanismo de gerenciamento de transações.

**IMPORTANTE:** Para classes que possuem ao menos um contexto transacional, seja de escopo de classe ou de método, que nunca serão obtidas direta e explicitamente a partir do contexto ou da fábrica de beans e que são obtidas a partir da camadas de persistência não é necessário anotar a classe com `@Bean`, pois a implementação padrão de *BeanPopulator* no momento da população do *Bean* para o retorno como resultado de uma operação de persistência, como consulta, cursores de saída de parâmetros, cursores de resultado de funções, etc... já verifica a necessidade da ativação ou não do contexto transacional para o *Bean* em questão. Então qualquer obtenção, por exemplo de uma entidade *ContaBancaria* que deve ter suas operações executadas dentro de uma transação, basta anotar a classe ou os métodos específicos com devem ser executados no contexto transacional com `@Transax`, e só isso, para que todo o gerenciamento de transações seja ativado e efetuado de forma transparente.

### \* Exemplos de uso

#### - Definindo um contexto transacional em um escopo de classe

Um escopo de classe é quando toda a classe é o ponto de entrada para o contexto transacional. Todo e qualquer método chamado da classe é executado dentro de uma transação. Um bom exemplo disso seria uma classe *ContaBancaria*, onde qualquer operação realizada dentro da mesma deve ser efetuada dentro de um contexto transacional. Então abaixo teríamos um exemplo de uma classe *ContaBancaria* anotada com `@Transax` em escopo de classe.

```
@Transax
public class ContaBancaria {

    public void transferir(ContaBancaria contaDestino, double valor) {
        //...
    }
}
```

```

    public void depositar(double valor){
        //...
    }

    public void sacar(double valor){
        //
    }

    public Extrato emitirExtrato(Date dataInicio, Date dataFim){
        return null; //implementação da emissão do extrato
    }

    public List<ComprovantePagamento> listarComprovantesPagamento(Date dataInicio, Date dataFim){
        return null; //implementação da listagem de comprovantes
    }
}

```

Na classe acima, *ContaBancaria*, todos os métodos são executados dentro de uma transação, pois o contexto transacional foi definido como escopo de classe. Observe que anotamos apenas com *@Transax* e não foi necessário a anotação *@Bean*. Isso foi feito, seguindo o que foi dito anteriormente, pois esta classe corresponde a uma entidade e nunca teremos a necessidade de obter a partir do contexto de beans (ou da fábrica) diretamente uma instância de *ContaBancaria*, pois as entidades serão adicionadas e recuperadas a partir da camada de persistência usando o *LINP* e como foi dito, neste caso basta a anotação *@Transax* para que *LINP* efetue a ativação do contexto transacional.

Da mesma forma considere agora que tivéssemos um controlador de negócio (*BusinessController*) para tratar das operações referentes a negócio da entidade *ContaBancaria*. Uma classe desse tipo, não será persistida nem muito menos recuperada. Ela é uma classe controladora que me fornecerá os métodos pertinentes no que diz respeito a entidade *ContaBancaria* referente ao negócio. Ela será usada por outros objetos atendendo solicitações por exemplo, *criação, atualização, listagem, consulta e exclusão* de contas bancárias. Uma vez que podemos usa-la por exemplo, em uma *Fachada (Facade)* do nosso subsistema, nos nossos controladores de visão (*ViewController*), ou em qualquer outro ponto, é necessário anota-la, caso desejarmos o controle transacional em algum ponto desta classe, com *@Bean*, pois através desta anotação definimos um *ID* para este *Bean* e podemos facilmente obter a instância a partir do contexto de beans, fábrica de beans ou injet-la diretamente em uma propriedade de classe específica via anotação:

*@Inject("id do bean")*

Onde o "id do bean" é o *ID* definido via anotação *@Bean* na classe desejada.

Abaixo vai um exemplo do *BusinessController (BC)* de conta bancária:

```

@Transax
@Bean("ContaBancariaBC")
public class ContaBancariaBCImpl implements ContaBancariaBC{

    //...

}

```

Uma vez definido o *ID* para o *BC* de conta bancária, agora podemos obter diretamente a instância do bean com o gerenciamento transacional ativo a partir do contexto de beans, da fábrica ou injeta-lo diretamente. Abaixo estão os exemplos desses três modos de uso respectivamente:

\* A partir do contexto de Beans:

```
ContaBancariaBC contaBancariaBC = UserBeanContext.getInstance().getBean("contaBancariaBC");
```

### \* A partir da fábrica de Beans:

```
ContaBancariaBC contaBancariaBC = CoreContext.getInstance().getBeanFactory().getBean("contaBancariaBC");
```

### \* Injeção direta em uma propriedade de classe:

```
@Bean("bancoFacade")
public class BancoFacade {

    @Inject("contaBancariaBC")
    private ContaBancariaBC contaBancariaBC;

    //...
}
```

### - Definindo um contexto transacional em um escopo de método:

Observe que no exemplo anterior, quando definimos o contexto transacional para um escopo de classe, todos os métodos da classe *ContaBancaria* estão dentro do contexto. Isso faz com que inclusive os métodos de listagem, *emitirExtrato* e *listarComprovantesPagamento*, também sejam executados em uma transação. Por serem métodos de leitura apenas para extração de dados, não necessariamente temos que ter uma transação para estas operações, deixando dentro do contexto transacional apenas os métodos que requerem realmente uma, *transferir*, *depositar* e *sacar*. Para fazer isto, basta que ao invés de definirmos o contexto com escopo de classe, este seja configurado como de método. Isso é feito anotando diretamente os métodos que comporão o contexto e não a classe. Desta forma os métodos de leitura da classe *ContaBancaria*, quando chamados, não serão executados dentro de uma transação. Já os demais métodos, anotados com *@Transax*, são obrigatoriamente executados em uma transação. Abaixo é mostrada a classe *ContaBancaria* com essas alterações.

**IMPORTANTE:** O fato de um método não estar anotado com *@Transax* não quer dizer que ele não possa participar de uma transação já existente. Um método *X* que não possui contexto transacional, ou seja, não está anotado com *@Transax* e nem está em uma classe que esteja anotada com esta mesma anotação, que é chamado por um método *Y* que é executado dentro de um contexto transacional, por fazer parte da pilha de chamadas dentro contexto transacional de *Y*, está participando da transação iniciada em *Y* e por esta razão caso *X* lance uma exceção, a transação de *Y* falhará e sofrerá *rollback*.

```
public class ContaBancaria {

    @Transax
    public void transferir(ContaBancaria contaDestino, double valor){
        //...
    }

    @Transax
    public void depositar(double valor){
        //...
    }

    @Transax
    public void sacar(double valor){
        //
    }

    public Extrato emitirExtrato(Date dataInicio, Date dataFim){
        return null; //implementação da emissão do extrato
    }

    public List<ComprovantePagameto> listarComprovantesPagamento(Date dataInicio, Date dataFim){
```

```

        return null; //implementação da listagem de comprovantes
    }
}

```

## - Definindo múltiplos contextos transacionais que interagem entre si

Como foi abordado nos dois exemplo anteriores, é possível declarar contextos transacionais tanto em escopo de classe quanto em escopo de método. A questão é, e quando um contexto transacional é chamado/usado por outro contexto, o que acontece? Usando as mesmas classes apresentadas para os exemplos anteriores, imagine que como na maioria dos bancos, para se abrir uma conta seja necessário efetuar um depósito de uma quantia inicial.

Para tal considere o código da classe *ContaBancariaBC* abaixo:

```

@Transactional
@Bean("contaBancariaBC")
public class ContaBancariaBCImpl implements ContaBancariaBC{

    public void criarConta(ContaBancaria contaBancaria, double depositoInicial){
        //...implementação da criação da conta
        contaBancaria.depositar(depositoInicial);
    }
}

```

Como demonstrado no código acima, o método de criação de conta estaria no *BC*, *ContaBancariaBC*, e o método de depósito estaria na classe *ContaBancaria*. O *BC* possui contexto transacional de classe e a classe *ContaBancaria* de método. O método *depositar*, que chamaremos para efetivar o depósito, faz parte do contexto transacional da classe *ContaBancaria*. Então no momento da chamada ao método *criarConta* na classe *ContaBancariaBC*, uma transação é requerida, então caso não exista uma para o contexto corrente uma nova será criada. Após a criação da conta o depósito requerido para criação da mesma é feito chamando o método *depositar* da classe *ContaBancaria*. Uma nova transação não será criada, pois o contexto do método *depositar* foi acessado a partir de outro contexto transacional (método *criarConta*) e como já existe uma transação corrente, este último método participará da mesma.

Observe que todo o processo de criação da conta e depósito foi efetuado dentro de uma mesma transação. Se por exemplo, a criação da conta em si, o *insert* no banco, foi feito sem problemas mas no momento da operação de depósito ocorrer algum problema, todo o processo sofre *rollback* e o *insert* da conta no banco não será efetivado.

## 5.12 – Integrando o LINP com o Springframework

É possível usar o LINP integrado com o Springframework de modo a deixar a injeção de dependências a cargo do *spring* enquanto toda a parte de persistência fica sobre responsabilidade do LINP. Para tal, é necessário uma simples configuração para que o gerenciamento de transações do LINP seja efetivado e trabalhe de forma integrada com o *spring*.

### - Injeção de dependências do spring integrada com o gerenciamento de transações do LINP

Você pode optar também por não usar o mecanismo de injeção do *lindbergframework* e usar o *spring* ao invés disso mas ainda sim usar o mecanismo de transações do LINP. Isso é feito facilmente,



bastando para tal apenas declarar a implementação de *BeanPostProcessor* provida pelo *LINP* que promove esta integração:

*org.lindbergframework.persistence.integration.spring.LinpTransactionBeanPostProcessor*

Exemplo de definição em um xml do spring:

```
<?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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <bean class="org.lindbergframework.persistence.integration.spring.LinpTransactionBeanPostProcessor" />

  </beans>
```

Apenas a simples definição do bean *LinpTransactionBeanPostProcessor* no arquivo de context do spring, como demonstrado acima, já é trabalhar com o mecanismo de injeção de dependências do *spring* usando o gerenciamento de transações do *LINP* via annotation *@Transax*.



## 6 – Importando o lindbergframework a partir do Maven

O lindbergframework está no repositório Maven da sonatype e para importá-lo para um projeto Maven basta declarar a dependência do mesmo no pom.xml e o repositório da sonatype, caso este já não esteja declarado.

O repositório maven onde se encontra o lindbergframework é declarado abaixo:

```
<repository>
  <id>sonatype-releases</id>
  <name>Sonatype Releases Repository</name>
  <url>http://oss.sonatype.org/content/repositories/releases/</url>
</repository>
```

### 6.1 – Importando apenas o módulo padrão do lindbergframework

O módulo padrão é composto pelo Core, Beans e o LINP (Persistence) e para importá-lo basta declarar no pom.xml a dependência abaixo. Isto basta para se usar o mecanismo de injeção de dependências, gerenciamento de transações, população multi-nível, repositório de comandos sql e todos os outros recursos do LINP e CORE:

```
<dependency>
  <groupId>org.lindbergframework</groupId>
  <artifactId>lindbergframework</artifactId>
  <version>1.0</version>
</dependency>
```

**NOTA:** Quando esta documentação foi escrita o lindbergframework estava na versão 1.0. Então para se certificar que está usando a última versão estável do mesmo verifique em [lindbergframework.org](http://lindbergframework.org)

### 6.2 – Importando o módulo padrão do lindbergframework para aplicações WEB

A simples dependência apresentada anteriormente não inclui os recursos para trabalhar com aplicações WEB, como por exemplo, o *listener* para configuração automática do framework na inicialização da aplicação no servidor. Para tal, é necessário a declaração das seguintes dependências:

```
<dependency>
  <groupId>org.lindbergframework</groupId>
  <artifactId>lindbergframework</artifactId>
  <version>1.0</version>
</dependency>

<dependency>
  <groupId>org.lindbergframework.web</groupId>
  <artifactId>lindbergframework-web</artifactId>
  <version>1.0</version>
</dependency>
```

## 6.3 – Importando lindbergframework para aplicações WEB integrando com JSF (Java Server Faces)

Para importar o lindbergframework para se trabalhar com aplicações WEB integrado com JSF fazendo como descrito na *sessão 4.3.1* basta declarar as seguintes dependências:

```
<dependency>
  <groupId>org.lindbergframework</groupId>
  <artifactId>lindbergframework</artifactId>
  <version>1.0</version>
</dependency>

<dependency>
  <groupId>org.lindbergframework.web</groupId>
  <artifactId>lindbergframework-jsf</artifactId>
  <version>1.0</version>
</dependency>
```

## 6.4 – Importando lindbergframework para aplicações WEB integrando com Adobe Flex

Para importar o lindbergframework para se trabalhar com aplicações WEB integrado com Adobe Flex fazendo como descrito na *sessão 4.3.2* basta declarar as seguintes dependências:

```
<dependency>
  <groupId>org.lindbergframework</groupId>
  <artifactId>lindbergframework</artifactId>
  <version>1.0</version>
</dependency>

<dependency>
  <groupId>org.lindbergframework.web</groupId>
  <artifactId>lindbergframework-flex</artifactId>
  <version>1.0</version>
</dependency>
```

## 7 – lindbergframework - Links

Site oficial: [www.lindbergframework.org](http://www.lindbergframework.org)

Google Code: [www.code.google.com/p/lindbergframework](http://www.code.google.com/p/lindbergframework)

Contato, sugestões, críticas, colaboração: [lindbergframework@lindbergframework.org](mailto:lindbergframework@lindbergframework.org)  
[victorlindberg713@gmail.com](mailto:victorlindberg713@gmail.com)

Subversion: <https://lindbergframework.googlecode.com/svn/trunk>

Grupo: <http://groups.yahoo.com/group/lindbergframework>

Issues: <http://www.code.google.com/p/lindbergframework/issues/list>

Postar nova issue: <http://www.code.google.com/p/lindbergframework/issues/entry>

Blog: <http://lindbergframework.blogspot.com>

Repositório Maven – Releases: <https://oss.sonatype.org/content/repositories/releases/org/lindbergframework>

Repositório Maven – Snapshots: <http://oss.sonatype.org/content/repositories/snapshots/org/lindbergframework>