3
A Camada de Negócios

O objetivo da camada de negócios é implementar a lógica da aplicação, expondo esta lógica para a camada de apresentação ou para outras aplicações clientes remotas, por exemplo, clientes móveis, através de interfaces bem definidas. Alternativas para a implementação da camada de negócios envolvem o modelo de componentes Enterprise JavaBeans (EJB, 2005) e o uso de POJOs (POJO, 2005) com frameworks de infra-estrutura.

Neste capítulo, é apresentado um resumo das principais características do framework de componentes Enterprise JavaBeans, focando os serviços de infra-estrutura que este oferece e apresentando as vantagens e desvantagens decorrentes do seu uso. Logo depois, são analisados os frameworks de infra-estrutura Hibernate (2005) e Spring (2005), que oferecem serviços de infra-estrutura similares aos do EJB. Por fim, são oferecidos argumentos que fundamentam a escolha de uma arquitetura com POJOs e frameworks de infra-estrutura ao invés de uma arquitetura com Enterprise JavaBeans.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.1.
O Modelo de Componentes Enterprise JavaBeans

Enteprise JavaBeans (EJB) é uma tecnologia da Sun Microsystems que possibilita o desenvolvimento de componentes distribuídos para a camada de negócios. EJB é um modelo de componentes, isto é, define uma especificação para a construção de componentes (Szyperski, 1997). Servidores de aplicações compatíveis com a especificação J2EE trazem frameworks de componentes que implementam este modelo. EJB também pode ser classificado como um framework de integração de middleware, pois provê suporte à comunicação remota entre componentes através de RMI-IIOP (2005) e também como framework de infra-estrutura, pois trata aspectos como, por exemplo, persistência, segurança e controle de transações, típicos de infra-estrutura.

Há 3 tipos de componentes EJB: os beans de sessão (*Session Beans*), que representam processos síncronos de negócios, os beans de entidade (*Entity Beans*) que representam entidades persistentes e os beans orientados a mensagem (*Message Driven Beans*) que representam processos de negócios assíncronos. Esta seção destaca os aspectos de infra-estrutura mais relevantes do EJB.

Componentes EJB podem ser distribuídos em várias máquinas virtuais Java, neste caso diz-se que os componentes têm interfaces remotas, ou podem estar contidos na mesma máquina virtual, diz-se que têm interfaces locais. Opcionalmente um componente oferece os dois tipos de interfaces. Cabe ao desenvolvedor do componente programar a classe do bean e as interfaces, sejam elas remotas ou locais.

Beans de entidade usam persistência gerenciada pelo bean (*bean managed persistency* - BMP), onde o próprio desenvolvedor do componente programa o código que persiste os dados, ou persistência gerenciada pelo container (*container managed persistency* - CMP), onde o container gera o código que persiste os dados. Beans de entidade podem usar relacionamentos gerenciados pelo container (*container managed relationship* - CMR) onde o container mantém a integridade referencial dos relacionamentos entre beans de entidade.

De forma similar, Enterprise JavaBeans oferece suporte às transações gerenciadas pelo bean (*bean managed transaction* – BMT) e às transações gerenciadas pelo container (*container managed transaction* – CMT). Com BMT, não aplicável a beans de entidade, o desenvolvedor delimita programaticamente o início e o fim de cada transação. Já com CMT o desenvolvedor declara em um arquivo descritor o escopo das transações.

Além destes aspectos, EJB também trata questões relacionadas à segurança. Através do uso da API de segurança do Enterprise JavaBeans, o acesso a um componente ou a um determinado conjunto de operações de um componente pode ser restrito a um grupo de usuários. Este controle é feito de forma declarativa ou de forma programática.

Por fim, o container Enterprise JavaBeans é responsável por gerenciar o ciclo de vida das instâncias dos EJBs. Utilizando pool de objetos, o container é capaz de reutilizar instâncias de beans, reduzindo a necessidade de criar novas instâncias e aumentando assim o desempenho da aplicação. O container também

PUC-Rio - Certificação Digital Nº 0410823/CA

medeia o acesso aos componentes, evitando problemas decorrentes do acesso simultâneo de várias *threads* a uma mesma instância de componente.

EJB oferece uma série de características que oferecem suporte aos aspectos de infra-estrutura de uma aplicação. O principal objetivo da incorporação de aspectos de infra-estrutura a um framework de componentes é possibilitar que o desenvolvedor de componentes concentre-se no domínio em que ele é especialista (no caso do AulaNet, colaboração) deixando aspectos como persistência de dados, por exemplo, para outros especialistas. EJB consegue em parte atingir estes objetivos. Nas próximas seções são vistas as principais vantagens e desvantagens decorrentes do uso de Enterprise JavaBeans.

3.1.1. Vantagens Decorrentes do Uso de Enterprise JavaBeans

Como visto na seção anterior, EJB é um modelo de componentes que também trata de aspectos de infra-estrutura. As principais vantagens do uso de EJB decorrem principalmente da incorporação destes aspectos.

O uso tanto de persistência gerenciada pelo container (CMP) quanto de relacionamentos gerenciados pelo container (CMR) em beans de entidade possibilitam que o desenvolvedor concentre-se na lógica de negócio em vez de concentrar-se na tecnologia que persiste os dados. Além do mais, o código da aplicação fica independente do SGBD (sistema de gerenciamento de banco de dados), pois todos os comandos para consultar, salvar, atualizar e remover entidades são gerados pelo container.

Além disso, o uso de transações declarativas possibilita que o desenvolvedor de componentes construa o componente sem precisar demarcar transações, incentivando o reuso já que o escopo da transação é determinado sem que seja necessário recompilar o código fonte do componente. O uso de CMT também evita a introdução de erros que surgem decorrentes do esquecimento de sessões com o banco de dados abertas e não confirmadas, já que o container se encarrega disto. Da mesma forma, o uso de segurança declarativa incentiva o reuso, pois o componente é customizado com uma política de segurança mais ou menos rígida, dependendo do escopo onde é usado, sem necessidade de mudanças no código fonte.

PUC-Rio - Certificação Digital Nº 0410823/CA

EJB também possibilita que componentes instalados em máquinas virtuais diferentes se comuniquem, de modo que os componentes da camada de negócio de uma aplicação podem ser instalados em máquinas diferentes. Este tipo de distribuição de componentes é útil, por exemplo, nos casos em que um componente reside no mesmo servidor em que está disponível um recurso de que ele necessita (por exemplo, quando um componente precisa acessar um recurso que só aceita conexões locais), no caso de componentes que demandam muito processamento (o componente pode ter um servidor dedicado a ele, não atrapalhando o desempenho dos outros componentes da aplicação) ou quando um componente precisa acessar outro componente do qual os desenvolvedores só tem acesso às interfaces (por exemplo, quando um componente precisa acessar um sistema de cartões de crédito).

O uso de Entreprise JavaBeans traz uma série de vantagens relacionadas a aspectos de infra-estrutura porém há uma série de desvantagens inerentes ao seu modelo de componentes, que são discutidas a seguir.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.1.2. Desvantagens Decorrentes do Uso de Enterprise JavaBeans

A principal desvantagem decorrente do uso de EJBs é o grande número de arquivos, incluindo código fonte escrito em Java e descritores XML, que precisam ser mantidos para definir os beans de sessão e de entidade, os tipos de beans mais usados no desenvolvimento de aplicações com EJB. Para desenvolver um destes componentes é preciso pelo menos 3 arquivos de código fonte: a interface do componente, a interface *home* a classe do componente. Eventualmente, este número pode chegar a 6 arquivos de código fonte, contando apenas as classes previstas na especificação do modelo de componentes EJB.

A interface do componente define os serviços prestados pelo componente e pode ser local ou remota. Se a interface do componente for local, ela deverá estender a interface javax.ejb.EJBLocalObject, se for remota deverá estender a interface javax.ejb.EJBObject.

A interface *home* é responsável por criar instâncias de componentes ou localizar instâncias existentes, no caso de beans de entidade. A interface *home* é definida como local ou remota, de acordo com a definição da interface do

componente. Se a interface home do componente for local, ela deverá estender a interface javax.ejb.EJBLocalHome, se for remota deverá estender a interface javax.ejb.EJBHome.

A classe do componente encapsula as regras de negócio e implementa também a interface javax.ejb.SessionBean no caso de beans de sessão ou a interface javax.ejb.EntityBean no caso de beans de entidade. A classe do componente pode, mas não é obrigada a implementar a interface do componente (a local ou a remota, sendo impossível implementar as duas). Se o desenvolvedor de componentes optar por desenvolver a classe do componente sem implementar a interface do componente, a sincronização entre interface e implementação deve ser realizada manualmente sem o suporte da linguagem Java, o que pode resultar em erros. Por outro lado, se o desenvolvedor de componentes optar por desenvolver a classe do componente implementando a interface do componente, ele deverá implementar os métodos definidos nas interfaces EJBObject, se o componente for remoto, ou EJBLocalObject, se o componente for local. A implementação destes métodos nunca é executada, pois eles são sobrescritos pelo container em tempo de implantação. A classe do componente deve ainda implementar métodos de *callback* definidos na especificação do EJB, como o ejbCreate e o ejbPostCreate por exemplo. Além destes arquivos, no caso de beans de entidade com chave primária composta é preciso mais uma classe para representar esta chave.

Todos os enterprise beans também precisam ser acompanhados de um arquivo descritor em formato XML, onde são declaradas as regras de transação, segurança, etc. Dependendo do servidor de aplicações usado, podem ser necessários outros descritores, onde são declaradas regras como o mapeamento de beans de entidade em tabelas do banco de dados, mapeamento de beans orientado a mensagens ao dispositivo de mensagem entre outros. Esta grande quantidade de arquivos traz problemas de manutenção. Esta desvantagem pode ser em parte amenizada pelo uso de ferramentas como o XDoclets (2005), porém permanece sendo uma solução mais complexa do que usar uma interface Java e uma classe que a implementa.

Além da grande quantidade de arquivos que precisam ser mantidos, componentes Enterprise JavaBeans precisam ser executados dentro de um container EJB. Isto implica em um leque de opções menor na hora de escolher o

PUC-Rio - Certificação Digital Nº 0410823/CA

servidor de aplicações. Alguns servidores populares como o Tomcat (2005) não podem ser usados, pois não possuem container EJB e servidores comerciais que possuem container EJB em geral são mais caros.

Além disso, os componentes EJB são dependentes do container, com diversos métodos de *callback* chamados pelo container EJB ao longo de seu ciclo de vida. Desta forma, componentes EJB são mais difíceis de testar unitariamente, pois precisam estar em execução dentro do container para serem testados. No caso de componentes com interfaces locais, o próprio código de teste deve ser implantado junto com o componente para que este possa ser testado.

No capítulo 25.1.2 da especificação do EJB 2.1 (EJB, 2003) são descritas várias restrições impostas aos componentes EJB, como por exemplo, proibição de escrita ou leitura de arquivos no disco, do uso de campos estáticos não finais e do uso da palavra reservada *this* como referência para a instância do componente em um método, seja como parâmetro ou valor de retorno. Estas restrições se seguidas rigorosamente impossibilitam a realização de atividades corriqueiras como, por exemplo, escrita de arquivos de logs para auditoria ou a implementação do padrão de projeto Singleton (Gamma et al., 1995). Além disso, desenvolvedores Java estão habituados a usar *this* para referenciar a instância do componente e podem ser introduzidos erros de programação decorrentes deste hábito. Todavia na prática, servidores de aplicação como o JBoss (2005) costumam relaxar algumas destas restrições para possibilitar operações como o log de ações.

O uso de EJBs na camada de negócio ao mesmo tempo que traz vantagens por tratar aspectos de infra-estrutura traz uma série de desvantagens expostas nesta seção. Em novembro de 2005, a especificação corrente do EJB era a 2.1, porém a especificação 3.0 estava em desenvolvimento, aberta ao público para discussões.

3.1.3. O Futuro do Enterprise JavaBeans

Apesar da especificação da versão 3.0 do Enterprise JavaBeans ainda não estar completamente definida, a tendência que pode ser percebida através das especificações desta nova versão é tornar o desenvolvimento com EJB menos complexo. Através do uso de anotações, um recurso inserido no J2SE 1.5 que

PUC-Rio - Certificação Digital Nº 0410823/CA

provê suporte a meta-informação no código fonte, o número de arquivos que precisam ser mantidos tende a diminuir.

Nota-se também que o criador do framework Hibernate, Gavin King, é membro do grupo de especialistas da especificação 3.0 do EJB (JSR 220, 2005) e tem exercido influência, tornando o modelo dos beans de entidade mais similar ao modelo de POJOs usado no framework Hibernate. Além disso, um suporte a *Dependency Injection* similar ao do framework Spring, porém mais limitado, está previsto na especificação (Johnson, 2004).

Levará um tempo considerável até que seja viável usar EJB 3.0 em projetos. É preciso esperar que a especificação final seja lançada, que os servidores de aplicação tornem-se compatíveis com a especificação e que as instituições se atualizem com esta nova tecnologia. Deve-se considerar também que as versões das especificações anteriores do EJB frustraram arquitetos e desenvolvedores devido à sua complexidade e que, mesmo que a nova versão amenize os problemas legados, será preciso reconquistá-los (Johnson, 2004; Fowler, 2002; Tate et al., 2003; Sharp et al., 2002).

PUC-Rio - Certificação Digital Nº 0410823/CA

3.2.
Uma Arquitetura Usando POJOs

Enquanto o futuro do EJB permanece incerto, uma arquitetura que utilize POJOs (Plain Old Java Objects) em conjunto com frameworks de infra-estrutura é mais adequada para uso no AulaNet 3.0. O framework Hibernate (2005) provê o mapeamento de POJOs em tabelas de bancos de dados, fornecendo uma funcionalidade similar a dos beans de entidade com persistência e relacionamentos gerenciados pelo container e (CMP/CMR). Já o framework Spring (2005) complementa o Hibernate fornecendo a POJOs serviços de gerenciamento de transações, gerenciamento de segurança declarativa e exportação de serviços remotos, além de outras vantagens não presentes no EJB. Em conjunto estes dois frameworks têm a maior parte das vantagens trazidas pelo EJB, eliminando a maior parte das desvantagens (Johnson, 2004).

Nas próximas seções são analisadas com mais detalhes as questões decorrentes do mapeamento de objetos em tabelas de banco de dados e como o framework Hibernate trata destas questões provendo funcionalidades similares às

dos beans de entidade para uma arquitetura de POJOs. Posteriormente é visto como o framework Spring prove um conjunto de funcionalidades possibilitando o uso de segurança, transações declarativas e exposição de serviços remotos.

3.3. Questões Decorrentes do Mapeamento Objetos em Tabelas

Quase todo tipo de aplicação requer que seus dados sejam persistidos de alguma forma. Um sistema de gerenciamento de banco de dados (SGBD) relacional não é especifico para Java tampouco destinado para uma aplicação específica. A tecnologia relacional provê uma forma de compartilhar dados entre aplicações diferentes e entre tecnologias diferentes usadas em uma mesma aplicação, por exemplo, um sistema gerador de relatórios e um sistema de gerenciamento de cadastro. Devido a estas características, os SGBDs são o meio de persistir dados das entidades de negócios mais utilizado em aplicações (King & Bauer, 2005).

Quando a tecnologia envolvida é Java, a forma mais baixo nível que esta plataforma oferece para enviar comandos ao SGBD é através da API *Java DataBase Connectivity* (JDBC, 2005). Através do JDBC uma aplicação Java é capaz de enviar comandos SQL para um SGBD e assim realizar operações de consulta, atualização, inserção e remoção de dados. O código JDBC fica responsável por realizar a conversão dos dados provenientes do SGBD em classes orientadas a objeto escritas em Java.

Escrever código de acesso a banco de dados, realizando as conversões necessárias do paradigma relacional para o paradigma OO é uma tarefa onerosa, repetitiva e algumas vezes propensa a erros devido à diferença entre estes paradigmas. Para entender o framework Hibernate, é preciso antes entender o conflito dos paradigmas OO e relacional, que ele se propõe a mediar.

3.3.1. Conflito de Paradigmas: Orientação a Objetos x Relacional

Ao migrar de um paradigma para outro surgem questões que devem ser resolvidas. Dentre elas, as principais são: a questão dos subtipos, a questão da igualdade, as questões dos relacionamentos e a questão do grafo de navegações

PUC-Rio - Certificação Digital Nº 0410823/CA

(King & Bauer, 2005). Esta seção não se propõe a apresentar as soluções destas questões, mas apenas apresentá-las. Posteriormente, é mostrado como o Hibernate trata destas questões.

A questão dos subtipos surge, pois o paradigma OO possibilita que classes herdem de outras classes, podendo compor modelos complexos através de herança. Os sistemas de gerenciamento de banco de dados relacionais não oferecem suporte a heranças entre tabelas.

A questão de igualdade surge quando é necessário verificar se um objeto é igual ao outro. Em Java, há duas maneiras de verificar a igualdade de objetos: através do conceito de identidade de objetos, que define que um objeto é igual ao outro se ambos ocupam o mesmo espaço de memória (é checado com o operador de igualdade ‘a == b’) e através do conceito de equivalência de objetos, que é determinada através do método equals() quando este é implementado (é checado através de ‘a.equals(b)’). No paradigma relacional a identidade de uma linha de uma tabela é definida através do valor da chave primária.

Relacionamentos em linguagens OO são expressos através de referências a objetos ou coleções de referências a objetos. Já no paradigma relacional os relacionamentos são expressos na forma de chaves estrangeiras. As questões dos relacionamentos surgem a partir das diferenças destas duas formas de representar relacionamentos.

Relacionamentos no paradigma OO são direcionais, ou seja, são definidos de uma classe para outra. Para criar um relacionamento bidirecional no paradigma OO é preciso definir dois relacionamentos unidirecionais, um em cada uma das classes envolvidas. A noção de direção em relacionamentos não existe no paradigma relacional. Além disto, classes no paradigma OO, podem ser definidas tendo relacionamentos com multiplicidades um-para-um, um-para-muitos ou muitos-para-muitos enquanto que no paradigma relacional os relacionamentos entre tabelas são sempre um-para-muitos, definidos usando chaves estrangeiras, ou um-para-um, definidos usando chaves estrangeiras únicas. Relacionamentos muitos-para-muitos são feitos através de uma tabela extra, chamada tabela de ligação (*link table)* que não aparece no modelo OO.

Finalmente, há a questão do grafo de navegações. O diagrama UML da Figura 3.1 apresenta duas classes: a classe Message, representando uma mensagem que possui uma propriedade *text* que armazena o texto da mensagem e

PUC-Rio - Certificação Digital Nº 0410823/CA

a classe User com duas propriedades, *username* que representa a identificação do usuário e *name*, que representa o nome do usuário. A classe Message tem um relacionamento unidirecional com multiplicidade um-para-um com a classe User.

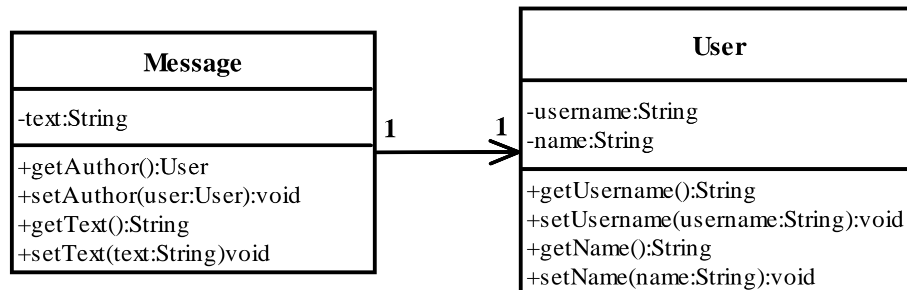

Figura 3.1 – Diagrama de Classes que Exemplifica a Questão do Grafo de Navegação

Em um paradigma orientado a objetos, a forma natural de acessar dados em objetos relacionados é através da técnica onde se navega de um objeto para outro usando o operador de navegação. Por exemplo, em Java para obter o nome do autor de uma mensagem realiza-se a seguinte operação sobre um objeto do tipo Message: message.getAuthor().getName(). Entretanto, esta não é a forma mais eficiente de recuperar dados em um SGBD relacional e pode levar ao problema conhecido como "n + 1 select problem" (King & Bauer, 2005) onde para cada navegação no grafo de objetos é realizada uma consulta na base de dados. Recuperar o grafo inteiro de objetos de uma só vez implicará em problemas de performance e escalabilidade, que tornam-se mais evidentes à medida que a massa de dados da base aumenta e os relacionamentos entre os objetos tornam-se mais complexos. A questão do grafo de navegação envolve então equilibrar estes fatores, determinando qual porção do grafo de navegação deve ser recuperada imediatamente e qual deve ser recuperada sob demanda.

Muitas questões devem ser consideradas quando é preciso converter um objeto em tabelas. Esta tarefa é repetitiva, propensa a erros e independente do domínio da aplicação, portando é susceptível à aplicação de frameworks de infra-estrutura. O modelo de componentes EJB, visto na seção 3.1, soluciona algumas destas questões com os beans de entidade. Outras questões, como o mapeamento de subtipos, não são tratadas. Nas próximas seções é visto como Frameworks de mapeamento objeto/relacional (*Object/Relational Mapping* – ORM) como o Hibernate tratam estas questões.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4. Frameworks ORM

Mapeamento objeto/relacional (object/relational mapping – ORM) é o nome dado para a persistência automática e transparente de classes de uma aplicação em tabelas de bancos de dados relacionais, transformando a representação de dados de um paradigma para o outro (King & Bauer, 2005). Um framework ORM é um framework de infra-estrutura que fornece serviços de ORM.

Mark Fussel (1997), um pesquisador no campo da ORM, define quatro níveis de maturidade para soluções ORM: relacional puro (*pure relational*), mapeamento leve (*light object mapping*), mapeamento médio (*medium object mapping*) e mapeamento completo (*full object mapping*).

Em soluções relacionais puras, a aplicação inteira, incluindo a interface com o usuário, é afetada pelo modelo relacional e por operações envolvendo acesso direto à base de dados relacional com JDBC e SQL. Esta solução pode ser adequada para sistemas pequenos onde um baixo grau de reuso de código é tolerado. Cada consulta SQL pode ser ajustada para se obter melhor performance, o que afeta a manutenabilidade. Está é a solução utilizada na arquitetura do AulaNet 2.1.

Com soluções de mapeamento leve, classes são mapeadas manualmente em tabelas relacionais. O código SQL/JDBC responsável pela persistência das classes é isolado da lógica de negócios através do uso de padrões de projeto bem conhecidos como o Data Access Object (DAO) (Alur et al., 2001). Esta solução é adequada para aplicações de porte pequeno, com um pequeno número de entidades.

Soluções de mapeamento médio são desenvolvidas ao redor de um modelo de objetos. O código SQL responsável pela persistência dos objetos é gerado automaticamente em tempo de compilação por uma ferramenta ou em tempo de execução pelo framework ORM. Este nível de solução oferece suporte a relacionamento entre objetos e fornece uma linguagem orientada a objetos para realização de consultas. Esta solução é adequada para aplicações de médio porte, principalmente quando a aplicação deve permanecer independente dos SGBDs. É

PUC-Rio - Certificação Digital Nº 0410823/CA

o nível de maturidade atingido pelos beans de entidade da especificação 2.1 do EJB.

Já soluções de mapeamento completo oferecem suporte a modelos complexos de objetos, envolvendo composição e herança. A persistência é transparente, ou seja, as classes persistidas não herdam de uma classe ou implementam uma interface do framework. Além disso, o framework implementa estratégias de otimização, como por exemplo o cache de entidades. É o nível de maturidade atingido pelo Hibernate, utilizado na arquitetura do AulaNet 3.0.

Uma solução ORM com mapeamento completo consiste de 3 módulos principais: uma API para realização de operações de consultas, atualizações, inserções e remoção de objetos de classes persistentes; uma linguagem ou API para especificação de consultas referentes a classes e a propriedades de classes; um recurso para especificação de metadados de mapeamento de objetos. A próxima seção apresenta o Hibernate como exemplo deste tipo de solução.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.1. O Framework Hibernate

Hibernate é um framework ORM adotado na arquitetura do AulaNet 3.0 para persistir os objetos de negócio. Hibernate fornece mapeamento completo (Fussel, 1997) e, ao contrário do EJB, trata de todas as questões relativas ao mapeamento de objetos em tabelas levantadas na seção 3.3.1. Através da API do Hibernate são realizadas operações de inserção, atualização, remoção e consulta de objetos, e através da linguagem de consulta do Hibernate (Hibernate Query Language - HQL), consultas mais complexas são realizadas. Os metadados de mapeamento de objetos são definidos em arquivos XML.

3.4.1.1. Hibernate - Um Exemplo Prático

Para melhor compreensão do Hibernate esta seção apresenta um exemplo prático que ilustra o seu funcionamento. O modelo de objetos a ser persistido escolhido é o mesmo apresentado na figura 3.1, com algumas alterações para tornar o exemplo mais ilustrativo: O campo *id*, identificador único do objeto, é

acrescentado em ambas as classes e o campo *date*, data da mensagem, é acrescentado na classe Message. O diagrama modificado é exibido na Figura 3.2.

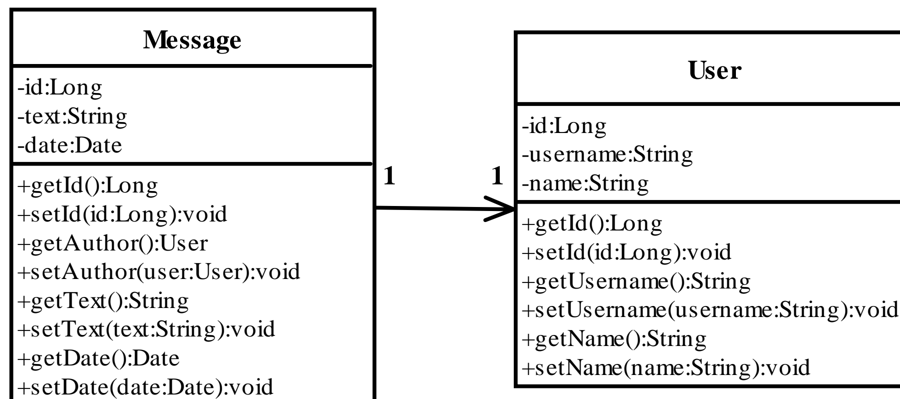

Figura 3.2 – Diagrama UML de Classes Persistidas no Exemplo

A implementação das classes Message e User é realizada usando POJOs, sem quaisquer dependências com classes ou interfaces da API do Hibernate. A Listagem 3.1 exibe o código fonte da classe Message e a Listagem 3.2 exibe o código fonte da classe User.

```
01: public class Message {

02:     private Long id;
03:     private String text;
04:     private Date date;
05:     private User author;

06:     public void setId(Long newValue) { id = newValue; }
07:     public Long getId() { return id; }

08:     public void setText(String newValue) { text = newValue; }
09:     public String getText() { return text; }

10:     public void setDate(Date newValue) { date = newValue; }
11:     public Date getDate() { return date; }

12:     public void setAuthor(User newValue) { author = newValue; }
13:     public User getAuthor() { return author; }

14: }
```

Listagem 3.1 – Classe Message

PUC-Rio - Certificação Digital Nº 0410823/CA

Os trechos entre as linhas 02 e 04 definem as propriedades *id*, *text* e *date* da classe Message. Na linha 05, o relacionamento de mensagem para autor é definido. Os métodos definidos entre as linhas 06 e 13 constituem métodos de acesso para a variável de relacionamento *author* e para as propriedades *id, text* e *date*.

```
15: public class User {

16:     private Long id;
17:     private String username;
18:     private String name;

19:     public void setId(Long newValue) { id = newValue; }
20:     public Long getId() { return id; }

21:     public void setUsername(String newValue) { username = newValue; }
22:     public String getUsername() { return username; }

23:     public void setName(String newValue) { name = newValue; }
24:     public String getName() { return name; }

25: }
```

Listagem 3.2 – Classe User

O trecho entre as linhas 16 e 18 definem as propriedades *id*, *username* e *name* enquanto que os trechos entre as linhas 19 e 24 definem os métodos de acesso para estas propriedades. O próximo passo é configurar o Hibernate.

A configuração do Hibernate pode ser feita de três formas: através de um arquivo de propriedades chamado hibernate.properties, instalado no *classpath* da aplicação, através de um arquivo XML chamado hibernate.cfg.xml, também deve instalado no *classpath* da aplicação, ou através da API do Hibernate. A Listagem 3.3 mostra um exemplo do arquivo XML de configuração do Hibernate.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
26: <?xml version='1.0' encoding='utf-8'?>
27: <!DOCTYPE hibernate-configuration
28:    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
29:    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

30: <hibernate-configuration>
31:     <session-factory>
32:         <property name="show_sql">true</property>
33:         <property name="connection.datasource">
34:             java:/comp/env/jdbc/AulaNetDB
35:         </property>
36:         <property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>

37:         <!-- Mapping files -->
38:         <mapping resource="Message.hbm.xml"/>
39:         <mapping resource="User.hbm.xml"/>
40:     </session-factory>
41: </hibernate-configuration>
```

Listagem 3.3 – Arquivo hibernate.cfg.xml de Configuração do Hibernate

PUC-Rio - Certificação Digital Nº 0410823/CA

O trecho entre a linha 26 e 29 define o esquema do documento XML, para que ele possa ser validado por ferramentas de *parser*. A linha 31 inicia a seção que configura um SessionFactory, classe que será explicada mais adiante. Na linha 32 a propriedade *show_sql* é definida com o valor *true*. Desta forma, o código SQL gerado pelo Hibernate será logado. Esta opção é útil para fins didáticos e para ajustes de performance. O DataSource, que provê conexões com o banco de dados, é configurado no trecho entre as linhas 33 e 35. Na linha 36 o dialeto do Hibernate é configurado. Através do dialeto, o Hibernate gera comandos SQL para diversos tipos de bancos de dados relacionais. Por último, nas linhas 38 e 39 é definido o nome dos arquivos de metadados de mapeamento para, respectivamente, as classes Message e User. Através destes arquivos o Hibernate realiza o mapeamento de objetos em tabelas. A Listagem 3.4 mostra o arquivo responsável pelo mapeamento da classe User.

```
42: <?xml version='1.0' encoding='utf-8'?>
43: <!DOCTYPE hibernate-configuration
44:    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
45:    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

46: <hibernate-mapping>
47:     <class name="User" table="TBL_USER">
48:         <id name="id"
49:              column="USR_ID"
50:              type="long">
51:              <generator class="native"/>
52:         </id>

53:         <property name="username"
54:                      type="string"
55:                      column="USRNAME"
56:                      length="50"
57:                      not-null="true"/>

58:         <property name="name"
59:                      type="string"
60:                      column="NAME"
61:                      length="100"/>
62:     </class>
63: </hibernate-mapping>
```

Listagem 3.4 – Arquivo User.hbm.xml de Mapeamento Objeto – Relacional

O trecho entre a linha 42 e 45 define o esquema do documento XML, para que ele possa ser validado por ferramentas de *parser*. A linha 47 indica que este arquivo se refere ao mapeamento da classe User na tabela TBL_USER. Entre as linhas 48 e 52 são definidas as características do identificador do objeto. A linha 48 indica que o nome da propriedade que identifica o objeto é *id*. A linha 49 indica que esta propriedade é mapeada na coluna USR_ID da tabela TBL_USER Já a linha 50 indica que a propriedade é do tipo *long*, que o Hibernate converterá para o tipo mais apropriado para a base de dados segundo o dialeto configurado. A linha 51 indica que o Hibernate usa o mecanismo nativo do banco de dados para gerar chaves primárias (campos auto incrementáveis no MySQL, seqüências no Oracle e assim por diante).

Entre as linhas 53 e 57 a propriedade *username* é mapeada para a coluna USRNAME. Nota-se que na linha 56 esta propriedade é definida como tendo tamanho máximo 50 caracteres e que na linha 57 a propriedade é definida como requerida, isto é, não pode receber valor nulo. Estas duas características são usadas pela ferramenta hbm2ddl, usada para gerar o esquema da base de dados

PUC-Rio - Certificação Digital Nº 0410823/CA

automaticamente. Por fim, o trecho entre as linhas 58 e 61 define o mapeamento da propriedade *name* na coluna *NAME* de tamanho máximo 100 caracteres. A Listagem 3.5 descreve o arquivo responsável pelo mapeamento da classe Message.

```
64: <?xml version='1.0' encoding='utf-8'?>
65: <!DOCTYPE hibernate-configuration
66:    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
67:    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

68: <hibernate-mapping>
69:     <class name="Message" table="TBL_MESSAGE">
70:         <id name="id"
71:             column="MSG_ID"
72:             type="long">
73:             <generator class="native"/>
74:         </id>
75:         <property name="text"
76:                     type="string"
77:                     column="MSG_TXT"/>
78:         <property name="date"
79:                     type="date"
80:                     column="MSG_DATE"/>

81:         <many-to-one name="author"
82:                         class="User"
83:                         column="AUTHOR_ID"
84:                         not-null="true"
85:                         cascade="save-update"/>

86:     </class>
87: </hibernate-mapping>
```

Listagem 3.5 - Arquivo Message.hbm.xml de Mapeamento Objeto – Relacional

Neste arquivo a classe Message é mapeada na tabela TBL_MESSAGE (linha 69). O arquivo também define o identificador com nome *id* e tipo *long*, mapeado na coluna MSG_ID (trecho entre as linhas 70 e 74), a propriedade *text* do tipo string mapeada na coluna MSG_TXT (trecho entre as linhas 75 e 77) e a propriedade *date* do tipo *date*, mapeada na coluna MSG_DATE (trecho entre as linhas 78 e 80).

O trecho entre as linhas 81 e 85 define o relacionamento entre as classes Message e User. Apesar de ser um relacionamento um-para-um, ele é definido como muitos-para-um, pois a classe Message não é capaz de determinar a cardinalidade do lado “User” do relacionamento. O nome da propriedade da classe

PUC-Rio - Certificação Digital Nº 0410823/CA

Message que referencia a classe User é listado na linha 81. A classe User, com a qual a classe Message se relaciona, é definido na linha 82. O campo AUTHOR_ID definido na linha 83 indica que este é o nome da chave estrangeira armazenada na tabela TBL_MESSAGE, que referencia a tabela TBL_USER. O atributo *not-null* é definido como *true* na linha 84 indica que o relacionamento é obrigatório. Por fim, o atributo *cascade* é configurado com o valor "*save-update*" na linha 83, indicando que as operações de inserção e atualização de entidades devem ser realizadas também para as instâncias relacionadas.

Com todas as configurações feitas, o próximo passo é usar a API do Hibernate para realizar as operações de inserção, atualização, consulta e remoção, também conhecidas com operações CRUD (C*reation, Retrieval, Update, Deletion*). A Listagem 3.6 exibe um exemplo de operação de inserção de um objeto persistente.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
088: public class CRUDOperations01 {

089:     public static void main(String[] args) {
090:         SessionFactory sfactory = null;
091:         Session sess = null;
092:         Transaction tx = null;

093:         try {
094:             sfactory = new Configuration().configure().buildSessionFactory();
095:             sess = sfactory.openSession();
096:             tx = sess.beginTransaction();

097:             User usr = new User();
098:             usr.setUsername("johnd"); usr.setName("John Doe");
099:             Message msg = new Message();
100:             msg.setText("Test message"); msg.setDate(new Date());
101              msg.setAuthor(usr);

102:             sess.save(message);
103:             tx.commit();
104:             sess.close();
105:         } catch (Exception e) {
106:             try {
107:               tx.rollback();
108:               sess.close();
109:             } catch (HibernateException e1) { }
110:             e.printStackTrace();
111:         }
112:     }
113: }
```

Listagem 3.6 – Operação de Criação com o Hibernate

Na linha 091 uma variável do tipo Session é declarada. A interface net.sf.hibernate.Session é a principal interface da API do Hibernate. A sessão do Hibernate, também chamada de gerenciador de persistência, pode ser vista como um gerenciador de objetos relacionados a uma mesma unidade de trabalho. A sessão é capaz de inserir, consultar, atualizar e remover objetos assim como detectar mudanças nos objetos pertencentes à unidade, atualizando seus estados persistentes quando a sessão é fechada. A sessão é um objeto leve, ou seja, não ocupa muito espaço em memória e não leva muito tempo para ser criado e destruído. Geralmente a sessão possui um tempo de vida curto.

Na linha 090 uma variável do tipo SessionFactory é declarada. A interface net.sf.hibernate.SessionFactory define como objetos de sessão são criados. Este é um objeto pesado quando comparado com a sessão, mas a aplicação só precisa criar uma instância deste para cada base de dados.

Uma variável do tipo Transaction é declarada na linha 092. A interface net.sf.hibernate.Transaction representa uma abstração de transações. Através desta interface podem ser usadas transações JDBC, JTA (2005), etc.

O objeto SessionFactory é instanciado na linha 094. A partir deste objeto, a sessão do Hibernate é criada na linha 095. A transação é iniciada na linha 096 a partir da sessão.

Entre as linhas 097 e 100 um objeto Message e um objeto User são criados e inicializados. Na linha 101, o relacionamento entre o objeto Message e o objeto User é configurado. Na linha 102 o objeto é salvo na sessão. Finalmente, a transação é confirmada na linha 103 e a sessão é fechada na linha 104. Como o atributo *cascade* do relacionamento é definido como *save-update* (linha 85 da listagem 3.5), o objeto User associado à classe Message também é salvo. O trecho de código entre as linhas 105 e 111 trata eventuais erros que podem ocorrer durante a operação.

Para efetuar a atualização de objetos é necessário recuperar o objeto que será atualizado, alterar seu estado e fechar a sessão. A Listagem 3.7 apresenta um exemplo da operação de atualização com o Hibernate.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
114: public class CRUDOperations02 {

115:     public static void main(String[] args) {
116:         SessionFactory sfactory = null;
117:         Session sess = null;
118:         Transaction tx = null;

119:         try {
120:             sfactory = new Configuration().configure().buildSessionFactory();
121:             sess = sfactory.openSession();
123:             tx = sess.beginTransaction();
124:             Message msg = (Message) sess.load(Message.class, new Long(1));
125:             msg.setText("Message text changed!");
126:             tx.commit();
127:             sess.close();
128:         } catch (Exception e) {
129              try {
130:                 tx.rollback();
131:                 sess.close();
132:             } catch (HibernateException e1) { }
133:             e.printStackTrace();
134:         }
135:     }
136: }
```

Listagem 3.7 – Operação de Atualização com o Hibernate

A linha 124 mostra um mecanismo usado pelo Hibernate para recuperar instâncias de classes persistidas a partir de sua chave primária. Na linha 125 o texto da mensagem é alterado e na linha 127, quando o objeto Session é fechado, a alteração realizada no objeto é salva, pois ele faz parte da unidade de trabalho gerenciado pela instância da sessão. Embora prático, o mecanismo utilizado na linha 124 para recuperar uma mensagem com um determinado identificador não contempla a maior parte das necessidades reais de consultas. Freqüentemente é necessário realizar consultas em uma aplicação que retornem um objeto ou uma coleção de objetos segundo algum critério.

Hibernate possibilita que consultas complexas sejam realizadas através do HQL (*Hibernate Query Language*). O HQL é de certa forma similar ao SQL, mas agrega conceitos de orientação a objetos, possibilitando a seleção de objetos em vez de tabelas. A Listagem 3.8 exemplifica uma consulta onde todos os usuários com o primeiro nome “Mark” são selecionados.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
137: public class CRUDOperations03 {

138:     public static void main(String[] args) {
139:         SessionFactory sfactory = null;
140:         Session sess = null;

141:         try {
142:             sfactory = new Configuration().configure().buildSessionFactory();
143:             sess = sfactory.openSession();

144:             String queryStr = "from User where user.name like ?";
145:             Query qry = sess.createQuery(queryStr);
146:             qry.setString(0, "Mark%");
147:             Collection usrs = qry.list();

148:             System.out.println(usrs.size() + " users selected.");
149:             for (Iterator i = usrs.iterator(); i.hasNext(); {
150:                 User u = (User) i.next();
151:                 System.out.println(u.getName());
152:             }

153:             sess.close();
154:         } catch (Exception e) {
155:             e.printStackTrace();
156:         }
157:     }
158: }
```

Listagem 3.8 – Operação de Consulta com o Hibernate Usando HQL

Nota-se que não há a necessidade de usar transações dado que apenas a consulta é realizada. A string com o comando HQL executado é criada na linha 144. Um parâmetro é usado no lugar para a propriedade *name*, de modo a tornar a consulta mais genérica. Na linha 145 um objeto Query é criado a partir da sessão. O parâmetro é configurado na linha 146 para que apenas os usuários com nome começando com “Mark” sejam selecionados e a consulta é realizada na linha 147. O trecho de código entre as linhas 148 e 152 imprime o resultado da consulta.

Encerrando o exemplo, a Listagem 3.9 apresenta um exemplo de remoção de uma entidade persistente.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
159: public class CRUDOperations04 {

160:     public static void main(String[] args) {
161:         SessionFactory sfactory = null;
162:         Session sess = null;
163:         Transaction tx = null;

164:         try {
165:             sfactory = new Configuration().configure().buildSessionFactory();
166:             sess = sfactory.openSession();
167:             tx = sess.beginTransaction();
168:             Message msg = (Message) sess.load(Message.class, new Long(1));
169:             sess.delete(msg);
170:             tx.commit();
171:             sess.close();
172:         } catch (Exception e) {
173:             try {
174:               tx.rollback();
175:               sess.close();
176:             } catch (HibernateException e1) { }
177:             e.printStackTrace();
178:         }
179:     }
180: }
```

Listagem 3.9 – Operação de Remoção com o Hibernate

PUC-Rio - Certificação Digital Nº 0410823/CA

Em primeiro lugar é preciso recuperar o objeto que será removido (linha 168). Depois, na linha 169, é chamado o método delete do objeto Session, sinalizando que o objeto recém recuperado deve ser removido. Quando a sessão é encerrada (linha 171) o objeto é efetivamente removido da base de dados.

Esta seção exemplificou como realizar operações de inserção, consulta, atualização e remoção de entidades de negócios com o framework ORM Hibernate. Nas próximas seções é mostrado como são tratadas outras questões decorrentes do conflito de paradigmas OO e relacional.

3.4.1.2. Hibernate e a Questão dos Subtipos

Como catalogado em Ambler (2002), há três maneiras de representar uma hierarquia de herança em uma linguagem orientada a objetos: usando uma tabela para cada classe concreta, uma tabela por hierarquia de classes e uma tabela por subclasse. O diagrama de classes da Figura 3.3 exemplifica uma hierarquia de herança com uma superclasse User e duas subclasses, Administrator e Learner.

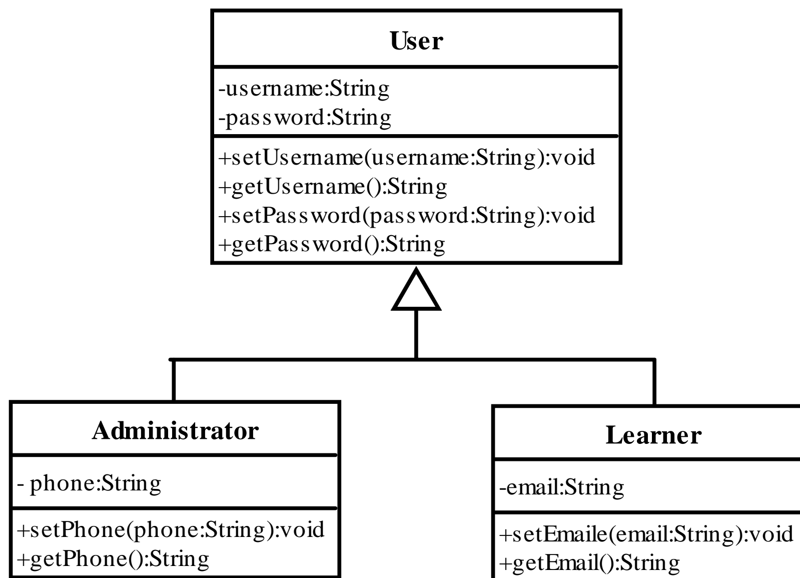

Figura 3.3 – Diagrama de Classes com Herança.

Ao aplicar o mapeamento usando uma tabela para cada classe concreta, são geradas duas tabelas: a tabela ADMINISTRATOR, com as colunas USERNAME, PASSWORD e PHONE além da tabela LEARNER, com as colunas USERNAME, PASSWORD e EMAIL. Os principais problemas desta abordagem são que ela não oferece suporte a consultas com polimorfismo e a dificuldade em manter o esquema das tabelas no banco de dados, já que os atributos da superclasse são replicados em cada tabela.

O mapeamento usando uma tabela por hierarquia de classes resulta em uma tabela com todas as propriedades em todas as classes e um identificador que determina o tipo da subclasse a qual a linha da tabela se refere. No exemplo da Figura 3.3, uma tabela seria criada com as propriedades USERNAME, PASSWORD, PHONE, EMAIL E USER_TYPE. A propriedade USER_TYPE é usada para discriminar a qual subclasse cada linha pertence. Esta abordagem possibilita o uso de consultas com polimorfismo e facilita a manutenção, contudo há um problema: os atributos referentes às subclasses não podem ser declarados como não nulos já que a mesma tabela armazena as instâncias das duas subclasses.

O mapeamento usando uma tabela por subclasse resulta que cada classe, abstrata ou concreta, seja mapeada em uma tabela. A ligação entre as superclasses e as subclasses é feita através de relacionamentos de chave estrangeira. O mapeamento do modelo descrito na Figura 3.3 resultaria em 3 tabelas: a tabela

PUC-Rio - Certificação Digital Nº 0410823/CA

USER, com as propriedades USER_ID, USERNAME e PASSWORD; a tabela ADMINISTRATOR, com as propriedades PHONE e USER_ID, que é chave estrangeira para a tabela USER; a tabela LEARNER, com as propriedades EMAIL e USER_ID, também chave estrangeira. A principal vantagem desta técnica é que ela resulta num modelo relacional normalizado ao mesmo tempo em que possibilita o uso de consultas com polimorfismo. Contudo, é uma técnica mais complexa do que as duas outras anteriormente citadas.

O Hibernate possibilita o uso das três técnicas, conforme a necessidade da aplicação. A técnica “uma tabela para cada classe concreta” pode ser aplicada naturalmente, sem que seja preciso qualquer configuração especial nos arquivos de mapeamento do Hibernate. A técnica “uma tabela por hierarquia de classes” é aplicada usando o elemento <subclass> no arquivo de mapeamento enquanto que a técnica “uma tabela por subclasse” é aplicada usando elemento <joined-subclass>.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.1.3. Hibernate e a Questão da Igualdade

Conforme visto na seção 3.3.1, há duas maneiras de representar igualdade em Java: através do conceito de identidade e de equivalência enquanto que no paradigma relacional a igualdade é determinada através de chaves primárias.

A identidade de objetos no Hibernate é determinada pelo valor do campo identificador, representado pelo elemento <id> no arquivo de mapeamento, mapeado para uma chave primária de uma tabela. Para checar se dois objetos representam a mesma entidade, pode-se consultar o método de acesso do identificador na classe de negócio (getId() no caso das classes Message e User do exemplo da sessão 3.4.1.1.) ou utilizar o método Session.getIdentifier(Object o).

Os identificadores podem ser naturais, com algum significado para o modelo de negócios, ou *surrogates*, sem significado para a lógica de negócios. O Hibernate fornece várias estratégias usadas para gerar identificadores *surrogates*, entre eles a estratégia *sequence*, que utiliza seqüências, *incremente*, que utiliza campos auto incrementáveis e *uuid.hex*, que gera um identificador único, mesmo quando ocorre processamento paralelo em várias máquinas.

3.4.1.4.
Hibernate e as Questões dos Relacionamentos

O Hibernate possibilita o uso de relacionamentos unidirecionais ou bidirecionais, com cardinalidade um-para-um, um-para-muitos ou muito-para-muitos. Há várias formas de mapear relacionamentos com o Hibernate e descrever todas seria excessivamente complexo e fugiria ao escopo desta dissertação. Contudo, o exemplo da seção 3.4.1.1. exemplifica o uso de relacionamentos direcionais um-para-um.

O Hibernate impõe uma limitação em classes com relacionamentos com cardinalidade de muitos: a variável do relacionamento deve ser determinada em função da interface da coleção em vez da classe concreta. Por exemplo, a interface java.util.List deve ser usada em vez da classe java.util.LinkedList. O Hibernate usa sua própria implementação de coleções, que oferece recursos como, por exemplo, busca tardia, que é vista na próxima seção. Esta limitação, contudo não é um problema já que programar para interfaces é uma conhecida boa prática de programação (Bloch, 2002).

3.4.1.5.
Hibernate e a Questão do Grafo de Navegação

Uma das principais questões em ORM é fornecer acesso eficiente a uma base de dados relacional através de um grafo de objetos (King & Bauer, 2005). Conforme visto na seção 3.3.1., é importante determinar a porção do grafo de objetos recuperada em cada consulta. Para lidar com esta questão, Hibernate define quatro estratégias possíveis de busca que podem ser usadas em quaisquer relacionamentos: busca imediata (*immediate fetching*), busca tardia (*lazy fetching*), busca antecipada (*eager fetching*) e busca em lote (*batch fetching*).

Com a estratégia da busca imediata, logo que uma entidade é recuperada da base de dados a entidade do relacionamento é recuperada através de uma consulta à base de dados ou então ao cache de entidades. Esta estratégia não costuma ser eficiente a não ser que as entidades relacionadas estejam quase sempre no cache.

A estratégia de busca tardia possibilita que a entidade relacionada seja recuperada sob demanda, apenas quando for necessário consultá-la. A busca tardia é um conceito fundamental para que uma performance adequada seja alcançada.

PUC-Rio - Certificação Digital Nº 0410823/CA

Com a estratégia de busca antecipada, as entidades relacionadas são recuperadas em uma mesma consulta através do uso do comando SQL OUTER JOIN. Otimizações em uma aplicação que usa Hibernate geralmente envolvem a configuração de relacionamentos, escolhendo a estratégia de busca antecipada para classes que quase sempre são usadas em conjunto.

A estratégia de busca em lote é uma técnica que pode aumentar a performance de relacionamentos com a estratégia de busca tardia ou imediata. Usualmente, ao recuperar um objeto da base de dados, uma consulta SQL é realizada com uma cláusula WHERE, especificando o identificador do objeto recuperado. Se a estratégia da busca em lote for usada, sempre que uma consulta for realizada, o Hibernate procura por outros objetos na mesma unidade de trabalho da sessão recuperáveis usando a mesma consulta, mas com valores múltiplos para a cláusula WHERE. Quase sempre a estratégia de busca tardia é mais eficiente que esta estratégia, porém a busca em lote é mais adequada para usuários do Hibernate que não desejam ou não podem usar seu tempo para ajustar a aplicação com uma combinação de busca tardia e antecipada.

Como visto nesta seção, o Hibernate fornece os meios para a resolução da questão do grafo de navegação, porém cabe ao desenvolvedor configurar os atributos dos relacionamentos na aplicação de forma a obter uma performance satisfatória.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.1.6. Considerações Finais Sobre o Hibernate

Hibernate possibilita o uso de persistência de POJOs de uma forma similar aos beans de entidades do EJB. Hibernate oferece vantagens sobre EJB. Hibernate é uma solução de mapeamento completo enquanto que os beans de entidade de EJB oferecem mapeamento médio (Fussel,1997). Por outro lado, EJB oferece outros serviços de infra-estrutura como, por exemplo, transações gerenciadas pelo container. Na seção 3.5 é visto como o Hibernate aliado ao Spring oferece também estes serviços.

Há outros frameworks ORM além do Hibernate disponíveis no mercado. Alguns gratuitos, outros comercias. A próxima seção apresenta um apanhado

destes frameworks, justificando a escolha do Hibernate para a arquitetura do AulaNet 3.0.

3.4.2. Outros Frameworks ORM

Segundo o artigo *Object Relational tool comparision* disponível em http://c2.com/cgi/wiki?ObjectRelationalToolComparison há cerca de 29 frameworks ORM disponíveis para Java atualmente, 14 deles são comerciais enquanto que 15 são gratuitos. Como o AulaNet é distribuído gratuitamente, os frameworks ORM comerciais são imediatamente descartados.

Dentre os 15 frameworks ORM gratuitos, 7 exigem que as classes persistidas estendam alguma classe ou implemente alguma interface, resultando em um forte acoplamento entre as classes do modelo e o framework, por isto também são descartados. Entre os restantes, apenas o Hibernate (2005), iBATIS (2005), Apache OJB (OJB, 2005) e JDOMax (2005), uma implementação da especificação JDO (2005) são soluções de mapeamento completo.

Para confirmar a escolha do Hibernate em face destas outras opções é realizada uma pesquisa, considerando aspectos não técnicos. Raible (2005) realiza uma comparação entre web frameworks, apresentada no capítulo 4, onde são considerados os seguintes itens: quantidade de documentação disponível, disponibilidade de suporte, disponibilidade de ferramentas compatíveis, grau de aceitação no mercado e disponibilidade de profissionais aptos a trabalhar com os frameworks. Nesta seção é feita uma comparação similar, porém entre frameworks ORM.

3.4.2.1. Quantidade de Documentação Disponível

Há duas formas de contabilizar a disponibilidade de documentação acessível sobre uma tecnologia: verificando o número de livros publicados sobre a tecnologia ou verificando o número de tutoriais disponíveis na web. O gráfico da Figura 3.4 mostra o número de livros publicados para cada framework ORM segundo uma pesquisa realizada em novembro de 2005 no site da Amazon.

PUC-Rio - Certificação Digital Nº 0410823/CA

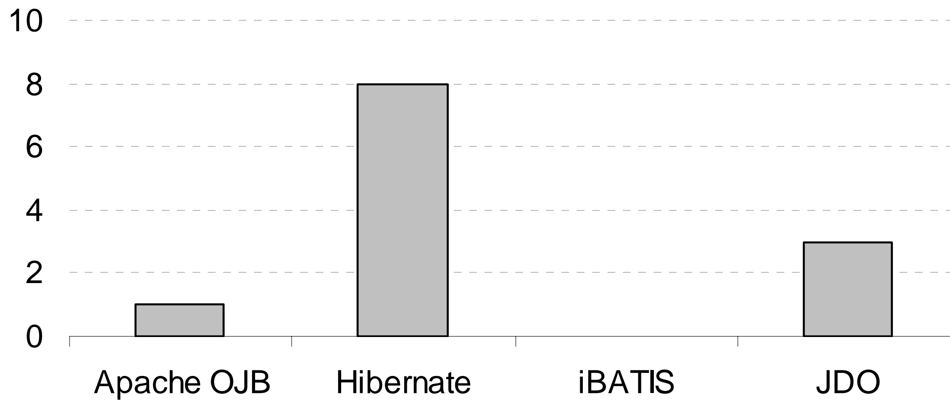

Figura 3.4 – Livros Publicados Para Cada Framework ORM
Fonte: Amazon.com, Novembro de 2005

Para contabilizar o número de tutoriais disponíveis na web sobre cada framework ORM, é feita uma consulta na ferramenta de busca Google. O resultado da pesquisa realizada em março de 2006 é apresentado no gráfico da Figura 3.5.

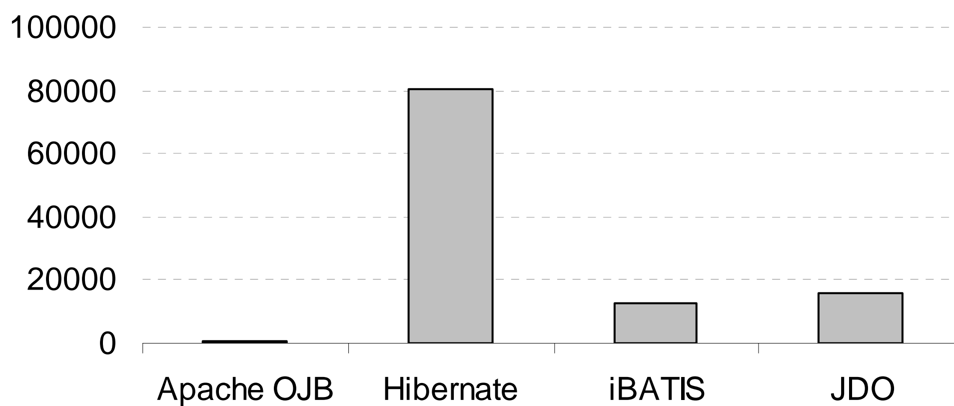

Figura 3.5 – Artigos e Tutoriais Para Cada Framework ORM
Fonte: Google.com, Março de 2006

Há uma quantidade significativamente maior de documentação disponível para o Hibernate, tanto na forma de livros quanto na forma de tutoriais e artigos com o JDO ocupando a segunda colocação.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.2.2. Disponibilidade de Suporte

É importante que o framework ORM escolhido disponha de suporte para resolver questões não tratadas pela documentação disponível. Os frameworks ORM Apache OJB e iBATIS possuem comunidades ativas que se comunicam através de listas de discussão, trocando informações e provendo suporte gratuito. Para mensurar a disponibilidade de suporte destes frameworks, realiza-se uma pesquisa contabilizando o volume de mensagem trocado nestas listas de discussão em determinado período de tempo.

O Hibernate também possui comunidade ativa, mas a troca de mensagem é feita por fórum. Como o fórum não fornece nenhum mecanismo para selecionar as mensagens enviadas em um determinado período de tempo, para mensurar o volume de mensagens trocado foi realizada comunicação pessoal com os administradores do fórum através de e-mail. Estes, que possuem mais privilégios que os usuários comuns, forneceram o número de mensagens trocadas no período especificado. O apêndice A apresenta uma transcrição dos e-mails trocados com os administradores do fórum.

O framework JDOMax não possui nenhuma comunidade que ofereça suporte a ele e a única maneira de obter qualquer tipo de auxílio é entrando em contato diretamente com os desenvolvedores.

O gráfico da Figura 3.6 mostra o volume de mensagens trocadas nas listas de discussão oficiais das comunidades do Apache OJB e iBATIs e no fórum do Hibernate durante o mês de novembro de 2005.

PUC-Rio - Certificação Digital Nº 0410823/CA

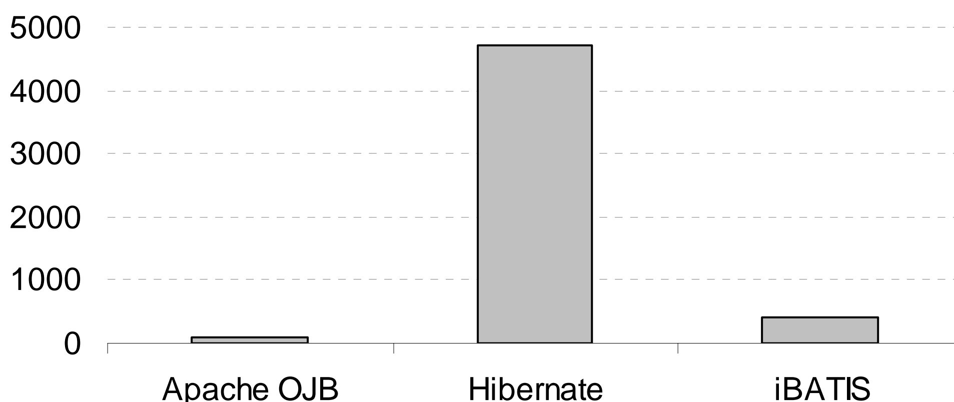

Figura 3.6 - Mensagens Trocadas em Listas de Discussão (11/2005)

A comunidade do Hibernate é a mais ativa. Vale ressaltar que o Hibernate é desenvolvido pelo mesmo grupo responsável pelo desenvolvimento do servidor de aplicações JBoss (2005), que também oferece suporte e treinamento comercial.

3.4.2.3. Disponibilidade de Ferramentas Compatíveis

Ferramentas compatíveis com um framework ORM auxiliam o desenvolvimento. A ferramenta pode fornecer editores gráficos, automatizar tarefas repetitivas, etc. Para contabilizar o número de ferramentas compatíveis com cada framework ORM foi verificada a quantidade de plugins disponíveis para as plataformas Eclipse e NetBeans, a compatibilidade com as IDEs Java mais populares (IBM WSAD, Sun Java Studio Enterprise, Borland JBuilder, Oracle JDeveloper, BEA Workshop e InteliJ IDEA), a compatibilidade com as ferramentas de produtividade Middlegen e XDoclets além das ferramentas disponíveis nos web sites dos próprios frameworks.

A Figura 3.7 mostra o número de ferramentas disponíveis compatíveis com cada framework ORM, conforme pesquisa realizada em novembro de 2005.

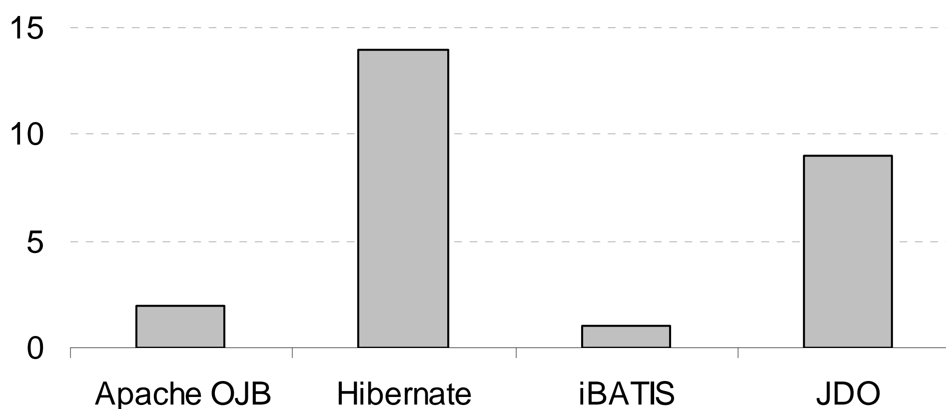

Figura 3.7 – Ferramentas Compatíveis com os Frameworks ORM em 11/2005

A pesquisa realizada em novembro de 2005 demonstra que o Hibernate é o framework ORM que possui mais ferramentas compatíveis, com o JDO em segundo lugar.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.2.4. Grau de Aceitação no Mercado

Ao selecionar um framework ORM é necessário realizar uma sondagem no mercado para saber qual tem sido mais requisitado por empresas. Para obter um indicativo do grau de aceitação no mercado para um framework, faz-se uma busca em sites especializados por ofertas de emprego abertas que solicitem desenvolvedores com habilidades em um determinado framework ORM. A Figura 3.8 mostra o resultado da pesquisa realizado no site Dice.com em dezembro de 2005.

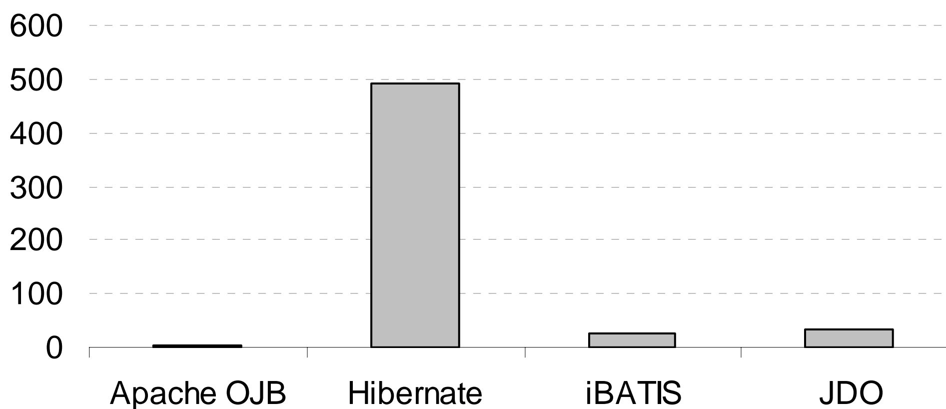

Figura 3.8 – Ofertas de Emprego Abertas para cada Framework ORM
Fonte: Dice.com, Dezembro de 2005

Segundo a pesquisa o Hibernate é o framework ORM mais solicitado em ofertas de emprego com o JDO em segundo, com uma pequena vantagem sobre o iBATIS em terceiro lugar. Este resultado reflete o estado do mercado norte-americano. Para tomar um indicativo do grau de aceitação dos frameworks ORM no mercado brasileiro apresenta-se uma pesquisa realizada em dezembro de 2005 no site nacional de empregos Manager Online (http://www.manageronline.com.br/).

PUC-Rio - Certificação Digital Nº 0410823/CA

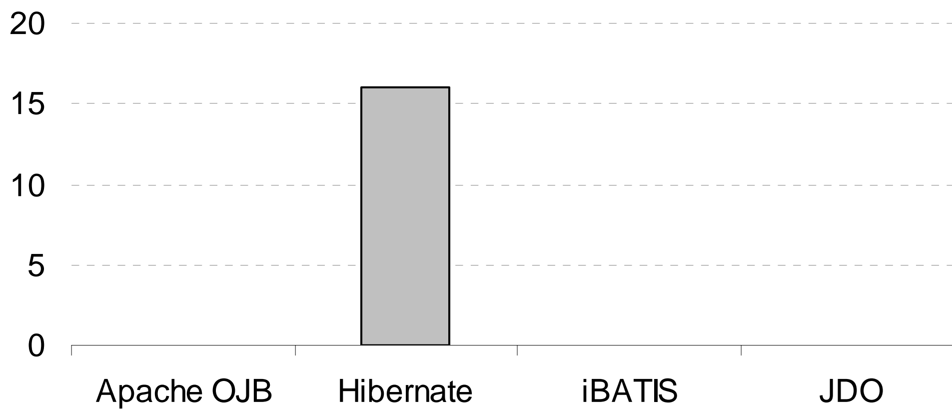

Figura 3.9 – Ofertas de Emprego Abertas para cada Framework ORM
Fonte: Manager Online, Dezembro de 2005

A pesquisa realizada em dezembro de 2005 revela que há pouco mais que 15 vagas abertas requisitando habilidade com o Hibernate e que não há vagas abertas requisitando habilidade com OJB, iBATIS e JDO.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.2.5. Disponibilidade de Profissionais

É preciso considerar a disponibilidade de profissionais habilitados no mercado para trabalhar com o framework ORM antes de efetuar uma escolha. A escolha de um framework com poucos profissionais aptos disponíveis no mercado implica em custos de treinamento mais elevados. Para verificar esta disponibilidade faz-se uma consulta em sites que hospedam currículos. A Figura 3.10 mostra o resultado da pesquisa realizada em dezembro de 2005 no site Jobs.net que reflete a realidade do mercado de trabalho norte-americano.

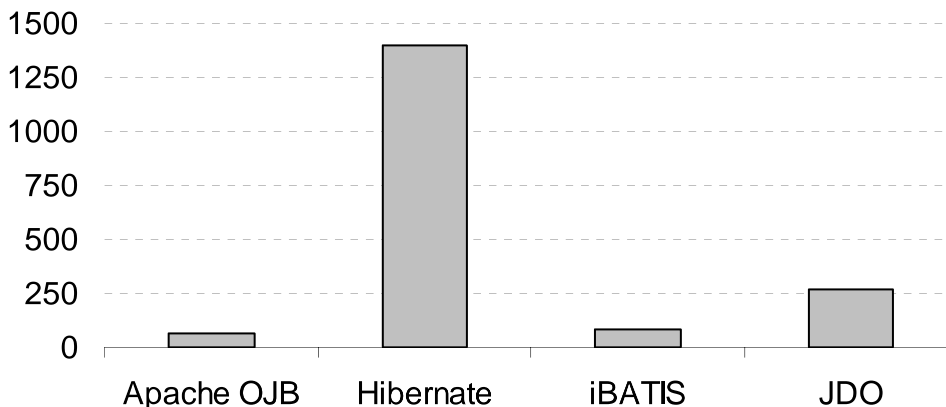

Figura 3.10 – Disponibilidade de Profissionais

Fonte: Jobs.net, Dezembro de 2005

O Hibernate é o framework ORM com mais profissionais aptos no mercado norte-americano. A Figura 3.11 reflete a realidade do mercado brasileiro, obtido através de uma busca no site brasileiro APInfo.com, realizada em dezembro de 2005.

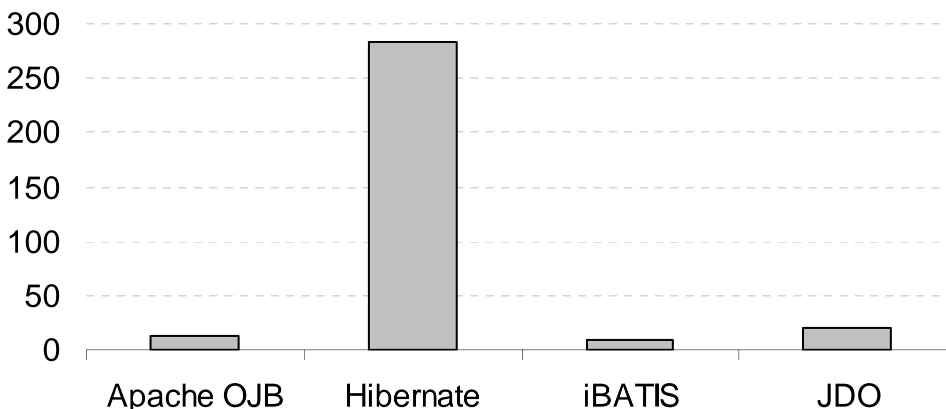

Figura 3.11 - Disponibilidade de Profissionais

Fonte: APInfo.com, Dezembro de 2005

O mercado nacional segue o mesmo padrão do mercado norte-americano com o Hibernate sendo o framework ORM com mais profissionais habilitados disponíveis, seguido do JDO.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.4.2.6. Resultado da Comparação entre Frameworks ORM

A análise mostra que o Hibernate é o framework ORM com mais documentação disponível, incluindo livros publicados e tutoriais disponíveis na web, com mais ferramentas compatíveis e com a comunidade mais ativa. Além disso, a análise mostrou que o Hibernate é o framerwork ORM mais presente tanto em ofertas de empregos quanto nos currículos de candidatos, nos mercados norte-americano e brasileiro. Com isto em vista e dado que o Hibernate atende às necessidades de persistência de dados do AulaNet, ele é incorporado à arquitetura do AulaNet 3.0.

Enterprise JavaBeans não é incluído nesta análise por dois motivos: porque ele não é uma solução de mapeamento completo (Fussel,1997) e porque os critérios técnicos já o haviam descartado. Vale lembrar que o EJB também fornece outros serviços de infra-estrutura como, por exemplo, segurança, transações declarativas e exposição de serviços remotos. Na próxima seção é visto como o Spring, incorporado à arquitetura do AulaNet 3.0, complementa o Hibernate fornecendo estes serviços.

3.5. O Framework de Infra-Estrutura Spring

Spring é um framework de infra-estrutura adotado na arquitetura do AulaNet 3.0 para complementar o Hibernate. Este Spring oferece os serviços de controle de transações, segurança e exposição de serviços remotos. O Spring também simplifica o desenvolvimento com o Hibernate através do uso de métodos template e contribui para um baixo acoplamento entre as classes da aplicação.

O Spring pode ser classificado como um framework de infra-estrutura, pois provê serviços como gerencia de transações e de segurança além de se integrar com frameworks ORM, que fornecem serviços de persistência. Spring também é usualmente chamado de container leve ou "*lightweight container*" (Walls & Breidenbach, 2005). O termo container é usado, pois Spring gerencia o ciclo de vida dos objetos configurados nele, assim como o container EJB. Porém é considerado um container leve, pois ao contrário do container EJB, ele não é

PUC-Rio - Certificação Digital Nº 0410823/CA

intrusivo e as classes da aplicação tipicamente não possuem dependências com o Spring.

Spring é um framework dividido em vários módulos usados de acordo com as necessidades da aplicação. Através do seu módulo de *Dependency Injection* (Fowler, 2004), consegue-se um baixo grau de acoplamento entre as classes da aplicação e através do módulo de programação orientada a aspectos (*Aspect Oriented Programming* – AOP) (Jacobson & Ng, 2004), pode-se usar AOP sem que seja necessário o aprendizado de uma nova linguagem ou a incorporação de uma nova ferramenta ao projeto. Por fim, o módulo de integração ORM possibilita a integração do Hibernate ao Spring.

A subseção a seguir descreve *Dependency Injection*, o principal conceito por trás do Spring.

3.5.1. Dependency Injection

O termo *Dependency Injection* (Fowler, 2004) é usado para classificar um tipo específico de inversão de controle onde a responsabilidade de configurar dependências entre classes é assumida por um terceiro objeto. Desta forma, atinge-se um acoplamento mais fraco entre as classes e por conseqüência maior reuso. A Figura 3.12 mostra um diagrama de classes com a classe ConferenceFacade, representando o *Façade* (Gamma et al., 1995) dos serviços da Conferência e a classe MessageDAOImpl e sua interface, que implementam o DAO (Alur et al., 2001) responsável por efetuar as operações de persistência na base de dados.

PUC-Rio - Certificação Digital Nº 0410823/CA

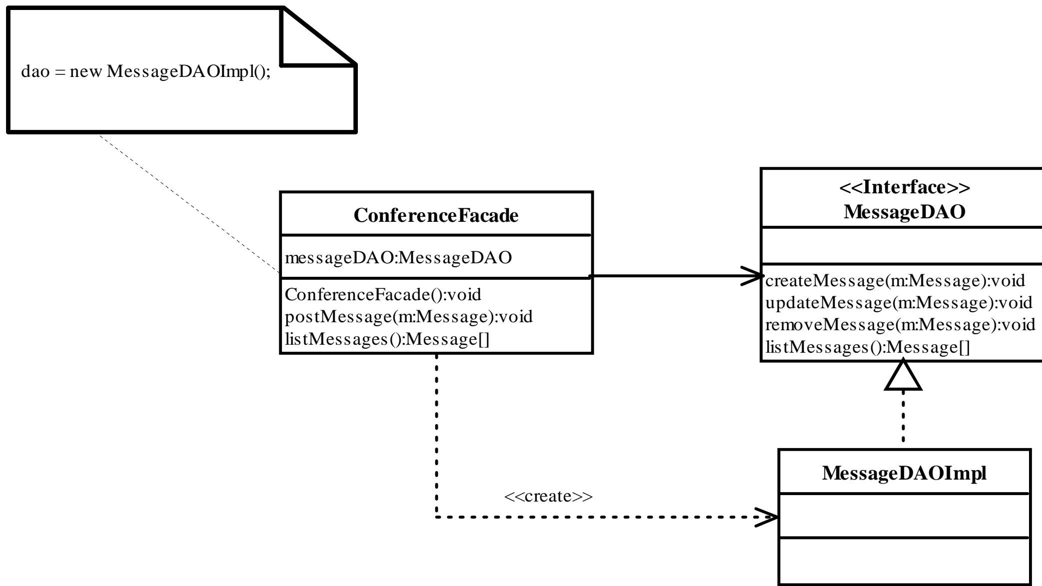

Figura 3.12 – Dependências Configuradas sem *Dependency Injection*

Apesar da classe ConferenceFacade referenciar a interface MessageDAO, ela é dependente da implementação da interface pois precisa instanciá-la para poder usá-la. Através de *Dependency Injection*, um terceiro objeto denominado Montador (*Assembler*) é inserido. Este objeto é responsável por instanciar e configurar as dependências das classes relacionadas. A Figura 3.13 exemplifica o mesmo modelo da Figura 3.12, mas desta vez usando *Dependency Injection*.

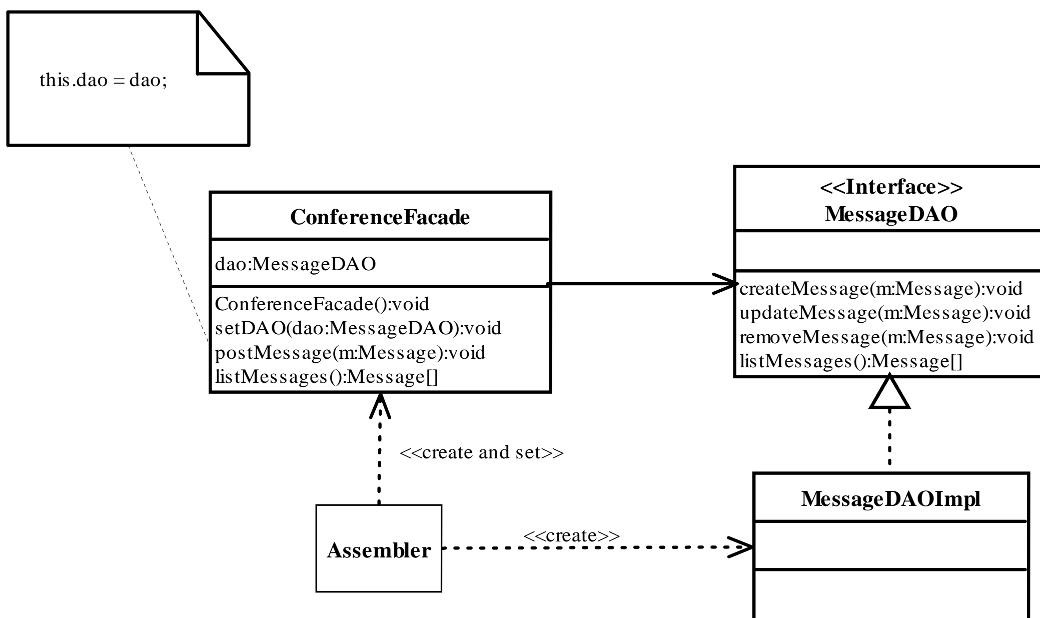

Figura 3.13 - Dependências Configuradas com *Dependency Injection*

PUC-Rio - Certificação Digital Nº 0410823/CA

Com a introdução do objeto Montador a classe ConferenceFacade passa a depender apenas da interface MessageDAO. O montador é responsável por instanciar a implementação adequada da interface e configurar a dependência da classe ConferenceFacade.

Spring e frameworks similares exercem a função do montador no *Dependency Injection* e como as classes relacionadas dependem apenas das interfaces, o Spring pode passar *Decorators* ou *Proxys* (Gamma et al., 1995) acrescentando funcionalidades de infra-estrutura.

3.5.2. Dependency Injection com Spring

Para melhor compreensão do Spring esta seção apresenta um exemplo que ilustra o funcionamento do seu módulo de *Dependency Injection*. Neste exemplo é implementado o modelo de classes apresentado na Figura 3.14. Nas seções seguintes, este exemplo será incrementado através da ilustração do uso de serviços de infra-estrutura com o Spring.

PUC-Rio - Certificação Digital Nº 0410823/CA

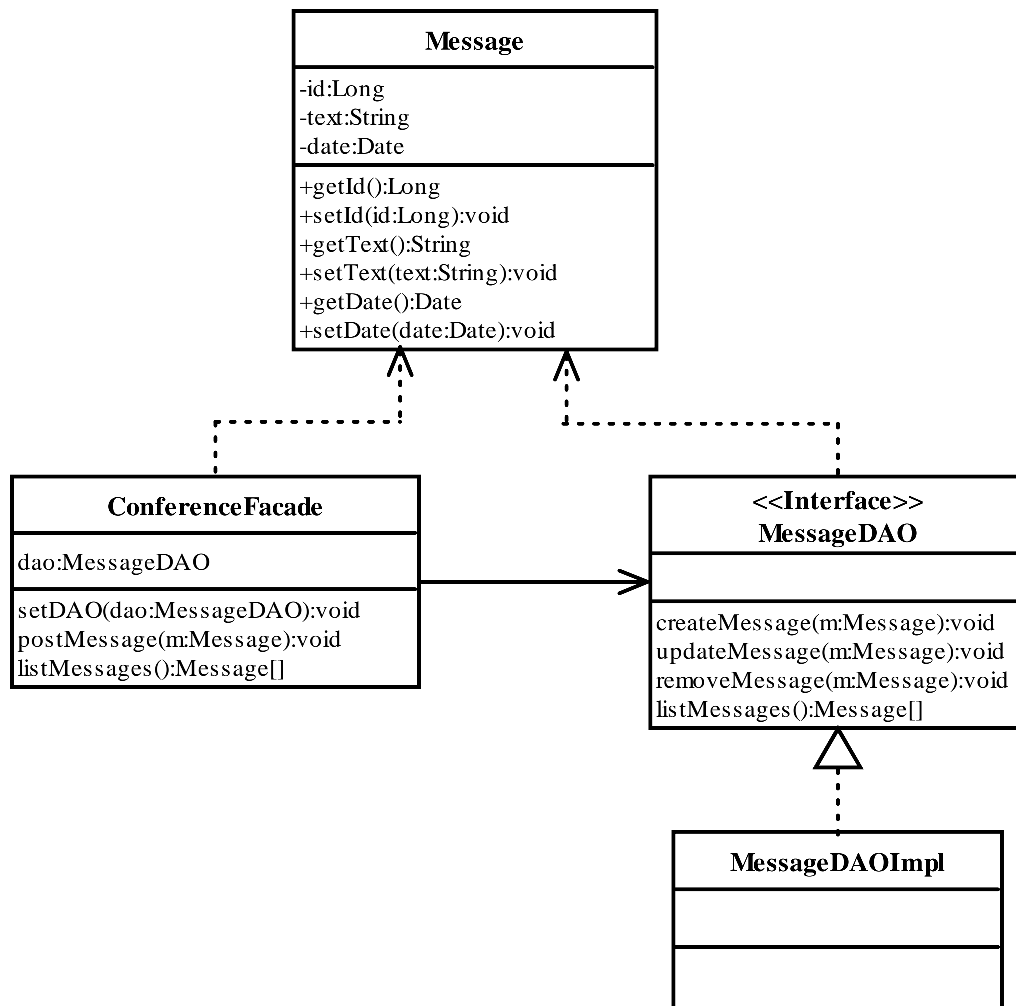

Figura 3.14 – Modelo de Classes do Exemplo do Spring

O modelo da Figura 3.14 corresponde ao modelo da Figura 3.13, com o Spring funcionando como o montador. A classe Message também é acrescentada ao modelo para tornar o exemplo mais ilustrativo. O exemplo desta seção se limita a mostrar o funcionamento básico do módulo de *Dependency Injection* do Spring. Nas próximas seções este exemplo é incrementado para exemplificar as outras funcionalidades de infra-estrutura que o Spring provê. A Listagem 3.1 apresenta a implementação da classe Message.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
01: public class Message {

02:     private Long id;
03:     private String text;
04:     private Date date;

05:     public void setId(Long newValue) { id = newValue; }
06:     public Long getId() { return id; }

07:     public void setText(String newValue) { text = newValue; }
08:     public String getText() { return text; }

09:     public void setDate(Date newValue) { date = newValue; }
10:     public Date getDate() { return date; }

11: }
```

Listagem 3.10 – Classe Message.

Como pode ser visto, a classe Message é implementada normalmente sem nenhuma dependência com o Spring. As propriedades da classe são definidas entre as linhas 02 e 04 e os métodos de acesso são definidos entre as linhas 05 e 10. A Listagem 3.11 apresenta o código fonte da interface MessageDAO.

```
12: public interface MessageDAO {

13:     public void createMessage(Message m);
14:     public void updateMessage(Message m);
15:     public void removeMessage(Message m);
16:     public Message[] listMessages();

17: }
```

Listagem 3.11 – Interface MessageDAO.

A interface MessageDAO também é construída se qualquer acoplamento com o Spring. A implementação da classe MessageDAOImpl, que implementa a interface MessageDAO, é apresentada na seção 3.5.4 onde é apresentada a integração do Hibernate ao Spring. A listagem abaixo apresenta o código da classe ConferenceFacade.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
18: public class ConferenceFacade {

19:     private MessageDAO messageDao;

20:     public void setMessageDao(MessageDAO dao) { messageDao = dao; }

21:     public void postMessage(Message m) {
22:         messageDao.createMessage(m);
23:     }

24:     public Message listMessages() {
25:         return messageDao.listMessages();
26:     }
27: }
```

Listagem 3.12 – Classe ConferenceFacade.

A variável que armazena a dependência com a interface MessageDAO é declarada na linha 19. A linha 20 explicita o método de acesso usado para configurar a dependência. O método de envio de mensagens é declarado no trecho entre a linha 21 e a linha 23 enquanto que o método que lista as mensagens da conferência é declarado entre as linhas 24 e 26. Os métodos de negócio nos *Façades* apenas delegam as chamadas recebidas para o DAO configurado. Um exemplo mais realista envolveria várias chamadas de métodos para um ou mais DAOs.

A dependência entre a classe ConferenceMessage e a interface MessageDAO deve ser configurada no arquivo de configuração do Spring para que este atue como o montador. A Listagem 3.13 apresenta este arquivo de configuração do Spring, que por padrão tem o nome applicationContext.xml.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
28: <?xml version="1.0" encoding="UTF-8"?>
29:  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
30:        "http://www.springframework.org/dtd/spring-beans.dtd">
31: <beans>

32:    <bean id="messageDAO"
33:            class="MessageDAOImpl">
34:    </bean>

35:    <bean id="conferenceFacade"
36:            class="ConferenceFacade">
37:        <property name="messageDao">
38:            <ref local="messageDAO"/>
39:        </property>
40:    </bean>

41: </beans>
```

Listagem 3.13 – Arquivo de Configuração do Spring

O trecho entre as linhas 32 e 34 define o bean messageDAO. O *id*, definido na linha 32, identifica unicamente este bean no contexto da aplicação Spring. A propriedade *class*, definida na linha 33, especifica a classe do bean.

O trecho entre as linhas 35 e 40 declaram o bean conferenceFacade. A dependência com o bean messageDAO é configurada no trecho entre as linhas 37 e 39. Na linha 37 é descrito que a propriedade que armazena a dependência se chama messageDao e na linha 38 é descrito que esta propriedade recebe o bean messageDAO, definido anteriormente no mesmo arquivo.

Os beans definidos no arquivo de configuração do Spring ficam acessíveis pela interface org.springframework.context.ApplicationContext. Há varias formas de iniciar o contexto do Spring. A mais comum em aplicações web como o AulaNet é através de um tratador de eventos executado automaticamente quando a aplicação é iniciada no servidor de aplicações. Para configurar este tratador de eventos, é preciso configurar o arquivo descritor da aplicação web.xml conforme o exemplo da Listagem 3.14.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
42: <?xml version="1.0" encoding="UTF-8"?>
43: <!DOCTYPE web-app PUBLIC
44:     "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
45:     "http://java.sun.com/dtd/web-app_2_3.dtd">

46: <web-app>
47:     <context-param>
48:         <param-name>contextConfigLocation</param-name>
49:         <param-value>/WEB-INF/applicationContext.xml</param-value>
50:     </context-param>
51:     <listener>
52:         <listener-class>
53:             org.springframework.web.context.ContextLoaderListener
54:         </listener-class>
55:     </listener>
56: </web-app>
```

Listagem 3.14 – Descritor web.xml da Aplicação: Iniciando o Contexto Spring

Na linha 49 o nome do arquivo de configuração do Spring é configurado. Neste exemplo optou-se pelo nome padrão, mas qualquer nome poderia ser usado ou mesmo uma lista separada por vírgulas com vários arquivos de configuração. O trecho entre as linhas 51 e 55 declara o tratador de eventos que inicia o contexto Spring.

Além de promover o acoplamento fraco entre classes através do módulo de *Dependency Injection,* o Spring também fornece serviços de infra-estrutura. A próxima seção mostra como o Spring integrado ao Hibernate provê serviços relacionados à persistência dos dados.

3.5.3. Integração do Hibernate com o Spring

Ao integrar o Hibernate ao Spring, a configuração do Hibernate que antes era feita separadamente passa a ser feita através do próprio Spring. Desta forma, a manutenção do arquivo de configuração da aplicação é feita em um só arquivo com uma sintaxe única. A Listagem 3.15 apresenta o arquivo de configuração do Spring onde é configurada a integração com o Hibernate.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
01: <?xml version="1.0" encoding="UTF-8"?>
02:  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
03:        "http://www.springframework.org/dtd/spring-beans.dtd">
04: <beans>
05:     <bean id="dataSource"
06:               class="org.springframework.jndi.JndiObjectFactoryBean">
07:         <property name="jndiName">
08:             <value>java:/comp/env/jdbc/AulaNetDB</value>
09:         </property>
10:     </bean>
11:     <bean id="sessionFactory"
12:              class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
13:         <property name="dataSource" ref="dataSource"/>
14:         <property name="mappingResources">
15:             <list>
16:                 <value>Message.hbm.xml</value>
17:             </list>
18:         </property>
19:         <property name="hibernateProperties">
20:             <props>
21:                 <prop key="hibernate.dialect">
22:                     net.sf.hibernate.dialect.MySQLDialect
23:                 </prop>
24:                 <prop key="hibernate.show_sql">false</prop>
25:             </props>
26:         </property>
27:     </bean>
28:     <bean id="messageDAO" class="MessageDAOImpl">
29:          <property name="sessionFactory" ref="sessionFactory">
30:     </bean>
31:     <bean id="conferenceFacade" class="ConferenceFacade">
32:         <property name="messageDao" ref="messageDAO"/>
33:     </bean>
34: </beans>
```

Listagem 3.15 – Arquivo de Configuração do Spring: Integração com Hibernate.

PUC-Rio - Certificação Digital Nº 0410823/CA

O trecho entre as linhas 05 e 10 define a base de dados acessada pelo Hibernate para persistir os dados. A fábrica de sessões do Hibernate é configurada no trecho de código entre as linhas 11 e 27. Na linha 13 a fábrica é configurada com a fonte de dados previamente configurada. Na linhas 16 a fábrica é configurada para reconhecer os arquivos de mapeamento do Hibernate para a classe Message. O dialeto é configurada na linha 22 e na linha 24 a opção de log dos comandos SQL gerados é desabilitada. O bean messageDAO é configurado no trecho entre as linhas 28 e 30. Na linha 29 ele é configurado com a fábrica de sessões do Hibernate.

A classe org.springframework.orm.hibernate.support.HibernateDaoSupport do Spring provê suporte para a implementação de DAOs usando Hibernate. Esta classe gerencia o acesso do DAO à fábrica de sessões e possibilita a criação de instâncias da classe org.springframework.orm.hibernate.HibernateTemplate, que fornece vários templates que facilitam a execução das tarefas mais usuais com o Hibernate. A Listagem 3.16 exibe a implementação da classe MessageDAOImpl.

```
35: public class MessageDAOImpl extends HibernateDaoSupport
36:         implements MessageDAO {

37:     public void createMessage(Message m) {
38:         getHibernateTemplate().save(m);
39:     }

40:     public void updateMessage(Message m) {
41:         getHibernateTemplate().saveOrUpdateCopy(m);
42:     }

43:     public void removeMessage(Message m) {
44:        getHibernateTemplate().delete(category);
45:     }

46:     public Message[] listMessages() {
47:         List messages = getHibernateTemplate().find("from Message m");
48:     }
49: }
```

Listagem 3.16 – Classe MessageDAOImpl.

Na linha 35 a classe é declarada estendendo a classe HibernateDaoSupport e na linha 36 é declarado que a classe implementa a interface do MessageDAO. O método getHibernateTemplate retorna instâncias da classe HibernateTemplate sobre a qual as operações de criação (linha 38), atualização (linha 41), remoção (linha 44) e busca (linha 47) são executadas.

O código resultante é consideravelmente menor do que o listado na seção 3.4.1.1., pois a classe HibernateTemplate do Spring se responsabiliza por abrir e fechar sessões além de efetuar o tratamento de erros, convertendo possíveis exceções em outras que não exigem tratamento. O código também não apresenta marcação de transações, que são feitas declarativamente e são apresentadas na próxima seção.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.5.4. Transações Declarativas com Spring

Spring possibilita o uso de transações declarativas que são descritas no arquivo de configuração sem que seja preciso alterar o código fonte. O controle das transações é realizado através de um gerenciador de transações. Há implementações do gerenciador para a API JDBC, para os frameworks ORM Hibernate, OJB, JDO, iBATIs e para a API JTA, que deve ser usada quando for preciso controlar transações entre bancos de dados distintos.

Configurado o gerenciador de transações, é preciso especificar a política de transação para os métodos das classes envolvidas. A política envolve um ou mais dos seguintes atributos: escopo de propagação da transação, nível de isolamento, dicas de somente leitura, tempo limite da transação e tolerância a erros.

O escopo de propagação da transação especifica o comportamento da transação quando um método que participa de uma transação chama outro. Spring possibilita que os seguintes valores sejam configurados para o escopo de propagação da transação: PROPAGATION_MANDATORY, uma exceção é lançada se um método marcado com este atributo é chamado fora de um contexto de transação; PROPAGATION_NESTED, um método marcado com este atributo deve sempre ser executado em um novo contexto de transação; PROPAGATION_NEVER, uma exceção é levantada se um método marcado com este atributo é chamado em um contexto de transação; PROPAGATION_NOT_SUPPORTED, indica que o método marcado com este atributo é executado fora de um contexto transacional sempre; PROPAGATION_REQUIRED, um método marcado com este atributo é sempre executado dentro de uma transação, ou seja, se este método é chamado dentro de uma transação, ele participa dela e no caso contrário é criada uma nova transação; PROPAGATION_REQUIRES_NEW indica que o método sempre é executado em seu próprio contexto de transação e PROPAGATION_SUPPORTS indica que um método só roda em um contexto de transação se ele for chamado dentro de um.

O nível de isolamento da transação especifica o comportamento entre transações concorrentes que podem levar a problemas de “leituras sujas”, quando uma transação lê um dado alterado por outra transação ainda não confirmada,

PUC-Rio - Certificação Digital Nº 0410823/CA

"leituras não repetitivas", quando uma transação realiza uma mesma consulta duas vezes e obtém resultados diferentes e "leituras fantasmas", quando uma transação realiza uma consulta uma vez, obtém um número determinado de linhas e repetindo a consulta obtém um número diferente de linhas. Quanto maior o nível de isolamento menor o desempenho da transação. Spring possibilita o uso de cinco níveis de isolamento: ISOLATION_DEFAULT é usado o nível de isolamento padrão da base de dados; ISOLATION_READ_UNCOMMITTED, que pode resultar em leituras sujas, não repetíveis e fantasmas. ISOLATION_READ_COMMITTED que evita leituras sujas; ISOLATION_REPEATEBLE_READ que evita leituras sujas e não repetíveis e ISOLATION_SERIALIZABLE que evita todos os problemas de concorrência listados anteriormente.

Dicas de somente leitura podem ser acrescentadas à política de transação possibilitando um melhor desempenho em consultas que recuperam dados apenas para leitura e o tempo limite da transação é usado para evitar que uma transação espere infinitamente para ser concluída. A tolerância a erros pode ser configurada de forma que a transação não seja abortada quando uma determinada exceção ocorrer. A Listagem 3.17 apresenta o arquivo de configuração onde o gerenciador de transação e os dados da política de transação são configurados.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
01: <?xml version="1.0" encoding="UTF-8"?>
02:  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
03:        "http://www.springframework.org/dtd/spring-beans.dtd">
04: <beans>

05:     <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
06:        <!--    Declaração da Fonte de Dados -->
07:     </bean>
08:     <bean id="sessionFactory"
09:              class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
10:        <!-- Declaração da fábrica de sessões -->
11:     </bean>

12:     <bean id="txManager"
13:             class="org.springframework.orm.hibernate.HibernateTransactionManager">
14:        <property name="sessionFactory" ref="sessionFactory"/>
15:     </bean>

16:     <bean id="messageDAOTarget"
17:             class="MessageDAOImpl">
18:         <property name="sessionFactory" ref="sessionFactory">
19:     </bean>

20:     <bean id="messageDAO"
21:             class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
22:        <property name="transactionManager" ref="txManager"/>
23:        <property name="target" ref="messageDAOTarget"/>
24:        <property name="proxyInterfaces">
25:           <value>MessageDAO</value>
26:         </property>
27:         <property name="transactionAttributes">
28:            <props>
29:               <prop key="list*">PROPAGATION_REQUIRED,readOnly</prop>
30:               <prop key="*">PROPAGATION_REQUIRED</prop>
31:            </props>
32:        </property>
33:     </bean>

34:     <bean id="conferenceFacade"
35:             class="ConferenceFacade">
36:        <property name="messageDao" ref="messageDAO"/>
37:     </bean>
38: </beans>
```

Listagem 3.17 - Arquivo de Configuração do Spring: Transações Declarativas.

O gerenciador de transações para o Hibernate é configurado no trecho entre as linhas 12 e 15. A classe MessageDAOImpl é configurada no trecho de código entre as linhas 16 e 19. A dependência da classe ConferenceFacade com a interface MessageDAO é configurada com um *Proxy* dinâmico gerado pelo Spring que acrescenta os serviços de transação e depois delega a chamada para o bean messageDAOTarget. Este *Proxy* é declarado entre as linhas 20 e 33.

Na linha 22 o *Proxy* é configurado com o gerenciador de transação apropriado e na linha 23, o bean messageDAOTarget para o qual o *Proxy* delega chamadas é configurado. Na linha 25 é declarado que o *Proxy* gerado pelo Spring dinamicamente deve implementar a interface MessageDAO, desta forma a classe

PUC-Rio - Certificação Digital Nº 0410823/CA

ConferenceFacade pode usar o *Proxy* da mesma forma que usaria a implementação concreta. Na linha 29 a política de transação PROPAGATION_REQUIRED com a dica de somente leitura para todos os métodos começando com “list” e na linha 30 a política de transação PROPAGATION_REQUIRED é especificada para os outros métodos.

O uso de transações declarativas resulta em um código mais limpo e com um maior grau de reuso já que classes podem ter seu comportamento transacional modificado sem que seja necessário mudar o código. Na próxima seção é apresentado outro serviço de infra-estrutura provido pelo Spring: gerenciamento de segurança.

3.5.5. Gerenciamento de Segurança com Spring

O módulo de programação orientada a aspectos do Spring possibilita que o desenvolvedor crie seus próprios aspectos, porém há vários aspectos que podem ser usados em situações corriqueiras. Dentre eles, há o aspecto que possibilita o uso de segurança declarativa onde se pode restringir o acesso aos métodos de uma classe a determinados usuários.

Para que o Spring possa prover segurança declarativa é preciso que sejam configurados um gerenciador de autenticação e um gerenciador de controle de acesso. O gerenciador de autenticação é responsável por autenticar que o usuário é quem diz ser. Tipicamente esta checagem é feita através de um par identificador – senha. O gerenciador de controle de acesso é responsável pela autorização, ou seja, ele decide se o usuário autenticado tem permissão para executar a operação.

O gerenciador de autenticação implementa a interface org.acegisecurity.AuthenticationManager. O Spring possui uma implementação desta interface chamada gerenciador de provedores (ProviderManager) cuja função é realizar autenticação em diversos “provedores de autenticação”. Há diversos tipos de provedores que podem ser usados para autenticar um usuário em uma base de dados, usando um serviço remoto de autenticação, um servidor LDAP, etc.

O gerenciador de controle de acesso é responsável por contabilizar os votos dos org.acegisecurity.vote.AccessDecisionVoter e tomar a decisão se o acesso é

PUC-Rio - Certificação Digital Nº 0410823/CA

garantido ou negado. Um AccesDecisionVoter é uma classe que vota se o acesso a um recurso é garantido, negado ou opcionalmente pode se abster da votação. Exemplo: o RoleVoter vota para garantir acesso sempre que o usuário possuir um papel com o qual ele está configurado. Há 3 implementações do gerenciador de controle de acesso disponíveis no Spring: o AffirmativeBased, que garante acesso se houver pelo menos um voto a favor, ConsensusBased que garante acesso apenas se todos votarem a favor e UnanimousBased que garante acesso se não houver votos contra.

A Listagem 3.18 mostra o arquivo de configuração do Spring modificado para prover segurança declarativa para a classe MessageDAOImpl.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
01: <beans>
02:     <!-- Declaração de Outros Beans -->
03:     <bean id="conferenceFacadeTarget"  class="ConferenceFacade">
04:         <property name="messageDao" ref="messageDAO"/>
05:     </bean>
06:     <bean id="messageDAOTarget"
07:               class="MessageDAOImpl">
08:         <property name="sessionFactory" ref="sessionFactory"/>
09:     </bean>
10:     <bean id="messageDAO"
11:               class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
12:         <property name="interceptorNames">
13:             <list><value>securityInterceptor</value></list>
14:         </property>
15:         <property name="beanNames">
16:             <list><value>messageDAOTarget</value></list>
17:          </property>
18:     </bean>
19:     <bean id='securityInterceptor'
20:               class='net.sf.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor'>
21:         <property name='authenticationManager' ref='authenticationManager'/>
22:         <property name='accessDecisionManager' ref='accessDecisionManager'/>
23:         <property name='objectDefinitionSource'>
24:             <value>
25:                 MessageDAO.createMessage=ROLE_APPRENTICE,ROLE_MEDIATOR
26:                 MessageDAO.updateMessage=ROLE_MEDIATOR
27:                 MessageDAO.removeMessage=ROLE_MEDIATOR
28:                 MessageDAO.list*=ROLE_APPRENTICE,ROLE_MEDIATOR
29:             </value>
30:         </property>
31:     </bean>
32:     <bean id='authenticationManager'
33:                class='net.sf.acegisecurity.providers.ProviderManager'>
34:         <property name='providers'>
35:             <list><ref bean='authenticationProvider'/></list>
36:         </property>
37:     </bean>
38:     <bean id='authenticationProvider'
39:               class='net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider'>
40:         <property name='authenticationDao' ref='memoryAuth'/>
41:     </bean>
42:     <bean id='authenticationDao'
43:               class='net.sf.acegisecurity.providers.dao.memory.InMemoryDaoImpl'>
44:         <property name='userMap'>
45:             <value>
46:                 paulo=123456,ROLE_APPRENTICE
47:                mauro=abacate,ROLE_MEDIATOR
48:            </value>
49:         </property>
50:     </bean>
51:     <bean id='accessDecisionManager'
52:               class='net.sf.acegisecurity.vote.AffirmativeBased'>
53:         <property name='decisionVoters'>
54:             <list><ref bean='roleVoter'/></list>
55:         </property>
56:     </bean>
57:     <bean id='roleVoter' class='net.sf.acegisecurity.vote.RoleVoter'/>
58: </beans>
```

Listagem 3.18 - Arquivo de Configuração do Spring: Segurança Declarativa.

O trecho entre as linhas 10 e 18 define o proxy que acrescenta o aspecto segurança a classe MessageDAOImpl sendo que o interceptador relativo a este proxy é configurado na linha 13 e declarado no trecho entre as linhas 19 e 31. O

PUC-Rio - Certificação Digital Nº 0410823/CA

gerenciador de autenticação é configurado na linha 21 e o gerenciador de controle de acesso na linha 22.

As linhas 25 e 28 indicam que os métodos de criação e listagem de mensagens estão disponíveis tanto para aprendizes quanto para mediadores. Já as linhas 26 e 27 indicam que apenas mediadores podem apagar ou atualizar mensagens. O trecho entre as linhas 32 e 50 referem-se à configuração do gerenciador de autenticação e o trecho entre as linhas 51 e 56 se referem à configuração do gerenciador de controle de acesso.

O framework Spring possibilita um serviço de segurança declarativa para POJOs, de forma similar ao dos EJBs. Este framework fornece ainda meios de expor serviços remotamente, como é visto na próxima seção.

3.5.6. Exposição de Serviços Remotos com Spring

PUC-Rio - Certificação Digital Nº 0410823/CA

Eventualmente é necessário expor as funcionalidades da camada de negócio de uma aplicação remotamente para outras aplicações e para outros clientes, inclusive clientes móveis. Spring possibilita a exposição de seus beans remotamente de quatro formas diferentes: através do uso de RMI (RMI-IIOP. 2005), de Hessian/Burlap (Caucho, 2005), do HttpInvoker (Spring, 2005) e de serviços web JAX-RPC (Singh et al., 2004).

RMI ou *Remote Method Invocation* é uma tecnologia que possibilita a chamada a métodos de componentes remotos, também usada no Enterprise JavaBeans. Sua principal vantagem é que ela é eficiente quando a comunicação é feita entre duas plataformas Java. Modelos complexos de objetos podem ser enviados através do mecanismo de serialização da plataforma Java. Sua principal desvantagem é que ela usa portas arbitrárias para se comunicar, o que exige esforço extra de configuração em firewalls, e não possibilita a comunicação de Java com outras plataformas.

Hessian e Burlap são protocolos desenvolvidos pelo grupo Caucho (2005) que possibilitam a comunicação remota entre serviços. Ao contrário do RMI, Hessian e Burlap são construídos sobre o HTTP e, portanto não costumam demandar configuração extra em firewall. Além disso, Hessian/Burlap possibilita a comunicação entre as plataformas Java, PHP, Python, C++ e C#. Hessian utiliza

um protocolo binário, ideal quando a largura da banda é limitada e Burlap utiliza um protocolo baseado em XML, ideal quando é preciso que as mensagens trocadas sejam lidas por pessoas. Estes dois protocolos usam técnicas próprias para enviar objetos pela rede que podem ser ineficientes quando se usa um modelo complexo de classes.

O HttpInvoker é um protocolo desenvolvido exclusivamente para o Spring e soluciona este problema. Assim como Hessian/Burlap ele usa HTTP e, portanto não demanda configurações extras de firewall. Assim como o RMI, ele usa a tecnologia de objetos serializáveis da plataforma Java, que é capaz de enviar modelos complexos de objetos pela rede. HttpInvoker é a opção ideal quando é preciso integrar duas aplicações que usam o framework Spring pela internet.

A tecnologia de serviços web possibilita a chamada de métodos remotos através da troca de arquivos XML, a descoberta automática de serviços dentre outras vantagens. Esta tecnologia é executada sobre o HTTP então firewalls não costumam ser um empecilho e, como utiliza um padrão bem conhecido, é viável realizar a comunicação entre diversas plataformas. Por outro lado, das quatro tecnologias para exposição de serviços remotos, é a mais pesada e sua performance pode não ser adequada em alguns cenários.

Spring possibilita que serviços sejam expostos sem que seja necessário alterar o código fonte da aplicação. Para expor um serviço usando RMI, é preciso configurar o *Proxy* adequado no arquivo de configuração. Para expor um serviço com Hessian/Burlap ou HttpInvoker, além de configurar o *Proxy* no arquivo de configuração é preciso declarar um servlet que trata as conexões HTTP. A exposição de serviços como serviços web é mais trabalhosa, pois envolve também a criação do arquivo descritor do serviço WSDL.

Para fins de demonstração, a Listagem 3.19 apresenta o arquivo de configuração do Spring onde o serviço Conferência é exposto utilizando a tecnologia RMI, a mesma usada pelo Enterprise JavaBeans.

PUC-Rio - Certificação Digital Nº 0410823/CA

```
01: <beans>
02:     <!-- Declaração de Outros Beans -->

03:     <bean id="conferenceFacade"  class="ConferenceFacadeImpl">
04:         <property name="messageDao" ref="messageDAO"/>
05:     </bean>

06:     <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
07:         <property name="serviceName" value="Conference"/>
08:         <property name="service" ref="conferenceFacade"/>
09:         <property name="serviceInterface" value="ConferenceFacade"/>
10:         <property name="registryPort" value="1199"/>
11:     </bean>
12: </beans>
```

Listagem 3.19 - Arquivo de Configuração do Spring: Exposição de Serviços Remotos.

O bean da conferência é declarado entre as linhas 03 e 05. Neste caso, considera-se que ConferenceFacadeImpl é a classe da conferência que implementa a interface ConferenceFacade. O uso de interfaces é obrigatório quando se lida com *Proxys* do Spring.

O trecho entre as linhas 06 e 11 declara o *Proxy* dinâmico responsável por expor o serviço Conferência remotamente usando RMI. Na linha 07 é associado um nome ao serviço que será usado para cadastrá-lo no registro RMI. Na linha 08 é especificado o bean cujos serviços são expostos pelo *Proxy*, no caso, o bean conferenceFacade. Na linha 09 é declarada a interface que este *Proxy* dinâmico implementa e na linha 10, a porta em que o serviço e exposto.

O framework Spring possibilita que serviços sejam expostos declarativamente, sem que seja necessário alterar o código fonte. Como conseqüência, pode-se trocar o mecanismo de comunicação sem alterar o código, aumentando o reuso.

3.5.7.
Considerações Finais Sobre o Spring

Spring é capaz de prover serviços de infra-estrutura como o controle de transações, segurança e exposição de serviços remotos assim como Enterprise JavaBeans mas, diferentemente do EJB, Spring não é intrusivo. As classes gerenciadas pelo Spring não dependem necessariamente do Spring (Johnson, 2004).

PUC-Rio - Certificação Digital Nº 0410823/CA

Além disso, o módulo de *Dependency Injection* promove o acoplamento fraco entre componentes. Graças a este fraco acoplamento as classes são mais fáceis de testar unitariamente. As dependências são explicitadas e podem ser substituídas por *mock objects* (Mackinnon et al., 2000), objetos que imitam objetos reais e os substituem em testes.

Finalmente, por ser um framework modular, apenas os módulos que interessam à aplicação precisam ser selecionados. Desta forma, evita-se complexidade desnecessária.

3.5.8. Outros Frameworks Para Dependency Injection

Spring não é o único framework/container leve disponível no mercado. Há o Avalon (2005), o HiveMind (2005), NanoContainer (2005) e o PicoContainer (2005). Todos estes frameworks são gratuitos e, possíveis substitutos para o Spring.

O projeto Avalon foi encerrado em 2004 e transformado no projeto Excalibur (2005). O projeto PicoContainer provê apenas o suporte essencial para um framework de *Dependency Injection* e tem suas funcionalidades estendidas pelo NanoContainer. O HiveMind é o mais similar ao Spring, provendo também um módulo de programação orientada a aspectos.

Spring aliado ao Hibernate oferece vários recursos de infra-estrutura. Para reforçar a escolha do Spring é feita uma análise similar a que é feita com os frameworks ORM: é verificado a quantidade de documentação disponível, a disponibilidade de suporte, a disponibilidade de ferramentas compatíveis, o grau de aceitação no mercado e a disponibilidade de profissionais com habilidades nos frameworks comparados.

3.5.8.1. Quantidade de Documentação Disponível

Pode-se ter uma idéia da quantidade de documentação disponível sobre um framework considerando o número de livros publicados e realizando uma consulta em uma ferramenta de busca para contabilizar o número de tutoriais disponíveis sobre este.

PUC-Rio - Certificação Digital Nº 0410823/CA

O gráfico apresentado na Figura 3.15 mostra o número de livros publicados para cada um dos frameworks para *Dependency Injection* segundo uma pesquisa realizada em dezembro de 2005 no site Amazon.

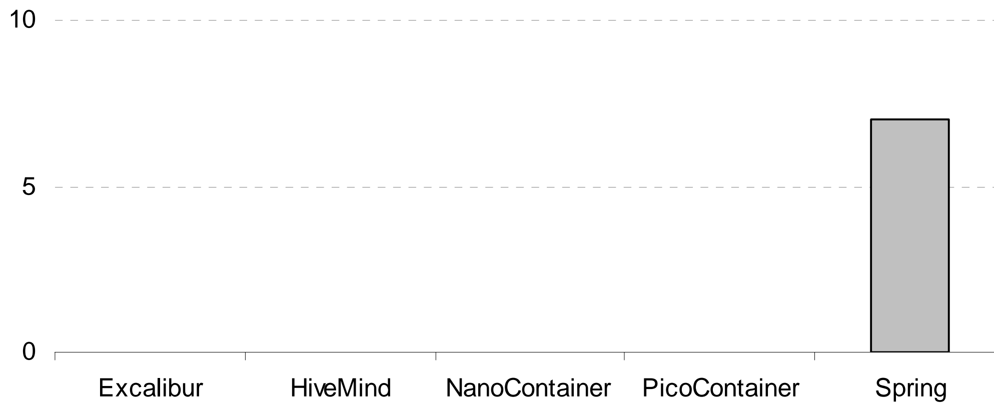

Figura 3.15 - Livros Publicados Para Cada Framework de *Dependency Injection*
Fonte: Amazon.com, Dezembro de 2005

PUC-Rio - Certificação Digital Nº 0410823/CA

Só há livros publicados sobre o Spring. O gráfico exibido na Figura 3.16 mostra o resultado de uma busca por tutorias na ferramenta de busca Google em março de 2006.

Figura 3.16 - Artigos e Tutoriais Para Cada Framework de *Dependency Injection*
Fonte: Google.com, Março de 2006

Há um número tão pequeno de tutoriais disponíveis sobre o NanoContainer, que não é possível visualizar sua contribuição no gráfico. Os frameworks

Excalibur, HiveMid e PicoContainer também possuem um número pequeno de tutoriais quando comparados com o Spring. Spring é claramente o framework para *Dependency Injection* com mais documentação disponível, levando em consideração os livros publicados e tutoriais disponíveis na web.

3.5.8.2. Disponibilidade de Suporte

É importante considerar a disponibilidade de suporte para cada framework antes de efetuar uma decisão. Excalibur e HiveMind possuem listas de discussão mantidas pela comunidade onde pode-se obter suporte gratuito. PicoContainer e o NanoContainer compartilham a mesma lista de discussão. Já o Spring conta com um fórum de discussão oficial e uma lista de discussão não-oficial onde podem ser obtidos suporte da comunidade além de suporte comercial fornecido pela empresa Interface 21 (2005).

Para mensurar a disponibilidade de suporte destes frameworks, realiza-se uma pesquisa contabilizando o volume de mensagem trocado nestas listas de discussão em determinado período de tempo. Infelizmente não é possível mensurar o volume de mensagens trocadas no fórum do Spring, pois este não possibilita a seleção de mensagens em um determinado período de tempo. Tentei entrar em comunicação pessoal com os responsáveis pela administração do fórum, mas até a data de fechamento deste texto não houve resposta (os e-mails trocados estão transcritos no apêndice A) e, portanto, serão contabilizados os dados da lista de discussão não-oficial.

O gráfico exibido na Figura 3.17 mostra o resultado da análise realizada em novembro de 2005.

PUC-Rio - Certificação Digital Nº 0410823/CA

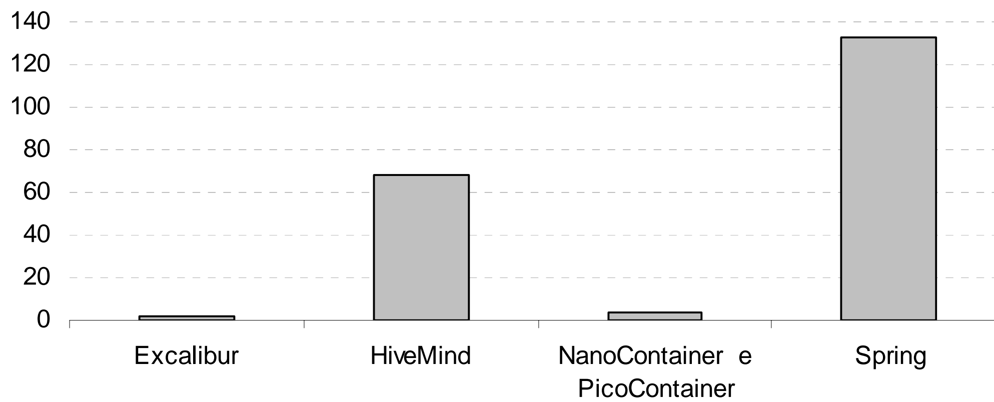

Figura 3.17 - Mensagens Trocadas em Listas de Discussão (11/2005)

Mesmo em uma lista não-oficial, o Spring se mostrou o framework para *Dependency Injection* com a comunidade mais ativa, tendo o HiveMind em segundo lugar. As listas de discussão dos frameworks Excalibur, NanoContainer e PicoContainer apresentou um volume consideravelmente pequeno de mensagens trocadas no mês de Novembro indicando que pode ser difícil obter suporte da comunidade para estes frameworks e que provavelmente há poucas pessoas usando estes frameworks.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.5.8.3. Disponibilidade de Ferramentas Compatíveis

Ferramentas compatíveis com um framework auxiliam o desenvolvimento, automatizando tarefas repetíveis, possibilitando a edição de documentos com ferramentas gráficas e a criação de artefatos com formulários estilo wizard, etc. Para contabilizar o número de ferramentas compatíveis com cada framework foram verificados a quantidade de plugins disponíveis para as plataformas Eclipse e NetBeans, a compatibilidade com as IDEs Java mais populares (IBM WSAD, Sun Java Studio Enterprise, Borland JBuilder, Oracle JDeveloper, BEA Workshop e InteliJ IDEA), a compatibilidade com a ferramenta XDoclets além das ferramentas disponíveis nos web sites dos próprios frameworks. A Figura 3.18 mostra o número de ferramentas disponíveis compatíveis com cada framework em uma análise realizada em dezembro de 2005.

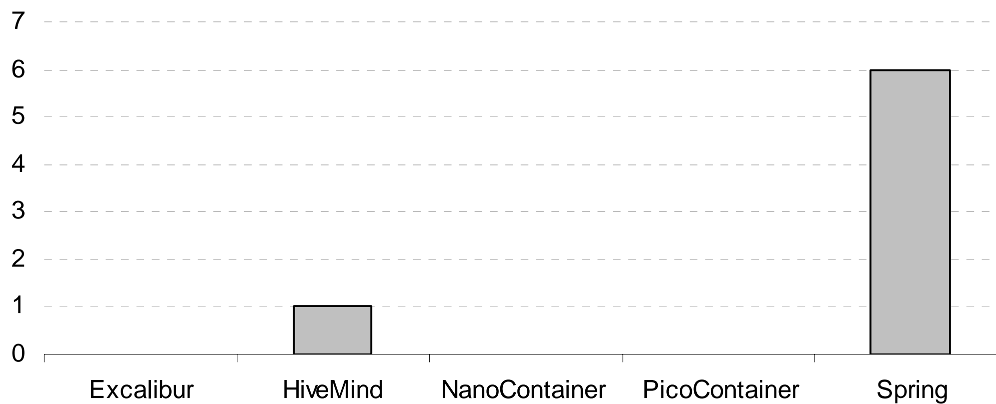

Figura 3.18 – Ferramentas Compatíveis com os Frameworks para *Dependency Injecton* em 12/2005

Constata-se que não há ferramentas que oferecem suporte ao desenvolvimento com Excalibur, NanoContainer e PicoContainer. Há 1 ferramenta que oferece suporte ao desenvolvimento com HiveMind e 6 que dão suporte ao desenvolvimento com o Spring.

PUC-Rio - Certificação Digital Nº 0410823/CA

3.5.8.4. Grau de Aceitação no Mercado

É importante saber o que outras instituições estão usando antes de tomar a decisão a respeito de um framework. Um framework com boa aceitação pelo mercado estimula outros fatores como a disponibilidade de ferramentas compatíveis, a disponibilidade de suporte, etc. Por outro lado, um framework que não é aceito pelo mercado corre o risco de desaparecer.

Para obter um indicativo do grau de aceitação do mercado para um framework, faz-se uma busca em sites de empregos, procurando por ofertas de emprego abertas que solicitem desenvolvedores com habilidades em um determinado framework. A Figura 3.19 mostra o resultado da pesquisa realizado em dezembro de 2005 no site Dice.com.

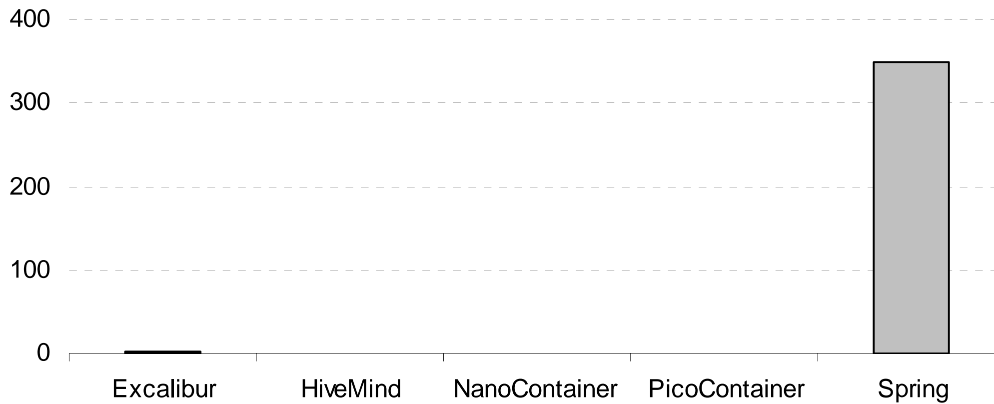

Figura 3.19 – Ofertas de Emprego Abertas para cada Framework para *Dependency Injection*. Fonte: Dice.com, Dezembro de 2005

Spring é o framework com mais vagas abertas no mercado de trabalho norte-americano. Não há vagas abertas exigindo conhecimento com os Frameworks NanoContainer e PicoContainer. O número de vagas abertas demandando habilidades com Excalibur (2 vagas) e o HiveMind (1 vaga) e tão pequeno que não pode ser notada nitidamente no gráfico.

O site Dice.com revela um retrato do mercado norte-americano, mas não representa necessariamente a realidade do mercado brasileiro. Para verificar se a realidade do mercado brasileiro é similar, realizou-se uma busca em dezembro de 2005 no site nacional Manager Online (http://www.manageronline.com.br/).

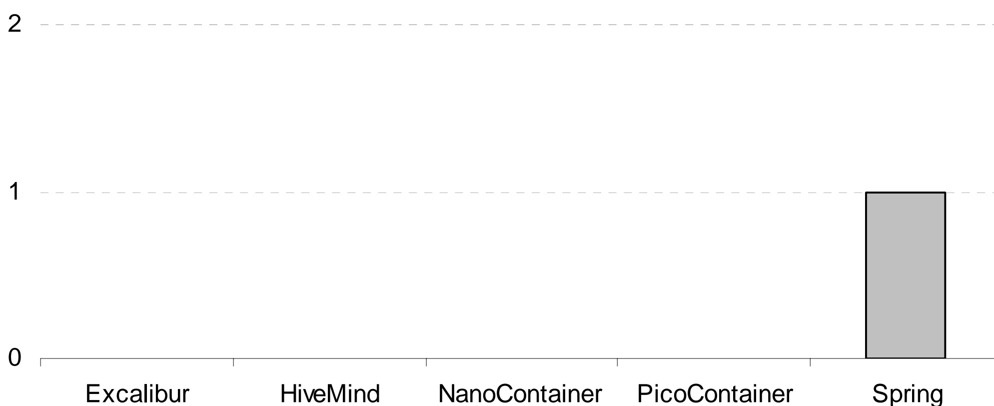

Figura 3.20 - Vagas Abertas para cada Framework de Dependency Injection
Fonte: Manager Online, Dezembro de 2005.

PUC-Rio - Certificação Digital Nº 0410823/CA

A pesquisa revela que só há 1 vaga aberta para o framework Spring segundo a busca feita no site ManagerOnline. O número consideravelmente pequeno dificulta qualquer comparação.

3.5.8.5. Disponibilidade de Profissionais

Para diminuir gastos com treinamento é importante verificar a disponibilidade de mão de obra no mercado de trabalho com aptidão a trabalhar com os frameworks comparados. Para verificar esta disponibilidade faz-se uma consulta em sites que hospedam currículos. A Figura 3.21 mostra o resultado da busca realizada em dezembro de 2005 no site Jobs.net que reflete a realidade do mercado de trabalho norte-americano.

PUC-Rio - Certificação Digital Nº 0410823/CA

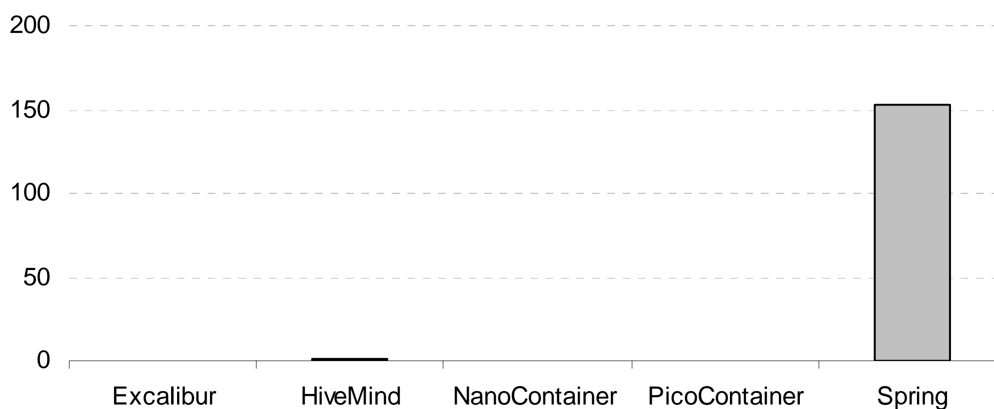

Figura 3.21 - Disponibilidade de Profissionais em 12/2005
Fonte: Jobs.net.

Há pouco mais que 150 profissionais que se dizem aptos a trabalhar com o framework Spring em seus currículos. Há um número consideravelmente pequeno de profissionais aptos a trabalhar com o HiveMind e nenhum profissional apto a para trabalhar com os outros frameworks. Para validar se a realidade do mercado norte-americano é compatível com a realidade do mercado brasileiro realizou-se uma busca no site nacional APInfo.com em dezembro de 2005.

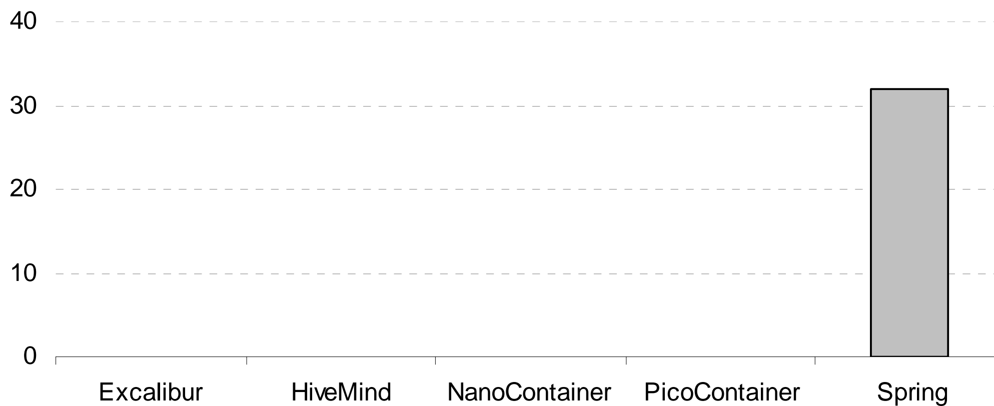

Figura 3.22 - Disponibilidade de Profissionais
Fonte: APInfo.com, Dezembro de 2005.

Segundo a busca no site APInfo só o Spring possui profissionais que se declaram aptos a trabalhar com o framework em seus currículos.

3.5.8.6. Resultado da Comparação

Após comparar todos os critérios chega-se a conclusão que Spring é a opção mais adequada para ser usada na arquitetura do AulaNet 3.0. Todos os outros frameworks são carentes em vários aspectos. Spring possui farta documentação disponível, suporte da comunidade e comercial e ferramentas compatíveis que auxiliam o desenvolvimento. Além disso, Spring tem recebido uma boa aceitação no mercado (principalmente no norte-americano) e há profissionais aptos a trabalhar com ele disponíveis.

O EJB novamente não é incluído na análise, pois ele não é um framework para *Dependency Injection* e porque ele é descartado na análise técnica ao considerar suas desvantagens.

3.6. Conclusão

O objetivo da camada de negócios é implementar a lógica da aplicação, expondo esta lógica para a camada de apresentação ou para outras aplicações clientes remotas, por exemplo, clientes móveis. Para cumprir este objetivo, pode-se optar por um framework de componentes, como o Enterprise JavaBeans ou

PUC-Rio - Certificação Digital Nº 0410823/CA

pode-se criar o próprio framework de componentes a partir de uma arquitetura com POJOs (*Plain Old Java Objects*).

EJB traz vantagens relacionadas a aspectos de infra-estrutura tratados por ele. Dentre estes aspectos os principais são: persistência automática dos beans de entidade, componentes representando entidades de negócio. A persistência automática livra o desenvolvedor de escrever código JDBC de baixo nível e possibilita que a aplicação seja executada em bancos de dados diferentes; controle de transações declarativo, a demarcação das transações é feita fora do código fonte, tornando o código mais limpo e incentivando o reuso; gerenciamento de segurança com demarcações declarativas que assim como a transação declarativa incentiva o reuso já que componentes podem ter seu acesso controlado sem que seja necessário alterar o código fonte e exposição de serviços remotos, que possibilita que aplicações externas acessem o componente.

Contudo, EJB também traz desvantagens. Dentre elas, as principais são: a grande quantidade de arquivos que precisam ser mantidos para definir um componente; a dependência com servidores de aplicações compatíveis com EJB que usualmente são mais caros e mais difíceis de configurar; elevado grau de intrusão que dificulta a execução de testes unitários em componentes EJB e restrições de programação impostas pela especificação que impossibilitam a realização de atividades corriqueiras.

Ao invés de usar EJB, é possível criar um framework de componentes próprio a partir de uma arquitetura com POJOs, agregando frameworks de infra-estrutura que provêm as mesmas vantagens do EJB sem a maior parte das desvantagens.

Para realizar a persistência de objetos, foi escolhido o Hibernate, um framework ORM que trata a persistência de objetos em base de dados relacionais. Ao contrário do EJB que é uma solução de mapeamento médio, Hibernate é uma solução de mapeamento completo, isto é, possibilita o mapeamento de modelos complexos envolvendo estruturas de herança de forma transparente, sem que as classes persistidas precisem estender classes ou implementar interfaces do framework. Hibernate trata também questões complexas relativas ao conflito de paradigmas objeto/relacional. Outros frameworks ORM de mapeamento completo gratuitos disponíveis são: Apache OJB, iBATIs e JDO. Contudo, Hibernate possui mais documentação disponível, um maior número de ferramentas compatíveis,

PUC-Rio - Certificação Digital Nº 0410823/CA

maior disponibilidade de suporte, melhor aceitação no mercado de trabalho e maior quantidade de mão de obra disponível.

Sozinho o Hibernate não oferece todos os serviços de infra-estrutura presentes no EJB, por isto ele deve ser combinado com o framework Spring. Este último integra-se ao Hibernate provendo também suporte a transações declarativas, gerenciamento de segurança declarativa e exposição de serviços remotos. Além disso, através de *Dependency Injection*, Spring promove o acoplamento fraco entre classes, facilitando a criação de testes unitários e incentivando o reuso. Há outros frameworks similares ao Spring, dentre eles o Excalibur, HiveMind, PicoContainer e o NanoContainer. Porém estes são carentes em quantidade de documentação disponível, ferramentas compatíveis, suporte da comunidade ou comercial, grau de aceitação no mercado e quantidade de profissionais disponíveis aptos a trabalhar com cada um.

O par Hibernate e Spring possibilita a incorporação de serviços de infra-estrutura a arquitetura do AulaNet 3.0, possibilitando que o desenvolvedor de componentes de groupware concentre-se em seu domínio ao invés de se preocupar com serviços de baixo nível. Por se tratar de uma tecnologia aplicada a POJOs, classes normais que não seguem nenhuma especificação rígida, pode também ser aplicada a um framework de componentes desenvolvido para gerar componentes de groupware. Este capítulo apresentou como um groupware se beneficia de frameworks de infra-estrutura quando aplicados na camada de negócios. No próximo capítulo é mostrado como a camada de apresentação também obtem benefícios da incorporação destes frameworks.

PUC-Rio - Certificação Digital Nº 0410823/CA