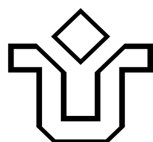

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

Relatórios Técnicos
do Departamento de Informática Aplicada
da UNIRIO
n° 0007/2011

Propagação de Identidade em Aplicações em Três Camadas

Felipe Leão
Leonardo Guerreiro Azevedo
Flávio Faria
Fernanda Baião
Claudia Cappelli

Departamento de Informática Aplicada

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Av. Pasteur, 458, Urca - CEP 22290-240
RIO DE JANEIRO – BRASIL

Projeto de Pesquisa

Grupo de Pesquisa Participante

Patrocínio

PETROBRAS

Propagação de Identidade em Aplicações em Três Camadas

Felipe Leão, Leonardo Guerreiro Azevedo, Flávio Faria, Fernanda Baião, Claudia Cappelli

Núcleo de Pesquisa e Prática em Tecnologia (NP2Tec)
Departamento de Informática Aplicada (DIA) – Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

{felipe.leao, azevedo, flavio.faria, fernanda.baiao, claudia.cappelli}@uniriotec.br

Abstract. Information security is a critical issue for organizations and comprises several perspectives, including that of information systems and data repositories development. In current distributed information systems architectures, information security involves identity propagation and data access authorization issues. This work describes the design and implementation of a simple, yet not trivial, architecture for effectively assuring information security, based on a series of standard technology.

Keywords: Information Security, Role-Based Access Control, Identity Propagation; Access Authorization.

Resumo. Segurança de informação é uma questão crítica em organizações que deve ser tratada sob diferentes perspectivas, incluindo a do desenvolvimento dos sistemas de informação e dos repositórios dos dados acessados por tais sistemas. Nos diversos cenários de arquitetura distribuída dos sistemas de informação atuais, o problema da segurança envolve as questões de propagação de identidade e de regras de autorização de acesso aos dados. Este artigo descreve o projeto e implementação de uma arquitetura eficaz para solução destes problemas que integra, de forma simples e não trivial, uma série de tecnologias disponíveis.

Palavras-chave: Segurança da Informação, Role-Based Access Control, Propagação de Identidade, Autorização de Acesso.

---

* Trabalho patrocinado pela Petrobras.

Sumário

1 Introdução 7

2 Autorização de acesso 8

2.1 FARBAC 8

2.2 Modelo conceitual das regras de autorização 9

2.3 Algoritmo para construção do predicado de autorização 10

3 Elaboração de proposta de solução 11

3.1 Aplicação Cliente 14

3.2 Servidor de aplicação 15

4 Conclusão 16

Referências Bibliográficas 18

Apendice A – Criação do protótipo para testes de propagação de identidade 20

A1. Configuração do FARBAC 20

A2. Criação da Interface para Controle de Acesso 21

A3. Criação do EJB para Acesso aos Dados 21

A3.1. Criação do Projeto 21

A3.2. Implementação das classes do projeto e configuração do Hibernate 24

A4. Aplicação Cliente 29

A5. Executando o protótipo 33

Apêndice B – Tecnologias e padrões relacionados 35

B1. JBoss security 35

B1.2. Referências de Segurança 35

B2. Injeção de Dependência 36

B2.2. *Contexts and Dependency Injection* (CDI) 37

B2.3. Spring Framework 38

B2.4. Google Guice 40

B2.5. OpenEJB 42

B3. Hibernate 42

B3.2. Configurações Básicas 42

B3.3. Interceptadores 45

B3.4. Interface Validatable 45

B3.4.1. Interface Lifecycle 46

B3.4.1.1. Interface *Interceptor* 46

B3.4.1.2. Análise do uso de interceptadores para os cenários 3 e 4 47

B3.4.2. Padrões de Projeto para Hibernate 48

B3.4.2.1. Padrões de projeto para utilização de *Sessions* 48

B3.4.2.2. Análise do uso de padrões Hibernate para os cenários 3 e 4 49

B4. *Design Patterns* (Padrões de Projeto) 49

B4.2. Singleton 50

B4.3. Service Locator 50

B4.4. DAO (Data Access Object) 51

Figuras

Figura 1- Modelo de entidade e relacionamento do FARBAC......................9

Figura 2 - Algoritmo de construção do predicado de autorização. 11

Figura 7 – Módulos da arquitetura de controle de acesso proposta 12

Figura 8 - Diagrama de Sequência descrevendo o protótipo proposto....... 13

Figura 9 - Interface gráfica da Aplicação Cliente 14

Figura 10 - Método get da classe EJBLocator responsável por recuperar um objeto EJB no servidor de aplicação... 14

Figura 11 - Implementação da Interface ControleAcesso........................ 15

Figura 12 - Método getSession da classe HibernateUtil 16

Figura 13 - Consulta realizada pelo EJB... 21

Figura 14 - Predicado gerado pelo FARBAC quando o protótipo é acessado pelo usuário Y5QL 21

Figura 15 - Implementação da interface ControleAcesso 21

Figura 16 - Criação de um novo projeto EJB 22

Figura 17 - Nomeação do novo projeto EJB 22

Figura 18 - Escolha do Servidor de aplicação do Projeto EJB.................. 23

Figura 19 - Estrutura de Pacotes do EJB.. 24

Figura 20 - *Wizard* para criação de um Bean de Sessão 24

Figura 21 - Interface do EJB.. 25

Figura 22 - Implementação da classe ManterVendas 26

Figura 23 - Trecho da implemenação da classe VendasDAO................... 27

Figura 24 - Importando JAR para o projeto... 28

Figura 25 - Implementações do método getSession da classe HibernateUtil 29

Figura 26 - Criação do Projeto Cliente.. 30

Figura 27 - Estrutura de pacotes de classes do projeto cliente 30

Figura 28 - Método "main" da classe "Main" inicializando interface gráfica 30

Figura 29 - Interface do protótipo 31

Figura 30 - Classe ServiceLocator 32

Figura 31 - Ação do botão Consultar.. 32

Figura 32 - Implementação do método para exibir mensagens ao usuário 33

Figura 33 - Implantação do EJB no servidor de aplicação 33

Figura 34 - Protótipo em execução.. 34

Figura 35 - Tela de Interrupção do Programa Cliente............................ 34

Figura 36 - O elemento security-role-ref 36
Figura 37 - Fragmento de um descritor *ejb-jar.xml* ilustrando o uso do elemento security-role-ref 36
Figura 38 - Fragmento de um descritor *web.xml* ilustrando o uso do elemento security-role-ref 36
Figura 39 – Classe *DefaultItemDAO* que implementa a interface *ItemDao*. 37
Figura 40 – Classe *ItemProcessor* implementada sem injeção de dependência 37
Figura 41 - Classe *ItemProcessor* implementada utilizando injeção de dependência 38
Figura 42 – Arquivo *contexto.xml* 38
Figura 43 – O *bean CadUser* 39
Figura 44 – O *Bean Foo* 39
Figura 45 – Classe TesteDI utilizada para testar a aplicação de exemplo .. 40
Figura 46 – Resultado da execução da classe *TesteDI* 40
Figura 47 – Módulo de configuração que implementa a *interface Module* .. 41
Figura 48 - Módulo de configuração que estende a classe *AbstractModule*. 41
Figura 49 – *Bean Foo* 41
Figura 50 – Método de teste que cria o injetor e recupera os *beans CadUser* e *Foo* 42
Figura 51 - Arquivo de configuração *hibernate.properties* 43
Figura 52 - Arquivo de configuração *hibernate.cfg.xml* 43
Figura 53 - Arquivo de configuração do log4j 44
Figura 54 - Arquivo de mapeamento para a classe Poço 44
Figura 55 - Definição de um interceptador *Session-scoped* 45
Figura 56 - Definição de um interceptador *SessionFactory-scoped* 45
Figura 57 - Classe implementando a interface *Validatable* 46
Figura 58 - Implementação da classe auxiliar *HibernateUtil* 48
Figura 59 - Implementação usual do padrão Singleton 50
Figura 60 - Implementação do padrão Service Locator 51
Figura 61 - Exemplo de método implementado em uma clsse DAO 52

1 Introdução

A segurança de informação é um das principais questões de organizações governamentais e privadas, as maiores questões se encontram nos mecanismos de controle (ou autorização) de acesso para garantia da integridade [Sandhu *et al.*, 1996; Yang, 2009; Cali e Martinenghi, 2008; Murthy e Sedlar, 2007]. Estes mecanismos fazem com que regras de negócio do tipo assertiva de ação de autorização sejam garantidas. Segundo [BRG 2009], regra de negócio é uma declaração que define ou restringe algum aspecto de uma organização. Regras de negócio têm como objetivo definir a estrutura de um negócio ou controlar ou influenciar o seu comportamento. Em particular, uma categoria de regras de negócio é a assertiva de ação de autorização, ou **regra de autorização**, a qual restringe **quem** é permitido realizar uma **ação** na organização sobre quais **informações**.

Em diversos cenários de arquitetura distribuída dos sistemas de informação atuais, onde as camadas de um sistema (aplicação cliente, servidor de aplicação, componente de acesso a dados, servidor de dados) encontram-se implementadas em componentes independentes e alocadas a unidades de processamento distintas, os principais mecanismos para implementação do controle de acesso são a propagação de identidade entre as camadas do sistema de informação e a aplicação de regras de autorização. Estes mecanismos fazem com que a identidade do usuário seja considerada quando do acesso ao dado de modo que sejam manipulados apenas os dados que o usuário tem acesso e também que a aplicação tenha conhecimento do que este determinado usuário pode fazer com a informação acessada.

O objetivo deste trabalho é propor uma abordagem para propagação de identidade que possa ser utilizada em cenários de arquiteturas de aplicação existentes descritas por Azevedo *et al.* [2009], os quais são:

- Cenário 1: Cliente acessa banco diretamente executando um processo identificável;
- Cenário 2: Cliente acessa o banco diretamente executando um processo não identificável;
- Cenário 3: Cliente acessa servidor de aplicação que acessa o banco.
- Cenário 4: Cliente acessa servidor de aplicação via serviços web.

Os cenários 1 e 2 foram analisados e abordagens que podem ser utilizadas são: *Secure Session-Based Application Context* e *Global Application Context* [Jeloka *et al*, 2008].

A *Secure Session-Based Application Context* é uma abordagem onde os valores dos atributos do contexto ficam armazenados na *User Global Area* (UGA), que é uma área da memória que armazena informações relacionadas à sessão. Isso significa que os atributos do contexto são definidos por sessão, ou seja, um usuário do banco pode possuir em duas sessões concorrentes valores distintos para um mesmo atributo. Além disso, ao fechar uma sessão o contexto é automaticamente limpo.

A *Global Application Context* é uma abordagem onde os valores dos atributos do contexto ficam armazenados na *System Global Area* (SGA), que é uma área da memória definida para cada instância do Oracle. Isso significa que os atributos do contexto são definidos para toda a instância, ou seja, se em uma sessão o usuário do

banco define o valor de um atributo, todas as suas outras sessões concorrentes verão esse mesmo valor.

Outra característica do contexto global é que os valores dos atributos são associados a identificadores, de forma que o mesmo atributo pode receber mais de um valor para cada identificador definido. Os valores podem ser depois recuperados através deste identificador.

No entanto, estas soluções causam um impacto muito grande nas aplicações quando aplicadas para os cenários 3 e 4. Dessa forma, o escopo deste trabalho busca soluções para o cenário 3 (correspondente a três camadas: Aplicação Cliente, Servidor de Aplicação e Servidor de Banco de dados), mas que também atendam aos cenários 1 e 2. As soluções para o cenário 4 são propostas de trabalho futuro.

As seguintes etapas foram consideradas para atender o objetivo:

- Analisar os cenários existentes e ilustrar com desenhos de arquiteturas que apresentem de forma resumida o que será tratado;
- Buscar por exemplos reais de arquiteturas para avaliação de propostas;
- Pesquisar por soluções existentes que representem o estado-da-arte e possam ser aplicadas na prática. Logo, será dada prioridade para soluções de mercado e com amplo uso;
- Avaliar soluções estudadas em laboratório;
- Analisar resultados obtidos a fim de definir melhor abordagem para adoção.

Este relatório foi produzido pelo Projeto de Pesquisa em Autorização de Informação como parte das iniciativas dentro do contexto do Projeto de Pesquisa do Termo de Cooperação entre UNIRIO/NP2Tec e a PETROBRAS/TIC-E&P/GDIEP.

Esse relatório está organizado em 6 capítulos, sendo o Capítulo 1 a presente introdução. O Capítulo 2 apresenta o *framework* para autorização de acesso. O Capítulo 3 apresenta um exemplo de cenário real para propagação de identidade e uma solução específica baseada para este cenário. O Capítulo 4 apresenta uma proposta de solução mais genérica que atende tanto ao cenário real como outros cenários. O Capítulo 5 apresenta a análise da proposta de soluções e as conclusões do trabalho. Além disso, o Apêndice A apresenta detalhadamente como as implementações da proposta de solução foram realizadas e os testes executados e o Apêndice B apresenta características de tecnologias utilizadas no trabalho.

2 Autorização de acesso

Uma vez que o usuário atual do sistema de informação tenha sido reconhecido pelo SGBD, à execução das regras de autorização em tempo de execução. Esta seção apresenta a arquitetura para autorização de acesso empregada neste trabalho, denominada FARBAC, descrita em detalhes por Azevedo *et al.* [2010a, 2010b].

2.1 FARBAC

O FARBAC (Flexible Framework for Role-Based Access Control) foi implementado utilizando o mecanismo VPD – Virtual Private Database [Jeloka *et al.*, 2008] presente no SGBD (Sistema Gerenciador de Banco de Dados) Oracle.

No FARBAC as regras de autorização são definidas como cláusulas WHERE, que são dinamicamente adicionadas aos comandos enviados pelas aplicações ao SGBD. Em seleções, por exemplo, as regras funcionam como um filtro, removendo da consulta os registros que o usuário não possui acesso. As regras são associadas a perfis (ou papéis) que são concedidos a cada usuário. O FARBAC recupera em tempo de execução o usuário executando o comando e aplica as regras referentes aos perfis desse usuário.

Uma premissa para o funcionamento do FARBAC é que a identidade do usuário seja propagada corretamente nas diversas camadas do sistema. Nesse caso, como o modelo foi implementado no Oracle, a identidade do usuário chega ao SGBD através da chamada à *procedure dbms_application_info.set_client_info*. Essa *procedure* define um valor para o atributo *client_info* do *namespace userenv*, que depois é recuperado pelo FARBAC.

2.2 Modelo conceitual das regras de autorização

As regras de autorização do FARBAC são armazenadas no banco de dados de regras de negócio de acordo com o modelo conceitual de entidades e relacionamentos apresentado na Figura 1 e descrito a seguir.

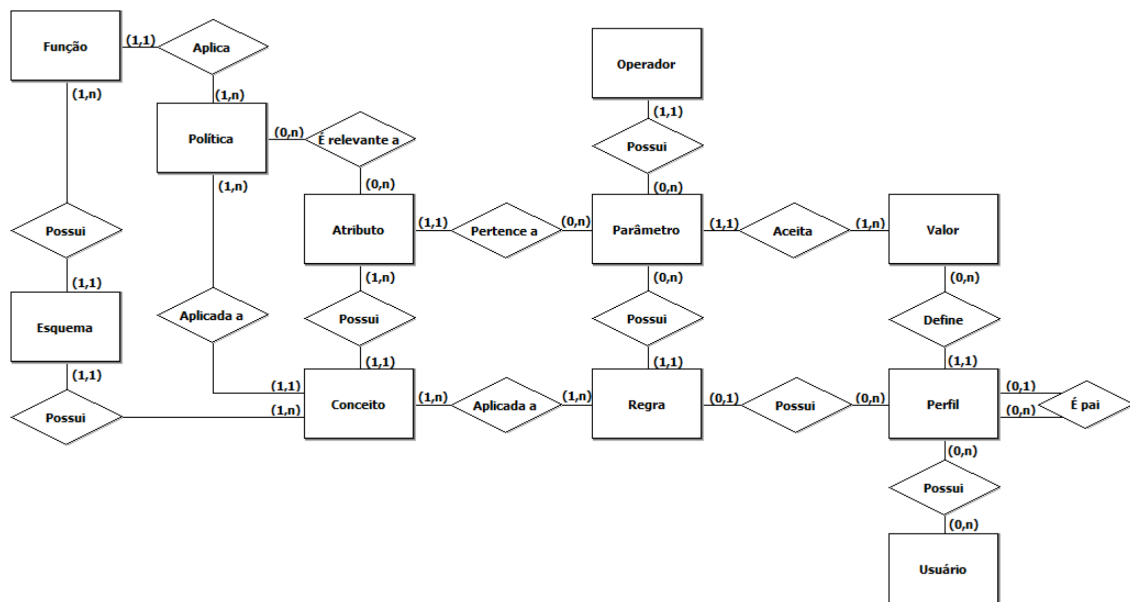

Figura 1- Modelo de entidade e relacionamento do FARBAC.

Os **Usuários** são agrupados em **Perfis**, onde cada usuário pode ser associado a diversos perfis, e cada perfil pode agrupar diversos usuários. O auto-relacionamento da entidade **Perfil** permite a representação de uma hierarquia de perfis. Por exemplo, os perfis de Gerente de Vendas da América e Ásia e de Gerente de Vendas da Europa podem ser especializações do perfil Gerente de Vendas. Isso significa que regras de autorização atribuídas a um perfil mais geral (Gerente de Vendas) devem também ser aplicadas a todos os seus descendentes. Além disso, perfis descendentes podem determinar valores específicos para os parâmetros do predicado. Por exemplo, o perfil Gerente de Vendas pode restringir o acesso aos pedidos baseado no atributo continenteDeOrigem; o perfil Gerente de Vendas da América e Ásia pode então especificar o(s) valor(es) para o atributo controlado (por exemplo, “America, Asia”). Dessa forma é possível que a política RBAC implemente

a regra, restringindo o acesso do perfil Gerente de Vendas América e Ásia somente para os pedidos onde “continenteDeOrigem in (‘America’,‘Asia’)”.

A entidade Regra representa as definições de regras que são armazenadas como cláusulas SQL, que serão usadas para transformar os comandos SQL provenientes da aplicação cliente. Uma cláusula restringe o acesso do Usuário a um subconjunto dos dados de um Conceito, que pode ser uma tabela, uma visão ou um sinônimo. Portanto, existe um relacionamento Regra × Conceito. A mesma cláusula SQL pode ser usada para definir regras de autorização em diferentes conceitos, e cada Conceito pode possuir mais de uma Regra.

Um Parâmetro denota uma expressão no formato "Atributo Operador Valor" relacionando as entidades correspondentes (Atributo × Operador × Valor). A entidade Atributo representa um atributo de um conceito do banco de dados utilizado para restrição de acesso. A entidade Operador representa um operador binário (incluindo =, <>, >, >=, <, <=, IN e NOT IN). A entidade Valor representa um valor (ou lista de valores) dentro do domínio do atributo, que delimitam um subconjunto de dados. O relacionamento Conceito × Atributo indica de que conceito um atributo pertence da mesma forma que o relacionamento Esquema × Conceito indica de que esquema o conceito pertence. Esse relacionamento denota uma representação em alto nível do esquema do banco de dados para o qual as regras de autorização são aplicadas.

A Política dita o controle de acesso que deve ser aplicado a um conceito. Uma política de controle de acesso é aplicada por uma Função (que também pertence a um Esquema). Durante sua execução, esta função identifica o Conceito sendo acessado e o Usuário executando o comando. Ela captura os dados das outras tabelas do modelo para construir o predicado da regra de autorização (ou predicado de autorização) daquele conceito para aquele usuário. Além disso, o relacionamento Política × Atributo determina quais atributos do conceito possuem dados sensíveis.

O mesmo usuário pode possuir mais de um perfil relacionado a um mesmo conceito. Dessa forma, em tempo de execução, mais de uma regra (predicado de autorização) será retornada e, portanto, devem ser compostas de alguma forma. Essa composição pode ser feita através do uso dos operadores OR ou AND. A composição por OR, que chamamos de composição permissiva, permite que o usuário tenha acesso à união dos subconjuntos de dados permitidos por cada regra. A composição por AND, que chamamos de composição restritiva, limita o acesso do usuário à interseção dos subconjuntos de dados permitidos por cada regra. Por exemplo, um usuário que possua os perfis Gerente de Vendas da Europa e Gerente de Vendas da França, com a composição permissiva teria acesso as dados de toda a Europa, já com a composição restritiva o seu acesso seria apenas aos dados da França. Tanto a composição permissiva quanto a composição restritiva são suportadas pelo FARBAC, e é dever da organização decidir que tipo usar.

2.3 Algoritmo para construção do predicado de autorização

O FARBAC possui uma função genérica responsável por construir, em tempo de execução, o predicado de autorização para cada conceito. O predicado de autorização é a composição (permissiva ou restritiva) de todas as regras de um usuário para um conceito. A função é aplicada a todos os conceitos com dados sensíveis e, de acordo com o usuário realizando o acesso e com os dados cadastrados no modelo de armazenamento de regras de autorização (Figura 2), retorna o predicado de autorização.

```
Recupera o usuário que está executando a consulta
A partir do relacionamento Usuário × Perfil, recupera os perfis do
usuário
Para cada perfil Faça
   A partir do relacionamento Perfil × Regra × Conceito, identifica
   as regras que devem ser aplicadas de acordo com o perfil pai da
   hierarquia
   Para cada regra Faça
      A partir do relacionamento Regra × Parâmetro, identifica os
      parâmetros específicos da regra
      Para cada parâmetro Faça
         A partir do relacionamento Perfil × Valor, recuperar o(s)
         valor(s) aceito(s) pelo parâmetro para filtragem dos
         dados
Finalmente, o predicado de autorização é montado e retornado
```

Figura 2 - Algoritmo de construção do predicado de autorização.

3 Elaboração de proposta de solução

Para elaborar uma solução genérica para a propagação de identidade que atenda ao cenário 3, foram combinados os padrões de projeto *Singleton* e *Service Locator* na aplicação cliente e os padrões *Singleton, HibernateUtil* e injeção de dependência (via *Setter*) na camada servidor. Nesta seção, estes conceitos são apresentados de forma resumida a fim de focar na proposta de solução. Maiores detalhes sobre estes conceitos são apresentados na Seção 5.

Foram realizados testes considerando o banco de dados prosposto pelo *benchmark* TPC-H (TPC Council, 2008), o qual consiste em uma especificação de grande relevância na indústria que simula um cenário de uma aplicação de apoio à decisão.

A arquitetura de controle de acesso foi projetada segundo o diagrama de classes apresentado na Figura 3. Na Aplicação Cliente encontram-se as classes: VendasBO que implementa a interface gráfica com o usuário e a invocação do objeto no Servidor de Aplicação; e a classe *Service Locator*[1] que é utilizada para acessar o objeto remoto. No Servidor de Aplicação encontram-se: a classe ManterVendas que será invocada pelo cliente; a interface ManterVendasRemote que é utilizada para o acesso remoto; a interface ControleAcesso, utilizada para transferir as informações do usuário; a classe VendasDAO que realiza o acesso ao banco de dados via sessão construída utilizando a classe *HibernateUtil*. No pacote correspondente ao SGBD estão caracterizadas a o banco de dados utilizado para a realização dos testes (TPC-H) e a autorização de acesso realizada pelo FARBAC.

m/technetwork/java/jndi/index.html.

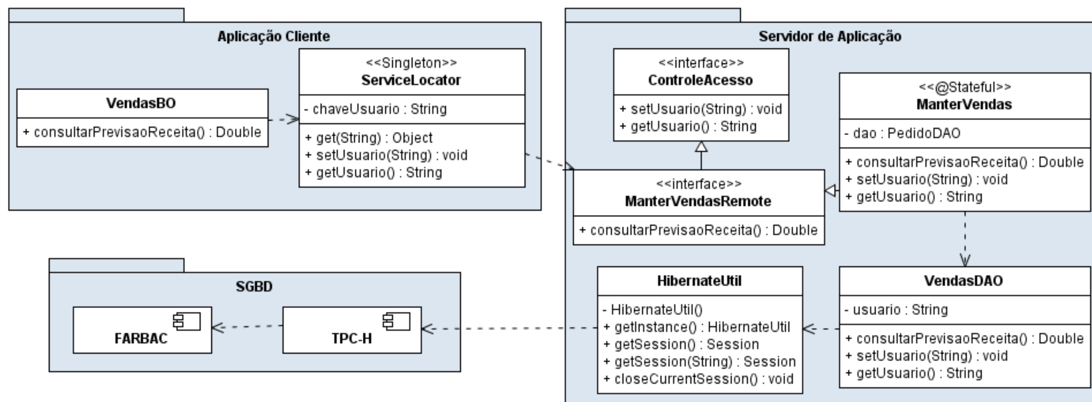

Figura 3 – Módulos da arquitetura de controle de acesso proposta

Para descrever a sequência de procedimentos que devem ser realizados pelo protótipo, foi elaborado o diagrama de sequências apresentado pela Figura 4.

Considerando o cenário de estudo, a proposta de solução foi dividida em aplicação cliente e módulo EJB, a fim de indicar o que foi implementado em cada caso para a propagação de identidade. A seção 3.1 apresenta a aplicação cliente, enquanto a seção 3.2 apresenta os detalhes na implementação do módulo EJB que executa no servidor de aplicação.

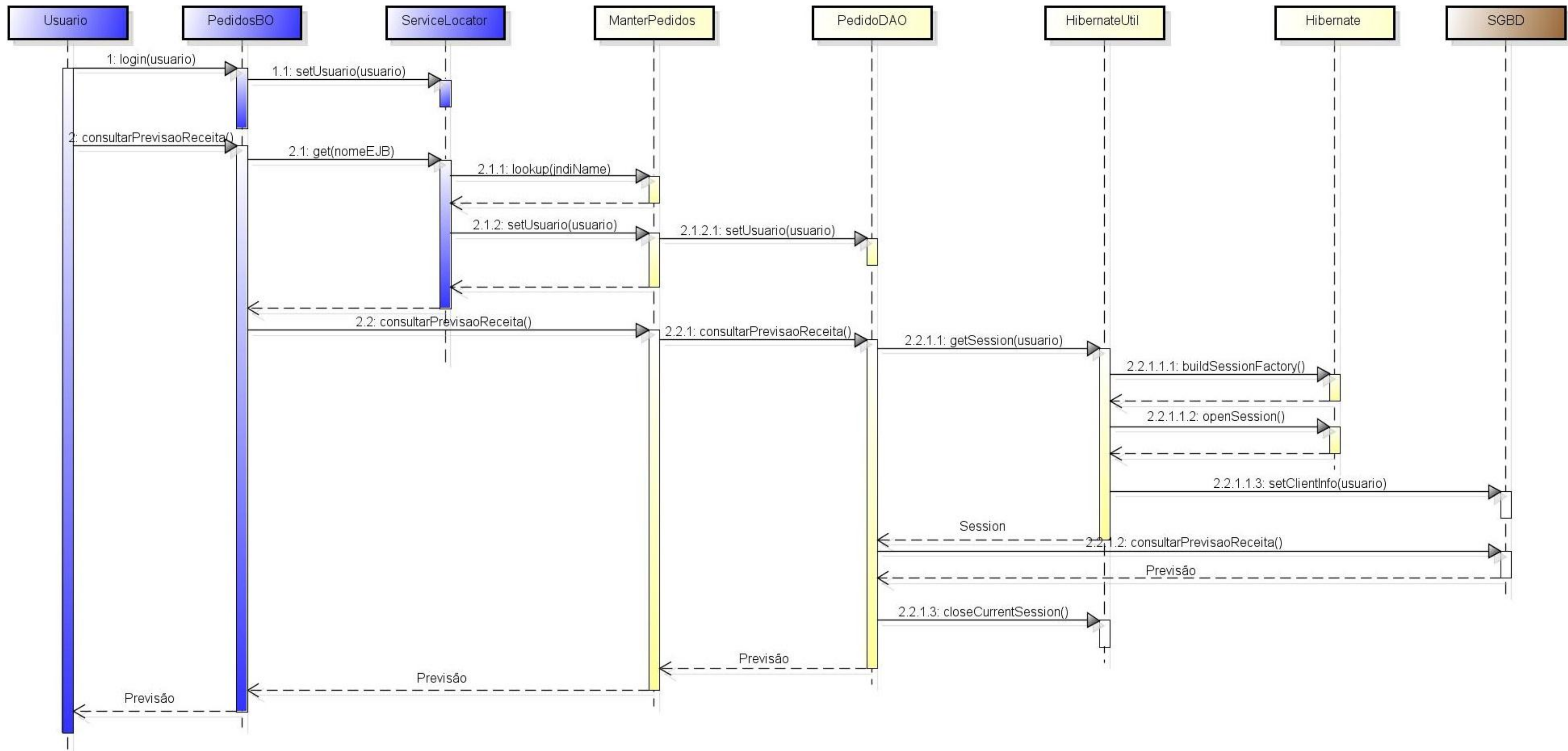

Figura 4 - Diagrama de Sequência descrevendo o protótipo proposto

3.1 Aplicação Cliente

Na arquitetura do protótipo implementado, a aplicação cliente utiliza a tecnologia Java *Swing* para solicitar a execução de uma consulta no banco de dados. O passo-a-passo de execução de uma requisição na aplicação Cliente é o seguinte:

1. Na aplicação cliente, o usuário informa a chave de acesso[2] (Figura 5).
2. O módulo cliente armazena a chave do usuário na classe *ServiceLocator*.
3. O usuário acessa a funcionalidade implementada, que foi a consulta à previsão de mudança na receita da empresa dado o corte de um desconto, através do botão *Buscar*.
4. A aplicação cliente acessa o servidor através do *Service Locator* para recuperar um componente EJB passando o nome JNDI[3] do EJB. O método *get*, apresentado na Figura 6, do *ServiceLocator* é invocado.
5. A classe *ServiceLocator*, executa o método *get* que verifica se o EJB recuperado implementa a interface *ControleAcesso*; em caso positivo, invoca o método "*setUsuario*" passando a chave armazenada e devolve o EJB à aplicação. A interface *ControleAcesso* é explicada na Seção 5.2.
6. A aplicação invoca o método de consulta do componente EJB.
7. A aplicação recebe o resultado da consulta.
8. A aplicação cliente apresenta as informações ao usuário.

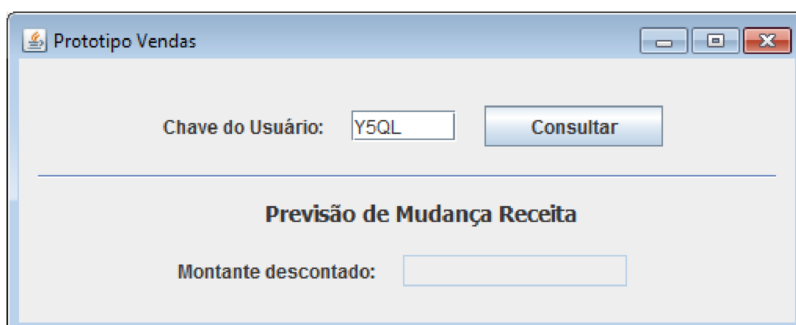

Figura 5 - Interface gráfica da Aplicação Cliente

```
51      public Object get(String jndiName) throws NamingException {
52          InitialContext ctx = recuperarContexto();
53          Object obj = ctx.lookup(jndiName);
54
55          if(obj instanceof ControleAcesso){
56              ControleAcesso controleAcesso = (ControleAcesso) obj;
57              controleAcesso.setUsuario(chaveUsuario);
58          }
59
60          return obj;
61      }
62  }
```

Figura 6 - Método get da classe EJBLocator responsável por recuperar um objeto EJB

---

m/technetwork/java/jndi/index.html.

m/technetwork/java/jndi/index.html.

no servidor de aplicação

3.2 Servidor de aplicação

No servidor de aplicação, as regras de negócio estão implementadas em projetos Java utilizando a tecnologia EJB (*Stateful*). Neste caso, temos uma classe Java EJB para cada projeto EJB. O acesso e persistência de dados foi implementado utilizando o framework de mapeamento objeto relacional Hibernate[4], acessando SGBD Oracle. A responsabilidade de gerenciar o ciclo de vida das conexões com o banco de dados (*pool* de conexão) e o contexto transacional é do servidor de aplicação JBoss[5].

Uma interface, chamada *ControleAcesso*, foi implementada para especificar os métodos que os EJB devem implementar para armazenamento das informações necessárias para controle de acesso aos dados. Esta interface é apresentada na Figura 7.

```
1  public interface ControleAcesso {
2      void setUsuario(String chaveUsuario);
3      String getUsuario();
4  }
```

Figura 7 - Implementação da Interface ControleAcesso

A interface *ControleAcesso* é parte de uma biblioteca independente, devendo ser importada para o classpath de cada projeto EJB antes de ser utilizada. Esta separação permite que posteriormente outros projetos possam utilizar a mesma interface ao implementar controle de acesso aos dados.

Em cada projeto EJB, é criada uma interface com o objetivo de expor os métodos que poderão ser acessados pelo cliente. Esta interface deve herdar a interface ControleAcesso, obrigando o EJB a implementar também os métodos *setUsuario* e *getUsuario* de ControleAcesso. O método *setUsuario* é utilizado pela aplicação cliente para ajustar no EJB as informações do usuário. O uso deste método segue o padrão de projeto Injeção de Dependência[6], que neste caso está sendo implementada via *setter* (*Setter Injection*). Injeção via *setter*, como o nome sugere, consiste em passar a um objeto suas dependências através de argumentos por um método setter [Prasanna, 2009].

O acesso ao banco de dados foi implementado utilizando o padrão HibernateUtil[7] sugerido pelos desenvolvedores do framework Hibernate, centralizando na aplicação a inicialização do Hibernate e o lookup à fábrica de sessões com o banco de dados. Foi considerada uma modificação nesta classe, onde, ao invés de a aplicação requisitar ao HibernateUtil uma fábrica de sessões, ela deverá requisitar diretamente uma sessão. O objetivo é informar ao SGBD o usuário

---

[6] Injeção de Dependência (do inglês *Dependency Injection*) é um padrão de projeto derivado do padrão Inversão de Controle, e assim como seu progenitor visa fornecer a uma classe informações necessárias para o seu funcionamento sem que a classe tenha que se preocupar com a geração destes dados.

m/technetwork/java/jndi/index.html.

[6] Injeção de Dependência (do inglês *Dependency Injection*) é um padrão de projeto derivado do padrão Inversão de Controle, e assim como seu progenitor visa fornecer a uma classe informações necessárias para o seu funcionamento sem que a classe tenha que se preocupar com a geração destes dados.

[7] A implementação base para a classe HibernateUtil é demonstrada na documentação do Hibernate e pode ser vista em http://community.jboss.org/wiki/sessionsandtransactions.

que irá realizar as operações no banco de dados. A Figura 8 apresenta a implementação do método *getSession*, da classe *HibernateUtil*, onde é criada uma sessão com o banco de dados e armazenam-se as informações do usuário neste banco antes de retornar o objeto de sessão para quem invocou o método. Neste caso, como o SGBD utilizado é o Oracle, o armazenamento das informações do usuário é feito através da chamada à procedure dbms_application_info.set_*client_info*. Esta chamada armazena a chave do usuário no atributo *client_info* do *namespace userenv* do contexto do banco de dados Oracle.

```
33      public Session getSession(String chaveUsuario) {
34          if (session.get() == null) {
35              session.set(sessionFactory.openSession());
36          }
37
38          //preparar o contexto da sessao no banco com a chave do usuario
39          Connection con = session.get().connection();
40          try{
41              CallableStatement st = con.prepareCall(
42                      "{call dbms_application_info.set_client_info(?)}");
43              st.setString(1, chaveUsuario);
44              st.executeUpdate();
45          }catch(SQLException e){
46              return null;
47          }
48
49          return session.get();
50      }
```

Figura 8 - Método getSession da classe HibernateUtil

O passo-a-passo de atendimento a uma requisição pelo EJB é o seguinte.

1. O EJB instancia um objeto DAO e armazena nele a chave do usuário recebida.
2. O EJB invoca o método do objeto DAO responsável por executar a operação no banco de dados.
3. O objeto DAO recupera a instância única da classe HibernateUtil e invoca o método *getSession* informando a chave do usuário.
4. A classe HibernateUtil cria uma sessão com o banco de dados e ajusta o usuário que está fazendo uso da sessão através da chamada à procedure *dbms_application_info.set_client_info*, então retorna o objeto de sessão.
5. A classe DAO executa a consulta ao banco de dados utilizando sessão retornada. Nesse momento, o FARBAC entra em ação para aplicar a regra de autorização e retornar apenas as informações que o usuário tem acesso.
6. O resultado da consulta é repassado para o EJB que repassa para a aplicação cliente.

4 Conclusão

Segurança da informação é uma questão importante para as organizações. Em geral questões desta área são resolvidas através da implementação de mecanismos de controle de acesso e da implementação de regras de autorização em Sistemas de Informação. Contudo, quando uma regra é alterada, todos os sistemas que implementam estas regras e controles devem ser atualizados, fazendo com que isso

se torne um problema bastante complexo principalmente em cenários onde existem sistemas legados e um número muito grande de regras de autorização.

Este trabalho apresentou uma proposta de implementação para uma arquitetura de controle de acesso através do uso de mecanismos de propagação de identidade e do uso de regras de autorização de informação. A implementação empregou uma série de tecnologias disponíveis e de amplo uso no mercado, fazendo com que possa ser reutilizada em outras instalações com o mesmo cenário. Testes experimentais foram realizados demonstrando a viabilidade e eficácia da proposta em um cenário que simula uma aplicação real de apoio à decisão.

A solução proposta se demonstrou ideal para um ambiente que faça uso de *beans* de sessão com estado, ou seja, EJBs *Stateful*. A proposta envolve o armazenamento de dados (a chave do usuário) no EJB utilizado. Logo, ao utilizar EJB *stateful*, o EJB instanciado não é compartilhado com nenhuma outra aplicação cliente. Garantindo isolamento na invocação dos métodos do EJB.

Este cenário não é o mesmo quando se utiliza EJB *stateless*. Neste caso, não é possível estabelecer um contexto privado entre a aplicação cliente e EJB, dado que, por natureza, o servidor de aplicação trabalha com uma única instância dos recursos do EJB, escalonando seu uso entre os clientes através de *threads* [Patrick et. al., 2009]. A chamada dos métodos do EJB é *thread safe*, ou seja, cada método do EJB invocado por uma aplicação cliente é totalmente isolado se o mesmo método for invocado por outra aplicação cliente. Neste caso, duas *threads* são abertas para tratar cada uma das requisições. No entanto, por exemplo, se uma aplicação cliente instanciar um EJB *stateless* e atualizar informações no mesmo (como a chave do usuário) e outra aplicação fizer o mesmo. As informações ajustadas inicialmente serão sobrescritas pelas informações ajustadas em seguida. Estas características foram avaliadas inclusive através de testes práticos.

Referências Bibliográficas

AZEVEDO, L.; DUARTE, D.; PUNTAR, S.; ROMEIRO, C.; BAIAO, F.; CAPPELLI, C. **Avaliação de Ferramentas para Gestão e Execução de Regras de Autorização.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0027/2009, 2009. Disponível (também) em: http://seer.unirio.br/index.php/monografiasppgi

AZEVEDO, L. G., PUNTAR, S., THIAGO, R., BAIÃO, F., CAPPELLI, C. **A Flexible Framework for Applying Data Access Authorization Business Rules.** In: 12th International Conference on Enterprise Information Systems, pp. 275-280, 2010a.

AZEVEDO, L. G.; PUNTAR, S., THIAGO, R., CAPPELLI, C., BAIÃO, F. **Avaliação Prática de Funcionalidades para Autorização de Informações (Label Security e Virtual Private Database)**, Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0002/2010, 2010b. Disponível (também) em: http://seer.unirio.br/index.php/monografiasppgi.

BANDY, T. R., BRADSHAW, D., RAJ, V. *et al.* **Oracle Application Server Concepts 10g Release 2 (10.1.2)**. Oracle Corparation, 2005.

BRG . **The Business Rules Group**. http://www.businessrulesgroup.org, 2009.

BURKE, B. e MONSON-HAEFEL, R. **Enterprise JavaBeans, 3.0.** O'Reilly, 2006, 760p. Bibliografia: ISBN-10: 0-596-00978-X.

CALÌ, A. E MARTINENGHI, D. **Querying data under access limitations**. In Proc. of ICDE, Cancun, 2008.

DESMOND, E., ABEDRABBO, F., BHOJ, P. **Oracle Identity and Access Management Introduction 10g (10.1.4.0.1).** Oracle Corporation, 2006.

HERTEL, C.R., 2003. **Implementing CIFS - The Common Internet FileSystem.** Disponível em http://ubiqx.org/cifs/. Acessado em 29 nov. 2010.

HUGHES, J., EVE, M., PHILPOTT, R., **Technical Overview of the OASIS Security Assertion Markup Language (SAML).** OASIS (Organization for the Advancement of Structured Information Standards) Standard, 2004**.**

JELOKA, S.; MULAGUND, G.; LEWIS N. *et al*. **Oracle Database 10gR2 Security Guide**. Oracle, 2008.

JOHNSON, R., LEE, B., 2009. **JSR-330: Dependency Injection for Java.** Disponível em http://jcp.org/en/jsr/detail?id=330. Acessado em 17 nov. 2010.

KING, G., 2009. **JSR-299: Contexts and Dependency Injection for the Java EE platform.** Disponível em http://jcp.org/en/jsr/detail?id=299. Acessado em 17 nov. 2010.

MURTHY, R., SEDLAR, E. **Flexible and efficient access control in oracle**. In *ACM SIGMOD 2007*, pp. 973-980, Beijing, 2007.

RAGOUZIS, N., HUGHES, J., EVE, M., PHILPOTT, R., **Security Assertion Markup Language(SAML) V2.0 Technical Overview.** OASIS (Organization for the Advancement of Structured Information Standards) Standard, 2006**.**

ORACLE, **Oracle Fusion Middleware Understanding Security for Oracle WebLogic Server 11g Release 1 (10.3.3).** Oracle Corporation, 2010.

ORACLE, **Oracle WebLogic Server Programming WebLogic Security, 10g Release 3 (10.3)**. Oracle Corporation, 2008.

PATEL, A., McROBERTS, M., CRENSHAW , M. **Identity Propagation in N-Tier Systems.** In: Military Communications Conference, Boston, MA, pp. 1-5, 2009.

PATRICK, R.; NYBERG, G.; ASTON, P. **Professional Oracle WebLogic Server**. 1st ed. Wrox, 2009.

PRASANNA, D.R. **Dependency Injection: Design Patterns Using Spring and Guice.** Manning, 2010

RAKSHIT, A., 2010. **Simplifying Dependency Injection.** Disponível em http://blog.architexa.com/2010/04/simplifying-dependency-injection/. Acessado em 17 nov. 2010.

SAMAR, V., **Single Sign-On Using Cookies for Web Applications.** In: **Proceedings** of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises, pp. 158-163, 1999.

SMITH, R., STEINER, D., VENNING S. **Oracle Internet Directory Administrator's Guide (Release 2.1.1).** Oracle Corporation, 2000.

ORACLE, **Oracle Web Services Manager, Administrator's Guide, 10g (10.1.3.1.0).** Oracle Corporation, 2006.

CANTOR, J., KEMP, J., PHILPOTT, R., MALER, E. **Assertions and Protocols for the OASIS, Security Assertion Markup Language (SAML) V2.0.** OASIS (Organization for the Advancement of Structured Information Standards) Standard, 2005.

SANDHU, R.S., COYNE, E.J., FEINSTEIN, H.L., YOUMAN, C.E. **Role-based access control models**. IEEE Computer, vol. 29, no. 2, pp 38-47, 1996.

SOLDANO, A., DIMITRIS, A., BURKE, B. *et al*. **JBOSS Application Server 4.2.2.** JBOSS.org Community Documentation, 2008.

TPCH. **TPC Benchmark H Standard Specification Revision 2.8.0**. *Transaction Processing Performance Council.* 2008. Disponível em <http://www.tpc.org/tpch/spec/tpch2.8.0.pdf>. Acessado em março de 2011.

WALLS, C. BREIDENBACH, R. **Spring in Action.** Manning, 2007.

YANG, L. **Teaching database security and auditing**. ACM SIGCSE'09, v.1, issue 1, pp. 241—245, 2009.

BAUER, C., KING, G., **Hibernate in Action**, Manning 2005.

KING, G., BAUER, C., BERNARD, E., EBERSOLE, S. **Hibernate Getting Started Guide**. Red Hat, Inc., 2010a.

KING, G., BAUER, C., ANDERSEN, M. R. *et al* **Hibernate Core Reference Manual**. Red Hat, Inc., 2010b.

PATRICIO, A., EBERSOLE, S. **Sessions and Transactions (Version 2)**. Hibernate Community, 2010.

Apendice A – Criação do protótipo para testes de propagação de identidade

Este tutorial tem como objetivo explicar o passo-a-passo de como foram criadas cada parte do protótipo utilizado para realizar os testes de Propagação de Identidade. Serão apresentadas a criação do projeto cliente (com o uso da API Swing), a criação do EJB invocado pelo cliente e as configurações necessárias para utilizar o Hibernate no acesso ao banco de dados relacional. Para a criação dos projetos foi utilizada a tecnologia Java e a IDE Netbeans (6.9.1). O banco de dados utilizados para os testes seguiu o esquema do *benchmark* TPC-H (TPC Council, 2008), o qual consiste em uma especificação de grande relevância na indústria que simula um cenário de uma aplicação de apoio à decisão.

Considerando as dependências entre cada uma das camadas que serão apresentadas, este tutorial irá seguir a abordagem *bottom-up*, contemplando primeiro a configuração do FARBAC para fornecer segurança à tabela de pedidos do TPC-H e finalizando com a criação da aplicação cliente que invocará o EJB.

O código apresentado neste anexo podem ser obtidos a partir do googlesource em http://code.google.com/p/pdac/, onde estão disponibilizados os projetos:

- EJB: correspondente ao EJB;
- Client: correspondente à aplicação cliente;
- Accesscontrol: correspondente à interface utilizada para indicar que uma classe está sobre controle de acesso.

A1. Configuração do FARBAC

A regra de autorização implementada no FARBAC foi: "O gerente de vendas do norte da América e da Ásia deve ter acesso somente aos pedidos provenientes de fornecedores das nações do hemisfério Norte das regiões Ásia e América". A implementação desta regra necessitou de algumas adaptações no modelo do TPC-H e no FARBAC, tais como: a chave do usuário na tabela *CUSTORMER* do TPCH referenciado a chave do usuário do FARBAC; inclusão de campo para nação do usuário no FARBAC; e, inclusão de campo para indicar o hemisfério do usuário.

Para implementar essa regra de autorização foi definida uma hierarquia de perfis, com um perfil pai chamado *Gerente de Vendas* e um perfil filho chamado *Gerente de Vendas Norte América e Ásia*. As informações que devem ser protegidas por essa política são armazenadas nas tabelas *ORDERS* e *LINEITEM*, portanto, foi criado um predicado para cada uma dessas tabelas.

Foram criados os usuários: *Y5QL* como gerente de vendas norte da América e da Ásia; e *Y2R7*, que é o presidente da empresa e recebeu o privilégio *EXEMPT ACCESS POLICY*, que faz com que todas as políticas sejam ignoradas.

Desta forma, quando a consulta exibida pela Figura 9 é executada pelo usuário Y2R7 (através do acesso ao EJB), nenhuma alteração é realizada, já ao ser executada pelo usuário Y5QL um novo predicado é adicionado para restringir os dados aos quais o usuário tem acesso. O predicado adicionado pode ser visto na Figura 10.

```
select sum(l_extendedprice * l_discount) as revenue
from lineitem
where l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1994-01-01' + interval '1' year
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24;
```

Figura 9 - Consulta realizada pelo EJB

```
(O_CUSTKEY in (
    select C_CUSTKEY from TPCH.CUSTOMER
        inner join TPCH.NATION on N_NATIONKEY = C_NATIONKEY
        inner join TPCH.REGION on R_REGIONKEY = N_REGIONKEY
    where R_NAME IN ('AMERICA' , 'ASIA') AND N_HEMISPHERE IN ('NORTH')
)
```

Figura 10 - Predicado gerado pelo FARBAC quando o protótipo é acessado pelo usuário Y5QL

A2. Criação da Interface para Controle de Acesso

Para estabelecer os métodos que deverão ser invocados pela aplicação cliente a fim de informar ao EJB a chave do usuário que está realizando o acesso, foi criada uma interface, nomeada *ControleAcesso* (Figura 11), contendo dois métodos básicos: *setUsuario(String chaveUsuario)* e *getUsuario()*, sendo o primeiro responsável por receber uma chave e armazená-la no EJB e o segundo apenas por retornar tal chave à aplicação, caso necessário.

```
public interface ControleAcesso {
    void setUsuario(String chaveUsuario);
    String getUsuario();
}
```

Figura 11 - Implementação da interface ControleAcesso

Esta interface deve ser construída em um projeto separado dos projetos da aplicação cliente e do EJB. A princípio, caso a interface pertencesse ao módulo EJB, poderia ser possível à aplicação cliente reconhecê-la, já que seria possível criar um *stub* desta interface e informá-la ao cliente. Entretanto, como será visto adiante, a aplicação cliente irá checar todos os EJBs invocados para verificar se a interface é implementada por eles, e para que outros EJBs possam utilizar o controle de acesso esta separação de projetos se faz necessária.

A3. Criação do EJB para Acesso aos Dados

Para criar e programar o EJB, foi utilizada a IDE de programação Netbeans (Versão 6.9.1).

A3.1. Criação do Projeto

Inicialmente deve-se criar um novo projeto do tipo EJB, através da opção "Arquivo > Novo Projeto". Na janela exibida (Figura 12), selecione a categoria "Java EE" e o tipo de projeto "Módulo EJB".

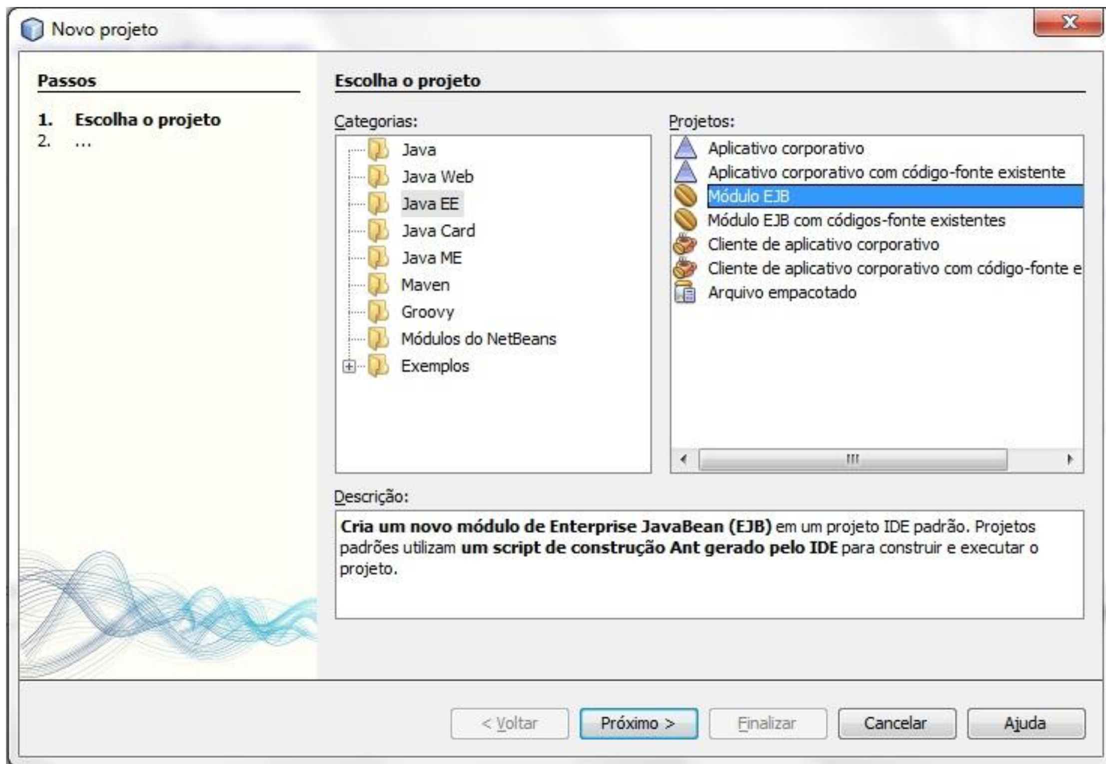

Figura 12 - Criação de um novo projeto EJB

Em seguida, escolha o nome do projeto que será criado e o seu local no computador (Figura 13).

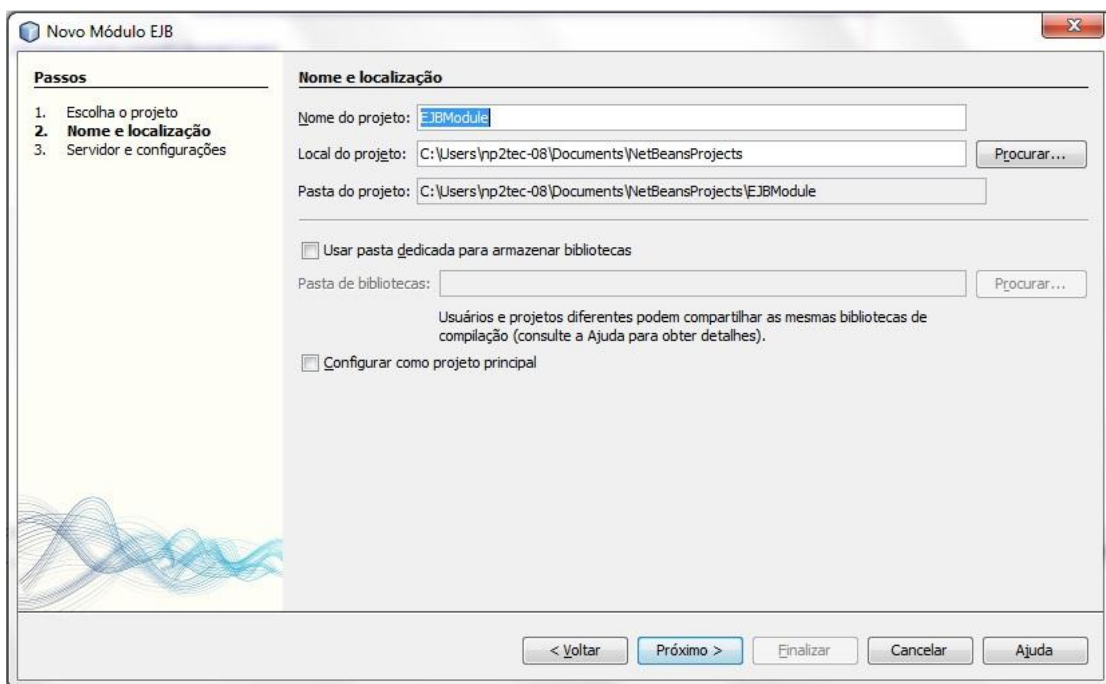

Figura 13 - Nomeação do novo projeto EJB

A seguir, escolha o servidor de aplicação (Figura 14). A fim de simular o ambiente real, foram utilizados, neste protótipo, o servidor de aplicação JBOSS na sua versão 4.2.0 e a versão 5 do JavaEE.

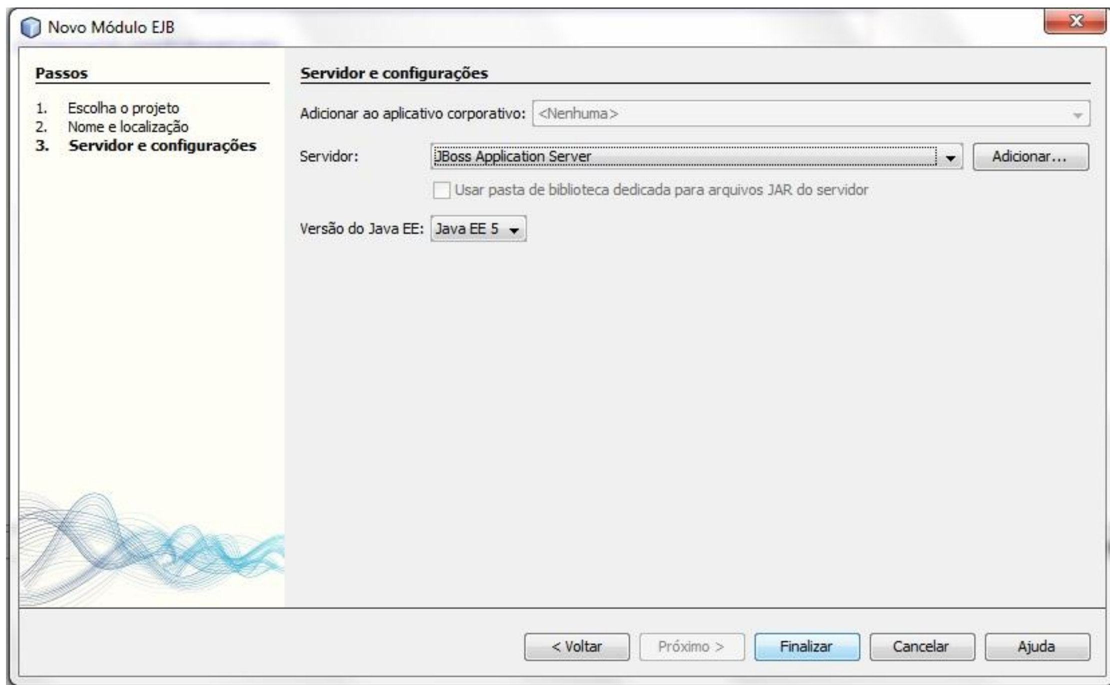

Figura 14 - Escolha do Servidor de aplicação do Projeto EJB

Caso o download do servidor de aplicação JBOSS já tenha sido feito, e sua extração para o diretório "C:" realizada, e mesmo assim o servidor não apareça na lista de servidores disponíveis, basta clicar no botão "Adicionar", selecionar a opção "Jboss Aplication Server" e na tela seguinte apontar o caminho e o nome do diretório onde se encontra o servidor.

Com o projeto criado, o próximo passo, é estabelecer a estrutura de pacotes demonstrada na Figura 15. Foram criadas as classes "ManterVendas" e "ManterVendasRemote" dentro do pacote "controle", as classes "VendasDAO" e "HibernateUtil" foram criadas no pacote "modelo.dao". Na raiz do projeto (para o Netbeans o <Pacote Padrão>) foram criados os arquivos de configuração "hibernate.cfg.xml" e "log4j.properties".

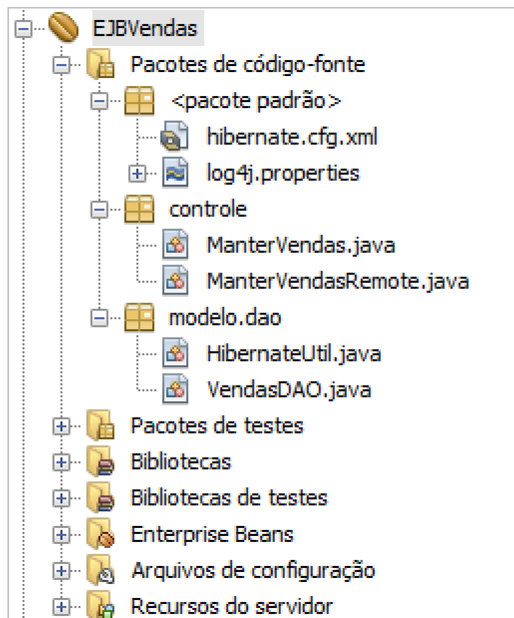

Figura 15 - Estrutura de Pacotes do EJB

A3.2. Implementação das classes do projeto e configuração do Hibernate

Ao criar um projeto "Módulo EJB" o usuário poderá pedir à própria IDE que crie um bean de sessão, para tal basta clicar com o botão direito em cima do pacote desejado (em nosso caso o pacote "controle") e seguir o menu "novo" e "Bean de Sessão". O Netbeans irá abrir um *wizard* que facilitará a criação do *bean* e pode ser vistona Figura 16.

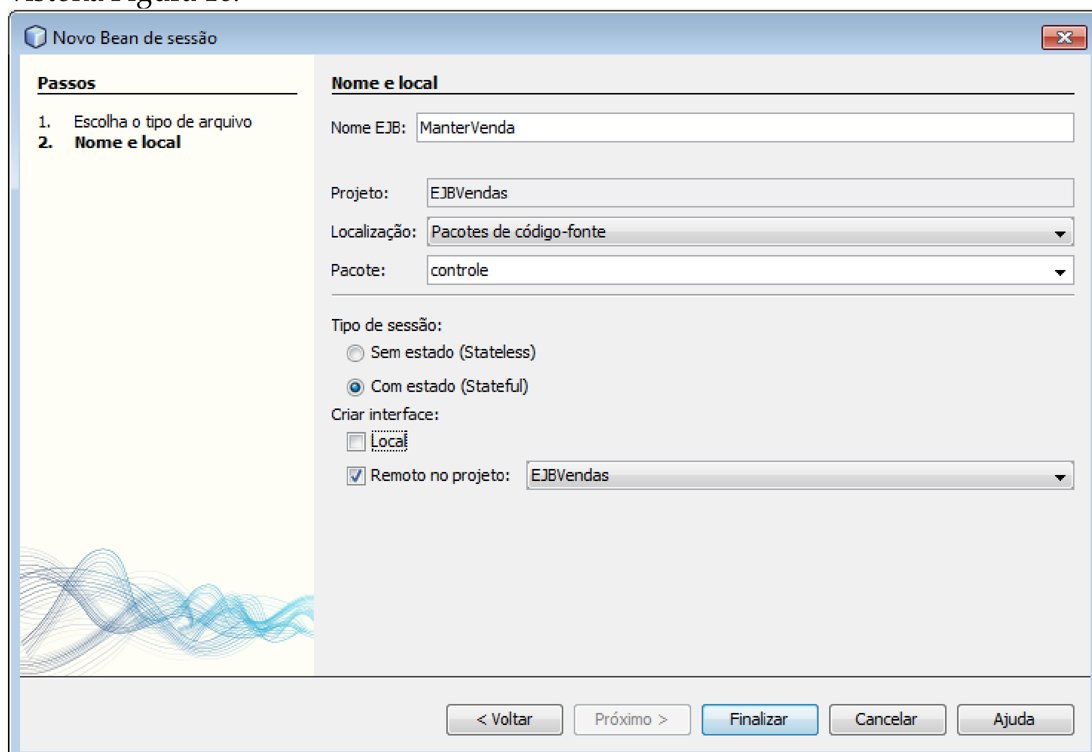

Figura 16 - *Wizard* para criação de um Bean de Sessão

Atente para o tipo de sessão do bean (que deve ser preenchido como Stateful) e quais interfaces devem ser criadas (somente a remota, pois o bean será acessado

apenas remotamente). O wizard irá gerar uma interface remota e uma classe que implementará o bean de sessão no pacote "controle", enquanto a interface deverá ter a assinatura dos métodos implementados pelo EJB o bean de sessão comportará a lógica de negócio.

Neste ponto do projeto será necessário adicionar ao classpath do projeto o arquivo JAR gerado na construção do projeto ControleAcesso, contendo a interface implementada na seção A1 deste tutorial. Para importar o JAR para o classpath do projeto basta acessar as propriedades do projeto e, na opção "Bibliotecas", indicar o local onde se encontra o JAR.

A interface gerada pelo wizard deverá estender a interface fornecida pelo JAR ControleAcesso, como visto na Figura 17.

```
1   package controle;
2
3   import br.np2tec.propid.controleacesso.ControleAcesso;
4   import java.sql.SQLException;
5   import javax.ejb.Remote;
6
    @Remote
8   public interface ManterVendasRemote extends ControleAcesso{
        double consultarPrevisaoMudancaReceita() throws SQLException;
10  }
```

Figura 17 - Interface do EJB

Uma vez estendendo a interface ControleAcesso, se tornará obrigatório ao bean de sessão implementar os métodos setUsuario() e getUsuario().

O bean de sessão (Classe "ManterVendas") deverá então implementar três métodos: setUsuario*()*, *getUsuario()* (da interface ControleAcesso) e um método para realizar a consulta aos dados (método *consultarPrevisaoMudancaReceita*, da interface ManterVendasRemote). A implementação básica do bean de sessão pode ser vista na Figura 18.

```java
@Stateful
public class ManterVendas implements ManterVendasRemote {
    VendasDAO dao = new VendasDAO();

    public double consultarPrevisaoMudancaReceita() throws SQLException {
        try{
            return dao.consultarPrevisaoMudanca();
        }catch(Exception e){
            throw new SQLException("Erro ao realizar consulta,");
        }
    }

    public void setUsuario(String chaveUsuario) {
        dao.setUsuario(chaveUsuario);
    }

    public String getUsuario() {
        return dao.getUsuario();
    }
}
```

Figura 18 - Implementação da classe ManterVendas

Para facilitar o acesso aos métodos de consulta ao banco de dados, deverá ser criada uma classe VendasDAO, no pacote "modelo.dao", contendo por exemplo o método consultarPrevisaoMudanca(). Esta classe deverá ser instanciada uma única vez pelo Bean de sessão.

A classe DAO deve ter também um atributo onde será armazenada a chave do usuário fornecida pelo método *setUsuario*. O Bean de sessão, ao ter seu método para ajustar o usuário invocado, deverá informar à classe DAO a chave do usuário. A implementação da classe VendasDAO, com o método responsável pela consulta ao banco de dados, pode ser vista na Figura 19.

```
public class VendasDAO {
    private String chaveUsuario;

    public double consultarPrevisaoMudanca() throws Exception{
        String sql = "select sum(l_extendedprice * l_discount) as revenue "+
                     "from tpch.lineitem "+
                     "where l_shipdate >= date '1994-01-01' "+
                      "and l_shipdate < date '1994-01-01' + interval '1' year "+
                      "and l_discount between 0.06 - 0.01 and 0.06 + 0.01 "+
                      "and l_quantity < 24";

        Session s = HibernateUtil.getSession(chaveUsuario);
        Connection con = s.connection();
        PreparedStatement ps = con.prepareStatement(sql);
        ResultSet rs = ps.executeQuery();
        HibernateUtil.closeCurrentSession();

        double resultado = 0;
        while(rs.next()){
            resultado = rs.getDouble("revenue");
        }
        return resultado;
    }

    public String getUsuario(){
        return this.chaveUsuario;
    }

    public void setUsuario(String chaveusuario){
        this.chaveUsuario = chaveusuario;
    }
```

Figura 19 - Trecho da implemenação da classe VendasDAO

A classe DAO faz uso do framework Hibernate para se conectar ao banco de dados, e executará uma consulta SQL através de uma sessão fornecida pela classe HibernateUtil.

Para configurar o Hibernate[8], após extraído para o computador, localize os arquivos "hibernate3.jar", "log4j-1.2.12.jar", "slf4j-log4j12-1.5.8.jar", "antlr-2.7.6.jar", "commons-collections-3.1.jar", "dom4j-1.6.1.jar", "javassist-3.9.0.GA.jar", "jta-1.1.jar" e "slf4j-api-1.5.8.jar" e importe-os para as bibliotecas do projeto. A maior parte dos arquivos está ou na raiz do diretório extraído do Hibernate ou no diretório "lib/required". Pode ser necessário baixar alguns dos arquivos da internet[9]. Os arquivos também foram disponibilizados no projeto http://code.google.com/p/pdac/ no googlesource. Nas "Propriedades" do projeto, em "Bibliotecas", adicione os diretório que contém os arquivos jar ("Adicionar JAR/Pasta") (Figura 20).

[8] Para baixar o Hibernate acessar http://sourceforge.net/projects/hibernate/files/hibernate3/

[9] Para localizar arquivos .JAR pode ser utilizado o site http://www.findjar.com, que disponibiliza diversos .JAR para download em suas mais variadas versões.

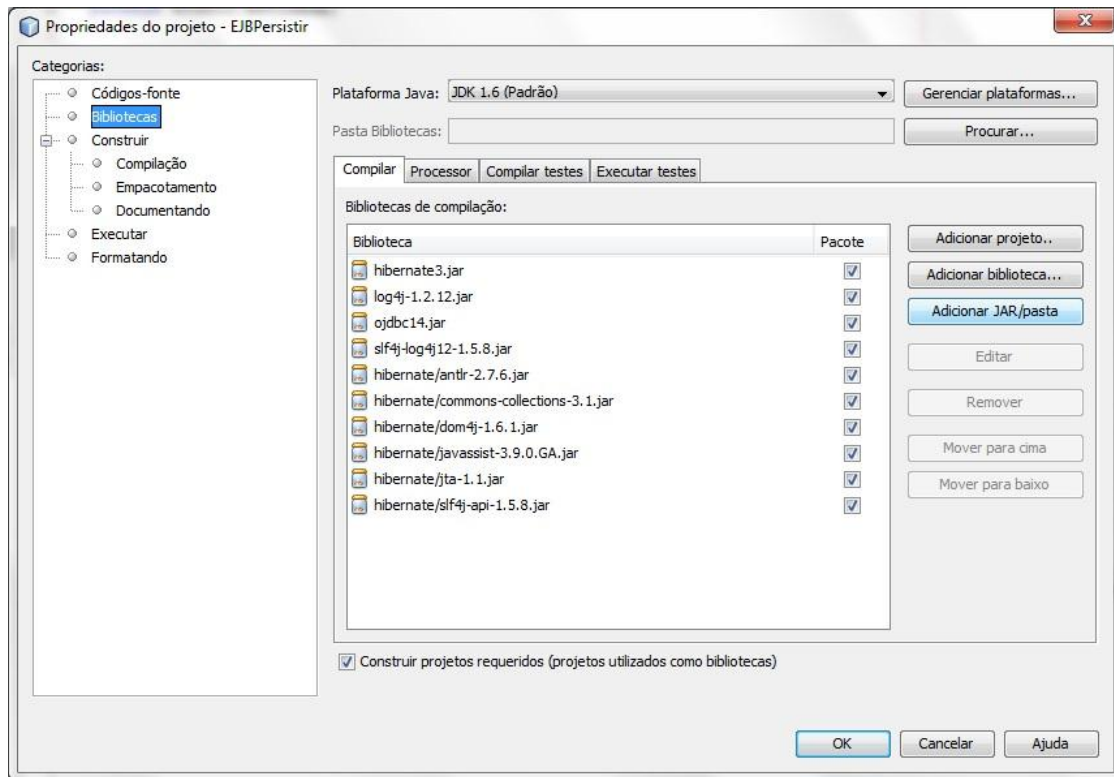

Figura 20 - Importando JAR para o projeto

Com os arquivos necessários importados, é preciso então informar ao hibernate alguns dados como o método de conexão com o banco de dados e a forma de mapear classes para tabelas do banco, entre outros.

Devem ser criados na raiz do projeto os arquivos “hibernate.cfg.xml” e “log4j.properties” (isto é, o <pacote padrão>), o código que deve ser incluído nos dois arquivos pode ser encontrado no código da aplicação disponibilizada googlesource (http://code.google.com/p/pdac/).

Observe que o arquivo “hibernate.cfg.xml” é responsável pela conexão com o banco de dados e por informar ao hibernate a localização dos arquivos de mapeamento objeto-relacional. Neste ponto será necessário ter importado nas bibliotecas do projeto o driver do banco de dados utilizado. O banco utilizado para a criação do protótipo foi o Oracle 10g R2, sendo então utilizado o driver “ojdbc14.jar” e o dialeto de conexão “org.hibernate.dialect.Oracle10gDialect”.

A fim de centralizar a inicialização do Hibernate, será utilizado o padrão HibernateUtil[10], sugerido pelos desenvolvedores do framework Hibernate. Este padrão serve para fornecer à toda a aplicação uma fábrica de sessões com o banco de dados. Entretanto nesta solução, é proposta uma modificação (já utilizada por alguns programadores) ao HibernateUtil. Quando a aplicação desejar uma sessão com o banco de dados, ao invés de requisitar a fábrica de sessões ao HibernateUtil, ela deverá requisitar diretamente uma sessão. Esta modificação é implementada através do método getSession().

O método getSes*sion* por sua vez é sobreescrito com uma implementação que possui em sua assinatura a passagem de uma String (a chave do usuário). Sempre

---

[10] A implementação base para a classe HibernateUtil é demonstrada na documentação do Hibernate e pode ser vista em http://community.jboss.org/wiki/sessionsandtransactions.

que a classe DAO necessitar de uma sessão com o banco de dados fará a chamada ao método *getSession* da classe HibernateUtil informando a chave do usuário que está realizando a consulta. O método irá criar a sessão e ajustar o contexto do usuário no banco de dados através da chamada ao método *dbms_application_info.set_client_info*, indicando qual usuário está executando operações no banco de dados. As duas implementações do método *getSession* podem ser vistas na Figura 21. Observe que a primeira implementação tem o objetivo de manter compatibilidade com aplicações em que não há controle de acesso, ou seja, não precisam que a sessão seja criada para um usuário específico.

```
public static Session getSession(){
    return getSession("");
}

public static Session getSession(String chaveUsuario) {

    if (sessions.get() == null) {
       sessions.set(sessionFactory.openSession());

        //preparar o contexto da sessao no banco com a chave do usuario
        Connection con = sessions.get().connection();
        try{
            CallableStatement st = con.prepareCall(
                    "{call dbms_application_info.set_client_info(?)}");
            st.setString(1, chaveUsuario);
            st.executeUpdate();
        }catch(SQLException e){
            return null;
        }
    }

    return sessions.get();
}
```

Figura 21 - Implementações do método getSession da classe HibernateUtil

Com o EJB implementado para ser disponibilizado no servidor de Aplicação JBOSS, o próximo passo é implementar o cliente que fará uso de seus métodos, o que é apresentado na próxima seção.

A4. Aplicação Cliente

O cliente criado no protótipo faz uso da API Swing para auxiliar na interação entre sistema e usuário.A aplicação cliente foi criada como um projeto no Netbeans. Ela foi criada como um aplicativo Java (Figura 22).

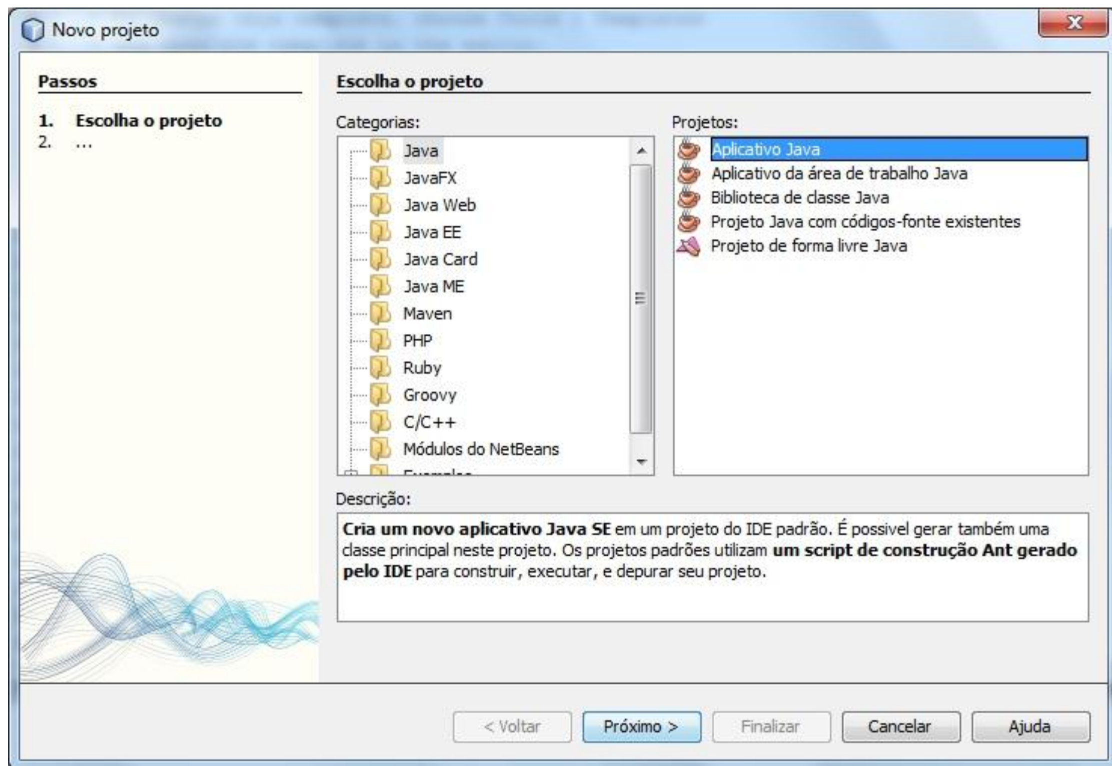

Figura 22 - Criação do Projeto Cliente

Na aplicação cliente, foram criadas as classes "ServiceLocator" e "Main" no pacote "default" do projeto. Além disso, foi criado um Formulário Jframe chamado "ClienteGUI". A classe Main (Figura 24) é responsável por inicializar o programa chamando a classe "ClienteGUI", que fará a interface gráfica com o usuário. A classe "Main" pode ser vista na Figura 24. A estrutura de classes e pacote que deve ser criada pode ser vista na Figura 23.

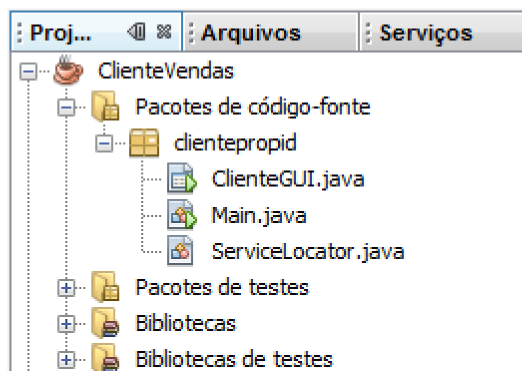

Figura 23 - Estrutura de pacotes de classes do projeto cliente

```
12  public class Main {
13      public static void main(String[] args) {
14          ClienteGUI.main(args);
15      }
16  }
```

Figura 24 - Método "main" da classe "Main" inicializando interface gráfica

Na classe "ClienteGUI", foram criados 2 campos textos (sendo 1 habilitado para inserção e 1 desabilitado), 3 rótulos (sendo um para cada campo de texto e um

somente para indicar o local onde serão exibidos os dados de retorno) e um botão para realizar a chamada ao método de consulta do EJB. A Figura 25 mostra o que se tem como resultado visual da montagem da tela.

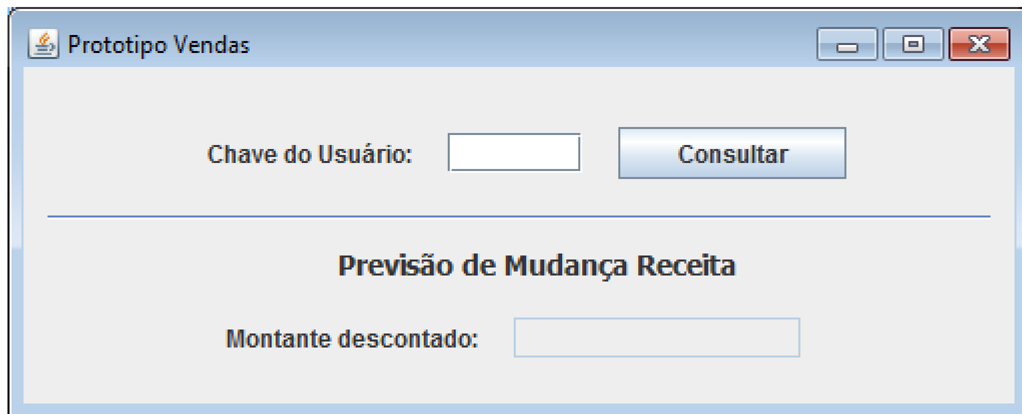

Figura 25 - Interface do protótipo

Os 2 campos de texto foram nomeados de acordo com os dados que comportarão. O campo que deve ser preenchido pelo usuário (Chave do usuário) recebeu o nome “jTextChaveUsuario”. Já o campo responsávei por exibir o resultado da consulta ao EJB (Montante Descontado) recebeu o nome “jTextMudancaReceita”.

Antes de programar a chamada ao método responsável pela consulta ao EJB, deverão ser implementadas a classe “Util” e o relacionamento entre o cliente e o EJB, além de adicionar ao classpath do projeto, os arquivos necessários para acessar um EJB rodando em um servidor JBOSS.

O relacionamento entre cliente e EJB pode ser feito de duas formas. Uma delas é a inclusão do arquivo “.jar” gerado na implantação do EJB ao classpath do projeto cliente. Para isto, basta acessar as propriedades do projeto cliente e na opções “Bibliotecas”, na aba “Compilar” clicar no botão “Adicionar JAR/Pasta” e então selecionar o “.jar” desejado. O .jar implantado no JBOSS estará localizado em [<root>\server\default\deploy] (considerando <root> como o diretório de instalação do servidor JBOSS). Caso o projeto EJB esteja no mesmo computador que o projeto cliente, pode-se ainda utilizar a importação do projeto como um todo. Para isto, a partir da opção “Bibliotecas” nas propriedades do projeto cliente o botão “Adicionar projeto...” e preencha com a localização do projeto no computador. A segunda opção pode ser vantajosa para testar o protótipo inicialmente, já que durante a programação alguns enganos podem ser cometidos e alterar o projeto EJB acarretará na sua reimplantação e na reimportação do arquivo .”jar” que lhe diz respeito. No entanto, para o acesso remoto em produção, a melhor opção é a primeira opção apresentada.

Para que o cliente consiga acessar uma aplicação rodando em um servidor JBOSS é necessário que os arquivos “jboss-aspect-jdk50-client.jar”, “jboss-client.jar”, “jboss-ejb3-client.jar” e “jbossall-client.jar” sejam adicionados ao seu classpath. Estes arquivos podem ser encontrados no subdiretório “client” existente no diretório de instalação do servidor JBOSS.

A classe “ServiceLocator” é implementada seguindo o padrão de projeto Singleton, portanto possui seu construtor com escopo privado e um método getInstance(). O ServiceLocator possui ainda o métodos setUsuario(), responsável por armazenar na única instância da classe ServiceLocator a chave de um usuário, o

método getUsuario(), o método recuperarContexto(), responsável por recuperar o contexto JNDI do servidor de aplicação e um método *get()* para logalizar um EJB dado seu nome na JNDI e fazer a verificação se ele implementa a interface *ControleAcesso*. Caso o EJB recuperado implemente a interface ControleAcesso o método *get()* fará a chamada ao método que passa a chave do usuário ao EJB antes de devolvê-lo à aplicação. Um trecho da classe ServiceLocator, contendo o método *get()* pode ser visto na Figura 26.

```
public Object get(String jndiName) throws NamingException {
    InitialContext ctx = recuperarContexto();
    Object obj = ctx.lookup(jndiName);

    if(obj instanceof ControleAcesso){
        ControleAcesso controleAcesso = (ControleAcesso) obj;
        controleAcesso.setUsuario(chaveUsuario);
    }

    return obj;
}
```

Figura 26 - Classe ServiceLocator

Como citado anteriormente, para que o cliente seja capaz de reconhecer se um EJB recuperado implementa a interface ControleAcesso é necessário que o JAR gerado no projeto ControleAcesso, apresentado na seção A2, tenha sido importado para o classpath do projeto.

É necessário ainda realizar a conexão entre a interface gráfica do protótipo e a funcionalidade implementada. Novamente na interface do protótipo dê dois cliques no botão responsável por realizar a consulta. O Netbeans abrirá o código fonte da tela no exato ponto responsável pela ação executada pelo botão. Nele, deverá ser instanciado um objeto ServiceLocator, informada a chave do usuário à instância única do ServiceLocator e recuperado o EJB desejado através do método *get()*,. Em seguida, invocar o método de consulta do EJB (Figura 27).

```
private void jBtnConsultarChaveActionPerformed(java.awt.event.ActionEvent evt) {
    ServiceLocator.getInstance().setUsuario(jTextChaveUsuario.getText());

    try{
        //Recuperar o EJB
        ManterVendasRemote ejb = (ManterVendasRemote)
                ServiceLocator.getInstance().get("ManterVendas/remote");
        //Interromper a execução
        showMessage("Execução interrompida. Clique para continuar.");
        //Recuperar Valor de descontos dados
        double retorno = ejb.consultarPrevisaoMudancaReceita();
        //apresentar o resultado da consulta
        jTextMudancaReceita.setText(String.valueOf(retorno));

    }catch(Exception e){
        e.printStackTrace();
        showMessage("Erro ao consultar Poço: " + e.getMessage());
    }
```

Figura 27 - Ação do botão Consultar

Com o valor retornado pelo método, o campo responsável por exibir os dados de resposta da consulta pode ser preenchido (Figura 27). Caso uma exceção ocorra no

acesso ao EJB, ou internamente no EJB, o protótipo deve ser capaz de capturar a falha e mostrar uma janela de erro ao usuário. Para isto, foi criado o método "showMessage", cuja implementação é exibida na Figura 28. O método é chamado pelo bloco "*Catch(Exception e)*" mostrado na figura (Figura 27).

```
private void showMessage(String message){
    JOptionPane.showMessageDialog(this, message,
            this.getTitle(), JOptionPane.ERROR_MESSAGE);
}
```

Figura 28 - Implementação do método para exibir mensagens ao usuário

A5. Executando o protótipo

Com o banco de dados criado, o EJB implantado, o Hibernate configurado e o cliente corretamente codificado, o ambiente pode ser testado.

Com o banco de dados ativo e populado, então implante o EJB no servidor de aplicação JBOSS (Figura 29).

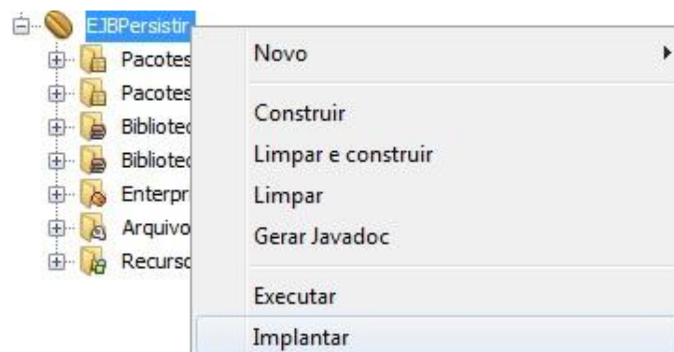

Figura 29 - Implantação do EJB no servidor de aplicação

Após a implantação do EJB, o projeto cliente deverá ser executado. Com o protótipo rodando basta preencher a chave do usuário e observar o retorno dos dados (Figura 30).

Como Felipe (chave Y5QL) é o gerente de vendas norte da América e da Ásia, ele tem acesso apenas aos itens de pedido do norte dessas regiões, enquanto Leonardo (Y2R7), que é o presidente, tem acesso a todos os itens de pedido. Para Leonardo, a mudança na receita foi de R$ *88.894.386,60*. Já para *Felipe* esta é de apenas R$ *21.445.915,50*, afinal a sua visão é somente dos itens de pedido da América e da Ásia.

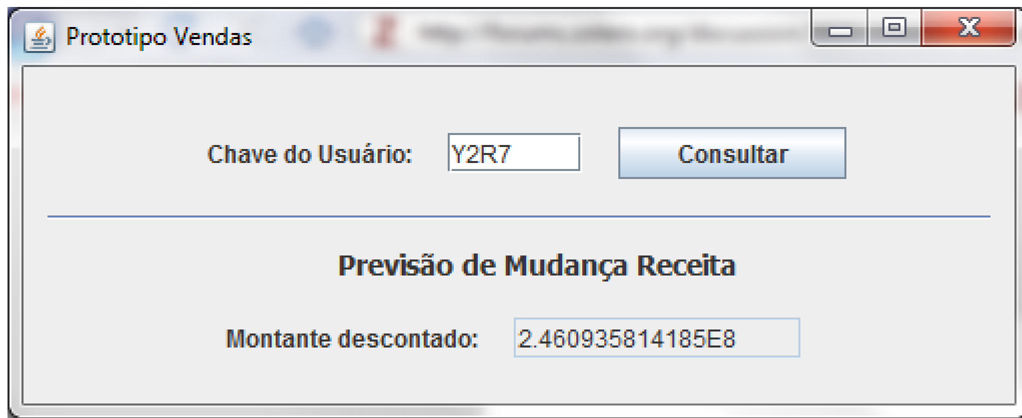

Figura 30 - Protótipo em execução

Múltiplas instâncias do protótipo podem ser executadas simultaneamente para demonstrar que mesmo que vários usuários estejam acessando o servidor de aplicação não haverá problemas de conflitos na propagação da identidade de cada um destes usuários.

Para executar este teste, basta abrir dois protótipos informando em cada um deles uma chave de usuário, uma pertencente a um usuário com acesso restrito e outra pertencente a um usuário com acesso irrestrito aos dados. Clique no botão para ativar a consulta de um dos protótipos e aguarde até que a tela de interrupção do programa (Figura 31) seja exibida. Antes de continuar com o processamento do protótipo clique no botão para ativar a consulta no segundo protótipo. Quando as duas telas de interrupção tiverem sido exibidas clique em ambas para continuar a execução dos dois protótipos. Em alguns instantes a consulta será realizada e cada protótipo mostrará o seu resultado, demonstrando o isolamento entre as múltiplas instâncias e a segurança da propagação estabelecida.

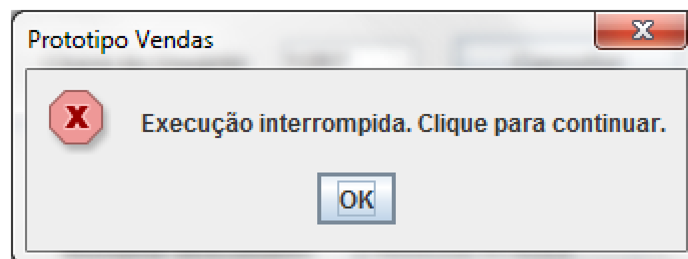

Figura 31 - Tela de Interrupção do Programa Cliente

Apêndice B – Tecnologias e padrões relacionados

Este apêndice apresenta características de tecnologias e padrões utilizados na solução proposta neste trabalho.

B1. JBoss security

O servidor de aplicação JBOSS é desenvolvido como projeto *open source* administrado pela JBOSS.inc[11], uma subdivisão da empresa Red Hat[12].

A versão 4.2.2 do JBOSS Application Server possui métodos para aumentar a segurança no acesso às aplicações web sem a necessidade de implementar detalhes de segurança diretamente nos componentes de negócio. Esta implementação é feita através da declaração de Security Roles e permissões em um descritor XML padrão. Isto isola a segurança do código de nível de negócio, dado que segurança tende a ser mais uma função de onde o componente é implantado do que um aspecto herdado da lógica de negócio do componente [Soldano *et al.*, 2008].

A atividade de assegurar uma aplicação J2EE[13] é baseada nas especificações dos requerimentos de segurança de aplicação via descritores de implantação do J2EE (J2EE Deployment Descriptors). O acesso à EJBs e componentes web em uma aplicação empresarial é assegurado utilizando os descritores de implantação ejb-jar.xml e web.xml [Soldano *et al.*, 2008].

A seguir é descrito o objetivo e formas de utilizar alguns dos vários elementos de segurança disponíveis.

B1.2. Referências de Segurança

Tanto EJBs quanto *Servlets* podem declarar um ou mais elementos do tipo `security-role-ref` (mostrado na Figura 32). Este elemento declara que um componente esta utilizando o valor de `role-name` como argumento para o método *isCallerInRole(String)*. Ao utilizar o método *isCallerInRole* um componente pode verificar se aquele que o chamou está em um *role* que foi declarado em um elemento `security-role-ref/role-name`. Caso o invocador do EJB não esteja definido em um elemento `security-role-ref` ele não obterá acesso ao mesmo.

O elemento *role-name* deve estar relacionado a um elemento security-role através do elemento *role-link*.

As Figura 33 e Figura 34 a mostram respectivamente exemplos de um descritor *ejb-jar.xml* e de um descritor *web.xml* que ilustram o uso do elemento *security-role-ref.*

Em ambas as figuras é possível notar as tags *security-role-ref*, *role-name* e *role-link*. As tags `role-name` e `role-link` definem respectivamente o pseudo-nome do *role* que pode acessar o EJB e a qual *role* ela de fato se refere. É necessário que esta *role* referenciada esteja definida em algum outro local do descritor.

---

[11] http://www.jboss.com/

[12] http://www.br.redhat.com/

[13] http://java.sun.com/j2ee/overview.html

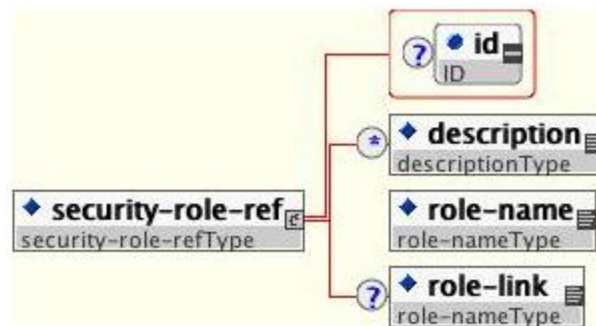

Figura 32 - O elemento security-role-ref

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
          <role-name>TheRoleICheck</role-name>
          <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

Figura 33 - Fragmento de um descritor *ejb-jar.xml* ilustrando o uso do `elemento security-role-ref`

```
<web-app>
    <servlet>
        <servlet-name>AServlet</servlet-name>
        ...
        <security-role-ref>
            <role-name>TheServletRole</role-name>
            <role-link>TheApplicationRole</role-link>
        </security-role-ref>
    </servlet>
    ...
</web-app>
```

Figura 34 - Fragmento de um descritor *web.xml* ilustrando o uso do elemento `security-role-ref`

Outros métodos para implementar segurança nas invocações de EJBs são explicitados em [Soldano *et al.*, 2008] e não serão expostos neste documento por não citarem meios de injetar dados do usuário no EJB invocado, sendo portanto uma tecnologia que não se aplica como solução ao problema de propagação de identidade.

B2. Injeção de Dependência

A idéia central da injeção de dependência consiste em eliminar a dependência explícita de código em uma classe [Fowler, 2004]. A injeção de dependência pode ajudar na manutenção, reuso e teste do código, inclusive, melhora a legibilidade do

código fonte [Rakshit, 2010]. Segundo Fowler [2004], existem três tipos principais de injeção de dependência: (i) injeção por construtores (*Constructor Injection*); (ii) injeção por métodos *setters (Setter Injection)*; (iii) injeção por interfaces (*Interface Injection*).

As próximas seções apresentam algumas soluções utilizadas para o desenvolvimento de aplicações utilizando injeção de dependência.

B2.2. *Contexts and Dependency Injection* (CDI)

A especificação JSR-299 [King, 2009] é o padrão Java EE[14] para tratar os aspectos referentes à injeção de dependência [Prassanna, 2009]. As figuras a seguir ilustram o uso de injeção de dependências utilizando a especificação CDI para injeção da classe *DefaultItemDao* (Figura 35) na classe *ItemProcessor* (Figura 36 – com dependência explícita; Figura 37 – com injeção de dependência).

```java
public class DefaultItemDao implements ItemDao{

  @Override
    public List<Item> fetchItems() {


        List<Item> results = new ArrayList<Item>();
        results.add(new Item(34, 7));
        results.add(new Item(4, 37));
        results.add(new Item(24, 19));
        results.add(new Item(89, 32));
        return results;
    }

}
```

Figura 35 – Classe *DefaultItemDAO* que implementa a interface *ItemDao*

A Figura 36 apresenta uma dependência explicita à classe *DefaultItemDAO*, que é instânciada através da palavra reservada *new*. Na Figura 37, podemos observar que não existe uma dependência explícita à classe *DefaultItemDAO*, ao invés disto, utilizamos a anotação *@Inject* [Johson e Lee, 2009] que define o ponto que a classe *DefaultItemDAO* será injetada.

```java
public class ItemProcessor {

    private ItemDao itemDao = new DefaultItemDao();

    public void execute() {
        List<Item> items = itemDao.fetchItems();
        for (Item item : items) {
            System.out.println("Found item " + item);
        }
    }

}
```

Figura 36 – Classe *ItemProcessor* implementada sem injeção de dependência

[14] http://www.oracle.com/technetwork/java/javaee/index.html

```java
public class ItemProcessor {

    @Inject
    private ItemDao itemDao;

    public void execute() {
        List<Item> items = itemDao.fetchItems();
        for (Item item : items) {
            System.out.println("Found item " + item);
        }
    }

}
```

Figura 37 - Classe *ItemProcessor* implementada utilizando injeção de dependência

B2.3. Spring Framework

O *Spring*[15] é um *framework* que tem a injeção de dependência como sua principal competência. Ele suporta, principalmente, injeção por métodos *setters* e injeção por construtores [Prassanna, 2009]. Segundo Walls e Breidenbach [2007] o *framework* foi criado para reduzir a complexidade no desenvolvimento de aplicações corporativas, utilizando simples *JavaBeans*[16] para resolver problemas que antes só eram possíveis com o uso de EJB (*Enterprise Java Beans*) [Burke e Monson-Haefel, 2006]. Além disso, o *Spring* promove um alto grau de abstração e oferece diversas formas de integração entre aplicações corporativas, *open-source* e comerciais. *Spring* também oferece suporte à programação orientada a aspectos (AspectJ[17] e AopAlliance[18]). A seguir apresentaremos um exemplo de uso do *Spring* 3.0 para injetar a dependência do bean *CadUser* com a classe *Foo*.

Inicialmente, criaremos o arquivo *contexto.xml* que representa o contexto da aplicação no *Spring*, apresentado na Figura 38. A seguir serão apresentadas as outras classes utilizadas na aplicação de exemplo.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="br.np2tec" />

</beans>
```

Figura 38 – Arquivo *contexto.xml*

A Figura 39 apresenta o *bean CadUser* onde a anotação *@Component*[19] define a classe *CadUser* como um componente que será gerenciado pela fábrica de *beans* do

[15] http://www.springsource.com

[16] http://download.oracle.com/javase/tutorial/javabeans/

[17] http://www.eclipse.org/aspectj/

[18] http://aopalliance.sourceforge.net/

[19] http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/stereotype/Component.html

Spring (BeanFactory[20]). O nome *cadUserBean* é utilizado para recuperar o *bean* da fábrica.

```
@Component("cadUserBean")
public class CadUser {

    private String chave;

    public String getChave() {
        return chave;
    }

    public void setChave(String chave) {
        this.chave = chave;
    }
```

Figura 39 – O *bean CadUser*

A Figura 40 apresenta o código do *bean Foo* onde a anotação *@Component*[21] indica que a classe também será gerenciada pela fábrica de *beans* do *Spring*. O *bean Foo* possui uma referência para o *bean CadUser* do qual será recuperada a informação de interesse. A classe possui ainda o método `setCadUser` com a anotação *@Autowired. O uso* desta anotação injeta o *bean CadUser,* automaticamente, utilizando injeção de dependência através de método *set*. O método *getChave* exibe na console uma informação através da invocação do método *getChave* do *bean* injetado *CadUser*.

```
@Component("fooBean")
public class Foo {

    private CadUser cadUser;

    @Autowired
    public void setCadUser(CadUser cadUser) {
        this.cadUser = cadUser;
    }


    public void getChave(){
        System.out.println("A chave é: "+cadUser.getChave());
    }

}
```

Figura 40 – O *Bean Foo*

A classe *TesteDI* (Figura 41) implementa o código utilizado para testar a aplicação de exemplo. Através da classe *ClassPathXmlApplicationContext(contexto.xml")* é recuperado o contexto da aplicação, definido no arquivo *contexto.xml*. Primeiro, recuperamos o *bean cadUserBean,* da fábrica de contextos do Spring, através do método *getBean.* Segundo, ajustamos o valor "eu75"através do método *setChave* do *bean CadUser.* Terceiro, recuperamos o *bean FooBean* da fábrica de *beans* e em seguida invocamos o método *getChave*.

---

[20] http://static.springsource.org/spring/docs/2.5.x/api/index.html?org/springframework/beans/factory/BeanFactory.html

[21] http://static.springsource.org/spring/docs/2.5.x/api/index.html?org/springframework/beans/factory/BeanFactory.html

```
public class TesteDI extends TestCase {

    public void testDI() {

        ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
                "classpath:contexto.xml");

        CadUser cadUserBean = (CadUser) applicationContext.getBean("cadUserBean");
        cadUserBean.setChave("eu75");

        Foo fooBean = (Foo) applicationContext.getBean("fooBean");
        fooBean.getChave();

    }

}
```

Figura 41 – Classe TesteDI utilizada para testar a aplicação de exemplo

O resultado da execução da classe *TesteDI* pode ser visto naFigura 42. Podemos observar que não existe um dependência direta entre o bean *CadUser* e o *bean Foo*. A dependência é realizada pelo *container*, que injeta o *bean Foo* em tempo de execução.

```
Problems  @ Javadoc  Declaration  Search  Console ⊠  Outline  Servers
<terminated> TesteDI [JUnit] C:\Sun\SDK\jdk\bin\javaw.exe (22/11/2010 21:38:43)
22/11/2010 21:38:44 org.springframework.context.support.AbstractApplicationConte
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationCont
22/11/2010 21:38:44 org.springframework.beans.factory.xml.XmlBeanDefinitionReade
INFO: Loading XML bean definitions from class path resource [contexto.xml]
22/11/2010 21:38:44 org.springframework.beans.factory.support.DefaultListableBea
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.
A chave é: eu75
```

Figura 42 – Resultado da execução da classe *TesteDI*

B2.4. Google Guice

Segundo [Prassana, 2009] o *Guice* é um framework de injeção de dependência criado pela Google, considerado leve (*lightweight*), que suporta injeção de dependência por métodos *set* e construtores. *Guice* também oferece suporte a programação orientada a aspectos [Kiczales *et al.*, 1997] e integração com *frameworks* como o *Spring*, permitindo que *beans* definidos no *Spring* sejam injetados nos *beans* configurados pelo *Guice*. O *Guice* pode ser visto como uma fábrica de injetores de dependência.

O *Guice* oferece uma maneira simples para realizar as configurações dos serviços e das dependências da aplicação. As configurações são realizadas através da implementação da *interface Module*. A Figura 43 ilustra um exemplo de um módulo de configuração que implementa a *interface Module*. Uma alternativa para implementar um módulo de configuração é através da extensão da classe *AbstractModule* (Figura 44).

```java
public class MeuModulo implements Module{

    @Override
    public void configure(Binder binder) {
        binder.bind(CadUser.class);
        binder.bind(Foo.class);

    }

}
```

Figura 43 – Módulo de configuração que implementa a *interface Module*

```java
public class OutroModulo extends AbstractModule{
    @Override
    protected void configure() {
        bind(CadUser.class);
        bind(Foo.class);
    }
```

Figura 44 - Módulo de configuração que estende a classe *AbstractModule*

A Figura 45 ilustra a injeção de dependência no Guice através da anotação *@Inject* no método *setCadUser()*.

```java
public class Foo {

    private CadUser cadUser;

    @Inject
    public void setCadUser(CadUser cadUser) {
        this.cadUser = cadUser;
    }

    public void getChave(){
        System.out.println("A chave é: "+cadUser.getChave());
    }
}
```

Figura 45 – *Bean Foo*

A Figura 46 apresenta um método de teste que cria o injetor, recupera a instância do *bean CadUser* do injetor, invoca o método *setChave()* passando a chave eu75 e recupera a instância do *bean Foo* e em seguida invoca o método *getChave()*.

```
public void testGuice(){

    Injector injector = Guice.createInjector();

    CadUser cadUserBean = injector.getInstance(CadUser.class);
    cadUserBean.setChave("eu75");

    Foo fooBean = injector.getInstance(Foo.class);
    fooBean.getChave();
}
```

Figura 46 – Método de teste que cria o injetor e recupera os *beans CadUser* e *Foo*

No entanto, não foi possível recuperar o valor da chave no método *getChave()* do *bean Foo* (Figura 45). Apesar do bean *CadUser* ter sido injetado, o valor setado no método de teste (Figura 46) não pode ser recuperado.

B2.5. OpenEJB

OpenEJB[22] é uma implementação leve (*lightweight*) de um container EJB 3.0 que pode ser utilizado como servidor ou embutido em outro container, por exemplo Tomcat[23]. Ele possui ainda integração com o *framework Spring,* permitindo que os *beans* definidos no *Spring* possam ser injetados em componentes JavaEE e vice-versa, inclusive permite que recursos JavaEE (@*Resources*) como EJB, conector JDBC, conextor JMS dentre outros sejam injetados em *beans* do *Spring*,.

B3. Hibernate

Hibernate[24] é uma solução para mapeamento Objeto/Relacional para ambientes Java [King *et al.*, 2010a]. O termo "Mapeamento Objeto/Relacional" faz referência à técnica de mapear os dados de um modelo de objetos para o modelo relacional de dados (e vice-versa). O framework auxilia na abstração de parte do conhecimento da linguagem de consulta a bancos de dados SQL. De acordo com seus criadores, o Hibernate é capaz de reduzir significativamente o tempo de desenvolvimento que de outra forma seria despendido com o tratamento dos dados através de SQL e JDBC.

Por encapsular o acesso ao banco de dados o Hibernate deve ser considerado no estudo de soluções para os cenários 3 e 4. Esta seção tratará dos aspectos necessários à configuração básica do Hibernate, de algumas de suas funcionalidades e de padrões de projeto recomendados pela *Hibernate Community*[25].

B3.2. Configurações Básicas

Objetivando simplificar a explicação de como o Hibernate deve ser configurado, será considerado somente a situação de uma classe que deve ser diretamente persistida em uma única tabela do banco de dados.

[22] http://openejb.apache.org/

[23] http://tomcat.apache.org/

[24] http://www.hibernate.org/

[25] http://community.jboss.org/en/hibernate

Inicialmente deve-se considerar que o framework precisa ser adicionado ao projeto. O framework Hibernate é disponibilizado para a linguagem Java através de um conjunto de arquivos .jar, sendo o arquivo principal o *hibernate.jar3* (considerando que a versão que está sendo utilizada é a 3.x). Além do *hibernate3.jar* é ainda necessário adicionar ao classpath do projeto todos os arquivos incluídos no diretório *lib/required*.

A configuração de acesso do Hibernate ao banco de dados pode ser feita através de um arquivo *plain-text* (chamado de *hibernate.properties*) ou de um arquivo xml (chamado *hibernate.cfg.xml*). O arquivo XML comporta mais opções de configuração e proporciona uma melhor estrutura visual. Em [Bauer *et al.*, 2005] é observado que o uso do arquivo *xml* possibilita ainda configurar integralmente a *sessionFactory* do Hibernate e centralizar a especificação das localizações dos arquivos de mapeamento das classes persistentes. A Figura 47 e a Figura 48 mostram respectivamente exemplos dos arquivos *hibernate.properties* e *hibernate.cfg.xml*.

```
01. hibernate.dialect org.hibernate.dialect.MySQLDialect
02. hibernate.connection.driver_class org.gjt.mm.mysql.Driver
03. hibernate.connection.url jdbc:mysql://localhost:3306/xyz
04. hibernate.connection.username root
05. hibernate.connection.password
06. hibernate.show_sql true
07. hibernate.format_sql true
```

Figura 47 - Arquivo de configuração *hibernate.properties*

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <!DOCTYPE hibernate-configuration PUBLIC
3          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5  <hibernate-configuration>
6      <session-factory>
7          <!-- Database connection settings -->
8          <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
9          <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
10         <property name="connection.username">system</property>
11         <property name="connection.password">101086</property>
12
13         <!-- JDBC connection pool (use the built-in) -->
14         <property name="connection.pool_size">1</property>
15
16         <!-- SQL dialect -->
17         <property name="dialect"> org.hibernate.dialect.Oracle10gDialect</property>
18
19         <!-- Enable Hibernate's automatic session context management -->
20         <property name="current_session_context_class">thread</property>
21
22         <!-- Disable the second-level cache -->
23         <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
24
25         <!-- Echo all executed SQL to stdout -->
26         <property name="show_sql">false</property>
27
28         <!-- Mapeamento de todos os "hbm.xml" -->
29         <mapping resource="modelo/entidade/Poco.hbm.xml"/>
30     </session-factory>
31 </hibernate-configuration>
```

Figura 48 - Arquivo de configuração *hibernate.cfg.xml*

Como é possível ver nas Figura 47 e Figura 48, quando a configuração é realizada através do arquivo xml é possível configurar mais variáveis, gerando um ambiente mais personalizado.

Além do arquivo de configuração do próprio Hibernate, é necessário ainda inserir o arquivo de configuração para a API do **log4j**[26], que consiste em um *logger* da execução do projeto. O arquivo de configuração *log4j.properties* pode ser visto na Figura 49.

```
1   #LINHA QUE DEFINE O OUTPUT DO HIBERNATE NO CONSOLE
2   log4j.rootLogger=DEBUG, dest1
3
4   log4j.appender.dest1=org.apache.log4j.ConsoleAppender
5   log4j.appender.dest1.layout=org.apache.log4j.PatternLayout
6   log4j.appender.dest1.layout.ConversionPattern=%d %-5p %-5c{3} %x -> %m%n
7
8   #log4j.appender.dest2=org.apache.log4j.RollingFileAppender
9   #log4j.appender.dest2.File=bridge.log
10
11  #log4j.appender.dest2.MaxFileSize=100KB
12  # Keep one backup file
13  #log4j.appender.dest2.MaxBackupIndex=3
14
15  #log4j.appender.dest2.layout=org.apache.log4j.PatternLayout
16  #log4j.appender.dest2.layout.ConversionPattern=%d [%t] %-5p %-5c{3}(%L) %x -> %m%n
```

Figura 49 - Arquivo de configuração do log4j

A classe da aplicação que se deseja mapear na tabela do banco de dados deve possuir um arquivo de mapeamento. Por padrão, define-se o nome do arquivo como o nome da classe mapeada seguido do sufixo ".hbm.xml". A documentação do Hibernate recomenda que um arquivo de mapeamento seja criado para cada classe mapeada e que eles se encontrem no mesmo pacote que as suas respectivas classes. Considere uma classe POJO nomeada "Poco" que implementa a interface Serializable e possui 3 atributos: *id*, *nome* e *estado*, representando respectivamente o ID de um poço de extração de petróleo, seu nome e o estado onde se localiza. A Figura 50 mostra o arquivo de mapeamento *Poco.hbm.xml* que serve para mapear a classe Poco para a tabela POCOS do banco. Os atributos da classe estão mapeados para colunas da tabela.

```
1   <?xml version="1.0"?>
2   <!DOCTYPE hibernate-mapping PUBLIC
3    "-//Hibernate/Hibernate Mapping DTD//EN"
4      "http://hibernate.sourceforge.net/hibernate-mapping.dtd">
5   <hibernate-mapping>
6       <class name="modelo.entidade.Poco" table="POCOS">
7           <id name="id" column="ID" type="int">
8               <generator class="assigned"/>
9           </id>
10          <property name="nome" column="NOME" type="string"/>
11          <property name="estado" column="ESTADO" type="string"/>
12      </class>
13  </hibernate-mapping>
```

Figura 50 - Arquivo de mapeamento para a classe Poço

Uma vez que a configuração tenha sido feita, o banco de dados implementado e o *driver* e dialeto corretos tenham sido escolhidos e configurados no arquivo de configuração do Hibernate (*hibernate.properties* ou *hibernate.cfg.xml*) o projeto deverá ser capaz de acessar o banco.

---

[26] http://logging.apache.org/log4j/index.html

B3.3. Interceptadores

Em algumas situações é desejável interceder em ações realizadas por métodos providos pelo Hibernate. Interceptadores permitem à aplicação realizar inspeções e/ou manipular os dados do objeto que se pretende persistir ou recuperar. De forma geral interceptadores são implementados através do uso de interfaces que comportam métodos que, quando implementados pela classe que as estende, são executados antes do *commit* na *session* utilizada.

Interceptadores podem ser definidos por sessão (*Session-scoped Interceptor*) ou por Fábrica de sessão (*SessionFactory-scoped Interceptor*). O primeiro é valido somente para a sessão em uso, e para definir seu uso é necessário utilizar o método sobrecarregado *openSession(Interceptor)*, presente na *SessionFactory*. Quando um interceptador é definido desta forma será válido somente para aquela sessão (Figura 51).

```
1  Configuration configuration = new Configuration();
2  SessionFactory sf = configuration.buildSessionFactory();
3
4  MeuInterceptador mInterceptador = new MeuInterceptador();
5  Session s = sf.openSession(mInterceptador);
```

Figura 51 - Definição de um interceptador *Session-scoped*

Quando um interceptador é definido por Fábrica de Sessão, ele será válido para toda e qualquer sessão criada através da *SessionFactory*. O interceptador deve ser definido no objeto *Configuration* antes da instanciação da *SessionFactory*, desta forma todas as sessões abertas por aquela *SessionFactory* estarão associadas ao interceptador (Figura 52).

```
1  Configuration configuration = new Configuration();
2  configuration.setInterceptor(new MeuInterceptador());
3  SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Figura 52 - Definição de um interceptador *SessionFactory-scoped*

O framework fornece 3 interfaces que podem ser implementadas para a criação de novos interceptadores: *Validatable*, *Lifecycle* e *Interceptor*. Como fonte extra de consulta, comportando exemplos de uso destas interfaces, é recomendado o site JavaBeat[27].

B3.4. Interface Validatable

A interface Validatable é a mais simples das três interfaces que fornecem métodos interceptadores. Possui um único método, chamado *validate()*, normalmente utilizado quando se deseja validar os dados do objeto a ser persistido. O método é invocado pelo framework sempre que a operação *save* é realizada. A operação *save* pode ocorrer quando os métodos *Session.save()*, *Session.update()*, *Session.saveOrUpdate()* ou *Session.flush()* são invocados. A Figura 53 mostra um exemplo de classe interceptadora implementada através do uso da interface *Validatable*. Caso haja algum problema na validação, o método *validate()* irá lançar uma exceção, impedindo a persistência do objeto.

---

[27] http://www.javabeat.net/articles/9-interceptors-in-hibernate-orm-framework-an-introducti-1.html

```java
import java.util.Date;
import org.hibernate.classic.Validatable;
import org.hibernate.classic.ValidationFailure;

public class ProjectDuration implements Validatable{

    private Date startDate;
    private Date endDate;

    /// Other Code here.

    public void validate(){
        if (startDate.after(endDate)){
            throw new ValidationFailure(
                "Start Date cannot be greater than the End Date.")
        }
    }
}
```

Figura 53 - Classe implementando a interface *Validatable*

B3.4.1. Interface Lifecycle

Um objeto persistente passa por diversas fases durante seu ciclo de vida, podendo ser criado, persistido, recuperado de um banco de dados, alterado ou deletado. A interface *Lifecycle* tem como objetivo encapsular operações que possibilitem a intervenção sempre que operações como estas forem realizadas. A interface possui 4 métodos:

- ***onLoad()*** – invocado pelo framework antes de o objeto ser carregado, isto é, quando o método *Session.load()* é chamado.
- ***onSave()*** – invocado pelo framework antes de o objeto ser salvo, isto é, quando os métodos *Session.save()* e *Session.saveOrUpdate()* são chamados.
- ***onUpdate()*** – invocado pelo framework antes de um objeto ser atualizado, isto é, quando o método *Session.update()* é chamado.
- ***onDelete()*** – invocado pelo framwork antes de um objeto persistente ser deletado do banco de dados, isto é, quando o método *Session.delete()* é chamado.

B3.4.1.1. Interface *Interceptor*

A mais completa das 3 interfaces, possuindo ao todo 18 métodos que possibilitam interceder no andamento das operações realizadas pelo framework Hibernate. Objetivando simplificar o uso da interface, os desenvolvedores do framework provêem a classe concreta *EmptyInterceptor* que possui implementação *default* e vazia para todos os métodos da interface *Interceptor*, podendo então ser estendida e possibilitando ao desenvolvedor implementar somente os métodos que julgar necessários, ao invés de obrigá-lo a implementar todos os métodos providos pela interface.

A seguir são descritos alguns dos métodos mais comuns disponibilizados pela interface. Para uma lista completa de todos os métodos disponíveis o JavaDoc[28] da interface deve ser consultado.

- ***afterTransactionBegin()*** – invocado pelo framework sempre que uma *transaction* for iniciada e antes de os métodos dentro dela serem executados.
- ***beforeTransactionClose()*** – invocado pelo framework antes de uma *transaction* iniciada pelo usuário ser finalizada.
- ***onLoad()*** – invocado pelo framework antes do objeto ser carregado, isto é, quando o método *Session.load()* é chamado.
- ***onSave()*** – invocado pelo framework antes de o objeto ser salvo, isto é, quando os métodos *Session.save()* e *Session.saveOrUpdate()* são chamados.
- ***onUpdate()*** – invocado pelo framework antes de um objeto ser atualizado, isto é, quando o método *Session.update()* é chamado.
- ***onDelete()*** – invocado pelo framwork antes de um objeto persistente ser deletado do banco de dados, isto é, quando o método *Session.delete()* é chamado.

Como pode ser visto na descrição acima, a interface *Interceptor* agrega os métodos fornecidos pela interface Lifecycle.

Deve ser observado que os métodos afterTransactionBegin() e beforeTransactionClose() são capazes de interceder no andamento comum de uma transação, por exemplo executando operações de validação nos dados. Entretanto não é possível, por natureza, evocar a sessão que está sendo utilizada pela transação afim de executar novas operações no banco de dados. Os dois métodos somente serão invocados caso uma *transaction* tenha sido inicializada e finalizada (através do método *commit()*), ou seja, métodos como *onLoad(), onSave(), onUpdate()* e *onDelete()* poderão ser executados sem que tratamentos estabelecidos naqueles dois métodos sejam executados.

B3.4.1.2. Análise do uso de interceptadores para os cenários 3 e 4

Interceptadores, apesar de fornecer grande utilidade e flexibilidade ao framework , não se provaram úteis aos cenários 3 e 4.

As interfaces possuem grande poder de intervenção no andamento padrão da execução do Hibernate, através de métodos como *onLoad*, *onSave* e *onUpdate*, entretanto estes métodos tem como proposta intervir nos dados tratados ou em outros pontos do sistema, ao invés de lidar com o banco de dados, ou seja, para executar qualquer operação de preparação no banco de dados seria necessário abrir uma nova sessão com o banco e executar as operações. A seção seria automaticamente encerrada após as tarefas ao invés de reaproveitada para a execução das operações originais (recuperar, salvar ou atualizar dados). Um pré-requisito essencial para o controle do acesso aos dados é garantir que a mesma sessão esteja sendo utilizada na preparação do banco e na execução das consultas.

---

[28] http://docs.jboss.org/hibernate/stable/core/api/org/hibernate/Interceptor.html

B3.4.2. Padrões de Projeto para Hibernate

Os desenvolvedores do Framework recomendam o uso de alguns padrões de projeto, como o *DAO* e o *Singleton* (utilizado na classe auxiliar HibernateUtil e explicados com maiores detalhes na Seção B4), a fim de facilitar o uso do Hibernate. Além disso, eles estabelecem outros padrões próprios a fim de direcionar as melhores formas de se trabalhar com a API.

Patricio *et al.* [2010] recomenda o uso da classe auxiliar *HibernateUtil* a fim de centralizar o lookup da SessionFactory na JNDI e a inicialização do Hibernate (feita através do carregamento do arquivo de configuração do Hibernate).

Uma vez que o lookup da SessionFactory tenha sido centralizado, toda classe que desejar abrir uma sessão deverá requisitar a fábrica à classe HibernateUtil (Figura 54). A fábrica por sua vez é definida através do uso do padrão de projeto *Singleton,* sendo inicializada uma única vez e armazenada em uma variável *static final*, desta forma é possível defini-la apenas uma vez e garantir que sempre que uma fábrica de sessões for solicitada a mesma instância será entregue ao requisitante.

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

Figura 54 - Implementação da classe auxiliar *HibernateUtil*

Da mesma forma que centraliza o *lookup* da *SessionFactory*, a classe centraliza a inicialização do Hibernate, colocando em um único ponto de todo o sistema a chamada ao método *configure()*, que carrega o arquivo *hibernate.cfg.xml*. É importante ressaltar que este arquivo comporta todas as configurações necessárias ao Hibernate para estabelecer o acesso a um banco de dados.

B3.4.2.1. Padrões de projeto para utilização de *Sessions*

Em [Patricio *et al.*, 2010] são citados 4 padrões para a utilização de sessões em uma aplicação que utiliza o framework Hibernate: *Session-per-request*, *Session-per-operation*, *session-per-request-with-detached-objects* e *Session-per-conversation*.

- ***Session-per-request*** – É o modelo de programação mais utilizado. Uma única sessão e uma única transação no banco de dados são utilizadas para processar uma requisição específica.
- ***Session-per-operation*** – É citado como um "anti-padrão", sendo o seu uso não recomendado. Neste modelo uma sessão é criada para cada operação

executada, por exemplo, se em um mesmo trecho de código deseja-se salvar dois objetos recebidos e posteriormente recuperar um terceiro, uma sessão seria aberta para cada uma das operações, ao invés de se aproveitar a sessão corrente.

- ***Session-per-request-with-detached-objects*** – Um padrão considerado para a implementação de uma situação semelhante a um *Wizard Dialog*, ou seja, quando uma interação entre sistema e usuário é realizada passo-a-passo (modelo de conversação). Nesta situação há ocorrência de múltiplos ciclos de *request/response* e o relacionamento entre os objetos e a sessão se perde, sendo necessário criar uma nova sessão toda vez que um destes ciclos chega ao fim.
- ***Session-per-conversation*** – É o padrão recomendado quando houver uma situação de conversação entre usuário e aplicação citada no padrão *Session-per-request-with-detached-objects*. Neste caso, uma sessão tem um escopo maior do que o de uma única transação com o banco de dados, podendo conter várias transações. Cada ciclo de *request/response* é executado em uma transação própria, mas o *flush* da sessão é adiado até o fim da conversação, fazendo com que a conversação se torne atômica.

Como citado os dois padrões recomendados para utilização são o *session-per-request* e o *session-per-conversation*. É importante enfatizar que , ao contrário do *session-per-operation*, o padrão *session-per-request-with-detached-objects* não é considerado um anti-padrão.

B3.4.2.2. Análise do uso de padrões Hibernate para os cenários 3 e 4

O uso dos padrões recomendados pelos desenvolvedores do framework Hibernate se provaram de grande utilidade, dado que simplificam o desenvolvimento de aplicações que fazem uso do framework. A classe HibernateUtil pode ser adaptada para, além de comportar o padrão de instanciação da SessionFactory e configuração do framework, abranger métodos que se responsabilizem por fornecer sessões à aplicação.

No caso de um método único responsável por criar sessões é possível centralizar em um único ponto da aplicação a preparação do banco de dados para lidar com o controle de acesso. Os problemas citados anteriormente na utilização de interceptadores não mais se aplica, já que ao ter um único método responsável por retornar sessões é possível criar uma sessão, preparar o banco de dados e então devolver à aplicação a mesma sessão.

B4. *Design Patterns* (Padrões de Projeto)

Uma forma de obter a propagação de identidade consiste em combinar padrões de projeto a fim de obter um comportamento esperado dos sistemas. Soluções envolvendo combinação entre padrões podem ser consideradas levemente intrusivas, dado que para serem aplicadas dependem da existência prévia de alguns dos padrões nos sistemas que serão adaptados ou da refatoração de trechos de código para que seja possível introduzir novos padrões.

Esta seção descreve alguns dos padrões estudados que são considerados de grande utilidade para a tarefa de propagação de identidade. Alguns padrões citados anteriormente serão explicados mais detalhadamente nesta seção.

B4.2. Singleton

O padrão de projeto *Singleton* tem como objetivo restringir a instanciação de uma classe a um único objeto. Este padrão é de grande utilidade quando apenas um único objeto de uma classe é necessário, evitando a múltipla instanciação da classe e consequentemente o gasto desnecessário de recursos. Este padrão é considerado por muitos um *anti-pattern*.

A Figura 55 mostra a forma mais comum de implementar o padrão Singleton, onde o escopo do construtor default da classe é definida como privado e o método getInstance definido como estático, desta forma para obter um objeto da classe é necessário invocar o método getInstance, que retornará a única instancia da classe.

```
1   Class MinhaClasse{
2
3       private MinhaClasse instancia = null;
4
5       private MinhaClasse(){}
6
7       public static synchronized MinhaClasse getInstance(){
8           if(instancia == null){
9               instancia = new MinhaClasse();
10          }
11          return instancia;
12      }
13
14      //Métodos da classe abaixo...
15      // ...
16  }
```

Figura 55 - Implementação usual do padrão Singleton

Outras formas foram estabelecidas para definir uma classe como singleton, um exemplo é a anotação @Singleton, presente nos pacotes `javax.ejb.Singleton` e `javax.inject.Singleton`.

B4.3. Service Locator

Aplicações empresariais precisam de um meio de localizar os objetos de serviço responsáveis por fornecer acesso à componentes distribuídos. Aplicações construídas com J2EE utilizam o Java Naming and Directory Interface (JNDI) para localizar as interfaces de enterprise beans.

O padrão Service Locator centraliza o método de procura por objetos, provendo um ponto central de controle na aplicação eliminando lookups redundantes e qualquer especificidade do processo de localização.

Na Figura 56 pode ser vista uma possível implementação do padrão onde o método get tem por função realizar o *lookup* do objeto e retorná-lo.

```java
public class ServiceLocator {

    public ServiceLocator(){}

    private InitialContext recuperarContexto() throws NamingException{
        // Access JNDI Initial Context.
        Properties p = new Properties();
        p.put("java.naming.factory.initial","org.jnp.interfaces.NamingContextFactory");
        p.put("java.naming.provider.url","jnp://localhost:1099");
        p.put("java.naming.factory.url.pkgs","org.jboss.naming:org.jnp.interfaces");

        InitialContext ctx = new InitialContext(p);

        return ctx;
    }

    public Object get(String jndiName) throws NamingException {
        InitialContext ctx = recuperarContexto();
        Object obj = ctx.lookup(jndiName);

        return obj;
    }
}
```

Figura 56 - Implementação do padrão Service Locator

B4.4. DAO (Data Access Object)

O padrão DAO (*Data-acces Object*), tem como objetivo isolar a implementação do acesso de objetos de um domínio em um mecanismo de armazenamento. Através deste padrão é possível separar a lógica do negócio da tecnologia utilizada para o armazenamento dos dados.

Na Figura 57 pode ser visto um caso simples de consulta SQL realizada através do driver JDBC do banco de dados. A consulta em questão é encapsulada pelo método *consultarCliente()*, que recebe como parâmetro o identificador de um cliente. A utilização do padrão DAO permite que qualquer alteração na estrutura do banco de dados, ou a necessidade de alterar a forma de armazenamento dos dados (por exemplo trocar entre banco de dados relacional e banco de dados orientado a objeto) gere impacto mínimo nas aplicações. A aplicação não sabe como é feito o acesso aos dados ou a sua forma de armazenamento, apenas executa o método *consultarCliente()* esperando receber o objeto Cliente desejado.

```java
public List<Cliente> consultarCliente(int pk) throws Exception {
    connection = DriverManager.getConnection(url, user, password);
    String sql = "select * from Cliente where cliente.id = ?";
    Cliente cliente = new Cliente();
    try {
        PreparedStatement stmt = connection.prepareStatement(sql);
        stmt.setString(1, Integer.toString(pk));
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            cliente.setCpf(rs.getString("cli_cpf"));
            cliente.setNome(rs.getString("cli_nome"));
            cliente.setEndereco(rs.getString("cli_endereco"));
        }
    } catch (SQLException e) {
        throw e;
    } finally {
        //Fechar Statement, ResultSet e connection...
    }

    return cliente;
}
```

Figura 57 - Exemplo de método implementado em uma clsse DAO