SIMONE CARDOSO DA SILVA

CONSTRUÇÃO DE UMA CLASSE DE PERSISTÊNCIA GENÉRICA UTILIZANDO O FRAMEWORK HIBERNATE

ASSIS
2009

CONSTRUÇÃO DE UMA CLASSE DE PERSISTÊNCIA GENÉRICA UTILIZANDO O FRAMEWORK HIBERNATE

SIMONE CARDOSO DA SILVA

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, analisado pela seguinte comissão examinadora:

Orientador: Ms. DOUGLAS SANCHES DA CUNHA

Analisador: Dr. ALMIR ROGÉRIO CAMOLESI

ASSIS
2009

SIMONE CARDOSO DA SILVA

CONSTRUÇÃO DE UMA CLASSE DE PERSISTÊNCIA GENÉRICA UTILIZANDO O FRAMEWORK HIBERNATE

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, analisado pela seguinte comissão examinadora:

Orientador: Ms. DOUGLAS SANCHES DA CUNHA
Área de Concentração: Persistência e Banco de Dados

ASSIS
2009

RESUMO

Desenvolver software orientado a objetos respeitando o modelo de arquitetura em camadas MVC, é uma boa prática de desenvolvimento e aprendizado. Construir uma classe de persistência genérica, que atua na camada de Controller, pode ser um grande desafio. Esta classe de persistência genérica tem como principal objetivo, reduzir o tempo dedicado na construção e substituição das classes DAO, proporcionando ao desenvolvedor a criação das operações básicas (consultar, inserir, alterar e excluir), para que, a persistência das classes de entidades seja realizada pela mesma classe genérica. Portanto, os desenvolvedores não precisam escrever instruções SQL's básicas, ganhando produtividade, reduzindo códigos para manutenção e mantendo o mesmo desempenho no acesso aos dados.

Palavras chave: Hibernate, DAO genérico, persistência genérica, SQL.

ABSTRACT

The development of object oriented software respecting the MVC architecture model, is a good development practice and learning. Building a generic persistence class, to act within the Controller layer, may be a great challenge. The main goal of this generic persistence class, is to reduce the time needed on the development of DAO classes, allowing developers (create, retrieve, update, delete) operations in order to persist entity classes with the same generic class. Therefore, developers don't have to write basic SQL's instructions, achieving more productivity and reducing maintenance code, with the same data access performance.

Keywords : Hibernate, generic DAO, generic persistence, SQL.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Atualmente a maioria dos sistemas construídos em linguagens Orientada à Objetos, utilizam o modelo MVC (Model-View-Controller), como arquitetura em camadas: apresentação, lógica de negócio e a camada de persistência. Onde a camada de apresentação define a interface do usuário, a camada de lógica de negócio define as necessidades que o sistema deve atender. Por fim, a camada de persistência, responsável por guardar e recuperar as informações pertinentes em um banco de dados.

A maioria dos sistemas computacionais que acessam os bancos de dados, utilizam a linguagem SQL (Structured Query Language). Caso a aplicação seja simples, poderá ser uma excelente opção, mas, se uma aplicação for muito extensa, relativamente grande demais? Como ficariam as expressões de filtro nas instruções SQL, quando as empresas fizerem uma mudança no Banco de Dados? Quais seriam os impactos e os problemas na manutenção? Imagine a construção de uma ou várias SQL para buscar dados em inúmeras tabelas diferentes, poderiam acontecer também uma demora maior no processamento dos dados.

Estes poderão ser alguns dos problemas de uma aplicação direta com o Banco de Dados Relacionais. Podendo aparecer também problemas com associações, identidades (quando se compara dois objetos), navegação entre os dados, entre outros.

Quando se trabalha com orientação a objetos deve se criar uma classe de persistência para cada tabela do banco de dados. É exatamente desta forma que os programadores de nível básico desenvolvem uma aplicação, sem preocupar-se com quantas classes de persistência a aplicação suportará. Alem de ser obrigado a saber quais as diferentes especificações de um SQL que podem variar de um banco de dados para outro.

Utilizar um *framework* que trabalhe como uma “ponte” entre os objetos da classe e o banco de dados poderá ser conveniente, é exatamente nisso que o Hibernate

poderá ajudar para o desenvolvimento de sistemas. Transformando os dados da aplicação, de objetos persistentes em dados relacionais, de uma maneira que o banco de dados em uso já tenha suporte aos dados relacionais. Não importando qual será o banco de dados utilizado, afinal o Hibernate possui vários dialetos para comunicar-se com muitos bancos de dados.

O intuito principal do Hibernate é reduzir o tempo dedicado na construção da persistência dos dados, direcionando esse tempo gasto na lógica de negócios. Sem perder a produtividade, manutenibilidade e agilidade. Muitas empresas não possuem DBA's (Database administrator) e os programadores acabam desenvolvendo está função e na maioria dos casos estes programadores não possuem total domínio sobre o banco de dados que a empresa utiliza, e os mesmos acabam perdendo muito tempo na construção de tabelas e SQL. Caso esta atividade for reduzida aos desenvolvedores, eles teriam mais tempo para pensar e desenvolver a lógica de negócio da aplicação.

O estudo de caso pretende estimar quais serão as vantagens e desvantagens no desenvolvimento de software orientado a objetos utilizando o *framework* Hibernate e a construção de uma classe de persistência genérica para atender as funções básicas de uma aplicação, desenvolvida em Java. Este código terá menos ou mais linhas de programação, afinal, não existe perspectiva de manutenção da aplicação ou sua complexidade.

Espera-se aprender mais a respeito de uma tecnologia que é muito utilizada no desenvolvimento Java e que possui grande abrangência de oportunidades no mercado de trabalho. Por meio da construção de apenas uma única classe de persistência: para realizar inserção, alteração, exclusão e seleção dos dados que são realizados durante a execução de qualquer sistema. Pesquisar o *framework* Hibernate e avaliar quais são as reais contribuições que esta tecnologia pode oferecer aos desenvolvedores. A classe de persistência poderá reduzir o tempo de construção sem que o mesmo perca eficiência e rapidez. Conhecimentos sobre outras tecnologias de Java que poderão abrir novas chances no mercado corporativo.

Será realizado um estudo de caso do Hibernate, afim de aprender seus conceitos principais como ferramenta de trabalho e desenvolvimento. Utilizar um banco de dados relacional para efetuar os testes necessários e a construção da classe de persistência genérica, para comprovar a sua eficiência. Estas serão realizadas através de pesquisas em revistas, fóruns e artigos relacionados ao assunto Hibernate. Também poderão ocorrer algumas entrevistas com pessoas que trabalhem com o *framework*, para transmitirem um pouco de suas experiências da utilização da ferramenta na prática. Para o desenvolvimento da pesquisa será necessário um computador com uma IDE (Integrated Development Environment), para desenvolvimento em Java, banco de dados MySQL e o *framework* Hibernate.

O trabalho foi divido em seis capítulos, conforme descritos a seguir.

Capítulo 1 – Introdução: Uma apresentação geral do estudo de caso, mostrando os objetivos principais, as justificativas, como o trabalho foi dividido e as ferramentas necessárias para o seu desenvolvimento.

Capítulo 2 – Objeto/Relacional: Apresenta os conceitos chave, os problemas que podão ocorrer quando está trabalhando com banco de dados relacionais.

Capítulo 3 – O Hibernate: Explicação do que é a ferramenta Hibernate, seu funcionamento na realização de comandos básicos como: select, insert, update, delete.

Capítulo 4 – Persistência Genérica: Construção de uma persistência genérica. Realizar a implementação de uma única classe de persistência, que possa ser compatível com qualquer implementação de sistema usando Java em qualquer banco de dados. E implantar a persistência, afim de realizar testes para avaliar seu desempenho.

Capítulo 5 – Conclusões: Serão destacadas as considerações finais do estudo de caso, bem como projeções futuras.

Capítulo 6 – Referências Bibliográficas: Apresentação de todas as referências utilizadas para desenvolver este trabalho.

2. OBJETO/RELACIONAL

A maioria dos softwares necessitam armazenar seus dados, afinal posteriormente o usuário vai ter acesso à eles. Portanto, precisa-se de um método de armazenamento, e é através dos banco de dados relacionais que encontra-se algumas soluções.

2.1. BANCO DE DADOS RELACIONAIS

Os banco de dados relacionais estão muito bem estabelecidos e possuem uma forte estrutura, por isto, são muito utilizados por pequenas e grandes empresas de software, porque disponibilizam um ótimo gerenciamento dos dados além de garantir a integridade dos mesmos.

Porém, não pode-se dizer que existe um sistema de gerenciamento de banco de dados relacionais específico para uma linguagem e também não existe um banco de dados relacional específico para cada aplicação, tem-se então o princípio de independência dos dados, afinal não se pode prender uma aplicação a apenas uma tecnologia ao banco de dados.

A tecnologia relacional oferece o compartilhamento dos dados entre as aplicações e garante que os dados possam existir mais tempo, algumas vezes mais tempo que a própria aplicação.

A linguagem que os bancos de dados relacionais entendem é a SQL, por isto algumas vezes utiliza-se o nome de sistema de gerenciamento de banco de dados SQL, ou até mesmo banco de dados SQL.

2.2. SQL

Entender a linguagem de um banco de dados relacional é essencial para se garantir a integridade dos dados, mas muitas vezes é difícil lembrar as SQLs que não são definidas igualmente por todos os tipos de bancos.

Em uma aplicação Java que utiliza banco de dados relacional, deve conter a API JDBC que será utilizada para ligar os argumentos afim de preparar os parâmetros necessários para executar a consulta, recuperar valores em conjunto de resultados, ou seja, ter acesso aos dados.

Quando desenvolve-se uma aplicação, o principal foco é a lógica de negócio, de maneira que o desenvolvedor não se preocupe tanto com a camada persistências dos dados.

Uma aplicação orientada à objetos pode conter objetos persistentes. Mas a maioria dos objetos não são persistentes e estes possuem um tempo de vida limitado. Conhecidos por objetos transientes, que possuem um tempo de vida restrito ao processo que o criou, ou seja, finalizando o processo o objeto “morre”.

Então, pode-se dizer que uma aplicação é formada por objetos persistentes e transientes, sendo que, serão tratados de maneiras diferentes e faz-se necessário o uso de um subsistema para gerenciá-los.

Os bancos de dados relacionais oferecem uma representação estruturada dos dados à serem persistidos e através deles pode-se manipular, classificar, buscar e agrupar os dados. O seu sistema de gerenciamento garante a integridade através do uso de restrições (*constraints*) e do compartilhamento dos dados com os demais usuários, ou seja, o sistema de gerenciamento irá fornecer segurança à nível de dados.

Em uma aplicação orientada à objetos não se trabalha diretamente com o resultado do SQL, ao invés de trabalhar diretamente com o conjunto de resultados, a lógica de negócios trabalha com o modelo de domínio OO (orientado à objeto) e com sua representação em tempo de execução, como se fosse uma rede de objetos

interconectados. Esse lógica de negócio não é executada diretamente no banco de dados, ela é implementada em uma camada física da aplicação, permitindo assim que está camada tenha acesso a mais conceitos sofisticados, como herança e polimorfismo.

Mas nem todas as aplicações Java são “desenhadas” desta forma, para aplicações triviais o modelo de domínio é bastante utilizado, pois ele ajuda a melhorar a reutilização e a manutenção do código.

Sistemas de gerenciamento de banco de dados relacionais é a única tecnologia de dados comprovada, porém elas também possuem falhas, destacando-se a disparidade de paradigma entre orientação a objetos e objetos relacionais.

2.3. DISPARIDADE DO PARADIGMA

Na orientação a objetos são utilizados conceitos de engenharia de software, como por exemplo herança, polimorfismo, encapsulamento, entre outras. Mas em um banco relacional alguns destes recursos estão restritos. Com base no livro de Christian Bauer e Gavin King (Java Persistence com Hibernate, capítulo 2), esta sessão irá destacar algumas diferenças.

2.3.1. Problema de granulosidade

A orientação a objetos permite que uma entidade faça referência a um objeto de outras classes, mas para representar este modelo em um banco de dados relacional seria necessário criar uma tabela com os atributos daquela entidade e uma coluna que representaria os atributos da outra classe, como se fossem duas tabelas em uma só. A granulosidade refere-se ao tamanho relativo dos tipos de dados e desta maneira a aplicação conteria um nível de granulosidade maior, diminuindo assim a performance da aplicação.

2.3.2. Problema de subtipos

Quando se implementa uma herança usando uma linguagem orientada a objetos, como por exemplo Java, o desenvolvedor precisa criar a superclasse e depois as subclasses que irão ter acesso as características da superclasse. Para representar isto em linguagem SQL, é necessário incluir uma supertabela para cada superclasse e uma subtabela para cada subclasse, a supertabela irá criar colunas virtuais afim de armazenar as novas colunas da subtabela. O conceito de colunas virtuais deve ser aplicado a tabelas virtuais e não a tabelas básicas. Além do que, a herança aplicada no Java é uma herança de tipo e uma tabela não é tipo.

Sempre que existir uma herança, existirá a possibilidade de realizar polimorfismo. Bancos de dados SQL não possuem um padrão para representar associações polimórficas. Uma chave estrangeira irá fazer referência somente para uma tabela, para que esta fizesse referência seria necessário escrever uma restrição procedimental para forçar este tipo de regra.

2.3.3. Problema de identidade

Em Java, quando necessita comparar dois números é utilizado o comparador (==) e quando precisa comparar dois objetos é utilizado o método equal(). Em SQL essas operações não são equivalentes ao valor da chave primária.

2.3.4. Problema de associações

Em um banco de dados relacional a associação é representada, por meio de uma chave estrangeira, como se fosse uma cópia da chave primária da tabela de referência, já na orientação a objetos a associação é representado por referências entre os objetos.

A associação é feita de um objeto para o outro, como se fossem ponteiros, portanto, se o diagrama conter uma associação bidirecional ela deverá definir duas vezes a

associação, uma para cada classe associada.

Associações bidirecionais não podem ser representadas em um banco de dados relacional, toda vez que existir uma associação bidirecional, ela será representada por uma tabela de ligação entre as tabelas relacionadas, ou seja, na orientação a objetos a associação pode ter multiplicidade muitos-para-muitos e em um banco de dados relacional pode conter apenas multiplicidade um-para-muitos e um-para-um.

2.3.5. Problema de navegação de dados

Geralmente os dados em Java são acessados pelos seus métodos get, é a maneira mais utilizada entre os desenvolvedores, ou seja, navegar entre um objeto e outro. Esta navegação no banco de dados relacional é realizada através de uma SQL que acessa todas as tabelas a serem utilizadas. Muitas vezes este comando SQL, tornam-se muito grande, e para melhorar seu tempo de resposta deve-se minimizar o número de requisições ao banco de dados e a maneira mais simples de realizar isto, é diminuindo o número de SQL's.

2.3.6. Custo de disparidade

Parte de qualquer aplicação, incluindo Java, é destinada ao desenvolvimento de SQL's e reduzir a disparidade entre objeto/relacional. Tomando muito tempo dos desenvolvedores e DBA(s).

Segundo o livro Java Persistence com Hibernate (2005, p. 19),

> O propósito principal de aproximadamente 30 por cento do código escrito de uma aplicação Java é para tratar o tedioso SQL/JDBC e criar uma 'ponte' manualmente da disparidade do paradigma objeto/relacional.

Um modelo de domínio relacional devem conter as mesmas entidades, mas o desenvolvedor montará o modelo de maneira diferente a de um DBA. Para equalizar esta disparidade, alguns ajustes de ambas as partes deverão ser feitos, ou seja,

algumas “qualidades” da orientação a objetos serão perdidas, para manter a integridade dos dados.

Outro tópico importante está na própria API (Application Programming Interface) JDBC (Java Database Connectivity) e o SQL, que são orientadas para realizar comandos, ou seja, para realizar um consulta ou manipular dados no banco de dados as tabelas envolvidas serão consultadas pelo menos três vezes (insert, update, select) aumentando assim o tempo de implementação. É também importante destacar, que cada banco de dados relacional possui seu próprio dialeto.

2.4. ARQUITETURA EM CAMADAS

Como uma aplicação possui muitas classes com diferentes funcionalidade é interessante agrupá-las de acordo com suas funcionalidades. Criando uma camada para cada funcionalidade, de modo que as modificações realizadas em uma camada não afete as outras.

A distribuição em camadas determinam o tipo de dependências que ocorrem entre elas, de forma que, elas comunicam-se do topo para a base, ou seja, uma camada é dependente somente da camada que está abaixo dela e cada camada não sabe das outras camadas existentes a não ser a que está abaixo dela.

A arquitetura em camadas mais utilizada é a de três camadas (MVC – Model, View, Controller), que contém uma camada de *apresentação*, camada do topo responsável pela interface com o usuário; a camada de *lógica do negócio*, está pode variar de acordo com a aplicação. Geralmente é utilizada para implementar as necessidades do sistemas, ou seja a regra de negócio, algumas vezes está camada possui um componente de controle que vai chamar a regra de negócio para solucionar o problema invocado. Por fim, a camada de *persistência,* que possui um conjunto de classes e componentes responsáveis por guardar, recuperar dados de um ou mais repositórios de dados.

O padrão de três camadas mais comum é o DAO (Data Access Object), pois este

trabalha com o JDBC e o SQL, escondendo a complexidade do código JDBC e a SQL da camada de negócio, porém tem-se que codificar manualmente a persistência para cada classe de domínio, sendo assim, muito cansativo. E quando ocorre alguma manutenção em um código escrito manualmente precisa-se de um pouco mais atenção e de esforço.

2.5. SISTEMAS DE BANCO DE DADOS ORIENTADOS PARA OBJETOS

Um sistema de gerenciamento de banco de dados orientado para objetos (SGBDOO's) funciona como se fosse uma extensão ao ambiente de uma aplicação, ele apresenta uma implementação de n-camadas com cache de objetos e um repositório de dados *back-end*, e interagem via protocolo de rede proprietário. Ele guarda uma rede de objetos, com todos os nós e ponteiros que são armazenados em páginas de memória para mais tarde serem transportados para o repositório de dados.

Os bancos de dados orientados a objetos oferecem uma integração com a aplicação orientada a objetos sem a necessidade de ajustes.

2.6. MAPEAMENTO OBJETO/RELACIONAL

Rapidamente pode-se dizer que o mapeamento objeto/relacional consiste em transformar os objetos de uma aplicação Java em tabelas do banco de dados relacional. O ORM (object/relational mapping) funciona como uma “ponte” para realizar está transformação, mas ele só realiza as operações CRUD básicas (Create, Retrieve, Update, Delete) nos objetos das classes persistentes.

Esta é uma API para especificar as consultas que referem-se as propriedades das classes ou as próprias classes. Possui algumas funções de otimização como por exemplo, *dirty checking* que realiza uma checagem de sujeira.

O ORM pode ser implementado de várias formas, mas as quatro mais importantes

são o *Relacional puro*, que é baseado no modelo relacional e suas operações são baseadas em SQL, somente é utilizado em aplicações simples, pois apresenta falta de portabilidade e manutenibilidade, especialmente se for em um longo prazo; o *Mapeamento leve,* são entidades representadas como classes que serão mapeadas manualmente para as tabelas relacionais, muito utilizado em pequenas aplicações, porque possuem um número pequeno de entidades; o *Mapeamento médio,* onde a aplicação é desenhada a partir do modelo de objetos, o SQL pode ser gerado por uma ferramenta de geração de código, ou em tempo de execução, através de um *framework* e as consultas podem ser realizadas utilizando uma linguagem orientada a objetos; e por fim o *Mapeamento completo* que suporta objetos sofisticados, como herança, polimorfismo, composição. A camada de persistência realiza uma implementação de maneira transparente, estratégias de recuperação e estratégias de cacheamento são implementadas diretamente para a aplicação.

Um desenvolvedor de orientação a objetos deve saber as SQL's básicas e não preocupar-se tanto com SQL's extensas, esta é um das vantagens que o ORM e o Hibernate trazem. Além de ter maior produtividade, estes eliminam tarefas rotineiras, deixando o desenvolvedor concentrar-se na regra de negócio. Possui também maior manutenibilidade, afinal reduz as linhas de código, tornando o sistema mais legível e de fácil entendimento. Proporciona mais agilidade, afinal o código é gerado automaticamente e irão gerenciar as requisições ao banco de dados, tornando as consultas mais rápidas, e sem se preocupar. E possui também a independência de fornecedor, ou seja, o cliente pode trocar de banco de dados relacional quando ele desejar, mudando apenas o dialeto do banco escolhido na aplicação.

3. O HIBERNATE

O *framework* Hibernate foi criado por desenvolvedores Java de todo o mundo, liderados por Gavin King. Disponível também para aplicações .Net, mas com o nome Nhibernate. Uma ferramenta *opensource* de código aberto que tem como principal objetivo transformar classes Java em tabelas de banco de dados, reduzir o trabalho de persistência dos desenvolvedores permitindo assim que eles prestem mais atenção à camada de negócios do projeto.

3.1. MÓDULO DE SOFTWARE DO HIBERNATE

O Hibernate implementa o Java Persistence por isso surgiram novas combinações de software, e dependendo da necessidade do desenvolvedor pode ser utilizada mais de uma tecnologia ao mesmo tempo.

3.1.1. Hibernate Core

Também conhecido como Hibernate3.2.x tem um serviço básico para a persistência, utiliza arquivos de mapeamento XML (Extensible Markup Language) e possui sua API nativa. Utiliza a linguagem de consulta chamada HQL (Hibernate Query Language), muito semelhante ao SQL. Ele é independente de qualquer *framework* e pode ser executado a partir de qualquer JDK (Java Development Kit). Funciona em qualquer servidor JEE/J2EE (Java Enterprise Edition) e em aplicações *swing*.

3.1.2. Hibernate Annotations

São anotações embutidas diretamente no código fonte, na camada de negócio, ficou disponível com o JDK5.0, a sintaxe e semântica das anotações são muito parecidas com as *tags* utilizadas nos arquivos de mapeamento XML do Hibernate Core. Mas

as anotações básicas não são proprietárias.

O Hibernate Annotations é um conjunto de anotações básicas que implementam o padrão JPA (API Java Persistence), um conjunto de anotações que necessitam de mapeamentos e ajustes avançados do Hibernate.

3.1.3. Hibernate EntityManager

A interface de JPA também pode ser utilizada para definir interfaces de programação, regras de ciclo de vida para objetos persistentes e características de uma consulta. Esta implementação de interfaces também está disponível para EntityManager, que pode ser utilizada por uma interface que utiliza Hibernate puro, ou até mesmo uma Connection do JDBC, ou seja, possui compatibilidade com arquivos de mapeamento XML ou com mapeamento mais avançados do JPA.

3.2. INICIANDO UM PROJETO

A máquina deve estar configurada, ou seja, possuir o Hibernate instalado (http://www.hibernate.org/). É necessário um sistema de gerenciamento de banco de dados relacional de sua escolha. Deixe separado o *driver* JDBC do banco escolhido.

Para a utilização do Hibernate será necessário algumas bibliotecas, tais como:

- antlr.jar
- asm.jar
- asm-attrs.jars
- c3p0.jar
- cglib.jar
- commons-collections.jar
- commons-loggin.jar
- dom4j.jar
- hibernate3.jar
- jta.jar
- <JDBC do banco escolhido>

Crie uma pasta *lib* dentro do projeto e cole todas estas bibliotecas dentro desta, assim não acontecerá de mudar alguma biblioteca de lugar e mais tarde ocorrer problemas de referência.

Todas as aplicações Hibernate definem as classes persistentes que serão mapeadas em tabelas do banco de dados. As classes devem ser baseadas na análise do modelo de domínio, ou seja, baseado na tabela do banco de dados.

Abaixo será apresentado um exemplo “Cliente” que consiste em uma classe JavaBean (Figura 2), e seu objetivo é mostrar os atributos de um “cliente”.

```
3   public class Cliente {
4
5       private Integer codigo; //atributo identificador do cliente
6       private String nome; //nome do cliente
7       private String endereco; //endereço do cliente
8
9       public Integer getCodigo() {
10          return codigo;
11      }
12
13      public void setCodigo(Integer codigo) {
14          this.codigo = codigo;
15      }
16
17      public String getEndereco() {
18          return endereco;
19      }
20
21      public void setEndereco(String endereco) {
22          this.endereco = endereco;
23      }
24
25      public String getNome() {
26          return nome;
27      }
28
29      public void setNome(String nome) {
30          this.nome = nome;
31      }
32  }
```

Figura 1. Exemplo de uma classe Bean

A partir do bean (Figura 1) é criado o arquivo de mapeamento (Figura 2), como se fosse uma copia dele, afinal este é o arquivo que o Hibernate utilizará para acessar

os dados no banco de dados. É neste arquivo que é realizado a conversão do tipo dos atributos para o respectivo banco de dados. Este será o arquivo XML que vai informar ao Hibernate qual tabela no banco de dados ele deverá acessar. Quais as colunas da tabela ele deverá usar, todas as relações que uma tabela pode ter deve ser representada aqui, como por exemplo, relacionamentos ou até mesmos campos que representem outra tabela - chave estrangeira.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4  <hibernate-mapping>
5    <class name="comecando.Cliente" table="cliente">
6      <id column="cliente_id" name="codigo" type="java.lang.Integer">
7        <generator class="increment"/>
8      </id>
9      <property column="cliente_nome" name="nome" type="string"/>
10     <property column="endereco" name="endereco" type="string"/>
11   </class>
12 </hibernate-mapping>
```

Figura 2. Cliente.hbm.xml – Exemplo de uma classe persistente

Com este XML, o Hibernate possui informações suficientes para inserir, alterar, excluir e recuperar as instâncias da classe Cliente.

O arquivo de mapeamento é um arquivo XML, mas antes da extensão *.xml* tem-se uma extensão do tipo .hbm que é apenas uma nomenclatura que a comunidade do Hibernate utiliza para representações dos arquivos de mapeamento.

O mapeamento da classe começa a partir da tag <hibernate-mapping> e será finalizada com a tag </hibernate-mapping>. Em seguida tem-se a tag <class>, que mostrará o nome da classe que está mapeando e da tabela do banco que representa. Dentro dela será apresentado os atributos da classe, cada atributo pode ou não apresentar o argumento *column*, pois caso não for especificada o Hibernate entenderá que possui o mesmo nome. O atributo código possui a tag <generator> que indicará que meu código será do tipo auto-incremento.

Portanto para que o *framework* possa guardar e recuperar os objetos é necessário implementar uma classe Main (Figura 3) para realizar um teste.

```java
package comecando;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class TestandoCliente {

    public static void main(String args[]) {

        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();
        Cliente novoCliente = new Cliente();
        novoCliente.setCodigo(1);
        novoCliente.setNome("Simone");
        novoCliente.setEndereco("R. 7 de Setembro");
        try {
            tx.commit();
        } catch (HibernateException ex) {
            tx.rollback();
            ex.printStackTrace();
        }
        session.close();
        HibernateUtil.shutdown();
    }
}
```

Figura 3. TestandoCliente.java – Classe para testar um cliente

Esta classe chama a interface Session, que é utilizada para recuperar e guardar objetos, internamente ela funciona como se fosse uma fila de declarações SQLs que devem ser sincronizadas, ou seja, monitoradas pela Session. A interface Transaction é opcional, mas é utilizada para definir limites de transação. A interface Query também é acessada, não diretamente, mas é ela que realiza a consulta ao banco de dados e pode ser escrita utilizando a própria linguagem orientada a objetos do Hibernate (HQL) ou até mesmo em SQL.

O modo mais comum de iniciar o Hibernate é criar um objeto SessionFactory a partir de um objeto Configuration. O Configuration é um arquivo de configuração, ou um arquivo de propriedade. Este arquivo de configuração também é desenvolvido pelo programador como mostra a Figura 4, geralmente estes arquivos de configuração são nomeados como hibernate.properties e fica localizado na raiz do classpath, se o desenvolvedor colocar ele em um pacote, este deverá ser informado.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
    DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
      <!-- No exemplo esta sendo utilizado o banco de dados Mysql  -->
    <property name="hibernate.connection.driver_class">
        com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">
        jdbc:mysql://127.0.0.1:3306/hibernate
    </property>
    <property name="hibernate.dialect">
        org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.username">
        root
    </property>
    <property name="hibernate.connection.password">
        mysql
    </property>
    <!-- Mostra as SQLs no console -->
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    <!-- Lista de todos os arquivos XML -->
    <mapping resource="comecando/Cliente.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Figura 4. hibernate.properties – Arquivo de configuração do Hibernate

Este é o arquivo que estabelece contato com o banco de dados, que gerencia as consultas ao banco e gerencia também o tempo de conexão. No fim do arquivo é obrigatório aparecer uma lista de todos os arquivos hbm que o projeto possui.

E por fim tem-se o HibernateUtil (Figura 5) esta classe será chamada para iniciar o Hibernate.

```java
package comecando;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory = null;
    static {
        try {
            Configuration c = new Configuration().configure();
            sessionFactory = c.buildSessionFactory();
        } catch(Throwable ex) {
            ex.printStackTrace();
        }
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

Figura 5. HibernateUtil.java – Utilizada para iniciar o Hibernate.

3.3. COMO USAR HIBERNATE ANNOTATIONS

O Hibernate Annotation será utilizado para substituir os arquivos de mapeamento XML, pois as anotações estarão dispostos dentro do próprio arquivo bean. Como mostra a Figura 6.

```java
package comecando;

import java.io.Serializable;
import javax.persistence.*;

@Entity
@Table(name = "cliente")
public class Cliente implements Serializable {
    @Id @GeneratedValue
    @Column(name="cliente_codigo")
        private Integer codigo;
    @Column(name="cliente_nome")
        private String nome;
    @Column(name="endereco")
        private String endereco;

    public Cliente() {
    }

    public Integer getCodigo() {
        return codigo;
    }
    public void setCodigo(Integer codigo) {
        this.codigo = codigo;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
}
```

Figura 6. Cliente.java – Classe Cliente reescrita, agora utilizando anotações

As anotações iniciam-se por arroba (@), por exemplo, assim que se declara a classe já se nomeia a table(@Table), ou seja, seta o nome da tabela. O código utiliza o @Id que indica para o Hibernate que aquele campo é do tipo primary-key. E o @GeneratedValue indica que o código é do tipo auto-incremento. Resumindo, tudo o que podia ser feito no arquivo de mapeamento poderá ser realizado com alguma Annotation, com uma diferença, um arquivo a menos para desenvolver.

Além de excluir o arquivo de mapeamento XML deverá ser realiza uma alteração no arquivo de configuração do Hibernate. Ao invés de mapear o XML chame direto o arquivo bean. Veja como ficaria: a expressão <mapping class=”Cliente.hbm.xml” /> sairia e entraria a tag <mapping class=”Cliente” />, ou seja, está voltado diretamente para o objeto Cliente.

4. PERSISTÊNCIA GENÉRICA

Segundo o modelo MVC a camada de persistência é responsável por guardar, alterar, excluir e selecionar um ou mais dados de um repositório. Como mencionado neste trabalho, o padrão de três camadas mais utilizado é DAO. Este padrão impõe ao desenvolvedor criar uma classe de persistência para cada tabela do banco de dados. As ações destas classes são sempre as mesmas, ou seja, cada classe terá seu próprio método de inserir, alterar, excluir e seleção, persistindo sempre a um tipo de objeto.

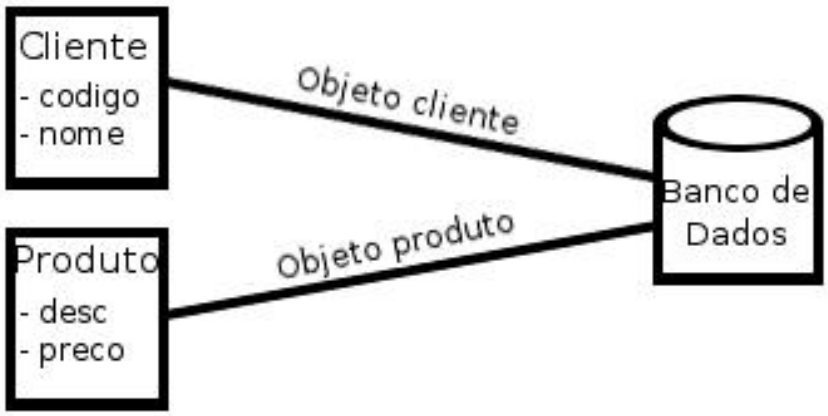

Figura 7. Representação dos diferentes tipos de objetos

A Figura 7 mostra o objeto cliente do tipo Cliente e o objeto produto do tipo Produto, como eles são de tipos diferentes é necessário uma classe de persistência para cada um. Caso o desenvolvedor deseje selecionar todos os clientes da classe de lógica de negócio, invocaria apenas o método de selecionar Clientes, que consultaria a tabela de clientes por exemplo. Se o mesmo fosse realizar a consulta em produtos a tabela consultada seria a tabela de produtos.

Num sistema com muitas tabelas, muitas classes de persistência seriam criadas, exigindo tempo de programação para realizar a mesma tarefa. Por meio de uma classe de persistência “genérica”, o desenvolvedor não precisa criar uma classe para cada tabela, ele utilizará apenas uma classe independente, para todos os tipos do objeto.

Com o Java, o desenvolvedor encontra um recurso que permite essa independência de tipos de objetos. Através do uso de Generics, pode-se definir classes, métodos ou coleções que seja de um tipo “X”, não determinados explicitamente, onde “X”, será uma instância da classe que deseja persistir ou recuperar dados.

Outro recurso que Java dispõe, para a construção da persistência é a serialização, que ajuda na recuperação e gravação dos dados. A serialização de objetos realiza o processo de conversão de um objeto em uma sequência de bytes. Onde um dos bytes será responsável por armazenar o estado do objeto, ou seja, se o objeto é do tipo transiente – objeto comum da aplicação, ou caso, ele é do tipo persistente – objeto a ser persistido no banco de dados ou em um arquivo, ou do tipo desligado – sem nenhum tipo definido.

Como apresentado no capítulo anterior, todos os métodos de acesso aos dados recebem um objeto do tipo Serializable como parâmetro e usam o método getSession() da classe HibernateUtil. Responsável por inicializar o Hibernate, ao final das operações é necessário chamar o método closeSession(), responsável por fechar a sessão, liberando todos os recursos do Hibernate, alterando assim o estado de todas as instâncias persistentes para desligado.

No inicio de todos os métodos o atributo *session* será instanciado, atribuindo a ele as configurações do banco de dados, por meio do método HibernateUtil.getSession(). Ao final de todos os métodos o método closeSession() será executado, responsável por fechar a conexão com o Hibernate. Os métodos que persistem um objeto no banco de dados: inserir, alterar, excluir e merge; precisam iniciar uma Transaction, pois, ela é necessária para executar as declarações SQL. A execução dela, só acontece quando o método commit() na Transaction é executada. Caso, algum erro ocorrer, será lançada uma exceção automaticamente, então é utilizado o método rollback(), revertendo a transação.

O método insert (Figura 8) realiza a inserção do objeto recebido por parâmetro no banco de dados, através do método save(), se ocorrer algum erro durante a inserção será lançada uma exceção e o objeto não será gravado retornando false, caso contrário, retorna true, ou seja, o objeto foi inserido com sucesso.

```
45      public boolean insert(Serializable obj){
46          try {
47              session = HibernateUtil.getSession();
48              transaction = session.beginTransaction();
49              session.save(obj);
50              transaction.commit();
51              retorno = true;
52          } catch (HibernateException e) {
53              retorno = false;
54              transaction.rollback();
55              e.printStackTrace();
56          } finally {
57              closeSession();
58          }
59          return retorno;
60      }
```

Figura 8. Método inserir da persistência genérica

O método update (Figura 9) realiza à alteração do objeto recebido por parâmetro no banco de dados, através do método update(), caso ocorra algum erro durante a alteração, será lançada uma exceção e o objeto não será alterado, retornando false, caso contrário retorna true, ou seja, o objeto foi alterado com sucesso.

```
70      public boolean update(Serializable obj) {
71          try {
72              session = HibernateUtil.getSession();
73              transaction = session.beginTransaction();
74              session.update(obj);
75              transaction.commit();
76              retorno = true;
77          } catch (HibernateException e) {
78              retorno = false;
79              transaction.rollback();
80              e.printStackTrace();
81          } finally {
82              closeSession();
83          }
84          return retorno;
85      }
```

Figura 9. Método alterar da persistência genérica

O método merge (Figura 10) realiza à inserção do objeto recebido por parâmetro no banco de dados. Caso objeto existir, é realizado apenas a alteração, através do método merge(). Caso ocorra algum erro será lançada uma exceção.

```
93      public void merge(Serializable obj) throws HibernateException {
94          try {
95              session = HibernateUtil.getSession();
96              transaction = session.beginTransaction();
97              session.merge(obj);
98              transaction.commit();
99          } finally {
100             closeSession();
101         }
102     }
```

Figura 10. Método salve/update da persistência genérica

O método delete (Figura 11) realiza a exclusão do objeto recebido por parâmetro do banco de dados, através do método delete(). Caso ocorra algum erro durante a exclusão será lançada uma exceção e o objeto não será excluído, retornando false, caso contrário retorna true, ou seja, o objeto foi excluído com sucesso.

```
112     public boolean delete(Serializable obj) {
113         try {
114             session = HibernateUtil.getSession();
115             transaction = session.beginTransaction();
116             session.delete(obj);
117             transaction.commit();
118             retorno = true;
119         } catch (HibernateException e) {
120             retorno = false;
121             transaction.rollback();
122             e.printStackTrace();
123         } finally {
124             closeSession();
125         }
126         return retorno;
127     }
```

Figura 11. Método excluir da persistência genérica

Os métodos de seleção (Figuras 12, 13 e 14), utiliza-se de polimorfismo para retornar os objetos desejados. Eles retornam uma lista de objetos que serão do mesmo tipo do objeto recebido por parâmetro. Todos estes métodos, utilizam o método createQuery() do Hibernate, onde está lista de resultados será armazenada em objeto do tipo Query que ao final do método retorna uma lista de objetos.

```
137     public <T extends Serializable> List<T>
138             selectAll(T obj) throws HibernateException {
139         List<T> list = null;
140         try {
141             session = HibernateUtil.getSession();
142             Query query = session.createQuery("from " + obj.getClass().getName());
143             list = query.list();
144         } finally {
145             closeSession();
146         }
147         return list;
148     }
```

Figura 12. Método selecionar todos

```
181     public <T extends Serializable> List<T>
182             selectAll(T obj, String column, Long value) throws HibernateException {
183         List<T> list = null;
184         try {
185             session = HibernateUtil.getSession();
186             Query query = session.createQuery("from " + obj.getClass().getName() +
187                     " where " + column + " = " + value);
188             list = query.list();
189         } finally {
190             closeSession();
191         }
192         return list;
193     }
```

Figura 13. Método selecionar todos com uma cláusula where

```
358     public <T extends Serializable> List<T>
359             selectAll(Vector campos, Vector registros, T obj) throws HibernateException {
360
361         String camporeg = "";
362         //concatena a primeira cláusula where
363         if (registros.elementAt(0) instanceof String) {
364             camporeg = (String) campos.elementAt(0) + " = '" + (String) registros.elementAt(0) + "'";
365         } else {
366             camporeg = (String) campos.elementAt(0) + " = " + String.valueOf(registros.elementAt(0));
367         }
368         //concatena as demais cláusulas where
369         for (int cont = 1; cont < campos.size(); cont++) {
370             if (registros.elementAt(cont) instanceof String) {
371                 camporeg = camporeg + " and " + (String) campos.elementAt(cont) +
372                         " = '" + (String) registros.elementAt(cont) + "'";
373             } else {
374                 camporeg = camporeg + " and " + (String) campos.elementAt(cont) +
375                         " = " + String.valueOf(registros.elementAt(cont));
376             }
377         }
378
379         List<T> list = null;
380         try {
381             session = HibernateUtil.getSession();
382             Query query = session.createQuery("from " + obj.getClass().getName() + " where " + camporeg);
383             list = query.list();
384         } finally {
385             closeSession();
386         }
387         return list;
388     }
```

Figura 14. Método selecionar todos com várias cláusulas where

O método de selectOne (Figura 15) realização a seleção de um único objeto no banco de dados, através do método get(), do mesmo tipo do objeto que é recebido

como parâmetro, retornando ao método que fez sua invocação. O segundo parâmetro recebido pelo método é a chave primária que será utilizada para encontrar o objeto.

```
226       public <T extends Serializable> T
227               selectOne(T obj, Serializable key) throws HibernateException {
228           try {
229               session = HibernateUtil.getSession();
230               obj = (T) session.get(obj.getClass().getName(), key);
231           } finally {
232               closeSession();
233           }
234           return obj;
235       }
```

Figura 15. Método que seleciona apenas um objeto

O método de selectLikeAll (Figura 16) procura na coluna (*column*) pela String que é recebida por parâmetro (*value*) e retorna uma lista de objetos que serão do mesmo tipo do objeto recebido por parâmetro.

```
288       public <T extends Serializable> List<T>
289               selectLikeAll(T obj, String column, String value) throws HibernateException {
290           List<T> list = null;
291           try {
292               value = "%" + value + "%";
293               session = HibernateUtil.getSession();
294               Criteria criteria = session.createCriteria(obj.getClass());
295               criteria.add(Restrictions.like(column, value).ignoreCase());
296               list = criteria.list();
297           } finally {
298               closeSession();
299           }
300           return list;
301       }
```

Figura 16. Método que procura por uma String em uma determinada coluna

O método de executeSQL (Figura 17) executa qualquer instrução SQL nativa desde que esta respeite um padrão.

SELECT * FROM <nomeTabelaBancoDados> <apelidoTabela> WHERE <apelidoTabela>.<nomeCampoBancoDados> = <valor>

Se a instrução SQL consultar mais de uma tabela, as mesmas deveram ser definidas entre vírgulas e cada uma com seu respectivo apelido.

É recomendado que a instrução SQL, faça sempre “select * “. Caso a tabela tenha

chave estrangeira o Hibernate realizará o select nas tabelas, onde foram especificadas a chave estrangeira. Senão, será lançada uma exceção.

Caso tenha-se que especificar somente alguns atributos em ambas as tabelas, é necessário especificar os apelidos das tabelas para tal retorno. Outras instruções de SQL (ORDER BY, GROUP BY, HAVING, etc), poderão ser utilizadas, desde que respeite os apelidos dados as tabelas. Ao final será retornado um list de objeto do mesmo do objeto recebido como parâmetro.

```
351         public <T extends Serializable> List<T>
352                 executeSQL(String sql, T obj) throws HibernateException {
353             List<T> list = null;
354             try {
355                 session = HibernateUtil.getSession();
356                 SQLQuery query = session.createSQLQuery(sql).addEntity(obj.getClass().getName());
357                 list = query.list();
358             } finally {
359                 closeSession();
360             }
361             return list;
362         }
363     }
```

Figura 17. Método que executa qualquer SQL

4.1. ESTUDO DE CASO

Durante o desenvolvimento da persistência genérica, notou-se a necessidade de testes comportamentais e desempenho. Assim, a persistência foi implantada em alguns sistemas computacionais, para avaliar suas vantagens. O aluno concluinte, Jabes Felipe Cunha do 3º ano de Processamento de Dados 2009 da FEMA desenvolveu um sistema de Orçamento Web. No trabalho de conclusão de curso do aluno, utilizou-se a persistência genérica com grande sucesso.

> Segundo Jabes, A utilização dos métodos de insert, delete e update possui uma boa produtividade no desenvolvimento na lógica de negócios, para telas de CRUD padrão. Entretanto, para a recuperação dos dados em mais de uma tabela, o desempenho não é tão eficiente pelo trabalho de escrever a SQL necessária.

Outro aluno concluinte de 2009, Rodrigo Merlin do 4º ano de Ciência da Computação da FEMA também utilizou a classe de persistência genérica em seu trabalho de conclusão. Aplicou a classe genérica na geração dos relatórios do

sistema de Orçamento do aluno Jabes, voltado para Web. Naquele trabalho o desenvolvedor utilizou os métodos de inserir, excluir e selecionar da persistência genérica, de modo que, a partir do momento que o usuário solicitar um relatório será enviado uma solicitação à tabela de relatórios. Será realizada uma requisição do relatório e manutenção na tabela de relatórios. Os registros serão apagados, e inseridos na tabela de relatórios de acordo com o relatório solicitado.

> Segundo o aluno Rodrigo Merlin, o uso da persistência genérica reduziu muito tempo da programação, não preocupar com a escrita de SQL. Outro ponto positivo, foi trabalhar com os objetos completos ao invés de trabalhar apenas com o código de uma determinada chave estrangeira, não fazer o select várias vezes.

No CEPEIN (Centro de Pesquisa e Informática) da FEMA foi desenvolvido um sistema *desktop* em Java para o setor de Xerox e para Sala de Impressão da FEMA, onde usuário do departamento, digita os lançamentos de xerox e ou impressões realizadas por um determinado aluno.

Outro sistema que utilizou a persistência genérica no CEPEIN, foi o sistema de Reservar Recursos ou equipamentos para uso em sala de aula ou palestras tais como: salas de aulas, projetores, computadores e etc. Funcionários e professores podem reservar um recurso através do sistema Web.

5. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho seguiu o padrão de desenvolvimento software OO, utilizando a linguagem Java, respeitando o modelo MVC. A construção de uma persistência genérica atuando na camada *Controller*. Esta persistência resolveu alguns dos problemas encontrados no desenvolvimento com DAO's e SQL puro.

Por meio de flexibilidade de utilização de API's externas em Java, recursos da orientação à objetos e do *framework* Hibernate foi possível desenvolver a classe de persistência genérica. Atuando apenas na camada de controle ela reduz o tempo de desenvolvimento de sistemas, afinal não é mais necessário criar uma classe de persistência para cada entidade. Permitindo que o desenvolvedor dedique mais tempo a lógica de negócios da aplicação. E também ajuda na manutenção do código, pois agora o código fonte ficará mais “limpo”, as operações SQL estarão encapsuladas nos métodos da persistência.

Caso o desenvolvedor desejar escrever a SQL, ele tem disponível o método executeSQL que realiza a consulta através de um código SQL nativo.

Com a utilização do Hibernate como *framework* e a persistência genérica em todos os projetos dos alunos e implementações no CEPEIN, concluiu-se que é possível utilizar recursos da orientação à objetos, como: mapeamento das associações, que realiza o select em cascata, se uma tabela possui uma determinada chave estrangeira. A mesma é definida como tipo da tabela relacionada, proporcionando ao Hibernate, selecionar os campos desta tabela, e assim por diante, trabalhando somente com objetos, facilitando a realização de herança, polimorfismo, associação.

Através do uso de Generics foi possível realizar está flexibilidade entre os sistemas, utilizando a mesma classe, pois cada sistema atendia uma necessidade diferente e mesmo assim foi possível utilizar a mesma classe de persistência. Mantendo a mesma rapidez no processamento dos dados, seja para inserir ou excluir objetos em massa. Além reduzir o tempo dedicado à programação da camada de persistência, direcionando o desenvolvedor a camada de lógica de negócio.

Depois de implantada, a persistência mostrou-se eficiente, entretanto, detectou-se alguns problemas na consulta de tabelas que possuem várias chaves estrangeiras, porque o Hibernate seleciona os dados em todas as tabelas associadas, ocasionando a consulta um pouco mais lenta.

Portanto, agora é realizar pesquisas entre comandos avançados do Hibernate, afim de, analisar uma maneira de controlar as execuções dos selects com várias chaves estrangeiras.

6. REFERÊNCIAS BIBLIOGRÁFICAS

BAUER, Christian; KING, Gavin. **Java Persistence com Hibernate**. Rio de Janeiro: Editora Ciência Moderna Ltda, 2007.

GLOBALCODE. **AW5 – Desenvolvimento da camada de persistência com Hibernate.** São Paulo, 2005.

GONÇALVES, Edson. **Desenvolvendo aplicações Web com JSP, Servlets, JavaServer Faces, Hibernate, EJB 3 Persistence e AJAX**. Rio de Janeiro: Editora Ciência Moderna Ltda, 2007.

HEMRAJANI, Anil. **Desenvolvimento ágil em Java com Spring, Hibernate e Eclipse**. Tradução de Edson Furmankiewicz & Sandra Figueiredo. São Paulo: Editora Person Prentice Hall, 2007.

OSWALD, Vincent C. **Oracle – Banco de Dados Relacional e Distribuído – Ferramentas para Desenvolvimento**. São Paulo: Editora Makron Books, 1995.

SOURCEFORGE. **Documentação de referência do Hibernate 3.2**. sourceforge.net . Disponível em <http://sourceforge.net/projects/hibrefptbr>. Acessado em: 20 maio de 2009.

CUNHA, JABES FELIPE. **Sistema de Controle e Gerenciamento de Orçamentos**. Processamento de Dados – Fundação Educacional do Município de Assis. Assis, São Paulo, 2009.

MERLIN, RODRIGO. **Implementação de um Módulo em Java para Gerar Listagem e Relatórios em Aplicações Desktop e Web.** Ciência da Computação – Fundação Educacional do Município de Assis. Assis, São Paulo, 2009.