UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

UTILIZAÇÃO DO FRAMEWORK HIBERNATE PARA MAPEAMENTO OBJETO/RELACIONAL NA CONSTRUÇÃO DE UM SISTEMA DE INFORMAÇÃO

ODILON HERCULANO SOARES FILHO

BLUMENAU
2006

2006/1-16

ODILON HERCULANO SOARES FILHO

UTILIZAÇÃO DO FRAMEWORK HIBERNATE PARA MAPEAMENTO OBJETO/RELACIONAL NA CONSTRUÇÃO DE UM SISTEMA DE INFORMAÇÃO

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Sistemas de Informação — Bacharelado.

Prof. Alexander Roberto Valdameri , Mestre - Orientador

BLUMENAU
2006

2006/1-16

UTILIZAÇÃO DO FRAMEWORK HIBERNATE PARA MAPEAMENTO OBJETO/RELACIONAL NA CONSTRUÇÃO DE UM SISTEMA DE INFORMAÇÃO

Por

ODILON HERCULANO SOARES FILHO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Alexander Roberto Valdameri , Mestre – Orientador, FURB
Membro:	Prof. Paulo Roberto Dias,Mestre – FURB
Membro:	Prof. – Ricardo Alencar Azambuja,Mestre – FURB

Blumenau, 12 de Julho de 2006

Dedico este trabalho a minha família que sempre acreditou em mim, que um dia eu estaria realizando o meu sonho. E também a minha esposa que em muitos momentos de desânimo foi minha fonte de equilíbrio e ajuda para continuar a minha jornada em busca do conhecimento.

AGRADECIMENTOS

À Deus, por sempre esta comigo nas horas em que eu pensei que não iria conseguir.

À minha família, que mesmo longe, nunca parou de me dar forças.

Aos meus amigos, pelos empurrões e pela ajuda

A minha esposa que sempre esteve no meu lado em todos os momentos.

A todos os meus professores, por todo conhecimento obtido.

Em especial ao meu orientador, Alexander Roberto Valdameri, por ter acreditado na conclusão deste trabalho mesmo quando as circunstâncias apontavam para a não conclusão do mesmo.

"O lucro dos nossos estudos é tornarmo-nos melhores e mais sábios.”

Michel de Montaigne

RESUMO

Acessar bancos de dados relacionais numa perspectiva orientada a objetos é um requisito comum das aplicações atuais. Enquanto o modelo relacional continuar hegemônico e os poderosos bancos de dados orientados a objetos não se tornarem populares , continuará sendo necessário o mapeamento objeto-relacional *ORM,* visando preencher essa lacuna semântica entre as estruturas de tabelas e visões e sua representação através de classes. Essa necessidade que veio a motivar a construção de *frameworks* de persistência de objetos, entre os mais populares está o *framework open-source hibernate,* e esse trabalho visa mostra como o mesmo pode ser utilizado como facilitador no desenvolvimento de sistemas de informação.

Palavras-chave: Banco de dados . Hibernate. Mapeamento objeto-relacional.

ABSTRACT

To have access relationary databases in a guided perspective the objects is a common requirement of the current applications. While the relationary model to continue hegemonic and the powerful guided databases the objects if not to become popular, will continue being necessary object-relationary mapping ORM, aiming at to fill this gap semantics it enters the table structures and views and its representation through classrooms. This necessity that came to motivate the construction of frameworks of object persistence, enters most popular is framework open-source hibernate, and this work aims at sample as the same it can be used as facilitator in the development of information systems.

Key-words: Database. Hibernate. Object relation mapping.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ORM – Object Relation Mapping

XML – eXtensible Markup Language

UML – Unified Modeling Language

SGBD – Sistema Gerenciador de Banco de Dados

SQL – Structured Query Language

JDBC – Java Database Connectivity

HQL – Hibernate Query Language

API – Application Programming Interface

DTD – Document Type Definitions

JDO – Java Data Object

SUMÁRIO

1 INTRODUÇÃO

Hoje em dia com a necessidade cada vez maior de se desenvolver sistemas complexos e que prezem pela facilidade de manutenção dos programas e pelo reaproveitamento de código-fonte, muitos desenvolvedores e analistas de sistema têm aderido à análise e desenvolvimento orientados a objetos, os quais têm os princípios acima citados como parte de sua filosofia. Mesmo sendo um modelo consistente de desenvolvimento, ainda existem fatores que dificultam a unificação de tecnologias como o modelo de dados relacionais e modelos orientados a objetos, que comumente são utilizados em um mesmo projeto.

Segundo Bauer e King (2005, p.11) os bancos de dados relacionais estão fortemente presentes no núcleo da empresa moderna, e por os mesmos serem largamente utilizados em projetos de softwares orientados a objetos, faz-se necessário um mapeamento entre as tabelas do banco de dados e os objetos da aplicação. A fim de tornar compatível o paradigma da orientação a objetos ao paradigma de entidade e relacionamento, foram desenvolvidos *frameworks* de mapeamento objeto/relacional (*Object Relational Mapping* - ORM).

Através do uso desses *frameworks* pode-se abstrair o conceito de tabelas do banco de dados e trabalhar apenas com objetos. O *Hibernate* é um *framework open-source* de mapeamento objeto/relacional desenvolvido exclusivamente para linguagem *Java*, e suas principais vantagens são permitir que a aplicação permaneça totalmente orientada a objetos, e também fazer com que possíveis mudanças na base de dados, impliquem em um menor impacto sobre a aplicação, tendo em vista que apenas os objetos envolvidos com essa base de dados precisem ser modificados, ficando evidenciadas as vantagens da adoção do *Hibernate* na análise e no desenvolvimento de sistemas orientados a objetos, como demonstra Bauer e King (2005, p.19).

Este trabalho visa fazer um estudo sobre o *framework open-source Hibernate*, apresentando seus principais recursos, fazer uma análise geral sobre seu funcionamento e utilizá-lo para construção de um exemplo simples de sistema de informação, o objetivo é verificar se a sua utilização irá realmente trazer ganhos nas fases de análise, projeto, desenvolvimento e manutenção de sistemas.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo principal fazer uma análise dos ganhos obtidos com o uso do *framework open-source Hibernate* no desenvolvimento de sistemas de informação que utilizem a tecnologia *Java* e o paradigma de orientação a objetos, tomando como base os itens e subitens da norma NBR 13596.

Os objetivos específicos do trabalho são:

a) demonstrar todos os passos para configuração e utilização do *framework*;

b) fazer uma avaliação crítica dos recursos disponibilizados pelo *framework,* e medir em que cenários há ganho real nas fases de análise, projeto, desenvolvimento e manutenção de sistemas orientados a objetos;

c) disponibilizar um sistema de informação para exemplificar os conceitos de mapeamento objeto/relacional e os benefícios ganhos com o uso do *framework*.

1.2 MOTIVAÇÃO

A motivação surgiu através dificuldade de se projetar, implementar e manter um sistema orientado a objetos, mas ainda utilizando na camada de persistência o paradigma dos

bancos de dados relacionais, uma prática ainda muito utilizada por empresas e profissionais da área de desenvolvimento de software. SAUVÉ (2006) mostra que a grande necessidade é conseguir desenvolver softwares independentes dos bancos de dados relacionais trazendo maior flexibilidade e facilidade ao processo de implantação, demonstrando assim a importância de se ter um modelo único de desenvolvimento. Dessa forma, será possível utilizar a infra-estrutura disponível na organização, reduzindo os custos com aquisição de softwares e equipamentos de hardware. Isto se torna mais relevante, na medida em que as organizações não aceitam possuir dois ou mais bancos de dados, face às particularidades tornando o processo de gerenciamento de banco de dados ainda mais complexo. Além disso, o software deve se adaptar a seus clientes e não os clientes serem obrigados a adaptar-se ao software

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo é realizada a fundamentação bibliográfica dos temas envolvidos no trabalho, tais como: *Hibernate*, mapeamento objeto-relacional, XML e *framework*.

2.1 HIBERNATE

Bauer e King (2005, p.19) explicam que o *Hibernate* é um *framework* de persistência que tem como finalidade armazenar objetos Java em bases de dados relacionais ou fornecer uma visão orientada a objetos de dados relacionais existentes. Isso se dá porque o *framework* utiliza arquivos de configuração *XML* para fazer o mapeamento dos dados contidos nas colunas de uma tabela em uma base de dados relacional para os atributos de uma classe *Java*.

A Figura 1 ilustra o funcionamento do *framework* no mapeamento dos objetos da aplicação para as tabelas de uma base de dados.

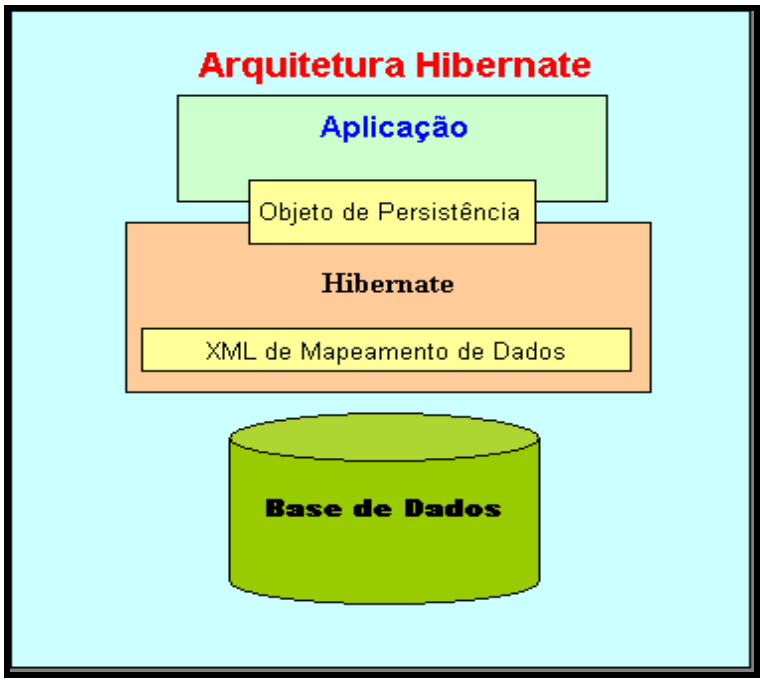

Figura 1: Arquitetura interna do *hibernate*

O objeto persistente é passado para o *hibernate* que valida sua estrutura de dados através do arquivo *XML* de mapeamento de dados, se a estrutura de dados do objeto e da tabela na base de dados estiver compatível, o *hibernate* inicia uma transação na base de dados e persiste as informações.

Em uma aplicação orientada a objetos, o *hibernate* permite que um objeto criado por uma aplicação seja armazenado em bancos de dados relacionais, conservando o estado do objeto, podendo buscar esse mesmo objeto futuramente com o mesmo estado. O *hibernate* não se limita a buscar os objetos e todas as suas referências de uma só vez, após o objeto e suas referências terem sido salvos no banco de dados, pode-se buscar apenas o objeto desejado e caso seja necessário é feita a busca por suas referências.

Uma aplicação orientada a objetos não trabalha diretamente com a representação tabular das entidades de negócio como mostram Bauer e King (2005, p.29); a aplicação tem seus próprios modelos das entidades de negócio. Se a base de dados tiver tabelas para artigo e para oferta, define-se para a aplicação as classes Artigo e Oferta, evitando de se trabalhar diretamente com os registros retornados por uma instrução SQL para depois ajusta-la a lógica de negócio do modelo da aplicação, e utiliza-se apenas os objetos das classes Artigo e Oferta para realizar as operações dentro do sistema. Essa abstração do conceito de tabelas de banco de dados segundo Bauer e King (2005, p.56) torna o desenvolvimento da camada de persistência de aplicações orientadas a objetos mais simples e intuitiva.

2.2 MAPEAMENTO OBJETO-RELACIONAL

É a tecnologia que provê a persistência de forma automatizada e transparente dos objetos em uma aplicação em tabelas de uma base de dados relacional, usando metadados

para fazer a troca de dados entre os objetos e a base de dados, (BAUER; KING, 2005, p.46) fala que o mapeamento objeto-relacional essencialmente trabalha com dados transformados de uma representação para a outra, isto implica em determinadas perdas no desempenho. Entretanto, se o mapeamento objeto-relacional for executado como um *middleware*, há muitos ganhos e otimizações que não existiriam para uma camada codificada sem esse tipo de persistência.

Como mostra STONEBRAKER (1996) mapeamento objeto-relacional foi criado com a proposta de suprir as carências do modelo relacional, oferecendo toda a naturalidade para modelar objetos complexos e suas características peculiares que os bancos de dados orientados a objetos propunham, e ao mesmo tempo ter toda carga tecnológica desenvolvida para os bancos relacionais em anos de desenvolvimento. A proposta é oferecer uma interface orientada a objetos e de forma transparente para esse, modelar tudo usando o paradigma que melhor se adequar.

.

2.3 XML

Segundo HOLZNER (2001), o XML é a abreviação de *e**X**tensible **M**arkup **L**anguage* (Linguagem extensível de formatação). Trata-se de uma linguagem que é considerada uma grande evolução na internet, é uma especificação técnica desenvolvida pela (*World Wide Web Consortium* W3C - entidade responsável pela definição da área gráfica da internet), para superar as limitações do HTML, que é o padrão das páginas da *Web*. A linguagem XML é definida como o formato universal para dados estruturados na *Web*. Esses dados consistem em tabelas, desenhos, parâmetros de configuração, exportação de dados.

Sua utilização é de vital importância para o *framework open-source hibernate.* Através de arquivos de configuração no padrão *XML* que é feito todo o trabalho de mapeamento de tabelas, colunas, restrições, índices, relacionamentos das bases de dados para os objetos *Java*, e é por meio desses arquivos de configuração que a persistência desses objetos é feita de maneira correta nas bases de dados relacionais.

2.4 FRAMEWORK

Segundo DURHAM e JOHNSON (1996, pg.34) no desenvolvimento de *software*, um *framework* é uma estrutura de suporte definida em que um outro projeto de *software* pode ser organizado e desenvolvido. Tipicamente, um *framework* pode incluir programas de apoio, bibliotecas de código, linguagens de script e outros *softwares* para ajudar a desenvolver e juntar diferentes componentes do seu projeto, ou seja *framework* é um conjunto de classes que tem como objetivo a reutilização de funcionalidades já desenvolvidas, provendo um guia para uma solução de arquitetura em um domínio específico de *software*. *Framework* se diferencia de uma simples biblioteca, pois esta se concentra apenas em oferecer implementação de funcionalidades, sem definir a reutilização de uma solução de arquitetura.

3 TRABALHOS CORRELATOS

SOUZA (2004) apresenta como trabalho de conclusão de curso, um estudo comparativo entre os *frameworks Java* para desenvolvimento *Web.* De forma geral mostra as vantagens, desvantagens e a comparação de ganho real no uso de *frameworks* no processo de desenvolvimento de sistemas.

Faz uma avaliação individual dos dois *frameworks Java* para desenvolvimento *Web* mais populares entre os desenvolvedores, o *Spring* e *Jakarta Struts*, mostrando os seus modelos de desenvolvimento, recursos e detalhes técnicos da arquitetura de cada um dos *frameworks.* Mostrando que a utilização de *frameworks* no desenvolvimentos de softwares, é um fator que tende cada vez mais facilitar e diminuir o tempo de desenvolvimento.

(VIANA; BORBA, 2005) mostram as dificuldade existentes na integração de base de dados relacionais e aplicações orientadas a objetos. Propõem a utilização de vários *frameworks* como meio de amenizar o choque tecnológico entre esses dois paradigmas.

4 DESENVOLVIMENTO DO TRABALHO

Este capítulo apresenta os requisitos, a especificação, a implementação e a operacionalidade do trabalho.

4.1 ESPECIFICAÇÃO

Para exemplificar a utilização do *framework open-source hibernate* no desenvolvimento de um sistema de informação, será construído um sistema simples de controle de gastos pessoais, para modelagem do sistema foram utilizados os diagramas da *UML* que foram feitos na ferramenta *open-source Poseidon for UML*. Também foi utilizada a ferramenta *DbDesigner* para criar o diagrama de entidade e relacionamento e geração dos *scripts SQL* para criação do banco de dados.

No Quadro 1 são apresentados os requisitos funcionais e sua rastreabilidade, ou seja, sua vinculação com os respectivos casos de uso.

Requisitos Funcionais	Caso de Uso
RF01: O sistema deverá permitir o cadastro e a manutenção de usuários.	UC01
RF02: O sistema deverá permitir o cadastro e a manutenção de Bancos.	UC02
RF03: O sistema deverá permitir o cadastro e a manutenção de contas bancárias.	UC03
RF04: O sistema deverá permitir o cadastro e a manutenção de receitas e despesas.	UC04
RF05: O sistema deverá permitir o cadastro e a manutenção de agências bancárias.	UC05
RF06: O sistema deverá permitir a consulta das informações (usuários, bancos, contas bancárias, receitas e despesas).	UC06

Quadro 1: Requisitos funcionais

O Quadro 2 apresenta os requisitos não funcionais previstos para o sistema.

Requisitos Não Funcionais
RNF01: O sistema deverá ser multiplataforma.
RNF02: O desenvolvimento do sistema deverá ser realizado com ferramentas *open-source*.
RNF03: O sistema deverá ser desenvolvido usando o *framework* de mapeamento Objeto-Relacioal *Hibenate* para fazer a persistência dos objetos.

Quadro 2 : Requisito não funcionais

4.1.1.1 Diagrama de casos de uso

Na Figura 2 é apresentado o digrama de casos de uso do sistema exemplo.

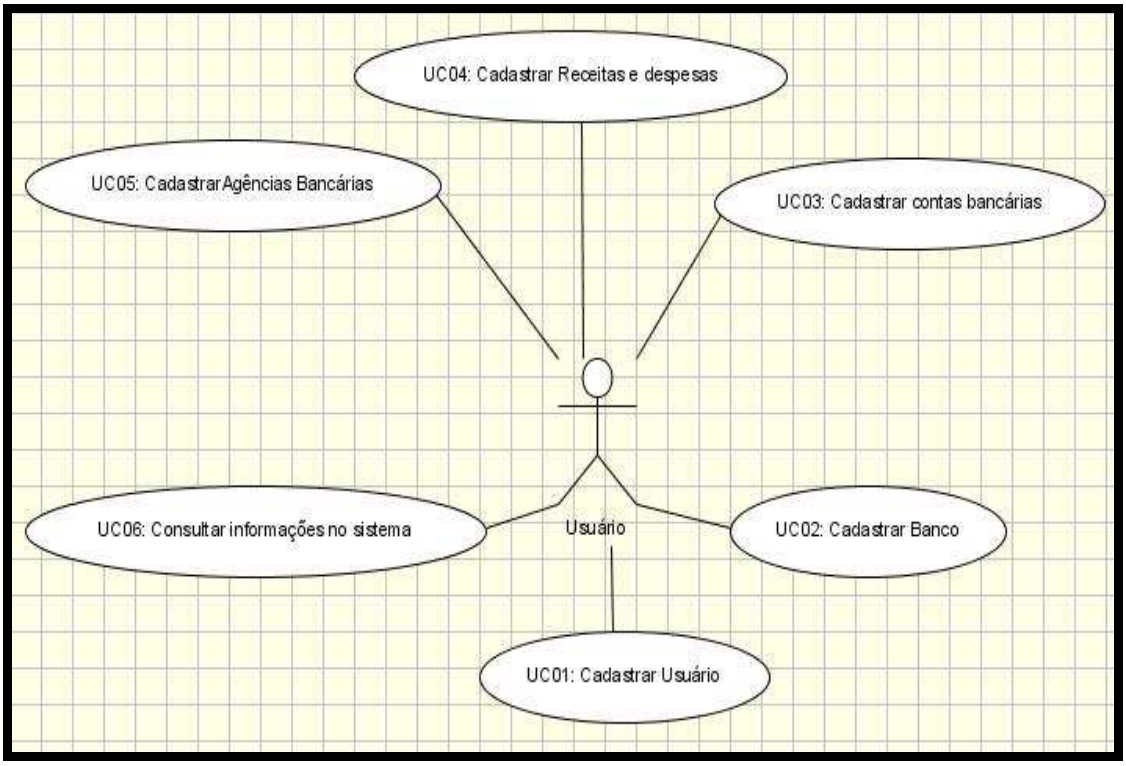

Figura 2: Diagrama de casos de uso do sistema exemplo

4.1.1.2 Diagrama de Classes

A Figura 3 apresenta o diagrama de classes do sistema exemplo.

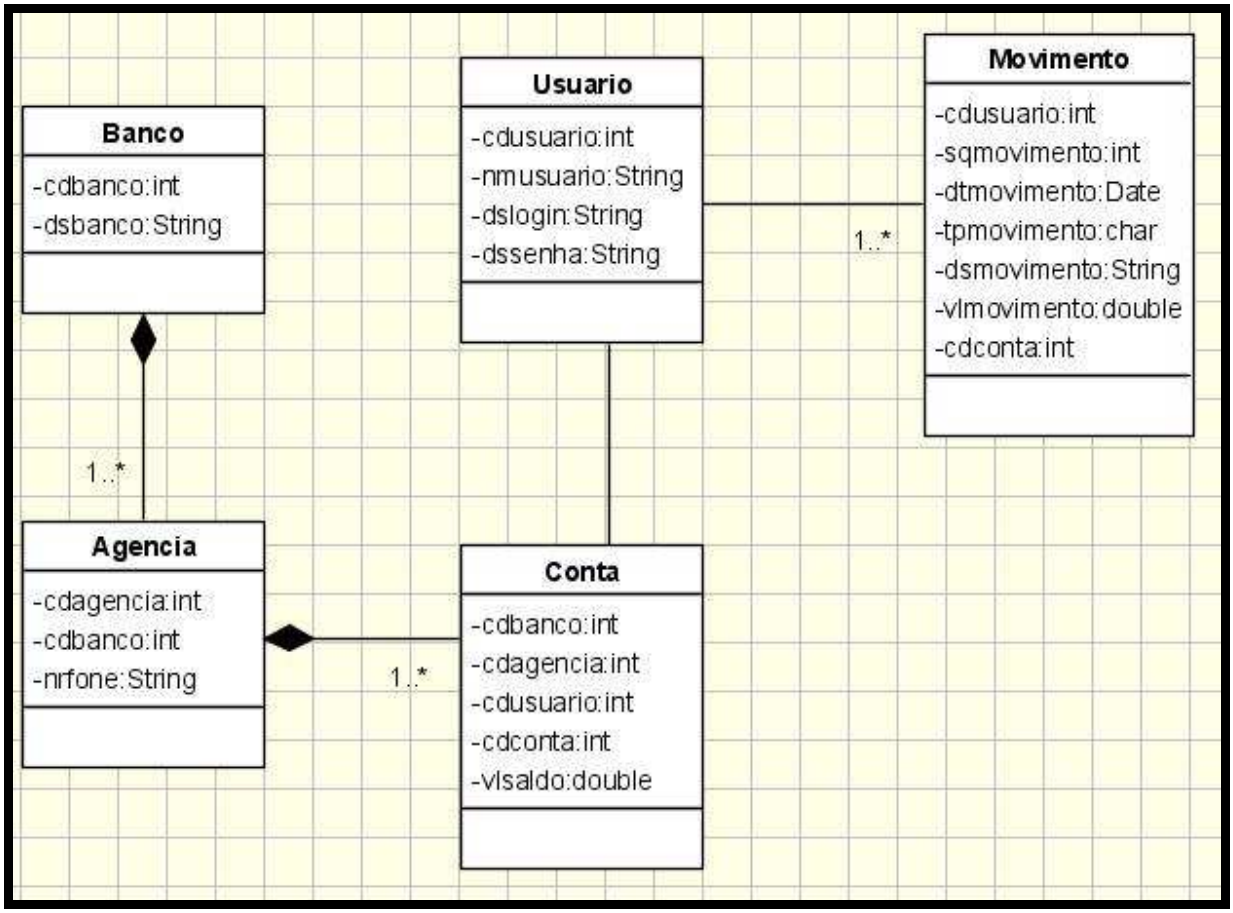

Figura 3: Diagrama de classes do sistema exemplo

4.1.1.3 Modelo de entidade e relacionamento

A Figura 4 demonstra o modelo de entidade e relacionamento do sistema exemplo utilizado para persistir os dados da aplicação.

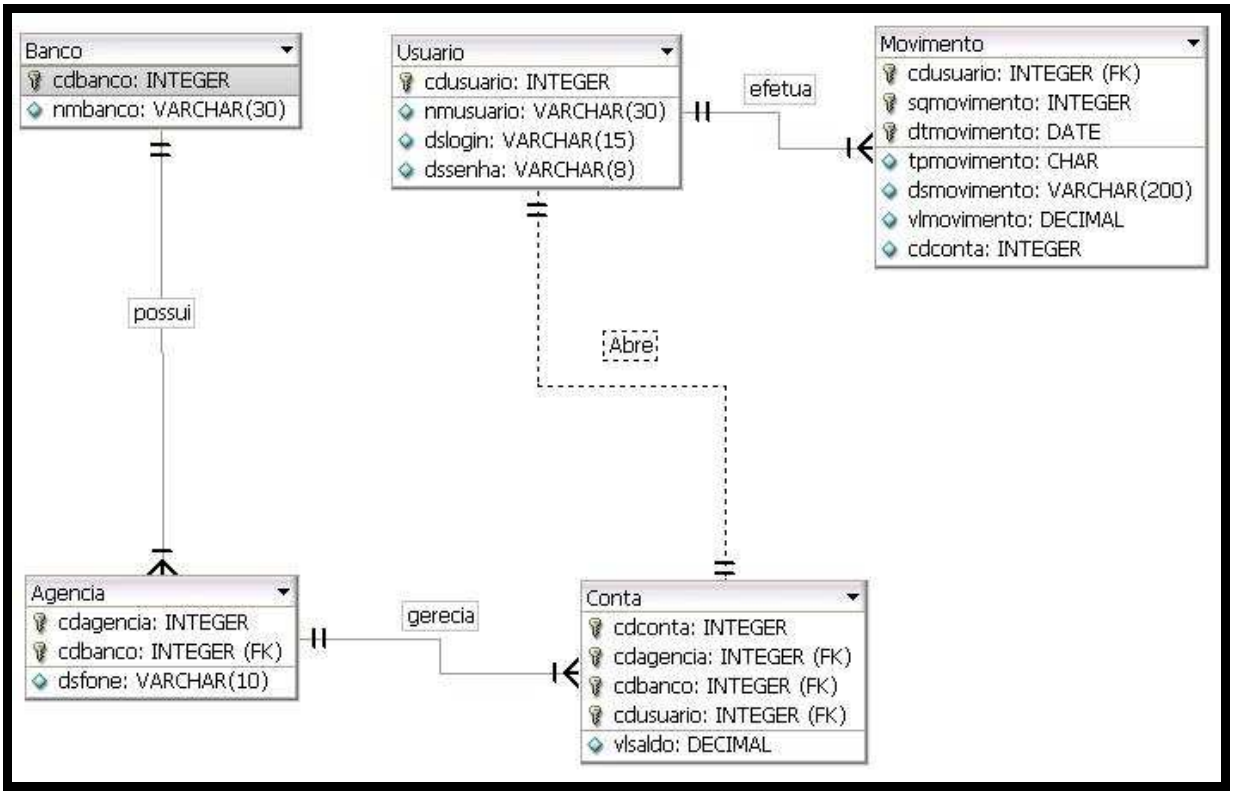

Figura 4: Modelo de entidade e relacionamento do sistema exemplo

Ao se comparar as Figuras 3 e 4, pode-se perceber que as entidades correspondem as classes do sistema. Essa é uma característica que o *hibernate* traz para facilitar na etapa de análise de um sistema de informação, unificando os modelos de dados, em outras palavras ele encapsula o conceito de tabelas fazendo com que os responsáveis por desenvolver o sistema se preocupem apenas em trabalhar com as classes, obtendo assim um melhor entendimento do sistema por completo.

4.2 IMPLEMENTAÇÃO

Esta seção apresenta a implementação apresentando a configuração do *hibernate,* o uso da IDE NetBeans bem como a persistência no SGBD Firebird.

4.2.1 Configurando o *hibernate*

Antes de iniciar o desenvolvimento do sistema exemplo deve-se primeiro criar a configuração do *framework* para que se possa utilizá-lo junto à ferramenta de desenvolvimento na qual o sistema será feito.

É recomendável utilizar a ultima versão estável do *hibernate* disponível no computador onde será desenvolvido o sistema. Nesta implementação foi utilizado o *hibernate* 3. Faz-se necessário configurar o *classpath* do computador, copiando os seguintes arquivos que estão no diretório onde foi colocado o *hibernate*:

hibernate3.jar
ehcache-1.1.jar
jta.jar
xml-apis.jar
commons-logging-1.0.4.jar
c3p0-0.8.5.2.jar
asm-attrs.jar
log4j-1.2.9.jar
dom4j-1.6.jar
antlr-2.7.5H3.jar
cglib-2.1.jar
asm.jar
jdbc2_0-stdext.jar
xerces-2.6.2.jar
commons-collections-2.1.1.jar

Quadro 3: Pacotes do *hibernate*

O *SGBD* utilizado foi o *Firebird SQL* por esse motivo também é necessário colocar o *driver JDBC* para *firebird SQL* no *classpath* do computador, pois o *hibernate* fará uso do *driver JDBC* para se conectar ao banco de dados. Após isso é necessário que o banco de dados seja criado a partir do modelo da Figura 4 para que se possa configurar o *hibernate* para o acesso aos dados e assim poder efetuar o mapeamento para as respectivas classes.

A *engine* do *hibernate* pode ser configurada de três modos diferentes, instanciando um objeto de configuração (*org.hibernate.cfg.Configuration*) e inserindo as suas propriedades programaticamente, usando um arquivo *.properties* com as suas configurações e indicando os arquivos de mapeamento programaticamente ou usando um arquivo *XML* (*hibernate.cfg.xml*) com as propriedades de inicialização e os caminhos dos arquivos de mapeamento. Para este trabalho foi utilizado o arquivo (*hibernate.cfg.xml*) para fazer a configuração do *hibernate*.

O Quadro 4 apresenta o arquivo de configuração.

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.dialect.FirebirdDialect</property>
        <property name="hibernate.connection.driver_class">org.firebirdsql.jdbc.FBDriver</property>
        <property name="hibernate.connection.url">jdbc:firebirdsql://localhost/c:\tcc\tcc.fdb</property>
        <property name="hibernate.connection.username">SYSDBA</property>
        <property name="hibernate.connection.password">masterkey</property>
        <!-- Configurações de debug -->
        <property name="show_sql">true</property>
        <property name="hibernate.generate_statistics">true</property>
        <property name="hibernate.use_sql_comments">true</property>
        <mapping resource="Usuario.hbm.xml"/>
        <mapping resource="Banco.hbm.xml"/>
        <mapping resource="Agencia.hbm.xml"/>
        <mapping resource="Conta.hbm.xml"/>
        <mapping resource="Cartao.hbm.xml"/>
        <mapping resource="Movimento.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Quadro 4: hibernate.cfg.xml

Na documentação do *hibernate* pode-se verificar todas as opções de propriedades que podem ser utilizadas e seus respectivos resultados, por isso o foco será dado ao que é importante para que se possa utilizar o *framework*. A seguir as propriedades do arquivo *hibernate.cfg.xml* (Quadro 5).

Propriedades de configuração	
hibernate.dialect	é a implementação do dialeto *SQL* específico do banco de dados a ser utilizado, o conceito de dialetos utilizado pelo

	hibernate é o que proporciona ao desenvolvedor construir a suas aplicações sem se preocupar com as particularidades dos diversos *SGBDs* do mercado, isso significa que para que suas aplicações passem a utilizar outro banco de dados basta apenas modificar esta propriedade para usar o dialeto do *SGBD* desejado.
hibernate.connection.driver_class	é o nome da classe do *driver JDBC* do banco de dados que está sendo utilizado, o *driver* utilizado neste exemplo é para o *firebird SQL*. Caso necessite de utilizar um *SGBD* diferente apenas altere essa propriedade e coloque o *driver* correspondente ao *SGBD* desejado.
hibernate.connection.url	é a URL de conexão específica do banco que está sendo utilizado, esta é outra propriedade que é específica para o *SGBD* que esta sendo utilizado, pois cada um disponibiliza o formato de URL a ser utilizado para conexão com o banco de dados físico ou a um serviço em servidor de banco de dados.
hibernate.connection.username	é o nome de usuário com o qual o *hibernate* deve se conectar ao banco de dados
hibernate.connection.password	é a senha do usuário com o qual o *hibernate* deve se conectar ao banco de dados.
show_sql	faz com que todo o código *SQL* gerado seja escrito na saída default.
hibernate.generate_statistics	faz com que o *hibernate* gere estatísticas de uso e possa diagnosticar uma má performance do sistema.
hibernate.use_sql_comments	adiciona comentários ao código SQL gerado, facilitando o entendimento das *queries*.

Quadro 5: Propriedades do *hibernate*

A última parte do arquivo é onde se indica os arquivos de mapeamento que o *hibernate*

deve processar para que seja feito o mapeamento das classes Java, se for esquecido de indicar um arquivo de mapeamento de qualquer classe, essa classe não vai poder ser persistida pela *engine* do *hibernate*.

4.2.2 Mapeamento das classes

Inicialmente deve ser feita a construção dos arquivos de mapeamentos ou seja os arquivos *XML* que definem as propriedades e os relacionamentos de uma classe para o *hibernate*, este arquivo pode conter classes, classes componentes e *queries* em *HQL* ou em *SQL*. O primeiro mapeamento abordado é o da classe Usuario e do seu relacionamento com a classe Conta. No modelo, uma Conta tem apenas um Usuario e um Usuario tem apenas uma Conta.

O Quadro 6 apresenta o mapeamento para a classe Usuario (o arquivo *Usuario.hbm.xml*):

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
        <class name="Usuario" table="USUARIO">
                <meta attribute="sync-DAO">false</meta>
                <id name="Id" type="integer" column="CDUSUARIO">
                    <generator class="sequence"/>
                </id>
                <property name="Ncusuario" column="NCUSUARIO" type="string" not-null="true" length="50"/>
                <property name="Dslogin" column="DSLOGIN" type="string" not-null="true" length="15"/>
                <property name="Dssenha" column="DSSENHA" type="string" not-null="true" length="8" />
        </class>
</hibernate-mapping>
```

Quadro 6: Arquivo *Usuario.hbm.xml*

No arquivo *Usuario.hbm.xml*, tem apenas uma classe sendo mapeada no arquivo, a

classe Usuario. O arquivo *XML* começa normalmente com as definições da *DTD* e do nó raiz, o <*hibernate-mapping*>, depois vem o nó que nos interessa neste caso, <*class*> que é onde define-se a classe que está sendo mapeada e para qual tabela ela será mapeada.

O único atributo obrigatório deste nó é *"name"*, que deve conter o nome completo da classe. Se o nome da classe for diferente do nome da tabela, deve-se colocar o nome da tabela no atributo *"table"*, como no exemplo, a tabela está com o nome todo em maiúsculas, o que obriga informar o mesmo.

Em seguida tem-se o nó <*id*> que é o identificador dessa classe no banco de dados. Neste nó é onde se define a propriedade que guarda o identificador do objeto no atributo *"name"*, que é *"cdusuario"*, como o nome da coluna no banco de dados também esta em maiúsculas e a propriedade do objeto em minúsculas foi informado no atributo *"column"* o nome da coluna do banco de dados. Ainda dentro deste nó, é encontrado mais um nó, o <*generator*>, que guarda a informação de como os identificadores (as chaves do banco de dados) são gerados, existem diversas classes de geradores, que são definidas no atributo *"class"* do nó, no nosso caso o gerador usado é o "sequence", que incrementa um ao valor da chave sempre que insere um novo objeto no banco.

Os próximos nós do arquivo são os <*property*> que indicam propriedades simples dos nossos objetos, como *Strings*, os tipos primitivos, objetos *Date*, *Calendar*, *Locale*, *Currency* e outros. Neste nó, os atributos mais importantes são *"name"*, que define o nome da propriedade, *"column"* para quando a propriedade não tiver o mesmo nome da coluna na tabela, *"type"* para definir o tipo do objeto que a propriedade guarda *"not-null"* para definir se a propriedade pode ser gravada com valores nulos ou não.

O arquivo de mapeamento para as tabelas que tem suas chaves primárias compostas por chaves de outras tabelas, é apresentado no Quadro 7.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
        <class name="Conta" table="CONTA">
                <meta attribute="sync-DAO">false</meta>
                <composite-id name="id" class="CONTAPK">
                        <key-many-to-one name="cdusuario" class="Usuario" column="CDUSUARIO"/>
                        <key-property name="cdconta" column="CDCONTA" type="string" />
                </composite-id>
                <property name="tpconta" column="TPCONTA" type="string" not-null="true" length="2"/>
                <property name="vlsaldo" column="VLSALDO" type="big_decimal" not-null="true" length="7" />
                <many-to-one name="cdbanco" column="CDBANCO" class="Agencia" not-null="true" > </many-to-one>
                <many-to-one name="cdagencia" column="CDAGENCIA" class="Agencia" not-null="true"> </many-to-one>
        </class>
</hibernate-mapping>
```

Quadro 7 : Arquivo *Conta.hbm.xml*

A única diferença entre o arquivo *Conta.hbm.xml* e o *Usuario.hbm.xml* é o nó <*composite-id*> o qual é formado pelo nó <*key-many-to-one*>, que nada mais é que a chave estrangeira da tabela Usuario, e pelo nó <*key-property*> que traz a propriedade "*cdconta"* e juntos formam a chave primária da tabela Conta.

Os mapeamentos das classes Banco, Agencia e Movimento são idênticos aos outros mapeamentos que já foram mostrados e não trazem nenhuma novidade para o estudo.

Os Quadros 8, 9 e 10 apresentam os arquivos *Banco.hbm.xml*, *Agencia.hbm.xml e Movimento.hbm.xml*.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
        <class  name="Banco" table="BANCO" >
                <meta attribute="sync-DAO">false</meta>
                <id name="id" type="integer" column="CDBANCO">
                        <generator class="sequence"/>
                </id>
                <property name="nmbanco" column="NMBANCO" type="string" not-null="true" length="30" />
        </class>
</hibernate-mapping>
```

Quadro 8: Arquivo Banco.hbm.xml.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
        <class name="Agencia" table="AGENCIA">
                <meta attribute="sync-DAO">false</meta>
                <composite-id>
                        <key-many-to-one name="cdbanco" class="Banco" column="CDBANCO" />
                        <key-property name="cdagencia" column="CDAGENCIA" type="string" />
                </composite-id>
                <property name="dsfone" column="DSFONE" type="string" not-null="false" length="10" />
        </class>
</hibernate-mapping>
```

Quadro 9: Arquivo Agencia.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
        <class  name="Movimento" table="MOVIMENTO" >
                <meta attribute="sync-DAO">false</meta>
                <composite-id>
                        <key-property name="sqmovimento" column="SQMOVIMENTO" type="integer" />
                        <key-property name="dtmovimento" column="DTMOVIMENTO" type="date" />
                        <key-many-to-one name="cdusuario" class="Usuario" column="CDUSUARIO" />
                </composite-id>
                <property name="tpmovimento" column="TPMOVIMENTO" type="string" not-null="true" length="1"/>
                <property name="dsmovimento" column="DSMOVIMENTO" type="string" not-null="true" length="200"/>
                <property name="vlmovimento" column="VLMOVIMENTO" type="big_decimal" not-null="true" length="7" />
                <property name="cdconta" column="CDCONTA" type="integer" not-null="true" length="10"/>
        </class>
</hibernate-mapping>
```

Quadro 10: Arquivo *Movimento.hbm.xml*

4.2.3 Hibernate na prática

Agora que o *hibernate* já está configurado e pronto para funcionar, é importante entender o seu mecanismo de persistência. Para o *hibernate*, existem três tipos de objetos, objetos *"transient"*, *"detached"* e *"persistent"* .

Objetos *"transient"* são aqueles que ainda não tem uma representação no banco de dados, eles ainda não estão sobre o controle do *framework* e podem não ser mais referenciáveis a qualquer momento, como qualquer objeto normal em Java.

Objetos *"detached"* têm uma representação no banco de dados, mas não fazem mais parte de uma sessão do *hibernate*, o que significa que o seu estado pode não estar mais sincronizado com o banco de dados. Objetos *"persistent"* são os objetos que tem uma representação no banco de dados e que ainda fazem parte de uma transação do *hibernate*, garantindo assim que o seu estado esteja sincronizado com o banco de dados.

No *hibernate*, assim como no *JDBC*, existem os conceitos de sessão e transação. Uma

sessão é uma conexão aberta com o banco de dados, onde pode-se executar *queries*, inserir, atualizar e deletar objetos, já a transação é a demarcação das ações, uma transação faz o controle do que acontece e pode fazer um *roolback*, assim como uma transação do *JDBC*, se forem encontrados problemas.

No Quadro 11 a classe *SessaoHibernate* que é a responsável por configurar e abrir as sessões do *hibernate.*

```
import javax.swing.JOptionPane;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class SessaoHibernate {
    private static SessionFactory factory;
    static {
        try {
            factory = new Configuration().configure().buildSessionFactory();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null,"Desculpe, Ocorreu um erro ao " +
                                                "criar a sessão!");
            factory = null;
        }
    }
    public static Session getSession() {
        return factory.openSession();
    }
}
```

Quadro 11: Classe SessaoHibernate.java

O bloco estático, ou seja, as chaves marcadas como *"static"* instancia um objeto de configuração do *hibernate org.hibernate.cfg.Configuration*, chama o método configure() que lê o arquivo *hibernate.cfg.xm*) e depois que ele está configurado, criamos uma *SessionFactory,* que é a classe que vai ficar responsável por abrir as sessões de trabalho do *hibernate*.

O código apresentado no Quadro 12 é um exemplo de persistência de um objeto da classe Usuário utilizando o *hibernate.*

```
import org.hibernate.Session;
import org.hibernate.Transaction;

public class Persistencia {
    public static void main(String[] args) {
        Session sessao = SessaoHibernate.getSession(); //Abrindo uma sessão
        Transaction transaction = sessao.beginTransaction(); //Iniciando uma transação
        Usuario usuario = new Usuario(); //Instanciando um objeto transiente
        usuario.setNcusuario("Odilon Herculano" +
                             " Soares Filho"); //Preenchendo as propriedades do objeto
        usuario.setDslogin("odilonjr");
        usuario.setDssenha("%34op12Q");
        sessao.save(usuario); //Transformando o objeto transiente em um objeto
                              //persistente no banco de dados
        transaction.commit(); //Finalizando a transação
        sessao.close(); //Fechando a sessão
    }
}
```

Quadro 12: Exemplo de persistência usando *hibernate*

Entendendo o que foi feito neste código, primeiro foi inicializado um *SessionFactory*, dentro do bloco estático na classe *SessaoHibernate*, depois que a *SessionFactory* é inicializada, o método *getSession()* retorna uma nova sessão para o código dentro do *main()*. Após a sessão ter sido retornada, é iniciamos uma transação, instanciamos um objeto □suário, são preenchidas as propriedades do objeto e é chamado o método *save()* na sessão. Após o método *save()*, é finalizada a transação e fechada a sessão. Assim foi inserido um registro no banco de dados sem ser escrita nenhuma linha de *SQL*, apenas com a chamada de um método.

O código de aplicações que usam o *hibernate* costumam mostrar esse mesmo comportamento, abrir sessão, inciciar transação, chamar os métodos *save(), update(), get(), delete(),* etc, fechar a transação e depois a sessão, um comportamento muito parecido com o de aplicações *JDBC* comuns, a diferença é que não é preciso escrever nem uma linha sequer de *SQL* para isso.

No quadro 13 são apresentadas as classes mais utilizadas no desenvolvimento utilizando o *hibenate*.

Classes do *hibernate*	
Session	*org.hibernate.Session*
SessionFactory	*org.hibernate.SessionFactory*
Configuration	*org.hibernate.cfg.Configuration*
Transaction	*org.hibernate.Transaction*
Query	*org.hibernate.Query*
Criteria	*org.hibernate.Criteria*
Criterion	*org.hibernate.criterion.Criterion*
Restrictions	*org.hibernate.criterion.Restrictions*

Quadro 13: Classes do *hibernate*

4.2.4 Pesquisas no banco de dados com o hibernate

O *hibernate* já esta configurado, as tabelas estão mapeadas, mas uma das partes mais importantes do funcionamento do *framework*, são as pesquisas. Existem três meios de se fazer buscas usando o *hibernate*, usando a sua linguagem própria de buscas, a *Hibernate Query Language* ou simplesmente *HQL*, usando a sua *Criteria Query API* para montar buscas programaticamente ou usando *SQL* puro. A maioria das necessidades dos desenvolvedores quase sempre é suprida com as duas primeiras alternativas, mas para situações onde as primeiras alternativas não forem suficientes ainda pode-se usar SQL pra resolver.

A *HQL* é uma extensão da *SQL* com alguns adendos de orientação a objetos, nela não se referencia tabelas, referenciam-se os objetos do modelo que foram mapeados para as tabelas do banco de dados. Além disso, por fazer pesquisas em objetos, não é preciso selecionar as colunas do banco de dados, um "*select * from usuario"* em *HQL* seria simplesmente "*from Usuario"*, porque não é utilizando o conceito de tabelas e sim de objetos.

A *Criteria Query API* é um conjunto de classes para a montagem de *queries* em código Java, onde se define todas as propriedades da pesquisa chamando os métodos e avaliações das

classes relacionadas. Como tudo é definido programaticamente, ganha-se todas as funcionalidades inerentes da programação orientada a objetos para montar as pesquisas e ainda garante a completa independência dos bancos de dados, pois a classe de dialeto *SQL* do seu banco vai se encarregar de traduzir tudo o que for utilizado.

O código apresentado no Quadro 14 é um exemplo de como implementar a classe *Query* para realizar consultas.

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ConsultaQuery {
    Session sessao = SessaoHibernate.getSession();
    Transaction tc = sessao.beginTransaction();
    public List pesquisaUsuario(){
        Query select = sessao.createQuery("from Usuario");
        List usuarios = select.list();
        System.out.println(usuarios);
        tc.commit();
        sessao.close();
    }
}
```

Quadro 14: Classe *ConsultaQuery*

O código ainda segue aquela mesma seqüência do Quadro 12, abrir uma sessão, iniciar uma transação e começar a acessar o banco de dados. Para criar uma *query*, é chamado o método *createQuery(String query)* na sessão, passando como parâmetro o String que representa a *query*. Depois disso, é chamado o método *list()*, que retorna um *List* com os objetos resultantes da *query*.

No Quadro 15 é apresento o mesmo exemplo mas agora implementando a classe *Criteria* para realizar consultas.

```
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ConsultaCriteria {
    Session sessao = SessaoHibernate.getSession();
    Transaction tc = sessao.beginTransaction();
    public void pesquisaUsuario(){
        Criteria select = sessao.createCriteria(Usuario.class);
        List usuarios = select.list();
        System.out.println(usuarios);
        tc.commit();
        sessao.close();
    }
}
```

Quadro15: Classe *ConsultaCriteria*

Como pode-se ver apenas a linha onde é criada a *query* mudou. Agora, em vez de chamar o método *createQuery(String query)*, é chamado o método *createCriteria(Class clazz)*, passando como parâmetro a classe que vai ser pesquisada, que no exemplo é Usuario.

Outro complemento importante do *hibernate* é o suporte a paginação de resultados. Para paginar os resultados, são chamados os métodos *setFirstResult(int first)* e setMaxResults(int max). O primeiro método indica de qual posição os objetos devem começar a ser carregados, o segundo indica o máximo de objetos que devem ser carregados. Estes métodos estão presentes tanto na classe *Query* quanto na classe *Criteria*.

O Quadro 16 apresenta um exemplo de como proceder para carregar apenas os dez primeiros usuários do banco de dados.

```
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ConsultaCriteria {
    Session sessao = SessaoHibernate.getSession();
    Transaction tc = sessao.beginTransaction();
    public void pesquisaUsuario(){
        Criteria select = sessao.createCriteria(Usuario.class);
        select.setFirstResult(0);
        select.setMaxResults(10);
        List usuarios = select.list();
        System.out.println(usuarios);
        tc.commit();
        sessao.close();
    }
}
```

Quadro16: Consulta paginada

Foram chamados os métodos *setFirstResult()* e *setMaxResults()* antes de listar os objetos da *Query*, a contagem de resultados assim como os *arrays* começa em zero, não em um. Uma propriedade específica da *HQL*, que foi herdada do *JDBC*, é o uso de parâmetros nas *queries*. Assim como no *JDBC*, os parâmetros podem ser numerados, mas também podem ser nomeados, o que simplifica ainda mais o uso e a manutenção das *queries*, porque a troca de posição não vai afetar o código que as usa.

No Quadro 17 é apresentado um exemplo do uso de parâmetros.

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ConsultaQuery {
    Session sessao = SessaoHibernate.getSession();
    Transaction tc = sessao.beginTransaction();
    public void pesquisaUsuario(){
        Query select = sessao.createQuery("from Usuario as usuario " +
                                          "where usuario.ncusuario = :nome");
        select.setString("nome", "Odilon Herculano Soaares Filho");
        List usuarios = select.list();
        System.out.println(usuarios);
        tc.commit();
        sessao.close();
    }
}
```

Quadro 17: Consulta com parâmetros

O exemplo traz mais algumas adições a *query HQL*. A primeira é o uso do “as”, que serve para apelidar uma classe na nossa expressão do mesmo modo do “as” em *SQL*. Outro aspecto a se destacar é o acesso as propriedades usando o operador “.” (ponto), pois sempre deve-se acessar as propriedades usando esse operador. A última parte é o parâmetro propriamente dito, que deve ser iniciado com “:” (dois pontos) para que o *hibernate* saiba que isso é um parâmetro que vai ser inserido na *query*. Insere-se um parâmetro nomeado, usando o método “*set*” correspondente ao tipo de parâmetro, passando primeiro o nome com o qual ele foi inserido e depois o valor que deve ser colocado.

O *hibernate* também facilita a externalização de *queries HQL* e até *SQL* com os nós <*query*> e <*sql-query*> nos arquivos de mapeamento. Pode-se adicionar uma *query* no mapeamento da classe Usuario no arquivo *Usuario.hbm.xml* como apresentado no Quadro 18.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>
        <class  name="Usuario" table="USUARIO" >
                <meta attribute="sync-DAO">false</meta>
                <id name="id" type="integer" column="CDUSUARIO">
                        <generator class="sequence"/>
                </id>
                <property name="ncusuario" column="NCUSUARIO" type="string" not-null="true" length="50" />
                <property name="dslogin" column="DSLOGIN" type="string" not-null="true" length="15"/>
                <property name="dssenha" column="DSSENHA" type="string" not-null="true" length="8" />
        </class>
        <query name="buscarUsuarioPeloNome">
            <![CDATA[from Usuario u where u.ncusuario = :nome]]>
        </query>
</hibernate-mapping>
```

Quadro 18: Pesquisa definida no arquivo de mapeamento.

As linhas adicionadas no arquivo de mapeamento vão instruir o *hibernate* a criar um *PreparedStatement* para esse *select*, fazendo com que ele execute mais rápido e possa ser chamado de qualquer lugar do sistema. Deve-se preencher o atributo *"name"* com o nome pelo qual esta *query* deve ser chamada na aplicação e no corpo do nó você deve colocar o código da *query*, de preferência dentro de uma tag CDATA, pra evitar que o *parser XML* entenda o que está escrito na *query* como informações para ele.

O Quadro 19 apresenta como executar uma *query* nomeada.

```
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

public class ConsultaQuery {
    Session sessao = SessaoHibernate.getSession();
    Transaction tc = sessao.beginTransaction();
    public void pesquisaUsuario(){
        Query select = sessao.getNamedQuery("buscarUsuarioPeloNome");
        select.setString("nome", "Odilon Herculano Soaares Filho");
        List usuarios = select.list();
        System.out.println(usuarios);
        tc.commit();
        sessao.close();
    }
}
```

Quadro19: Usando uma query nomeada

A única diferença para a consulta do Quadro 17 é que em vez de escrever a *query* dentro do código Java, ela ficou externalizada no arquivo de mapeamento. Para instanciar o objeto, chama-se o método *getNamedQuery(String name)*, passando como parâmetro o nome da *query* definido no atributo *"name"* no arquivo *XML* de mapeamento.

4.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Foi desenvolvido um sistema exemplo, para demonstração das funcionalidades do *framework hibernate* como criar registros nas tabelas, fazer a atualização de registros, consultar informações na base de dados, sempre utilizando a tecnologia de mapeamento objeto-relacional.

Na Figura 5 é apresentada a primeira tela do sistema, onde é feito o *login* do usuário. Nesta tela são informados o usuário e senha.

Figura 5: Tela de login do sistema.

Após a identificação do usuário, é apresentada a tela principal do sistema simples de controle de gastos pessoais. Através desta tela é possível acessar todas as funcionalidades do sistema, através dos menus de cadastros e consultas, conforme apresenta a Figura 6.

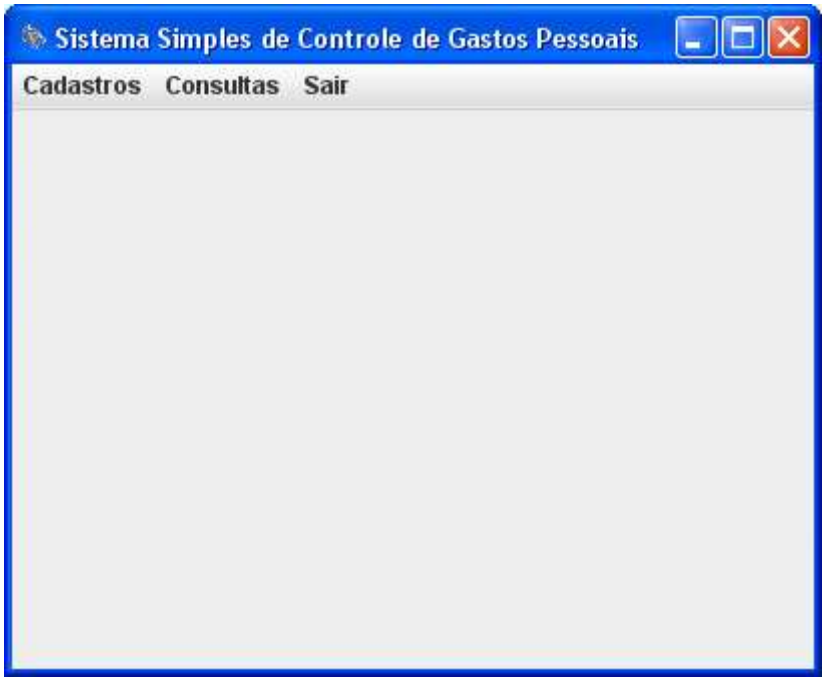

Figura 6: Tela principal do sistema de controle de gastos pessoais.

Acessando o menu de cadastros, o sistema disponibiliza opções de cadastro de usuários, bancos, agências, contas e cadastro de movimentos. As telas de cadastros do sistema foram desenvolvidas utilizando o conceito de ergonomia, fazendo com que todos os cadastros tenham basicamente as mesmas funcionalidades como os botões para cadastrar, excluir, alterar registros, o botão salvar que efetiva a gravação dos dados nas tabelas, o botão cancelar que aborta a operação que estiver sendo executada na tela, e o botão para sair da tela do cadastro. Utilizando um *layout* muito semelhante para todas as telas, facilitando assim a compreensão por parte dos usuários no aprendizado da utilização o sistema.

Na Figura 7, é apresentada a tela de cadastro de usuários, onde é possível cadastrar um novo usuário informando nome completo, usuário e senha, assim como as demais opções disponíveis já mencionadas.

Figura 7: Tela de cadastro de usuários.

Na Figura 8 pode-se observar a tela de cadastro de bancos, onde se cadastra o banco, informando o nome do mesmo.

Figura 9: Tela de cadastro de bancos.

Ao excluir um banco é respeitado o conceito de composição da orientação a objetos, todos as agências que compõem o banco e suas respectivas contas também serão excluídas do banco de dados.

No cadastro de agências, é feito o cadastro das agências disponíveis, de acordo com o

banco. O telefone da agência também pode ser informado neste cadastro.

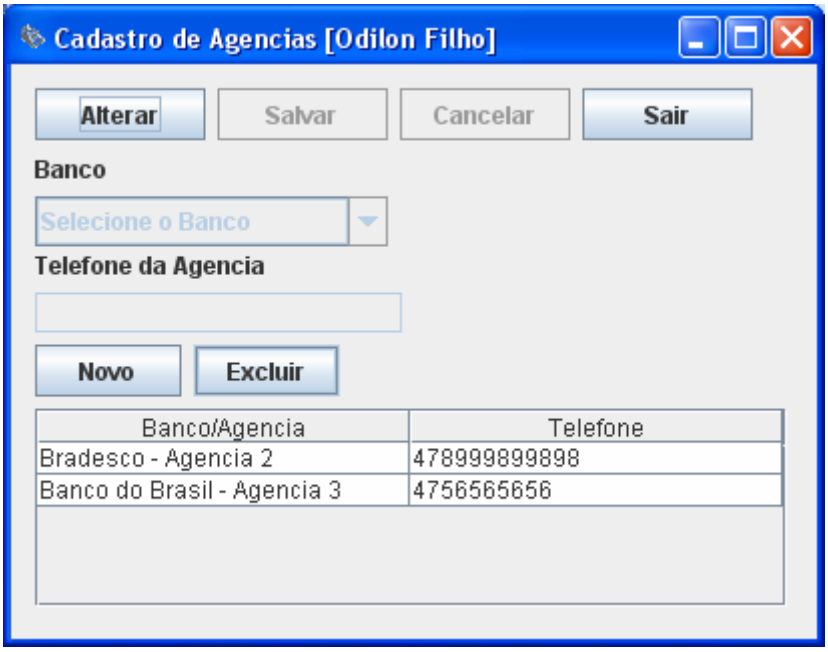

Figura 10: Tela de cadastro de agências.

Da mesma forma que no cadastro de bancos, ao excluir uma agência é respeitado o conceito de composição da orientação a objetos, onde todas as contas e seus respectivos movimentos que compõem a agência serão eliminadas do banco de dados.

A tela de Cadastro de Contas apresentada na Figura 11 permite que o usuário possa ter acesso à determinada conta e o saldo da mesma, de acordo com a agência.

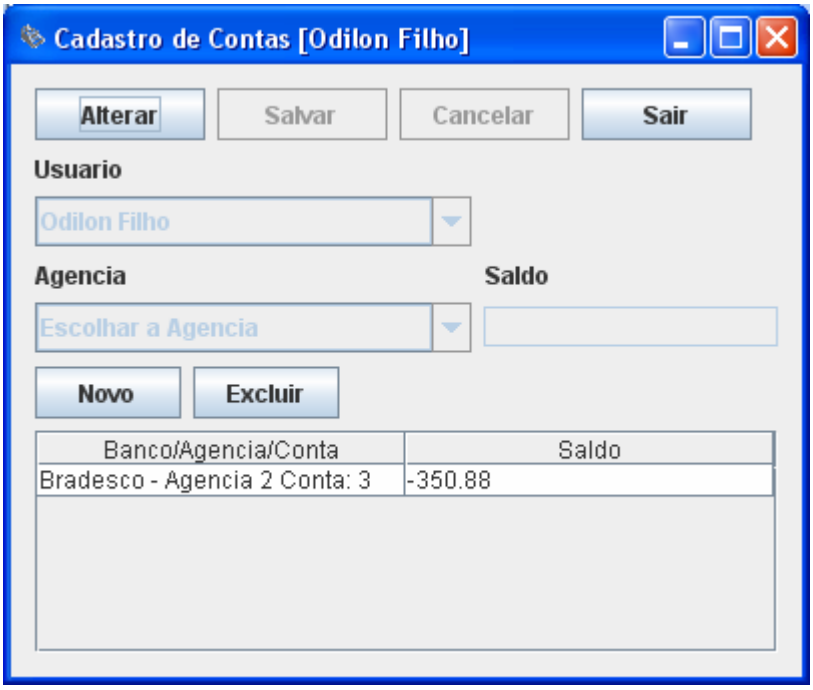

Figura 11: Tela de cadastro de contas.

Ao excluir uma conta todos os seus movimentos serão também excluídos do banco de dados.

A Figura 12 apresenta a tela de Cadastro de Movimentos, onde são indicados os movimentos de receita e despesa e o valor dos mesmos, de acordo com a conta do usuário. Estes movimentos atualizam o saldo indicado no cadastro de contas.

Figura 12: Tela de cadastro de movimentos.

Voltando a tela principal do Sistema Simples de Controle de Gastos Pessoais há também o menu de Consultas. Acessando este menu, está disponível a opção consulta movimentos. Clicando nesta opção do sistema, a Pesquisa de Movimentos será disponibilizada, conforme mostra a Figura 13.

A pesquisa de movimentos pode ser feita através de uma faixa de data, por tipo (receita ou despesa) ou pela descrição do movimento. A pesquisa irá apresentar resultados se houve algum movimento dentro dos parâmetros informados. Na pesquisa são apresentados os movimentos na ordem de: seqüência, data, conta, tipo e valor.

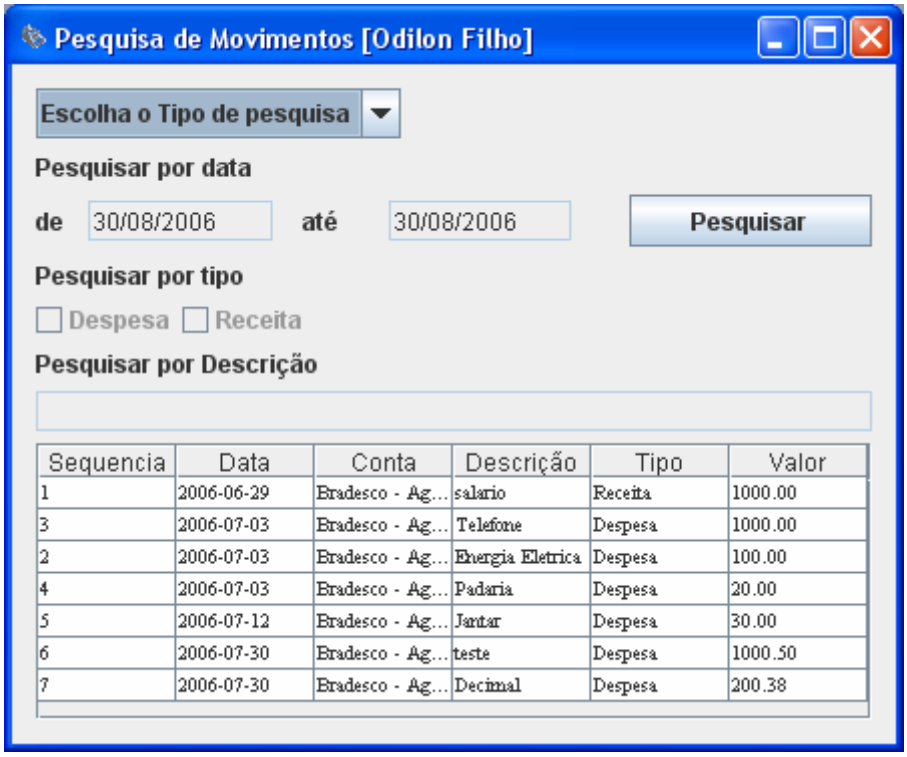

Figura 13: Tela de pesquisa de movimentos.

4.4 RESULTADOS E DISCUSSÃO

O *framework hibernate* demonstrou-se prático, pois facilitou o desenvolvimento de aplicações. Um aspecto a ser destacado é a ênfase do programador em se preocupar mais com o seu modelo de objeto e seus comportamentos, do que com as tabelas do banco de dados. O *hibernate* também evita o trabalho de escrever dúzias de código repetido para fazer as mesmas coisas, como "*inserts*", "*selects*", "*updates*" e "*deletes*" no banco de dados, além de ter duas opções para se montar buscas complexas em grafos de objetos e ainda uma saída para o caso de nada funcionar, usar *SQL*.

Além de mecanismo de mapeamento objeto-relacional, o *hibernate* também pode trabalhar com um sistema de *cache* das informações do banco de dados, aumentando ainda mais a performance das aplicações. O esquema de *cache* do *hibernate* é complexo e extensível, existindo diversas implementações possíveis, cada uma com suas próprias

características. Pode-se ser feitas associações das facilidades apresentadas pelo *framework* com alguns dos itens do modelo de qualidade da *NBR 13596*. A manutenibilidade certamente é uma das características mais marcantes do *hibernate*, pois fornece ao desenvolver de sistemas o conceito do mapeamento dos dados para um modelo de classes, deixando o código mais conciso e claro facilitando um melhor entendimento. Este aspecto faz influencia em futuras manutenções no sistema, ainda mais quando as manutenções devem ser feitas por equipes que não participaram do desenvolvimento.

Outra característica no *framework hibernate* é a portabilidade. Como o mesmo foi desenvolvido na linguagem *Java,* torna-se adaptativo, pois pode estar presente em diversas plataformas sem usando um mesmo conjunto de pacotes, isso quer dizer que em qualquer plataforma onde se possa rodar programas *Java,* o *hibernate* pode ser utilizado. Por utilizar o conceito de dialetos para os bancos de dados suportados, o *hibernate* liberta o desenvolvedor para apenas se preocupar com a lógica de negócio do seu sistema. Isto se dá pois o próprio *framework* se encarrega de resolver as particularidades sintáticas de banco de dados isso põe *hibernate* em uma situação de destaque a frente de outras tecnologias de mapeamento objeto-relacional encontradas no mercado.

O *hibernate* não é a solução perfeita para todos os problemas, nas aplicações que fazem uso intenso de *stored procedures*, *triggers, user defined functions* e outras extensões específicas dos bancos de dados, normalmente não vão se beneficiar do mecanismo oferecido pelo *framework*. Além disso, aplicações que tem uma alta freqüência de muito intensa de comandos que geram tráfico na rede, como seqüências de *"inserts*", *"updates"* e *"deletes"* terminariam por perder performance, graças a instanciação e carregamento de diversos objetos na memória.

Outra situação que não é aconselhada para o uso do *framework* é quando sefaz necessário o uso de chaves primárias compostas, pois o mapeamento desse tipo de chave é

complexo e bastante problemático nos arquivos *XML* de mapeamento do *hibernate* e é desaconselhado por Bauer e King (2005, p.96) pois resulta em uma notável perda de performance.

5 CONCLUSÕES

Os objetivos previstos para o trabalho foram alcançados. O *framework* de mapeamento obejto-relacional *hibernate* é um facilitador no desenvolvimento de softwares orientados a objetos. Apresenta o *middleware* fazendo com a camada de persistência seja desenvolvida com muita liberdade, tanto por dar mais legibilidade ao código, pois o desenvolvedor abstrai o conceito de tabelas do banco de dados, quanto pela portabilidade oferecida pelos dialetos de banco de dados que abstraem as particularidades de cada servidor de banco de dados.

Ao utilizar o *framework é possível* observar não só as vantagens oferecidas, mas também os pontos fracos do *hibernate.* A de se destacar a dificuldade de se fazer um mapeamento de tabelas que utilizem chave primária composta, ou seja, chave formada por dois ou mais atributos. Outro aspecto importante é utilização em massa de transações que geram tráfego na rede fazendo com que a performance seja muito abaixo do desejado.

Conclui-se que o *framework* desempenha bem o que lhe é incumbido que é fazer a persistências de objetos em bases de dados relacionais, mas para que o mesmo tenha seu desempenho satisfatório deve-se analisar bem em que situações o *hibernate* deve ser aplicado.

5.1 EXTENSÕES

Para um futuro trabalho sobre o *hibernate,* é interessante fazer um estudo aprofundado sobre seu sistema de gerenciamento de *cache*, que é bem complexo de se implementar, porém que é extensível, e se bem implementado pode trazer muitos ganhos de performance para as aplicações.

Outro trabalho que pode ser feito é uma comparação critica entre o *hibernate* e os seus concorrentes diretos, ferramentas como *JDO, Apache OJB Oracle TopLink.* Analisar quais

são as características que são comuns a todos, quais desempenham o papel de persistir objetos em base de dados relacionais com tanta flexibilidade, portabilidade quanto o *hibernate* ou até mais.

REFERÊNCIAS BIBLIOGRÁFICAS

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 13596** - tecnologia de informação - avaliação de produto de software: características de qualidade e diretrizes para o seu uso. Rio de Janeiro: ABNT, 1996.

BAUER, Christian; KING, Gavin. **Hibernate in action**. Greenwitch: Manning Publications, 2005.

DURHAM, A; JOHNSON R. **A Framework for Run-time Systems and its Visual Programming Language**. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS. San Jose, CA, October 1996 Proceedings**...** OOPSLA 1996 p. 20-25.

Hibernate - **Relational Persistence** For Idiomatic Java. [S.l.] 2006. Disponível em: <www.hibernate.org> Acesso em: 10/06/2006.

HOLZNER, Steven. **Desvendando XML.** Rio de Janeiro: Campus, 2001.

SAUVÉ, Jacques Philippe. **Sistemas de Informação** 1. [S.l.] 2006 Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/2005.1/si1/>. Acesso 11/06/2006

SOUZA, M. V. B. **Estudo Comparativo entre Frameworks Java para Construção de Aplicações Web**. 2004. Monografia (Conclusão de Curso de Bacharel em Ciências da Computação).Universidade Federal de Santa Maria. Santa Maria, 2004.

Sun Corporation. **Java Tecnology**. Disponível em: <http://java.sun.com/>Acesso em: 10/06/2006

STONEBRAKER, M. *Object-Relational Database Systems.* Morgam Kanfman,1996

VIANA, Euricelia; BORBA, Paulo. **Integrando Java com Bancos de Dados Relacionais.** 2003. Monografia (Conclusão de Curso de Bacharel de Ciências da Computação). Universidade Federal de Pernambuco.

.

.