Persistência em banco de dados relacionais usando linguagens orientada a objetos

Marcos Pinheiro Vilhanueva

Maringá – Paraná
2007

	Universidade Estadual de Maringá Centro de Tecnologia Departamento de Informática	

Persistência em banco de dados relacionais usando linguagens orientada a objetos

Marcos Pinheiro Vilhanueva

Trabalho apresentado ao Departamento de Informática da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de especialista. Orientador: *Prof. Marco Aurélio Lopes Barbosa.*

Maringá – Paraná
2007

Marcos Pinheiro Vilhanueva

Persistência em banco de dados relacionais usando linguagens orientada a objetos

Trabalho apresentado ao Departamento de Informática da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Especialista em Desenvolvimento de Sistemas para WEB.

Orientador: Prof. Marco Aurélio Lopes Barbosa
Departamento de Informática, DIN

Prof. Ademir Carniel
Departamento de Informática, DIN

Prof. Letícia Rodrigues Bueno
Departamento de Informática, DIN

Maringá – Paraná
2007

Sumário

1. Introdução ... 1
2. O modelo em camadas ... 3
2.1. A Evolução da programação em camadas ... 3
2.2. A Camada de persistência ... 5
3. O JDBC ... 10
3.1. Visão geral sobre JDBC ... 10
3.2. Funcionalidades ... 12
3.2.1. Conexão com o Banco de Dados ... 12
3.2.2. Instruções DDL ... 13
3.2.3. Consultas SQL ... 14
3.2.4. Outras características ... 15
4. ORM ... 16
5. Hibernate ... 18
5.1. As interfaces do Hibernate ... 18
5.2. Configuração mínima ... 20
5.3. Os tipos do Hibernate ... 22
5.4. Arquivos de Mapeando de classes ... 24
5.5. O HQL ... 26
5.6. Ferramentas do Hibernate ... 26
6. Estudo de caso: Comparação entre JDBC e Hibernate ... 28
6.1. A aplicação ... 28
6.2. A implementação ... 31
6.3. Resultados ... 36
7. Conclusão ... 37

Índice de Figuras

FIGURA 1 – MODELO MONOLÍTICO 3
FIGURA 2 – MODELO CLIENTE SERVIDOR 4
FIGURA 3 – MODELO EM CAMADAS 5
FIGURA 4 – ESTRUTURA DE UM SGBD 7
FIGURA 5 - JDBC 11
FIGURA 6 – VISÃO GERAL DA API DO HIBERNATE 19
FIGURA 7 – DIAGRAMA DE OBJETOS 29
FIGURA 8 - DIAGRAMA DE ENTIDADE E RELACIONAMENTO 30

Índice de Quadros

Quadro 1 - Exemplo de conexão JDBC com o banco de dados 12

Quadro 2 – Exemplo de um Create Table com JDBC..14

Quadro 3 – Exemplo de um Insert na tabela TB_PESSOAS com JDBC........................14

Quadro 4 – Exemplo de um Insert com parâmetros na tabela TB_PESSOAS com JDBC...14

Quadro 5 – Exemplo de consulta a toda uma tabela com JDBC....................................15

Quadro 6 - Exemplo do arquivo hibernate.cfg.xml para firebird..................................21

Quadro 7 – Mapeamento simples de uma tabela TB_Pessoas para classe Pessoa..........24

Quadro 8 – Mapeamento de herança...25

Quadro 9 – Mapeamento de um para muitos...25

Quadro 10– Exemplo de uma consulta com HQL..26

Quadro 11- Inclusão de registro com JDBC...32

Quadro 12 - Inclusão de registro com Hibernate..32

Quadro 13 - Alteração de registro com JDBC..33

Quadro 14 - Alteração de registro com Hibernate..33

Quadro 15 - Exclusão de registro com JDBC...33

Quadro 16 - Exclusão de registro com Hibernate...34

Quadro 17 - Pesquisa de registro com JDBC..34

Quadro 18 - Pesquisa de registro com Hibernate..35

Índice de Tabelas

Tabela 1 – Tipos Primitivos..23
Tabela 2 – Tipos de data e hora...23
Tabela 3 – Tipos de objetos binários e grandes...23

Dedico este trabalho

A Deus que me deu saúde e força e a minha mãe Neucy Pinheiro e a minha esposa Olga pelo amor, incentivo, compreensão e ajuda durante todos estes anos e a meus filhos Rafael e João Marcos.

Agradecimentos

Ao criador de tudo, que habita em luz inacessível ao qual nenhum olho viu, nosso criador. Sem o qual nunca poderia ter realizado este sonho que busquei por tantos anos.

Ao professor Marco Aurélio, meus sinceros agradecimentos pela disposição e compreensão, pela orientação firme e segura na elaboração deste trabalho;

Aos amigos Maycol Alencar e Carlos Roberto Ueda pela ajuda prestada na parte prática.

E a todos aqueles que me ajudaram e torceram por mim, meu afeto e minha eterna gratidão.

Resumo

Os sistemas de banco de dados relacionais e as linguagens orientada a objetos são bastantes utilizados hoje em dia. Estes dois paradigmas diferentes têem que conviver juntos, criando esta incompatibilidade semântica entre os paradigmas objeto/relacional. Neste trabalho é apresentado um estudo dos mecanismos de integração de linguagens orientada a objetos com persistência em banco de dados relacionais. Para tanto, foi criado um protótipo de um sistema de contas a pagar e receber e feito uma avaliação da implementação utilizando JBDC e o *framework* Hibernate.

Palavras chave: Persistência, Java, JDBC, ORM, Hibernate.

Abstract

The relational database systems and object-oriented languages are very used nowadays. These two different paradigms have to live together, making a semantic gap between object/relational paradigms. In this work a study of object oriented languages and relational database persistence integration mechanisms is presented. In order to do that, an account prototype system was created and the implementation using JDBC and Hibernate framework was evaluated.

Key Works: Persistence, Java, JDBC, ORM, Hibernate.

1. Introdução

Os sistemas de banco de dados relacionais e as linguagens orientada a objetos ganharam um espaço considerável no mercado nos últimos anos. O primeiro firmou-se no mercado e tem tido investimento pesado da indústria há muitos anos. O segundo vem ganhando aceitação generalizada, principalmente por causa da linguagem Java. Temos então as linguagens orientada a objetos e os banco de dados relacionais, dois paradigmas diferentes mas que tem que conviver juntos, criando assim esta incompatibilidade semântica entre os paradigmas objeto/relacional. Esta incompatibilidade pode ser solucionada usado banco de dados orientado a objetos, já disponíveis atualmente. No entanto, os bancos de dados orientados a objetos ainda não são tão eficientes e difundidos quanto os bancos de dados relacionais.

Segundo Bauer e King (2005), tradicionalmente, foram subestimados a importância e o custo desta incompatibilidade e as ferramentas para solucionar a incompatibilidade têm sido insuficientes. Enquanto isso, os desenvolvedores Java culpam a tecnologia relacional pela incompatibilidade; e os profissionais de banco de dados culpam a tecnologia de objetos. Isto acabou criando uma solução chamada mapeamento objeto/relacional (ORM). O mapeamento objeto/relacional é a persistência de objetos automatizada (e transparente) dentro de um aplicativo Java para as tabelas em um banco de dados relacional, usando metadados que descrevem o mapeamento entre os objetos e o banco de dados. O ORM, essencialmente, trabalha transformando dados (de modo reversível) de uma representação em outra.

Os objetivo deste trabalho são:

- Apresentar um estudo sobre a incompatibilidade entre os paradigmas objetos/relacional usando a linguagem Java;
- Comparar o acesso a banco de dados usando uma API convencional (JDBC) e o *framework* de mapeamento objeto/relacional Hibernate.

Este trabalho está organizado da seguinte forma: no capítulo 2 apresentamos um breve histórico sobre o modelo em camadas e como a persistência de dados se enquadra neste modelo.

O capítulo 3 contém um visão geral do JDBC (tecnologia que permite acessar banco de dados com a linguagem Java). No capítulo 4 aprensentamos os conceitos de ORM e no capítulo 5 o *framework* Hibernate (biblioteca ORM bastante utilizada).

No capítulo 6 apresentamos um estudo de caso comparando as tecnologias JDBC e Hibernate.

Por fim, concluímos o trabalho no capítulo 7.

2. O modelo em camadas

2.1. A Evolução da programação em camadas

Com a disseminação e evolução da Internet, surgiram muitas tecnologias e criaram-se padrões para muitas coisas, a programação em camadas é um exemplo, que é usado na maioria dos sistemas desenvolvidos em Java e para a *web*. Mas como se chegou a esta divisão das camadas e em qual camada vamos nos concentrar neste trabalho? Isto que vamos ver agora.

FIGURA 1 – MODELO MONOLÍTICO

Podemos dividir todas as aplicações de computadores em 3 partes: interface com o usuário, lógica de negócios e acesso aos dados. Mas isto não foi sempre assim, de acordo com (Qualid), durante mais de duas décadas a maioria dos sistemas de informações corporativos estava em um mainframe, que centralizavam os dados e os processos. A vantagem disto era uma melhor administração, um controle de acesso maior e os ajustes de performance e serviços do suporte técnico eram centralizados. Neste caso, a aplicação era composta de uma camada somente e por isso era chamada de **aplicações monolíticas** (Figura 1). As 3 partes do sistema estavam em um computador somente e isto trazia não só as vantagens já mencionadas, mas alguns problemas como pouca flexibilidade, custo muito alto para resolver problemas de pequeno e médio porte. Além disso, devido às limitações da tecnologia de interface dos aplicativos e à falta de liberdade do usuário para manipular seus dados, a qualidade dos serviços ficava abaixo das expectativas. Com estes problemas e surgindo a evolução das tecnologias de rede e ambientes distribuídos como sistemas operacionais e

sistemas gerenciadores de banco de dados, criou-se a primeira mudança de foco na área de sistemas computacionais, isto é, sistemas monolíticos para sistemas cliente/servidor (Figura 2).

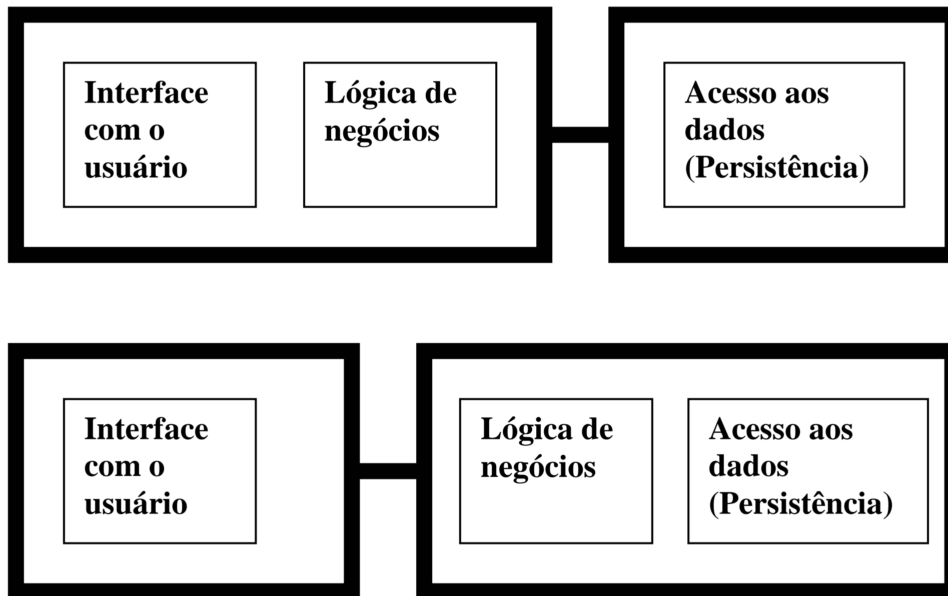

FIGURA 2 – MODELO CLIENTE SERVIDOR

O modelo **cliente/servidor** consiste em dividir o sistema em 2 camadas, podendo ficar a interface e a lógica de um lado e o acesso aos dados do outro ou a interface sozinha e a lógica e o acesso aos dados de outro. A abordagem cliente/servidor obteve sucesso porque solucionava muitos problemas presentes nos ambientes de mainframe, no entanto, também apresentava problemas, tais como: limitações quanto ao número de usuários simultâneos, a dificuldade de interação entre servidores e problemas causados pela distribuição de parte da lógica do negócio do lado do cliente.

Naturalmente surgiu uma demanda por aplicações cliente/servidor corporativas, ou seja, aplicações com as qualidades providas pela arquitetura cliente/servidor e com a robustez exigida por sistemas que integram toda a corporação. Para atender essa demanda, a arquitetura cliente/servidor (2-camadas) evolui para uma arquitetura 3-camadas, também conhecida como n-camadas, *fat server* ou corporativa (Figura 3). A arquitetura **3-camadas** surgiu com o advento dos servidores web e tem como objetivo separar banco de dados, regras de negócio e

interface com o usuário em camadas distintas permitindo que uma camada seja substituída ou sofra alterações sem grandes efeitos colaterais nas outras camadas. De acordo com (Ferreira, 2005), um sistema em camadas é organizado hierarquicamente e cada camada provê serviços para a camada superior e usa os serviços da camada inferior. Os conectores são definidos pelos protocolos que determinam como as camadas irão interagir.

FIGURA 3 – MODELO EM CAMADAS

Os sistemas divididos em 3 camadas apresentam um alto nível de abstração no projeto, permitindo o particionamento do problema, suporte a melhorias e grandes oportunidades de reuso, além da liberdade de modificar uma das partes, sem que isto gere alterações nas outras.

A seguir falamos sobre a camada de persistência, que é o foco deste trabalho.

2.2. A Camada de persistência

A persistência é um dos conceitos fundamentais em desenvolvimento de aplicativos. Segundo (Freitas e Martins, 2006), é difícil imaginar uma aplicação de software, independente do porte, que não tenha a necessidade de utilizar todos ou parte de seus dados entre as diversas execuções, ou de disponibilizá-los, para que a mesma aplicação os utilize em outros computadores ou por outras aplicações. Estes dados podem ser simples parâmetros para os componentes gráficos, como posicionamento e aparência, ou dados que descrevem recursos externos ou que fazem parte do negócio para o qual a aplicação tenha sido projetada. O processo de armazenamento e recuperação de dados, independente do meio de armazenamento, é conhecido como **persistência**. A forma como os dados se tornam disponíveis entre as diversas execuções de uma aplicação são variadas. De maneira geral,

podem ser simples arquivos com formato texto ou binário, estruturas padronizadas em arquivos XML (*eXtensible Markup Language*), estruturas organizadas em tabelas disponíveis em sistemas de gerenciamento de banco de dados, entre outros.

A forma mais utilizada de armazenamento é através de um sistema de gerenciamento de banco de dados (SGBD). Segundo (Korth e Silberschatz, 1995), o principal objetivo de um SGDB é prover um ambiente que seja adequado e eficiente para recuperar e armazenar informações. Os SGBD's são projetados para gerenciar grandes grupos de informações e isto envolve a definição de estruturas para armazenamento e o fornecimento de mecanismos para manipulá-las. Devem fornecer segurança das informações armazenadas, não permitindo acessos não autorizados e quando vários usuários manipularem dados, devem evitar possíveis resultados anômalos. As partes funcionais de um sistema de banco de dados incluem um gerenciador de arquivos, um gerenciador do banco de dados, um processador de consultas, um pré-compilador da DML (*Data Manipulation Language*) e um compilador da DDL (*Data Defination Language*). Além disto temos diversas estruturas de dados que fazem parte da implementação do sistema físico que são os arquivos de dados, o dicionário de dados e os índices. A figura 4 mostra estes componentes e as suas conecções.

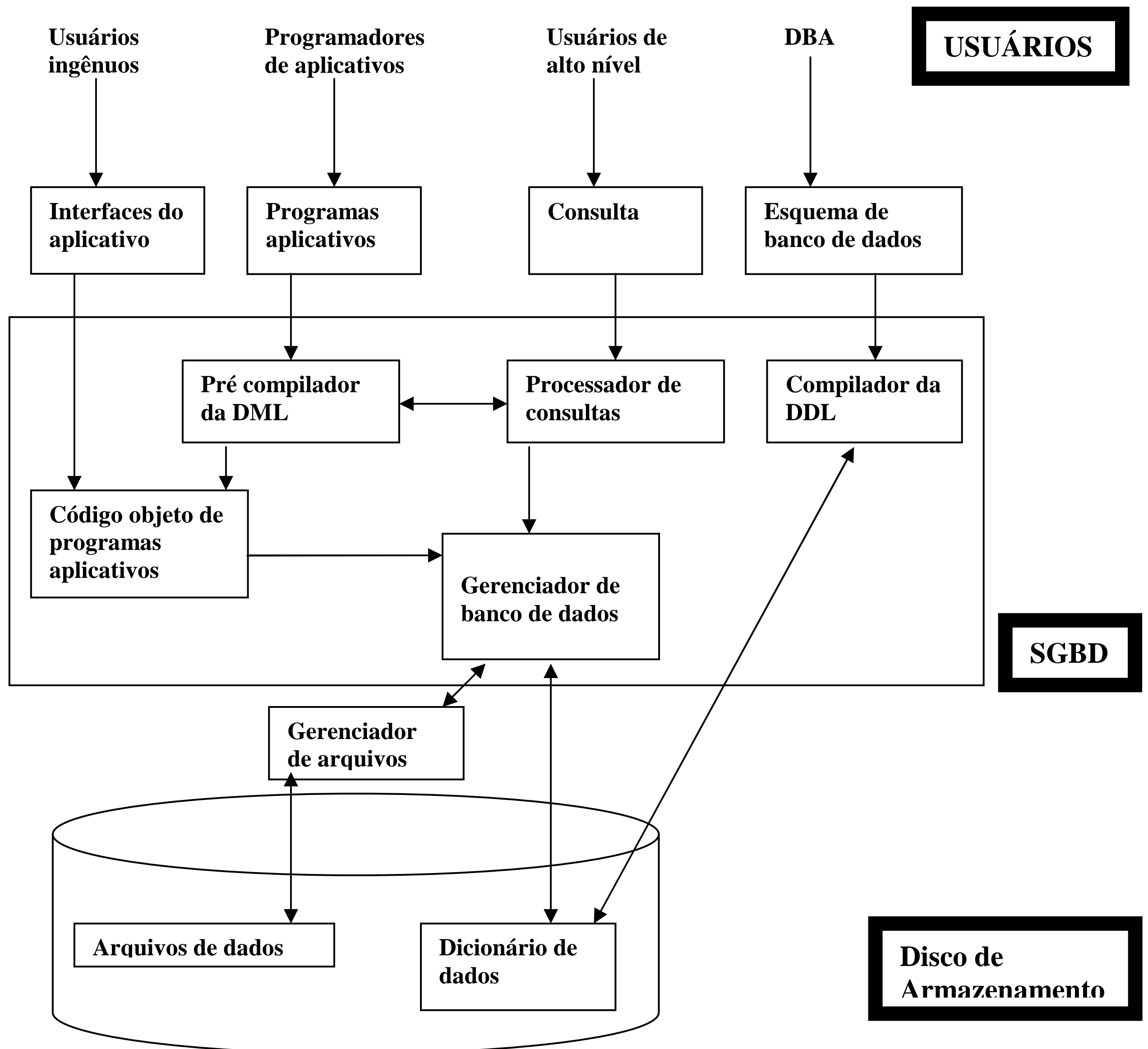

FIGURA 4 – ESTRUTURA DE UM SGBD

Os dois tipos de SGBD de maior relevância neste trabalho são o relacional e o orientado a objetos, por isto vamos ver as características mais importantes de cada um deles.

Os SGBD , segundo (Korth e Silberschatz, 1995), representa os dados e relacionamentos entre dados por um conjunto de tabelas, cada uma tendo um número de colunas com números únicos. Um banco de dados relacional é formado pelos componentes:

- Tabelas, também conhecidas como *tuplas,* que são um conjunto de registros ordenados, que são as linhas.
- Identificador, mais conhecido como chaves, que servem para identificar um registro entre tabelas, estas podem ser primarias, conhecidas como PK (de Primary Key), que pode ser constituída de uma coluna ou mais, definidos para distinguir registros. Podem ser Estrangeiras, estas são utilizadas para localizar o registro de uma tabela relacionada em outra, normalmente a chave primária de outra tabela.
- Integridade, são regras de consistência de dados, cabe a esta varias tarefas, como regras de valores nulos (uma campo pode ou não aceitar um valor nulo), integridade entre tabelas ou se as informações fornecidas em uma Chave estrangeira existe em sua tabela de origem.

A linguagem de consulta padrão para banco de dados relacional é SQL. Ela é composta de uma linguagem de definição de dados (DDL), uma linguagem de manipulação de dados (DML), comandos para definição de visões, comandos para especificação de autorização de acesso, comando para restrições complexas de integridade e comandos para especificação de inicio e fim de transações.

Os Bancos de Dados Orientados a Objeto possuem muitas características semelhantes aos sistemas de banco de dados relacionais, como persistência de dados, controle de transações e outros, mas existem também características próprias dos sistemas Orientados a Objeto, como o conceito de classes, objetos, encapsulamento e herança.

Os banco de dados orientada a objetos, de acordo com (Korth e Silberschatz, 1995)**,** é baseado num conjunto de objetos e estes contêm valores armazenados em variáveis de instância dentro do objeto e possui trechos de códigos que operam sobre o objeto que são chamados de métodos. Estes são a único modo de um objeto fazer acesso ao dado de um outro objeto, isto é enviar uma mensagem ao objeto.

Para que possamos entender melhor os bancos de dados Orientados a Objetos, seguem algumas características do modelo Orientado a Objetos:

- Classes: São formadas a partir de um grupo de objetos semelhantes. Os objetos são na verdade instâncias de uma classe, cada uma é tratada como um objeto e possui um identificador único.
- Objetos: São utilizados para gerar uma abstração sobre algo real, um objeto é basicamente composto por:
    - Um conjunto de variáveis, onde são armazenados os dados;
    - Um conjunto de mensagens, que são as responsáveis pela comunicação entre os objetos;
    - Um conjunto de métodos, que é o código para programar uma mensagem.
- Identidade de Objeto: O objeto deve manter sua identidade, mesmo que seus valores sofram alterações com o tempo. Cada objeto possui um identificador único, que o seguira por toda sua existência.
- Encapsulamento: Mecanismo pelo qual os dados e métodos ficam *escondidos* dentro do objeto. Podemos definir que um objeto só pode ter sua estrutura visível para o desenvolvedor do objeto.
- Herança: Através da herança podemos "transferir" características de uma classe à outra, sua principal vantagens é a da reusabilidade de código, pois os programas podem compartilhar um grande número de objetos.

A linguagem de consulta para bancos orientados a objetos, é chamada de OQL *(Object Query Language).* Da mesma forma que existe a DDL e a DML nos bancos relacionais, existe a ODL *(*Linguagem de Definição de Objetos, do inglês "*Object Definition Language")* e a OQL *(*Linguagem de Consulta de Objetos, do inglês "*Object Query Language").* Os requisitos básicos para que essa linguagem possa ser integrada as linguagens de programação Orientada a Objeto, são que sejam declarativas e que trabalhem com as principais características do modelo de dados Orientados a Objetos, como por exemplo a herança.

A maioria das linguagens de programação fornecem meios para acessar SGBD e Java não poderia ser exceção. Com esta finalidade foi criada a API JDBC, que é a forma nativa e de baixo nível da linguagem para manipulá-los. Isto será melhor explicado no próximo capítulo.

3. O JDBC

3.1. Visão geral sobre JDBC

A linguagem Java permite escrever programas que sejam independentes de plataforma, da mesma forma é possível acessar banco de dados relacionais com esta linguagem e manter esta independência, para tanto, a tecnologia JDBC é usada. O JDBC (Java Database connectivity) é uma interface padrão de acesso a bancos de dados SQL. Esta interface oferece acesso uniforme a diversos bancos de dados relacionais, possibilitando que um programa Java envie instruções SQL de forma genérica. Em outras palavras, não há necessidade de desenvolver uma aplicação voltada para um banco de dados específico, pois a API para a programação é a mesma para qualquer banco de dados.
O JDBC foi desenvolvido pela Sun Microsystems em conjunto com líderes mundiais no setor de banco de dados, garantindo desta forma, grande aceitação no mercado.

O elemento de software que faz a ligação entre uma aplicação Java e um determinado SGBD (Sistema Gerenciador de Banco de Dados) através da API JDBC é chamado de ***Driver.*** Um *driver* JDBC pode ser escrito completamente em Java, de forma que possa ser baixado como parte de uma *applet* ou pode ser implementado usando métodos nativos, que criariam uma espécie de ponte para as bibliotecas existente. A figura 5 ilustra a tecnologia do JDBC. Os programas Java acessam o SGBD efetuando declarações SQL padrão. Estas declarações são enviadas ao gerenciador de *drivers* , que se encarrega de utilizar o driver disponível para o SGBD utilizado. Caso um driver específico para o SGBD na esteja disponível, então uma ponte JDBC/ODBC, que envia as declarações SQL para o driver ODBC (interface de programação C para SQL), é utilizada.

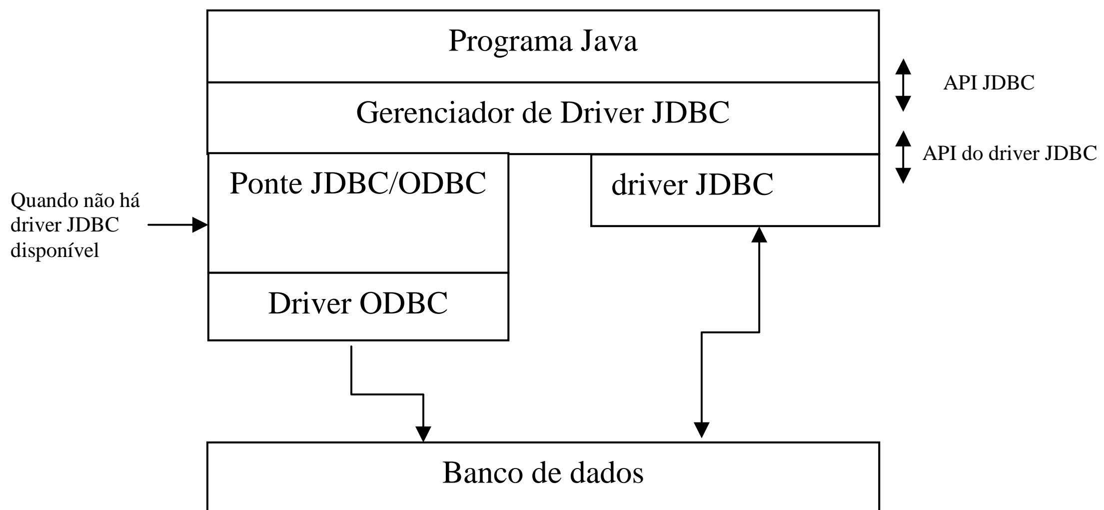

FIGURA 5 - JDBC

De acordo com Umeda, as APIs JDBC define classes Java para representar conexões com o Banco de Dados, setenças SQL, Resultados, meta dados etc. Em outras palavras, JDBC dá condições de realizar três operações: estabelecer uma conexão com um banco de dados, enviar instruções SQL, recuperar e processar os resultados. As principais interfaces do JDBC são:

- `java.sql.DriverManager`
- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.ResultSet`
- `java.sql.Driver`

3.2. Funcionalidades

O primeiro passo para utilizar o banco de dados com JDBC, é conectar ao banco de dados, em seguida é possível executar comandos para manipular os dados (DDL) e realizar consulta (DML), sendo que cada tipo de comando possui um modo ou uma sintaxe diferente. Em seguida apresentamos alguns exemplos utilizados que utilizam o banco de dados firebird. Note que o JDBC oferece independência do banco de dados, no entanto, é necessário especificar o nome do banco de dados para o carregamento do driver.

3.2.1. Conexão com o Banco de Dados

Para entendermos melhor como o JDBC funciona, vamos mostrar um exemplo bem simples do que é preciso para um programa Java fazer uma consulta a um banco de dados.

```
01: Import  java.sql.*
02: Public class MiniJDBC
03: {
04:    static final String JDBC_DRIVER = "org.firebirdsql.jdbc.FBDriver";
05:    static final String DATABASE_URL = "jdbc:firebirdsql://localhost/3050:
                            /ctmo/java/financeiro/financeirojava.fdb ";
06:   plubic static void main( String args[])
07:    Connection connection = null;
08:    Statement statement = null;
09:    try
10:   {
11:     Class.forName(JDBC_DRIVER);
12:     Connection = DriverManager.getConnection(DATABASE_URL,"usuario",
13: "senha");
14:     Statement = connection.createStatement();
15:     ResultSet resultSet = statement.executeQuery("Select * from  pessoas");
16:     .......
17:   }
18:   catch (ClassNotFoundException e) {
19:             e.printStackTrace();
20:         } catch (SQLException e) {
21:             e.printStackTrace();
22:       }
23: }
```
Quadro 1 - Exemplo de conexão JDBC com o banco de dados

Primeiro é preciso de uma cadeia que contém o nome do *driver* JDBC (podemos chamá-la de JDBC_DRIVER). Esta cadeia é usada para carregar o *driver* para a memória. É preciso também cadeia da URL do banco de dados (podemos chamá-la de DATABASE_URL), que identifica o nome do banco de dados ao qual a conexão será realizada e as informações sobre o protocolo utilizado pelo *driver* . Nesta cadeia estão contidos o protocolo de comunicação, o

subprotocolo de comunicação (que identifica qual banco de dados será usado) e a localização do banco de dados É usado o método estático *forName* da classe Class é então utilizado para carregar o *drive*. Depois, é criado um objeto *Connection,* que gerencia a conexão entre o programa Java e o banco de dados. Este objeto permite aos programas criar instruções SQL que acessam o banco. O método *getConnection* aceita 3 argumentos: o primeiro é a cadeia DATABASE_URL (descrita anteriormente), o nome do usuário e a senha do usuário. Até este ponto é feita a conexão com o banco, e já estamos prontos para enviar comandos SQL, isto pode ser realizado pelo método Connection *createStatement* que obtêm um objeto Statement. Deste é usado o método *executeQuery* para enviar a consulta SQL finalmente. A consulta é retornada em um objeto que implementa a interface *resultSet*, que permite manipular o resultado da consulta. No quadro 1 é mostrado um exemplo do código citado acima.

3.2.2. Instruções DDL

Para executar operações de DDL (*Data Defination Language)*, que permite enviar comandos para o banco de dados para criar tabelas, indices, visão, *store procdure*, etc, o JDBC disponibiliza o método *executeUpdate()*. Para executá-lo, é preciso obter antes um objeto *Statement*, através de um objeto *Connection* já instanciado, como no exemplo mostrado acima, depois é só montar o comando desejado e enviá-lo através do método citado. Nos quadros abaixo são mostrados alguns exemplos deste método, já levando em consideração que a conexão com o banco já foi efetuada, como foi demonstrado acima. No quadro 2 é criada a tabela tb_pessoas, no quadro3 e 4 é mostrado um exemplo que insere um registro na tabela que acabamos de criar, com a diferença que um passa todos os valores para o método na cadeia que é enviada ao banco e o outro usa instruções preparadas. Estas instruções são características que possibilitam pré-compilar uma instrução SQL que está freqüentemente sendo utilizada, proporcionando um melhor desempenho, pois não é necessário compilá-la toda vez que é enviada para o banco de dados. Também permite definir parâmetros posicionais que podem tornar o código mais legível. Sempre que o banco de dados executa uma consulta, ele primeiramente processa uma estratégia de como executá-la de maneira eficiente. Ao preparar a consulta e reutilizá-la, essa etapa de planejamento é feita apenas uma vez. Lembrando que esta estratégia utilizada pelo banco, se altera conforme a quantidade de

dados armazenados. Portanto, deve-se combinar as instruções preparadas, com outras formas de otimização do banco. Cada variável em uma consulta preparada é indicada por um "?". Se houver mais de uma variável, um registro de posições do "?" deverá ser mantido ao definir os valores, como é mostrado no quadro 4.

```
01: Statement.executeUpdate("
02: CREATE TABLE TB_PESSOAS (
03:    CODIGO DMNCODUN NOT NULL,
04:    NOME DMNVARCHAR50,
05:    ENDERECO DMNVARCHAR50,
06:    CIDADE DMNVARCHAR30,
07:    TELEFONE DMNVARCHAR20);
08:");
```

Quadro 2 – Exemplo de um Create Table com JDBC

```
01: Statement.executeUpdate("
02:          INSERT INTO TB_PESSOAS(CODIGO, NOME, ENDERECO, CIDADE, TELEFONE)
03:          VALUES (1, 'Marcos', 'Rua do Jose', 'Maringa', '044 3598-4578')
04:     ")
```

Quadro 3 – Exemplo de um Insert na tabela TB_PESSOAS com JDBC

```
01:  Statement = con.prepareStatement("insert into tb_pessoas(CODIGO, NOME,
02:                  ENDERECO, CIDADE, TELEFONE) VALUES (?,?,?,?,?) ");
03:            Statement.setInt(1,1);
04:            Statement.setString(2,'Marcos');
05:            Statement.setString(3, 'Rua do Jose');
06:            Statement.setString(4, , 'Maringa');
07:            Statement.setString(5, '044 3598-4578');
08:            Statement.executeUpdate();
```

Quadro 4 – Exemplo de um Insert com parâmetros na tabela TB_PESSOAS com JDBC

3.2.3. Consultas SQL

Como dito anteriormente, o JDBC disponibiliza o método *executeQuery()* para realizar consultas SQL. Estas consultas seguem o padrão SQL e podem ser executadas passando uma string com o comando SQL, com ou sem parâmetros posicionais. No quadro 5 podemos ver que depois de executar a consulta, recebemos um *resultset*(linha 5). Depois para obter o primeiro registro usamos o método *get()* levando em consideração o tipo da coluna, *getString()* para ler uma string, *getInteger()* pra ler um inteiro e assim por diante. Vemos na linha 6 até a 12 que podemos fazer um laço para obter todos os registros e todos os seus conteúdos.

```
01:        Statement stm = null;
02:        try {
03:            stm = con1.createStatement();
04:            vSelect = "select first 1 * from tb_pessoas ";
05:            ResultSet rs = stm.executeQuery(vSelect);
06:               while (rs.next()) {
07:                pessoa.setCodun(rs.getInt("CODUN"));
08:                pessoa.setNome(rs.getString("NOME"));
09:                pessoa.setEndereco(rs.getString("ENDERECO"));
10:                pessoa.setCidade(rs.getString("CIDADE"));
11:                pessoa.setTelefone(rs.getString("TELEFONE"));
12:            }
13:            return pessoa;
14:
15:        } catch (SQLException ex) {
16:            ex.printStackTrace();
17:        }
```
Quadro 5 – Exemplo de consulta a toda uma tabela com JDBC.

3.2.4. Outras características

Outros recursos do JDBC que estão disponíveis, mas não são usados freqüentemente são os procedimentos armazenados(Store procedures) e a consulta a metadados. O primeiro permite executar rotinas armazenadas dentro do banco de dados com uma linguagem própria que é mais conhecida como Store Procedures e o outro permite que se tenha acesso a informações sobre a estrutura do banco de dados, que também é chamado de metadados. O JDBC possui dois tipos de metadados: um sobre banco de dados e o outro sobre um resultado de consulta, para isso utiliza-se o método *DatabaseMetaData*

Um outro recurso é o Escape do JDBC, que mapeia uma sintaxe padrão JDBC para a sintaxe específica do fabricante. Quando é utilizado um driver que dá suporte a essa sintaxe, não é preciso mudar o modo em que o programa utiliza a API JDBC; somente é alterada a maneira pelo qual as instruções SQL são geradas. A sintaxe de escape do JDBC oferece meios padronizados para mapear chamadas de procedimentos armazenados, funções escalares, expressões de data e hora, e especificações externas de ligação para SQL específica do fabricante.

4. ORM

De acordo com (Bauer e King, 2005), ORM(Mapeamento objeto-relacional) é a persistência de objetos automatizada (e transparente) dentro de um aplicativo Java para as tabelas de banco de dados relacional, usando metadados que descrevem o mapeamento entre os objetos e o bando de dados. Esta técnica de desenvolvimento é utilizada para reduzir a diferença semântica da programação orientada a objetos e a persistência em bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes.

É importante notar que a diferença semântica citada anteriormente é solucionada de forma **automatizada** pelo ORM, sem a necessidade do desenvolvedor conhecer a linguagem SQL e o funcionamento do JDBC, é necessário apenas configurar os arquivos de mapeamento da aplicação com o banco de dados. Esta técnica elimina linhas de código repetitivas e propensas a erro usadas com acessos de baixo nível ao banco de dados, facilitando a manutenção posterior do sistema. Em contrapartida, existe um perda de desempenho e aumento no custo de configuração.

Como dito anteriormente, o mapeamento da relação entre as tabelas onde originam os dados e o objeto que os disponibiliza é configurada pelo programador, não é necessário uma correspondência direta entre as tabelas de dados e as classes do programa, o código do programa é isolado das alterações à organização dos dados nas tabelas do banco de dados. A forma como este mapeamento é configurado depende da ferramenta utilizada.

Segundo (Bauer e King, 2005), ORM consiste de quatro partes:

- Uma API para executar operações de criar, ler, atualizar e apagar em objetos de classes persistentes.
- Uma linguagem ou API para especificar consultas que referenciam classes e propriedades de classes.
- Um recurso para especificar o mapeamento de metadados.
- Uma técnica para interagir com objetos transacionais a fim de executar a verificação suja, buscas de associações ociosas e outras funções de otimização.

O ORM pode ser dividido em quatro níveis: O relacional puro, o mapeamento de objetos leves, o mapeamento de objeto médio e o mapeamento completo de objetos. O nível de dependência ou de independência do modelo relacional vai diminuindo a medida que o mapeamento fica mais completo. E no último nível a modelagem sofisticada de objetos é suportada como herança, polimorfismo, etc.

Podemos listar alguns benefícios do ORM como produtividade, manutenção, desempenho e independência de fornecedor. A produtividade é alcançada porque elimina muito trabalho bruto que a camada de persistência gera por ser um código tedioso de se implementar. Gera menos linhas de código tornando o sistema mais compreensível e por isto a manutenção é mais fácil de se ser realizada. Além do que, como agora o sistema e o banco de dados são orientada a objetos, uma mudança no objeto gera menos problemas para o banco. Em relação ao desempenho podemos pensar que a persistência feito pelo acesso nativo pode ser mais rápida que a feita por ORM, isto é uma verdade se levarmos só em comparação a velocidade, mas o que ocorre é que os problemas de tempo e de orçamento é que são os relevantes em um projeto. As otimizações que podem ser feitas com uma persistência automatizada, são muito mais importantes a outra. Sem falar que isto melhora o tempo a produtividade do desenvolvedor. E por último, a independência de fornecedor que abstrai o desenvolvedor do dialeto SQL e dá ao aplicativo uma maior portabilidade.

No próxima seção vamos falar sobre o Hibernate, que é uma *framework* ORM de código aberto mais utilizado.

5. Hibernate

Segundo Bauer e King, 2005, o Hibernate é um *framework* de mapeamento objeto/relacional para Java. Ele transforma os dados tabulares de um banco de dados em um grafo de objetos definido pelo desenvolvedor. Usando o Hibernate, o desenvolvedor se livra de escrever muito do código de acesso a banco de dados e de SQL, acelerando a velocidade do seu desenvolvimento de uma forma fantástica. É uma implementação de ORM de código aberta. Ele tenta solucionar o problema de gerenciamento de dados persistentes em Java. Ele é o intermediário entre a iteração do aplicativo com um banco de dados relacional, deixando o desenvolvedor livre para se concentrar na lógica de negócio.

Para usar o Hibernate devemos definir classes, criar arquivos de mapeamento e dependendo do caso, criar um arquivo de configuração para o Hibernate. Mas antes de vermos um exemplo de como isto é feito, mostraremos uma visão geral do Hibernate.

5.1. As interfaces do Hibernate

O primeiro passo para utilizar o Hibernate dentro da camada de persistência de um aplicativo, é conhecer as interfaces de programação.. A figura 6 mostra os papéis das interfaces mais importantes do Hibernate dentro das camadas de persistência e da camada de negócios.

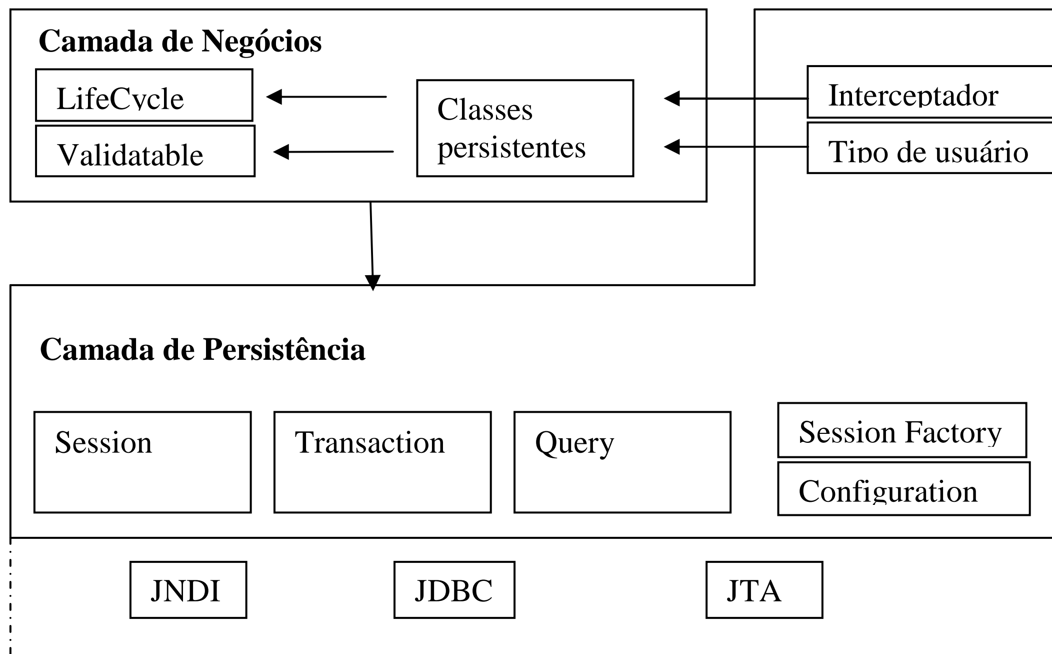

FIGURA 6 – VISÃO GERAL DA API DO HIBERNATE

As APIs de Java como JDBC, a JTA(API de transação) e a JNDI (API da interface de diretórios e de atribuição de nomes) são usadas pelo Hibernate para suportar acesso a quase todos os bancos de dados relacional e integração com os servidores de aplicativos J2EE.

As interfaces principais ou essências para os aplicativos que usam hibenate são: *Session, Session Factory, Configuration, Transaction e Query e Critéria*. Elas permitem armazenar e obter objetos persistentes e controlar transações.

A interface *Session* é a principal do Hibernate, e precisará ser criada e destruída o tempo todo. Ela está entre a conexão e a transação e pode ser pensada como um armazenamento em cachê ou uma coleção de objetos carregados que se relaciona com uma única unidade de trabalho.

A interface *Session Factory* é única por aplicativo ou melhor por banco de dados e pode ser criada na inicialização do aplicativo. Armazena em cache as instruções SQL, outros metadados de mapeamento que o Hibernate usa em tempo de execução e dados armazenados que tenham sido lidos em uma unidade de trabalho e podem ser reutilizados futuramente.

A interface de *Configuration* é usada para a inicialização, usando uma instância de configuration para especificar o local dos arquivos de mapeamento e das propriedades específicas do Hibernate para criar a SessionFactory.

A interface *Transaction* é opcional, deixando que as transações sejam tratadas diretamente. Esta interface abstrai o modo que é usado o controle das transações no Hibernate, podendo ser por JDBC, JTA, CORBA, etc. Isto é interessante porque ajuda a manter os aplicativos portáveis entre os diferentes tipos de ambientes de execução e de containers.

As interfaces *query* e *criteria* permitem que se faça consultas ao banco de dados e controle como ela é executada. As consultas são escritas em HQL ou no dialeto nativo SQL do banco de dados.

Uma outra interface que pode ser usada no Hibernate é a interface de *callback*. Ela permite que o aplicativo receba uma notificação quando alguma coisa interessante acontecer ao objeto , isto pode ser quando um objeto é carregado, salvo ou deletedo. Isto pode ser útil para criar registros de auditoria. As interfaces do Hibernate para isto são: *Lifecycle, Validatable e Interceptor.*

5.2. Configuração mínima

A primeira coisa a fazer para usar o Hibernate em uma aplicação e iniciá-lo e isto é feito criando uma sessão *Session Factory*. Mas ele precisa de um *Configuration* que é utilizado para definir o local dos arquivos de mapeamento, e por convenção tem a extensão .hbm.xm. Depois disto cria-se a *SessionFactory*, tendo um banco de dados somente, caso sejam múltiplos bancos pode ser criada uma para cada um deles. Uma outra convenção é ter um arquivo de mapeamento para cada classe que será mapeado do banco de dados e é recomendado que estas classes estejam e um diretório único.

Para configurar a *SessionFactory* corretamente existem algumas técnicas, mas a forma mais utilizada é usar um arquivo de configuração. Ele pode ser o Hibernate.cfg.xml, que contém os parâmetros de configuração do Hibernate e pode também especificar a localização dos

documentos de mapeamento. No quadro 6 temos uma parte do arquivo que foi usado para fazer acesso ao firebird nos testes da parte prática.

```
01: <!DOCTYPE hibernate-configuration PUBLIC
02:     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
03:     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
04:    E:\Arquivos de programas\Java\jdk1.5.0_06\jre\hibernate-3.2.1.ga\hibernate-
05:       3.2\src\org\hibernate\
06:
07: <hibernate-configuration>
08:    <session-factory >
09:
10:         <property name="connection.driver_class"> org.firebirdsql.jdbc.FBDriver
11:        </property>
12:         <property name="connection.url">
13:       jdbc:firebirdsql:localhost/3050:c:\ctmo\java\financeiro\financeirojava.fdb
14:   </property>
15:         <property name="connection.username">SYSDBA</property>
16:         <property name="connection.password">masterkey</property>
17:         <property name="dialect">org.hibernate.dialect.FirebirdDialect</property>
18:
19:        <property name="show_sql">true</property>
20:
21:        <!-- Condiguração do c3p0 -->
22:        <property name="hibernate.c3p0.max_size">10</property>
23:        <property name="hibernate.c3p0.min_size">2</property>
24:        <property name="hibernate.c3p0.timeout">5000</property>
25:        <property name="hibernate.c3p0.max_statements">10</property>
26:        <property name="hibernate.c3p0.idle_test_period">3000</property>
27:        <property name="hibernate.c3p0.acquire_increment">2</property>
28:        <!-- Configurações de debug -->
29:        <property name="show_sql">true</property>
30:        <property name="hibernate.generate_statistics">true</property>
31:        <property name="hibernate.use_sql_comments">true</property>
32:
33:        <mapping resource="Pessoa.hbm.xml"/>
34:        <mapping resource="Cliente.hbm.xml"/>
35:
36:    </session-factory>
37: </hibernate-configuration>
```

Quadro 6 - Exemplo do arquivo hibernate.cfg.xml para firebird

Este arquivo pode ser dividido em partes: declaração do tipo de documento que engloba a sessão <hibernate-configuration> (linha 7), as propriedades do Hibernate que são os <property> (linha 14) e por último os arquivos de mapeamento das classes (linha 33 e 34). Uma opção que não foi usada neste exemplo, mas que está disponível é o nome da *SessionFactory.* Com um arquivo como este pode-se iniciar uma sessão da seguinte forma:

```
SessionFactory sessions = new Configuration()
  .configure().buildSessionFactory();
```

neste caso o arquivo de configuração é buscado no *classpath*, mas pode ser passado o local e o nome do arquivo, como mostrado abaixo:

```
SessionFactory sessions = new Configuration()
  .configure("\hibernate-config/meuhibernate.cfg.xml")
  .buildSessionFactory();
```

Com esta opção de criar um arquivos de configuração para iniciar o Hibernate, pode-se criar vários arquivos para bancos de dados diferentes e iniciá-los de acordo com configurações que podem ser feitas via programação.

5.3. Os tipos do Hibernate

Um tipo no Hibernate serve para mapear um tipo Java para um tipo de coluna do banco de dados, isto engloba também associações das classes persistentes. Esta característica torna o Hibernate extremamente flexível e extensível. Já que ela abrange todos os tipos primitivos de Java. Se isto não bastasse, o Hibernate suporta tipos definidos pelo usuário, através das interfaces *UserType* e *CompositeUserType*.

Vamos nos concentrar nos tipos básicos, assim teremos uma visão geral de como usar os tipos do Hibernate. Os tipos nativos do Hibernate podem compartilhar o mesmo nome do tipo de Java, apesar de poder existir mais de um tipo do Hibernate equivalente ao um tipo do Java. Não é permitido usar conversões de tipos primitivos entre Java e Hibernate, como por exemplo, mapear um *varchar* para um *integer*. As tabelas abaixo mostram os tipos equivalentes para o mapeamento de tipos primitivos, data e hora e para objetos blob e clob.

Tipo de Mapeamento	Tipo Java	Tipo nativo SQL padronizado
Integer	Int ou java.lang.integer	INTEGER
Long	Long ou java.lang.Long	BIGINT
Short	Short ou java.lang.Short	SMALLINT
Float	Float ou java.lang.Float	FLOAT
Double	Double ou java.lang.Double	DOUBLE
big_decimal	Java.math.BigDecimal	NUMERIC
Character	java.lang.String	CHAR
String	java.lang.String	VARCHAR
Byte	byte ou java.lang.Byte	TINYINT
Boolean	Boolean ou java.lang.Boolean	BIT
yes_no	Boolean ou java.lang.Boolean	CHAR(1) (‘Y’ OU ‘N’)
true_false	Boolean ou java.lang.Boolean	CHAR(1) (‘T’ OU ‘F’)

Tabela 1 – Tipos Primitivos

Tipo de Mapeamento	Tipo Java	Tipo nativo SQL padronizado
Date	Java.util.date ou Java.sql.date	DATE
Time	Java.util.date ou Java.sql.time	TIME
TimeStamp	Java.util.date ou Java.sql.timestamp	TIMESTAMP
Calendar	Java.util.calendar	TIMESTAMP
Calendar_date	Java.util.calendar	DATE

Tabela 2 – Tipos de data e hora

Tipo de Mapeamento	Tipo Java	Tipo nativo SQL padronizado
Binário	Byte [ ]	VARBINARY(ou BLOB)
Texto	Java.lang.String	CLOB
Serializável	Qualquer classe Java que implemente Java.io.serializable	VARBINARY(ou BLOB)
Clob	Java.sql.clob	CLOB
Blob	Java.sql.blob	BLOB

Tabela 3 – Tipos de objetos binários e grandes

A seguir veremos os arquivos de mapeamento das classes.

5.4. Arquivos de Mapeando de classes

Vamos analisar o mapeamento simples de uma tabela pessoa (quadro 7), através de arquivos XML. Este arquivo começa normalmente com as definições da DTD e do nó raiz (linhas 2 e 3), o <hibernate-mapping>, e depois vem o nó <class> (linha 5). Como o projeto foi dividido em pacotes temos no "name" o nome do pacote, seguido do nome da classe, financeiro.modelo.Pessoa. Indicando pela palavra *table* que esta classe foi mapeada para a tabela TB_PESSOAS do banco de dados.

Em seguida, temos o nó <id> (linha 7) que é o identificador dessa classe no banco de dados. Neste nó nós definimos a propriedade que guarda o identificador do objeto no atributo "name", que no nosso caso é "id", se o nome da coluna no banco de dados fosse diferente da propriedade do objeto, ela poderia ter sido definida no atributo "column". Ainda dentro deste nó, nós encontramos mais um nó, o <generator>, este nó 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 "increment", que incrementa um ao valor da chave sempre que insere um novo objeto no banco, esse gerador costuma funcionar normalmente em todos os bancos. Os próximos nós do arquivo são os <property> (linha 10 a linha 13) que indicam propriedades simples dos nossos objetos, como Strings, objetos Date, Calendar, Locale, Currency e outros.

```
01: <?xml version="1.0"?>
02: <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
03:  "http://hibernate.sourceforge.net/hibernate-mapping.dtd">
04: <hibernate-mapping>
05:    <class name="financeiro.modelo.Pessoa" table="TB_PESSOAS">
06:
07:        <id name="codigo" column="codun" type="integer">
08:            <generator class="assigned"/>
09:        </id>
10:        <property name="nome" type="string"/>
11:        <property name="endereco" type="string"/>
12:        <property name="telefone" column="telefone" type="string"/>
13:        <property name="cidade" column="cidade" type="string"/>
14:    </class>
15: </hibernate-mapping>
```

Quadro 7 – Mapeamento simples de uma tabela TB_Pessoas para classe Pessoa

Neste nó os atributos mais importante são “name”, que define o nome da propriedade, “column” para quando a propriedade não tiver o mesmo nome da coluna na tabela e “type” para definir o tipo do objeto da propriedade. Normalmente, o próprio Hibernate é capaz de descobrir qual é o tipo de objeto da propriedade, não sendo necessário escrever isso no arquivo de configuração, ele também usa o mesmo nome da propriedade para acessar a coluna, se o atributo não tiver sido preenchido.

No quadro 8 temos um exemplo de arquivo de mapeamento de herança, sendo que a palavra *extends* na linha é que identifica a herança.

```
01: <?xml version="1.0"?>
02: <!DOCTYPE hibernate-mapping PUBLIC
03: "-//Hibernate/Hibernate Mapping DTD//EN"
04:    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
05:
06: <hibernate-mapping>
07:  <joined-subclass name="financeiro.modelo.Cliente" extends=
"financeiro.modelo.Pessoa">
08:
09:        <key column="codigoCliente"/>
10:         <property name="cpf"/>
11:     </joined-subclass>
12: </hibernate-mapping>
```

Quadro 8 – Mapeamento de herança

No quadro 9 vemos um arquivo de mapeamento de um para muitos, onde o nó <set> e o <one-to-many> que indica o mapeamento de um para muitos.

```
01: <hibernate-mapping>
02:     <class name="financeiro.modelo.TituloReceber" table="FN_TITULOSRECEBER">
03:         <id name="CODIGO" column="CODIGO" type="integer">
04:             <generator class="assigned"/>
05:         </id>
06:         <set name="CLIENTE" inverse="true">
07:              <key column="" />
08:              <one-to-many class="financeiro.modelo.Cliente"/>
09:        </set>
10:     </class>
11: </hibernate-mapping>
```

Quadro 9 – Mapeamento de um para muitos

5.5. O HQL

O HQL é a linguagem de consulta do Hibernate que segundo (Bauer e King, 2005), é um dialeto orientado para objetos da linguagem de consulta relacional familiar a SQL, e não é uma linguagem de manipulação de dados como SQL, mas é usado para obter objetos, nela você não vai referenciar tabelas, vai referenciar os objetos do modelo que foram mapeados para as tabelas do banco de dados. Além disso, por fazer perquisas em objetos, você não precisa selecionar as "colunas" do banco de dados, por exemplo para fazer um select de pessoa (select * from pessoa) em HQL será assim "from Pessoa", porque não estamos mais pensando em tabelas e sim em objetos.

Esta não é a única forma de se fazer consultas com o Hibernate, existe a API critéria para consulta por critérios e o acesso direto com SQL pelo mapeamento.Dos três métodos, o mais poderoso é o HQL, as suas consultas são fáceis de entender, e elas utilizam classe persistente e nomes de propriedades no lugar de nomes de tabelas e de colunas. A HQL é polimórfica: pode-se obter todos os objetos com uma determinada interface consultando essa interface. Com a HQL, você tem todo o poder de restrições arbitrárias e projeção de resultados, com operadores lógicos e chamadas de função exatamente como em SQL, mas sempre em nível de objetos usando nomes de classes e propriedade. Você pode usar parâmetros nomeados para ligar os argumentos de consulta de uma maneira segura de tipo. Outra característica que faltam em outras soluções ORM são as consultas com estilo de relatórios.

No quadro 10 é mostrado um exemplo simples de consulta HQL, onde consultamos uma tabela que foi mapeada para uma classe chamada pessoa. Esta consulta traz as 10 primeiras pessoas (identificado pelas linhas 4 e 5), ordenadas pelo campo nome.

```
01: Session sessao = HibernateUtility.getSession()
02: Transaction transaction = sessao.beginTransaction()
03: Query query = sessao.createQuery("from pessoas p order by nome");
04: Query.setFirstResult(0);
05: Query.setMaxResult(10);
```

Quadro 10 – Exemplo de uma consulta com HQL

Alguns dos recursos disponiveis para consulta no HQL são:

- Junção – Combinar uma ou mais tabelas em uma consulta
- Projeção – Claúsula select para colunas específicas da consulta
- Agregação – Funções count(), min(), max(), sum(), avg()
- Agrupamento – Claúsula group by
- Restrições em Grupos - Cláusula having

5.6. Ferramentas do Hibernate

O Hibernate disponibiliza algumas ferramentas para ajudar no processo de criação das classes Java ou do esquema para o banco de dados. Estas ferramentas podem ajudar quando se está partindo das classes Java para se criar o esquema de um banco de dados ou ao contrário, com um esquema de banco de dados já existente, gerar as classes Java. Nos dois casos é preciso ter o mapeamento das classes em xml, para que a partir dele seja possível gerar o esquema ou as classes Java. No primeiro caso é usado a ferramenta hbm2ddl e no outro caso a ferramenta usada é a hbm2java.

O hbm2dll parte do mapeamento das classes feita em xml e gera um uma DDL SQL que consiste basicamente de instruções CREATE e ALTER. O cuidado que se deve ter quando se usa esta ferramenta é que o mapeamento deve ter sido cuidadoso em relação aos nomes das tabelas e das suas colunas.

Já o hbm2java parte do mapeamento das classes e gera um arquivo de classes persisntentes Java. Por padrão, o arquivo gerado possui o construtor da classe e os métodos de acesso e implementa os métodos recomendados toString() e equals() com a semântica padrão.

Os arquivos gerados pelas ferramentas acima , devem ser revisados, para verificar se precisam de alguma ajuste, devido a limitação da ferramenta, já que o objetivo delas é livrar de executar tarefas repetitivas que ocorrem quando se trabalha com o Hibernate.

6. Estudo de caso: Comparação entre JDBC e Hibernate

6.1. A aplicação

A aplicação que foi desenvolvida é um sistema financeiro simplificado, composto dos seguintes itens:

- Pessoa – Uma tabela base que depois será usada para herança
- Cliente – Um cadastro de clientes
- Fornecedor – Um cadastros de fornecedor
- Titulo a pagar – Contas a serem pagas
- Título a receber – Contas as serem recebidas
- Lançamentos – Lançamentos das contas dos titulos

A pessoa é a base para cliente e fornecedor, isto quer dizer que eles herdam as características de pessoa. Os títulos a pagar e a receber identificam as contas que existem no sistema para serem pagas e recebidas, sendo que o a pagar está relacionado com fornecedor e o receber com clientes. Estes títulos podem ter lançamentos que identificam o tipo da conta, que pode ser uma peça comprada por um cliente ou uma determinada duplicata a ser paga para um fornecedor, por exemplo.

Nas figuras 7 e 8 são mostradas o diagrama de objetos e o diagrama de entidade-relacionamento da aplicação.

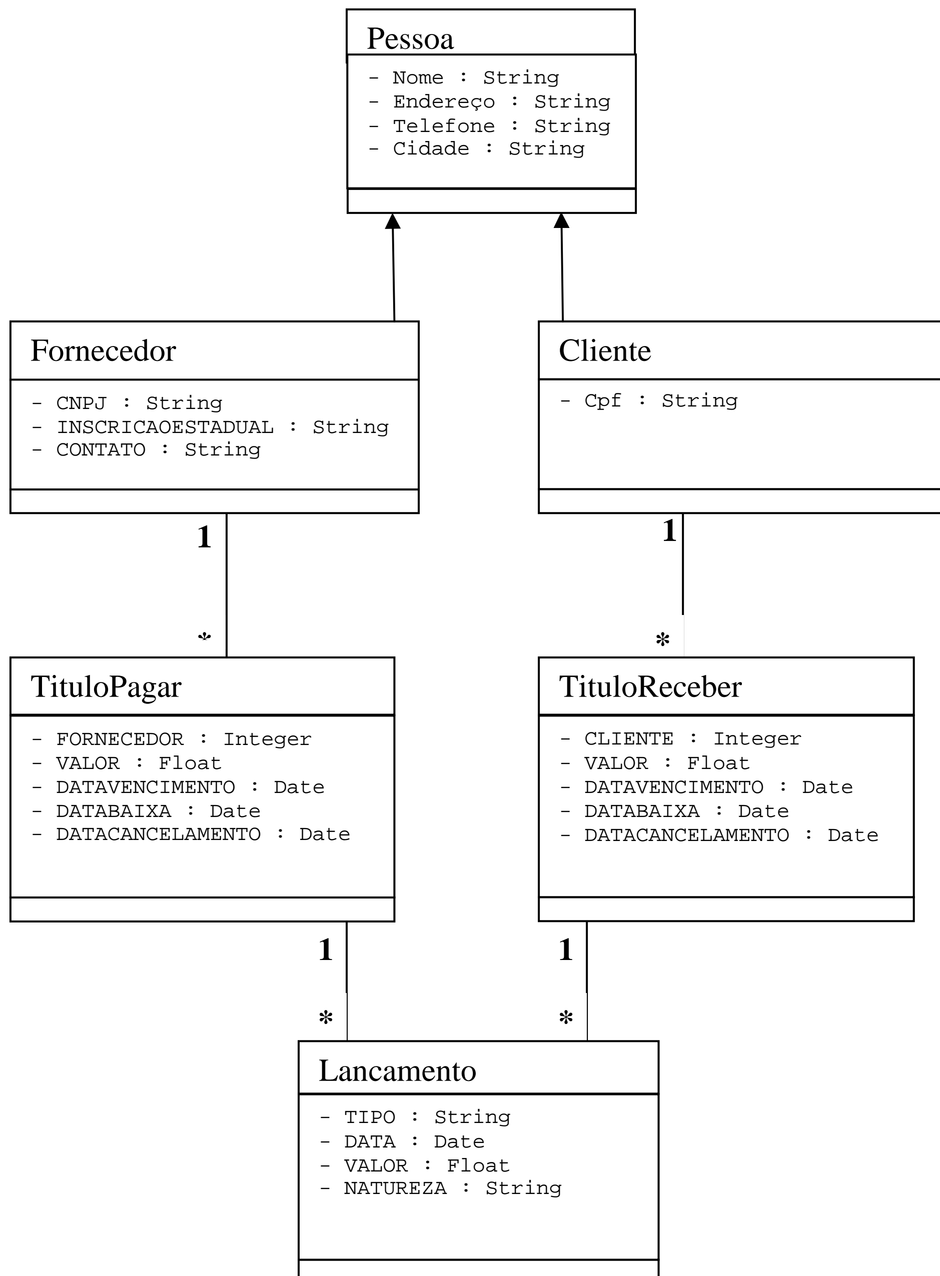

FIGURA 7 – Diagrama De Objetos

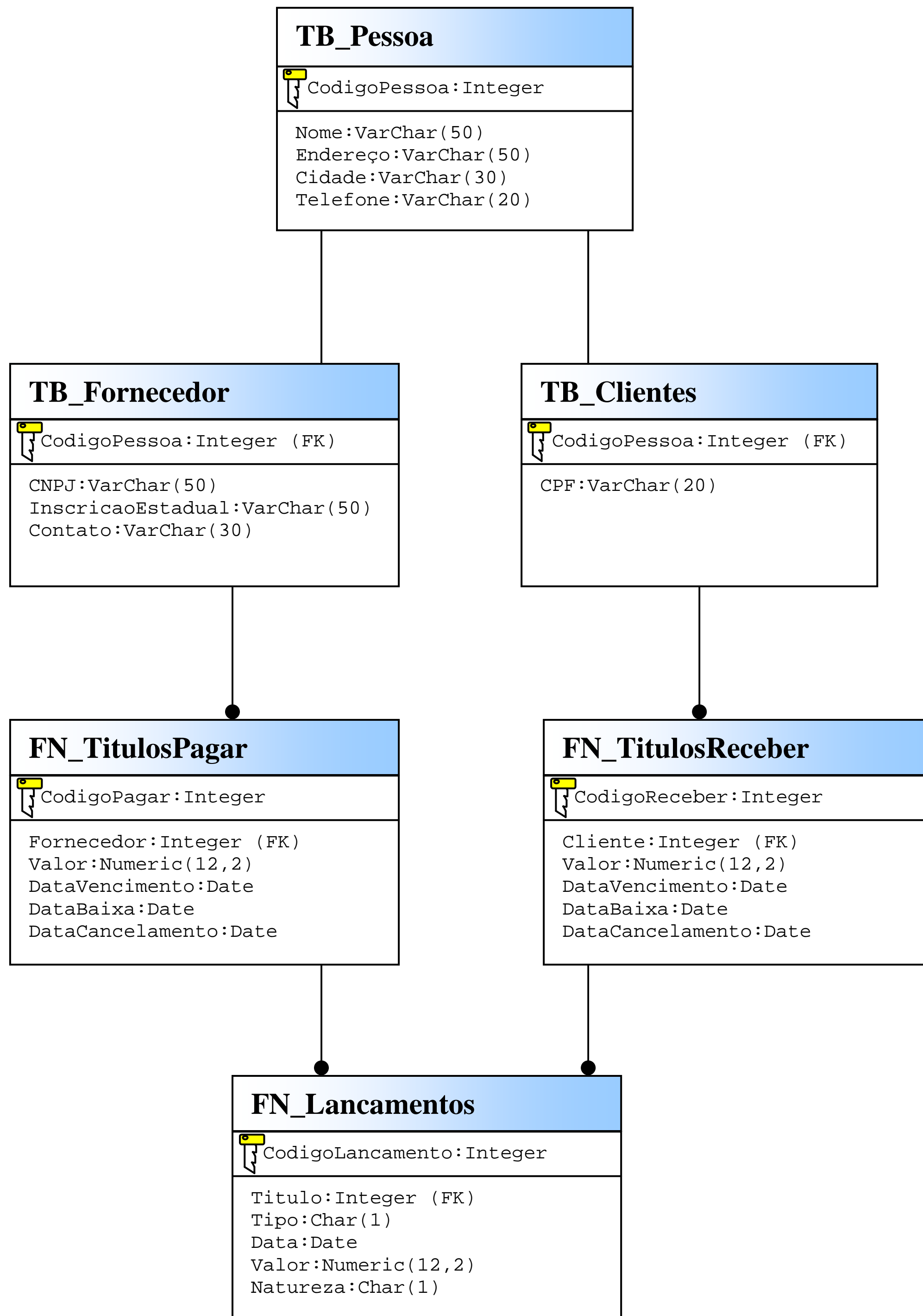

FIGURA 8 - Diagrama De Entidade E Relacionamento

6.2. A implementação

A implementação foi desenvolvida em Java, usando o ambiente de desenvolvimento NetBeans 5.0. Foi usado o banco de dados Firebird 1.5, sendo que este é acessado pelo JDBC e o pelo Hibernate. O script de criação do banco de dados está no anexo 1.

Na implementação foram criadas 4 classes para cada tabela do banco de dados. Por exemplo, para a classe pessoas foram criadas:

- pessoa.java
- pessoaDao.Java
- pessoaDaoJDBC.Java
- pessoaDaoHibernate.Java

As classes que implementam as operações de inclusão, alteração, exclusão e consulta de registros de uma tabela no banco de dados tem como parâmetro a classe básica, isto no exemplo acima quer dizer que na classe pessoaDaoJDBC e pessoaDaoHibernate tem como parâmetro a classe pessoa para os métodos incluir, altera, exclui.

Para exemplificar melhor a diferença do código implementado para JDBC e Hibernate , vamos dar uma olhada como foi implementado os códigos da classe pessoa para inserção, alteração, remoção e consulta. Pode-se notar que com JDBC é usado o padrão SQL para estas operações (insert, update, delete e select), recebendo a classe pessoa com parâmetro. Já com o Hibernate usamos o método *save()* para inclusão e alteração, método *delete()* para exclusão, passando como parâmetro a classe pessoa e finalmente método *createQuery()* para consulta.

```
01: private static final String INSERE_PESSOA = "insert into tb_pessoas(CODUN, NOME,
02: ENDERECO, CIDADE, TELEFONE) VALUES (?,?,?,?,?) " ;
03:
04:     public void incluir(Pessoa pessoa) {
05:         PreparedStatement stm = null;
06:         try {
07:             System.out.println("incluir");
08:             if (con == null) {
09:                 System.out.println("  con null");
10:             }
11:             stm = con.prepareStatement(INSERE_PESSOA);
12:             stm.setInt(1,pessoa.getCodun());
13:             stm.setString(2,pessoa.getNome());
14:             stm.setString(3,pessoa.getEndereco());
15:             stm.setString(4,pessoa.getCidade());
16:             stm.setString(5,pessoa.getTelefone());
17:             stm.executeUpdate();
18:         } catch (SQLException ex) {
19:             ex.printStackTrace();
20:         }
21:     }
```

Quadro 11 - Inclusão de registro com JDBC

```
01:     Public void incluir(Pessoa pessoa) {
02:         Session sessao = HibernateUtility.getSession();
03:         Transaction transaction = sessao.beginTransaction();
04:         sessão.save(pessoa);
05:         transaction.commit();
06:         sessão.close();
07:     }
```

Quadro 12 - Inclusão de registro com Hibernate

```
01:     private static final String ALTERA_PESSOA = “update tb_pessoas set NOME=?,
02: ENDERECO=?, CIDADE=?, TELEFONE=? Where CODUN  = ?  “
03:
04:     public void altera(Pessoa pessoa) {
05:         PreparedStatement stm = null;
06:         if (con == null) {
07:             System.out.println(“  con null”);
08:         } else {
09:             System.out.println(“  con “);
10:         }
11:         Try {
12:             stm = com.prepareStatement(ALTERA_PESSOA);
13:             stm.setString(1,pessoa.getNome());
14:             stm.setString(2,pessoa.getEndereco());
15:             stm.setString(3,pessoa.getCidade());
16:             stm.setString(4,pessoa.getTelefone());
17:             stm.setInt(5,pessoa.getCodun());
18:             stm.executeUpdate();
19:         } catch (SQLException ex) {
20:             ex.printStackTrace();
21:         }
22:     }
```

Quadro 13 - Alteração de registro com JDBC

```
01:     Public void altera(Pessoa pessoa) {
02:         Session sessao = HibernateUtility.getSession();
03:         Transaction transaction = sessao.beginTransaction();
04:         sessao.save(pessoa);
05:         transaction.commit();
06:         sessao.close();
07:     }
```

Quadro 14 - Alteração de registro com Hibernate

```
01: private static final String EXCLUI_PESSOA = "delete from tb_pessoas where CODUN=?";
02:
03:     Public int exclui(Pessoa pessoa) {
04:         PreparedStatement stm = null;
05:         try {
06:             stm = con.prepareStatement(EXCLUI_PESSOA);
07:             stm.setInt(1, pessoa.getCodun());
08:             stm.executeUpdate();
09:             return 1;
10:         } catch (SQLException ex) {
11:             ex.printStackTrace();
12:             return -1;
13:         }
14:     }
```

Quadro 15 - Exclusão de registro com JDBC

```
01:     Public int  exclui(Pessoa pessoa) {
02:         Session sessao = HibernateUtility.getSession();
03:         Transaction transaction = sessao.beginTransaction();
04:         sessao.delete(pessoa);
05:         Transaction.commit();
06:         sessao.close();
07:         return 1;
08:     }
```

Quadro 16 - Exclusão de registro com Hibernate

```
01:     Public Pessoa findPessoa(int Pcodigo) {
02:         String vSelect;
03:         Pessoa pessoa = new Pessoa();
04:         JdbcConnection jdbcConnection = new JdbcConnection();
05:         Connection con1 = jdbcConnection.getConnection();
06:         Statement stm = null;
07:         try {
08:             stm = con1.createStatement();
09:             if (Pcodigo == -1) {
10:                 vSelect = "select first 1 * from tb_pessoas ";
11:             } else {
12:                 vSelect = "select * from tb_pessoas where codun =
"+String.valueOf(Pcodigo);
13:             }
14:             ResultSet rs = stm.executeQuery(vSelect);
15:
16:             rs.next();
17:             pessoa.setCodun(rs.getInt("CODUN"));
18:             pessoa.setNome(rs.getString("NOME"));
19:             pessoa.setEndereco(rs.getString("ENDERECO"));
20:             pessoa.setCidade(rs.getString("CIDADE"));
21:             pessoa.setTelefone(rs.getString("TELEFONE"));
22:
23:             return pessoa;
24:
25:         } catch (SQLException ex) {
26:             ex.printStackTrace();
27:         }
28:         return null;
29:     }
```

Quadro 17 - Pesquisa de registro com JDBC

```
01:     Public Pessoa findPessoa(int Pcodigo) {
02:         String vSelect;
03:         Pessoa pessoa = new Pessoa();
04:
05:         try {
06:             Session sessao = HibernateUtility.getSession();
07:             Transaction transaction = sessao.beginTransaction();
08:             List<Pessoa> pessoas;
09:             if (Pcodigo == -1) {
10:                 Query select = sessao.createQuery("from Pessoa");
11:                 pessoas = select.list();
12:                 if (pessoas.isEmpty()) {
13:                     System.out.println("VAzio");
14:                 }
15:                 pessoa = pessoas.get(0);
16:             } else {
17:                Query select = sessao.createQuery("from Pessoa where codun=:pcodun")
18:                 .setInteger("pcodun", Pcodigo);
19:                 pessoas = select.list();
20:             sessao.close();
21:             return pessoa;
22:         } finally {
23:         }
```

Quadro 18 - Pesquisa de registro com Hibernate

As classes clientes, fornecedores, títulos a receber, títulos a pagar e lançamentos foram implementadas seguindo estas mesmas características.

6.3. Resultados

A implementação utilizando JDBC mostrou-se bastante repetitiva. Por exemplo, para transformar uma linha da tabela representada pela classe ResultSet, é necessário copiar o valor de cada campo utilizando o método ResultSet.getTipo(). Se houver uma alteração no modelo e um campo for removido ou adicionado, o código deve ser alterado. Ainda existe o problema de conversão de tipo, qualquer conversão entre tipos deve ser feita manualmente, o que implica em mais códigos para manutenção. No código apresentado não houve preocupação com gerenciamento de transação, pois esta atividade requer a utilização da API JTA. Uma pequena vantagem da utilização do JDBC é que não é necessário nenhum arquivo de configuração, mantendo o código centralizado.

O acesso ao banco de dados utilizando a ferramenta ORM Hibernate não apresenta os problemas acima. Ao contrário, possibilita a escrita de código mais enxuto, as alterações no modelo não implicam em alterações no código e, além disso, oferece gerenciamento de transação de forma simples e direta. O Hibernate necessita do arquivo de configuração e dos arquivos de mapeamento, o que pode gerar uma manutenção a mais. No entanto, a geração dos arquivos de mapeamento podem ser automatizadas utilizando anotações ou a ferramenta hbm2java, que faz parte do pacote Hibernate.

7. Conclusão

Neste trabalho apresentamos o problema semântico existente na utilização de banco de dados relacionais com linguagens orientada a objetos. Apresentamos também uma visão geral sobre JDBC, ORM e Hibernate.

Apresentamos um protótipo de um sistema um sistema de contas a pagar e receber e fizemos uma avaliação da implementação da camada de persistência utilizando JDBC e o *framework* Hibernate. O JDBC é muito trabalhoso e requer detalhes de baixo nível, como a escrita de cláusulas SQL, seu código é extenso, repetitivo e propenso a erros. Já o Hibernate é bastante fácil de usar, não sendo necessário preocupação com detalhes, oferece, ainda suporte a gerenciamento de transações de forma simples e direta com código pequeno e conciso. Existe a necessidade da criação dos arquivos de mapeamento, no entanto, este processo pode ser automatizado utilizando anotações ou ferramenta hbm2java.

8. Bibliografia

(Bauer e King, 2005) - Bauer, Christian; King, Gavin; “Hibernate Em Ação”, Ciência Moderna Ltda., 2005.

(Deitel, 2005) – Deitel, H. M.; Deitel P. J.; “Java com programar – 6º edição”, Pearson Prenteice Hall, 2005.

(Lemay e Candenhead, 1999) – Lemay, Laura; Cadenhead, Rogers; “Aprenda Em 21 Dias Java 2”, Editora Campus, 1999.

(Korth e Silberschatz, 1995) - Korth , Henry F.; Silberschatz, Abraham; “Sistema De Banco De Dados”, Makron Books.

(Kolb e Fields) - Kolb, Mark A.; Fields, Duane; “Desenvolvendo Na Web Com Java Server Pages”, Ciência moderna.

(HÜBNER) - HÜBNER, Jomi Fred. Acesso a Bancos de Dados em Java (JDBC). Disponível em <http://www.inf.furb.br/~jomi/java/pdf/jdbc.pdf >. Acesso em: 28/12/2006.

(Umeda) - Umeda, Nelson Naoki. Java – Integrando banco de dados via JDBC. Disponível em < http://www.pr.gov.br/batebyte/edicoes/1996/bb58/java.htm>. Acesso em: 28/12/2006.

(Qualid) - ... Pesquisado na Internet em 10 de janeiro de 2007. http://www.qualisoft .com.br/qsnews/edicao8.asp

(Jeveaux, 2003) – Jeveaux , Paulo César M. ; Pesquisado na Internet em 28 de dezembro de 2006. http://www.portaljava.com/home/modules.php?name=Content&pa=showpage&pid=5

(Ferreira, 2005) Ferreira, Cleiton. Arquitetura em camadas. Publicado em 12-02-2005. Portaljava. Pesquisado na internet em 10 de janeiro de 2007; http://www.portaljava.com.br/home/modules.php?name=Content&pa=showpage&pid=112.

(Freitas e Martins, 2006) – Freitas, Romualdo Rubens de; Martins, Jefferson Gustavo; Persistência em Java; SQL Magazine, edição 30, 2007.

9. Anexos

```
SET SQL DIALECT 3;

SET NAMES NONE;

CREATE DATABASE 'C:\ctmo\Java\Financeiro\FinanceiroJava.fdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 1024
DEFAULT CHARACTER SET NONE;

CREATE DOMAIN DMNCODUN AS INTEGER NOT NULL;
CREATE DOMAIN DMNVARCHAR50 AS VARCHAR(50);
CREATE DOMAIN DMNVARCHAR20 AS VARCHAR(20);
CREATE DOMAIN DMNVARCHAR30 AS VARCHAR(30);
CREATE DOMAIN DMNVALOR AS NUMERIC(15,2);
CREATE DOMAIN DMNTIPO AS CHAR(1);
CREATE DOMAIN DMNDATA AS DATE;
CREATE DOMAIN DMNINTEGER AS INTEGER;

CREATE TABLE TB_PESSOAS (
    CODIGOPESSOA DMNCODUN NOT NULL,
    NOME DMNVARCHAR50,
    ENDERECO DMNVARCHAR50,
    CIDADE DMNVARCHAR30,
    TELEFONE DMNVARCHAR20);

CREATE TABLE TB_CLIENTES (
    CODIDOCLIENTE DMNCODUN NOT NULL,
    CPF DMNVARCHAR20);

CREATE TABLE TB_FORNECEDOR (
    CODIGOFORNECEDOR DMNCODUN NOT NULL,
    CNPJ DMNVARCHAR20,
    INSCRICAOESTADUAL DMNVARCHAR20,
    CONTATO DMNVARCHAR50);

CREATE TABLE FN_TITULOSPAGAR (
    CODIGOPAGAR DMNCODUN NOT NULL,
    FORNECEDOR DMNINTEGER,
    VALOR DMNVALOR,
    DATAVENCIMENTO DMNDATA,
    DATABAIXA DMNDATA,
    DATACANCELAMENTO DMNDATA);

CREATE TABLE FN_TITULOSRECEBER (
    CODIGORECEBER DMNCODUN NOT NULL,
    CLIENTE DMNINTEGER,
    VALOR DMNVALOR,
    DATAVENCIMENTO DMNDATA,
    DATABAIXA DMNDATA,
    DATACANCELAMENTO DMNDATA);

CREATE TABLE FN_LANCAMENTOS (
    CODIGOLANCAMENTO DMNCODUN NOT NULL,
    TIPO DMNTIPO,
    DATA DMNDATE,
    VALOR DMNVALOR,
    NATUREZA DMNTIPO);

ALTER TABLE TB_PESSOAS ADD CONSTRAINT PK_TB_PESSOAS PRIMARY KEY (CODIGOPESSOA);

ALTER TABLE TB_CLIENTES ADD CONSTRAINT PK_TB_CLIENTES PRIMARY KEY (CODIDOCLIENTE);
ALTER TABLE TB_FORNECEDOR ADD CONSTRAINT PK_TB_FORNECEDOR PRIMARY KEY
(CODIGOFORNECEDOR);
ALTER TABLE TB_CLIENTES ADD CONSTRAINT FK_TB_CLIENTES_PESSOA FOREIGN KEY
(CODIDOCLIENTE) REFERENCES TB_PESSOAS (CODIGOPESSOA);
ALTER TABLE TB_FORNECEDOR ADD CONSTRAINT FK_TB_FORNECEDOR FOREIGN KEY
(CODIGOFORNECEDOR) REFERENCES TB_PESSOAS (CODIGOPESSOA);
```

Anexo 1– Script de criação de banco de dados em firebird.

Anexo 2 - Arquivo de configuração do Hibernate

```
01: <!DOCTYPE hibernate-configuration PUBLIC
02:     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
03:     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
04: E:\Arquivos de programas\Java\jdk1.5.0_06\jre\hibernate-3.2.1.ga\hibernate-
05: 3.2\src\org\hibernate\
06:
07: <hibernate-configuration>
08:     <session-factory >
09:          <property name="connection.driver_class">org.firebirdsql.jdbc.FBDriver
10:         </property>
11:         <property name="connection.url">
12:      jdbc:firebirdsql:localhost/3050:c:\ctmo\java\financeiro\financeirojava.fdb
13:         </property>
14:         <property name="connection.username">SYSDBA</property>
15:         <property name="connection.password">masterkey</property>
16:         <property name="dialect">org.hibernate.dialect.FirebirdDialect</property>
17:        <property name="show_sql">true</property>
18:
19:        <!-- Condiguração do c3p0 -->
20:        <property name="hibernate.c3p0.max_size">10</property>
21:        <property name="hibernate.c3p0.min_size">2</property>
22:        <property name="hibernate.c3p0.timeout">5000</property>
23:        <property name="hibernate.c3p0.max_statements">10</property>
24:        <property name="hibernate.c3p0.idle_test_period">3000</property>
25:        <property name="hibernate.c3p0.acquire_increment">2</property>
26:        <!-- Configurações de debug -->
27:        <property name="show_sql">true</property>
28:        <property name="hibernate.generate_statistics">true</property>
29:        <property name="hibernate.use_sql_comments">true</property>
30:
31:        <mapping resource="Pessoa.hbm.xml"/>
32:        <mapping resource="Cliente.hbm.xml"/>
33:        <mapping resource="Fornecedor.hbm.xml"/>
34:        <mapping resource="TituloReceber.hbm.xml"/>
35:        <mapping resource="TituloPagar.hbm.xml"/>
36:        <mapping resource="Lancamento.hbm.xml"/>
37:    </session-factory>
38: </hibernate-configuration>
```

Anexo 3 - Arquivos de mapeamento das classes

```
01: <?xml version="1.0"?>
02: <!DOCTYPE hibernate-mapping PUBLIC
03:     "-//Hibernate/Hibernate Mapping DTD//EN"
04:    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
05:
06: <hibernate-mapping>
07:     <class name="financeiro.modelo.Pessoa" table="PESSOAS">
08:
09:         <id name="codigo" type="integer">
10:            <generator class="assigned"/>
11:         </id>
12:         <property name="nome" type="string" length="50"/>
13:         <property name="endereco" type="string"/>
14:         <property name="telefone" column="telefone" type="string"/>
15:         <property name="cidade" column="cidade" type="string"/>
16:     </class>
17: </hibernate-mapping>
```

Arquivo de mapeamento de pessoas

```
01: <?xml version="1.0"?>
02: <!DOCTYPE hibernate-mapping PUBLIC
03: "-//Hibernate/Hibernate Mapping DTD//EN"
04:    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
05:
06: <hibernate-mapping>
07:  <joined-subclass name="financeiro.modelo.Cliente" extends=
08: "financeiro.modelo.Pessoa">
09:
10:        <key column="codigoCliente"/>
11:         <property name="cpf"/>
12:     </joined-subclass>
13: </hibernate-mapping>
```

Arquivo de mapeamento de clientes

```
01: <?xml version="1.0"?>
02: <!DOCTYPE hibernate-mapping PUBLIC
03:     "-//Hibernate/Hibernate Mapping DTD//EN"
04:     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
05:
06: <hibernate-mapping>
07:     <joined-subclass name="financeiro.modelo.Fornecedores" table="TB_FORNECEDOR">
08:         <key column="CodigoFornecedor"/>
09:
10:         <property name="CNPJ"/>
11:     <property name="INSCRICAOESTADUAL"/>
12:     <property name="CONTATO"/>
13: </joined-subclass>
14: bernate-mapping>
```

Arquivo de mapeamento de Fornecedores

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
03: "http://hibernate.sourceforge.net/hibernate-mapping.dtd">
04:
05: <hibernate-mapping>
06:     <class name="financeiro.modelo.TituloPagar" table="FN_TITULOSPAGAR">
07:
08:         <id name="CODIGO" column="CODIGO" type="integer">
09:             <generator class="assigned"/>
10:         </id>
11:
12:         <property name="VALOR" type="FLOAT"/>
13:         <property name="DATAVENCIMENTO" type="DATE"/>
14:         <property name="DATABAIXA" type="DATE"/>
15:         <property name="DATACANCELAMENTO" type="DATE"/>
16:         <set name=" FORNECEDOR" inverse="true">
17:              <key column="" />
18:              <one-to-manu class="financeiro.modelo.Fornecedores"/>
19:         </set>
20:     </class>
21: </hibernate-mapping>
```

Arquivo de mapeamento de Títulos a pagar

```
01: <?xml version="1.0" encoding="UTF-8"?>
01: <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
01: "http://hibernate.sourceforge.net/hibernate-mapping.dtd">
01:
01: <hibernate-mapping>
01:     <class name="financeiro.modelo.TituloReceber" table="FN_TITULOSRECEBER">
01:
01:         <id name="CODIGO" column="CODIGO" type="integer">
01:             <generator class="assigned"/>
10:         </id>
11:         <property name="VALOR" type="FLOAT"/>
12:         <property name="DATAVENCIMENTO" type="DATE"/>
13:         <property name="DATABAIXA" type="DATE"/>
14:         <property name="DATACANCELAMENTO" type="DATE"/>
15:         <set name="CLIENTE" inverse="true">
16:              <key column="" />
17:              <one-to-manu class="financeiro.modelo.Cliente"/>
18:         </set>
19:     </class>
20: </hibernate-mapping>
```

Arquivo de mapeamento de Títulos a Receber

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
03: "http://hibernate.sourceforge.net/hibernate-mapping.dtd">
04:
05: <hibernate-mapping>
06:     <class name="financeiro.modelo.Lancamento" table="FN_LANCAMENTOS">
07:
08:         <id name="CODIGO" column="CODIGO" type="integer">
09:             <generator class="assigned"/>
10:         </id>
11:         <property name="TIPO" type="STRING"/>
12:         <property name="DATA" type="DATE"/>
13:         <property name="VALOR" type="FLOAT"/>
14:         <property name="NATUREZA" type="STRING"/>
15:     </class>
16: </hibernate-mapping>
```

Arquivo de mapeamento de Lançamentos