**UNIVERSIDADE DO EXTREMO SUL CATARINENSE – UNESC**
**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**CRISTINA DA SILVA MATOS**

# COMPARAÇÃO ENTRE O USO DO JDBC E DO HIBERNATE PARA PERSISTÊNCIA DE DADOS

**CRICIÚMA, DEZEMBRO DE 2011.**

**CRISTINA DA SILVA MATOS**

# COMPARAÇÃO ENTRE O USO DO JDBC E DO HIBERNATE PARA PERSISTÊNCIA DE DADOS

Trabalho de Conclusão de Curso apresentado para obtenção do Grau de Bacharel em Ciência da Computação da Universidade do Extremo Sul Catarinense.

Orientador: Prof. MSc. Paulo João Martins
Co-orientador: Prof. Esp. Fabrício Giordani

**CRICIÚMA, DEZEMBRO DE 2011.**

**CRISTINA DA SILVA MATOS**

**Comparação entre o uso do JDBC e do Hibernate para Persistência de Dados**

Submetido ao corpo docente do Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense como um dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

**Profa. MSc. Ana Claudia Garcia Barbosa**
Coordenadora do Curso de Ciência da Computação

Banca Examinadora:

**Prof. MSc. Paulo João Martins (UNESC)**
Orientador

**Prof. Esp. Fabrício Giordani (UNESC)**

**Prof. MSc. Gustavo Bisognin (UNESC)**

*Aos meus pais, João Paulo Cardoso de Matos e Vanir da Silva Matos pela minha formação moral e ao meu noivo amado Tiago de Alcântara Esmeraldino.*

**AGRADECIMENTOS**

Agradeço primeiramente a Deus por seu amor incondicional, por colocar pessoas maravilhosas na minha vida e ter me capacitado desde o inicio.

Agradeço aos meus pais João Paulo e Vanir, pelo cuidado e amor por mim e pelo esforço dedicado a minha educação.

Agradeço ao meu noivo, Tiago, que sempre está ao meu lado, me apoiando e fazendo meus dias melhores.

Agradeço meus familiares, amigos e colegas que simplesmente enriquecem a minha vida, especialmente minha prima Rosiléia, que me deu um grande apoio no início do curso.

Aos professores do curso de Ciência da Computação, pelas experiências compartilhadas ao longo destes anos.

Ao Paulo (orientador) e Fabrício (co-orientador), pelo tempo e paciência dispensados, sugestões e contribuições durante a elaboração deste projeto.

À coordenação e secretaria do curso, pelo atendimento gentil e eficiente.

O Senhor é a minha força e o meu escudo; nele meu coração confia, e dele recebo ajuda. Meu coração exulta de alegria, e com o meu cântico lhe darei graças (Sl 28. 6-7).

# RESUMO

O modelo de programação Orientada a Objetos (OO) e o modelo de Bancos de Dados Relacional são amplamente utilizados, porém, esses modelos apresentam incompatibilidade entre seus dados, intitulada Impedância de Dados. Para que haja compatibilidade entre os eles é feito um processo de conversão chamado mapeamento objeto relacional, que pode ser feito de forma manual diretamente com o JDBC, ou por meio de uma ferramenta como o Hibernate. O presente trabalho teve como objetivo, comparar esses dois modos. Para isso foram feitas pesquisas sobre as principais características dos bancos de dados relacionais, sobre os princípios da programação OO. Também foi estudada a ferramenta Hibernate. Após a pesquisa foram construídos dois protótipos para a realização dos testes, um com uma aplicação orientada a objetos com bancos de dados relacionais utilizando o JDBC e outra utilizando o Hibernate. A comparação foi realizada por meio da avaliação dos protótipos desenvolvidos, foram considerados fatores como facilidade de codificação, legibilidade, tamanho do código-fonte e tempo de acesso. Os resultados demonstram que o ORM é uma maneira eficiente de resolver a impedância de dados, facilitando o desenvolvimento sem perder o desempenho.

**Palavras-Chave:** Banco de Dados; Linguagem Orientada a Objetos; Mapeamento Objeto Relacional; Hibernate.

## ABSTRACT

The model of Object Oriented programming (OO) model and Relational Databases are widely used, however, these models feature incompatibility between your data, called impedance data. For compatibility between them is made a conversion process called Object Relational Mapping, which can be done manually with JDBC directly, or through a tool like Hibernate. This study aimed to compare these two modes. For this, were done researches about main features of relational databases and about the principles of OO programming. Also was studied the Hibernate. After the search, two prototypes were built for testing, an application with an object-oriented relational databases using JDBC and the other using Hibernate. The comparison was performed by evaluating the prototypes were considered factors such as ease of coding, readability, size of the source code and access time. The results show that the ORM is an efficient way of solving the impedance data, facilitating the development without losing performance.


**Keywords**: Database; Object-Oriented Language; Object Relational Mapping; Hibernate.

**ILUSTRAÇÕES**

Figura 1 - Representação de um SGBD........................................................................21

Figura 2 . Abrangência do SQL..................................................................................26

Figura 3 – Mapeamento Objeto-Relacional..................................................................31

Figura 4 – Arquitetura mínima do Hibernate ..............................................................35

Figura 5 - Arquitetura Completa Hibernate..................................................................36

Figura 6 - Como o Hibernate funciona.........................................................................37

Figura 7 - Cliclo de vida do objeto mapeado................................................................39

Figura 8 – Caso de uso Cliente....................................................................................50

Figura 9 – Caso de uso Administrador.........................................................................50

Figura 10- Diagrama ER..............................................................................................51

Figura 11 – Diagrama de classes ................................................................................52

Figura 12 – Estrutura dos cenários desenvolvidos ......................................................53

Figura 13 – Classe Estado em uma implementação JDBC .........................................55

Figura 14- Classe “conexão.java” do cenário JDBC....................................................55

Figura 15 – Código do arquivo de configuração “persistence.xml”...........................57

Figura 16 – Classe EntityManagerUtil ........................................................................57

Figura 17 - Bibliotecas utilizadas no protótipo ORM ..................................................58

Figura 18 – Exemplo de mapeamento utilizando *annotations* ...................................59

Figura 19 – Salvando uma classe no cenário JDBC....................................................60

Figura 20 – Atualizando uma classe no cenário JDBC ...............................................61

Figura 21 – Salvando e Atualizando uma classe no cenário ORM .............................61

Figura 22 – Recuperando uma lista de objeto no cenário JDBC..................................62

Figura 23 – Recuperando uma lista de objetos no cenário ORM.................................63

Figura 24 - Buscando objeto específico no cenário JDBC....................................................64

Figura 25 - Buscando objeto específico no cenário ORM ....................................................64

Figura 26 – Excluindo uma trupla no cenário JDBC..........................................................65

Figura 27 – Excluindo uma tupla no cenário ORM.............................................................65

Figura 28 - Persistindo objetos herdados no cenário JDBC ..................................................66

Figura 29 – Persistendo objetos herdados no cenário ORM ..................................................67

Figura 30 - Usando funções de agregação no cenário JDBC ................................................68

Figura 31 - Usando funções de agregação no cenário ORM.................................................68

Figura 32 – Teste do tempo para adicionar objetos no SGBD ...............................................69

Figura 33 – Teste do tempo gasto para editar os objetos. .......................................................70

Figura 34 – Teste de tempo ao buscar lista de dados. ............................................................71

Figura 35 – Teste de tempo para buscas dados utilizando a filtro...........................................72

Figura 36 - Teste de tempo para excluir objetos ....................................................................73

Figura 37 - Media geral dos métodos testados .......................................................................74

**LISTA DE SIGLAS**

| | |
|---|---|
| ACID | Atomicidade, Consistência, Isolamento, Durabilidade |
| ANSI | *American National Standart Institute* |
| API | *Application Programming Interface* |
| BD | Banco de Dados |
| CRUD | *Create, Read, Update, Delete* |
| DDL | *Dynamic-Link Library* |
| DML | *Data Manipulation Language* |
| JDBC | *Java Database Connectivity* |
| JTA | *Java Transaction API* |
| ODBC | *Open Data Base Connectivity* |
| OO | Orientado a Objetos |
| ORM | *Object-Relational Mapping* |
| POO | Programação Orientada a Objetos |
| SGBDR | Sistema Gerenciador de Banco de Dados Relacional |
| SQL | *Structured Query Language* |
| XML | *eXtensible Markup Language* |

**SUMÁRIO**

**1 INTRODUÇÃO ........ 15**

1.1 OBJETIVO ........ 17

1.2 OBJETIVOS ESPECÍFICOS ........ 17

1.3 JUSTIFICATIVA ........ 18

1.4 ESTRUTURA DO TRABALHO ........ 19

**2 BANCO DE DADOS ........ 20**

2.1 PERSISTÊNCIA DE DADOS ........ 20

2.3 SISTEMA GERENCIADOR DE BANCO DE DADOS ........ 21

2.4 BANCO DE DADOS RELACIONAL ........ 23

**2.4.1 Tabela ........ 24**

**24.2 Coluna ........ 24**

**2.4.3 Índices e Chaves ........ 24**

**2.4.5 Consulta e Manipulação de Dados ........ 25**

**3 PARADIGMA DE PROGRAMAÇÃO ORIENTADA A OBJETOS ........ 27**

3.1 CLASSE ........ 27

3.2 OBJETOS ........ 28

3.3 ENCAPSULAMENTO ........ 28

3.4 MÉTODO ........ 29

3.5 HERANÇA ........ 29

3.6 POLIMORFISMO ........ 30

**4 MAPEAMENTO OBJETO RELACIONAL ........................................................ 30**

4.1 MAPEAMENTO OBJETO RELACIONAL VIA JDBC ........................................ 31

4.2 MAPEAMENTO OBJETO RELACIONAL VIA FRAMEWORK ........................ 32

**5 HIBERNATE ........................................................................................ 33**

5.1 ARQUITETURA ........................................................................................ 34

**5.1.1 APIs Básicas ........................................................................................ 36**

5.2 COMO O HIBERNATE FUNCIONA ............................................................ 37

**5.2.1 Hibernate Session .................................................................................. 38**

**5.2.2 *Session Factory* .................................................................................... 38**

**5.2.3 Ciclo de Vida de um Objeto Mapeado ...................................................... 39**

5.2.3.1 Estado Transiente .................................................................................. 40

5.2.3.2 Estado Persistente .................................................................................. 40

5.2.3.3 Estado Desanexado ................................................................................ 41

5.3 CONFIGURAÇÕES .................................................................................... 41

**6 TRABALHOS CORRELATOS ....................................................................... 42**

6.1 ANÁLISE E AVALIAÇÃO DO FRAMEWORK HIBERNATE EM UMA APLICAÇÃO CLIENTE/SERVIDOR ........................................................................................ 42

6.2 IMPEDÂNCIA DE DADOS EM BANCO DE DADOS ........................................ 42

6.3 FRAMEWORK’S PARA MAPEO OBJETO RELACIONAL: UM ANÁLISIS COMPARATIVO ........................................................................................ 43

**7 COMPARAÇÃO ENTRE JDBC E HIBERNATE .............................................. 45**

7.1 METODOLOGIA ........................................................................................ 45

7.2 PROTÓTIPO DE APLICAÇÃO ........................................................................ 45

**7.2.1 Concepção e Modelagem do Protótipo ........................................................ 46**

7.2.1.1 Funcionamento do sistema e levantamento dos requisitos. ................................ 47

7.1.1.2 Análise e levantamento de requisitos ........................................................ 48

7.2.1.3 Casos de uso ........................................................................ 49

7.2.1.4 Diagramas de casos de uso ........................................................................ 49

7.2.1.5 Modelagem do banco de dados ........................................................................ 51

7.2.1.6 Diagrama de classes do cenário ........................................................................ 52

**7.2.2 Desenvolvimento do Protótipo ........................................................................ 53**

7.2.2.1 Visão geral do protótipo do cenário JDBC ........................................................ 54

7.2.2.1.2 Bibliotecas utilizadas ........................................................................ 56

7.2.2.1.3 Mapeamento ........................................................................ 56

7.2.2.1Visão Geral do Cenário ORM ........................................................................ 56

7.2.2.2.2 Bibliotecas Utilizadas ........................................................................ 57

7.2.2.2.3 Mapeamento ........................................................................ 58

**7.2.3 Comparação entre as classes persistentes ........................................................ 59**

7.2.3.1 Salvando e Editando os Dados ........................................................................ 60

7.2.3.2 Recuperando Lista de Dados ........................................................................ 62

7.2.3.3 Buscando um Dado pela Chave ........................................................................ 63

7.2.3.4 Deletando uma Linha ........................................................................ 65

7.2.3.6 Persistindo Herança ........................................................................ 65

7.2.3.7 Consultas com Funções de Agregação ........................................................ 67

7.2 Análise do TEMPO ........................................................................................ 69

**CONCLUSÃO............................................................................................................ 75**

**REFERÊNCIAS .......................................................................................................... 77**

**Apêndice A – SCRIPT DE CRIAÇÃO DAS TABELAS NA bASE DE DADOS............. 80**

# 1 INTRODUÇÃO

Com o uso da informática para as mais diversas finalidades, a era da informação fica evidente. As informações são consideradas mais importantes, e ao mesmo tempo mais fáceis e baratas de se obter. Nesse contexto os bancos de dados se tornaram importantes pilares para o armazenamento dessas informações (SILVA, 2007).

Diante dessa realidade surgem vários métodos de implantação de persistência de dados, sendo, os mais empregados atualmente os Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDR's) (LOPES, 2007). A abordagem relacional está baseada no princípio de que as informações em uma base de dados podem ser consideradas como relações matemáticas representadas de maneira uniforme, por meio do uso de tabelas bidimensionais (MACHADO; ABREU, 2004).

Para facilitar a integração entre as linguagens de programação e os bancos de dados, sugiram camadas de persistência que realizam a comunicação por meio de *drivers*, tornando-os possíveis de serem aplicados a qualquer linguagem de programação (LOPES, 2007).

Contudo, existem diferenças significativas do paradigma de programação OO para o modelo de banco de dados relacional. Para que seja possível obter os benefícios de ambos, é necessária uma compatibilidade entre esses modelos. Uma das propostas para diminuir essas diferenças é o Mapeamento Objeto Relacional (ORM), que pretende diminuir o tempo de desenvolvimento, reduzir o custo geral do sistema e proporcionar um código de fácil manutenção. Essa técnica consiste na criação de uma camada na aplicação que mapeia objetos em tuplas, no banco de dados relacional, tornando transparente a persistência dos objetos (HIBERNATE, 2010).

Segundo Manente (2007) a ferramenta ORM mais utilizada é o Hibernate, que é um framework responsável pela camada de mapeamento objeto relacional usada para armazenar objetos Java em uma base de dados relacional. Por meio dessa camada, ele permite o desenvolvimento de classes orientada a objetos persistentes, permitindo o uso de associação, herança, polimorfismo, composição e coleções.

Dada à importância da persistência de objetos e da grande aceitação e utilização de SGBDR, propõe-se uma comparação entre um sistema desenvolvido em linguagem orientada a objetos que utilize o JDBC para acessar o banco de dados, comparando a mesma aplicação, porém, com o uso de uma ferramenta ORM. Para essa avaliação serão definidas métricas como tempo de acesso, desempenho, e suporte a vários tipos de representação de herança.

Esse estudo se faz necessário para entender como esses modos de acesso afetam o desempenho, o tempo de implementação e a facilidade de manutenção da aplicação, dado que alguns desenvolvedores optam pelo método de acesso direto ao banco, considerando que o desempenho será muito superior, e outros, optam por ferramentas ORM sem considerarem se esta escolha acarretará perda de desempenho do software. Além disso, segundo Santos e Martins (2008) existem alguns empecilhos enfrentados pelo Mapeamento Objeto Relacional, como a dificuldade em fazer algumas consultas mais complexas e a perda de desempenho.

Essa pesquisa objetiva a realização de uma análise das metodologias de persistência de dados, SGBD nativo e ORM, para definir em que momento o uso do Hibernate oferece vantagens e quando ele adiciona complexidade ao projeto. Serão considerando fatores como, operações CRUD, e alguns dos principais recursos dos bancos de dados como chaves primárias e estrangeiras.

Com essa comparação pretende-se demarcar quais a limitações as vantagens e os recursos oferecidos por cada um dos métodos utilizados na persistência de dados em cenários

orientados a objeto na linguagem de programação Java. Facilitando para que haja uma escolha dos recursos de persistência de uma maneira consciente que satisfaça as necessidades do projeto à qual está sendo utilizado.

## 1.1 OBJETIVO

Comparar uma aplicação desenvolvida por meio de uma linguagem orientada a objetos e banco de dados relacional com o uso do SGBD nativo versus o uso do framework de Mapeamento Objeto Relacional Hibernate.

## 1.2 OBJETIVOS ESPECÍFICOS

A fim de atingir o objetivo geral dessa pesquisa foram definidos os seguintes objetivos específicos:

a) descrever e utilizar os conceitos de banco de dados relacional;

b) documentar e aplicar os conceitos de Mapeamento Objeto Relacional;

c) descrever as características de persistência de objetos;

d) desenvolver uma aplicação para comparar os métodos de acesso ao banco de dados;

e) comparação entre as aplicações desenvolvidas descrevendo suas vantagens baseado em métricas.

## 1.3 JUSTIFICATIVA

A necessidade de armazenar o crescente volume de dados demandou o desenvolvimento dos Banco de Dados (BD), que são gerenciados por SGBD's, de forma que este acessa, manipula e organiza os dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Entre os bancos de dados atuais está o relacional, que possui lugar de destaque e é amplamente utilizado por possuir confiabilidade e robustez (SILBERSCHATZ; KORTH; SUDARSHAN, 2006). Entretanto este modelo não apresenta características para armazenar objetos.

As ferramentas ORM propõem facilitar a persistência de objetos. Resolvendo problemas como, impedância de dados, representação de tipos de dados criados pelo usuário, herança e modelos mais complexos, além de proporcionar interdependência de banco de dados (PLÁCIDO, 2008). Esse framework diminui o tempo de desenvolvimento, pois com o uso do ORM o programador não perde tempo com operações de persistência simples e repetitivas, por exemplo, as operações CRUD; reduz o custo do sistema, por proporcionar uma implementação mais rápida; permite um código de fácil manutenção por ser uma ferramenta automática que segue padrões de codificação.

Existem bancos de dados orientados a objetos que prometem um melhor desempenho e maior facilidade de uso em longo prazo. No entanto, os SGBD's relacionais por estarem maduros são mais utilizados, conjuntamente com linguagens de programação orientadas a objetos, percebe-se então a importância de comparar os principais aspectos da persistência de objetos nos modelos ORM e JDBC enfatizando o custo benefício da escolha entre um e outro (RUMBAUGH, 1994). Essa comparação se faz necessária, pois apesar do

JDBC e do Hibernate possuírem arquitetura diferente, ambos são usados com o mesmo objetivo, o de salvar os dados em uma linguagem OO em um uma base de dados relacional.

Conforme Machado e Abreu (2004) os dados são um dos recursos mais importantes das corporações. Levando em conta essa afirmação, e a importância de persistir objetos e o tempo de desenvolvimento que é utilizado para isso, este estudo poderá auxiliar na tomada de decisão para os desenvolvedores e arquitetos de software no que diz respeito a decidir a forma de persistência mais adequada para o tipo de aplicação que predende-se desenvolver.

## 1.4 ESTRUTURA DO TRABALHO

O presente trabalho contem sete capítulos, sendo o primeiro composto pela introdução, objetivos e justificativa.

A pesquisa realizada sobre banco de dados, SGBD's e Banco de Dados Relacional e seus principais elementos é abordada no Capítulo 2.

No Capítulo 3 é apresenta o paradigma de programação orientada a objetos.

O Capítulo 4 conceitua mapeamento objeto relacional e mostra a diferença entre mapeamento via JDBC e via framework.

Capítulo 5 apresenta a ferramenta Hibernate, com sua arquitetura, forma de funcionamento e principais APIs.

Estão descritos no Capítulo 6 alguns trabalhos correlatos a nível regional e mundial que contem um teor semelhante a este trabalho.

Por fim, o Capítulo 7 apresenta a comparação entre o JDBC e Hibernate, utilizando como base os protótipos desenvolvidos.

# 2 BANCO DE DADOS

Banco de Dados é um sistema computadorizado de armazenamento de registros inter-relacionados, cujo propósito é armazenar informações de um domínio específico, e por meio de um conjunto de programas permitirem ao usuário buscar e atualizar essas informações quando for solicitado (DATE, 2000; SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Uma das principais finalidades dos BD é oferecer uma visão abstrata ao usuário. “Isto é, o sistema oculta certos detalhes de como os dados são armazenados e mantidos”. Silberschatz (2006, p. 20). Esses dados representam o mundo real e devem ser mantidos para atender os requisitos da empresa (OLIVEIRA, 2000).

## 2.1 PERSISTÊNCIA DE DADOS

A persistência de dados consiste na característica dos dados utilizados em uma aplicação serem duráveis. Eles ficam armazenados em um meio estável, que possibilite a restauração posterior. Os sistemas primordiais armazenavam dados em arquivos texto puros, separavam-se os campos com algum caractere especial (como por exemplo, uma vírgula), e demandava-se um novo registro com o caractere especial de quebra de linha. A gravação era feita em First In, First Out (FIFO), ou seja, os dados eram recuperados na mesma ordem em que eram gravados (SANTOS; MARTINS, 2007; SILVA, 2007).

Com o aumento da necessidade e da quantidade dos dados a serem armazenados esses meios tornavam-se inadequados, pois para consultar informações específicas eram necessárias rotinas complexas, inúmeras operações de I/O, conversões de formatos, além da

falta de segurança dos arquivos e da dificuldade em evitar inconsistência nos dados. Em busca da resolução desses tipos de problemas os primeiros bancos de dados foram criados (SILVA, 2007).

## 2.3 SISTEMA GERENCIADOR DE BANCO DE DADOS

O banco de dados refere-se apenas aos dados, o sistema responsável por controlar os dados é denominado Sistema Gerenciador de Banco de Dados (SGBD), Figura 1. Conforme Silberschatz (1999, p. 01), um SGBD é “constituído por um conjunto de dados associados e um conjunto de programas que fornece o acesso a esses dados”, ou seja, uma coleção de programas que permite criar estruturas, manter dados e gerenciar as transações efetuadas, além de permitir a extração das informações de maneira rápida e segura (SILVA, 2007). O principal objetivo do SGBD é oferecer um ambiente para o armazenamento e recuperação de informações do banco de dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2006; PLÁCIDO, 2008).

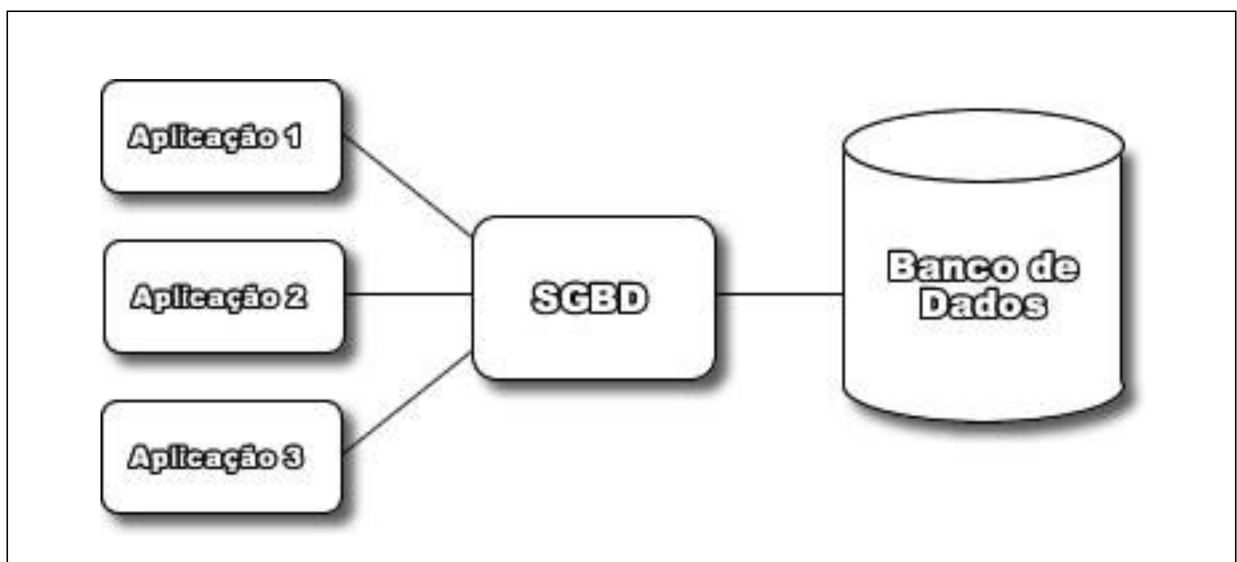


Figura 1 - Representação de um SGBD

Segundo Oliveira (2002) apud Silva (2007) um SGBD deve apresentar algumas característica a respeito do armazenamento e manutenção dos dados, tais como:

a) **controle de redundância**: os dados devem possuir o mínimo de redundância;
b) **compartilhamento de dados**: as informações devem estar disponíveis para qualquer número de usuários de forma paralela e segura;
c) **controle de acesso**: é necessário saber quem pode realizar determinada função nos dados;
d) **esquematização**: os relacionamentos devem estar armazenados no banco de dados para garantir a facilidade de entendimento e aplicação do modelo;
e) **backup**: deve haver rotinas específicas para realizar a cópia de segurança dos dados armazenados.

O gerenciamento das informações implica a definição das estruturas de armazenamento e a definição dos mecanismos para a manipulação dessas informações armazenadas contra eventuais problemas no sistema. Para manter a segurança das informações os SGBD's têm como requisito as propriedades: Atomicidade, Consistência, Isolamento, Durabilidade (ACID). Onde, Atomicidade é a propriedade que define que, caso uma parte de uma transação falhe, toda a transação deve ser cancelada. A Consistência garante que os dados não estejam equivocados. O Isolamento define que as transações devem operar de forma isolada sem interferirem entre si. E a Durabilidade refere-se ao próprio conceito de persistência, a garantia da durabilidade é feita por meio de *logs* de transação e *backup* dos dados.

Os SGBD's utilizam diferentes formas de representação ou modelagem de dados para descrever a estrutura das informações contidas em seus bancos de dados. Atualmente alguns dos modelos existentes são: modelo hierárquico, modelo em redes, modelo relacional e o modelo orientado a objetos. Entre esses modelos, o mais utilizado é o modelo relacional.

## 2.4 BANCO DE DADOS RELACIONAL

Um banco de dados relacional pode ser considerado como um conjunto de dados vistos segundo um conjunto de tabelas. Sendo que as tabelas possuem vários atributos, cada um com um nome único que representam os dados e as relações entre eles. Uma linha em uma tabela representa um relacionamento entre um conjunto de valores. (MACHADO; ABREU, 2004; SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Os bancos de dados relacionais baseiam-se no princípio de que os dados podem ser considerados como relações matemáticas e que estão representados de maneira uniforme, com o uso de tabelas bidimensionais. “O conceito principal vem da teoria dos conjuntos (álgebra relacional) atrelado à idéia de que não é relevante ao usuário saber onde os dados estão nem como os dados estão (Transparência)” (Machado; Abreu, 2004, p. 182).

Conforme Silva (2007) o modelo relacional atende três aspectos:

a) **estruturas**: são objetos que armazenam ou acessam os dados, tais como tabelas, visões e índices;
b) **regras de Integridade**: são responsáveis por proteger os dados e as estruturas de um banco de dados, garantindo a segurança da estrutura e dos dados. Para isso essas regras delimitam quais são as operações permitidas nos dados e nas estruturas de um banco de dados.
c) **operações**: são ações que permitem aos usuários manipular os dados e as estruturas de um banco de dados. Elas devem obedecer a um conjunto predefinido de regras de integridade.

### 2.4.1 Tabela

As tabelas de um banco de dados relacional consistem em um conjunto de campos (linhas) e registros (colunas). Onde cada linha da tabela representa uma única instância de uma entidade ou um relacionamento entre entidades. Cada tabela tem um nome único e contém zero ou mais registros (VIEIRA, 2003).

Se fizermos uma relação com a orientação a objetos, podemos deduzir que os campos são como os atributos dos objetos. O preenchimento dos campos acarreta a verificação das regras sobre cada um destes, como por exemplo, verificação do tipo de dados (numérico, alfanumérico, data, entre outros) e verificação da integridade referencial[1] (VIEIRA, 2003).

### 24.2 Coluna

Cada atributo ou coluna tem um nome e refere-se a uma característica da entidade representada. Os atributos contêm informações básicas que qualificam uma entidade e descrevem seus elementos ou características.

### 2.4.3 Índices e Chaves

São campos usados para pesquisa em uma tabela, na prática são criados independentemente dos dados. Fazem parte da lógica, portanto, podem ser excluídos e criados a qualquer momento. Tem como objetivo reduzir o tempo de acesso aos registros. Os índices podem ser compostos por mais de um campo da tabela. Apesar do uso de índices reduzirem o

---

[1] Garante a não corrupção dos dados

tempo de acesso, ele aumenta o tamanho de arquivo, por isso é necessário que o uso ocorra de forma controlada.

As chaves são um conjunto de um ou mais atributos que identificam linhas e estabelecem relações entre linhas e tabelas de um banco de dados relacional (SILVA, 2007). Podem ser usadas para identificação ou para implementação de relacionamentos, sendo que existem dois tipos de chave, a primária e a estrangeira, a seguir serão descritas as principais características desses tipos de chave (SILVA, 2007).

Uma chave primária é um tipo especial de índice, usada a fim de identificar cada registro como único, sendo assim, nunca deve se repetir em uma mesma tabela. As chaves primárias também podem ser criadas com mais de um campo, contanto que a combinação dos dados nunca seja repetida na mesma tabela (VIEIRA, 2003).

Uma chave estrangeira é representada por uma ou mais colunas. Formada pelo relacionamento com a chave primária de outra tabela. Diferentemente da chave primária pode ocorrer repetidas vezes (SILVA, 2007).

### 2.4.5 Consulta e Manipulação de Dados

Para a consulta e manipulação no banco de dados relacional, são usadas linguagens de definição e manipulação de dados. A linguagem de Definição de Dados (DDL) é usada na definição da estrutura e organização dos dados armazenados. Já a Linguagem de Manipulação dos Dados (DML) permite ao usuário ou aplicação, a inclusão, remoção, seleção ou atualização dos dados armazenados (MACHADO; ABREU, 2004).

Os dados nos bancos de dados relacionais são geralmente acessados e manipulados pela linguagem *Structured Query Language* (SQL), que atende os requisitos de DDL e DML. Foi criada em 1974, nos laboratórios de pesquisa da IBM, e depois com o seu

sucesso e grande utilização nos SGBDs, em 1982 o American National Standart Institute (ANSI) tornou o SQL o padrão oficial de linguagem em ambiente relacional (MACHADO; ABREU, 2004).

É uma linguagem projetada para gerenciar dados, originalmente baseada em álgebra relacional. Seu escopo inclui inserção de dados, consulta, atualização, exclusão, esquema de criação e modificação de dados e controle de acesso. A Figura 2 demonstra algumas aplicações possíveis com o uso do SQL.

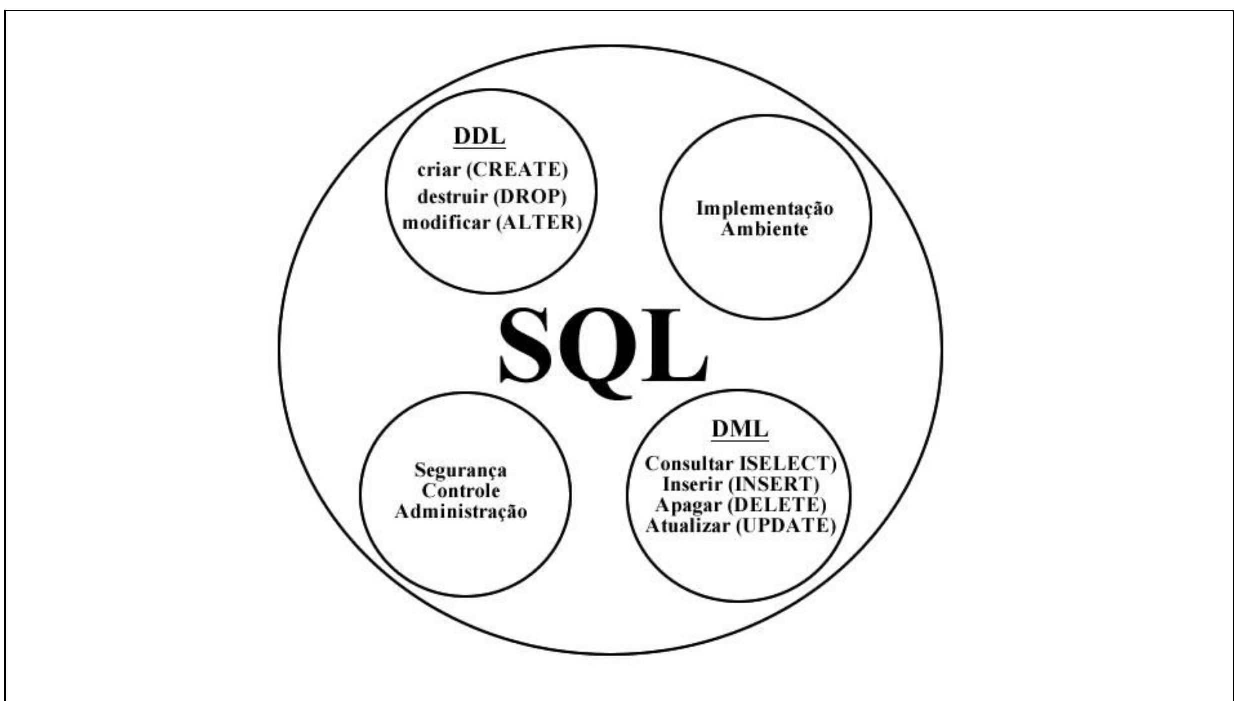


Figura 2 . Abrangência do SQL
Fonte: Adaptado de MACHADO, F.; ABREU, M. (2004, p. 198)

Os bancos de dados são o modo mais adequado de salvar os dados de uma aplicação que então na memória. O modelo mais utilizado atualmente é o relacional que, além de maduro oferece muitas opções de gerenciadores. Esses modelos são utilizados no desenvolvimento dos sistemas juntamente com linguagem orientada a objetos.

# 3 PARADIGMA DE PROGRAMAÇÃO ORIENTADA A OBJETOS

O conceito de Programação Orientada a Objetos (POO) tem suas origens no final dos anos 60 com a chegada da linguagem SIMULA, que introduziu alguns conceitos desse paradigma, tais como classes, subclasses e objetos. Já na década de 80, surgiram as primeiras linguagens de programação orientada a objetos comercialmente viáveis, entre elas Smalltalk e C++ (NASCIMENTO, 2003). Segundo Vieira (2003) a POO permitiu uma maior abstração de sistema por parte de projetista e facilitou a “tradução” do sistema real em um sistema computacional. (SANTOS; MARTINS, 2007).

Enquanto na programação procedural toda a lógica da aplicação é baseada em chamadas seqüenciais de funções que agem sobre os dados. A modelagem OO parte de premissa que todo o objeto da vida real possui atributos e comportamentos. As aplicações são constituídas por objetos que representam entidades do mundo real e que cooperam entre si para realizar as tarefas solicitadas. Onde cada objeto processa seus próprios dados e devolve o resultado por de troca de mensagens (MANENTE, 2007; VIEIRA, 2003).

Por ser um paradigma diferente, as linguagens OO apresentam novos conceitos em relação às linguagens relacionais, tais como classes, objetos, encapsulamento, herança e polimorfismo. Iremos descrever sobre cada um desses conceitos.

## 3.1 CLASSE

Representa o conceito geral de algo no mundo real, elas especificam os objetos definindo um conjunto de características (propriedades, atributos e campos) e comportamentos (métodos, ações e habilidades) que um conjunto de objetos que pertence a ela pode assumir. Uma classe pode ter muitos objetos (SANTOS, 2003).

## 3.2 OBJETOS

Os objetos são as estruturas fundamentais da abordagem OO, eles combinam a estrutura (estado, valor) e o comportamento (operações) dos dados em uma única entidade. No mundo real um objeto pode ser definido por algo tangível, que pode ser entendido e aprendido. Em programação objetos são "módulos que contêm dados e instruções para operar sobre estes dados" (NASCIMENTO, 2003). Os objetos são criados a partir de classes, sendo que cada um representa uma instância individual da classe e possuem um identificador único (SANTOS, 2003; HAHN, 2009). O valor das propriedades em um objeto definem seu estado e os métodos que podem ser executados especificam o comportamento do objeto (NASCIMENTO, 2003; SANTOS, 2003).

## 3.3 ENCAPSULAMENTO

Encapsulamento está diretamente relacionado com abstração, sendo que abstrair significa considerar apenas os aspectos essenciais a uma entidade, e ignorar propriedades menos importantes. Em POO os dados são considerados privados e o acesso a estes é feito por meio de métodos declarados como públicos em suas classes. Isto permite que a alteração da implementação de um objeto não afete outras partes do sistema (NASCIMENTO, 2003; HAHN, 2009).

## 3.4 MÉTODO

Métodos definem o comportamento dos objetos, e assim como as funções da programação procedural possuem assinatura, parâmetros e tipos de retorno.

Os métodos são usados para a comunicação entre os objetos, podendo ser utilizado para fazer uma alteração ou para obter uma informação sobre um objeto (SANTOS; MARTINS, 2007; SANTOS, 2003).

## 3.5 HERANÇA

A herança é um mecanismo de especialização de classes, que permite a uma determinada classe possuir características particulares e ainda herdar as características de outra classe (HAHN, 2009). Para isso a linguagem OO reaproveita o código e dados de uma classe (superclasse) em outras classes (subclasses) (VIEIRA, 2003). Assim, um objeto do tipo da subclasse também é um objetivo do tipo da superclasse. Além disso, a subclasse pode adicionar novos atributos e métodos a sua definição fora aqueles herdados pela superclasse (SANTOS; MARTINS, 2007).

Com a orientação a objetos a subclasse criada herda todas as características de sua classe mãe. As propriedades de uma classe mãe não precisam ser repetidas na classe filha, pois esta herda as propriedades automaticamente, proporcionando uma das principais características da orientação a objetos, a reutilização (NASCIMENTO, 2003).

A criação de subclasses a partir de uma classe mãe é denominada especialização, sendo que o caminho inverso é denominado generalização (VIEIRA, 2003).

A definição de herança é transitiva, ou seja, se uma classe C1 é superclasse de outra classe C2 que por sua vez é superclasse de outra classe C3, C3 também é do tipo C2 e

C1 e C2 também é do tipo C1. Assim a herança possibilita obtermos uma hierarquia de classes (SANTOS; MARTINS, 2007).

Existem duas formas de herança, a simples, que herda características de uma única classe e a herança múltipla, onde uma classe herda características de mais de uma classe. No caso da herança múltipla, nem todas as linguagem OO oferecem suporte. Por exemplo em Java não há herança múltipla, e em C++ sim (HAHN, 2009).

### 3.6 POLIMORFISMO

Segundo Ricarte (2001, p.6) “polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos”. A decisão de qual implementação será usada é feita em tempo de execução, conforme o tipo de objeto (tipo da superclasse ou da subclasse) que está invocando o método (HAHN, 2009; RICARTE, 2001).

## 4 MAPEAMENTO OBJETO RELACIONAL

O mapeamento objeto relacional, mais conhecido como O*bject Relational Mapping* (ORM) tem como objetivo “ligar” ambientes que utilizam de um lado o paradigma de orientação a objeto, e do outro lado o modelo relacional, a Figura 3 ilustra essa situação. Podendo ser entendido como o ato de conversão de objetos armazenado em memória em dados relacionais, esses dados devem ficar de forma que os dados possam ser guardados em uma linha de uma tabela de um banco de dados. O processo inverso também é válido (COELHO; SARTORELLI, 2004; SANTOS; MARTINS, 2007).

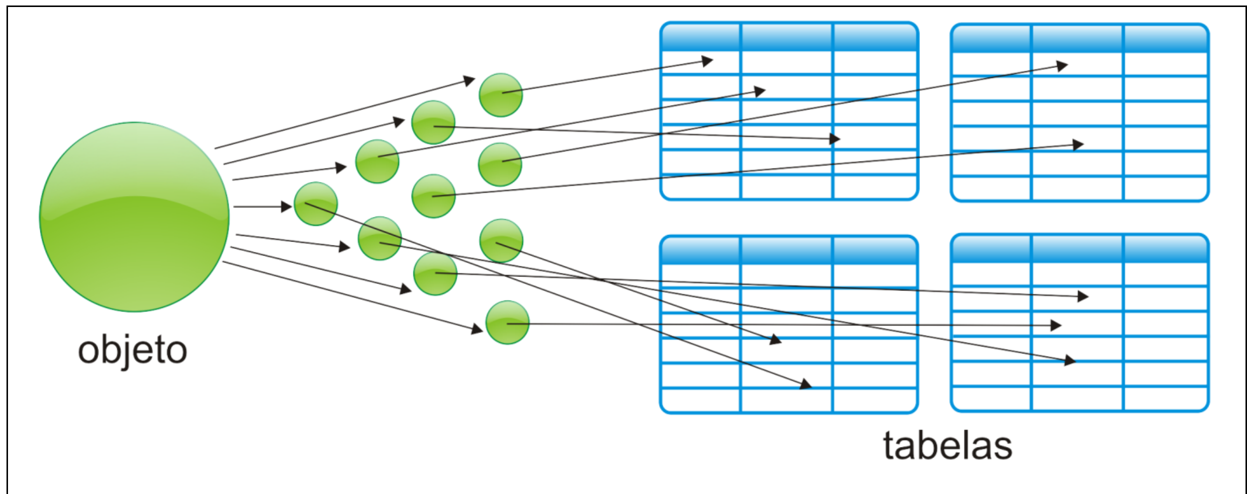


Figura 3 – Mapeamento Objeto-Relacional

O significado para mapeamento objeto relacional é o mesmo tanto via JDBC manualmente, como também, para a persistência feita automaticamente com o uso de framework, pois quando é implementado um código SQL que executa um *select* [2]e extrai os resultados para seus objetos, ou até mesmo, quando é persistido um objeto para dentro do banco de dados, seja qual for a forma utilizada, o mapeamento está sendo realizado (REIS, 2007).

## 4.1 MAPEAMENTO OBJETO RELACIONAL VIA JDBC

Aplicações Java que necessitam de comunicação com o SGBD utilizam uma API denominada *Java Database Connectivity* (JDBC). É um conjunto de classes e interfaces escritas em Java que faz o envio de instruções SQL para qualquer banco de dados relacional. Essa API é de baixo nível e é a base para as outras de alto nível. Todo o trabalho é feito no mesmo nível do banco de dados, sendo o acesso às informações armazenadas feito por meio de comandos SQL (BROGNOLI, 2008; LOPES, 2008). A comunicação entre o aplicativo e o

[2] Instrução do SQL, usada para selecionar os dados em um DB

banco propriamente dita é feira com o uso de um *driver* [3] JDBC. Esses *drivers* são bibliotecas Java, e são especificas para cada SGBD.

Em outras palavras, como o uso dessa API, uma aplicação Java pode se conectar a qualquer banco de dados relacional, submeter comandos SQL para execução e recuperação dos resultados gerados pela execução desses comandos. Além de permitir acesso aos metadados do banco de dados, permitindo a construção de ferramentas para administração do próprio banco e apoiando o desenvolvimento de sistemas (REIS, 2007).

## 4.2 MAPEAMENTO OBJETO RELACIONAL VIA FRAMEWORK

As ferramentas ORM fazem o mapeamento entre o modelo orientado a objetos e o modelo relacional, agindo como um intermediário entre um código orientado a objetos e um banco de dados relacional, tornado a persistência de objetos transparente ao desenvolvedor. O framework deve ser capaz de interpretar e resolver diferentes situações de impedância por meio do uso de *joins*[4]e outros artifícios disponíveis (PINHEIRO, 2005).

Com a utilização desses frameworks é criada uma camada de persistência. Essa nova camada oferece melhor abstração orientada a objetos do que somente o uso do JDBC, pois, por meio dela podem ser realizados mapeamentos como herança e agregação. Existem diversas ferramentas que realizam a persistência automática, sendo a mais conhecida o Hibernate (REIS, 2007).

[3] Componente de software que permite que uma aplicação interaja com outra.
[4] Cláusula SQL usada para combinar duas ou mais tabelas em um banco de dados.

# 5 HIBERNATE

É um framework de código aberto baseado em mapeamento objeto relacional que permite que a persistência dos dados seja feita de forma simples, onde o desenvolvedor pode trabalhar apenas com a *bean* [5], e não com instruções SQL (DOWNEY, 2007).

Teve sua implementação iniciada por Gavin King no final de 2001, logo foi aberto para a comunidade de desenvolvedores Java, depois passou a ser parte do projeto JBoss, uma divisão de Red Hat (SAM-BODDEN, 2006, tradução nossa). Segundo San-Boddem (2006, tradução nossa) o Hibernate tornou-se um exemplo de sucesso de um projeto *open source*[6], com boa documentação e apoio da comunidade.

Segundo Sam-bodden (2006, tradução nossa) é um mecanismo de persistência orientada a objetos transparente para Java, utilizado para mapear classes em tabelas de bancos de dados relacionais e vice-versa, para isso suporta herança, polimorfismo, composição, coleção e mapeamento de associação entre objetos. Além de realizar o mapeamento objeto relacional disponibiliza um mecanismo de consulta de dados.

Utiliza reflexão e codificação de *bytecode* em tempo de execução e a geração de SQL ocorre na inicialização do sistema. Isso assegura que o Hibernate não impactará o processo de *debug* [7]de compilação incremental.

Essa ferramenta oferece suporte à diversos bancos de dados e o mapeamento entre os objetos e as tabela podem ser definidos em um documento XML ou como o uso de anotações. O framework fica entre os objetos Java tradicionais e manipula todo o trabalho na persistência desses objetos.

---

[5] Classes Java escritas em determinado padrão.
[6] Projeto de código aberto, disponível para uso e modificação da comunidade.
[7] Programa, ou componente de um programa, que auxilia o programador a encontrar erros de programação.

Segundo Kumar (2006) apud Martins; Santos (2007) O Hibernate pode ser observado dividido em três camadas principais:

a) **gerenciador de conexão**: gerencia de forma eficiente a conexão com o banco de dados;
b) **gerenciador de transações**: permite ao usuário executar mais de um conjunto de operações no banco de dados;
c) **mapeamento objeto relacional**: Faz o mapeamento por meio de recuperação, inserção, atualização e remoção dos dados nas tabelas.

## 5.1 ARQUITETURA

O Hibernate possui o uso muito flexível, por isso sua arquitetura pode possuir várias abordagens. Para uma boa compreensão de sua arquitetura basta entender a abordagem mais simples, na qual se usa o mínimo dos recursos, e a abordagem completa, que oferece o máximo de abstração.

Na arquitetura mais simples, veja Figura 4, o aplicativo é responsável por fornecer as conexões JDBC e gerenciar suas transações. Usando o mínimo das APIs do Hibernate.

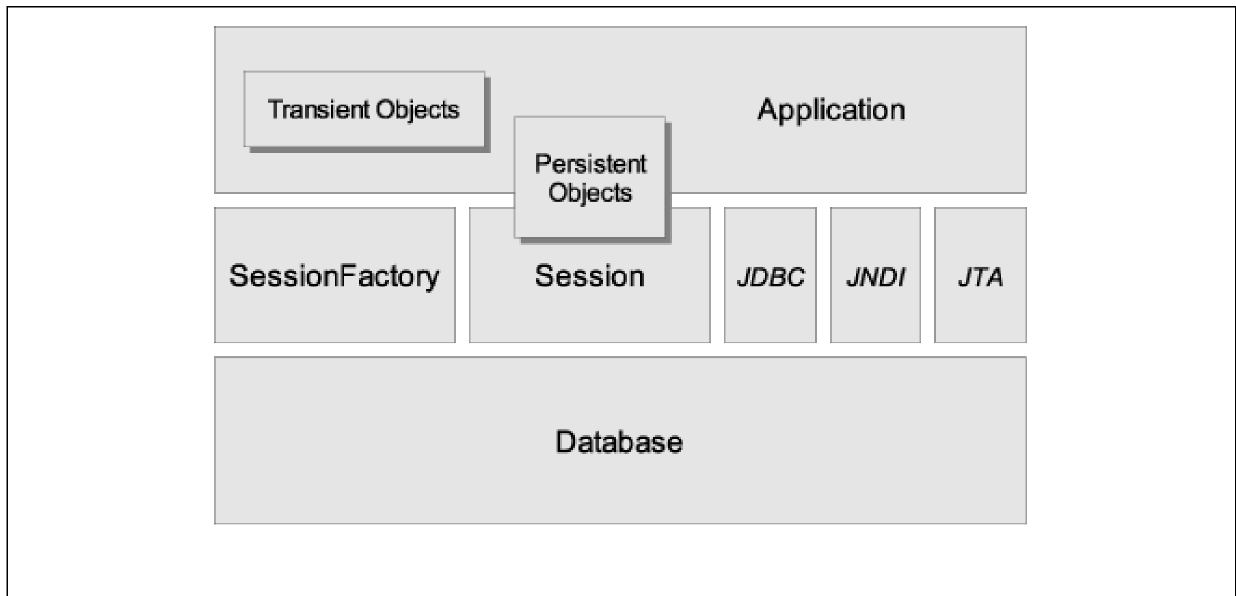


Figura 4 – Arquitetura mínima do Hibernate
Fonte : KING, K at al (2010, p 27)

Para King (2006, tradução nossa) a arquitetura completa, veja Figura 4, abstrai aplicação de ter que lidar diretamente como JDBC/JTA, deixando a framework gerenciar todos os detalhes para a persistência dos dados, aproveitando o maior número de APIs oferecidas pelo framework.

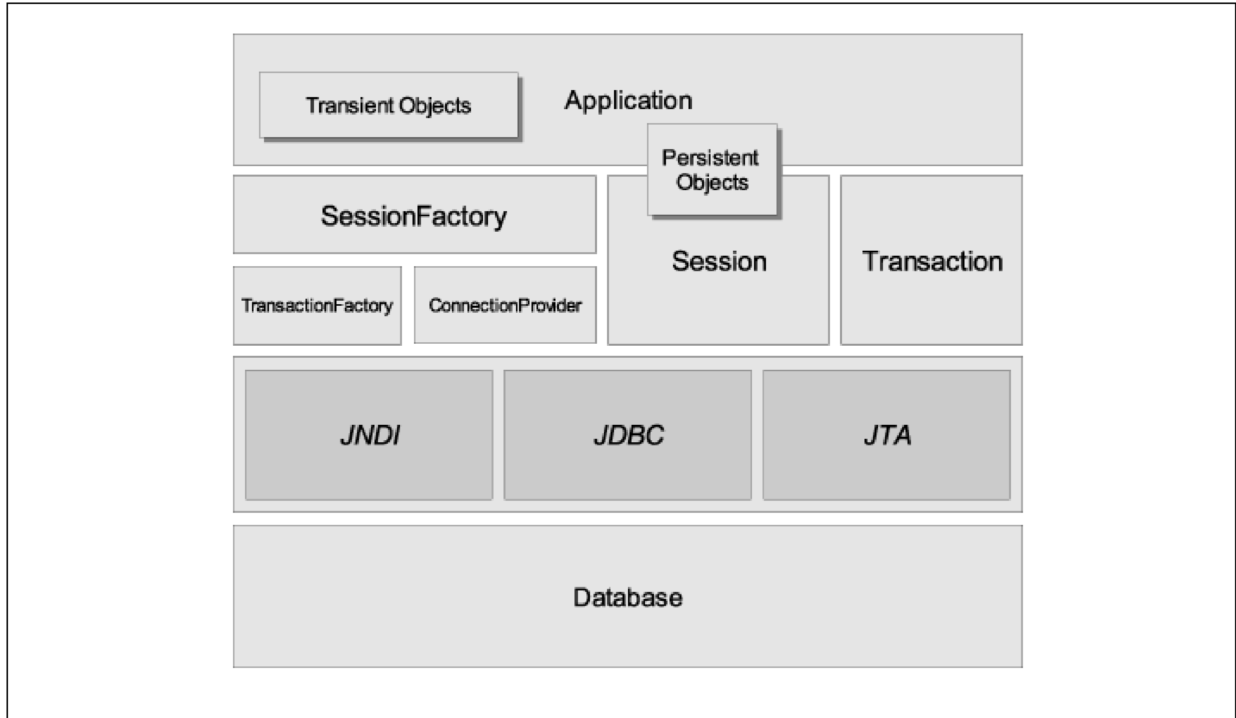


Figura 5 - Arquitetura Completa Hibernate
Fonte : KING, K at al (2010, p 27)

### 5.1.1 APIs Básicas

Na Figura 5 pode-se ver as APIs básicas para o uso do Hibernate, entre elas estão:

a) ***session Factory***: é a fábrica de sessões, composta de identidades compiladas para um único banco de dados.

b) **s*ession***: Objeto de vida curta, que faz a ponte entre o aplicativo e o banco. Cria uma camada sobre uma conexão.

c) **p*ersistent Objects***: contem o estado persistente e a função de negócios. É um Java Beans associado a uma *Session*.

d) **t*ransaction***: Usado pela aplicação para especificar unidades atômicas de trabalhos. Abstrai o aplicativo de lidar diretamente com transações JDBC.

e) ***connectionProvider***: Uma fábrica de conexões JDBC. Abstrai a aplicação de lidar diretamente com *DataSourse* PI *DriverManager*.

f) ***transactionFactory***: Uma fábrica para instâncias de *Transaction*

## 5.2 COMO O HIBERNATE FUNCIONA

No centro de cada iteração entre o código e o banco de dados encontra-se a *Session.* A Figura 6 fornece uma visão geral de como o Hibernate funciona. Ilustrando os arquivos XML para o mapeamento, os bancos de dados, as tabelas, as classes, a *Factory* e a *Session*, bem como o relacionamento entre esses elementos.

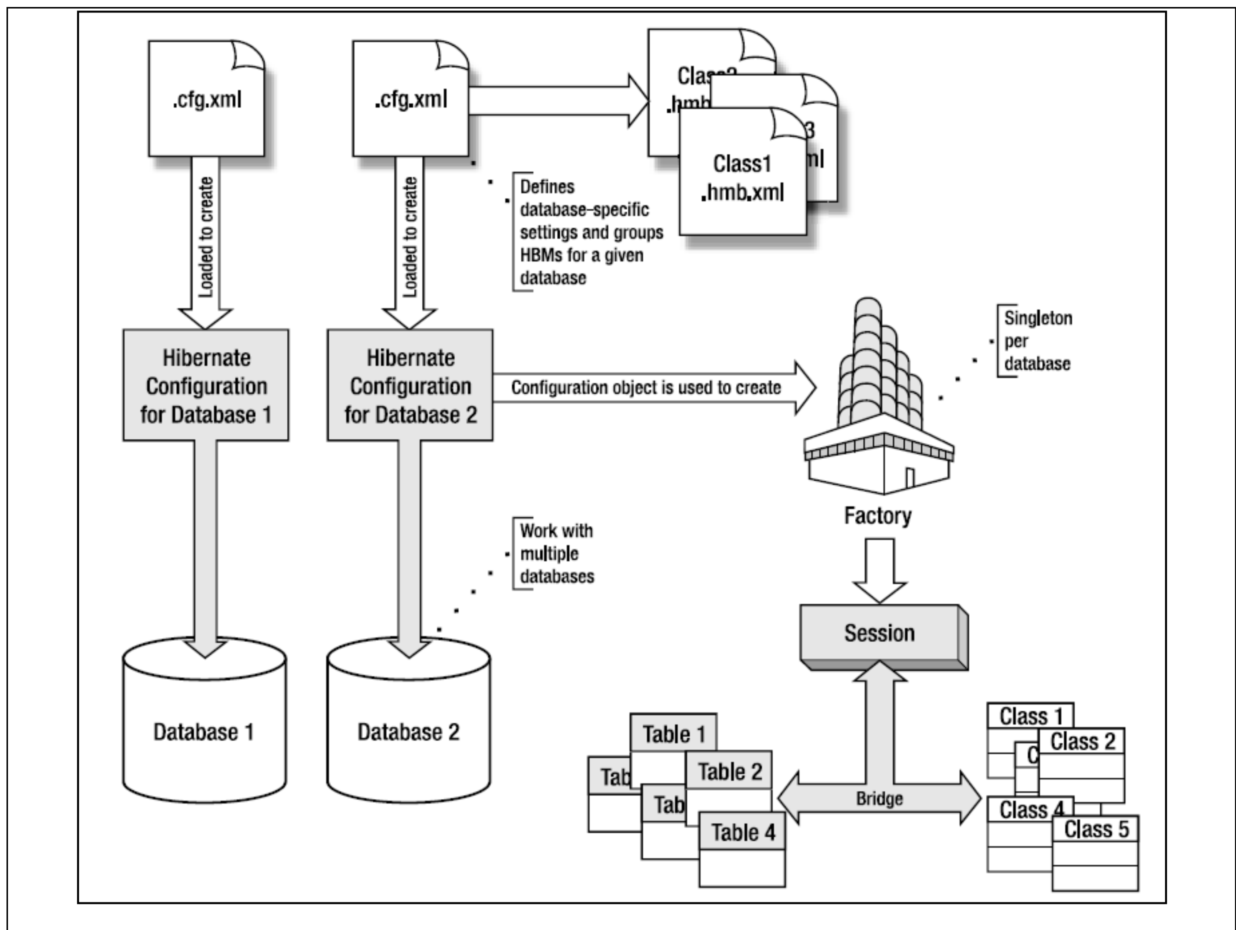


Figura 6 - Como o Hibernate funciona
Fonte: SAM-BODDEN, B. (2006, p. 92)

### 5.2.1 Hibernate Session

*Hibernate Session* é a principal interface entre uma aplicação Java e o Hibernate. Essa é a classe principal da API que abstrai a complexidade do serviço de persistência (ORG.HIBERNATE, 2010).

*Session* é um objeto leve e de vida curta que representa uma conversação entre o aplicativo e o armazenamento persistente, agrupando o conceito de gerenciador de persistência. Cria uma camada sobre uma conexão JDBC. E é usado para executar operações na instância de uma classe mapeada pelo framework, tais como, consulta, inserção, atualização e exclusão (KING et al, 2010, tradução nossa ; SAM-BODDEN, 2006, tradução nossa).

### 5.2.2 *Session Factory*

É um objeto pesado configurado para trabalhar com uma plataforma especifica de banco de dados, que idealmente deveria ser criado apenas uma vez e disponibilizado para o aplicativo executar suas operações de persistência (SAM-BODDEN, 2006, tradução nossa).

O *Session Factory* é uma classe composta de identidades compiladas para um único banco de dados, responsável por compilar e armazenar informação sobre as classes mapeadas e sobre e as informações necessárias para a conexão ao banco usado no aplicativo, além das classes mapeadas. É usado pra recuperar o Hibernate *Sessions* (KING at al, 2010, tradução nossa; SAM-BODDEN, 2006, tradução nossa).

### 5.2.3 Ciclo de Vida de um Objeto Mapeado

O ciclo de vida de uma *Session* é limitado pelo início e no final de uma operação lógica. (Transações longas podem abranger várias operações de banco de dados) (ORG.HIBERNATE, 2010).

Os objetos mapeados podem ter três estados possíveis, que são definidos respeitando um contexto persistente. O objeto Session do Hibernate é o contexto persistente. Entender esses estados é importante para quando se lida com problemas mais complexos. A Figura 7 auxilia na compreensão do ciclo de vida de um objeto mapeado (KING at al, 2010, tradução nossa).

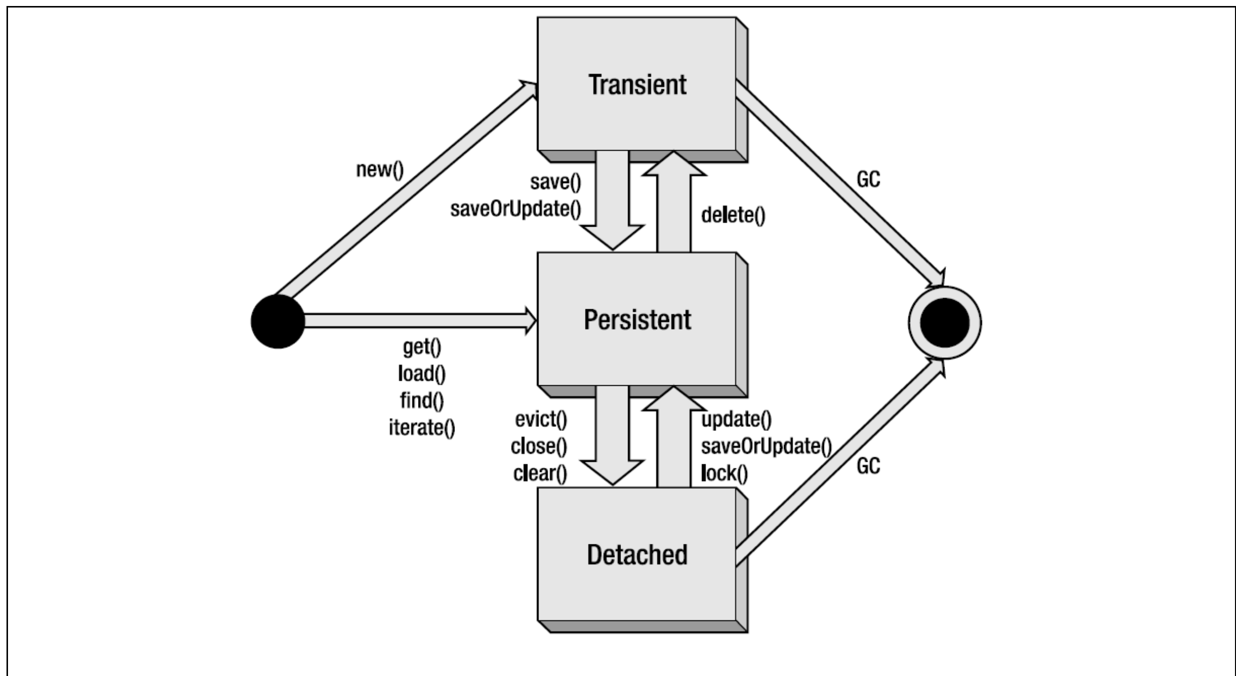


Figura 7 - Ciclo de vida do objeto mapeado.
Fonte: SAM-BODDEN, B. (2006, p. 92)

A Figura 7 mostra como diferentes métodos fornecidos pela Hibernate *Session* fazem transação de um objeto mapeado de estado para estado.

#### 5.2.3.1 Estado Transiente

No estado transiente, o objeto não está associado com nenhum contexto persistente. Ou seja, seu estado ainda não foi salvo em uma tabela, e o objeto não possui nenhuma identidade associada ao banco (Valor da chave primária). Objetos no estado transiente são não-transacional, o que significa que não participam a uma operação vinculada a uma sessão do Hibernate. Depois de uma chamada bem sucedida do método *save* ou um objeto deixa de ser transiente e se torna persistente. A método *delete* (ou uma consulta de exclusão) produz o efeito inverso tornando um objeto persistente em transiente (KING et al, 2010, tradução nossa; SAM-BODDEN, 2006, tradução nossa).

#### 5.2.3.2 Estado Persistente

Objetos persistentes são aqueles já associados a um registro no banco de dados, que possuem uma identidade persistente (valor da chave primária). Se eles não tiverem sido atribuídos a uma chave primária, mas ainda não foram salvos no banco de dados, eles são referidos como estando no estado "novo".

Os objetos persistentes são transacionais, o que significa que eles participam de transações associadas com a sessão (no final da operação o estado do objeto será sincronizado com o banco de dados (KING et al, 2010, tradução nossa; SAM-BODDEN, 2006, tradução nossa).

#### 5.2.3.3 Estado Desanexado

Objeto desanexado é aquele cujo a instância foi associada com um contexto persistente, porém este contexto foi fechado, ou seja já não está associado a sessão.

Possui uma identidade persistente, e, pode corresponder a um registro no banco de dados. Para instâncias desanexadas, o Hibernate não garante o relacionamento entre identidade persistente e identidade Java. Isto acontece depois de uma transação é concluída, quando a sessão é fechada, excluída, ou se o objeto for despejado do cache (KING et al, 2010, tradução nossa; SAM-BODDEN, 2006, tradução nossa).

## 5.3 CONFIGURAÇÕES

O Hibernate oferece suporte para trabalhar com diversos bancos de dados, isso requer que o ele seja altamente configurável, podendo, assim se adaptar a diferentes ambientes. Também existem mudanças nos ambientes Java, por exemplo, a obtenção da conexão com o banco de dados em um ambiente web pode ter diferenças significativas em relação a uma aplicação *desktop*. Os requisitos variam entre os diferentes ambientes.

Segundo Peak e Heudecher (2005, tradução nossa) o Hibernate é geralmente configurado em dois passos. Primeiro se configura o *Hibernate Service*, com os parâmetros de conexão do banco de dados, e, a coleção das classes persistentes. Em segundo lugar, informa-se as dados sobre as classes a serem persistidas. Essa configuração faz a ligação entre as classes e o banco de dados.

No Hibernate, o mapeamento entre objetos e tabelas pode ser definido em documentos XML, ou em código Java via anotações de persistência (SAM-BODDEN, 2006, tradução nossa).

# 6 TRABALHOS CORRELATOS

Essa seção relaciona alguns dos trabalhos encontrados com o conteúdo semelhante a esta fundamentação teórica, utilizados no desenvolvimento nessa pesquisa.

## 6.1 ANÁLISE E AVALIAÇÃO DO FRAMEWORK HIBERNATE EM UMA APLICAÇÃO CLIENTE/SERVIDOR

Trabalho de Conclusão de Curso de Caroline Fernando Silva, apresentado ao Curso de Ciência da Computação pela Faculdade de Jaguariúna no ano de 2007.

Nesse projeto são descritos os objetivos, vantagens e desvantagens do *Hibernate*, realizando a persistência de objetos em diferentes SGBDs, apresentando os aspectos de sua arquitetura a fim de conhecer seu papel dentro de aplicações desenvolvidas no paradigma orientado a objetos.

Silva concluiu que o mapeamento objeto relacional soluciona de maneira bastante eficiente os problemas de incompatibilidade, apresentando diversas técnicas de mapeamento e de busca visando um melhor desempenho.

## 6.2 IMPEDÂNCIA DE DADOS EM BANCO DE DADOS

Trabalho de Conclusão de Curso de Daniel Guessi Plácido apresentado para obtenção do grau de bacharel em Ciência da Computação da Universidade do Extremo Sul Catarinense em 2008.

Este trabalho teve como objetivo estudar os problemas de mapeamento de dados entre bancos de dados orientados a objetos e relacionais, por meio de uma linguagem de

programação orientada a objetos. Na pesquisa foi utilizado o *framework* Hibernate. Foram construídos dois modelos de testes, um com uma aplicação orientada a objeto com bancos de dados relacionais (MySQL e PostGreSQL), utilizando-se do *framework* Hibernate afim de se realizar o mapeamento objeto relacional e outro com sistema orientado a objetos com bancos de dados orientado a objetos (Caché e DB4O) verificando o comportamento de padrões de objetos.

Por fim este trabalho apresenta os resultados obtidos, diferenciando os modelos de bancos de dados, e quais as melhores opções para se evitar o problema de impedância de dados. As conclusões a respeito do mapeamento foram:

a) os bancos de dados relacionais limitam a aplicação ao banco de dados utilizado. Nesse casso o uso de um *framework* ORM se faz necessária. Porem perde-se tempo com mapeamento das classes.
b) com o DB4O a diferença de mapeamento é nula, Contudo a pesquisa no banco é limitado pela falta de comandos SQL.
c) no banco pós relacional Caché notou-se uma grande diferença no controle do problema de impedância de dados. Com sua arquitetura unificada de dados é possível trabalhar com objetos e atributos atendendo todos os seus tipos ao mesmo tempo, sendo que a tradução entre eles é automatizada, além de permitir consultas SQL.

## 6.3 FRAMEWORK’S PARA MAPEAMENTO OBJETO RELACIONAL: UM ANÁLISIS COMPARATIVO

Trabalho de graduação de Víctor Adolfo González García pra obtenção do título de Engenheiro em Ciências e Sistemas da Universidade de San Carlos de Guatemala em 2009.

Faz uma comparação entre os frameworks Hibernate e Ibatis focando nas funcionalidades e recursos disponíveis em cada um estabelecendo as vantagens e desvantagens, fornecendo assim as informações necessárias para determinar qual ferramenta utilizar em determinada situação. Essa comparação foi feita considerando as seguintes características: reusabilidade, persistência transitiva, facilidade de uso, curva de aprendizado, facilidade de testes, escalabilidade, ferramentas de apoio, integração, segurança, facilidade de refatoração, concorrência, desempenho, cachê, facilidade de debug, suporte a transações e instruções SQL.

Em conclusão foi relatado para que tipo de projeto cada um é indicado, García concluiu que Itabis é mais simples, também é mais indicado quando se tem sistemas com muitas camadas e procedimentos armazenados e instruções SQL e como único mecanismo para obtenção de modificação de dados. Por outro lado o Hibernate é um ORM que proporciona independência entra a base de dados e a lógica de negócio, possui linguagem de consulta própria e é menos intrusivo que os outros frameworks, pois usa reflexão e gera os bytecodes em tempo de execução (GARCÍA, 2009);

# 7 COMPARAÇÃO ENTRE JDBC E HIBERNATE

Hibernate e JDBC apesar de serem diferentes possuem o objetivo em comum de permitir a persistência de dados de uma linguagem orientada a objetos. Cada uma possui suas particularidades, porém ambos são utilizados para salvar os objetos em um SGBDR.

## 7.1 METODOLOGIA

Para a realização da avaliação foi estudado as características gerais de cada um. Além do estudo foram criados dois protótipos a fim de testar os diferentes modos de implementação, buscando focar na utilização do paradigma OO, os protótipos também foram submetidos à comparação do tempo gasto na persistência e recuperação dos objetos.

## 7.2 PROTÓTIPO DE APLICAÇÃO

A fim de avaliar as diferenças na codificação e desempenho nas formas de se persistir os objetos foi desenvolvido um cenário de aplicação de uma loja na web. Foram criados dois protótipos, um com o uso do JDBC e outro com a utilização do Hibernate.

O funcionamento do sistema foi bastante simplificado, com o objetivo de demonstrar a aplicação desses métodos de forma clara, prática e didática.

Os protótipos foram desenvolvidos em linguagem Java, usando a versão 7 da JVM , na IDE de desenvolvimento NetBeans 7.0.1. O SGBD utilizado foi o PostGresSql 9.0, com o JDBC correspondente versão 4. A versão do Hibernate foi a 3.

### 7.2.1 Concepção e Modelagem do Protótipo

Com o objetivo de avaliar os métodos de persistência de dados em uma linguagem OO criou-se um cenário com requisitos comumente encontrados em um sistema real como cadastro, exclusão, edição, listagem e relatório dos dados.

Optou-se pelo desenvolvimento de uma aplicação *e-commerce*, onde o usuário poderá efetuar compras. O cliente poderá navegar entre itens organizados em categorias, adicionar ao carrinho de compras e encerrar a transação. Também haverá a parte administrativa onde um usuário administrador poderá gerenciar, por exemplo, os produtos e categorias.

A concepção do protótipo foi criada visando o uso dos principais recursos oferecidos pelo paradigma relacional (usado nos SGBDR's) e pelo paradigma de programação Orientado a Objetos. Já que, como mencionado no começo dessa pesquisa, ambos muitas vezes são utilizados conjuntamente, causando a impedância de dados. Para tal foram criados requisitos que abrangessem o uso dos recursos de OO, listados na tabela 1 e dos recursos dos das bases relacionais, listados na tabela 2.

**Tabela 1 – Recursos OO utilizados**

| **Recursos de Programação Orientada a Objetos** |
|---|
| Recurso |
| Classe simples |
| Classe com tipo complexo |
| Herança |
| Coleção |

**Tabela 2 – Recursos de bancos relacionais utilizados**

| **Recursos de Banco de Dados Relacionais** |
|---|
| Recurso |
| Relacionamento “one to many” e “many to one” |
| Relacionamento “many to many” |
| Funções de agregação |
| Consulta com “order by” |
| Consulta com “inner join” |
| Consulta com “right join” |

A partir da definição dos cenários relevantes à avaliação criou-se uma loja online buscando preencher todas as características do cenário possibilitando os testes e avaliações.

Por se tratar de um protótipo de aplicação, a intenção é criar os cenários necessários para a realização dos testes, por isso muitas características quer seriam necessárias a uma aplicação real não serão implementadas, tais como controle de acesso ou finalização de uma compra, que não seriam relevantes na avaliação dos métodos de persistência dos dados.

#### 7.2.1.1 Funcionamento do sistema e levantamento dos requisitos.

A partir da escolha do cenário do protótipo foram estabelecidos os seguintes requisitos:

a) o sistema iniciará exibindo uma lista dos cadastros e listagens disponíveis;
b) se for selecionado um cadastro, o sistema irá para a tela de cadastro correspondente;
c) se for selecionado uma listagem, o sistema mostrará a lista dos itens cadastrados na base de dados, também deixará disponível uma opção para editar os itens listados;
d) se for selecionado o cadastro de pedidos, o sistema abrira uma tela com uma lista de produtos para serem adicionados ao carrinho;

e) no decorrer do processo, uma listagem com os itens previamente marcados para compra devera estar acessíveis ao usuário, para possível confirmação da operação ou retornar e selecionar outros itens para compra.

7.1.1.2 Análise e levantamento de requisitos

Após a obtenção de dados, passa-se à etapa de análise destas informações e levantamento de requisitos do protótipo a ser desenvolvido. A tabela 3 apresenta os alguns dos requisitos funcionais levantados.

**Tabela 3 – Requisitos Funcionais**

| REQUISITO | DESCRIÇÃO |
|---|---|
| **F01: Exibir listagem de categorias** | O sistema deverá exibir em um painel específico as categorias de produtos, para que quando uma for selecionada exiba a listagem dos respectivos itens. |
| **F02: Exibir listagem de itens** | O sistema deverá retornar a listagem de itens das categorias selecionada pelo usuário. Os itens em descrição serão dispostos em um painel específico, que proporcione a navegação de usuário pela listagem e posterior seleção de item. |
| **F03: Adicionar itens ao carrinho** | O usuário poderá adicionar o item visualizado ao carrinho de compras. |
| **F04: Visualizar carrinho de compras** | O usuário poderá consultar no carrinho de compras os itens marcados para concluir a compra, caso necessário. |
| **F05: Cadastro de cliente** | Cadastro de um novo cliente no sistema |
| **F06: Exibir listagem de** | Exibe listagem dos clientes cadastrados na base de dados |

| Clientes | |
|---|---|
| **F07: Edição de Clientes** | A partir da escolha na listagem o usuário poderá escolher a opção de edição. |

#### 7.2.1.3 Casos de uso

A Tabela 4 lista os casos de uso encontrados, de forma resumida.

**Tabela 4 – Casos de Uso**

| NOME | DESCRIÇÃO |
|---|---|
| **C01** | Navegar pela aplicação. |
| **C02** | Listar itens de uma categoria |
| **C03** | Adicionar itens ao carrinho de compras. |
| **C04** | Concluir compra |
| **C05** | Cadastrar novo cliente |
| **C06** | Listar clientes cadastrados |
| **C07** | Editar cliente da lista |

#### 7.2.1.4 Diagramas de casos de uso

Os casos de uso mais relevantes levantados no processo de análise e modelagem podem ser representados pelos diagramas expostos nas Figuras 8 e 9 a seguir.

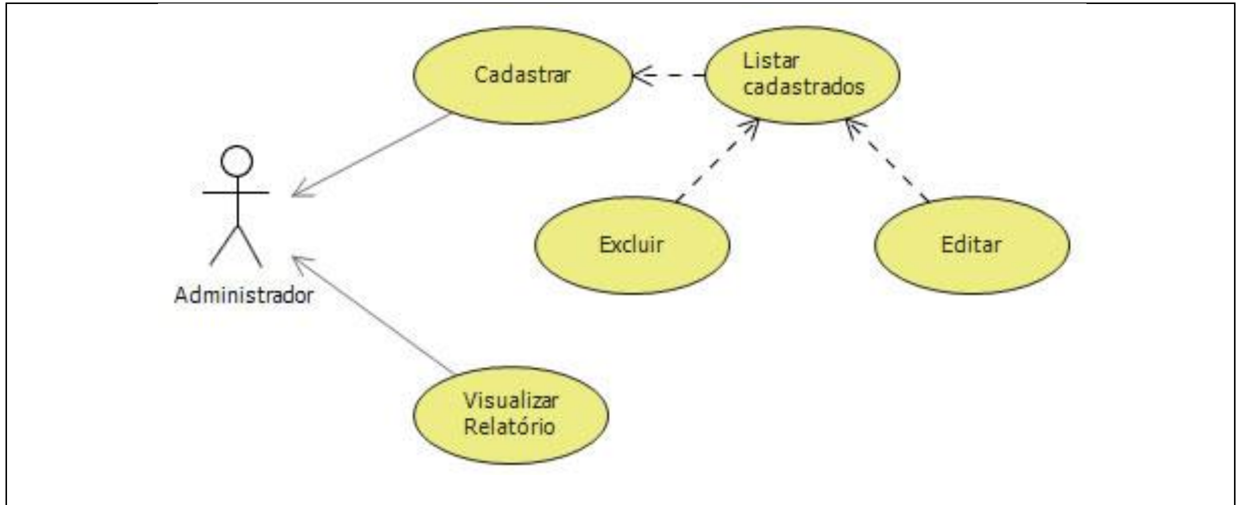


Figura 8 – Caso de uso Administrador

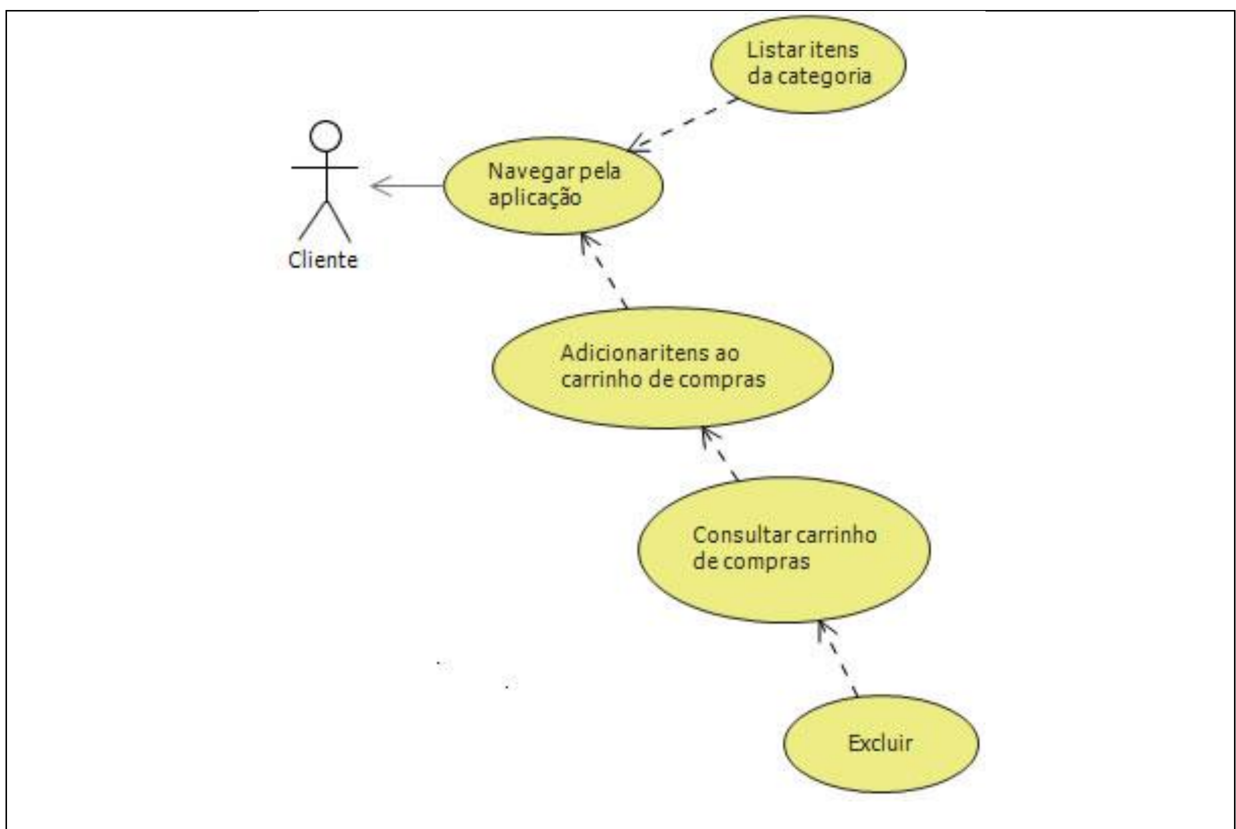


Figura 9 – Caso de uso Cliente

### 7.2.1.5 Modelagem do banco de dados

A Figura 10 apresenta o diagrama de entidade e relacionamento bando de dados, representando todas às estruturas de tabelas do projeto.

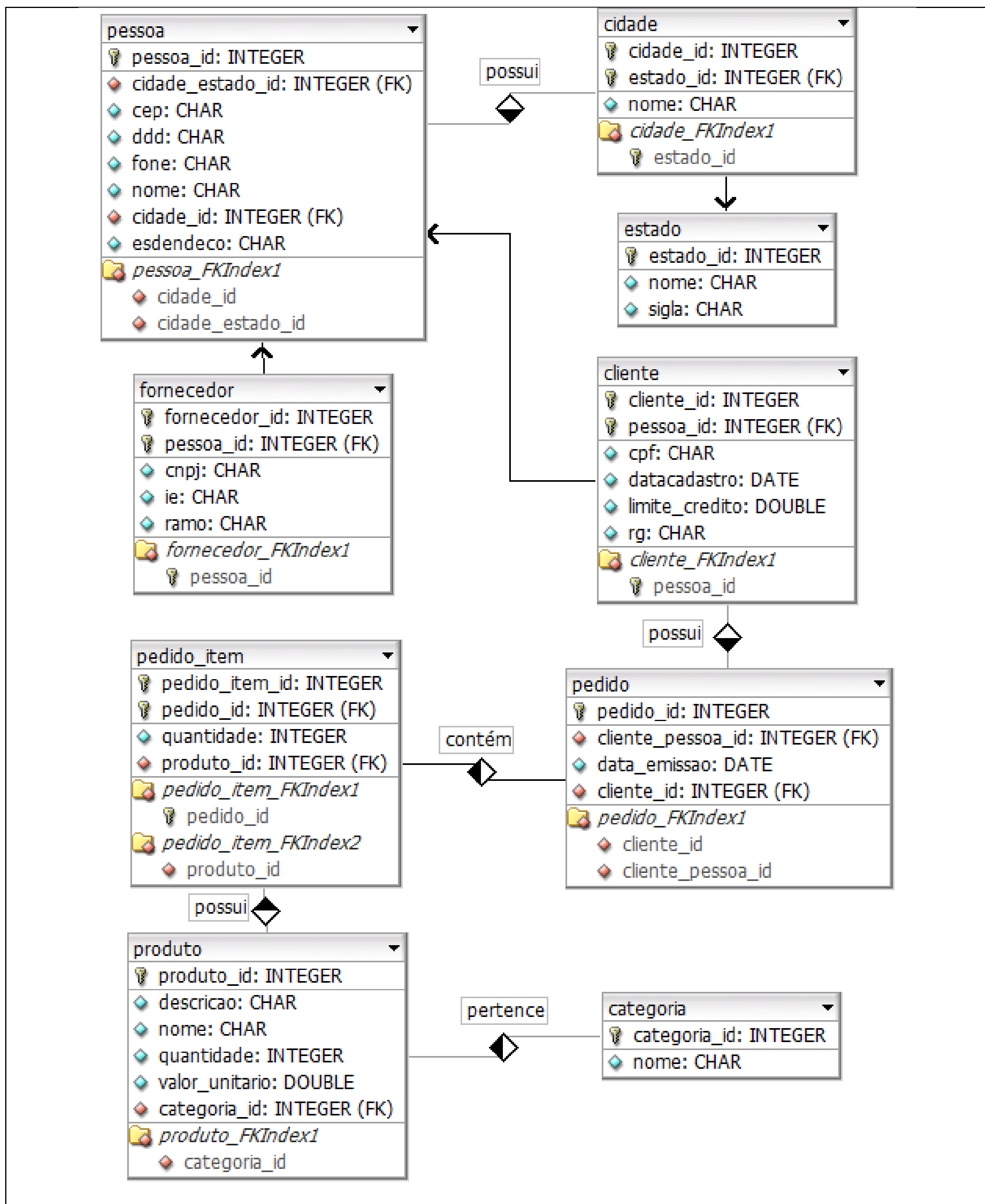


Figura 8- Diagrama ER

7.2.1.6 Diagrama de classes do cenário

A Figura 11 demonstra o diagrama de classes dos protótipos desenvolvidos. Nele, é possível observar as classes do sistema com seus devidos relacionamentos.

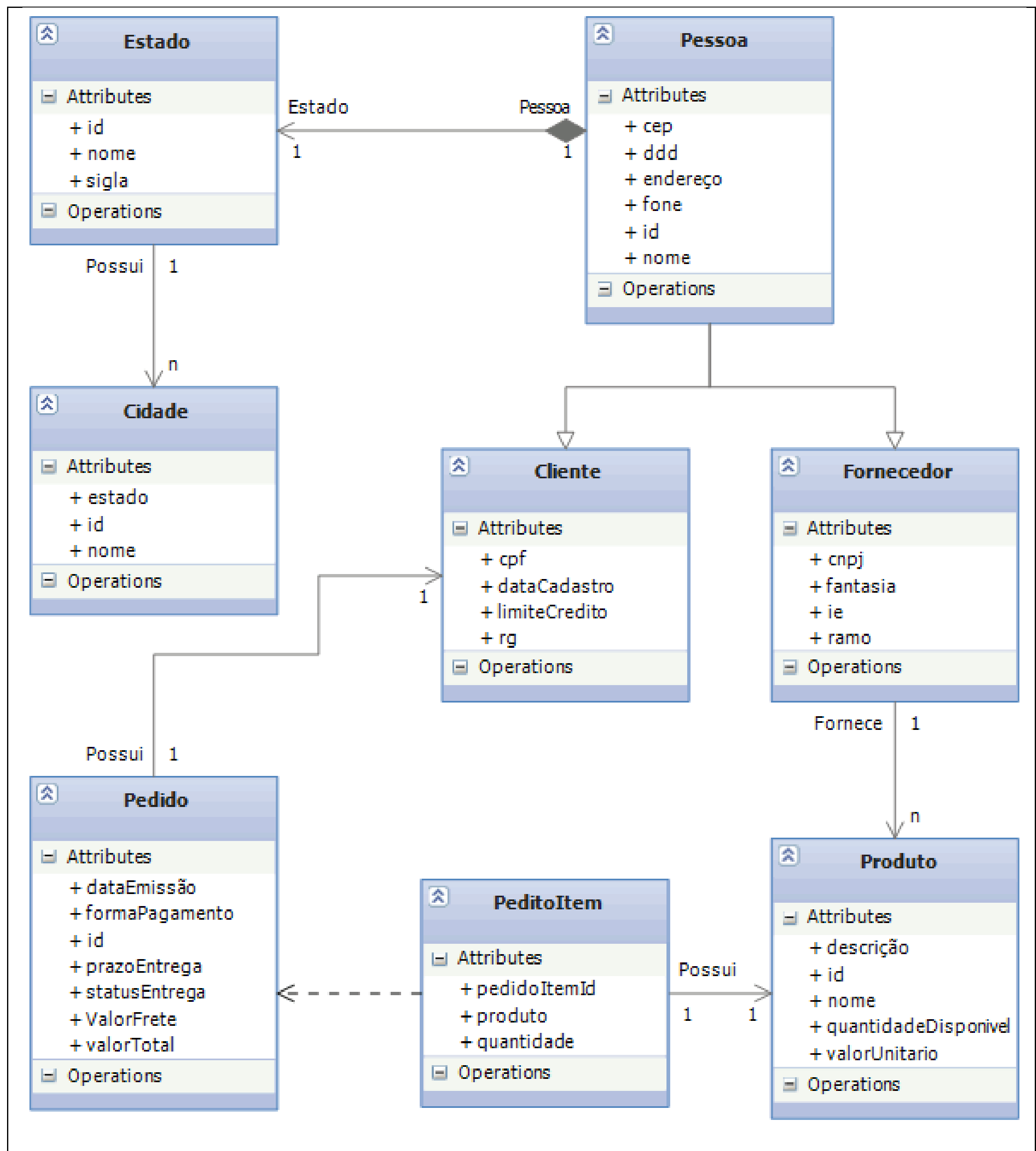

Figura 9 – Diagrama de classes

### 7.2.2 Desenvolvimento do Protótipo

Após análise das informações levantadas e da modelagem, passou-se a fase de desenvolvimento da aplicação. Para isso buscou-se seguir o padrão MVC[8].

No decorrer do desenvolvimento foram criados os seguintes pacotes principais: “modelo”, “mb” , “dao” e “Páginas Web”, como pode ser observado na Figura 12. Os dois protótipos desenvolvidos apresentam exatamente os mesmos códigos-fonte nos pacotes “Páginas Web”, e “mb”. No pacote “modelo” as classes e atributos são os mesmos, diferindo entre o protótipo ORM e o JDBC apenas as anotações adicionadas ao protótipo ORM, responsáveis por fazer o mapeamento.

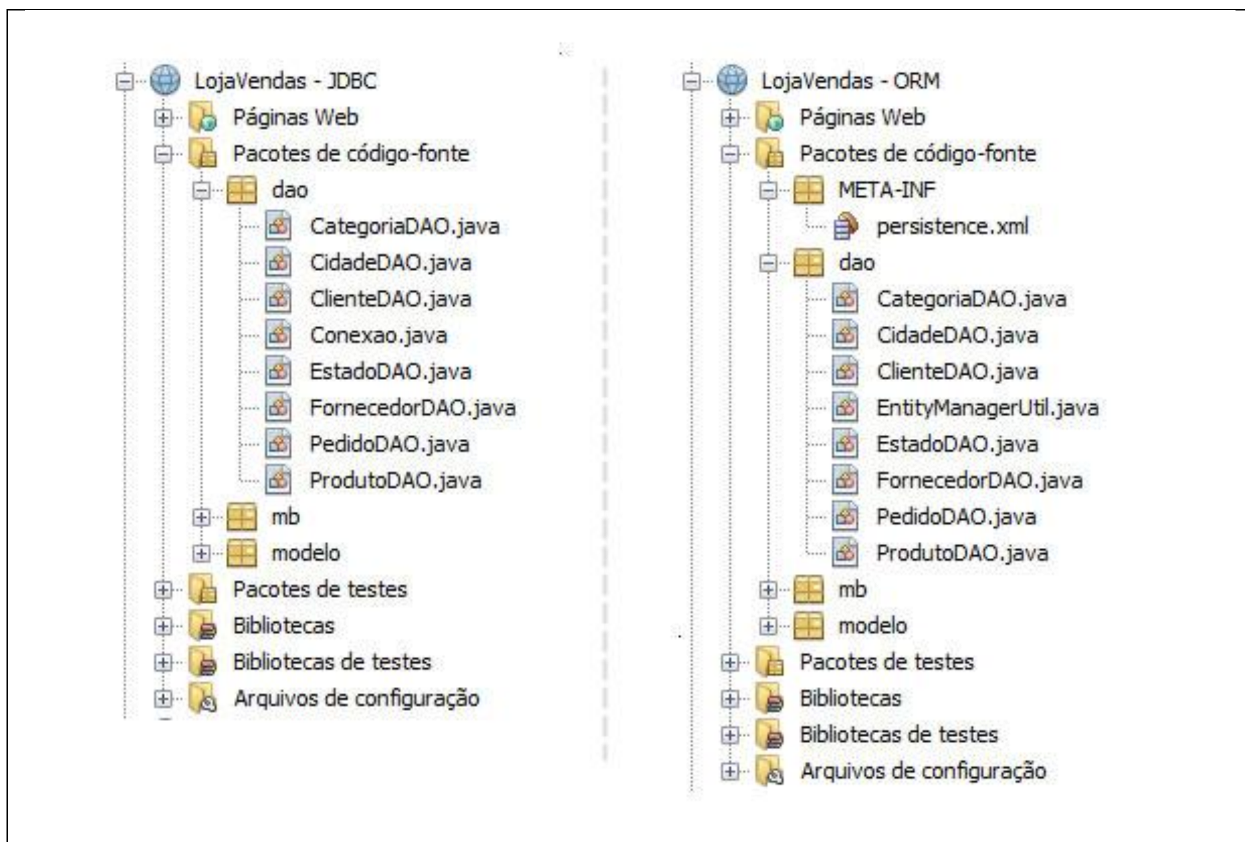

Figura 10 – Estrutura dos cenários desenvolvidos

[8] Model-View- Control, padrão de arquitetura de software que visa separar os dados ou lógica de negócios (Model) da interface do usuário (View) e do fluxo da aplicação (Control)

A diferença mais evidente entre os dois cenários se encontra na camada “dao”, nessa camada é onde efetivamente se utiliza os recursos que serão analisados nessa pesquisa.

Outra diferença entre os dois cenários são as classes “Conexão.java” e “EntityManagerUtil.java”. Onde, “Conexao.java” é utilizada no cenário JDBC, sendo responsável pela conexão ao SGBDR. E “EntityManagerUtil.java”, necessária no cenário ORM, encapsula a criação da *sessionFactory,* que gerencia a criação das *sessions*, esta, por sua vez se conecta ao banco.

No cenário ORM também é preciso criar um pacote auxiliar, como o nome “META-INF”, que tem o arquivo “persistence.xml”, que apresenta as propriedades das Entidades e mecanismos de acesso ao SGBD e configuração do ORM, para que a aplicação possa acessar a base de dados.

A seguir serão mostradas de forma mais detalhada as principais diferenças encontradas nos dois cenários analisados.

#### 7.2.2.1 Visão geral do protótipo do cenário JDBC

O cenário JDBC é o modo mais simples, pois não precisa de mapeamento e arquivo de configuração.

Foram utilizados os seguintes pacotes, “dao”, ”mb” e “modelo”, além de “Páginas Web”. Onde, como já dito apenas “modelo” e “dao” são diferente entre os cenários.

No pacote modelo as classes não necessitam de qualquer artifício adicional. Para ilustrar as diferenças entre os cenários vamos utilizar a classe “Estado.java” do pacote “modelo”. Observe a Figura 13, são necessários apenas os atributos e seus métodos de acesso.

```
package modelo;
import java.io.Serializable;

public class Estado implements Serializable {
```

```
    public void Estado() {
    }
    private Integer id;
    private String nome;
    private String sigla;

    //atributos gets e sets
    ...
}
```

Figura 11 – Classe Estado em uma implementação JDBC

No pacote “dao”, onde os dados são acessados, e por isso onde existe a maior diferença na codificação além dos métodos possuírem código diferente existe uma classe “Conexao.java”, essa classe não é obrigatória para a implementação, porém é usada para evitar repetição de código, ele contém as informações de acesso ao banco, tais como usuário senha e nome do banco e é responsável por retorna uma conexão como mesmo (Figura 14).

```
package dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Conexao {
    public static Connection criarInstancia() {
        Connection c = null;
        try {
            Class.forName("org.postgresql.Driver");
            c = DriverManager.getConnection(
            "jdbc:postgresql://localhost:5432/teste8", "postgres","2412");

        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(ex.getMessage());
        } catch (SQLException ex) {
            throw new RuntimeException(ex.getMessage());
        }
        return c;
    }
}
```

Figura 12- Classe “conexão.java” do cenário JDBC

#### *7.2.2.1.2 Bibliotecas utilizadas*

Somente o driver JDBC correspondente ao SGBDR utilizado é necessário. No caso dessa pesquisa foi utilizado o ProstGreSQL 9.0 com o JDBC 4. O arquivo adicionado foi "postgresql-9.0-801.jdbc4.jar".

#### *7.2.2.1.3 Mapeamento*

Quando se usa o JDBC puro não há necessidade de mapeamento prévio. Ele ocorre quando se manipula dos dados utilizando SQL, para isso é necessário que se saiba exatamente os nomes das tabelas e seus campos no SGBDR.

### 7.2.2.1Visão Geral do Cenário ORM

Além dos pacotes utilizados no cenário JDBC o cenário ORM necessita de um pacote especial chamado "META-INF", nele se encontra o arquivo "persistence.xml", que contem as o nome das classes mapeadas e as informações de acesso ao SGBDR (Figura 15).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="persistence" transaction-type="RESOURCE_LOCAL">
        <class>modelo.Cidade</class>
        <class>modelo.Estado</class>
        <class>modelo.Pessoa</class>
        <class>modelo.Cliente</class>
        <class>modelo.Fornecedor</class>
        <class>modelo.Produto</class>
        <class>modelo.Pedido</class>
        <class>modelo.PedidoItem</class>
        <class>modelo.Categoria</class>
        <properties>
```

```
        <property
                  name="javax.persistence.jdbc.url"
                  value="jdbc:postgresql://localhost:5432/teste8" />
        <property name="javax.persistence.jdbc.driver"
                  value="org.postgresql.Driver"/>
        <property name="javax.persistence.jdbc.user" value="postgres"/>
        <property name="javax.persistence.jdbc.password" value="2412"/>
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.PostgreSQLDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

Figura 13 – Código do arquivo de configuração “persistence.xml”

Nesse caso no pacote “dao” não é utilizado o arquivo “Conexao.java”, no seu lugar se usa o “EntityManagerUtil.java” (Figura 16) que cria uma *sessionFactory* do Hibernate. Diferentemente de “Conexao.java” essa classe não possui as informações sobre o banco, essas agora são encontradas em “persistence.xml”.

```java
package dao;
import javax.persistence.*;

public final class EntityManagerUtil {
    private static EntityManagerFactory emf;

    private EntityManagerUtil() {
    }

    public static EntityManagerFactory getEntityManagerFactory() {
        if (emf == null) {
            emf = Persistence.createEntityManagerFactory("persistence");
        }
        return emf;
    }

    public static EntityManager getEntityManager() {
        return getEntityManagerFactory().createEntityManager();
    }
}
```

Figura 14 – Classe EntityManagerUtil

*7.2.2.2.2 Bibliotecas Utilizadas*

Além do driver JDBC, foram utilizadas as seguintes bibliotecas, Figura 17.

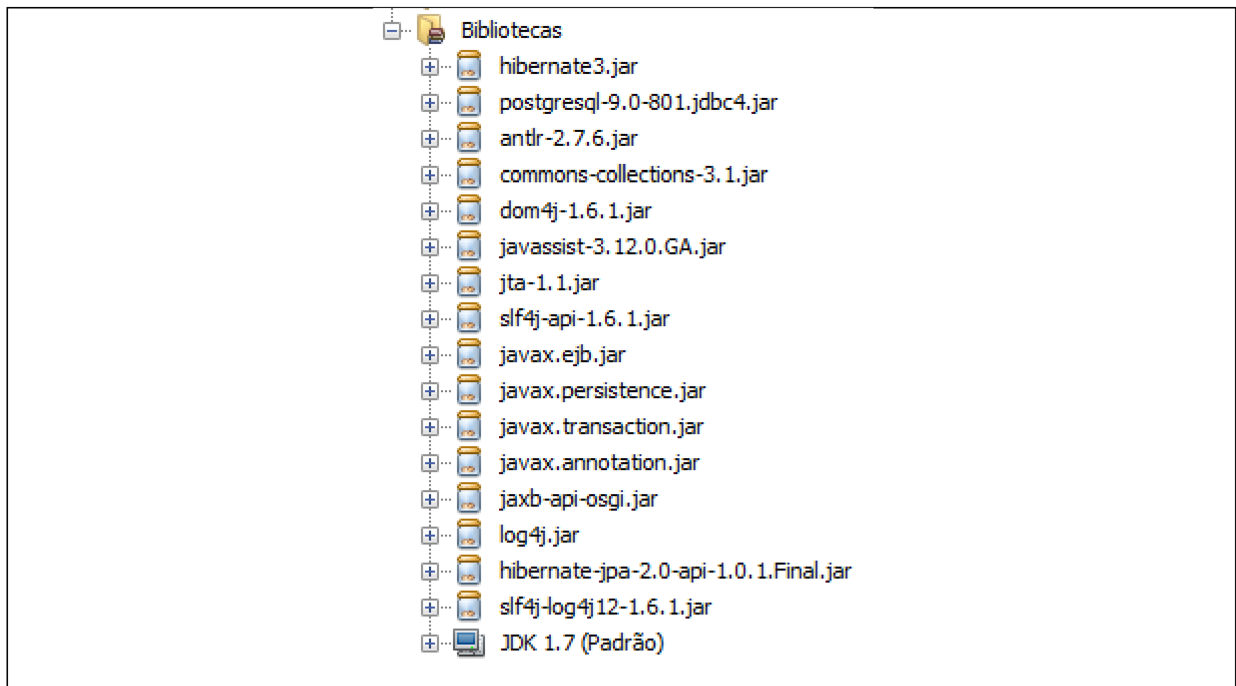

Figura 15 - Bibliotecas utilizadas no protótipo ORM

##### *7.2.2.2.3 Mapeamento*

O mapeamento utilizando o Hibernate pode ser feito de duas formas, uma com arquivos XML e outra utilizando o recurso de *annotations*. Optamos pela segunda opção por ser a mais recente. Na Figura 18 podemos ver um exemplo de mapeamento, onde é possível observar as anotações mais básicas.

Para que o Hibernate entenda que a classe deve ser mapeada usa-se as anotações *@Entity* e *@Id*, onde, a primeira indica que a classe será uma tabela na base de dados e a segunda qual campo será usado como identificador. As outras anotações são opcionais e dependem da classe que esta sendo mapeada, nesse exemplo as seguintes foram usadas:

a) @GeneratedValue(strategy = GenerationType.*SEQUENCE*, generator = "seg_cidade") : Usada em complemento com o @Id indica a estratégia de

criação do identificados e o nome da seqüência que deverá ser usada no SGBDR;

b) @Column(name = "cidade_id"): usado para indicar que o campo terá um nome diferente o SGBDR;

c) @ManyToOne(cascade = CascadeType.*ALL*, fetch = FetchType.*EAGER*): indica relacionamento entre as classes, e qual o comportamento o ORM deverá ter em relação a esse relacionamento. Nesse caso, é importante destacar o atributo "FetchType", que pode ser "Eager" ou "Lazy", onde, "Eager" recupera todas as classes vinculada no momento da busca, o que pode causar problemas de desempenho caso existam muitos relacionamento. O tipo "Lazy", conhecido como busca preguiçosa recupera somente a classe que será imediatamente usada, fazendo a busca das classes relacionadas conforma a necessidade.

d) @JoinColumn(name = "estado_id"): complementa o @ManyToOne, indicando o nome da coluna que será a chave estrangeira no SGBDR.

```
@Entity
@Table(name = "cidade")
@SequenceGenerator(name = "seg_cidade", sequenceName = "seg_cidade",
allocationSize = 1, initialValue = 1)
public class Cidade implements Serializable {

    public Cidade() {
    }
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seg_cidade")
    @Column(name = "cidade_id")
    private Integer id;
    private String nome;
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "estado_id")
    private Estado estado;

    //métodos gets e sets
}
```

Figura 16 – Exemplo de mapeamento utilizando *annotations*

### 7.2.3 Comparação entre as classes persistentes

Como mencionado anteriormente, a maior diferença entre os cenários é encontrada na camada de acesso aos dados, nela afetivamente fazemos uso dos recursos oferecidos pelos nossos alvos de comparação. Iremos comparar as classes equivalentes nos

dois protótipos criados. Avaliando o uso dos conceitos de OO no código-gerado, observando o tamanho, legibilidade e complexidade do código-fonte necessário para as operações de manipulação dos dados armazenados do SGBDR.

Como grande parte da persistência dos dados está em operações básicas, conhecidas como CRUD, faremos nossa comparação avaliando essas operações, além um relatório e o uso de uma função de agregação.

#### 7.2.3.1 Salvando e Editando os Dados

No cenário JDBC são necessários dois métodos, um para salvar (Figura 19) e outro para atualizar (Figura 20). Com o uso do SQL é preciso que se saiba os nomes das tabelas e seus campos, conforme está no banco, necessitando de consultas ao esquema do banco, isso causa uma maior possibilidades de erros, já que os nome podem ser confundidos ou até mesmo esquecidos.

```
Connection c = Conexao.criarInstancia();
try {
String sql = "INSERT INTO produto("
        + "descricao, nome, quantidade_disponivel, valor_unitario, categoria_id)"
        + " VALUES (?, ?, ?, ?, ?);";
            PreparedStatement ps = c.prepareStatement(sql);
            ps.setString(1, o.getDescricao());
            ps.setString(2, o.getNome());
            ps.setInt(3, o.getQuantidadeDisponivel());
            ps.setDouble(4, o.getValorUnitario());
            ps.setInt(5, o.getCategoria().getId());
            ps.execute();
            ps.close();
```

Figura 17 – Salvando uma classe no cenário JDBC

```
        Connection c = Conexao.criarInstancia();
        try {

            String sql = "UPDATE produto"
                    + " SET nome=?, descricao=? , quantidade_disponivel=? ,
                      valor_unitario=? , categoria_id=? "
                    + " WHERE produto_id = ?";
```

```
        PreparedStatement ps = c.prepareStatement(sql);

        ps.setString(1, o.getNome());
        ps.setString(2, o.getDescricao());
        ps.setInt(3, o.getQuantidadeDisponivel());
        ps.setDouble(4, o.getValorUnitario());
        ps.setInt(5, o.getCategoria().getId());
        ps.setInt(6, o.getId());

        ps.executeUpdate();

        ps.close();
```

Figura 18 – Atualizando uma classe no cenário JDBC

Para persistir um objeto utilizando Hibernate são necessários poucos passos, declara-se uma *session*, e uma transação, abre-se a transação o método é executado e transação é encerrada. O método *merge* recebe o objeto e o salva no banco, caso o objeto já esteja cadastrado é feita a atualização dos dados, veja Figura 21.

```
    EntityManager em = EntityManagerUtil.getEntityManager();
    EntityTransaction trasaction = em.getTransaction();

    try {
        trasaction.begin();
        em.merge(o);
        trasaction.commit();
```

Figura 19 – Salvando e Atualizando uma classe no cenário ORM

Para salvar e atualizar uma classe, o Hibernate é mais atraente por salvar diretamente a classe, sem a necessidade de criação de SQL e atribuição de parâmetros. O código-fonte tem menor número de linhas e é necessário apenas um método para salvar e atualizar.

#### 7.2.3.2 Recuperando Lista de Dados

O procedimento para buscar uma lista de objetos salvos no banco usando o JDBC puro é abrir uma conexão, criar um SQL especificando as tuplas a serem recuperadas, atribuir parâmetros, percorrer o *resultSet* até o ultimo registro utilizando uma estrutura de repetição, enquanto a estrutura é percorrida o objeto recuperado é adicionado a lista (Figura 22).

```
        List<Produto> produtos = new ArrayList<Produto>();
        Connection c = Conexao.criarInstancia();

        try {
            String sql = " SELECT produto_id, descricao, nome,
                                 quantidade_disponivel, valor_unitario,
                                 categoria_id"
                    + " FROM produto"
                    + " ORDER BY nome";

            PreparedStatement ps = c.prepareStatement(sql);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                Produto produto = new Produto();
                produto.setId(rs.getInt(1));
                produto.setDescricao(rs.getString(2));
                produto.setNome(rs.getString(3));
                produto.setQuantidadeDisponivel(rs.getInt(4));
                produto.setValorUnitario(rs.getDouble(5));
                produto.setCategoria(new Categoria());
                produto.getCategoria().setId(rs.getInt(6));
                produtos.add(produto);
            }
            rs.close();
            ps.close();
```

Figura 20 – Recuperando uma lista de objeto no cenário JDBC

Para a recuperação dos dados com Hibernate existem 3 opções, o uso de *criteria*[9], de HQL e SQL. Nessa pesquisa optou-se pelo uso do HQL, que é uma linguagem de consulta aos dados baseada em SQL, porém é orientada a objetos. Ao usá-la referenciam-se as classes do programa e não as tabelas do banco, com no caso acima.

[9] API Orientada a Objetos que permite consultas dinâmicas.

Para buscar a lista são seguidos os seguintes passo, uma *session* é declarada, uma *query* é criada e quando executada já retorna uma lista do objetos (Figura 22).

```
EntityManager em = EntityManagerUtil.getEntityManager();
List<ResultProdutoItem> lista = null;
try {
    Query query = em.createQuery("Select p.produto.id, p.produto.nome,
              p.produto.valorUnitario, sum(p.quantidade) as soma "
            + " from PedidoItem p "
            + " group by p.produto.id, p.produto.nome,
              p.produto.valorUnitario "
            + " order by soma ");

    lista = ResultProdutoItem.toList(query.getResultList());
  }
```

Figura 21 – Recuperando uma lista de objetos no cenário ORM

Mais uma vez o ORM obteve o melhor código, apesar de também ter que criar uma *query* HQL, está quando executada já retornava a lista pronta, ao contrario do que acontece com o JDBC que precisa percorrer uma estrutura de repetição referenciando os atributos pela ordem declarada na *query* e montando o objeto e a lista.

#### 7.2.3.3 Buscando um Dado pela Chave

Para recuperar um registro específico por seu código de identificação, escreve-se o nome de cada campo e seguidamente na cláusula "*where*" especifica-se o campo do código, logo após a execução dos métodos é necessário ler os dados da tupla um a um, sendo que os campos devem ser referenciados numericamente conforme foram citados na clausula "*select*", veja Figura 22, nela podemos ver que no "*select"* o primeiro campo chamado é "descrição", logo, a leitura desse campo vai ser feito com o "*getString(1)*", sendo que 1 é a referencia em relação ao SQL escrito ao campo "descrição".

```
public Produto getProduto(Integer codigo) {
      Produto produto = new Produto();
      produto.setId(codigo);

      Connection c = Conexao.criarInstancia();

      try {
          String sql = "SELECT descricao, nome, quantidade_disponivel,
                        valor_unitario, categoria_id"
                  + " FROM produto"
                  + " WHERE produto_id = ?"

          PreparedStatement ps = c.prepareStatement(sql);

          ps.setInt(1, codigo);

          ResultSet rs = ps.executeQuery();
          if (rs.next()) {

              produto.setDescricao(rs.getString(1));
              produto.setNome(rs.getString(2));
              produto.setQuantidadeDisponivel(rs.getInt(3));
              produto.setValorUnitario(rs.getDouble(4));
              produto.setCategoria(new Categoria());
              produto.getCategoria().setId(rs.getInt(5));

          }
          rs.close();
          ps.close();
```

Figura 22 - Buscando objeto específico no cenário JDBC

Usando Hibernate também é preciso escrever uma *query* para a busca, porém usando HQL. O "*where*" também está presente e o código é passado por meio de parâmetro referenciando no nome do campo. Ao executar a *query* o objetos já é retornado pronto (Figura 23).

```
      EntityManager em = EntityManagerUtil.getEntityManager();
      Produto o = null;
      try {
          Query query = em.createQuery("Select o from Produto o "
                                + "where o.id = :idParam");
          query.setParameter("idParam", codigo);
          o = (Produto) query.getSingleResult();
```

Figura 23 - Buscando objeto específico no cenário ORM

Assim como no caso anterior o ORM obteve o código menor e mais legível em relação ao JDBC.

#### 7.2.3.4 Deletando uma Linha

Apagar uma tupla na tabela é simples, tanto no cenário JDBC, quando no cenário ORM. Observe as Figuras 23 e 24. Mesmo nesse caso o ORM demonstra vantagem, no que diz respeito ao tamanho do código-fonte.

```
Connection c = Conexao.criarInstancia();

try {
    String sql = "DELETE FROM produto"
            + " WHERE (produto_id = ?);";

    PreparedStatement ps = c.prepareStatement(sql);
    ps.setInt(1, o.getId());

    ps.execute();
    ps.close();
```

Figura 24 – Excluindo uma trupla no cenário JDBC

```
EntityManager em = EntityManagerUtil.getEntityManager();
EntityTransaction transaction = em.getTransaction();
try {
    transaction.begin();
    o = em.merge(o);
    em.remove(o);
```

Figura 25 – Excluindo uma tupla no cenário ORM

#### 7.2.3.6 Persistindo Herança

Um dos recursos de OO é o uso de herança, por isso implementamos seu uso em nosso protótipo. Vamos demonstrar como a classe Fornecedor, herdada de Pessoa foi persistida. Para salvar um objeto do tipo Fornecedor com JDBC temos que abrir uma

transação, e dentro dela precisamos primeiramente salvar os atributos de pessoa na tabela correspondente, depois precisamos saber o número da chave primaria dessa tabela, depois disso podemos salvar os dados da tabela fornecedor usando a chave recuperada como chave primária (Figura 28).

```
Connection c = Conexao.criarInstancia();

        try {
            c.setAutoCommit(false);
            //salvar pessoa (classe mãe)
            String sql = "INSERT INTO pessoa("
                    + " cep, ddd, fone, nome, cidade_id, endereco)"
                    + " VALUES (?, ?, ?, ?, ?, ?);";

            PreparedStatement ps = c.prepareStatement(sql);

            ps.setString(1, o.getCep());
            ps.setString(2, o.getDdd());
            ps.setString(3, o.getFone());
            ps.setString(4, o.getNome());
            ps.setInt(5, o.getCidade().getId());
            ps.setString(6, o.getEndereco());

            ps.execute();

            //pegar id do ultimo registro cadastrado
            sql = "SELECT max(pessoa_id) "
                    + "FROM pessoa";
            ps = c.prepareStatement(sql);

            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                o.setId(rs.getInt(1));
            }

            //salvar fornecedor (classe filha)
            sql = "INSERT INTO fornecedor("
                    + "fornecedor_id, cnpj, ie, ramo)"
                    + " VALUES (?, ?, ?, ?);";

            ps = c.prepareStatement(sql);

            ps.setInt(1, o.getId());
            ps.setString(2, o.getCnpj());
            ps.setString(3, o.getIe());
            ps.setString(4, o.getRamo());

            ps.execute();

            c.commit();
            ps.close();
```

Figura 26 - Persistindo objetos herdados no cenário JDBC

Usando Hibernate o procedimento para salvar uma classe herdada é o mesmo usado para salvar uma classe simples (Figura 29).

```
        EntityManager em = EntityManagerUtil.getEntityManager();
        EntityTransaction transaction = em.getTransaction();

        try {
            transaction.begin();
            em.merge(o);
            transaction.commit();
```

Figura 27 – Persistindo objetos herdados no cenário ORM

7.2.3.7 Consultas com Funções de Agregação

Agregação é um recurso oferecido pelo SQL, as funções de agregação agrupam valores de acordo com alguns campos e retornam um valor baseado no conjunto de valores dos campos agregados, como a soma ou a média de um campo da tabela. Em nosso cenário criamos um relatório a fim de testar esse recurso. Nos dois protótipos foi necessária a criação de uma *query* e ambas apresentam os mesmo modo de usar a função "*sum*[10]". Como já ocorrido anteriormente, o cenário ORM apresentar vantagem por usar o HQL, que é OO e devolver a classe inteira, observe as Figuras 30 e 31.

```
    public List<ResultProdutoItem> getProdutosMaisVendidos() {

        List<ResultProdutoItem> resultados = new ArrayList<ResultProdutoItem>();
        Connection c = Conexao.criarInstancia();

        try {
            String sql = " SELECT p.produto_id as produto_id, p.nome as nome, "
                    + " p.valor_unitario as valorUnitario, "
                    + " sum(pi.quantidade) as soma "
                    + " FROM pedido_item as pi INNER JOIN produto as p ON
                    + " pi.produto_id = p.produto_id  "
                    + " GROUP BY p.produto_id, nome, valorUnitario  "
```

[10] Função SQL básica que faz somatório de um campo numérico

```
                + " ORDER BY soma";

        PreparedStatement ps = c.prepareStatement(sql);
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            ResultProdutoItem r = new ResultProdutoItem();
            r.setIdProduto(rs.getInt(1));
            r.setNomeProduto(rs.getString(2));
            r.setQuantidade(rs.getLong(3));
            r.setValor(rs.getDouble(4));

            resultados.add(r);
        }
        rs.close();
        ps.close();
```

Figura 28 - Usando funções de agregação no cenário JDBC

```
    EntityManager em = EntityManagerUtil.getEntityManager();
    List<ResultProdutoItem> lista = null;
    try {
        Query query = em.createQuery("Select p.produto.id, p.produto.nome,
                + " p.produto.valorUnitario, sum(p.quantidade) as soma "
                + " from PedidoItem p "
                + " group by p.produto.id, p.produto.nome,
                + " p.produto.valorUnitario "
                + " order by soma ");

        lista = ResultProdutoItem.toList(query.getResultList());
```

Figura 29 - Usando funções de agregação no cenário ORM

Em todas as situações testadas o ORM apresentou vantagem sobre o JDBC, apesar de que em alguns casos as duas soluções apresentavam resultados muito próximos. Vale destacar que o ORM precisa de um mapeamento prévio nas classes e o JDBC, o desenvolvedor tem que implementar isso na hora da criação do SQL. Lembrando que no ORM é preciso ter cuidado com os relacionamentos entre as classes, caso elas usem o "*fetch*" e não o "*lazy load*", pois caso se use o "fetch" as classes associadas serão carregadas junto, o que em alguns casos pode sobrecarregar a memória com dados que não serão imediatamente utilizados.

## 7.2 Análise do Tempo

A fim de medir a diferença no tempo de acesso entre JDBC o ORM, foram feitos testes com carga de dados, o objetivos foram comparar as operações CRUD (Create, Read, Update e Delete), que são contemplados pelos métodos demonstrador a seguir, tais como Adicionar, Editar, Recuperar com e sem filtro e Excluir .Os testes foram feitos respectivamente com mil e dez mil operações consecutivas nos cenários JDBC e ORM.

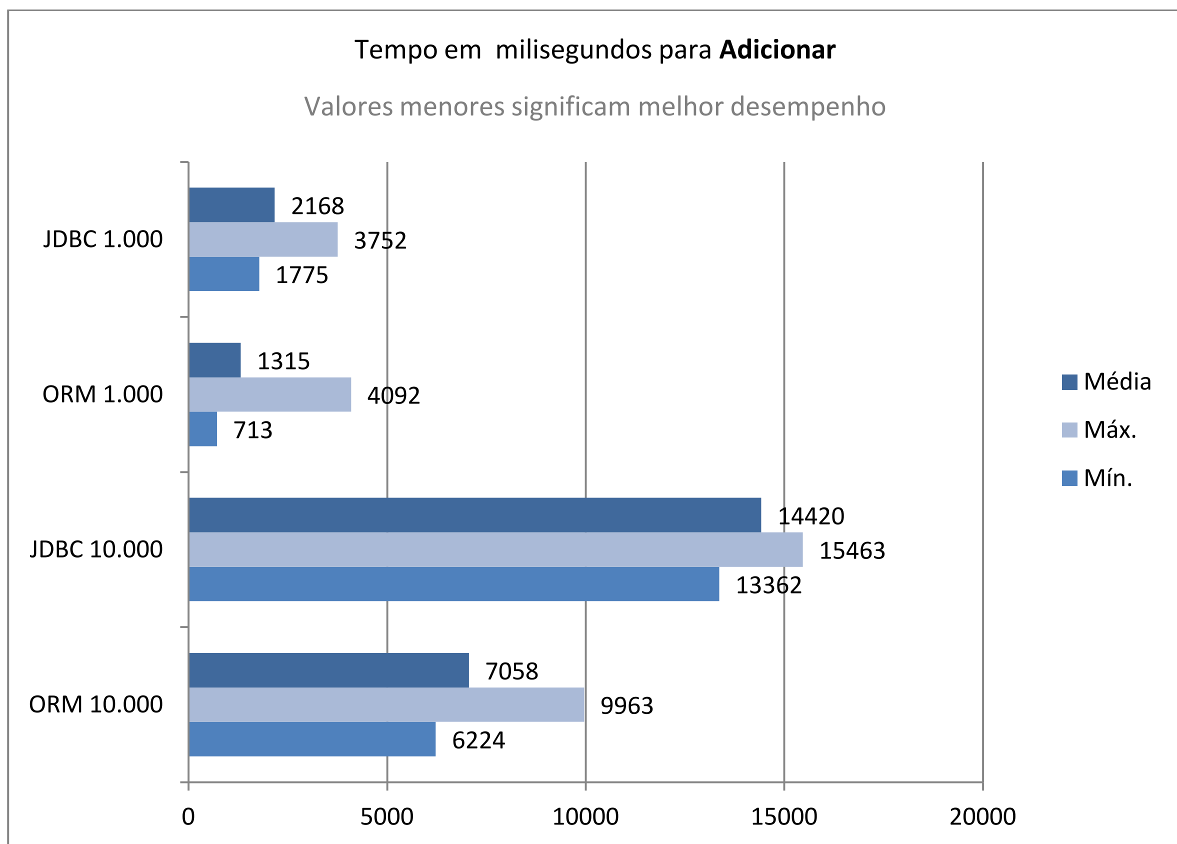


Figura 30 – Teste do tempo para adicionar objetos no SGBD

Como apresentado na Figura 32, o Hibernate concluiu a adição dos dados no banco mais rápido do que com o uso do JDBC, contrariando o que se esperava, já que com o uso do ORM ainda se usa o JDBC, o ORM cria uma nova camada na aplicação. Ao avaliar onde estaria o gargalo do JDBC, percebeu que a cada método chamado era necessário

conectar-se e desconectar-se do SGBD, e a conexão ao banco era mais demorada usando o JDBC puro do que o ORM criar a *session* que contem a conexão.

Também foi observado que a primeira execução do Hibernate era mais demorada, e as outras execuções seguidas eram mais performáticas. Isso acontece pois na primeira execução é criado o *sessionFactory*, que é um objetos pesado que gerencia a criação das *sessions*.

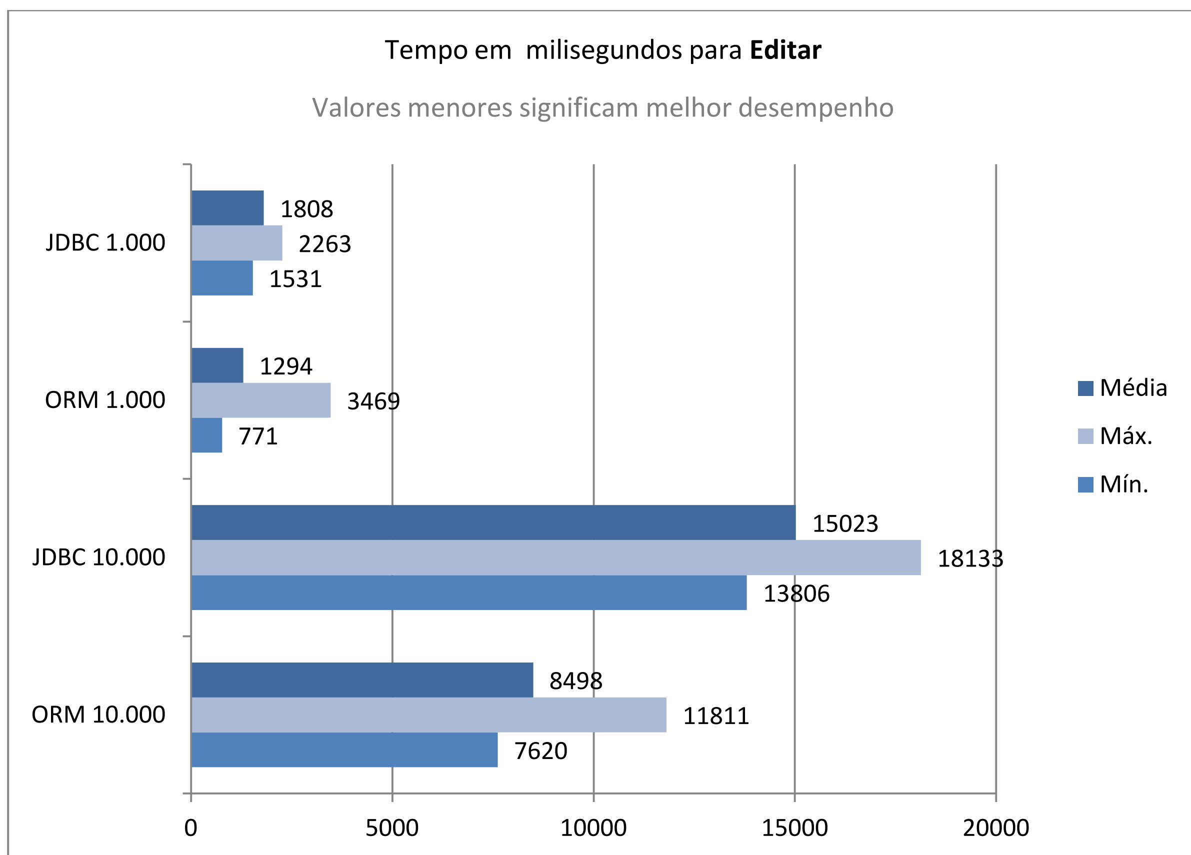

Figura 31 – Teste do tempo gasto para editar os objetos.

Na edição dos dados o Hibernate continuou com a menor média de tempo na execução (Figura 33). Percebeu-se que o tempo de execução da edição é muito próximo ao tempo de adição.

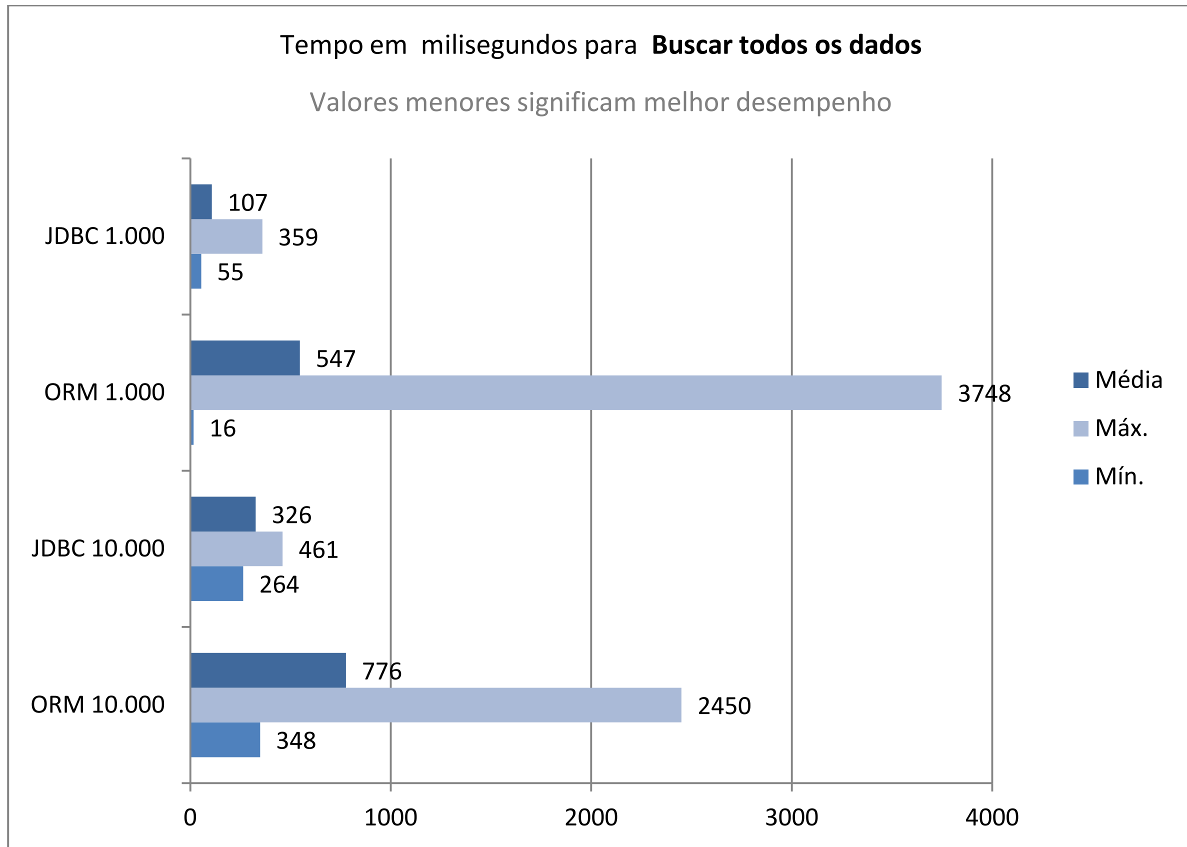


Figura 32 – Teste de tempo ao buscar lista de dados.

Para buscar os dados sem qualquer tipo de filtro o JDBC obteve a melhor média (Figura 34). Nesse caso o ORM teve a maior diferença entre o maior e menor tempo, fazendo com que sua média fosse maior.

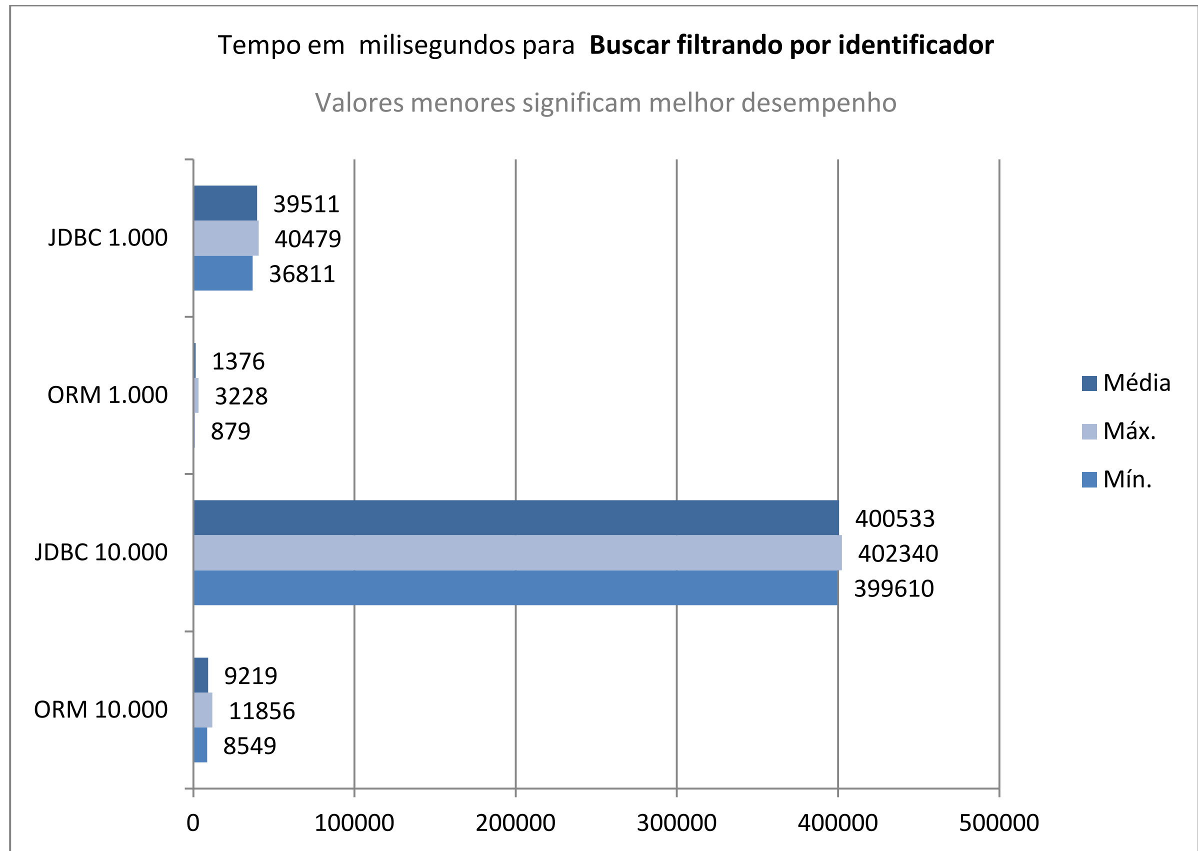


Figura 33 – Teste de tempo para buscas dados utilizando a filtro.

Como pode ser visto na Figura 35, para recuperar um objeto por seu identificador, utilizando a cláusula *where* o ORM obteve o menor tempo. Já o JDBC teve o pior desempenho dentre as operações testadas e a maior diferença de tempo em relação ao Hibernate.

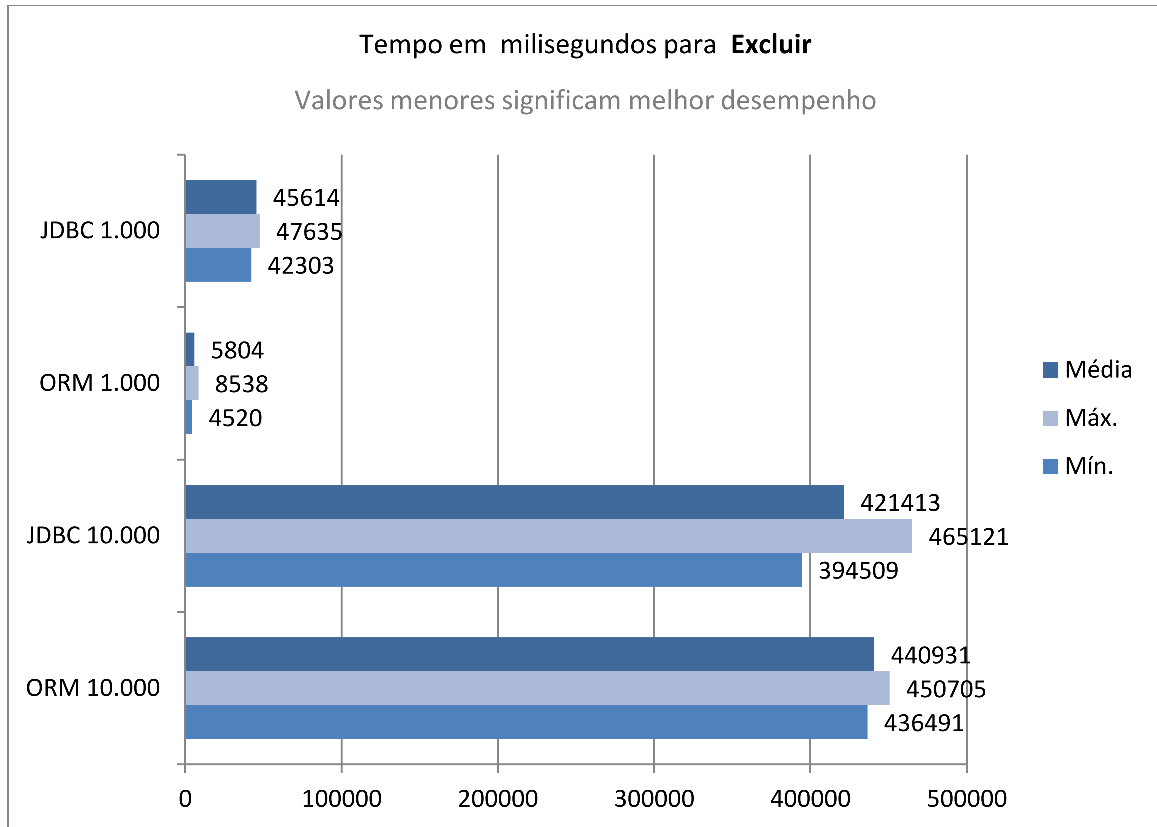


Figura 34 - Teste de tempo para excluir objetos

Para excluir o ORM obteve o menor tempo de execução com mil dados. Já com 10 mil dados os tempos foram parecidos, porém o JDBC puro foi mais rápido do que o Hibernate (Figura 36).

Tanto o JDBC, como o ORM, apresentam tempos satisfatórios para adicionar e editar os dados, sendo que seus tempos de acesso foram semelhantes, com o ORM sendo um pouco mais rápido em ambos. Para buscar dados sem uso da cláusula *where*, o JDBC obteve o melhor tempo. Para buscar os objetos filtrando por seu identificados o ORM obteve o menor tempo, sendo que nesse casso a diferença entre os dois foi maior. Para excluir o ORM foi mais rápido com mil dados, e o JDBC foi melhor com 10 mil.

Por fim apresentamos um gráfico com a média geral das operações testadas. Na média geral o ORM apresentou o melhor tempo (Figura 35).

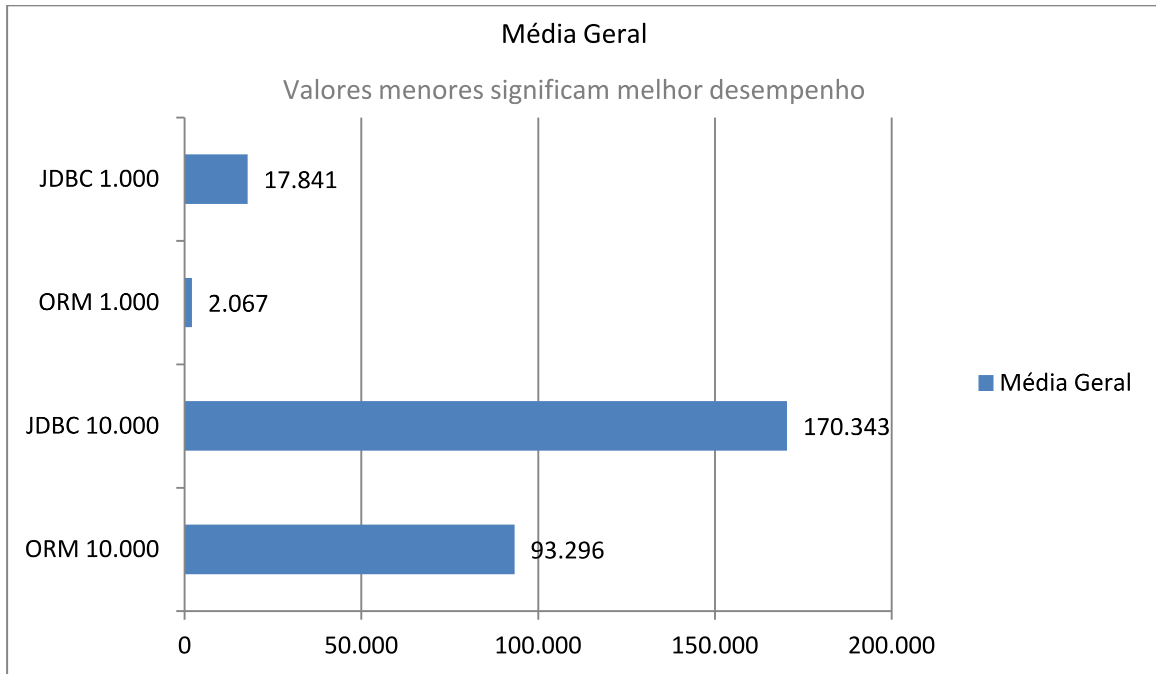


Figura 35 - Media geral dos métodos testados

# CONCLUSÃO

Nessa pesquisa, foi realizada a comparação entre JDBC puro e Hibernate, dois métodos de persistir os objetos em uma base de dados relacional. Para realizar a comparação partiu-se da pesquisa e do desenvolvimento de dois protótipos equivalentes, um para casa modo de persistência sendo que ambos diferem apenas na camada de acesso aos dados.

Após o desenvolvimento do protótipo foram analisadas as principais diferenças na codificação entre o JDBC puro e do JDBC com Hibernate. Foi possível perceber que para a criação do projetos, no que diz respeito a parte estrutural, o JDBC é mais simples do que o ORM, pois não necessita de arquivo adicionais, nem mapeamento. Porém, para manipular os objetos na base de dados o ORM precisa de menor quantidade de linhas de código, e trabalha diretamente com os objetos, já com o JDBC puro é preciso trabalhar com os campos do SGBD, sendo necessário conhecer a estrutura da base de dados.

Na comparação de tempo de execução o ORM obteve a melhor média global, contrariando o que se esperava, pois ao usá-la não deixamos de usar o JDBC, mas criamos uma nova camada no aplicativo. Vale destacar que no primeiro acesso o Hibernate tem o maior tempo, porem o tempo dos acessos seguintes é reduzido consideravelmente. O JDBC, por sua vez é mais constante em todos os acessos.

Nos cenários criados o ORM mostrou-se mais vantajoso, pois trabalhando com objetos torna o mapeamento da linguagem OO para o SGBD mais transparente e rápido, principalmente quando se utiliza recurso a linguagem como Herança. O código-fonte ficou menor e mais legível, facilitando futuras manutenções. E o tempo de acesso geral também foi reduzido.

O Hibernate ofereceu uma redução de código significativa, podemos afirmar que com o seu uso há um ganho de produtividade, onde o desenvolvedor pode se concentra na

lógica da aplicação, e não na criação de SQL. Um fator relevante é que os desenvolvedores devem conhecer bem a ferramenta, pois seu mau uso pode causar muitos erros, o que diminuiria a produtividade. Conforme King (2010), o Hibernate não é indicado para consultas muito complexas, nessa pesquisa não foi possível testar essa afirmação, pois o protótipo tem um tamanho pequeno e essas consultas são geralmente encontradas em aplicações reais de grande porte.

Na fase de desenvolvimento do protótipo teve a dificuldade de aprender a criar o projeto usando o Hibernate, por precisar de arquivos adicionais e mapeamento. Também saber quais bibliotecas adicionar ao projeto. Essas dificuldades foram facilmente superadas, pois há uma vasta documentação disponível, além de desenvolvedores ativos em fóruns de usuários.

Os objetivos específicos e geral foram alcançados com sucesso e os resultados documentados podem nortear na decisão do modo de persistir do projeto.

Como sugestão para futuros trabalhos e continuidade nessa pesquisa sugere-se.

a) comparação em uma aplicação legada;

b) uso de outros modos de persistir;

c) persistência em linguagem orientada a aspectos;

# REFERÊNCIAS

BROGNOLI, Samuel Nicolau. **Sistema de Apoio a Engenharia Reversa de Bancos de Dados Relacionais por Meio da Extração de Metadados**. 2008. 86 f. Monografia (Bacharelado) - Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense, Criciúma, 2008. Disponível em: <http://www.kiron.unesc.net/tcc/?id=606&proj=182>. Acesso em: 18 de maio 2010.

COELHO, Camila Arnellas; SARTORELLI, Reinaldo Coelho.**Persistência de Objetos Via Mapeamento Objeto-Relacional**. 2004. 110 f. Monografia (Bacharel em Sistemas de Informação) - Curso de Sistemas de Informação, Universidade Federal de Santa Maria, São Paulo, 2004. Disponível em: <http://www.dominiopublico.gov.br/pesquisa/DetalheObraForm.do?select_action=&co_obra=86596>. Acesso em: 18 de maio 2010.

DOWNEY, Tim. **Web Development With Java:**Using Hbernate, JSPs, and Servlets. Miami: Springer, 2007.

DATE, C. J. **Introdução a sistemas de bancos de dados**. 7. ed. Rio de Janeiro: Ed. Campus, 2000.

HAHN, Prissila Gomes. **Transformação de Esquemas de Bancos de Dados Relacionais em Orientados a Objetos**. 2009. 82 f. Monografia (Bacharelado) - Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense, Criciúma, 2009. Disponível em: <http://www.kiron.unesc.net/tcc/?id=606&proj=222>. Acesso em: 18 maio 2010.

HIBERNATE; Hibernate.org. Disponível em: http://www.hibernate.org/ Acesso em: 24/05/2010.

KING, K et al **Documentação de Referência Hibernate.** 2010. Disponível em: <http://docs.jboss.org/hibernate/core/3.6/reference/pt-BR/pdf/>. Acesso em: 11 nov. 2010

LOPES, Roberto Baselio. **Estudo Comparativo de Implementação de Modelos de Banco de Dados**. 2007. 47 f. Monografia (Bacharelado) - Curso de Ciência da Computação, Faculdade de Jaguariúna, Jaguariúna, 2008.

MACHADO, Felipe; ABREU, Mauricio**. Projeto de Banco de Dados**: Uma Visão Prática. 11. ed. São Paulo: Érica, 2004.

MANENTE, Rogério. **Avaliação da ferramenta Hibernate em um ambiente de produção de larga escala**. 2007. 27 f. Monografia (Bacharel e Ciência da Computação) - Curso de Ciência da Computação, Universidade Federal de Santa Maria, São Paulo, 2007. Disponível em: <http://www.ime.usp.br/~cef/mac499-07/monografias/rogerio/monografia.pdf>. Acesso em: 18 maio 2010.

NASCIMENTO, Alexandre. **Modelagem Orientada a Objetos Utilizando UML para Sistemas de Manutenção Industrial**. 2003. 100f. Monografia (Bacharelado) – Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense, Criciúma, 2003. Disponível em: <WWW.colocar o link> Acesso em: 11 de outubro de 2010.

ORG.HIBERNATE Interface Session. 2010. Disponível em: <http://docs.jboss.org/hibernate/stable/core/api/org/hibernate/Session.html>. Acesso em: 08 nov. 2010.

PEAK, Patrick; HEUDECHER, Nick. **Hibernate Quickly.** Greenwich: Manning, 2005.

PINHEIRO, José Francisco Viana. **Um Framework Para Persistência de Objetos em Banco de Dados Relacional**. 2005. 72 f. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Santa Maria, Niterói, 2005. Disponível em: <http://www.midiacom.uff.br/downloads/pdf/pinheiro_2005.pdf>. Acesso em: 18 maio 2010.

PLÁCIDO, Daniel Guessi. **Impedância de Dados em Banco de Dados**. 2008. 86 f. Monografia (Bacharelado) - Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense, Criciúma, 2008. Disponível em: <http://www.kiron.unesc.net/tcc/?id=606&proj=162>. Acesso em: 18 maio 2010.

REIS, Adailton Pimentel.**Programação orientada a aspectos com persistência no desenvolvimento de sistemas.** 2007. 51 f. Monografia (Bacharel em Análise de Sistemas) - Curso de Análise de Sistemas, Faculdade de Ciências Exatas e Tecnológicas, Arapiraca, 2007. Disponível em: <http://www.ebah.com.br/tcc-sobre-programaaao-orientada-a-aspectos-hibernate-pdf-a65134.html>. Acesso em: 11 nov. 2010.

RICARTE, Ivan Luiz Marques. **Programação Orientada a Objetos**: Uma Abordagem com Java. 2001. Unicamp, Campinas, 2001. Disponível em: <http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf>. Acesso em: 12 nov. 2010.

RUMBAUGH, James. **Modelagem e projetos baseados em objetos**. Rio de Janeiro: Ed. Campus, 1994.

SAM-BODDEN, Brian. **Beginning:** From Novice to Professional. Nove York: Apress, 2006.

SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java.** Rio de Janeiro: Elsevier, 2003.

MARTINS, M. R; SANTOS, D. V. **Métricas para avaliação das alternativas de persistência de dados num ambiente objeto relacional.** 2007. 63 f. Monografia (Bacharelado) - Usp, São Paulo, 2007. Disponível em: <http://www.ime.usp.br/~cef/mac499-07/monografias/diogo-marcelo/monografia.pdf>. Acesso em: 12 nov. 2010.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de banco de dados**. Rio de Janeiro: Elsevier, 2006. 781 p.

SILVA, Carolina Fernanda da. **Análise e Avaliação do Framework Hibernate em uma Aplicação Cliente/Servidor**. 2007. 99 f. Monografia (Bacharelado) - Curso de Ciência da Computação, Faculdade de Jaguariúna, Jaguariúna, 2007.

VIEIRA JUNIOR, Itamar. **Protótipo De Ferramenta Case Orientada A Objetos Para Geração De Modelos Conceituais De Dados**. 2003. 84f. Monografia (Bacharelado) – Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense, Criciúma, 2003. Disponível em: <WWW.colocar o link> Acesso em: 11 de outubro de 2010.

# APÊNDICE A – SCRIPT DE CRIAÇÃO DAS TABELAS NA BASE DE DADOS

```
-- Table: categoria
-- DROP TABLE categoria;
CREATE TABLE categoria
(
  categoria_id serial NOT NULL,
  nome character varying(255),
  CONSTRAINT categoria_pkey PRIMARY KEY (categoria_id)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE categoria OWNER TO postgres;


-- Table: estado
-- DROP TABLE estado;
CREATE TABLE estado
(
  estado_id serial NOT NULL,
  nome character varying(255),
  sigla character varying(255),
  CONSTRAINT estado_pkey PRIMARY KEY (estado_id)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE estado OWNER TO postgres;



-- Table: cidade
-- DROP TABLE cidade;
CREATE TABLE cidade
(
  cidade_id serial NOT NULL,
  nome character varying(255),
  estado_id integer,
  CONSTRAINT cidade_pkey PRIMARY KEY (cidade_id),
  CONSTRAINT fk_cidade_estado_id FOREIGN KEY (estado_id)
      REFERENCES estado (estado_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE cidade OWNER TO postgres;
```

```
-- Table: pessoa
-- DROP TABLE pessoa;

CREATE TABLE pessoa
(
  pessoa_id serial NOT NULL,
  dtype character varying(31),
  cep character varying(255),
  ddd character varying(255),
  fone character varying(255),
  nome character varying(255),
  cidade_id integer,
  endereco character varying(255),
  CONSTRAINT pessoa_pkey PRIMARY KEY (pessoa_id),
  CONSTRAINT fk_pessoa_cidade_id FOREIGN KEY (cidade_id)
      REFERENCES cidade (cidade_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE pessoa OWNER TO postgres;


-- Table: cliente
-- DROP TABLE cliente;
CREATE TABLE cliente
(
  cliente_id integer NOT NULL,
  cpf character varying(255),
  datacadastro date,
  limite_credito double precision,
  rg character varying(255),
  CONSTRAINT cliente_pkey PRIMARY KEY (cliente_id),
  CONSTRAINT fk_cliente_cliente_id FOREIGN KEY (cliente_id)
      REFERENCES pessoa (pessoa_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE cliente OWNER TO postgres;
```

```
-- Table: fornecedor
-- DROP TABLE fornecedor;
CREATE TABLE fornecedor
(
  fornecedor_id integer NOT NULL,
  cnpj character varying(255),
  ie character varying(255),
  ramo character varying(255),
  CONSTRAINT fornecedor_pkey PRIMARY KEY (fornecedor_id),
  CONSTRAINT    fk_fornecedor_fornecedor_id    FOREIGN    KEY
(fornecedor_id)
      REFERENCES pessoa (pessoa_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE fornecedor OWNER TO postgres;



-- Table: produto
-- DROP TABLE produto;
CREATE TABLE produto
(
  produto_id serial NOT NULL,
  descricao character varying(255),
  nome character varying(255),
  quantidade_disponivel integer,
  valor_unitario double precision,
  categoria_id integer,
  CONSTRAINT produto_pkey PRIMARY KEY (produto_id),
  CONSTRAINT     fk_produto_categoria_id     FOREIGN     KEY
(categoria_id)
      REFERENCES categoria (categoria_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE produto OWNER TO postgres;


-- Table: pedido
-- DROP TABLE pedido;
CREATE TABLE pedido
(
  pedido_id serial NOT NULL,
  data_emissao date,
  cliente_id integer,
```

```
  CONSTRAINT pedido_pkey PRIMARY KEY (pedido_id),
  CONSTRAINT fk_pedido_cliente_id FOREIGN KEY (cliente_id)
      REFERENCES pessoa (pessoa_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE pedido OWNER TO postgres;



-- Table: pedido_item
-- DROP TABLE pedido_item;
CREATE TABLE pedido_item
(
  pedido_item_id serial NOT NULL,
  quantidade integer,
  produto_id integer,
  item_id integer,
  CONSTRAINT pedido_item_pkey PRIMARY KEY (pedido_item_id),
  CONSTRAINT fk_pedido_item_item_id FOREIGN KEY (item_id)
      REFERENCES pedido (pedido_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT     fk_pedido_item_produto_id     FOREIGN     KEY
(produto_id)
      REFERENCES produto (produto_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
ALTER TABLE pedido_item OWNER TO postgres;



-- Sequence: categoria_categoria_id_seq
-- DROP SEQUENCE categoria_categoria_id_seq;
CREATE SEQUENCE categoria_categoria_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE categoria_categoria_id_seq OWNER TO postgres;



-- Sequence: cidade_cidade_id_seq
-- DROP SEQUENCE cidade_cidade_id_seq;
```

```
CREATE SEQUENCE cidade_cidade_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE cidade_cidade_id_seq OWNER TO postgres;


-- Sequence: estado_estado_id_seq
-- DROP SEQUENCE estado_estado_id_seq;
CREATE SEQUENCE estado_estado_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE estado_estado_id_seq OWNER TO postgres;


-- Sequence: pedido_item_pedido_item_id_seq
-- DROP SEQUENCE pedido_item_pedido_item_id_seq;
CREATE SEQUENCE pedido_item_pedido_item_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE pedido_item_pedido_item_id_seq OWNER TO postgres;



-- Sequence: pedido_pedido_id_seq
-- DROP SEQUENCE pedido_pedido_id_seq;
CREATE SEQUENCE pedido_pedido_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE pedido_pedido_id_seq OWNER TO postgres;


-- Sequence: pessoa_pessoa_id_seq
-- DROP SEQUENCE pessoa_pessoa_id_seq;
CREATE SEQUENCE pessoa_pessoa_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
```

```
ALTER TABLE pessoa_pessoa_id_seq OWNER TO postgres;


-- Sequence: produto_produto_id_seq
-- DROP SEQUENCE produto_produto_id_seq;
CREATE SEQUENCE produto_produto_id_seq
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE produto_produto_id_seq OWNER TO postgres;


-- Sequence: seg_categoria
-- DROP SEQUENCE seg_categoria;
CREATE SEQUENCE seg_categoria
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seg_categoria OWNER TO postgres;


-- Sequence: seg_cidade
-- DROP SEQUENCE seg_cidade;
CREATE SEQUENCE seg_cidade
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seg_cidade OWNER TO postgres;


-- Sequence: seg_estado
-- DROP SEQUENCE seg_estado;
CREATE SEQUENCE seg_estado
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seg_estado OWNER TO postgres;


-- Sequence: seg_pedido
-- DROP SEQUENCE seg_pedido;

CREATE SEQUENCE seg_pedido
```

```
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seg_pedido OWNER TO postgres;


-- Sequence: seg_pedido_item
-- DROP SEQUENCE seg_pedido_item;
CREATE SEQUENCE seg_pedido_item
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seg_pedido_item OWNER TO postgres;


-- Sequence: seg_produto
-- DROP SEQUENCE seg_produto;
CREATE SEQUENCE seg_produto
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seg_produto OWNER TO postgres;


-- Sequence: seq_gen_sequence
-- DROP SEQUENCE seq_gen_sequence;
CREATE SEQUENCE seq_gen_sequence
  INCREMENT 50
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE seq_gen_sequence OWNER TO postgres;
```

A

# Comparação Entre o uso do JDBC e do Hibernate para a Persistência de Dados

**Cristina da Silva Matos[1], Paulo João Martins[2]**

[1]Acadêmica do curso de Ciência da Computação – Unidade Acadêmica de Ciências, Engenharias e Tecnologias - Universidade do Extremo Sul Catarinense (UNESC) – Criciúma, SC – Brasil

[2]Professor do curso de Ciência da Computação - Unidade Acadêmica de Ciências,

Engenharias e Tecnologias - Universidade do Extremo Sul Catarinense (UNESC) – Criciúma,

SC – Brasil

cristina.silvamatos@hotmail.com,pjm@unesc.net

***Abstract.*** *Este artigo apresenta uma comparação entre dois modos de manipular as bases relacionais em uma linguagem Orientada a Objetos, uma utilizando JDBC puro e outro utilizando a ferramenta Hibernate. For this, were done researches about main features of relational databases and about the principles of OO programming. Also was studied the Hibernate. After the search, two prototypes were built for testing, an application with an object-oriented relational databases using JDBC and the other using Hibernate. The comparison was performed by evaluating the prototypes were considered factors such as ease of coding, readability, size of the source code and access time. The results show that the ORM is an efficient way of solving the impedance data, facilitating the development without losing performance.*

***Resumo.*** *Este artigo apresenta uma comparação entre dois modos de manipular as bases relacionais em uma linguagem Orientada a Objetos, uma utilizando JDBC puro e outro utilizando a ferramenta Hibernate. Para isso foram feitas pesquisas sobre as principais características dos bancos de dados relacionais, sobre os princípios da programação OO. Também foi estudada a ferramenta Hibernate. Após a pesquisa foram construídos dois protótipos para a realização dos testes. A comparação foi realizada por meio da avaliação dos protótipos desenvolvidos, foram considerados fatores como facilidade de codificação, legibilidade, tamanho do código-fonte e tempo de acesso. Os resultados demonstram que o ORM é uma maneira eficiente de resolver a impedância de dados, facilitando o desenvolvimento sem perder o desempenho.*

## 1. INTRODUÇÃO

Com o uso da informática para as mais diversas finalidades, a era da informação fica evidente. As informações são consideradas mais importantes, e ao mesmo tempo mais fáceis e baratas

de se obter. Nesse contexto os bancos de dados se tornaram importantes pilares para o armazenamento dessas informações (SILVA, 2007).

Diante dessa realidade surgem vários métodos de implantação de persistência de dados, sendo, os mais empregados atualmente os Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDR's) (LOPES, 2007).

Para facilitar a integração entre as linguagens de programação e os bancos de dados, sugiram camadas de persistência que realizam a comunicação por meio de drivers, tornando-os possíveis de serem aplicados a qualquer linguagem de programação (LOPES, 2007).

Contudo, existem diferenças significativas do paradigma de programação OO para o modelo de banco de dados relacional. Para que seja possível obter os benefícios de ambos, é necessária uma compatibilidade entre esses modelos. Uma das propostas para diminuir essas diferenças é o Mapeamento Objeto Relacional (ORM), que pretende diminuir o tempo de desenvolvimento, reduzir o custo geral do sistema e proporcionar um código de fácil manutenção. Essa técnica consiste na criação de uma camada na aplicação que mapeia objetos em tuplas, no banco de dados relacional, tornando transparente a persistência dos objetos (HIBERNATE, 2010).

Segundo Manente (2007) a ferramenta ORM mais utilizada é o Hibernate, que é um framework responsável pela camada de mapeamento objeto relacional usada para armazenar objetos Java em uma base de dados relacional. Por meio dessa camada, ele permite o desenvolvimento de classes orientada a objetos persistentes, permitindo o uso de associação, herança, polimorfismo, composição e coleções.

Nessa pesquisa iremos comparar um sistema desenvolvido em linguagem orientada a objetos que utilize o JDBC para acessar o banco de dados, comparando a mesma aplicação, porém, com o uso de uma ferramenta ORM. Para essa avaliação serão definidas métricas como tempo de acesso, desempenho.

Essa pesquisa objetiva a realização de uma análise das metodologias de persistência de dados, SGBD nativo e ORM, para definir em que momento o uso do Hibernate oferece vantagens e quando ele adiciona complexidade ao projeto. Serão considerando fatores como, operações CRUD, e alguns dos principais recursos dos bancos de dados como chaves primárias e estrangeiras.

Com essa comparação pretende-se entender como esses modos de acesso afetam o desempenho, o tempo de implementação e a facilidade de manutenção da aplicação, demarcando quais as limitações, vantagens e os recursos oferecidos por cada um dos métodos utilizados na persistência de dados em cenários orientados a objetos. Facilitando para que haja uma escolha dos recursos de persistência de uma maneira consciente que satisfaça as necessidades do projeto à qual está sendo utilizado.

## 2 BANCO DE DADOS

Banco de Dados é um sistema computadorizado de armazenamento de registros inter-relacionados, cujo propósito é armazenar informações de um domínio específico, e por meio de um conjunto de programas permitirem ao usuário buscar e atualizar essas informações quando for solicitado (DATE, 2000; SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

O banco de dados refere-se apenas aos dados, o sistema responsável por controlar os dados é denominado Sistema Gerenciador de Banco de Dados (SGBD). Conforme Silberschatz (1999, p. 01), um SGBD é "constituído por um conjunto de dados associados e um conjunto de programas que fornece o acesso a esses dados", ou seja, uma coleção de

programas que permite criar estruturas, manter dados e gerenciar as transações efetuadas, além de permitir a extração das informações de maneira rápida e segura (SILVA, 2007).

Os SGBD's utilizam diferentes formas de representação ou modelagem de dados para descrever a estrutura das informações contidas em seus bancos de dados. Atualmente alguns dos modelos existentes são: modelo hierárquico, modelo em redes, modelo relacional e o modelo orientado a objetos. Entre esses modelos, o mais utilizado é o modelo relacional.

### 2.1 BANCO DE DADOS RELACIONAL

Os bancos de dados relacionais baseiam-se no princípio de que os dados podem ser considerados como relações matemáticas e que estão representados de maneira uniforme, com o uso de tabelas bidimensionais. "O conceito principal vem da teoria dos conjuntos (álgebra relacional) atrelado à idéia de que não é relevante ao usuário saber onde os dados estão nem como os dados estão (Transparência)" (Machado; Abreu, 2004, p. 182).

Os bancos de dados são o modo mais adequado de salvar os dados de uma aplicação que então na memória. O modelo mais utilizado atualmente é o relacional que, além de maduro oferece muitas opções de gerenciadores. Esses modelos são utilizados no desenvolvimento dos sistemas juntamente com linguagem orientada a objetos.

## 3 PARADIGMA DE PROGRAMAÇÃO ORIENTADO A OBJETOS

Segundo Vieira (2003) a POO permite uma maior abstração de sistema por parte de projetista e facilitou a "tradução" do sistema real em um sistema computacional. (SANTOS; MARTINS, 2007).

Enquanto na programação procedural toda a lógica da aplicação é baseada em chamadas seqüenciais de funções que agem sobre os dados. A modelagem OO parte de premissa que todo o objeto da vida real possui atributos e comportamentos. As aplicações são constituídas por objetos que representam entidades do mundo real e que cooperam entre si para realizar as tarefas solicitadas. Onde cada objeto processa seus próprios dados e devolve o resultado por de troca de mensagens (MANENTE, 2007; VIEIRA, 2003).

Por ser um paradigma diferente, as linguagens OO apresentam novos conceitos em relação às linguagens relacionais, tais como classes, objetos, encapsulamento, herança e polimorfismo. Iremos descrever sobre cada um desses conceitos.

- CLASSE: Representa o conceito geral de algo no mundo real, elas especificam os objetos definindo um conjunto de características e comportamentos que um conjunto de objetos que pertence a ela pode assumir.
- OBJETOS: Os objetos são as estruturas fundamentais da abordagem OO, eles combinam a estrutura (estado, valor) e o comportamento (operações) dos dados em uma única entidade.
- ENCAPSULAMENTO: Encapsulamento está diretamente relacionado com abstração, sendo que abstrair significa considerar apenas os aspectos essenciais a uma entidade, e ignorar propriedades menos importantes.
- MÉTODO: definem o comportamento dos objetos.
- HERANÇA: é um mecanismo de especialização de classes, que permite a uma determinada classe possuir características particulares e ainda herdar as características de outra classe (HAHN, 2009).

- POLIMORFISMO: Segundo Ricarte (2001, p.6) “polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos”.

## 4 MAPEAMENTO OBJETO RELACIONAL

O mapeamento objeto relacional (ORM) tem como objetivo “ligar” ambientes que utilizam de um lado o paradigma de orientação a objeto, e do outro lado o modelo relacional, a Figura 1 ilustra essa situação.

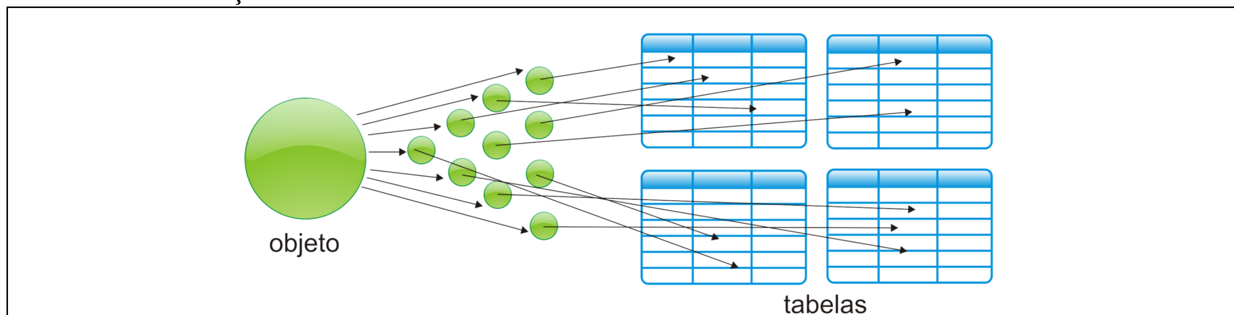


**Figura 1 – Mapeamento Objeto-Relacional**

O significado para mapeamento objeto relacional é o mesmo tanto via JDBC manualmente, como também, para a persistência feita automaticamente com o uso de framework, pois quando é implementado um código SQL que executa um select e extrai os resultados para seus objetos, ou até mesmo, quando é persistido um objeto para dentro do banco de dados, seja qual for a forma utilizada, o mapeamento está sendo realizado (REIS, 2007).

## 5 HIBERNATE

Segundo Sam-bodden (2006, tradução nossa) é um mecanismo de persistência orientada a objetos transparente para Java, utilizado para mapear classes em tabelas de bancos de dados relacionais e vice-versa, para isso suporta herança, polimorfismo, composição, coleção e mapeamento de associação entre objetos. Além de realizar o mapeamento objeto relacional disponibiliza um mecanismo de consulta de dados.

Essa ferramenta oferece suporte à diversos bancos de dados e o mapeamento entre os objetos e as tabela podem ser definidos em um documento XML ou como o uso de anotações. O framework fica entre os objetos Java tradicionais e manipula todo o trabalho na persistência desses objetos.

Segundo Kumar (2006) apud Martins; Santos (2007) O Hibernate pode ser observado dividido em três camadas principais:

- gerenciador de conexão: gerencia de forma eficiente a conexão com o banco de dados;
- gerenciador de transações: permite ao usuário executar mais de um conjunto de operações no banco de dados;
- mapeamento objeto relacional: Faz o mapeamento por meio de recuperação, inserção, atualização e remoção dos dados nas tabelas.

Para a utilização do Hibernate é necessário que se faça um configuração prévia, segundo Peak e Heudecher (2005, tradução nossa) o Hibernate é geralmente configurado em dois passos. Primeiro se configura o Hibernate Service, com os parâmetros de conexão do banco de dados, e, a coleção das classes persistentes. Em segundo lugar, informa-se as dados

sobre as classes a serem persistidas. Essa configuração faz a ligação entre as classes e o banco de dados.

No Hibernate, o mapeamento entre objetos e tabelas pode ser definido em documentos XML, ou em código Java via anotações de persistência (SAM-BODDEN, 2006, tradução nossa).

# 7 COMPARAÇÃO ENTRE JDBC E HIBERNATE

A fim de avaliar as diferenças na codificação e desempenho nas formas de se persistir os objetos foi desenvolvido um cenário de aplicação de uma loja na web. Foram criados dois protótipos, um com o uso do JDBC e outro com a utilização do Hibernate.

Os protótipos foram desenvolvidos em linguagem Java, usando a versão 7 da JVM , na IDE de desenvolvimento NetBeans 7.0.1. O SGBD utilizado foi o PostGresSql 9.0, com o JDBC correspondente versão 4. A versão do Hibernate foi a 3.

A concepção do protótipo foi criada visando o uso dos principais recursos oferecidos pelo paradigma relacional (usado nos SGBDR's) e pelo paradigma de programação Orientado a Objetos. Foram criados requisitos que abrangessem o uso dos recursos de OO, e dos recursos dos das bases relacionais, listados na tabela 1.

**Tabela 5 – Recursos OO e Relacionais utilizados**

| **Recursos Utilizados** | |
|---|---|
| Recursos de Programação Orientada a Objetos | Recursos de Banco de Dados Relacionais |
| Classe simples | Relacionamento "one to many" e "many to one" |
| Classe com tipo complexo | Relacionamento "many to many" |
| Herança | Funções de agregação |
| Coleção | Consulta com "order by" |
| | Consulta com "inner join" |
| | Consulta com "right join" |

## 7.1 Desenvolvimento do Protótipo

Após análise das informações levantadas e da modelagem, passou-se a fase de desenvolvimento da aplicação. No decorrer do desenvolvimento foram criados os seguintes pacotes principais: "modelo", "mb" , "dao" e "Páginas Web", como pode ser observado na Figura 2. Os dois protótipos desenvolvidos apresentam exatamente os mesmos códigos-fonte nos pacotes "Páginas Web", e "mb". No pacote "modelo" as classes e atributos são os mesmos, diferindo entre o protótipo ORM e o JDBC apenas as anotações adicionadas ao protótipo ORM, responsáveis por fazer o mapeamento.

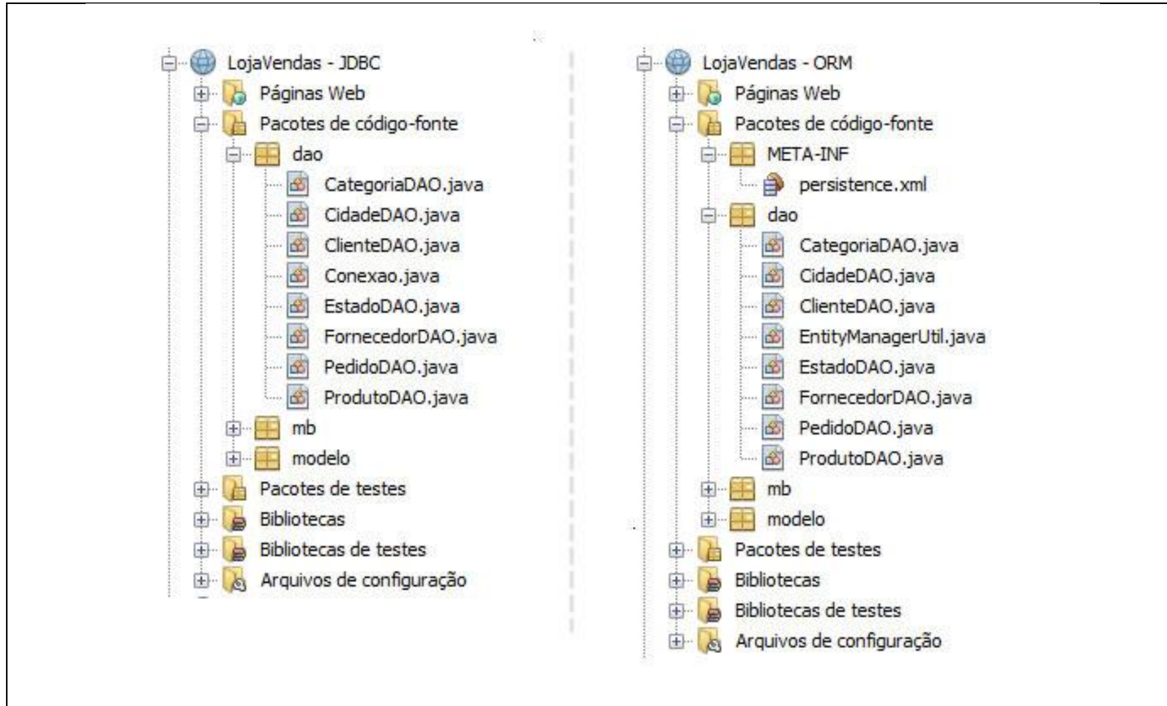


**Figura 2 – Estrutura dos cenários desenvolvidos**

A diferença mais evidente entre os dois cenários se encontra na camada “dao”, nessa camada é onde efetivamente se utiliza os recursos que serão analisados nessa pesquisa. Outra diferença entre os dois cenários são as classes “Conexão.java” e “EntityManagerUtil.java”. Onde, “Conexao.java” é utilizada no cenário JDBC, sendo responsável pela conexão ao SGBDR. E “EntityManagerUtil.java”, necessária no cenário ORM, encapsula a criação da sessionFactory, que gerencia a criação das sessions, esta, por sua vez se conecta ao banco.

No cenário ORM também é preciso criar um pacote auxiliar, como o nome “META-INF”, que tem o arquivo “persistence.xml”, que apresenta as propriedades das Entidades e mecanismos de acesso ao SGBD e configuração do ORM, para que a aplicação possa acessar a base de dados.

## 7.2 COMPARAÇÃO ENTRE AS CLASSES PERSISTENTES

A maior diferença entre os cenários é encontrada na camada de acesso aos dados, nela afetivamente fazemos uso dos recursos oferecidos pelos nossos alvos de comparação. Foram comparadas as classes equivalentes nos dois protótipos criados. Avaliando o uso dos conceitos de OO no código-gerado, observando o tamanho, legibilidade e complexidade do código-fonte necessário para as operações de manipulação dos dados armazenados do SGBDR. Nesse artigo demonstraremos a os métodos de salvar, editar e recuperar os dados.

- **Salvando e Editando os Dados**: No cenário JDBC são necessários dois métodos, um para salvar (Figura 3) e outro para atualizar (Figura 4). Com o uso do SQL é preciso que se saiba os nomes das tabelas e seus campos, conforme está no banco, necessitando de consultas ao esquema do banco, isso causa uma maior possibilidades de erros, já que os nome podem ser confundidos ou até mesmo esquecidos.

```
Connection c = Conexao.criarInstancia();

try {

String sql = "INSERT INTO produto("
```

```
        +    "descricao,    nome,    quantidade_disponivel,    valor_unitario,
categoria_id)"
        + " VALUES (?, ?, ?, ?, ?);";
            PreparedStatement ps = c.prepareStatement(sql);
            ps.setString(1, o.getDescricao());
            ps.setString(2, o.getNome());
            ps.setInt(3, o.getQuantidadeDisponivel());
            ps.setDouble(4, o.getValorUnitario());
            ps.setInt(5, o.getCategoria().getId());
            ps.execute();
            ps.close();
```

**Figura 3 – Salvando uma classe no cenário JDBC**

```
         Connection c = Conexao.criarInstancia();
        try {

            String sql = "UPDATE produto"
                    + " SET nome=?, descricao=? , quantidade_disponivel=? ,
                      valor_unitario=? , categoria_id=? "
                    + " WHERE produto_id = ?";

            PreparedStatement ps = c.prepareStatement(sql);

            ps.setString(1, o.getNome());
            ps.setString(2, o.getDescricao());
            ps.setInt(3, o.getQuantidadeDisponivel());
            ps.setDouble(4, o.getValorUnitario());
            ps.setInt(5, o.getCategoria().getId());
            ps.setInt(6, o.getId());

            ps.executeUpdate();

            ps.close();
```

**Figura 4 – Atualizando uma classe no cenário JDBC**

Para persistir um objeto utilizando Hibernate são necessários poucos passos, declara-se uma session, e uma transação, abre-se a transação o método é executado e transação é encerrada. O método merge recebe o objeto e o salva no banco, caso o objeto já esteja cadastrado é feita a atualização dos dados, veja Figura 5.

```
        EntityManager em = EntityManagerUtil.getEntityManager();

        EntityTransaction trasaction = em.getTransaction();


        try {

            trasaction.begin();

            em.merge(o);

            trasaction.commit();
```

**Figura 5 – Salvando e Atualizando uma classe no cenário ORM**

Para salvar e atualizar uma classe, o Hibernate é mais atraente por salvar diretamente a classe, sem a necessidade de criação de SQL e atribuição de parâmetros. O código-fonte tem menor número de linhas e é necessário apenas um método para salvar e atualizar.

- **Recuperando Lista de Dados:** O procedimento para buscar uma lista de objetos salvos no banco usando o JDBC puro é abrir uma conexão, criar um SQL especificando as tuplas a serem recuperadas, atribuir parâmetros, percorrer o resultSet até o ultimo registro utilizando uma estrutura de repetição, enquanto a estrutura é percorrida o objeto recuperado é adicionado a lista (Figura 6).

```
         List<Produto> produtos = new ArrayList<Produto>();

        Connection c = Conexao.criarInstancia();


        try {

            String sql = " SELECT produto_id, descricao, nome,
                                  quantidade_disponivel,
valor_unitario,
                                  categoria_id"
                    + " FROM produto"
                    + " ORDER BY nome";


            PreparedStatement ps = c.prepareStatement(sql);

            ResultSet rs = ps.executeQuery();

            while (rs.next()) {

                Produto produto = new Produto();
```

```
                produto.setId(rs.getInt(1));
                produto.setDescricao(rs.getString(2));
                produto.setNome(rs.getString(3));
                produto.setQuantidadeDisponivel(rs.getInt(4));
                produto.setValorUnitario(rs.getDouble(5));
                produto.setCategoria(new Categoria());
                produto.getCategoria().setId(rs.getInt(6));
                produtos.add(produto);
            }
            rs.close();
            ps.close();
```

**Figura 6 – Recuperando uma lista de objeto no cenário JDBC**

Para a recuperação dos dados com Hibernate existem 3 opções, o uso de criteria11, de HQL e SQL. Nessa pesquisa optou-se pelo uso do HQL, que é uma linguagem de consulta aos dados baseada em SQL, porém é orientada a objetos. Ao usá-la referenciam-se as classes do programa e não as tabelas do banco, com no caso acima. Para buscar a lista são seguidos os seguintes passo, uma session é declarada, uma query é criada e quando executada já retorna uma lista do objetos (Figura 7).

```
         EntityManager em = EntityManagerUtil.getEntityManager();
        List<ResultProdutoItem> lista = null;
        try {
            Query    query    =    em.createQuery("Select    p.produto.id,
p.produto.nome,
                      p.produto.valorUnitario, sum(p.quantidade) as soma "
                    + " from PedidoItem p "
                    + " group by p.produto.id, p.produto.nome,
                      p.produto.valorUnitario "
                    + " order by soma ");

            lista = ResultProdutoItem.toList(query.getResultList());
          }
```

**Figura 7 – Recuperando uma lista de objetos no cenário ORM**

[11] API Orientada a Objetos que permite consultas dinâmicas.

Mais uma vez o ORM obteve o melhor código, apesar de também ter que criar uma query HQL, está quando executada já retornava a lista pronta, ao contrario do que acontece com o JDBC que precisa percorrer uma estrutura de repetição referenciando os atributos pela ordem declarada na query e montando o objeto e a lista.

7.3 ANÁLISE DO TEMPO

A fim de medir a diferença no tempo de acesso entre JDBC o ORM, foram feitos testes com carga de dados, o objetivos foram comparar as operações CRUD (Create, Read, Update e Delete), que são contemplados pelos métodos demonstrador a seguir, tais como Adicionar, Editar, Recuperar com e sem filtro e Excluir .Os testes foram feitos respectivamente com mil e dez mil operações consecutivas nos cenários JDBC e ORM. Por fim apresentamos um gráfico com a média geral de todas as operações testadas. Na média geral o ORM apresentou o melhor tempo (Figura 8).

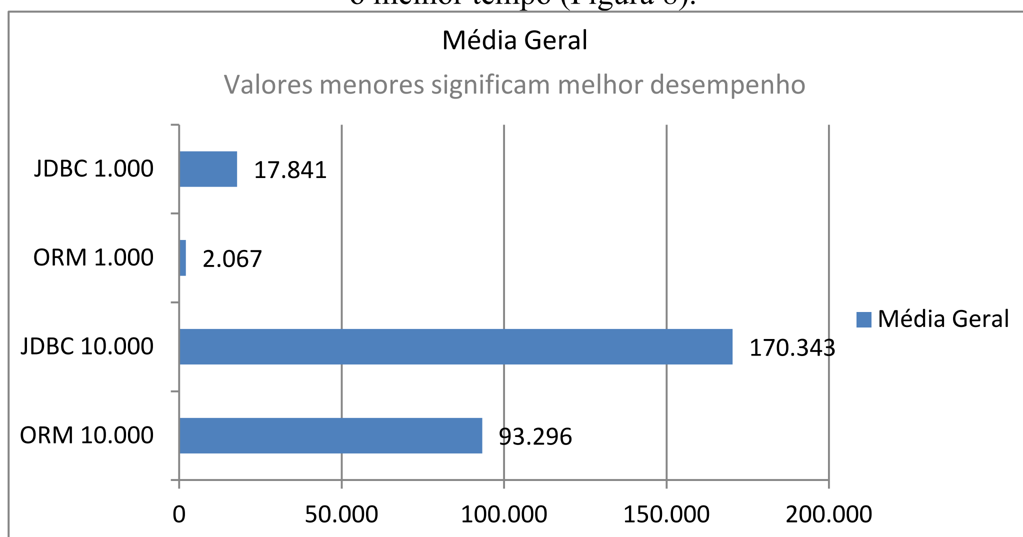


**Figura 8 - Media geral dos métodos testados**

## CONCLUSÃO

Nessa pesquisa, foi realizada a comparação entre JDBC puro e Hibernate, dois métodos de persistir os objetos em uma base de dados relacional. Para realizar a comparação partiu-se da pesquisa e do desenvolvimento de dois protótipos equivalentes, um para casa modo de persistência sendo que ambos diferem apenas na camada de acesso aos dados.

Após o desenvolvimento do protótipo foram analisadas as principais diferenças na codificação entre o JDBC puro e do JDBC com Hibernate. Foi possível perceber que para a criação do projetos, no que diz respeito a parte estrutural, o JDBC é mais simples do que o ORM, pois não necessita de arquivo adicionais, nem mapeamento. Porém, para manipular os objetos na base de dados o ORM precisa de menor quantidade de linhas de código, e trabalha diretamente com os objetos, já com o JDBC puro é preciso trabalhar com os campos do SGBD, sendo necessário conhecer a estrutura da base de dados.

Na comparação de tempo de execução o ORM obteve a melhor média global, contrariando o que se esperava, pois ao usá-la não deixamos de usar o JDBC, mas criamos uma nova camada no aplicativo. Vale destacar que no primeiro acesso o Hibernate tem o maior tempo, porem o tempo dos acessos seguintes é reduzido consideravelmente. O JDBC, por sua vez é mais constante em todos os acessos.

Nos cenários criados o ORM mostrou-se mais vantajoso, pois trabalhando com objetos torna o mapeamento da linguagem OO para o SGBD mais transparente e rápido, principalmente quando se utiliza recurso a linguagem como Herança. O código-fonte ficou menor e mais legível, facilitando futuras manutenções. E o tempo de acesso geral também foi reduzido.

O Hibernate ofereceu uma redução de código significativa, podemos afirmar que com o seu uso há um ganho de produtividade, onde o desenvolvedor pode se concentra na lógica da aplicação, e não na criação de SQL. Um fator relevante é que os desenvolvedores devem conhecer bem a ferramenta, pois seu mau uso pode causar muitos erros, o que diminuiria a produtividade. Conforme King (2010), o Hibernate não é indicado para consultas muito complexas, nessa pesquisa não foi possível testar essa afirmação, pois o protótipo tem um tamanho pequeno e essas consultas são geralmente encontradas em aplicações reais de grande porte. Os resultados documentados podem nortear na decisão do modo de persistir do projeto.

## REFERENCES

DATE, C. J. (2000) "Introdução a sistemas de bancos de dados", 7. ed. Rio de Janeiro: Ed. Campus.

HAHN, Prissila Gomes (2009), "Transformação de Esquemas de Bancos de Dados Relacionais em Orientados a Objetos", http://www.kiron.unesc.net/tcc/?id=606&proj=222

HIBERNATE; Hibernate.org (2010) http://www.hibernate.org/

KING, K et al (2010), "Documentação de Referência Hibernate", http://docs.jboss.org/hibernate/core/3.6/reference/pt-BR/pdf/

LOPES, Roberto Baselio. (2008), "Estudo Comparativo de Implementação de Modelos de Banco de Dados".

MACHADO, Felipe; ABREU, Mauricio. (2004), "Projeto de Banco de Dados: Uma Visão Prática". 11. ed. São Paulo: Érica.

MANENTE, Rogério. (2007), "Avaliação da ferramenta Hibernate em um ambiente de produção de larga escala". http://www.ime.usp.br/~cef/mac499-07/monografias/rogerio/monografia.pdf.

PEAK, Patrick; HEUDECHER, Nick. (2005), "Hibernate Quickly**".** Greenwich: Manning.

REIS, Adailton Pimentel. (2007), "Programação orientada a aspectos com persistência no desenvolvimento de sistemas**",** http://www.ebah.com.br/tcc-sobre-programaaao-orientada-a-aspectos-hibernate-pdf-a65134.html.

RICARTE, Ivan Luiz Marques. (2001), "Programação Orientada a Objetos: Uma Abordagem com Java", http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf

SAM-BODDEN, Brian.(2006), "Beginning**:** From Novice to Professional". Nove York: Apress.

MARTINS, M. R; SANTOS, D. V. (2007), "Métricas para avaliação das alternativas de persistência de dados num ambiente objeto relacional", http://www.ime.usp.br/~cef/mac499-07/monografias/diogo-marcelo/monografia.pdf

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. (2006), "Sistema de banco de dados". Rio de Janeiro: Elsevier. 781 p.

SILVA, Carolina Fernanda da. (2007), “Análise e Avaliação do Framework Hibernate em uma Aplicação Cliente/Servidor”.