Métricas para avaliação das alternativas de persistência de dados num ambiente objeto-relacional.

Autores:
Diogo Vernier dos Santos
Marcelo de Rezende Martins

Orientador:
Professor João Eduardo Ferreira

Índice Analítico.

1 Introdução.

Na área de computação, onde mudança é uma palavra de ordem, os processos e metodologias de produção de software estão se transformando cada vez mais ao longo dos anos. O crescimento da complexidade do software e dos sistemas computacionais nas últimas décadas motivou a busca da criação de processos e métodos mais simples e eficazes que possibilitassem que o desenvolvimento da área continuasse avançando sem ser prejudicado por essa complexidade. Um dos conceitos que surgiram nesse contexto de melhoria e simplificação foi o de **orientação a objetos**.

A orientação a objetos busca atingir objetivos como os de reusabilidade e modularização além de tornar a modelagem do software mais inteligível ao ser humano. Ao invés de pensar na lógica de um programa com uma sequência de instruções e procedimentos voltados para que uma máquina os entenda e execute, uma das coisas que a orientação a objetos propõe é que a ação alcançada por essas instruções e procedimentos seja remodelada dentro de uma visão de um mundo de objetos que cooperam para atingir o mesmo resultado. Esses objetos podem ser vistos então, grosso modo, como entidades, possuidoras de estados e comportamentos, que interagem entre si como fazem as pessoas do mundo real. E, assim, tendo em vista seguir esses ideais citados, surgiram as **linguagens orientadas a objetos** que são baseadas nos conceitos da orientação a objetos para o uso dentro de uma linguagem de programação.

Nessa mesma onda de transformação, apareceu o **modelo relacional** de banco de dados que é baseado na teoria dos conjuntos e na lógica de predicados. Falando de maneira simples, a idéia a chave do modelo relacional é representar os dados em tabelas permitindo manipulá-los através de operações matemáticas relativas à teoria da álgebra relacional. Os bancos de dados que seguem esse modelo são chamados **bancos de dados relacionais**.

Grande parcela dos projetos de software da atualidade utiliza linguagens de orientação a objeto e sistemas de gerenciamento de banco de dados relacionais (RDBMS) como tecnologias de base. O intuito dessa prática é modelar a lógica da aplicação através de objetos que cooperam entre si e persistir os dados necessários para a aplicação utilizando os RDBMS.

Enquanto que de um lado, no modelo orientado a objetos, os dados são apresentados de forma mais natural e inteligível para o ser humano, sendo representados e manipulados através dos conceitos de objetos e seus estados e comportamentos, de outro lado, no modelo relacional, os dados são visualizados através de conceitos matemáticos mais complexos, sendo representados e manipulados através de relações e operações matemáticas.

Para poder trabalhar com esses dois paradigmas em paralelo, é necessário estabelecer uma correspondência entre a representação e manipulação dos dados num modelo e no outro, e fazer uma “ponte” entre os dois modelos. Isso significa utilizar os dados na forma de objetos, ser capaz de

armazená-los utilizando os RDBMS, conseguir resgatá-los do banco de dados relacional e transformá-los novamente em objetos, etc.

Ao longo do tempo, muitas maneiras foram criadas para se poder utilizar orientação a objetos e RDBMS. Numa delas, por exemplo, os desenvolvedores incluem toda lógica necessária para isso em código de acesso ao banco de dados junto ao código das classes dos objetos (Força Bruta) o que possui certas desvantagens como o trabalho duplo de escrever código para utilizar os dados nos dois modelos. Dadas as divergências de representação e manipulação dos dados existentes entre o modelo de objetos e o modelo relacional, não é uma tarefa trivial fazer essa ligação entre os dois e existem diversos obstáculos que surgem ao pensarmos nisso. O conceito de "impedance mismatch" que explicaremos mais adiante caracteriza isso.

2 Conceitos.

Neste tópico abordaremos os principais conceitos que formam a base da persistência de dados num ambiente Objeto-Relacional. Posteriormente a luz destes conceitos, descreveremos as alternativas de persistência de dados que serão avaliadas e o motivo pela qual as escolhemos.

2.1 Persistência de Dados.

O termo persistência de dados consiste na característica dos dados de serem acessíveis além do tempo de execução das aplicações que utilizam-no. Eles ficam armazenados em algum meio que possibilite sua recuperação posterior. O ato de atribuir essa característica as dados denomina-se persistir dados.

Há diversos meios nos quais os dados são persistidos: armazenamento em arquivos, utilização de Sistemas Gerenciadores de Banco de Dados (SGBD), etc. E algumas formas de persistir é: serialização, através de transações...

2.2 Orientação a Objetos.

2.2.1 Introdução.

O conceito de programação orientada a objetos tem suas origens no final dos anos 60 junto ao advento da linguagem SIMULA que introduziu conceitos da programação orientada a objetos como os de classe, subclasse e objetos.

Na programação procedural, a lógica da aplicação é baseada numa chamada sequencial de funções que operam sobre os dados, enquanto que na programação orientada a objetos ou POO (Programação Orientada a Objetos)

este paradigma é rompido. A idéia ao utilizar POO é construir a lógica da aplicação utilizando **objetos** que representam e modelam entidades do mundo real e que cooperam entre si para realizar as tarefas e objetivos da aplicação através de troca de mensagens entre os objetos e o processamento dos dados feitos por cada um. Como um exemplo, um objeto pode representar um cliente do mundo real que pode realizar ações como a de realizar uma compra e possui características como sexo, número de telefone ou RG. Falando em termos da POO, essas possíveis características (**atributos**) e ações (**métodos**) que os objetos podem possuir e realizar estão relacionadas respectivamente ao **estado** e ao **comportamento** do objeto.

Na literatura existem diversas definições para o termo orientação a objetos (OO) que às vezes competem entre si. Como nossa intenção é dar ao leitor uma visão geral dos principais conceitos relacionados à OO presentes no mapeamento objeto relacional e não definir de forma exata o que é OO, optamos por elaborar uma lista definindo esses principais conceitos. É importante observar que as diversas definições de OO adotam os conceitos que iremos apresentar de forma diferente, mas, de modo geral, compartilham em suas definições os mais populares: **herança, polimorfismo, encapsulamento, método, objeto, classe, tipo abstrato de dados e passagem de mensagem**.

2.2.2 Herança.

Conceito muito importante da OO em que uma classe filha (subclasse) "herda" o comportamento e atributos de uma classe pai (superclasse). Assim, um objeto do tipo da subclasse também é um objeto do tipo da superclasse. Além disso, a subclasse pode adicionar novos atributos e métodos a sua definição além daqueles herdados pela superclasse.

Mostramos abaixo um exemplo de herança na linguagem Java. Neste exemplo, a classe Quadrado **estende** as funcionalidades da classe Forma2D. Ela herda então o atributo da cor da linha que desenha uma forma de duas dimensões e os métodos para mudar a cor da linha. Além disso, a classe Quadrado adiciona o atributo tamanho que representa o tamanho de um de seus lados e métodos para manipular o tamanho.

```
public class Forma2D{

        private Color corDaLinha;

        public Forma2D(Color corDaLinha) {
                this.corDaLinha = corDaLinha;
        }

        public Color getCorDaLinha() {
                return corDaLinha;
        }

        public void setCorDaLinha(Color corDaLinha) {
```

```
                this.corDaLinha = corDaLinha;
        }
}



public class Quadrado extends Forma2D{

        private int tamanho;

        public Quadrado(int tamanho) {
                this.tamanho = tamanho;
        }


        public void aumentarTamanho(int quantia){
                tamanho = tamanho + quantia;
        }

        public int getTamanho() {
                return tamanho;
        }

        public void setTamanho(int tamanho) {
                this.tamanho = tamanho;
        }
}
```

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**.

Um fato importante a mencionar é que determinadas linguagens permitem o uso de **herança múltipla** e outras não. Herança múltipla é a capacidade de uma classe poder estender as funcionalidades de mais de uma classe possuindo dessa forma mais de uma superclasse. Em Java ou Smalltalk, por exemplo, não há herança múltipla, mas em C++ ou Python isso já é possível.

2.2.3 Polimorfismo.

Outro conceito importante de OO que permite que diferentes **tipos de dados** sejam manipulados através de uma interface comum. Existem diversas definições e divisões sobre os tipos de polimorfismo existentes na literatura, mas iremos descrever 3 tipos que consideramos os principais:

a) Polimorfismo ad hoc: Descrito informalmente por Christopher Strachey em 1967. Está relacionado à técnica de **sobrecarga de funções** na qual funções podem ser aplicadas a argumentos de diferentes tipos criando-se **funções polimórficas** que podem reagir de maneira diferente de acordo com o tipo de argumento. Por exemplo, o caso da função **abs** da classe **Math** de Java. Através da sobrecarga de funções podemos chamar esse método passando

um **double** ou um **int** como argumento e o ambiente Java escolhe em tempo de execução o código específico apropriado ao argumento passado.

b) Polimorfismo paramétrico: De acordo com Strachey é quando uma função funciona uniformemente sobre um conjunto de tipos que normalmente exibem uma estrutura comum. Exemplos concretos são os templates de C++ e o Generics de Java. Esse conceito não é exclusivo das linguagens OO.

c) Polimorfismo de subtipo: Permite que código seja escrito, em linguagens que aceitam subtipos, de forma genérica para um supertipo funcionando para qualquer subtipo pertencente ao supertipo. Por exemplo, se criarmos um método em Java que concatena numa lista objetos do tipo T, podemos utilizar o mesmo método para qualquer objeto que seja subtipo de T. Outro exemplo é a **sobrescrita de método** que possibilita que subtipos sobrescrevam com sua própria implementação métodos herdados. Ou seja, se inseríssemos na classe de exemplo Forma2D o método perímetro, poderíamos fazer uma implementação para a subclasse Quadrado que devolveria como resultado desse método o atributo tamanho do multiplicado por quatro. Para outra possível subclasse de Forma2D, TrianguloEquilatero, poderíamos devolver o atributo tamanho multiplicado por três. Dessa forma, podemos chamar o método perímetro para a classe Forma2D de forma genérica sem se preocupar com o tipo de dado específico que está sendo utilizado.

2.2.4 Encapsulamento.

Encapsulamento pode ser entendido com a combinação de dois conceitos:

a) Interface de Acesso: União dos dados com os métodos que atuam sobre eles em forma de classes sendo possível modificar os dados apenas através da interface da classe. Ou seja, modificar o estado do objeto só é possível através de chamadas de métodos disponibilizados por sua interface.

b) Esconder os detalhes de implementação dos objetos: Apenas a interface do objeto fica visível e, desde que o objeto mantenha o mesmo comportamento e lógica, é possível trocar a implementação de seus métodos sem preocupação com “efeitos colaterais” na lógica da aplicação.

2.2.5 Método.

Um método é parecido com uma função típica de linguagem procedural com sua assinatura, parâmetros e tipo de retorno. Porém, está associado ao conceito de classe, objeto, dados e encapsulamento. São ações que o objeto pode realizar e que definem **comportamento** e que, como definido na seção de encapsulamento, podem manipular os dados do objeto. Por exemplo, para a classe Cachorro, alguns métodos interessantes seriam latir, rosnar, farejar, etc.

2.2.6 Classe e Objeto.

Uma **classe** define um conjunto de características (propriedades, atributos, campos) e um comportamento (métodos, ações, habilidades) que um conjunto de objetos pertencentes à classe pode assumir. Por exemplo, suponha a definição da classe Cachorro que especifica os atributos cor, tamanho e raça e possui o comportamento de latir, rosnar e farejar. Um objeto pertencente à classe Cachorro deve atribuir um conjunto de valores específicos ao domínio dos atributos especificados pela classe e se comportam da maneira definida por ela.

Assim, um objeto é uma **instância** particular de uma classe que assume um **estado** específico e se comporta de acordo com a especificação da classe. Por exemplo, suponha uma classe hipotética chamada Bola que possui os atributos cor, tamanho e material e seu conjunto de métodos. Essa classe especifica as características possíveis para todos os tipos de bola e seu comportamento. Um objeto pode ser entendido com a concretização de uma bola específica que possui a cor vermelha, tem 20 cm de tamanho e é feita de plástico e tem o comportamento definido pela classe. Outro objeto poderia ser uma bola azul, feita de borracha e com 15 cm de tamanho e o mesmo comportamento.

Classe define as características de uma entidade, define suas características (campos, atributos) e também o seu comportamento(métodos, propriedades). Por exemplo, uma classe *Funcionário* pode conter os atributos *nome, id, idade, cargo*(características) e um método chamado *executaServico* (comportamento). Uma classe é somente uma definição, já o **objeto** representa a classe instanciada, quer dizer, representa uma entidade com estados (atributos com valores atribuídos num determinado instante de tempo) e comportamentos(métodos definidos na classe, que são chamados por outros objetos ou por ele mesmo através de mensagens). Por exemplo, uma instância da classe *Funcionário* pode ser um objeto *funcionario* que num determinado estado contém nome=Marcelo de Rezende Martins, id=87, idade=22, cargo=Analista, além do método *executaServiço* (definido na classe).

2.2.7 Tipo abstrato de dados.

A idéia de tipo abstrato de dados consiste em separar o tipo de dado, conjunto de valores com um conjunto de operações definidas sobre esses valores, de sua implementação. Por exemplo, ao criarmos uma classe Pilha, com o intuito de prover as funcionalidades de uma pilha (stack) comum, e definirmos na interface da classe as operações de iniciar a pilha, empilhar e desempilhar, estamos construindo um tipo abstrato de dados se limitarmos o acesso das funcionalidades de um objeto da classe Pilha apenas através dessa interface da classe. Dessa forma é feita a separação ente a representação interna da pilha e suas funcionalidades. Não é preciso se preocupar se a implementação utiliza um vetor ou qualquer outra estrutura de dados para

armazenar os elementos da pilha. Basta instanciar um novo objeto da classe e utilizar os seus métodos.

2.2.8 Passagem de mensagem.

Consiste no ato de mandar mensagens a objetos, a fim de invocar os métodos dos mesmos ou enviar dados sem a manipulação direta do objeto. É importante lembrar que algumas linguagens como Java e C++ não seguem esse princípio à risca, pois permitem a manipulação direta do objeto. Em Java, por exemplo, se um atributo, campo da classe for do tipo **public** ele pode ser modificado diretamente.

2.2.9 Especialização.

Segundo Broinizi [10], especialização é o processo de definição do conjunto das subclasses de uma entidade. Como um exemplo concreto, podemos citar o caso em que as classes **Gerente**, **Secretário** e **Técnico** especializam a classe **Funcionário** a respeito do tipo de trabalho exercido.

Quando a especialização é **disjunta**, cada entidade pode ser membro de no máximo uma subclasse de especialização, ou seja, não poderíamos ter uma entidade que assumisse concomitantemente as funções de Gerente e Secretário, mas quando a especialização é **sobreponível** as entidades podem participar de mais de uma subclasse de especialização [10].

2.3 Persistência de Objetos.

Analogamente à definição de persistência de dados, a capacidade dos objetos de serem acessíveis através das diversas execuções de uma aplicação denomina-se persistência de objetos. Assim, os **objetos persistentes** são armazenados em algum meio e persistem além do término da execução da aplicação sendo que os **objetos transientes** apenas existem durante a execução do programa e desaparecem ao final da execução.

Para tornar os objetos persistentes, é preciso armazenar de alguma forma as informações que o definem, ou seja, seu valor (estado) e seu comportamento (operações) em algum repositório recuperável que pode ser um banco de dados comum, arquivos do sistema ou um banco de dados orientado a objetos, o que na verdade significa fazer a persistência desses dados definidores do objeto e conseguir reconstruir o objeto à partir dessas informações armazenadas no repositório através de alguma técnica.

É importante enfatizar, como explicado anteriormente na introdução, que estamos interessados no caso em que o repositório de dados é um **banco de dados relacional** e a técnica de persistir os objetos é o **mapeamento objeto-relacional**.

2.4 Modelo de Dados Relacional.

2.4.1 Introdução.

Uma das propriedades principais dos bancos de dados é permitir a **abstração dos dados** que significa, entre outras coisas, esconder detalhes de implementação. Uma maneira de se alcançar o objetivo da abstração dos dados feita pelos sistemas gerenciadores de bancos de dados é a utilização de um **modelo de dados**.

Os modelos de dados definem um conjunto de conceitos que permitem especificar, descrever e modelar a estrutura de um banco de dados - tipos de dados e relacionamento entre os dados, restrições sobre os dados, etc. – e, desse forma, abstrair os dados.

Existem diverso tipos de modelos de dados: conceituais ou de alto nível, físicos, representacionais, etc. Sob a categoria dos **modelos de dados representacionais** encontra-se o **modelo de dados relacional** concebido inicialmente, em 1970, por Tedd Codd, pesquisador da IBM Research, em um artigo clássico. Neste capítulo, introduziremos os conceitos básicos do modelo relacional e discutiremos sobre sua importância.

2.4.2 Conceitos do Modelo Relacional.

O modelo de dados relacional representa os dados através de um conjunto de **relações matemáticas** que, de maneira informal, são similares a uma tabela de valores relacionados. Na prática, utilizam-se essas relações, análogas a uma tabela, para representar os dados que, em geral, representam fatos do mundo real como uma entidade ou um relacionamento. Por exemplo, uma forma de se representar uma entidade ALUNO seria através de uma tabela com as colunas *Nome*, *RG*, *Idade*, *Sexo* e *Classe* na qual cada linha dessa tabela possui um conjunto de valores para cada coluna representando um possível aluno e que esses valores de cada coluna respeitem restrições de **tipo de dado**, ou seja, que um valor para a coluna *Nome* seja obrigatoriamente uma seqüência de caracteres, um valor de *RG* seja um número inteiro, um valor de *Sexo* seja a letra M (masculino) ou a letra F (feminino) e assim por diante.

Utilizamos a explicação da tabela ALUNO e suas colunas apenas para fins de melhor esclarecimento. Na realidade, traduzindo para termos do modelo relacional, uma tabela está ligada a uma **relação**, uma linha a uma **tupla**, uma coluna a um **atributo**, sendo que o fato de os valores de uma coluna respeitarem o mesmo tipo de dados significa que eles pertencem ao mesmo **domínio** de valores possíveis.

Eis as definições de forma mais precisa:

2.4.3 Domínio.

Conjunto de valores indivisíveis (atômicos) dentro do modelo relacional. Uma maneira comum de se estabelecer um domínio, como exemplificado no caso da tabela ALUNO, é definir um tipo de dado a que os valores do domínio devem pertencer [1].

2.4.4 Tipo de Dado.

É o formato a que um dado deve pertencer. No exemplo do aluno, os dados da coluna RG devem ser pertencem ao formato de número inteiro. Outros exemplos de tipos de dados: cadeia de caracteres (String), número em ponto flutuante, número binário 1 representando verdadeiro e 0 representando falso, etc. [1].

2.4.5 Atributo.

Os atributos fazem parte dos **esquemas de relação** que são compostos por um nome da relação R e um conjunto de atributos A1, A2, A3, ..., Ai, ..., An. Um esquema de relação R é representado pela notação R(A1, A2, A3, ..., Ai, ..., An). Os atributos são nomes utilizados para representar a semântica de algum domínio – **dom(Ai)** denota o domínio do atributo Ai - dentro do esquema de relação R. No caso da tabela ALUNO, a palavra Nome representa um atributo e a notação *ALUNO Nome*, *RG*, *Idade*, *Sexo*, *Classe)* indica o esquema de relação ALUNO. O **grau** de uma relação é o número que indica a quantidade de atributos do esquema de relação e que no exemplo dos alunos é igual a 5 [1].

2.4.6 Tupla.

Uma n-tupla t é uma lista de valores (c1, c2, c3, ..., cn) que possuem uma ordenação sendo que cada valor ci ou pertence ao domínio de Ai, dom(Ai), ou é nulo (null). Novamente para comparar com o caso dos alunos, suponha um possível conjunto de valores (“Lucas”, 123456, 19, ‘M’, “Turma 54”) como sendo uma linha da tabela ALUNO que descreve o aluno de nome Lucas. Esse conjunto representa uma 5-tupla t=<“Lucas”, 123456, 19, ‘M’, “Turma 54”> [1].

2.4.7 Relação.

Uma relação r pertencente a R(A1, A2, A3, ..., Ai, ..., An) pode ser interpretada com um conjunto de n-tuplas em que cada n-tupla respeita as restrições, como as de domínio, da maneira definida acima. De maneira mais formal uma relação ou estado da relação de grau n, **r(R)**, é uma **relação matemática** ou subconjunto do conjunto formado pelo produto cartesiano dos domínios de cada atributo Ai que compõe R [1].

Segundo Navathe [1]:
r(R) ⊆ (dom(A1) × dom(A2) × ... × dom(Ai) × ... × dom(An))

2.4.8 Tipo e Instância de Relacionamento.

Definição de **tipo de relacionamento**, segundo Ferreira [2]:

"Tipo e Instância de Relacionamento: Um **tipo de relacionamento** R entre n tipos de entidades E1, E2, ..., En é um conjunto de associações entre entidades desses tipos. Diz-se que cada entidade E1, E2, ..., En **participa** no tipo de relacionamento R e que as entidades individuais e1, e2, ..., en participam na instância do relacionamento ri=(e1, e2, ..., en). O índice i indica que podem existir várias instâncias de relacionamento."

Conceitos importantes ligados aos conjuntos de relacionamento que utilizaremos mais tarde:

- **Relacionamento binário:** Dizemos que um tipo de relacionamento é **binário recursivo** quando existem 2 entidades participantes E1 e E2 [3].

- **Relacionamento binário recursivo:** Dizemos que um tipo de relacionamento é **binário recursivo** quando existem 2 entidades participantes E1 e E2 e E1 = E2.

- **Cardinalidades de mapeamento para relacionamentos binários:**
Segundo Silberschatz [3], a cardinalidade de mapeamento, que "expressa o número de entidades ao qual outra entidade pode ser associada por um conjunto de relacionamento", é alguma das seguintes:

Um-para-um: "Uma entidade em A é associada a no máximo uma entidade em B, e uma entidade em B é associada a no máximo uma entidade em A."

Um-para-muitos: "Uma entidade em A é associada a qualquer número de entidades em B, entretanto, uma entidade em B pode ser associada a no máximo uma entidade em A."

Muitos-para-um: "Uma entidade em A é associada a no máximo uma entidade em B, entrentanto, uma entidade em B pode ser associada a qualquer número de entidades em A."

Muitos-para-muitos: “Uma entidade em A é associada a qualquer número de entidades em B e uma entidade em B pode ser associada a qualquer número de entidades em A.”

2.4.9 Restrição de Integridade.

Segue uma explicação sobre restrição de integridade, segundo Silberschatz [3]:

“As restrições de integridade garantem que as mudanças feitas num banco de dados por usuários autorizados não resultem em uma perda da consistência dos dados.”

Como exemplos concretos de restrições de integridade, podemos citar:
- A coluna logradouro da tabela Endereço não pode ser nula.
- Os valores da chave primária de uma tabela não podem ser repetidos.
- As chaves estrangeiras contidas em uma tabela devem ser consistentes referenciando instâncias de entidades existentes.

2.4.10 Relacionamento n-ário.

Segundo Teorey [4], o grau de um relacionamento é o número de entidades associadas ao relacionamento. E um relacionamento n-ário é um **relacionamento** de grau n. Exemplificando, o relacionamento binário possui n = 2.

2.4.11 Agregação.

Segue a definição de agregação, segundo Silberschatz [3]:

“Agregação é uma abstração pela qual os relacionamentos são tratados como entidades de nível superior.”

Nesse caso, um relacionamento binário muitos-para-muitos denominado Consulta entre as possíveis entidades Médico e Paciente, poderia ser representado com uma entidade que possuísse atributos como nome do médico, nome do paciente, hora da consulta, etc.

A agregação é uma boa forma de se representar relacionamentos que possuem atributos de relacionamento. Os atributos se tornam atributos da entidade de nível superior.

2.4.12 Representação de Herança.

Herança é o principal conceito presente na orientação a objetos, porém representá-los no modelo relacional é uma tarefa complexa. Há, no entanto, três soluções comumente utilizadas: tabela por hierarquia de classes (table per class hierarchy), tabela por classe (table per class), tabela por classe concreta (table per concrete class).

Hierarquia de classes

Hierarquia de classes é obtida através de heranças que iniciam-se na classe pai, raiz da hierarquia de classes. Cada nó nesta hierarquia representa uma classe, com cada classe herdando informações de suas respectivas classes pais (nó pai). [16].

Para demonstrar estas três soluções, considere o exemplo abaixo de hierarquia de classes, no qual temos uma classe abstrata *Funcionario* e suas respectivas especializações.

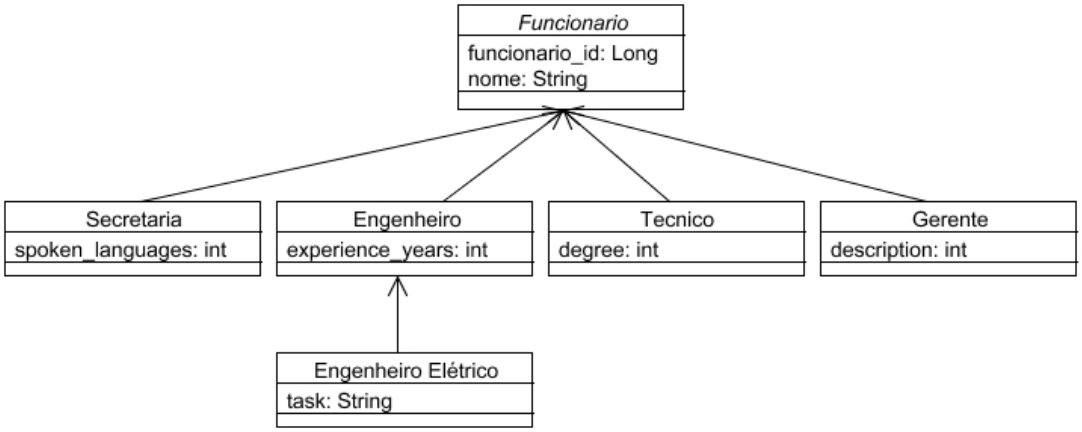
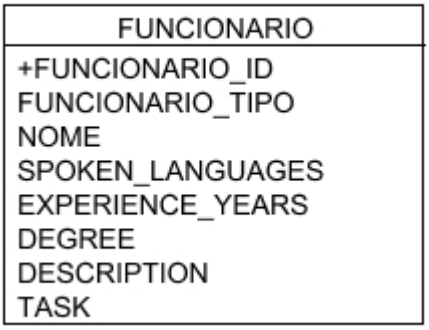


Tabela Por Hierarquia de Classes (*Table Per Class Hierarchy*)

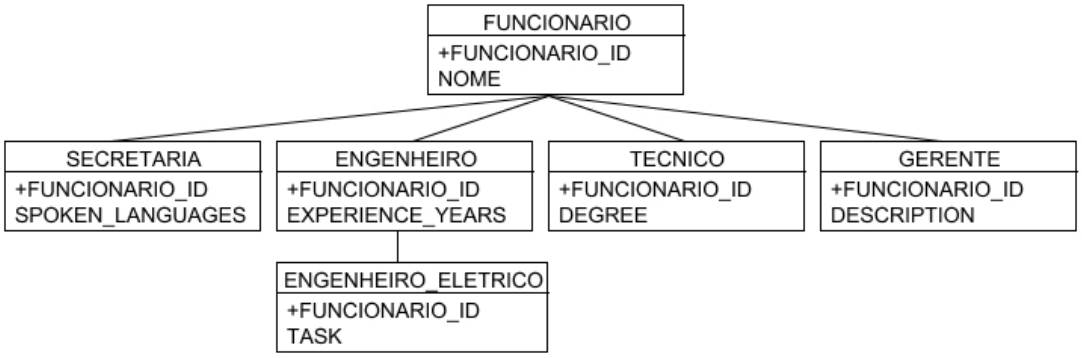
Esta solução provê apenas uma tabela por hierarquia de classes, sendo que esta tabela inclue todas as propriedades de todas as classes da hierarquia.



Para diferenciar as subclasses concretas é preciso adicionar uma coluna diferenciadora (*discriminator column*), que permite discriminar cada tupla da tabela. Por exemplo, a subclasse *Secretaria* seria discriminada por *"SECRETARIA"*, *Engenheiro* por *"ENGENHEIRO"*, e assim por diante. Esta coluna, neste caso, é *FUNCIONARIO_TIPO*. Neste exemplo, ao inserir um objeto do tipo *Engenheiro Elétrico*, será feita uma inserção apenas na tabela *FUNCIONARIO*.

Tabela Por Classe (Table Per Class)

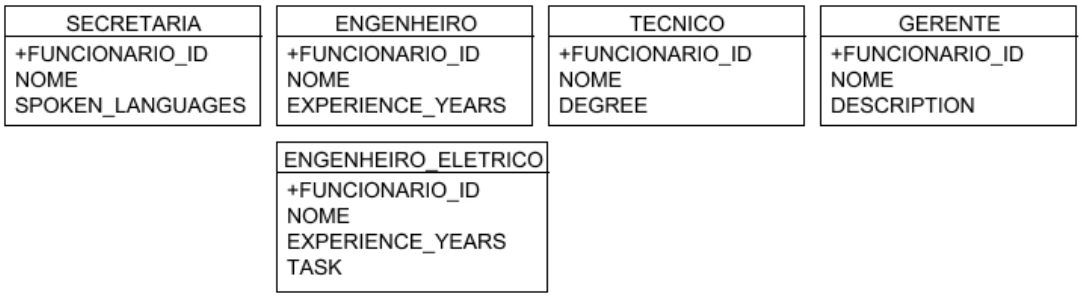
Esta estratégia consiste na criação tabela para cada classe da hierarquia, e a tabela de cada classe contém apenas colunas correspondentes aos atributos da própria classe.



Neste exemplo, ao inserir um objeto do tipo *Engenheiro Elétrico*, resultará na inserção de tuplas nas tabelas *FUNCIONARIO, ENGENHEIRO, ENGENHEIRO_ELETRICO*.

Tabela por classe concreta (Table Per Concrete Class)

Esta solução descarta completamente herança e polimorfismo do esquema SQL, provendo uma tabela para cada classe concreta da hierarquia, contendo todos os atributos herdados do relacionamento.



Neste exemplo, ao inserir um objeto do tipo *Engenheiro Elétrico*, será feita apenas uma inserção na tabela *ENGENHEIRO_ELETRICO*.

2.5 Objetos, Relações e a "Impedance Mismatch".

O termo "impedance" ou impedância, da terminologia da Engenheira Elétrica, é uma propriedade de um circuito de corrente alternada que mede a

oposição total ao fluxo da corrente elétrica nesse circuito. Quando dois circuitos, ou linhas de transmissão com impedâncias diferentes são conectados ocorre o problema da reflexão de sinal que pode resultar em barulho e atenuação. O ideal é que os dois circuitos possuam valores de impedância o mais próximo possível. Daí a origem do termo “impedance mismatch”, pois os valores deveriam coincidir (match).

De forma abstrata, “impedance mismatch” ou conflito de impedância, em Engenharia da Computação, é quando dois sistemas diferentes não funcionam de maneira ideal e eficiente quando conectados para funcionarem conjuntamente. É o que ocorre quando utilizamos bancos de dados relacionais e sistemas baseados em linguagens orientadas a objetos.

De forma análoga, poderíamos dizer que os conceitos do modelo de orientação a objetos e do modelo relacional possuem valores de impedância diferentes. Enquanto o modelo relacional dá mais ênfase aos dados em si e sua representação e organização, o modelo orientado a objetos enfoca na abstração dos dados e seu comportamento, operações na forma de objetos.

Essas diferenças entre os dois modelos geram problemas, que caracterizam a existência de conflito de impedância, quando os dois paradigmas operam juntos. Por exemplo, suponha a existência de uma hierarquia de classes em que a classe **A** é superclasse das classes **B**, **C** e **D**. Suponha também que a classe **A** possui um método **x** que recebe parâmetros que atualizam seus dados e as subclasses podem sobrescrevê-los para fazer a atualização correta especificamente para o caso da subclasse. Agora, imagine que estamos a ponto de percorrer uma coleção de objetos da classe A, que podem muito bem pertencer aos subtipos **B**, **C** ou **D**, e que iremos invocar o método **x** para todos esses objetos da coleção, mas utilizando os objetos na forma de referências da classe **A** independente do subtipo ao qual pertençam. É um caso típico de um benefício do modelo orientado a objetos, o polimorfismo possibilitado pelo uso de herança e o modelo OO reage normalmente a essa operação que queremos realizar. Mas e no modelo relacional? Não existe um meio direto ao ponto, “nativo” de se mapear a capacidade polimórfica. O que fazer no caso em que cada objeto de cada subclasse está representado numa tabela diferente do banco de dados relacional?

São casos como o do polimorfismo que dificultam uma relação harmoniosa direta entre os dois paradigmas. Na tabela abaixo, exibimos mais alguns exemplos polêmicos:

Modelo OO	**Modelo Relacional**	**Problema...**
Os tipos de dados nas linguagens OO divergem entre si. Enquanto em algumas linguagens um determinado tipo possui certa precisão e é implementado de um	O mesmo acontece para os tipos de dados em bancos de dados que implementam o modelo relacional.	Se entre as implementações do mesmo modelo já não existem unanimidade entre os tipos de dados. Como mapear corretamente as

jeito, em outra linguagem o cenário é outro.		diferenças entre os modelos em si?
Tipos de dados e classes numa linguagem orientada a objetos podem possuir várias granularidades em suas representações. Por exemplo, uma classe pode possuir apenas atributos que são primitivas, mas pode possuir atributos que são outras classes. Um exemplo seria uma classe Pessoa que possuísse uma classe Contato como classe de atributo que guarda informações de telefones, etc.	Nos bancos de dados relacionais, ao realizarmos consultas em SQL, o desempenho costuma ser melhor quando o número de joins entre as tabelas é menor e não precisamos envolver muitas tabelas em uma consulta.	No caso da classe Pessoa, o mais natural é mapear a classe Pessoa em uma tabela e a classe contato em outra tabela do banco, estabelecendo uma relação entre elas. Mas como mapear classes de diferentes granularidades, o que pode exigir o uso de mais de uma tabela, levando em consideração um bom desempenho nas consultas?

2.6 Mapeamento objeto-relacional (ORM).

Existem diversas soluções para se mapear os objetos no modelo relacional. Uma maneira muito utilizada, e que ainda a é para sistemas de pequeno porte, é escrever o código encarregado de fazer a comunicação com o banco de dados relacional nas próprias classes dos objetos que se encarrega de tarefas como a de mapear os atributos do objeto em campos das tabelas do banco.

Soluções como a anterior possuem problemas. De início, existe uma duplicação de informação, pois os nomes dos atributos dos objetos estão repetidos em nomes de colunas de tabelas do banco de dados relacional. Outro problema é a tarefa repetitiva e cansativa de se escrever esses códigos para toda a hierarquia de classes e que, na maioria dos casos, pouco varia entre o conjunto das classes. Além disso, essa solução tem a característica de não ser flexível para mudanças nos modelos. Se quisermos adicionar um atributo em alguma classe ou, analogamente, adicionar um campo numa tabela do banco de dados, se torna necessário fazer alterações no código de mapeamento.

O mapeamento objeto-relacional é um conceito que busca “aliviar” problemas como os citados acima - eliminando a duplicação de informação ou tornando-a transparente ao desenvolvedor - além de reduzir o conflito de impedância que foi mencionado anteriormente.

O termo mapeamento-objeto relacional, ou “object relational mapping” (ORM), não possui uma definição padrão. De maneira informal, podemos afirmar que seu significado está relacionado, como dissemos antes, ao trabalho de “ligar” ambientes, sistemas, ferramentas e aplicações que utilizam de um lado o paradigma da orientação a objetos, e de outro lado o modelo relacional.

Fazer uma conexão entre essas duas “visões” de mundo inclui a realização de um conjunto de tarefas para permitir o funcionamento simultâneo e cooperativo dos lados envolvidos. Estas tarefas podem significar ações como transformar as tabelas de um banco de dados relacional em um conjunto de classes, e alterar as informações das tabelas do banco através de uma API de comunicação para refletir as mudanças ocorridas no “mundo dos objetos”.

Pode-se dizer então que, de forma geral, realizar um conjunto de tarefas como as anteriores para tornar possível a interoperabilidade dos dois modelos, o relacional e o de orientação a objetos, de forma consistente, eficiente, robusta e transparente, resolvendo problemas do conflito de impedância é fazer mapeamento objeto-relacional.

Existem diversas ferramentas que implementam o mapeamento objeto-relacional através de várias maneiras e técnicas. Na próxima seção discutiremos sobre essa diversidade.

2.7 Implementações de ORM.

2.7.1 Introdução.

Existem muitas implementações ORM para as diversas linguagens OO, inviabilizando uma análise de cada uma. Segue abaixo uma lista de implementações ORM, retirado da Wikipedia: [28].

ColdFusion

- ARF - Active Record Factory;
- CFCPowerTools;
- Reactor;
- ObjectBreeze;

Common Lisp

- cl-perec

Java

- Carbonado
- Castor
- Cayenne
- CocoBase PURE POJO V5 For JPA
- CrossDB
- Ebean

- Enterprise Objects Framework
- WebObjects
- FireStorm/DAO
- Hibernate
- Hydrate
- iBATIS
- intelliBO
- Java Data Objects (JDO)
- JDBCPersistence
- JDX
- JPOX
- Kodo
- Lychee
- OpenAccess
- OJB
- OpenJPA
- POEM
- PriDE
- SavePoint
- SimpleORM
- Speedo
- TopLink
- Torque
- WebObjects

.NET

- .netTiers
- Briyante Integration Environment
- Business Logic Toolkit for .NET
- Castle ActiveRecord
- GURA
- Habanero
- iBATIS.NET
- IdeaBlade DevForce
- Lattice.DataMapper
- LightSpeed
- LLBLGen Pro
- LLBLGen
- Neo
- NHibernate
- NJDX
- Nolics
- Opf3
- ObjectMagix
- ObjectMapper .NET
- OpenAccess
- ORM.NET
- Persistor.NET
- Puzzle.NPersist

- Sooda
- Subsonic (DAL)
- TierDeveloper

Perl

- Class::DBI
- Rose::DB::Object
- OOPS
- ORM
- DBIx::Class
- Alzabo
- Tangram (Perl)

PHP

- ADOdb Active Record
- CakePHP
- Doctrine
- DB DataObject
- EZPDO
- Junction PHP
- Metastorage
- PhpMyObject
- PHP Object Generator (POG)
- pork.dbObject
- Propel
- QCodo
- xPDO
- Xyster Framework

Python

- Axiom
- Ape
- SQLAlchemy
- SQLObject
- PyDO
- QLime
- Twisted Asynchronous Database Api
- PyDAO

Ruby

- Active Record
- Og
- Rubernate
- Lafcadio
- Sequel
- DataMapper

- Momomoto
- Kansas

Smalltalk

- GLORP

C++

- DTL
- DataXtend CE for C++
- Object Builder
- SOCI

Devido a isso, foi necessário uma seleção de quais ferramentas seriam analisadas, segundo tais critérios:

1. Demonstram muito bem os conceitos;
2. Interessantes por parte teórica;
3. Apresentam soluções inovadoras;
4. Grau de aceitação por parte da comunidade e do mercado.

Seguindo tais critérios, para termos uma análise viável, restringimo-nos somente a três implementações:

- Active Record;
- Hibernate;
- Toplink Essentials;

Nas próximas seções relativas a cada implementação, teremos inicialmente uma visão geral e um breve histórico. Posteriormente as análises relativas a sua arquitetura, metadados, linguagens de manipulação de dados(DML),linguagem de definição de dados(DDL) e caching.

2.7.2 Active Record.

É uma implementação do padrão de projeto, de mesmo nome, descrito por Martin Fowler, para mapeamento objeto relacional. A implementação de estudo é a desenvolvida para Ruby, e que faz parte da camada de persistência do framework *Ruby on Rails*.

Arquitetura [17].

A arquitetura do Active Record é baseada, é claro, no padrão *Active Record*, diferenciando-se das outras implementações ORM que iremos analisar posteriormente, que é baseado no padrão *Data Mapper/Identy Map/ Unit of Work*.

Padrão de projeto Active Record

Active Record é "um objeto que envolve uma tupla numa tabela ou *view* de um banco de dados, encapsulando o acesso a este banco de dados e adicionando lógica de domínio nestes dados". [18]. Isto significa que o Active Record contém métodos "de classe" para encontrar instâncias, e cada instância é responsável por atualizar, apagar e inserir si mesma no banco de dados.

Padrão de projeto Data Mapper

Data Mapper é "uma camada de mapeamento que movimenta os dados entre objetos e banco de dados enquanto os mantêm independentes um do outro e do mapeamento em si.". [18]. Isto redireciona a responsabilidade de persistência para fora do domínio do objeto, utilizando geralmente um ***identy map*** para manter os relacionamentos entre o domínio de objetos e o banco de dados. Além disso, ele utiliza ***Unit of Work*** para manter o caminho dos objetos que são mudados e para ter certeza que ele persiste corretamente.

Metadados [8].

Active Record não possui metadados, quer dizer, não há arquivos XML de configuração, ou anotações. Ele utiliza um conjunto de padrões sensíveis, que minimizam a quantidade de configurações que o desenvolvedor precisa fazer. Segue abaixo um exemplo de uma classe *departament* e que contém uma tabela correspondente *departaments*.

```
CREATE TABLE  departments (
  id int(11) NOT NULL auto_increment,
  nome varchar(45) NOT NULL
);

class Department < ActiveRecord::Base
end
```

Isto é o suficiente para termos o mapeamento objeto-relacional, não sendo necessário nenhuma configuração, apenas, é claro, a configuração para a conexão com o banco de dados.

As subclasses de *ActiveRecord::Base* estão associadas a uma tabela correspondente no banco de dados. Por padrão, Active Record assume que o nome da tabela está no plural do nome da classe (nomes em inglês), e caso a classe contenha muitos nomes com iniciais maiúsculas, assume-se que o nome da tabela contenha *underscores* entre as palavras. Exemplo abaixo, retirado de [8], mostra também que alguns plurais irregulares são controlados.

Nome da Classe	Nome da Tabela	Nome da Classe	Nome da Tabela
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

Caso queira desabilitar esta opção, basta acrescentar o comando abaixo no arquivo *environment.rb*, contido no diretório *config*.

```
ActiveRecord::Base.pluralize_table_names = false
```

Caso a opção de nomes de tabelas no plural tenha sido desabilitada ou queira mudar o nome padrão da tabela, basta chamar a diretivva *set_table_name*:

```
class Department < ActiveRecord::Base
      set_table_name "Departamentos"
end
```

As colunas da tabela do banco de dados estão associadas aos atributos da classe. Porém, nos exemplos anteriores, as classes não definiram os seus atributos, isto acontece porque o Active Record associa-os dinamicamente em tempo de execução.

```
CREATE TABLE  departments (
  id int(11) NOT NULL auto_increment,
  nome varchar(45) NOT NULL
);

class Department < ActiveRecord::Base
end
```

O Active Record determina também o tipo do atributo correspondente ao tipo da coluna.

Tipo SQL	**Classe Ruby**	**Tipo SQL**	**Classe Ruby**
int, integer	FixNum	float, double	Float
decimal, numeric	Float	char, varchar, string	String
interval, date	Date	datetime, time	Time

Linguagem de Manipulação de Dados (DML)

O Active Record utiliza o SQL para a manipulação dos dados, e através do método *find()* e outros métodos dinâmicos, ele facilita a escrita e criação de consultas. Todas as subclasses de *ActiveRecord::Base* suportam o método *find()* que provê diferentes formas de especificar uma consulta. O exemplo abaixo retorna todos os registros da classe *Department*:

```
Department.find(:all)
```

No caso, esta consulta retorna todos os registros da tabela *departaments* associada a classe *Department*. O SQL equivalente seria:

```
SELECT * FROM departaments;
```

Neste caso, o Active Record retorna já um vetor de objetos *Department*. A consulta retornará apenas um objeto, não um vetor, caso esteja especificado no método *find()* o retorno do primeiro registro ou a identidade do objeto.

```
Department.find(:first) #retorna o primeiro registro
Department.find(1)      #retorna o objeto de id=1
```

Porém, caso seja especificado um vetor contendo valores para identidades, o Active Record retornará uma vetor de objetos.

```
Department.find([1])  #retorna um vetor de objetos de id=1
```

Além disso, o método *find()* suporta dois tipos de parâmetros nas consultas, parâmetros nomeados e os seqüencias. Com os parâmetros nomeados ao invés de termos consultas como `Department.find(:all,:conditions => "name = 'Department 1'")`, temos:

```
params = {:name => "Department 1"}
Department.find(:all,:conditions => "name = :name",params)
```

Já com os parâmetros seqüencias temos:

```
name = "Department 1"
place = "SP"
Department.find(:all,:conditions => "name = ? and place =
?",name,place)

# ou um vetor contendo SQL e os valores a serem substituídos
array = ["name = ?","Department 1", "place = ?","SP"]
Department.find(:all,:conditions => array)
```

O Active Record possui métodos dinâmicos de consulta, para facilitar algumas consultas freqüentemente utilizadas. Tais métodos são dinamicamente criados a partir dos atributos que a classe possui. Por exemplo, a classe *Department* possui de acordo com a tabela associada, atributos como *id, name*. A partir disto, podemos criar uma consulta no qual as colunas precisam conter valores iguais ao especificado no método.

```
Department.find_by_id(2)
Department.find_by_name("Department 1")
```

Isto é equivalente a:

```
Department.find(:first,:conditions => "id = ?",id)
Department.find(:first,:conditions => "name = ?",name)
```

Neste caso, ele retorna apenas a primeira ocorrência que obedece tal condição, porém para retornar todas as ocorrências, que, no caso, retornaria um vetor de objetos, basta acrescentar *all*.

```
Department.find_all_by_name("Department 1")
```

Isto é equivalente a:

```
Department.find(:all,:conditions => "name = ?",name)
```

SQL Nativo

O Active Record permite expressar as consultas no dialeto SQL específico do banco de dados através do método *find_by_sql()*. Um exemplo simples é retornar todos os registros da tabela *departments*.

```
Department.find_by_sql("select * from departments")
```

Isto retornará um vetor de objetos com seus respectivos atributos associados as colunas especificadas na consulta, pois somente estarão disponíveis os atributos, cujas colunas foram retornadas.

Linguagem de Definição de Dados(DDL)

O Active Record exporta e gerencia o esquema de banco de dados através da *migração*. Isto permite criar tabelas, adicionar colunas, removê-las, controlando toda a evolução do esquema do banco de dados durante a fase de produção. Active Record exporta somente para o banco de dados, e isto é feito através de subclasses do *ActiveRecord::Migration*, que são obrigadas a implementar os métodos *up* e *down*, especificando o momento de migração e o momento de reversão de tal, respectivamente.[19].

Caching

Active Record não suporta cache.

2.7.3 Hibernate.

É um framework para persistência de objetos, baseado no mapeamento objeto relacional e a sua análise deve-se principalmente pela maturidade do projeto, sendo que este framework é muito utilizado, se não o mais utilizado. Ele originou-se em 2001 e foi concebido inicialmente para Java, porém atualmente este projeto conta implementações para .NET, NHibernate, que iniciou nos meados de 2005.

Arquitetura [20]

O Hibernate é praticamente dividido, segundo Deepak Kumar, em três camadas principais:

- **Gerenciador de conexões:** Esta camada provê o gerenciamento eficiente das conexões com o banco de dados. A conexão com o banco de dados é a parte mais custosa, devido à enorme quantidade de recursos necessários para abri-las e fechá-las.
- **Gerenciador de transações:** Esta camada provê ao usuário a possibilidade de executar uma ou mais sentenças(conjunto de operações) no banco de dados;
- **Mapeamento Objeto Relacional:** Esta camada é utilizada para operações de recuperação, inserção, atualização e remoção de

dados em tabelas. É através desta camada, que o Hibernate escreve as *queries* apropriadas ao salvar um objeto, por exemplo.

As camadas que mais críticas do Hibernate são a **Gerenciador de conexões** e **Gerenciador de transações**, faltando em performance e capacidade de execução.

Metadados

O Hibernate utiliza-se de metadados em arquivos XML para a definição de mapeamento objeto relacional. Estes metadados são definidos pelo próprio Hibernate, seguindo um formato padrão. Segue abaixo um formato deste arquivo:

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

<hibernate-mapping package="org.polepos.teams.hibernate.data">
<class name="H3Department" table="h3departments">
    <id name="id">
       <generator class="native"/>
    </id>
    <property name="name"/>
</class>
</hibernate-mapping>
```

O arquivo de mapeamento contém inicialmente o DTD(Document Type Definition), isto serve para definir a estrutura do documento contendo um conjunto de atributos e elementos permitidos.

O principal elemento é o *hibernate-mapping*, este elemento define o início da declaração das classes a serem persistidas. Estas classes são declaradas através do elemento *class*, contida dentro do elemento *hibernate-mapping*. Uma boa prática é declarar uma classe(ou uma hierarquia de classes) por arquivo de mapeamento e posteriormente renomear o arquivo com o nome da classe, por exemplo H3Department.hbm.xml ou H3Employee.hbm.xml(nome da classe pai numa hierarquia de classes). [11].

O elemento *class* permite o desenvolvedor associar a classe, especificado pelo atributo *name*, à tabela, cujo nome é definido por *table*:

```
<class name="H3Department" table="h3departments">
```

As classes mapeadas devem declarar a coluna que é chave primária da tabela no banco de dados. O elemento *id* define o mapeamento do atributo da classe, cujo nome é especificado pelo atributo *name*, a esta coluna que é chave primária da tabela.

```
<id name="id">
  <generator class="native"/>
</id>
```

O elemento *generator* gera identificadores únicos para cada instância de classe. Todas as classes que funcionam como geradores implementam a interface *org.hibernate.id.IdentifierGenerator*, uma interface muito simples. Porém o próprio Hibernate provê classes que já implementam esta interface, facilitando o trabalho do desenvolvedor.

Cada classe deve declarar os atributos a serem persistidos através do elemento *property*, contido dentro do elemento *class*. No exemplo acima, temos a classe *H3Departments*, contém um atributo *name* do tipo String a ser persistido. O elemento *property* contém atributos para especificar a coluna correspondente na tabela e o tipo de dado equivalente no modelo relacional, caso isto seja omitido, o Hibernate consegue associar este atributo a uma coluna no banco de dados, além de associar a um tipo de dado do modelo relacional, equivalente ao tipo de dado deste atributo do modelo OO.

No exemplo acima, o Hibernate associará o atributo *name* da classe *H3Department* a uma coluna cujo nome é *name* também, pois omitimos o atributo *column*, que especifica a coluna associada a este atributo.Já o tipo de dado equivalente no modelo relacional será *VARCHAR*, pois o atributo é do tipo *java.lang.String* no modelo OO e esta equivalência de tipo de dados é feita automaticamente pelo Hibernate (,pois omitimos o atributo *type*, que especifica o tipo de dado equivalente no modelo relacional), associando o tipo de dado do modelo OO a um correspondente no modelo relacional.

```
<property name="name"/>
```

Segue abaixo um exemplo equivalente, porém com a adição dos atributos *type* e *column* :

```
<property name="name" column="name" type="string"/>
```

Os metadados do Hibernate possibilitam a configuração em tempo de execução, permitindo alterações dinamicamente. Segue abaixo um exemplo, no qual adicionamos o atributo *place* ao arquivo de metadados da classe H3Department:[21]

```
PersistentClass mpg = cfg.getClassMapping(H3Department.class);
SimpleValue simpleValue = new SimpleValue();
  simpleValue.addColumn(new Column("place"));
  simpleValue.setTypeName(String.class.getName());

        PersistentClass persistentClass = getPersistentClass();
        simpleValue.setTable(mpg.getTable());

        Property property = new Property();
        property.setName("place");
        property.setValue(simpleValue);
        mpg.addProperty(property);
```

```
//Atualiza o mapeamento
 SessionFactory factory = cfg.buildSessionFactory();
```

Linguagem de Manipulação de Dados(DML)

O Hibernate contém várias formas de trabalhar com a Linguagem de Manipulação de Dados(DML):

1. Hibernate Query Language(HQL)
2. Criteria
3. Outras formas(Example, Detached queries, SQL Nativo)

Hibernate Query Language

Hibernate Query Language(HQL) é uma linguagem de manipulação de dados(DML), com sintaxe semelhante ao SQL(Structured Query Language) e totalmente orientado a objetos, com suporte a herança, polimorfismo. [11]. Como HQL é orientado a objetos, as consultas são feitas com nomes e referências das próprias classes. Segue um exemplo:

```
List list = session.createQuery("from H3Department").list();
```

No caso, esta consulta retorna todos os registros da tabela *h3departments* associada a classe *H3Department*. O SQL equivalente seria:

```
SELECT * FROM h3departments;
```

O HQL permite omitir alguns termos, como exposto no exemplo acima, no qual o omitimos *"SELECT *"* e somente era necessário *"from H3Department*" para que fosse possível retornar todos os registros da tabela *h3departments*.

O HQL permite recuperar o resultado através de uma **java.util.List**, bastando chamar o método **list()** da instância da classe **org.hibernate.Query.**

```
Query q = session.createQuery("from H3Department");
List list = q.list();
```

Além disso, HQL suporta dois tipos de parâmetros nas consultas, parâmetros nomeados e os seqüencias. Com os parâmetros nomeados ao invés de termos consultas como from H3Department h where h.name = 'Financy', teríamos:

```
String query = "from H3Department h where h.name = :searchName";

Query q = session.createQuery(query)
.setString("searchName", "Mary");

List result = q.list();
```

Já com os parâmetros seqüencias teríamos:

```
String query = "from H3Department h where h.name = ? and h.place
= ?";

Query q = session.createQuery(query)
query.setString(2, "São Paulo");
query.setString(1,"Maria");
List result = q.list();
```

As consultas com parâmetros nomeados podem estar definidas em arquivos metadados também:[22].

```
<query name="h3departments.from "><![CDATA[
    from H3Department as h where h.place = :searchPlace
]]></query>
```

Estas *consultas* são chamadas através do método **getNamedQuery()**:

```
Query q = session
.getNamedQuery("h3departments.from")
.setString("searchPlace", "Sao Paulo");
```

Criteria

A interface *org.hibernate.Criteria* representa uma consulta sobre uma classe persistida. A *Session* é a fábrica de instâncias de *Criteria*

```
Criteria crit = sess.createCriteria(H3Department.class);
crit.setMaxResults(50);
List departmets = crit.list();
```

Para que possamos adicionar restrições as consultas, existe a interface *org.hibernate.criterion.Restrictions*, permitindo adicionar alguns tipos pré-definidos de *Criterion*, pois as consultas com *criteria* são instâncias da interface *org.hibernate.criterion.Criterion*:

```
List departmets = sess.createCriteria(H3Department.class)
    .add( Restrictions.like("name", "F%") )
    .list();
```

Outras formas(Example, Detached queries, SQL Nativo)

O Hibernate dispõe de vários outros modos de fazer a manipulação de dados, entre estas, temos também:

Example

A classe *org.hibernate.criterion.Example* permite criar consultas com restrições a partir de uma dada instância:

```
H3Department department = new H3Department();
department.setName('Adm');
department.setPlace('Sao Paulo');
List results = session.createCriteria(H3Department.class)
    .add( Example.create(department) )
    .list();
```

Detached Queries

Permitem através da classe *DetachedCriteria* criar consultas fora do escopo de sessão:

```
  DetachedCriteria query =
DetachedCriteria.forClass(H3Department.class)
      .add( Property.forName("name").eq('Fdm') );

  Session session = ....;
  Transaction txn = session.beginTransaction();
  List results =
query.getExecutableCriteria(session).setMaxResults(100).list();
  txn.commit();
  session.close();
```

SQL Nativo

O Hibernate permite expressar as consultas no dialeto SQL específico do banco de dados. As execuções de consultas SQL nativas são controladas pela interface *org.hibernate.SQLQuery*, que pode ser obtida chamando *Session.createSQLQuery()*. Um exemplo simples é retornar todos os valores escalares da tabela *h3departments*.

```
sess.createSQLQuery("SELECT * FROM h3departments").list();
```

Esta consulta retornará uma lista de vetores de objetos (Object[]) com valores escalares para cada coluna da tabela *h3departments*. O Hibernate utilizar-se-á do *ResultSetMetadata* para definir a ordem e os tipos de dados dos valores escalares retornados. [11].

Linguagem de Definição de Dados(DDL)

O Hibernate exporta o esquema de banco de dados a partir dos arquivos de metadados XML. Este esquema de banco de dados pode ser exportado ou para um arquivo .txt, ou diretamente pro banco de dados, permitindo a aplicação criar e apagar tabelas do banco de dados em tempo de execução, tornando-se uma ferramenta muito útil num ambiente de testes. Esta exportação é feita através das classes contidas no pacote *org.hibernate.tool.hbm2ddl*, que fazem a validação, exportação quando uma *SessionFactory* é criada.

Caching [23].

O Hibernate provê de dois níveis de *cache*:

Primeiro nível

Este nível não pode ser desativado, ele trabalha ativamente com o sessão do Hibernate e suas transações. Este nível visa principalmente:

- reduzir o número de consultas SQL geradas por transação;
- Por exemplo, caso um obejto seja modificado muitas vezes numa mesma transação, Hibernate irá gerar apenas um SQL UPDATE no final, contendo todas as modificações;
- Além disso, caso um mesmo objeto seja requisitado duas vezes numa mesma sessão, o Hibernate retornará duas referências deste objeto, ao invés de duas cópias. Isto garante que mudanças feitas numa referência do objeto seja imediatamente visível ao outro código que acessa esta mesma referência através da mesma sessão também.

Segundo Nível

Segundo nível mantém os objetos armazenados no nível do *SessionFactory* (entre as transações). Os objetos são disponíveis para toda a aplicação, não somente para consultas, em particular.

- Deste modo, cada vez que uma consulta retorna um objeto, este já é lido no cache, permitindo potencialmete uma ou mais transações.

2.7.4 Toplink.

TopLink Essentials é uma implementação de código aberto da especificação JPA(Java Persistence API), que provê um padrão POJO de persistência de objetos baseado no mapeamento objeto relacional(ORM). Pelo fato do TopLink Essentials ser uma implementação da especificação JPA e esta basear-se no Hibernate, TopLink e JDO, este framework contém muitas funcionalidades do próprio Hibernate, descrito anteriormente. Porém ele introduziu e padronizou as anotações Java para persistência, em contrapartida aos arquivos XML's. [7]. [12].

Padrão POJO(Plain Old Java Object)

O termo POJO, segundo Wikipedia, é usado principalmente para denotar um objeto Java que não segue qualquer um dos modelos Java de objetos, convenções, ou frameworks como EJB(Enterprise Java Beans). Ou seja, todos os objetos Java, que são POJO's, não são limitados por qualquer restrição, que não seja as próprias da especificação da linguagem Java. Um POJO, portanto, não deve:

1. Herdar uma classe pré-estabelecida

```
1. public class Foo extends javax.servlet.http.HttpServlet{
   ...
```

2. Implementar uma interface pré-estabelecida

```
1. public class Bar implements javax.ejb.EntityBean{ ...
```

3. Ou conter uma anotação Java também pré-estabelecida

```
1. @javax.ejb.Entity
   public class Baz{ ...
```

Porém, por dificuldades técnicas e outras razões, muitas aplicações ou frameworks (no nosso caso, TopLink Essentials) necessitam de anotações Java pré-estabelecidas para funcionar corretamente, como é o caso da persistência, no TopLink Essentials. [24].

Entidades

JPA refere-se a classes passíveis da utilização dos seus serviços como **entidades**. Entidade persistível é uma classe Java, que tipicamente representa uma tabela no banco de dados relacional. Instâncias destas entidades representam tuplas nesta tabela. Entidades usualmente tem relacionamentos com outras entidades, e estes relacionamentos são expressos através de metadados objeto relacionais. [25].

Arquitetura

TopLink Essentials é estruturado da mesma forma que o Hibernate, em três camadas principais:

1. **Gerenciador de conexões;**
2. **Gerenciador de transações;**
3. **Mapeamento Objeto Relacional;**

Assim como o Hibernate, as camadas que mais críticas são a **Gerenciador de conexões** e **Gerenciador de transações**, faltando em performance e capacidade de execução.

Metadados

TopLink Essentials foi a primeira implementação de referência da especificação JPA,mas atualmente temos muitas outras implementações como o próprio Hibernate citado anteriormente. Pelo fato de ser uma implementação de referência, esta baseou-se nos princípios do JPA, introduzindo-nos as anotações Java para persistência, em contrapartida aos arquivos XML's. Segue um exemplo:

```
@Entity
public class Departamento {

        @Id @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;

        private String nome;

        public Departamento() {}
```

```
        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getNome() {
                return nome;
        }

        public void setNome(String nome) {
                this.nome = nome;
        }

}
```

O principal elemento é o *@Entity*, este elemento designa uma classe POJO(Plain Old Java Object) como persistível e/ou utilizável para serviços JPA. Por padrão todas as classes são não-persistíveis e não-elegíveis para a utilização de serviços JPA, a não ser que esteja marcado com esta anotação. Ou seja, esta anotação designa uma classe Java como **entidade**, que foi definido anteriormente. [26].

JPA assume que o nome da tabela associada é o nome da classe entidade. No nosso caso, o nome da tabela seria *Departamento*.

```
@Entity
public class Departamento
```

Caso queira adicionar o nome da tabela associada, basta acrescentar a anotação Java, *@Table*:

```
@Entity
@Table(name="DEP")
public class Departamento
```

Assim como o Hibernate, é preciso especificar o atributo que estará associado a chave primária da tabela no banco de dados. A anotação Java @Id define o atributo que representará a chave primária da entidade e, por padrão, JPA assume que o nome do atributo estará associado a uma coluna com o mesmo nome. No nosso caso, o atributo *id* da classe *Departamento* estará associado a uma coluna chave primária, cujo nome também é *id*.

```
        @Id @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;
```

A anotação Java *@GeneratedValue* gera identificadores únicos para cada instância da entidade,e o próprio JPA provê estratégias para gerar estes identificadores automaticamente.

JPA assume que todos os atributos de uma entidade são persistíveis. Portanto não é preciso adicionar anotação nos atributos.

```
private String nome;
```

Caso este atributo esteja associado a uma coluna, cujo nome não seja o mesmo que o definido por padrão pelo JPA (a coluna associada contém o mesmo nome do atributo), então pode-se acrescentar a anotação Java @Column:

```
@Column(name="NAME")
private String nome;
```

JPA associa este atributo a uma coluna no banco de dados, além de associar a um tipo de dado do modelo relacional, equivalente ao tipo de dado deste atributo do modelo OO e esta equivalência de tipo de dados é feita automaticamente pelo JPA, associando, por exemplo, o atributo *nome* do tipo *java.lang.String* a um tipo de dado equivalente no modelo relacional que é, neste caso, *VARCHAR*. Caso seja necessário especificar o tipo correspondente no modelo relacional, basta acrescentar o parâmetro *columnDefinition* da anotação @*Column*:

```
@Column(name="NAME",columnDefinition=”VARCHAR(255)”)
private String nome;
```

Linguagem de Manipulação de Dados(DML)

O JPA provê, atualmente, apenas duas forma de se trabalhar com a Linguagem de Manipulaçõa de Dados(DML):

1. Java Persistence Query Language(JPQL)
4. SQL Nativo

Java Persistence Query Language(JPQL)

Baseado no EJB QL(Enterprise Java Beans Query Language), JPQL é uma linguagem portável de manipulação de dados combinando uma sintaxe simples de SQL com a expressividade de uma linguagem orientada a objetos. As consultas escritas nesta linguagem são compiladas para a maioria dos servidores de banco de dados. [7].Segue um exemplo:

```
Query q = entityManager.createQuery("select d from Departmento
d");
```

No caso, esta consulta retorna todos os registros da tabela *departamento* associada a classe *Departmento*. O SQL equivalente seria:

```
SELECT * FROM departamento;
```

A sintaxe é muito semelhante ao Hibernate, porém na camada de transação, temos algumas diferenças:

- O Gerenciamento de persistência do Hibernate é feito pela interface *org.hibernate.Session*, obtida através de sua respectiva *Factory*, *org.hibernate.SessionFactory*.
- Já no JPA, o gerenciamento de persistência é controlada pela interface *javax.persistence.EntityManager*, que é obtida através da *Factory*, *javax.persistence.EntiyManagerFactory*.

O JPA permite recuperar o resultado através de uma **java.util.List**, bastando chamar o método **getResultList()** da instância da classe **javax.persistence.Query.**

```
	Query q = entityManager.createQuery("select d from Departmento
d");
	List list = q.getResultList();
```

Além disso, JPA suporta dois tipos de parâmetros nas consultas, parâmetros nomeados e os seqüencias. Com os parâmetros nomeados ao invés de termos consultas como `select d from Departmento d where d.nome = 'Financeira'`, teríamos: [27]

```
	String query = "select d from Departmento d where d.nome =
:nomeProcurado";

	Query q = entityManager.createQuery(query)
	.setParameter("nomeProcurado", "Maria");

	List result = q.getResultList();
```

Já com os parâmetros seqüencias teríamos:

```
	String query = "select d from Departmento d where d.nome = ?1
	and d.lugar = ?2";

	Query q = entityManager.createQuery(query)
	query.setParameter(2, "São Paulo");
	query.setParameter(1,"Maria");
	List result = q.getResultList();
```

As consultas com parâmetros nomeados podem estar definidas através de metadados também:

```
	@NamedQuery (
	name = "recuperaDepartamentosPorNome",
	      query = "select d from Departamento d where d.nome =
	:nomeProcurado"
	)

	public class Departamento {
	...
	}
```

Estas consultas são chamadas através do método **createNamedQuery()**:

```
Query q =
entityManager.createNamedQuery("recuperaDepartamentosPorNome ")
    .setParameter("nomeProcurado ", "Maria")

List departamentos = q.getResultList();
```

SQL Nativo

Outra forma de manipular os dados em JPA é através do SQL nativo. Isto permite expressar as consultas no dialeto SQL específico do banco de dados. As execuções de consultas SQL nativas são executadas através do método *createNativeQuery*. Segue um exemplo simples:

```
    entityManager.createNativeQuery("SELECT * FROM departamento",
Departamento.class).getResultList();
```

Neste caso, como o resultado é limitado a apenas entidades da classe *Departamento*, foi necessário apenas especificar a classe, na qual os dados devem ser retornados, tendo como resultado uma *java.util.Collection* de entidades *Departamento*. Caso o resultado contenha múltiplas entidades, ou os nomes das colunas não sejam compatíveis com o mapeamento objeto-relacional, ou o resultado contenha além entidades, valores escalares também, é necessário utilizar a anotação *@SqlResultSetMapping,* que permite mapear o conjunto de resultados JDBC aos atributos de entidades ou a propriedades e valores escalares. Por exemplo, utilizaremos esta anotação para retornar um resultado que contém duas entidades *Divisao*, *Funcionario* e um escalar *nome*. Neste caso teremos uma lista de vetores de objetos como: `{[Divisao, Funcionario, "Divisao 1"], [Divisao, Funcionario, "Divisao 2"], ...}`.

Entidade *Divisao* com @SqlResultSetMapping: [7]. [26].

```
@SqlResultSetMapping(
    name="resultadoDivisao",
    entities={
        @EntityResult(
            entityClass=Divisao.class,
            fields={
                @FieldResult(name="id",
column="divisao_id"),
             @FieldResult(name="nome",    column="divisao_nome"),
                @FieldResult(name="chefe",
column="divisao_chefe")
            }
        ),
        @EntityResult(
            entityClass=Funcionario.class,
            fields={
                @FieldResult(name="id",
column="funcionario_id"),
                @FieldResult(name="nome",
column="funcionario_nome")
            }
        )
    }
    columns={
```

```
            @ColumnResult(
                name="divisao_nome"
            )
        }
    )
    @Entity
    public class Divisao {
        @Id
        private int id;
        private String nome;
        private Funcionario chefe;
        ...
    }
```

Entidade Funcionario:

```
    @Entity
    public class Funcionario {
        @Id
        private int id;
        private String nome;
        ...
    }
```

Consulta através de SQL Nativo utilizando @SqlResultSetMapping com @EntityResult:

```
    Query q = entityManager.createNativeQuery(
        "SELECT d.id       AS divisao_id, " +
               "d.nome     AS divisao_nome, " +
               "d.chefe    AS divisao_chefe, " +
               "f.id       AS funcionario_id, " +
               "f.nome     AS funcionario_nome, " +
        "FROM Departamento d, Funcionario f " +
        "WHERE (divisao_chefe = funcionario_id)",
        "resultadoDivisao");

    List resultList = q.getResultList();
    // Lista de vetores de objetos: {[Divisao, Funcionario, "Divisao
    1"], [Divisao, Funcionario, "Divisao 2"], ...}
```

Linguagem de Definição de Dados(DDL)

O Toplink Essentials exporta o esquema de banco de dados a partir das entidades JPA. Este esquema de banco de dados pode ser exportado ou para um arquivo .sql, ou diretamente pro banco de dados, permitindo a aplicação criar e apagar tabelas do banco de dados em tempo de execução, tornando-se uma ferramenta muito útil num ambiente de testes. Esta exportação é feita através da classe *oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider*, que contém constantes String pré-definidas que especificam a exportação do esquema, o tipo de saída. Por exemplo, basta acrescentar uma *property* no arquivo *persistence.xml*.

```
    <property name="toplink.ddl-generation" value="create-tables"/>
```

Caching [26].

TopLink Essentials provê dois tipos de *cache*, *session cache* e *unit of work cache*. *Unit of work cache* provê cache baseado no padrão *unit of work*, descrito por Martin Fowler: [18].

"Mantém uma lista de objetos afetados por uma transação de negócios e coordena a escrita de alterações e resoluções de problemas de concorrência. "

Session Cache

O cache de sessão é um cache compartilhado que permite serviços de clientes serem acoplados para uma dada sessão. Quando objetos são recuperados ou inseridos num banco de dados através de uma sessão cliente, TopLink Essentials armazena uma cópia dos objetos no cache da sessão pai e torna estes acessíveis a todos os outros processos da sessão. Os objetos são armazenados no cache da sessão a partir de:

- Armazenamento de dados, quando TopLink Essentials executa uma operação de leitura;
- *Unit of work cache*, quando *units of work* enviam com sucesso uma transação.

Unit of Work Cache

Unit of work cache mantém e isola os objetos do cache de sessão, além de escrever as alterações ou os novos objetos para o cache, após *unit of work* enviar as mudanças para o banco de dados.

3 Métricas.

3.1 Introdução e Conceitos.

3.1.1 Introdução.

Nas seções anteriores, introduzimos os conceitos de persistência de dados, de persistência de objetos, de orientação a objetos, do modelo de dados relacional e descrevemos como esses conceitos interagem para criar uma noção sobre a idéia do **mapeamento objeto-relacional**. Além disso, discutimos sobre o **conflito de impedância** e os empecilhos enfrentados pelo mapeamento objeto-relacional.

Esse conjunto de adversidades existentes no ORM representa um indicativo da complexidade elevada encontrada pelas ferramentas, que aplicam a técnica, de realizar sua tarefa. Esse fator, aliado ao fato de que há muitas maneiras de se construir uma ferramenta e aplicar o conceito do ORM, reforça a necessidade de se avaliar a qualidade **da persistência de dados** gerada

pelas ferramentas, e a qualidade da aplicação do mapeamento objeto-relacional.

Motivado por essa questão, nosso trabalho, então, tem como objetivo criar um conjunto de métricas que sirvam como um meio para **avaliar a persistência de dados** no ambiente objeto-relacional. E é o que explicaremos melhor na seção seguinte.

3.1.2 Conceitos gerais.

De forma geral, uma **métrica** pode ser definida como uma *maneira de medir, mensurar alguma propriedade*. Por exemplo, para medir a altura de um edifício poderíamos utilizar as medidas em centímetros ou polegadas como métricas; para medir a massa de uma pessoa poderíamos utilizar o quilograma como uma possível métrica; para medir o volume de uma bexiga, uma métrica plausível seria utilizar a medida em centímetros cúbicos. Podemos medir através de métricas conceitos mais abstratos como o desempenho de uma equipe num jogo de futebol. Para isso, poderíamos utilizar o número de gols, faltas, penalidades máximas, cartões recebidos pelos jogadores, tempo de posse da bola, etc.

Em Engenharia de Software, as métricas são muito utilizadas para medir a qualidade dos processos de produção de software e o próprio software. Número de linhas de código, total de classes do programa, tempo e custo levado para produzir o software são métricas comumente utilizadas no ramo.

Dessa forma, elaboramos as métricas separando-as por **categorias** que são, na realidade, a propriedade que elas devem medir e, assim, cada propriedade possui então seu conjunto de métricas. Além disso, classificamos cada métrica como sendo **quantitativa** ou **qualitativa**. Informalmente, métricas quantitativas são aquelas que utilizam informações que se podem “contar” como números coletados, conceitos físicos, espaciais e temporais (distâncias, tempo, velocidade), etc. De forma contrária, métricas qualitativas não possuem “algo para contar diretamente”, elas utilizam fatores subjetivos como, por exemplo, a satisfação de um consumidor com um produto. As métricas qualitativas se baseiam nos conceitos de qualidade, tipo e gênero.

Em suma, nas próximas seções, iremos apresentar cada **propriedade** com seu título e sua descrição e, sob cada propriedade, seu conjunto de **métricas** com suas respectivas descrições e nomes.

3.2 Propriedades e Métricas.

3.2.1 Propriedade 1: Manipulação dos dados no ambiente objeto-relacional.

Descrição:

Como explicado anteriormente, no ambiente objeto-relacional, os dados ora são representados no modelo de objetos ora são representados no modelo relacional. Uma medida importante é a **eficiência** com que os dados são manipulados de um meio para o outro, ou seja, a eficiência na transformação dos dados no contexto objeto-relacional.

Por exemplo, suponha a existência de um objeto pertencente à classe **Funcionário** que possui os atributos **nome**, **idade** e **ID**. Essa é a maneira como os dados são visualizados no modelo de objetos. Já no modelo relacional, esse objeto poderia ser representado, por alguma implementação do ORM, através de uma tabela denominada **FUNCIONARIO** que possuísse os campos denominados **nome**, **idade** e **ID** para representar os respectivos atributos do objeto. Nos casos mais simples como esse, a técnica que acabamos de citar é praticamente um padrão entre as implementações, mas para outros casos, como o que foi citado anteriormente que envolve a decisão de se mapear ou não atributos em mais de uma tabela, existem variações quanto ao procedimento utilizado.

Por essas razões, consideramos que a eficiência da manipulação dos dados no ambiente objeto-relacional é uma propriedade fundamental. E, por isso, definimos o conjunto de métricas a seguir o qual procura medir justamente essa propriedade.

MÉTRICA 1.

Nome: **Inserção de uma entidade na especialização disjunta**.

Tipo: Quantitativa.

Descrição:

Essa métrica tem a função de medir a eficiência da inserção de uma entidade de **especialização disjunta**, definida pelo item 2.1.2.9 da seção de conceitos do modelo de objetos, no modelo relacional. O tempo da inserção será a medida utilizada pela métrica.

MÉTRICA 2.

Nome: Recuperação dos objetos que estão num nível N de uma árvore de relacionamentos binários recursivos.

Tipo: Quantitativa.

Descrição:

Suponha uma hierarquia de objetos, representada por uma árvore, no qual cada nó da árvore possui no máximo 2 filhos. As ligações entre os elementos da árvore, arestas, representam instância de **relacionamentos binários recursivos** (conforme definido pelo item 2.1.4.8), pois todos os objetos pertencem à mesma entidade.

Definimos como **nível** (altura) n da árvore como sendo a distância entre o nó e a raiz da árvore.

Por exemplo, seja três instâncias e1,e2 e e3 da entidade Funcionário, no qual e1 é definida como raiz e que estejam relacionadas da seguinte forma:

r1(e1,e2);r2(e2,e3), no qual ri, i = 1,2 são relacionamentos binários recursivos. Logo e2 está no nível 1, enquanto e3 está no nível 2 da árvore de hierarquia.

A métrica 2 visa medir a eficiência da recuperação dos objetos que estão num nível N da árvore. A medida será o tempo de recuperação

MÉTRICA 3.

Nome: Recuperação de entidades num relacionamento M-N.

Tipo: Quantitativa.

Descrição:

Seguindo a definição dada pela seção 2.1.4.8**,** fixada uma instância de entidade **A** participante de um conjunto de relacionamentos binários de cardinalidade muitos para muitos, queremos encontrar todas as instância que estão associadas à entidade A neste conjunto de relacionamentos.

Exemplificando, imagine uma instância da entidade Engenheiro que se associa à várias instâncias da entidade Associação, através de um relacionamento binário muitos-para-muitos. A definição acima, neste caso, consistiria em recuperar todas as instâncias de entidade Associação ao qual a instância de entidade Engenheiro em questão está associada.

A métrica 3 objetiva medir a eficiência da recuperação dessas instâncias de entidade num relacionamento M-N. Novamente, o tempo de recuperação será a medida.

MÉTRICA 4.

Nome: Busca em profundidade no modelo.

Tipo: Quantitativa.

Descrição:

Definição recursiva:

1 - Dentro do modelo conceitual de uma aplicação, selecione um tipo de relacionamento **A**.

2 - Escolha um elemento **a1** do conjunto de todas as instâncias de relacionamento de **A**.

3 - Em **a1,** selecione uma das instâncias de entidade participante e chame-a de **e1**.

4- Agora, escolha o conjunto **C** de todas as instâncias de relacionamento de **A** do qual **e1** participar.

5- Em **C** escolha o conjunto **D** de todas as instâncias de entidade que participam junto com **e1** de algum elemento de **A**.

6- Escolha uma das opções:

a) Escolha algum elemento **f1** do conjunto **F** de todos os tipos de relacionamento dos quais os integrantes de **D** possam participar, se esse conjunto for não-vazio. Caso seja vazio pare e a reposta é o conjunto **D**.

b) Pare e a resposta é **D**.

7- Se não parou na etapa anterior, volte para a etapa 2, porém considere **A** como sendo o conjunto **F** e **a1** como sendo **f1**.

A métrica 4 utiliza o tempo gasto para encontrar D.

MÉTRICA 5.

Nome: Criação de uma árvore de relacionamentos binários recursivos de nível N.

Tipo: Quantitativa.

Descrição: Esta métrica mede o tempo gasto para se realizar a operação de se construir uma árvore de relacionamentos binários recursivos – definida na descrição da métrica 2 - de nível N.

MÉTRICA 6.

Nome: Criação de conjunto de N instâncias de relacionamentos binários de cardinalidade muitos-para-muitos.

Tipo: Quantitativa.

Descrição: Esta métrica mede o tempo gasto para se criar um conjunto de N instâncias de relacionamentos binários de cardinalidade muitos-para-muitos – conceito definido pela métrica 2.

3.2.2 Propriedade 2: Representação dos dados no modelo de objetos.

MÉTRICA 1.

Nome: Suporte aos vários tipos de representação de herança.

Tipo: Qualitativa.

Descrição: Atualmente, para se representar uma **relação de herança** do modelo de orientação a objetos, existem várias **estratégias** de comum utilização. Como citamos anteriormente, sendo que algumas das principais formas são: table per class hierarchy, table per subclass, table per concrete class. Está métrica visa classificar uma implementação de mapeamento objeto-relacional, conforme o suporte a tais estratégias.

MÉTRICA 2.

Nome: Relacionamentos n-ários.

Tipo: Qualitativa.

Descrição: Esta métrica visa classificar as implementações de mapeamento objeto-relacional conforme o suporte ou não aos **relacionamentos n-ários**, conforme definido anteriormente.

MÉTRICA 3.

Nome: Suporte das restrições de integridade entre os modelos relacional e de objetos.

Tipo: Qualitativa.

Descrição: Esta métrica classifica as implementações de mapeamento objeto-relacional perante o suporte ao conjunto de restrições de integridade do modelo relacional.

MÉTRICA 4.

Nome: Suporte à abstração de agregação.
Tipo: Qualitativa.

Descrição: Esta métrica qualifica uma implementação de mapeamento objeto-relacional a respeito do suporte a representação da abstração de agregação, definida pela seção de conceitos do modelo relacional de dados.

MÉTRICA 5.

Nome: Suporte à especialização sobreponível.
Tipo: Qualitativa.

Descrição: A intenção da métrica 6 é avaliar uma implementação de mapeamento objeto-relacional objeto-relacional quanto ao suporte às especializações sobreponíveis.

4 Testes e Resultados.

4.1 Introdução.

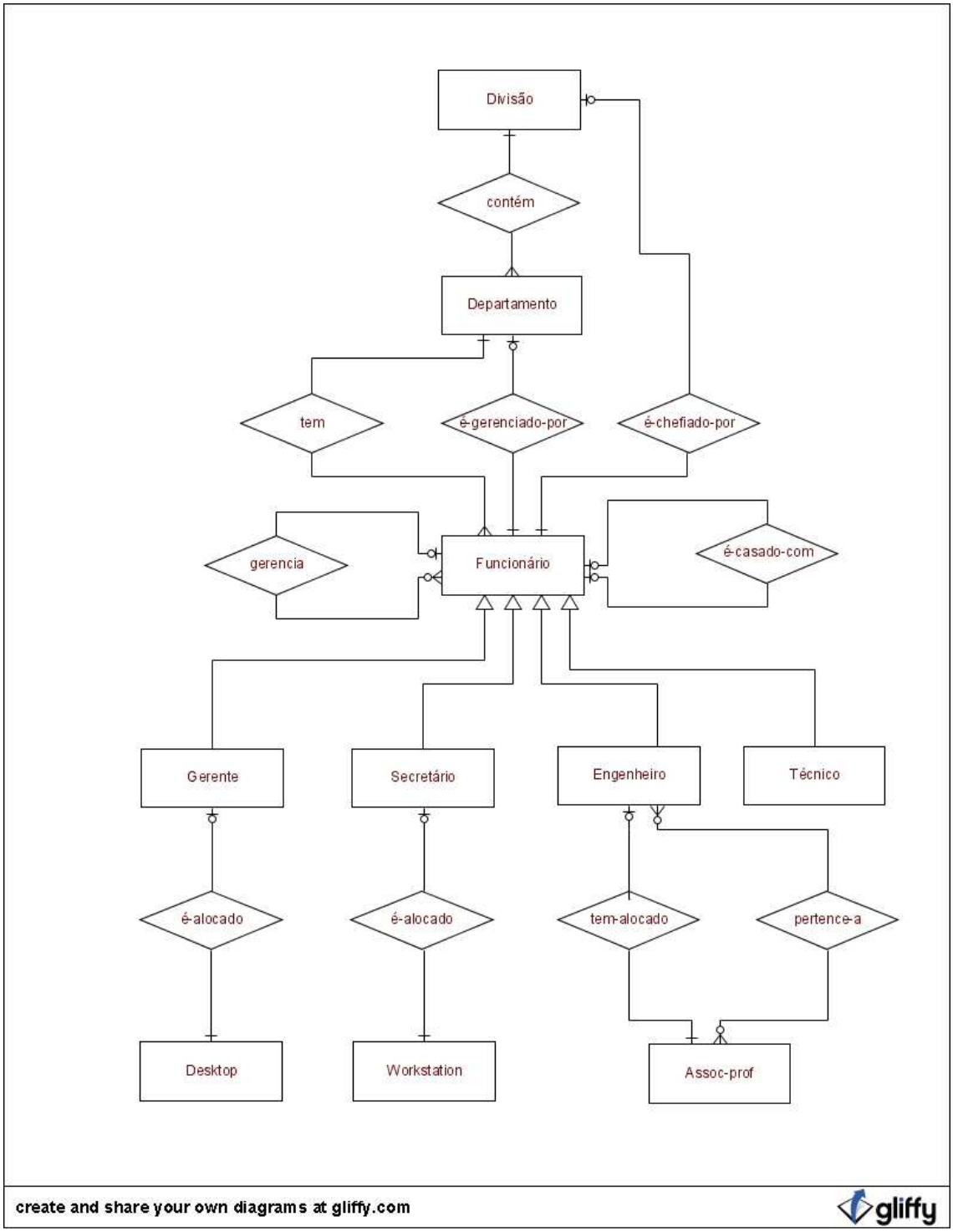
Na seção anterior, elaboramos o conjunto de métricas, separadas por suas respectivas propriedades. Agora, o objetivo desta parte do nosso trabalho, é testar as implementações de mapeamento objeto-relacional, que escolhemos estudar, perante o conjunto de métricas elaboradas.

Para fazer parte dos testes, utilizamos o arcabouço de testes **Poleposition** [5], que detalharemos mais adiante. Em suma, o Poleposition provê mecanismos para comporar a performance de implementações de ORM num ambiente unificado.

De início, para realizar os testes por meio do Poleposition, codificamos, utilizando cada umas das 3 ferramentas de ORM escolhidas, 3 aplicações de testes, porém optamos por utilizar apenas as aplicações construídas com o Hibernate e o Toplink Essentials para rodar testes no Poleposition. O Active

Record do Ruby não foi utilizado nesta etapa de testes de performance, devido às limitações do arcabouço Poleposition e da versão do interpretador JRuby que existia na fase inicial dos testes.

As 3 aplicações baseiam-se em um modelo conceitual comum, adaptado do livro do Teorey [4]:



Nas próximas seções, explicaremos os detalhes de como realizamos nosso trabalho e os resultados obtidos. Descreveremos o funcionamento do

ambiente de testes Poleposition e os principais conceitos envolvidos. Além disso, exibiremos os resultados dos testes acompanhados da avaliação das métricas.

4.2 Atividades realizadas.

Dividimos nosso projeto em 5 fases:

1. Construção da aplicação modelo:
    a. Construção de aplicações modelo com um esquema comum de BD cada qual utilizando uma ferramenta específica para realizar o ORM.

2. Preparação do ambiente de testes:
    a. Preparação do ambiente de testes;
    b. Preparação das aplicações para serem utilizadas dentro do ambiente de testes.

3. Elaboração das métricas:
    a. Definição das métricas.

4. Realização dos testes:
    a. Execução dos códigos de testes utilizando o Polepositon;
    b. Avaliação das métricas qualitativas.

5. Análise dos resultados:
    a. Avaliação dos testes utilizando as métricas.

4.3 Arcabouço de testes Poleposition.

Como mencionado, utilizamos o arcabouço testes **Poleposition** (**http://polepos.sourceforge.net/**), licenciado sob a **GPL** (http://www.gnu.org/copyleft/gpl.html), para realizar os testes de desempenho relacionados às nossas métricas.

Abstratamente, o arcabouço, como o próprio nome indica, “organiza uma corrida” em que os participantes podem ser “engines” de bancos de dados ou ferramentas de mapeamento objeto-relacional. Os participantes “correm” em um “circuito” e o arcabouço monitora a **corrida**, produzindo relatórios sobre o desempenho de cada participante. Um **circuito** é um conjunto de “**voltas**” que os participantes precisam percorrer. As voltas representam, na realidade, **operações** realizadas pelos participantes sobre o mesmo conjunto de dados. Essas operações são as tarefas em que se deseja comparar o desempenho dos participantes. Ou seja, é definida uma tarefa a ser realizada e cada participante a implementa numa volta no circuito. Exemplo de operações:

persistir um conjunto de objetos, realizar operações CRUD (create, read, update, delete) no banco de dados relacional. Os participantes são representados por **carros** que correm no circuito. Um conjunto de participantes do mesmo tipo, por exemplo, bancos de dados do mesmo gênero que requerem código semelhante, formam um **time**.

A classe **Circuit** equivale ao conceito de circuito que descrevemos. Da mesma forma, as classes **Lap**, **Car** e **Team** representam respectivamente uma volta, um carro e um time.

4.4 Resultados e avaliação das métricas.

4.4.1 Testes da propriedade 1: Manipulação dos dados no ambiente objeto-relacional.

Pelo fato desta propriedade medir a eficiência do mapeamento objeto-relacional, definimos as métricas como mensuráveis quantitativamente com relação ao tempo (ms). Nesta seção iremos mostrar os resultados de cada métrica estipulada, além disso, no início de cada métrica, explicaremos a operação realizada para mensurar tal, seguido de uma análise do gráfico logo abaixo. O código-fonte está disponível no repositório do projeto. [29].

Métrica 1: Inserção de uma entidade na especialização disjunta.

Operação: Implementamos uma operação de inserção numa especialização disjunta através de inserções de instâncias de entidades Gerente do modelo conceitual exposto anteriormente.

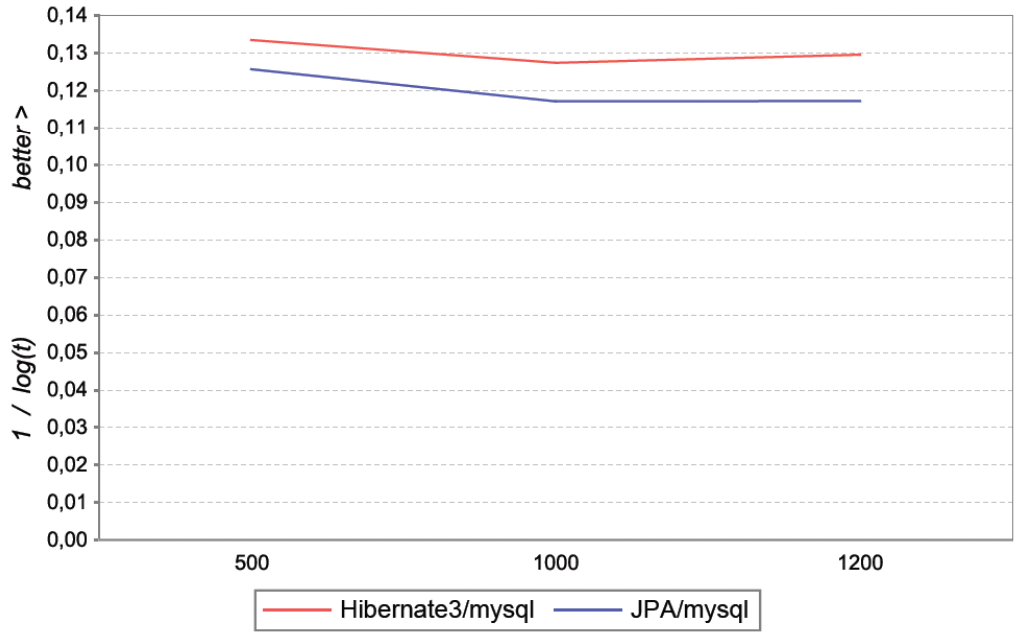
Análise: Pela figura abaixo, Hibernate é mais eficiente que o TopLink Essentials, no qual fica evidente a inclinação maior de crescimento do Hibernate, com relação ao TopLink Essentials.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: insercao_gerente

t [time in ms]	**selectdepth:3 objects:500 depth:7**	**selectdepth:6 objects:1000 depth:9**	**selectdepth:6 objects:1200 depth:11**
Hibernate3/mysql	1791	2560	2245
JPA/mysql	2848	5094	5055



Métrica 2: Recuperação dos objetos que estão num nível N de uma árvore de relacionamentos binários recursivos.

Operação 1: Implementamos, para esta métrica, uma operação de recuperação de objetos, para um dado nível N, de uma árvore de relacionamentos binários recursivos através da recuperação de instâncias de entidades de Funcionários participantes de um conjunto de relacionamentos (relacionamento *gerencia* do modelo de dados conceitual), no qual obedece a regra estabelecida pela definição da métrica 2.

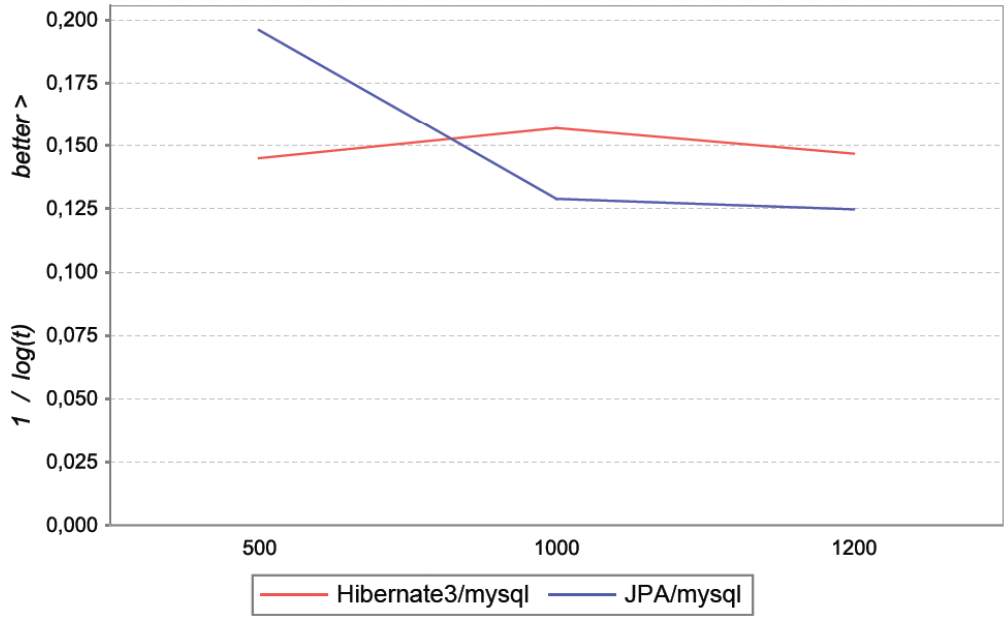
Análise: Nesta figura, tivemos pontos nos quais TopLink Essentials foi mais eficiente que o Hibernate, mas, em geral, ele não supera o Hibernate nesta métrica.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: consulta_relacionamento_recursivo

t [time in ms]	selectdepth:3 objects:500 depth:7	selectdepth:6 objects:1000 depth:9	selectdepth:6 objects:1200 depth:11
Hibernate3/mysql	989	585	909
JPA/mysql	164	2333	3018



Operação 2: Esta operação é semelhante a anterior, porém consiste na análise da funcionalidades das implementações referentes a performance, como o caching. Para isto, é feita a chamada de função, posteriormente a execução da *operação 1* referente a métrica 2.

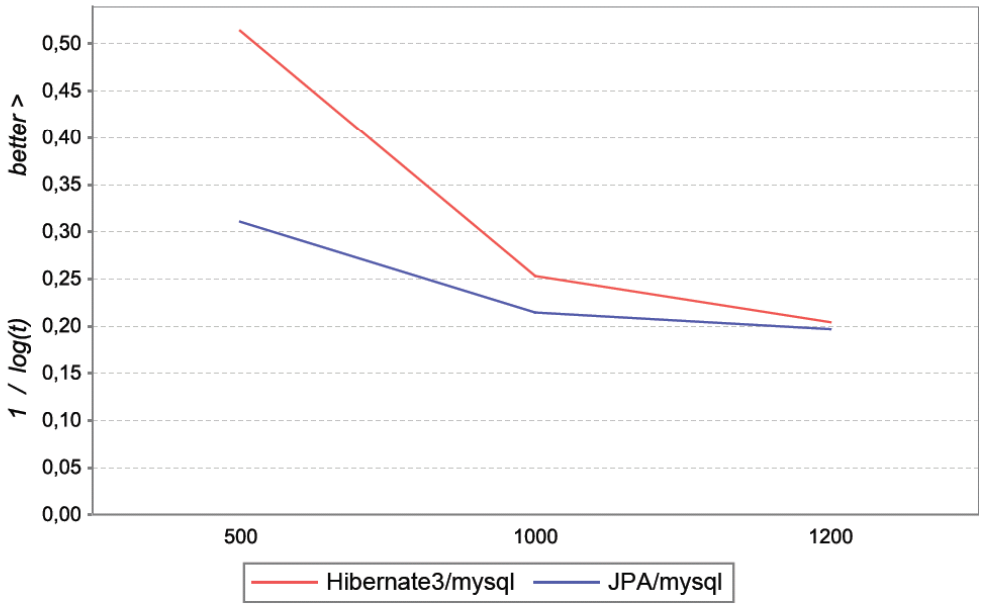
Análise: Neste resultado, podemos perceber o substancial ganho de performance de execução, perante a *operação 1* executada anteriormente. Na figura abaixo, podemos ver que tanto o Hibernate, quanto o TopLink Essentials têm performances semelhantes, não havendo diferenças críticas ou relevantes entre eles.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: consulta_relacionamento_recursivo_hot

t [time in ms]	**selectdepth:3 objects:500 depth:7**	**selectdepth:6 objects:1000 depth:9**	**selectdepth:6 objects:1200 depth:11**
Hibernate3/mysql	7	52	134
JPA/mysql	25	106	161



Métrica 3: Recuperação de entidades num relacionamento M-N.

Operação: Esta operação visa mensurar a performance de cada implementação ORM perante a recuperação de entidades participantes de um conjunto de relacionamentos binários com cardinalidade M-N (muitos-para-muitos).

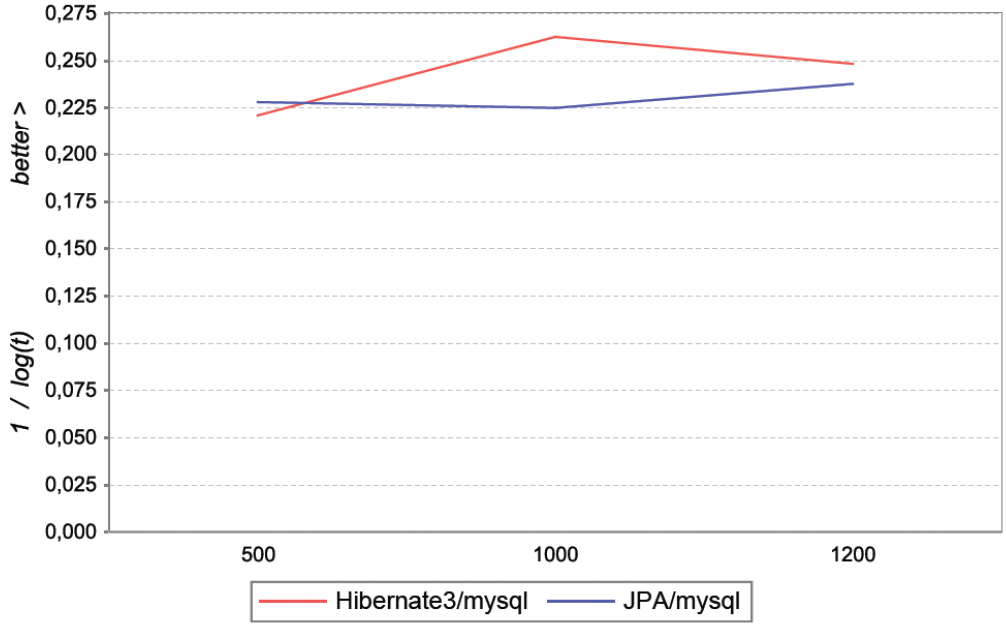
Análise: Neste resultado, assim como o anterior, não temos necessariamente uma diferença substancial entre as implementações, porém o Hibernate ainda continua tendo as melhores performances.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: consulta_relacionamento_mn

t [time in ms]	selectdepth:3 objects:500 depth:7	selectdepth:6 objects:1000 depth:9	selectdepth:6 objects:1200 depth:11
Hibernate3/mysql	92	45	56
JPA/mysql	80	85	67



Métrica 4: Busca em profundidade no modelo.

Operação 1: Esta operação, a partir de muitas operações do tipo JOIN, visa medir a eficiência das implementações quanto a métrica 3 descrita anteriormente.

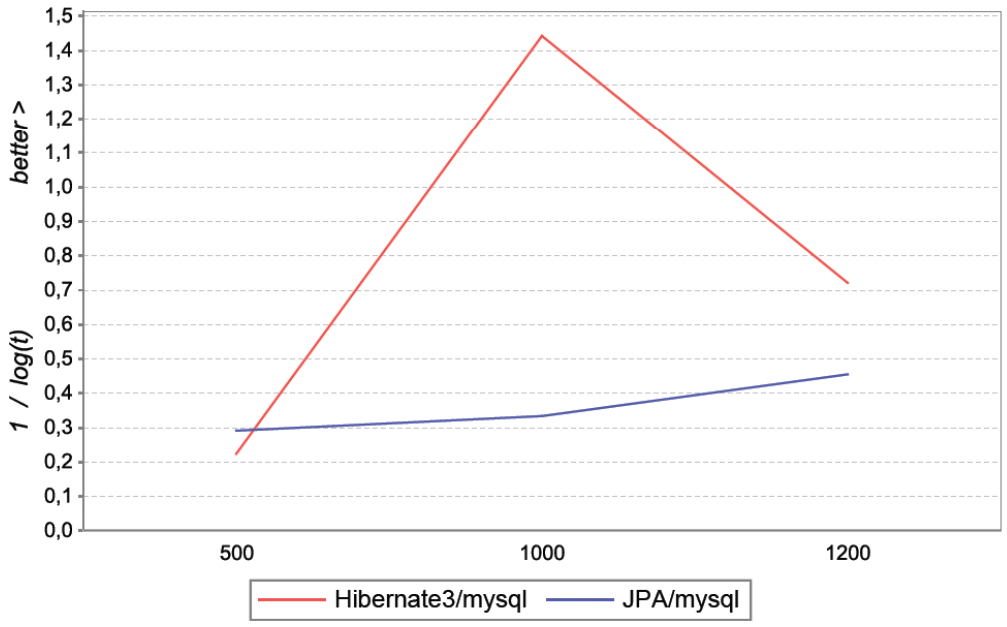
Análise: Nesta figura, temos uma relativa superioridade do Hibernate quanto a performance, apesar de inicialmente TopLink Essentials ter obtido um ganho com relação ao Hibernate.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: buscaProfundidadeModelo

t [time in ms]	selectdepth:3 objects:500 depth:7	selectdepth:6 objects:1000 depth:9	selectdepth:6 objects:1200 depth:11
Hibernate3/mysql	87	2	4
JPA/mysql	31	20	9



Operação: Esta operação visa medir a eficiência as funcionalidades de caching das implementações ORM.

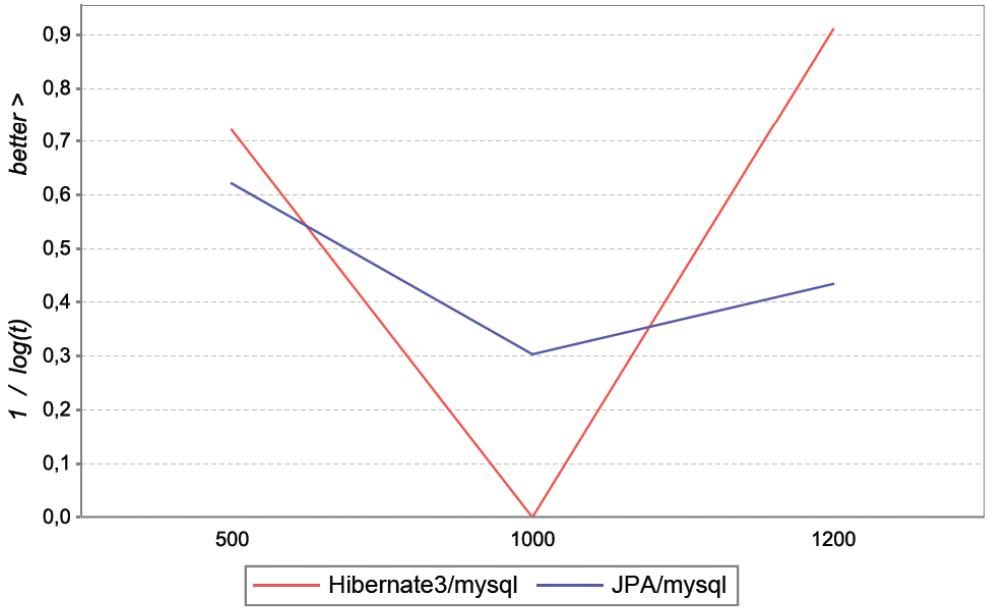
Análise: Neste resultado, não temos diferenças consideráveis de performance entre o Hibernate e o TopLink Essentials. O Hibernate continua obtendo resultados melhores do que os obtidos pelo TopLink Essentials.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: buscaProfundidadeModelo_hot

t [time in ms]	**selectdepth:3 objects:500 depth:7**	**selectdepth:6 objects:1000 depth:9**	**selectdepth:6 objects:1200 depth:11**
Hibernate3/mysql	4	1	3
JPA/mysql	5	27	10



Métrica 5: Criação de uma árvore de relacionamentos binários recursivos de nível N.

Operação: Esta operação, visa mensurar a eficiência das implementações ORM para com a criação e execução de sequências de operações de INSERT e UPDATE, que são necessárias para a criação de árvore de relacionamentos binários recursivos.

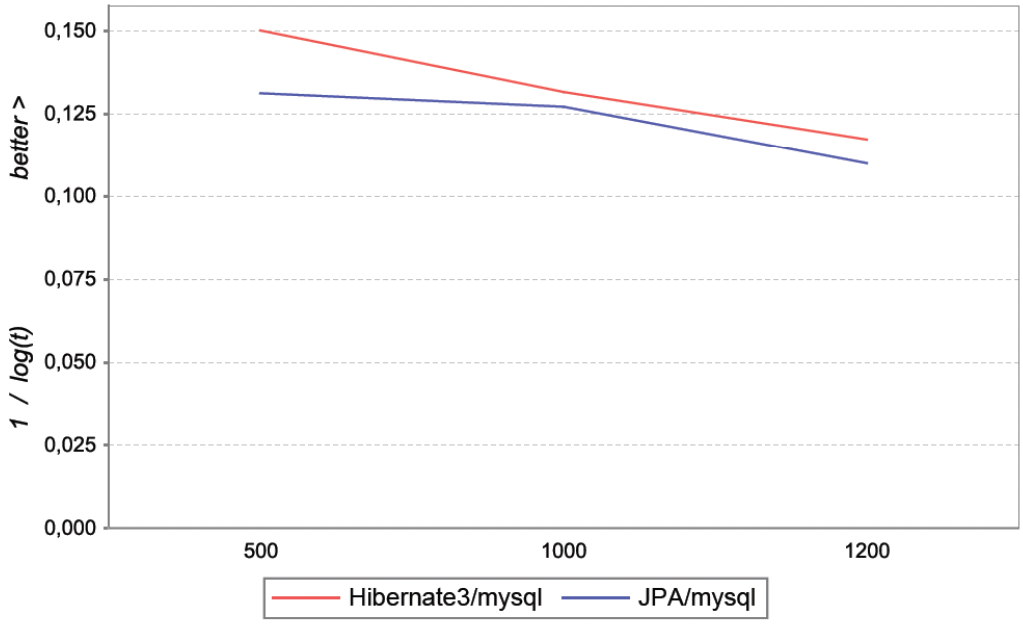
Análise: Neste resultado, percebemos a substancial superioridade do Hibernate, que apresenta quedas não muito bruscas, em relação ao TopLink Essentials.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: insercaoArvoreModelo

t [time in ms]	**selectdepth:3 objects:500 depth:7**	**selectdepth:6 objects:1000 depth:9**	**selectdepth:6 objects:1200 depth:11**
Hibernate3/mysql	780	1990	5039
JPA/mysql	2035	2594	8899



Métrica 6: Criação de conjunto de N instâncias de relacionamentos binários de cardinalidade muitos-para-muitos.

Operação: Esta operação, visa mensurar a eficiência das implementações ORM para com a criação e execução de sequências de operações de INSERT em tabelas que participam do cojunto de relacionamentos binários de cardinalidade muitos-para-muitos.

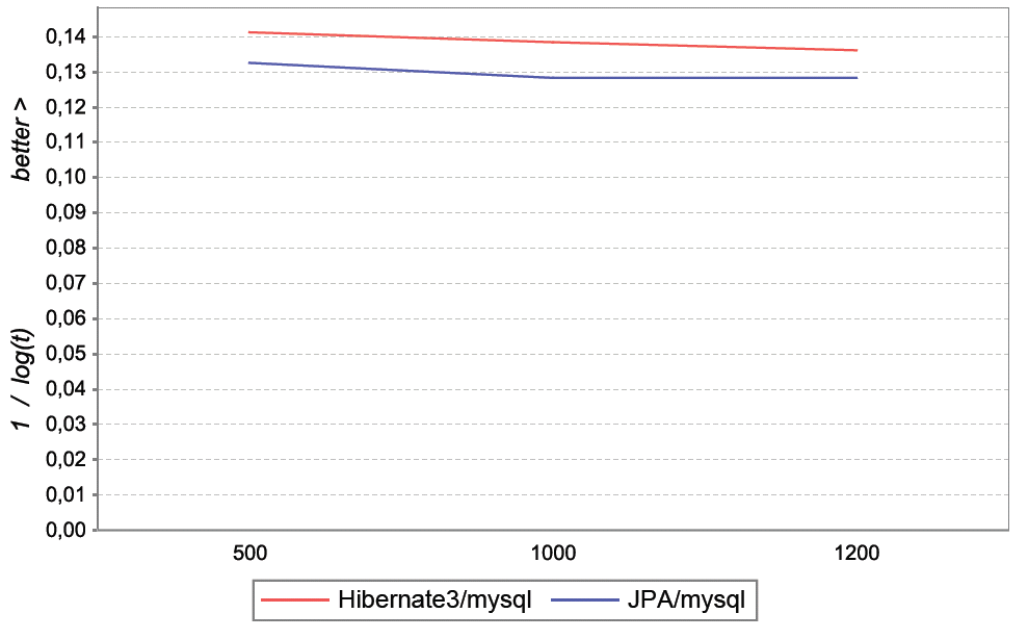
Análise: Neste resultado, percebemos a superioridade do Hibernate, que apresenta uma queda linear e constante, enquanto que o Toplink Essentials apresenta tal de forma não linear.

Circuit: Matao

writes, reads and deletes unstructured flat objects of one kind in bulk mode

Lap: insercao_relacionamento_mn

t [time in ms]	selectdepth:3 objects:500 depth:7	selectdepth:6 objects:1000 depth:9	selectdepth:6 objects:1200 depth:11
Hibernate3/mysql	1182	1365	1541
JPA/mysql	1877	2411	2408



4.4.2 Testes da propriedade 2: Representação dos dados no modelo de objetos.

A função desta etapa do nosso trabalho é avaliar qualitativamente a propriedade 2, através de seu conjunto de métricas.

Métrica 1: Suporte aos vários tipos de representação de herança.

Análise: A equipe de desenvolvimento do Active Record, optou por limitar a estratégia de herança somente ao uso do table per class hierarchy (Single Table Inheritance), para seguir a filosofia, “convention over configuration”. Uma das conseqüências imediatas do uso desta estratégia é o fato de que o uso de uma tabela única para representar a hierarquia de classes, implica na existência de tabelas desnormalizadas, contendo muitas colunas com valores nulos.

No caso do Hibernate, o suporte é mais abrangente, cobrindo todas as opções. Já no caso Toplink Essentials, apesar da especificação do JPA permitir o uso das 3 opções, só foram cobertas 2 das 3 estratégias.

De qualquer, os frameworks não escolhem automaticamente a melhor estratégia para cada caso de representação do domínio de aplicação. O desenvolvedor precisa decidir qual estratégia será utilizada configurando o ambiente por conta própria.

Ferramenta	**Table per class hierarchy**	**Table per subclass**	**Table per concrete class**
Active Record	X	N/A	N/A
Hibernate	X	X	X
Toplink Essentials	X	X	N/A

Métrica 2: Relacionamentos n-ários.

Análise: Os relacionamentos binários são amplamente cobertos por todas as implementações. No caso dos relacionamentos ternários, apenas o Hibernate possui mapeamento nativo. E no caso de relacionamentos n-ários com N maior que 3, nenhuma ferramenta possui suporte nativo. O desenvolvedor precisaria implementar a lógica do relacionamento na aplicação, definindo todas as restrições de integridade e fazendo o tratamento de exceções.

Ferramenta	N=2	N=3	N>3
Active Record	X	N/A	N/A
Hibernate	X	X	N/A
Toplink Essentials	X	N/A	N/A

Métrica 3: Suporte das restrições de integridade entre os modelos relacional e de objetos.

Análise: No caso das restrições de integridade, o Hibernate e o Toplink Essentials criam no banco de dados relacional, a partir dos metadados de mapeamento, o esquema com todas as restrições de integridade. O único que verifica a existência ou não de "constraints" no banco de dados é o Hibernate. O Toplink Essentials apenas as cria, mas não faz verificações e atualizações ao iniciar a execução de uma aplicação. Não existe nenhum suporte no caso do Active Record.

Ferramenta	Not Null	Unique	Check	Foreign Key	Composite Key
Active Record	X	N/A	N/A	N/A	N/A
Hibernate	X	X	N/A	X	X
Toplink Essentials	X	X	N/A	X	X

Métrica 4: Suporte à abstração de agregação.

Análise: Apenas o Active Record possui suporte nativo. Nos outros frameworks, o caso é análogo ao dos relacionamentos n-ários com n maior que 3. O desenvolvedor precisa fazer algum "workaround" para representar a abstração de agregação.

Ferramenta	Agregação
Active Record	X
Hibernate	N/A
Toplink Essentials	N/A

Métrica 5: Suporte à especialização sobreponível.

Análise: Em nenhum dos casos, existe suporte para representar uma hierarquia de classes com especializações sobreponíveis, pois as linguagens em que as ferramentas são implementadas não adotaram em suas implementações esse conceito pertencente ao modelo de orientação a objetos.

Ferramenta	Especialização sobreponível
Active Record	N/A
Hibernate	N/A
Toplink Essentials	N/A

5 Conclusão.

A partir das informações coletadas, dos testes realizados e de todo nosso estudo sobre o mapeamento objeto-relacional, pode-se concluir que ainda existem muitos problemas pendentes precisando de solução:

- A performance ainda precisa ser melhorada;
- A escalabilidade precisa ser maior possibilitando trabalhar com um volume maior de dados;
- A representação semântica dos dados precisa ser melhor elaborada;
- O conflito de impedância não foi resolvido totalmente.

Na questão da performance, através dos experimentos, verificou-se que a realização de operações custosas em termos de memória e processamento se tornam impraticáveis dentro de um ambiente que utiliza as ferramentas estudadas. É uma prática comum implementar tais operações complexas, custosas e com alto “overhead” através de meios como “stored procedures”, entretanto como já dissemos, implementá-las utilizando as ferramentas estudadas se mostrou inviável.

O fator acima implica diretamente em pontos como o da escalabilidade. Por exemplo, o processamento necessário para um computador realizar a operação de inserção de uma árvore de relacionamentos binários demonstrou ser exponencial em relação ao número de objetos da árvore, dificultando, portanto, a escalabilidade do sistema.

6 Bibliografia

[1] Elmasri, Ramez. Navathe , Shamkant B. Fundamentals of Database Systems, 4th Edition, Addison-Wesley, 2004.

[2] Ferreira, João Eduardo. Takai, Osvaldo Kotaro. Italiano, Isabel Cristina. INTRODUÇÃO A BANCO DE DADOS, 2005.

[3] Silberschatz, Abraham. Korth, Henry F. . Sudarshan, S. . Database System Concepts, 5th Edition, McGraw-Hill, 2005.

[4] Lightstone, Sam. Teorey, Toby. Nadeau, Tom. Database Modeling and Design Logical Design, 4th Edition, Morgan Kaufmann, 2006.

[5] Poleposition: the open source benchmark. URL: http://www.polepos.org/

[6] Bauer, Christina. King, Gauvin. Java Persistence with Hibernate. Manning, 2007.

[7] Keith, Mike. Schincariol, Merrick. PRO EJB3:Java Persistence API. Apress, 2006.

[8] Thomas, Dave. Hansson, David H.. Agile Web Development with Rails: Second Edition. The Pragmatic Bookshelf, 2006.

[9] Ruby on Rails: Active Record. Hansson, David Heinemeier. 2007. URL: http://wiki.rubyonrails.org/rails/pages/ActiveRecord.

[10] Broinizi, Marcos Eduardo Bolelli. Validação ágil e precisa de projetos conceituais de banco de dados. IME-USP, 2007.

[11] Hibernate Reference. Hibernate. Red Hat, 2006. URL: http://www.hibernate.org/hib_docs/v3/reference/en/html/

[12] Java Persistence API FAQ. Sun Microsystems, 2007. URL: http://java.sun.com/javaee/overview/faq/persistence.jsp

[13] Wegner, Peter. Concepts and paradigms of object-oriented programming. Volume 1 , Issue 1, ACM SIGPLAN OOPS Messenger, 1990. p.7-87.

[14] Jordan, Mick. A Comparative Study of Persistence Mechanisms for the Java™ Platform. 2004. URL: http://research.sun.com/techrep/2004/

[15] Zyl, Pieter Van. Kourie, Derrick G.. Boake, Andrew. Comparing the Performance of Object Databases and ORM Tools. ACM International Conference Proceeding Series, Vol. 204, 2006.

[16] Tuppa, Walter. VMAKE - A CASE-Oriented Configuration Management Utility, 1996. URL: http://www.iue.tuwien.ac.at/phd/tuppa/node7.html

[17] Peak, Patrick. Hibernate vs. Rails: The Persistence Showdown. 2005. http://www.theserverside.com/tt/articles/content/RailsHibernate/article.html

[18] Fowler, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley, 2003.

[19] Tate, Bruce. Crossing borders: Exploring Active Record. IBM, 2006. URL: http://www.ibm.com/developerworks/java/library/j-cb03076/index.html

[20] Kumar, Deepak. Complete Hibernate 3.0 Tutorial. 2006. URL: http://www.roseindia.net/hibernate/hibernate_architecture.shtml

[21] Enterra Inc.. Hibernate to Support Custom Domain Object Fields. InfoQUsing, 2005. URL: http://www.infoq.com/articles/hibernate-custom-fields

[22] Lorimer, R.J. Hibernate: Externalize HQL Queries. JavaLobby, 2005. URL: http://www.javalobby.org/java/forums/m91885316.html

[23] Sang, Shin. Hibernate Caching. Java Passion. URL: http://www.javapassion.com/j2ee/hibernatecaching.pdf

[24] Plain Old Java Object. Wikipedia. URL: http://en.wikipedia.org/wiki/Plain_Old_Java_Object

[25] Java Persistence API. Wikipedia. URL: http://en.wikipedia.org/wiki/Java_Persistence_API

[26] JPA Annotation Reference. Oracle, 2007. URL: http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html

[27] JPQL SELECT Queries. JPOX: Java Persistent Objects, 2007. URL: http://www.jpox.org/docs/1_2/jpa/jpql.html

[28] List of object-relational mapping software. Wikipedia. URL: http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

[29] Analysis of persistence tools. Google Code, 2007. URL: http://code.google.com/p/analysis-of-persistence-tools/source