UNIVERSIDADE DO EXTREMO SUL CATARINENSE – UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

JULIANO MARQUES

COMPARAÇÃO DE DESEMPENHO E CONSUMO DE MEMÓRIA ENTRE FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL JAVA HIBERNATE E ECLIPSELINK

CRICIÚMA

2012

JULIANO MARQUES

COMPARAÇÃO DE DESEMPENHO E CONSUMO DE MEMÓRIA ENTRE FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL JAVA HIBERNATE E ECLIPSELINK

Trabalho de Conclusão de Curso, apresentado para obtenção do grau de Bacharel no curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientador: Prof. MSc. Paulo João Martins

CRICIÚMA

2012

JULIANO MARQUES

COMPARAÇÃO DE DESEMPENHO E CONSUMO DE MEMÓRIA ENTRE FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL JAVA HIBERNATE E ECLIPSELINK

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Banco de Dados

Criciúma, 29 de novembro de 2012. (data da defesa)

BANCA EXAMINADORA

Prof. Paulo João Martins – Mestre – (Universidade do Extremo Sul Catarinense – UNESC) – Orientador

Prof. Gilberto Vieira da Silva – Especialista – (Universidade do Extremo Sul Catarinense – UNESC)

Prof. Fabricio Giordani – Especialista – (Universidade do Extremo Sul Catarinense – UNESC)

AGRADECIMENTOS

Agradeço aos meus pais, pois sem eles, não seria nada. Obrigado por todo amor, apoio e carinho.

Agradeço ao Professor Paulo, pela orientação e apoio para a elaboração deste trabalho.

Por fim, agradeço a todos, que direta ou indiretamente, contribuíram, mesmo que de forma mínima, para que este trabalho se concretizasse.

“Alea jacta est” – “A sorte está lançada” - César

RESUMO

Devido à utilização de bancos de dados relacionais com linguagens de programação orientadas a objeto, ocorre o problema da Impedância de Dados que é uma incompatibilidade de utilização pelo fato de os bancos de dados terem base em um paradigma matemático, enquanto as linguagens de programação são baseadas em princípios de engenharia de software. Para contornar esse problema, foram criadas várias soluções, sendo que a mais viável até o momento é a utilização de frameworks de Mapeamento Objeto-Relacional, que mapeiam tabelas do banco de dados para objetos. Este trabalho tem por objetivo principal comparar duas ferramentas que realizam esse mapeamento, o Hibernate e o EclipseLink, em critérios de desempenho e consumo de memória, para avaliação do custo-benefício dos mesmos. Para isso foram realizados estudos sobre o paradigma OO, características de bancos de dados relacionais, e os frameworks citados. Após a pesquisa foram desenvolvidos dois protótipos para a realização dos testes, um utilizando o Hibernate e o outro utilizando o EclipseLink. As comparações foram realizadas por meio da execução de operações de inserções, edições, remoções e consultas de registros no banco de dados, sendo avaliadas nos critérios especificados com a ferramenta NetBeans Profiler, disponível no ambiente de desenvolvimento NetBeans. Os resultados demonstram que o framework Hibernate obteve melhor desempenho e menor consumo de memória na maioria dos testes realizados, dessa forma, possuindo um melhor custo-benefício, onde o custo seria o consumo de memória e o benefício o desempenho na realização da operações.

Palavras-Chave: Banco de Dados. Programação Orientada a Objeto. Mapeamento Objeto-Relacional. Hibernate. EclipseLink.

ABSTRACT

Due to the use of relational databases with object-oriented programming languages, the Impedance Mismatch problem occurs, which is an incompatibility of use because databases have base on a mathematical paradigm, while programming languages are based on principles of software engineering. To fix this problem, several solutions have been created, and the most viable yet is the use of Object-Relational Mapping frameworks, which map database tables to objects. This study aims to compare two tools that perform this mapping, Hibernate and EclipseLink, on performance and memory consumption criteria, to evaluating cost-benefit. For this, studies have been performed on the OO paradigm, characteristics of relational databases and the mentioned frameworks. After the research were developed two prototypes for the tests, one using Hibernate and the other using EclipseLink. The comparisons were performed by executing insert, edit, removal and query operations in the database and evaluated on the criteria specified with the NetBeans Profiler tool, available in NetBeans integrated development environment. The results demonstrate the Hibernate framework got better performance and lower memory consumption in most of tests, thus having a better cost-benefit, where the cost was the memory consumption and benefit was the performance in the realization of operations.

Keywords: Database. Object-Oriented Programming. Object-Relational Mapping. Hibernate. EclipseLink.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de classe com um atributo, dois métodos e uso de modificadores. 18
Figura 2 – Exemplo de classe que herda de outra 19
Figura 3 – Exemplo de classe abstrata 20
Figura 4 – Exemplo de interface com dois métodos 21
Figura 5 – Exemplo de classe com annotations.. 21
Figura 6 – Exemplo de entidade com mapeamentos de classe, atributo e relacionamentos 31
Figura 7 – Exemplo de arquivo *persistence.xml*... 33
Figura 8 – Exemplo de criação de consulta JPQL.. 34
Figura 9 – Exemplo de atualização em massa com JPQL... 35
Figura 10 – Exemplo de remoção com JPQL.. 35
Figura 11 – Página inicial do protótipo com o menu principal 41
Figura 12 – Guia Dados da Nota Fiscal Eletrônica 42
Figura 13 – Guia Consulta da Nota Fiscal Eletrônica 43
Figura 14 – Casos de Uso da Nota Fiscal Eletrônica 44
Figura 15 – Diagrama Entidade-Relacionamento do Banco de Dados 45
Figura 16 – Diagrama das classes do protótipo.. 46
Figura 17 – Estrutura dos protótipos com Hibernate e EclipseLink 47
Figura 18 – Exemplo de mapeamento para a classe Cidade 48
Figura 19 – Classe AbstractDao com os métodos CRUD principais do EntityManager 49
Figura 20 – Classe CidadeDao filha de AbstractDao 50
Figura 21 – Tela principal do NetBeans Profiler - Teste de Desempenho 51
Figura 22 – Tela principal do NetBeans Profiler – Teste de Memória.................................. 52
Figura 23 – Página de automação dos testes 53
Figura 24 – Método que executa o teste de inserção.. 55
Figura 25 – Resultados dos testes de inserção – Desempenho.. 55
Figura 26 – Resultados dos testes de inserção – Memória.. 56
Figura 27 – Método que executa o teste de remoção 57
Figura 28 – Resultados dos testes de edição – Desempenho... 57
Figura 29 – Resultados dos testes de edição – Memória... 58
Figura 30 – Método que executa o teste de remoção 59
Figura 31 – Resultados dos testes de remoção – Desempenho 59
Figura 32 – Resultados dos testes de remoção – Memória 60
Figura 33 – Consulta realizada para o teste.. 61
Figura 34 – Resultado do teste de consulta – Desempenho 61
Figura 35 – Resultado do teste de consulta – Memória... 62

LISTA DE TABELAS

Tabela 1 – Recursos de Orientação a Objetos 40
Tabela 2 – Recursos de Banco de Dados 40
Tabela 3 – Requisitos funcionais da aplicação 43
Tabela 4 – Casos de uso da aplicação 44
Tabela 5 – Pacotes dos protótipos 47
Tabela 6 – Função dos componentes da página de teste 53

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade Consistência Integridade e Durabilidade
CRUD	Create Read Update Delete
DAO	Data Acess Objects
EJB	Enterprise JavaBeans
JDBC	Java Database Connectivity
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSF	Java Server Faces
JSR	Java Specification Request
JVM	Java Virtual Machine
NF-E	Nota Fiscal Eletrônica
ORM	Object Relational Mapping
POJO	Plain Old Java Object
SGBD	Sistema Gerenciador de Banco de Dados
SGBDR	Sistema Gerenciador de Banco de Dados Relacionais
SQL	Structured Query Language
UDT	User Defined Types
UML	Unified Modeling Language
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO 11
1.1 OBJETIVO GERAL 13
1.2 OBJETIVOS ESPECÍFICOS 13
1.3 JUSTIFICATIVA 14
1.4 ESTRUTURA DO TRABALHO 15
2 A LINGUAGEM DE PROGRAMAÇÃO JAVA 16
2.1 PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA 16
2.1.1 Classe 16
2.1.2 Atributos 17
2.1.3 Métodos 17
2.1.4 Modificadores 17
2.1.5 Encapsulamento 18
2.1.6 Herança 18
2.1.7 Polimorfismo 19
2.1.8 Classes abstratas 20
2.1.9 Interfaces 20
2.2 ANNOTATIONS 21
3 BANCOS DE DADOS 22
3.1 BANCOS DE DADOS RELACIONAIS 22
3.1.1 Tabela 22
3.1.2 Chaves 23
3.1.3 Integridade de tabela e integridade referencial 23
3.1.4 Relacionamentos em banco de dados 24
3.1.4.1 Um-para-muitos 24
3.1.4.2 Um-para-um 24
3.1.4.3 Muitos-para-muitos 24
3.1.5 Índice 24
3.1.6 Structured Query Language (SQL) 25
4 JAVA PERSISTENCE API – JPA 26
4.1 ENTIDADES 26
4.1.1 Contexto de persistência 27
4.1.2 Ciclo de vida 27

4.1.3 Mapeamento 27
4.1.3.1 Mapeamento – Classe 28
4.1.3.2 Mapeamento – Atributos 28
4.1.3.3 Chaves primárias 28
4.1.3.4 Relacionamentos entre entidades 29
4.1.3.4.1 Propriedades dos Relacionamentos – Cascade 30
4.1.3.4.2 Outras Propriedades 30
4.1.3.4.3 Annotation @JoinColumn 31
4.2 ENTITY MANAGER 31
4.2.1 Métodos principais 31
4.2.2 EntityTransaction 32
4.2.3 O arquivo persistence.xml 33
4.3 JAVA PERSISTENCE QUERY LANGUAGE – JPQL 33
4.3.1 Criação de uma consulta 34
4.3.2 Cláusula UPDATE 34
4.3.3 Cláusula DELETE 35
4.4 FERRAMENTAS DE MAPEAMENTO OBJETO-RELACIONAL 35
4.4.1 Hibernate 35
4.4.2 EclipseLink 36
5 TRABALHOS CORRELATOS 37
5.1 PERFORMANCE EVALUATION OF JAVA OBJECT-RELATIONAL MAPPING TOOLS 37
5.2 COMPARAÇÃO ENTRE O USO DO JDBC E HIBERNATE PARA PERSISTÊNCIA DE DADOS 37
5.3 ESTUDO COMPARATIVO ENTRE OS FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL HIBERNATE E TOPLINK 38
6 COMPARAÇÃO DE DESEMPENHO E CONSUMO DE MEMÓRIA ENTRE FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL JAVA HIBERNATE E ECLIPSELINK 39
6.1 METODOLOGIA 39
6.2 PROTÓTIPO DE APLICAÇÃO 39
6.2.1 Concepção e Modelagem do Protótipo 39
6.2.1.1 Fundamentos do Sistema 41
6.2.1.2 Análise e levantamento de requisitos 43

6.2.1.3 Casos de uso 44
6.2.1.4 Diagramas de casos de uso 44
6.2.1.5 Modelagem do banco de dados 45
6.2.1.6 Diagrama de classes do cenário 46
6.2.2 Desenvolvimento do protótipo 46
6.2.2.1 Mapeamento 48
6.2.2.2 Gerenciamento das entidades 49
6.3 FERRAMENTA PARA REALIZAÇÃO DAS COMPARAÇÕES 50
6.3.1 Desempenho 50
6.3.2 Memória 51
6.4 REALIZAÇÃO DAS COMPARAÇÕES 52
6.4.1 Teste de Inserções 55
6.4.2 Teste de Edições 57
6.4.3 Teste de Remoções 59
6.4.4 Teste de Consultas 61
7 CONCLUSÃO 63
REFERÊNCIAS 65

1 INTRODUÇÃO

A utilização de bancos de dados tornou-se imprescindível para praticamente todas as aplicações atuais, principalmente as corporativas, visto a necessidade imperiosa de se coletar, transformar e demonstrar informação, algo extremamente valorizado. Por isso, a escolha de um Sistema Gerenciador de Banco de Dados – SGBD é muito bem pensada levando-se em consideração vantagens, desvantagens e diferenciais dos sistemas existentes. Conforme Schincariol e Keith (2009, tradução nossa), o armazenamento e obtenção de dados se tornaram um negócio multibilionário, evidenciado tanto pelo crescimento do mercado de banco de dados, quanto pelo surgimento de serviços de armazenamento seguro de dados e facilidades para obtenção dos mesmos.

Dentre os tipos de SGBDs, um dos mais populares são os Sistemas Gerenciadores de Bancos de Dados Relacionais - SGBDRs, surgidos por volta da década de 1970, com o conceito de armazenar os dados em tabelas que se relacionam entre si, tudo isso com base em conceitos lógicos e matemáticos. Graças à implementação das propriedades Atomicidade, Consistência, Integridade e Durabilidade - ACID, os bancos de dados relacionais tornaram-se sinônimo de confiabilidade e segurança dos dados armazenados. Devido a essas características são utilizados pela maioria das empresas, armazenando um grande volume de informações.

Uma das linguagens de programação mais utilizadas para desenvolvimento de aplicações que utilizam bancos de dados relacionais é a Java, de paradigma orientado a objetos, de alto nível, influenciada por C, C++ e Smalltalk (POTHU, 2008, tradução nossa). Conforme Tiobe (2012, tradução nossa), é a segunda linguagem de programação mais utilizada no planeta, principalmente por corporações, tendo em vista a sua grande gama de recursos, como portabilidade entre sistemas operacionais, orientação a objetos, multiplataforma, entre outros, estando presente desde *websites* e a *gateways*.

Por ser a linguagem Java orientada a objetos, utilizada de forma conjunta com bancos de dados relacionais, ocorre um problema chamado Impedância Objeto-Relacional. Tecnicamente, isso ocorre pela diferença de paradigmas entre a tecnologia orientada a objetos e a tecnologia relacional, sendo que a primeira tem base em princípios comprovados de engenharia de software, enquanto a última é baseada em princípios matemáticos comprovados. Pelo fato de os paradigmas serem diferentes, as tecnologias não funcionam conjuntamente de forma natural (AMBLER, 2012, tradução nossa). Além disso, segundo

Bauer e King (2005, tradução nossa), há mais problemas sutis na utilização conjunta dessa abordagem, que são os seguintes:

a) granularidade: um banco de dados relacional suporta precariamente a criação de tipos definidos do usuário (*User Defined Types* - UDT), porém é totalmente suportada na linguagem Java;
b) subtipos: inexiste suporte à herança e polimorfismo em um banco de dados relacional, porém é existente na linguagem Java;
c) identidade: refere-se à diferença entre conceitos de igualdade de registros e igualdade de objetos;
d) associações: em bancos de dados relacionais, utilizam-se chaves estrangeiras que copiam o valor da chave em várias tabelas, enquanto na linguagem Java um objeto pode ter coleções de referências a outros objetos;
e) navegação pelo grafo de objetos: refere-se à utilização de junções para ligação de tabelas com os dados desejados nos bancos de dados relacionais, enquanto um objeto pode acessar os atributos e comportamentos dos objetos associados diretamente na linguagem Java.

Os custos desse problema são inflexibilidade, perda de produtividade, perda de vantagens da orientação a objetos, e até 30% a mais de código escrito para contorná-lo (BAUER; KING, 2005, tradução nossa).

Conforme Bauer e King (2005, tradução nossa), possíveis soluções para a resolução do problema são:

a) a utilização de uma arquitetura de camadas;
b) codificar manualmente a integração entre *Structured Query Language* - SQL e *Java Database Connectivity* - JDBC;
c) utilizar serialização;
d) utilização de *Enterprise JavaBeans* – EJB Entity Beans;
e) utilização de sistemas gerenciadores de bancos de dados orientados a objetos - SGBDOR;
f) utilização de Mapeamento Objeto-Relacional (*Object-Relational-Mapping* - ORM).

A escolha pela utilização de ORM se deve ao fato de ser uma solução madura, de vasta utilização, comprovada, e que proporciona benefícios tais como produtividade, manutenibilidade, desempenho e independência de distribuidor, para dizer as principais.

O Mapeamento Objeto-Relacional é o mapeamento das tabelas e de suas relações, de um banco de dados relacional para objetos, de uma linguagem de programação orientada a objetos, com a utilização de metadados e/ou convenções de codificação que o parametrizem. Conforme Pothu (2008, tradução nossa), o ORM permite a resolução de problemas de lógica de negócio, ao invés de se preocupar em escrever código de baixo nível de acesso a banco de dados. Existem inúmeros *frameworks* ORM em Java, sendo os mais conhecidos o JBoss Hibernate, Apache OpenJPA, Eclipse EclipseLink, Oracle TopLink, Apache Cayenne e muitos outros.

Devido à grande adoção de ORM como a melhor solução disponível para a resolução do problema da Impedância Objeto-Relacional, e a importância da eficácia da ferramenta aliada com utilização racional de recursos, propõe-se este estudo, com o objetivo de realizar um comparativo entre o framework Hibernate, que é o mais utilizado, e o EclipseLink, que é a implementação de referência da *Java Persistence API* - JPA 2.0, considerando os critérios de desempenho e consumo de memória RAM.

1.1 OBJETIVO GERAL

Avaliar duas ferramentas de Mapeamento Objeto-Relacional da linguagem Java, Hibernate e EclipseLink, considerando os critérios de desempenho e utilização de memória RAM em operações com o banco de dados.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos desse trabalho consistem em:

a) compreender os conceitos de mapeamento-objeto-relacional;
b) compreender o funcionamento dos frameworks Hibernate e EclipseLink;
c) realizar uma comparação entre esses dois frameworks em desempenho e consumo de memória, para operações de inserção, edição, remoção e consulta de registros no banco de dados;
d) desenvolvimento de um protótipo de aplicação Web, simulando uma aplicação empresarial real, para realização das comparações e análise dos resultados obtidos.

1.3 JUSTIFICATIVA

Com o surgimento e a popularização de computadores, e com sua adoção expressiva em corporações, surgiu a necessidade de se manter os dados obtidos pelas aplicações após o desligamento da máquina. Após a utilização de sistemas de arquivos computadorizados, procurou-se uma nova solução de armazenamento mais simples, segura, rápida e confiável, e aí surgiu o conceito de banco de dados, que é uma estrutura integrada e compartilhada que armazena uma coleção de dados do interesse do usuário e metadados, ou seja, dados sobre os dados armazenados (CORONEL; MORRIS; ROB, 2011, tradução nossa).

Os bancos de dados tiveram uma adoção massiva com o modelo relacional, criado na década de 1970 por Codd, e implementado comercialmente a partir da década de 1980 (ELMASRI; NAVATHE, 2011). Isso se deve a recursos fornecidos como maior segurança, melhor integração, aumento de produtividade e acesso facilitado à informação.

Para lidar com esses bancos de dados, as empresas adotaram linguagens de programação que utilizavam o paradigma da orientação a objeto. O mesmo visa à modelagem de domínio de problemas em objetos, componentes compostos de estados e comportamentos. O paradigma diferencia-se por recursos como reutilização, redução e menor escrita de código, flexibilidade e manutenibilidade (BAUER, KING, 2005, tradução nossa).

Uma das linguagens que utilizam esse paradigma é a Java, criada por James Gosling, da Sun Microsystems. Conforme Oracle (2012, tradução nossa), possui uma sintaxe amigável, segura, orientada a objeto e portável, foi utilizada em larga escala para aplicativos empresariais, tanto nas plataformas desktop, web e móvel.

Porém, a utilização de linguagens orientadas a objeto, como Java, em conjunto com bancos de dados relacionais, levou ao surgimento do problema conhecido como Impedância Objeto-Relacional, caracterizado pela não integração dos paradigmas orientados a objeto e relacional, por possuírem princípios diferentes (AMBLER, 2012, tradução nossa). Com isso, buscaram-se soluções para o mesmo, e a mais viável foi o mapeamento objeto-relacional.

Com uma premissa simples, de mapear tabelas do banco de dados para classes, e pela eficácia da mesma, tornou-se muito utilizada e considerada, culminando na criação de uma especificação na linguagem Java para realização de persistência, a Java Persistence API - JPA, que segue este modelo.

Com isso, surgiram inúmeros frameworks seguindo esse preceito, acarretando dúvidas nas corporações, levando a questionamentos, tais como:

a) Qual ferramenta possui o melhor desempenho em operações com o banco de dados?
b) Qual ferramenta consome menor quantidade de memória RAM?

Para esse fim, será desenvolvido um estudo, que pretende realizar um comparativo entre os frameworks com maior visibilidade no mercado, o Hibernate e o EclipseLink, com base no desempenho e utilização de memória RAM em operações com o banco de dados. Considerando a importância dos quesitos da celeridade no processamento de informações e utilização otimizada dos recursos da máquina, valida-se este estudo, como auxílio na tomada de decisão de arquitetos e engenheiros de software sobre a ferramenta de mapeamento objeto-relacional entre as elencadas que possui um melhor custo-benefício, sendo o custo o consumo de memória e benefício como o desempenho na execução de operações.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho contém sete capítulos, sendo o primeiro constituído pela introdução, objetivos e justificativa.

O segundo capítulo apresenta os principais conceitos sobre Programação Orientada a Objetos na linguagem Java.

O terceiro capítulo demonstra os elementos principais dos Banco de Dados Relacionais.

O quarto capítulo discorre sobre a *Java Persistence API*, conceitos de entidade, mapeamento, o gerenciador de entidades, *Java Persistence Query Language* – JPQL, linguagem de consulta das entidades, e sobre os frameworks Hibernate e EclipseLink.

No quinto capítulo são apresentados alguns trabalhos correlatos, em nível nacional e mundial, com teor semelhante a este trabalho.

No sexto capítulo é discorrido sobre as comparações que são o objetivo deste trabalho, sobre os protótipos desenvolvidos, conceitos e desenvolvimento, sobre a ferramenta utilizada para a realização das comparações e sobre os resultados obtidos com as mesmas.

No sétimo capítulo é apresentada a conclusão do trabalho.

2 A LINGUAGEM DE PROGRAMAÇÃO JAVA

A linguagem Java foi criada em 1991, baseada em C++, criada com o propósito de ser uma linguagem pequena, orientada a objeto, simples e multiplataforma, para rodar em dispositivos embarcados multimídia. Seus autores foram James Gosling, em parceria com Patrick Naughton, engenheiros da Sun, em um projeto de codinome "Green".

Em 1996, foi lançada a primeira versão, a 1.02. Apesar de ter surpreendido devido ao fato de ser amigável, fácil, divertida de usar e voltada para Web, não foi considerada uma linguagem de nível corporativo por não dispor de muitos recursos, alguns básicos, e ser muito lenta. Após os lançamentos das versões seguintes, a 1.1 e 1.2, que corrigiram erros e adicionaram muitos recursos, começou a se tornar muito popular, sendo adotada em larga escala para desenvolvimento de aplicações corporativas.

Atualmente, segundo TIOBE (2012), é a segunda linguagem de programação mais utilizada no mundo, e encontra-se na versão 1.7. Fornece uma grande gama de recursos tais como orientação a objetos, simplicidade, robustez, portabilidade, segurança, vasta documentação, entre outros. Possui versões para desktop (*Java Standard Edition*), Web (*Java Enterprise Edition*) e Móvel (*Java Micro Edition*).

2.1 PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA

O paradigma da programação orientada a objetos surgiu na década de 1960, substituindo o paradigma procedural, com a criação da linguagem Simula-67. Esse paradigma cria uma abstração para resolução de problemas diversos com o mapeamento do domínio do problema para objetos, que são elementos que possuem estados e comportamentos (ECKEL, 2006, tradução nossa). Esses objetos comunicam-se uns com os outros através de mensagens. Um objeto é definido por uma classe.

2.1.1 Classe

Segundo Ricarte (2001, p. 3), "a definição da classe e seus relacionamentos é o principal resultado da etapa de projeto de software, geralmente sendo modelado em alguma linguagem, como *Unified Modeling Language* – UML". Uma classe define a forma como um objeto será criado, descrevendo quais os estados – comumente chamados de atributos, campos ou propriedades, e comportamentos – comumente chamados de métodos ou funções-membro,

que o objeto possuirá. Uma classe é definida em Java através de um modificador (opcional), a palavra-chave *class*, o nome da classe e o seu corpo entre chaves.

2.1.2 Atributos

Os atributos de uma classe representam o seu estado. Consistem em declarações de variáveis, antecedidas ou não por um modificador. O atributo pode ser uma referência a outro objeto.

2.1.3 Métodos

Representam o comportamento de uma classe, as funcionalidades da mesma. São eles que efetuam a troca de mensagens entre objetos. Um método possui duas partes, a assinatura, que consiste em um modificador (opcional), o tipo de retorno[1], o nome do método, uma lista de argumentos separados por vírgula e circundados por parênteses[2], e a declaração opcional de lançamento de exceções, sendo composta da palavra-chave *throws*, seguida dos nomes das exceções a serem lançadas separados por vírgula. A segunda parte, chamada de corpo, consiste no código do método envolvido por chaves (GOSLING et al, 2005, tradução nossa).

2.1.4 Modificadores

Modificam o comportamento do elemento em que estão presentes. Podem estar presentes na declaração de classes, atributos e métodos. Os mais utilizados são os seguintes:

a) ***public***: O elemento marcado com este modificador pode ser acessado globalmente, ou seja, de qualquer local classe ou método;
b) ***protected***: Este modificador define que só a própria classe e suas subclasses podem acessar este elemento;
c) ***private***: Quando marcado com este modificador, o elemento só é acessível dentro da classe onde está declarado. Classes só podem utilizá-lo se forem internas[3];

---

[1] Se a função não tiver retorno (procedimento), se utiliza a palavra-chave *void*.
[2] Essa lista pode ser vazia.
[3] Consiste na declaração de uma classe dentro de outra (RICARTE, 2001).

d) ***abstract***: Define que uma classe é abstrata, ou seja, que não pode ser instanciada e que pode possuir métodos abstratos.

e) ***final*****:** Quando utilizado em classes, indica que a mesma não poderá ser herdada por outras. Em atributos, indica que os mesmos possuem valor fixo (constantes). Em métodos, indica que os mesmos não poderão ser reescritos pela subclasse.

Figura 1 – Exemplo de classe com um atributo, dois métodos e uso de modificadores.

```
public class Estado
{
    private String sigla;

    public String getSigla()
    {
        return sigla;
    }

    public String setSigla(String sigla)
    {
        this.sigla = sigla;
    }
}
```

Fonte: Do autor

2.1.5 Encapsulamento

O encapsulamento consiste em ocultar do usuário o modo de funcionamento do programa. É um dos princípios da orientação a objeto, assim facilitando que os clientes concentrem-se na interface, e não na implementação, evitando que tenham que se preocupar com detalhes alheios ao domínio do seu problema (POTHU, 2008, tradução nossa). Também evita a alteração indevida no funcionamento, devido ao acesso irrestrito.

Em Java, para aplicar o encapsulamento, basta definir os atributos de uma classe como *private*, e forçar o acesso através de métodos de acesso e modificação. Outro benefício desse princípio seria de validação de conteúdo, evitando que valores inválidos sejam informados.

2.1.6 Herança

Um dos principais recursos da orientação a objetos consiste na criação de classes mais específicas (chamadas de subclasses) a partir de uma classe existente (chamada de

superclasse). A subclasse recebe os estados e comportamentos da superclasse, podendo reescrevê-los conforme a necessidade. A herança provê reutilização de código, e conforme Deitel (2011, p. 360, tradução nossa), "permite economizar tempo por basear as novas classes em outras já testadas e debugadas, de alta qualidade".

Para herdar de uma classe em Java, adiciona-se a palavra-chave *extends* após o nome da classe, e após especifica-se o nome da superclasse. Serão herdados todos os atributos com modificador *public* e *protected*, exceto construtores[4]. Em Java, não é suportada a herança múltipla, por isso uma classe pode estender somente outra classe.

Figura 2 – Exemplo de classe que herda de outra

```
@Stateless
public class EmpresaDao extends AbstractDao<Empresa>
{
    @PersistenceContext(unitName = "tcc")
    private EntityManager em;

    public EmpresaDao()
    {
        super(Empresa.class);
    }

    @Override
    protected EntityManager getEntityManager()
    {
        return em;
    }
}
```

Fonte: Do autor

2.1.7 Polimorfismo

Conforme Eckel (2006, tradução nossa, p.193), "o polimorfismo é a terceira característica mais importante de uma linguagem de programação orientada a objetos, após a abstração de dados e a herança". Consiste em permitir que instâncias de uma classe sejam tratadas como instâncias de sua superclasse. Por exemplo, em um método que recebe como argumento uma instância de uma determinada classe, são argumentos aceitáveis instâncias da classe especificada e de suas subclasses. Isso permite programas mais simples, extensíveis, legíveis e organizados. As formas mais comuns de prover o polimorfismo são através da

[4] Elemento que serve para a criação de instâncias de uma classe. Semelhante a declaração do método, exceto que não possui o tipo de retorno (RICARTE, 2001).

utilização de classes abstratas e interfaces (ECKEL, 2006, tradução nossa; DEITEL, 2011, tradução nossa).

2.1.8 Classes abstratas

A classe abstrata não permite que sejam criadas instâncias dela própria. Para isso, adiciona-se o modificador *abstract* na declaração da classe. Uma classe abstrata pode possuir métodos abstratos, que possuem somente a assinatura. Para tornar um método abstrato, adiciona-se o modificador *abstract* e finaliza-se a assinatura com um ponto-e-vírgula.

Essa abordagem comumente é utilizada para a criação de classes-modelo para outras, através da herança, mas por serem muito gerais, não faz sentido a criação de novas instâncias (RICARTE, 2001). O método abstrato pode ser utilizado quando não há uma implementação geral do mesmo, assinalando que cada subclasse deve sobrescrevê-lo. As subclasses devem sobrescrever todos os métodos abstratos da sua subclasse.

Figura 3 – Exemplo de classe abstrata

```
public abstract class AbstractDao<T>
{
    private Class<T> entityClass;

    protected abstract EntityManager getEntityManager();

    public AbstractDao(Class<T> entityClass)
    {
        this.entityClass = entityClass;
    }

    public void create(T entidade)
    {
        getEntityManager().persist(entidade);
    }

    public void edit(T entidade)
    {
        getEntityManager().merge(entidade);
    }
```

Fonte: Do autor

2.1.9 Interfaces

A interface é uma espécie de classe totalmente abstrata. É composta de declarações de métodos abstratos e constantes. A declaração de uma interface é semelhante à

de uma classe, mas utilizando a palavra-chave *interface* ao invés de *class*. São implementadas por classes, através do uso da palavra-chave *implements* seguido do nome da mesma, após o nome da classe.

Ao implementar uma interface, uma classe se compromete a sobrescrever todos os métodos especificados na mesma (CAMPOS, 2010), senão deve ser declarada como abstrata. O uso de interfaces permite estabelecer comportamentos comuns entre classes não relacionadas, ou seja, classes que não possuam uma superclasse comum. Uma classe em Java pode implementar múltiplas interfaces.

Figura 4 – Exemplo de interface com dois métodos

```
public interface DAO<T>
{
    void create(T obj);
    T find(Object id);
}
```

Fonte: Do autor

2.2 ANNOTATIONS

São interfaces especializadas que não interferem com a semântica dos programas escritos que as utilizam. Servem para anotar declarações, que providenciam informações sobre o elemento anotado. (GOSLING et al, 2005, tradução nossa).

Uma *annotation* consiste no sinal gráfico @ (arroba), seguido de um nome. Ela pode conter atributos, que são elementos chave-valor, e que podem ter um valor padrão. Podem-se adicionar *annotations* em qualquer tipo de declaração, como pacotes, classes, interfaces, atributos, métodos, parâmetros, variáveis locais e construtores.

Figura 5 – Exemplo de classe com annotations

```
@Entity
@Table(name = "unidade")
public class Unidade implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Id
    private Integer unidade;

    private String sigla;

    private String descricao;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "unidade")
    private Collection<Produto> produtoCollection;
```

Fonte: do autor

3 BANCOS DE DADOS

Bancos de dados são uma coleção de dados relacionados, que possuem as seguintes propriedades (ELMASRI; NAVATHE, 2010):

a) representa um aspecto do mundo real, chamado às vezes de minimundo ou universo do discurso;

b) é uma coleção coerente e lógica de dados com algum significado inerente;

c) é definido, construído, povoado com dados para um propósito específico.

O modelo de dados é uma abstração de um ambiente complexo do mundo real. Composto de entidades, atributos, relacionamentos e restrições, definidos com base em regras de negócio. Existem vários modelos de dados, e o mais utilizado é o relacional, que enuncia ao usuário final os dados armazenados em tabelas inter-relacionadas. Esse modelo tornou-se um padrão de implementação para banco de dados, sendo utilizado em larga escala, pela simplicidade e confiabilidade do armazenamento dos dados (ELMASRI; NAVATHE, 2010; DATE, 2004).

3.1 BANCOS DE DADOS RELACIONAIS

Em 1970, Ted Codd, funcionário da IBM, criou o modelo de dados relacional, que ficou conhecido pela simplicidade e origem matemática (CORDEIRO, 2011). Os primeiros bancos de dados implementando o modelo surgiram na década de 1980 e, atualmente, é o modelo mais utilizado, com muitas versões, comerciais como IBM DB2, Microsoft SQL Server, Sybase, e versões gratuitas, como MySQL e PostgreSQL.

3.1.1 Tabela

A tabela é uma estrutura bidimensional que armazena dados em linhas e colunas. Também conhecida por relação, armazena um grupo de ocorrência de entidades relacionadas. Possui as seguintes características (CORONEL; MORRIS; ROB, 2009, tradução nossa):

a) Ser composta de linhas e colunas;
b) Cada linha, também chamada de tupla, representa uma ocorrência da entidade no conjunto de entidades;
c) Cada coluna representa um atributo, e cada coluna possui um nome diferente;
d) Cada célula da tabela representa um valor simples;

e) Todos os valores representados numa coluna são do mesmo valor;
f) Cada coluna possui uma faixa de valores conhecida como domínio;
g) A ordem das linhas e colunas é indiferente;
h) Cada tabela deve ter um atributo ou combinação deles que a identifiquem unicamente.

3.1.2 Chaves

Chaves são atributos simples ou compostos que determinam outros atributos. O conceito de determinação seria que, sabendo-se o valor dessa chave, pode-se saber o valor de outros atributos presentes na tabela (ELMASRI; NAVATHE, 2010). As chaves podem identificar linhas da tabela de forma única, ou estabelecer relacionamentos entre as mesmas. Existem os seguintes tipos de chaves:

a) superchave: é um atributo simples ou conjunto deles que identifica de forma única cada linha da tabela;
b) chave candidata: é uma superchave mínima, ou seja, não possui outros atributos na sua composição que também sejam superchaves;
   chave primária: uma chave candidata que identifica todos os outros atributos numa dada linha. Não pode possuir valores nulos;
c) chave secundária: um atributo ou conjunto deles que é utilizado somente para obtenção de dados;
d) chave estrangeira: um atributo ou conjunto deles numa tabela que devem corresponder a chave primária de outra tabela ou serem nulos;
e) chave primária: Uma chave que possui mais de um atributo é chamada de chave composta.

3.1.3 Integridade de tabela e integridade referencial

São princípios que devem ser seguidos para a criação de um bom projeto de banco de dados (CORONEL; MORRIS; ROB, 2009, tradução nossa). A integridade de tabela enuncia que todas as chaves primárias são únicas e não podem possuir valor nulo, enquanto a integridade referencial enuncia que as chaves estrangeiras devem corresponder a uma chave primária em outra tabela, ou serem nulas.

3.1.4 Relacionamentos em banco de dados

3.1.4.1 Um-para-muitos

O relacionamento um-para-muitos acontece quando, para uma determinada linha em uma tabela, pode haver múltiplas linhas correspondentes em outra. Esse relacionamento é o mais comum em bancos de dados (DATE, 2004). É composto geralmente de uma tabela com chave primária, e outra tabela com uma chave estrangeira correspondente a primeira. O relacionamento obtido através da inversão da posição das tabelas é chamado de muitos-para-um.

3.1.4.2 Um-para-um

No relacionamento um-para-um, para cada linha presente numa tabela, deve haver somente uma linha correspondente em outra tabela. Cada tabela possui uma chave estrangeira que referencia a chave primária da outra. Esse relacionamento é raramente utilizado em bancos de dados.

3.1.4.3 Muitos-para-muitos

Nesse relacionamento, para múltiplas linhas em uma tabela, correspondem múltiplas linhas em outra tabela. Essa forma não é suportada diretamente em banco de dados relacionais (ELMASRI; NAVATHE, 2010; CORONEL; MORRIS; ROB, 2009, tradução nossa), sendo geralmente implementada através da criação de uma tabela associativa, que é uma tabela que possui duas chaves estrangeiras, uma para cada tabela do relacionamento, em um relacionamento muitos-para-um com as mesmas. Essa tabela pode possuir outras colunas, conforme a necessidade.

3.1.5 Índice

O índice é uma estrutura que permite a referência imediata a linhas específicas de uma tabela, sem a necessidade de percorrê-la por inteiro. Segundo Coronel, Morris e Rob (2009, tradução nossa, p.86), “formalmente é definido como um arranjo ordenado de chaves e ponteiros”. A chave é composta de uma ou mais colunas da tabela, enquanto os ponteiros são ligações para linhas que correspondam a essa chave. Uma tabela pode ter vários índices, mas

cada um pode pertencer a somente uma única tabela. O índice pode ser único, quando só há uma linha correspondente a chave.

Por padrão, é definido um índice na chave primária e em cada chave estrangeira de uma tabela. Índices são de grande importância para o banco de dados, por aumentar o desempenho de consultas que utilizam colunas com os mesmos, sendo uma tarefa complexa, por vezes, a determinação de um bom índice para a tabela.

3.1.6 Structured Query Language (SQL)

A linguagem SQL é uma linguagem para definição e manipulação de dados em bancos de dados relacionais. Considerada de quarto nível, com uma sintaxe semelhante a linguagem natural inglesa, tem vocabulário simples e uma curva de aprendizado relativamente pequena (ELMASRI; NAVATHE, 2010; DATE, 2004; CORDEIRO, 2011).

Na parte de definição de dados, permite a criação de tabelas, esquemas, índices, visões, entre outros, e permissões para esses itens. Os comandos principais de definição são CREATE, ALTER e DROP, que criam, alteram e removem estruturas do banco, respectivamente.

Na parte de manipulação de dados, permite a inserção, consulta, edição e remoção de dados, através dos comandos INSERT, SELECT, UPDATE e DELETE, respectivamente.

4 JAVA PERSISTENCE API – JPA

A *Java Persistence API* – JPA é uma especificação oficial da linguagem Java para frameworks de Mapeamento Objeto-Relacional. A versão 1.0 foi liberada em 11 de maio de 2006 como parte da *Java Specification Request* – JSR 220, e a versão atual, a 2.0, foi liberada em 10 de dezembro de 2009, na JSR 317.

Em suma, para a realização do mapeamento, a JPA utiliza *Plain Old Java Objects* – POJOs, que são classes Java que obedecem às seguintes convenções de codificação (ORACLE, 2012, tradução nossa):

a) A classe deve possuir um construtor *public* ou *protected*, sem argumentos. A classe pode ter outros construtores;
b) A classe não deve ser declarada como *final*. Nenhum método ou atributo deve ser declarado como *final*;
c) A classe deve implementar a interface *Serializable*[5];
d) Atributos devem ser declarados como *private* e serem acessados diretamente pelos próprios métodos da classe. Clientes devem acessar o estado da classe somente através de métodos acessores.

Cada POJO representa uma tabela do banco de dados, enquanto seus atributos representam as colunas da tabela (POTHU, 2008, tradução nossa; MAKI, 2007, tradução nossa). As propriedades globais do *framework*, tais como parâmetros de conexão com o banco de dados, mapeamentos de classes e outras, são definidas em um arquivo *Extensible Markup Language* – XML, geralmente chamado *persistence.xml*. As principais operações, como as operações *Create, Read, Update, Delete* – CRUD, são feitas por uma classe especial chamada *EntityManager*.

4.1 ENTIDADES

Uma entidade é um POJO que utiliza metadados para descrever-se. Em outras palavras, "uma entidade é um objeto de domínio persistente leve" (ORACLE 2012, p. 583, tradução nossa). Sua função é representar uma tabela em um banco de dados relacional, onde cada instância representa uma linha da tabela. (KEITH; HALEY; SCHINCARIOL, 2006, tradução nossa). Deve possuir, obrigatoriamente, a *annotation* @Entity.

[5] Interface que indica que um objeto de uma classe pode ser serializado, ou seja, comprimido. Após, pode ser desserializado, ou seja, descomprimido (GOSLING et al, 2005, tradução nossa).

4.1.1 Contexto de persistência

"Contexto de persistência é um conjunto de instâncias de entidades gerenciadas que existem num conjunto de dados particular, definindo o escopo sobre o qual instâncias particulares de entidades são criadas, persistidas, ou removidas" (KEITH; SCHINCARIOL, 2009, p. 131, tradução nossa). A interface *EntityManager* define os métodos que são utilizados para interação com o contexto de persistência.

4.1.2 Ciclo de vida

O ciclo de vida de uma instância de entidade demonstra os possíveis estados em que ela pode estar. Uma instância está em um dos quatro estados, a seguir (SUN, 2009, tradução nossa):

a) novo: a instância foi criada, mas não foi persistida e não está associada a um contexto de persistência;
b) gerenciado: quando a instância foi persistida e está associada a um contexto de persistência;
c) desanexado: quando a instância foi persistida, mas não está associada a um contexto de persistência;
d) removido: quando a instância foi persistida, está associada a um contexto de persistência e está agendada para remoção do banco de dados.

4.1.3 Mapeamento

Para realização do mapeamento objeto-relacional, são utilizados metadados, que permitem a JPA reconhecer e manipular a entidade de forma adequada. São suportadas duas formas (MAKI, 2007, tradução nossa):

a) annotations: Presentes em nível de classe, atributo ou método. Todas as annotations utilizadas na JPA situam-se no pacote *javax.persistence*;
b) arquivo de configuração XML: Um arquivo central com um dialeto XML específico.

As duas formas são igualmente válidas e podem ser combinadas. Neste trabalho são utilizadas *annotations*, por ser uma forma mais intuitiva para a descrição das entidades.

4.1.3.1 Mapeamento – Classe

As principais *annotations* utilizadas para mapeamento de entidade, em nível de classe, são (GONCALVES, 2010, tradução nossa):

a) @Entity: *annotation* principal, indica que a classe anotada é uma entidade;
b) @Table: identifica a tabela primária mapeada pela entidade. Dos seus atributos, destaca-se *name*, que indica o nome da tabela no banco de dados.
c) @Embeddable: identifica se uma classe é um objeto embutido. Um objeto embutido não é uma entidade, mas uma classe auxiliar, que serve para agrupar logicamente atributos da entidade em uma classe separada.

4.1.3.2 Mapeamento – Atributos

Em nível de atributos, as principais *annotations* utilizadas são (YOUSAF, 2012, tradução nossa; CORDEIRO, 2011):

a) @Id: indica que o atributo é a chave primária da entidade;
b) @Column: Define o mapeamento da coluna da tabela para o atributo. Destaca-se a propriedade *name*, que serve para indicar o nome da coluna, caso o nome do atributo e o nome da coluna sejam diferentes;
c) @Embedded: Define se um atributo utilizado na entidade é um objeto embutido;
d) @EmbeddedId: Define se a chave primária da entidade é um objeto embutido. Utilizado para chaves primárias compostas.

4.1.3.3 Chaves primárias

Todas as entidades devem possuir uma chave primária, ou seja, um atributo da classe que a identifique de forma única, assim possibilitando a localização de uma instância da entidade específica (KEITH; HALEY; SCHINCARIOL, 2006, tradução nossa). Essa chave representa a chave primária da tabela mapeada, e pode ser simples ou composta.

a) simples: utilizada quando a chave primária possui somente uma coluna. O atributo correspondente deve ser marcado com a *annotation* @Id.

b) composta: utilizada quando a chave primária possui mais de uma coluna. Nesse caso, deve-se criar uma classe POJO contendo todos os atributos da chave, e marcá-la com a

annotation @Embedded. Devem ser implementados os métodos *equals*[6] e *hashcode*[7]. Na entidade, deve haver uma referência à classe criada, marcada com a *annotation* @EmbeddedId.

4.1.3.4 Relacionamentos entre entidades

É o mapeamento dos relacionamentos existentes entre as tabelas do banco de dados. Em questão de número de valores, podem ser de dois tipos, a seguir (SUN, 2009, tradução nossa):

a) univalorados: quando o atributo para entidade relacionada consiste em uma única referência de objeto;
b) multivalorados: quando o atributo para a entidade relacionada consiste em uma coleção de referências de objetos.

Em questão de direção, os relacionamentos podem ser de dois tipos, a seguir (MAKI, 2007, tradução nossa):

a) unidirecionais: quando somente uma das duas entidades possui ligação com a outra;
b) bidirecionais: quando as duas entidades possuem ligação uma com a outra.

Em questão de cardinalidade, os relacionamentos podem ser de quatro tipos, a seguir (KEITH; SCHINCARIOL, 2009, tradução nossa):

a) um-para-um: univalorado, uma instância de uma entidade se relaciona com uma única instância de outra entidade. Definido pela *annotation* @OneToOne no campo correspondente;
b) um-para-muitos: multivalorado, uma instância de uma entidade se relaciona com múltiplas instâncias de outra entidade. Definido pela *annotation* @OneToMany no campo correspondente;
c) muitos-para-um: univalorado, múltiplas instâncias de uma entidade referindo-se a uma instância de outra entidade. Definido pela *annotation* @ManyToOne no campo correspondente;
d) Muitos-para-muitos: multivalorado, múltiplas instâncias de uma entidade referindo-se a múltiplas instâncias de outra entidade. Definido pela *annotation* @ManyToMany no campo correspondente.

---

[6] Método herdado de *Object*. Indica se uma instância de uma classe é igual a outra (GOSLING et al, 2005).
[7] Método herdado de *Object*. Gera um código de *hashing* para a instância do objeto (GOSLING et al, 2005).

4.1.3.4.1 Propriedades dos Relacionamentos – Cascade

Indicam que operações devem ser propagadas para as entidades relacionadas. Os tipos válidos de cascade são (YOUSAF, 2012, tradução nossa; GONCALVES, 2010, tradução nossa):

a) *detach* (Desanexado): Se a entidade pai for desanexada, as entidades filhas também serão desanexadas;
b) *merge* (Reunido): Se a entidade pai for reunida, as entidades filhas também serão reunidas;
c) *persist* (Persistido): Se a entidade pai for persistida, as entidades filhas também serão persistidas;
d) *refresh* (Atualizado): Se a entidade pai for atualizada, as entidades filhas também serão atualizadas;
e) *remove* (Removido): Se a entidade pai for removida, as entidades filhas também serão removidas;
f) *all* (Todos): Propaga todas as operações supracitadas.

4.1.3.4.2 Outras Propriedades

Outras propriedades comuns em relacionamentos são:

a) *optional*: Se o relacionamento é opcional (aceita valor nulo);
b) *fetch*: define como são obtidas as entidades relacionadas. Se o tipo for *EAGER* (ansioso), são trazidas todas as entidades vinculadas, enquanto que com o tipo *LAZY* (preguiçoso) são trazidas somente as entidades referenciadas. O segundo tipo pode aumentar consideravelmente a velocidade de execução de consultas e a obtenção de objetos (MAKI, 2007, tradução nossa; KEITH; HALEY; SCHINCARIOL, 2006, tradução nossa).
c) *mappedBy*: requerido somente se o relacionamento for bidirecional, e multivalorado. Indica qual atributo da entidade relacionada que vincula com a entidade atual.

4.1.3.4.3 Annotation @JoinColumn

Annotation que especifica colunas de chaves estrangeiras, podendo também ser utilizada para mapeamento de relacionamento muitos-para-muitos, com tabela associativa. São dignas de nota as propriedades *name*, que indica o nome da coluna de chave estrangeira da tabela, e *referencedColumnName*, que indica a coluna a qual a chave estrangeira referencia.

Figura 6 – Exemplo de entidade com mapeamentos de classe, atributo e relacionamentos

```
@Entity
@Table(name = "produto")
public class Produto implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Id
    private String produto;

    private String descricao;

    @Column(name = "ncm_sh")
    private String ncmSh;

    @JoinColumn(name = "unidade", referencedColumnName = "unidade")
    @ManyToOne(optional = false)
    private Unidade unidade;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "produto")
    private Collection<NfeProduto> nfeProdutoCollection;
```

Fonte: Do autor

4.2 ENTITY MANAGER

Uma instância da interface *javax.persistence.EntityManager*, que gerencia um contexto de persistência.

4.2.1 Métodos principais

As operações principais realizadas pelo *EntityManager* com as instâncias de entidades são (KEITH; SCHINCARIOL, 2009, tradução nossa; SUN, 2009, tradução nossa; MAKI, 2007, tradução nossa; GONCALVES, 2010, tradução nossa):

a) *persist*: Torna persistente e gerenciada uma instância de entidade no estado novo. Se o estado da instância for gerenciado, a operação é ignorada. Se for removido, a instância torna-se gerenciada. Caso seja desanexado, é lançada uma *exception*[8]. Com exceção do último estado mencionado, entidades associadas em relacionamentos com o tipo *PERSIST* ou *ALL* também são persistidas;

b) *merge*: Esta operação pode persistir ou atualizar uma instância, conforme o estado dela. Se o estado for novo, atua como o método *persist*. Se o estado for gerenciado ou desanexado, atualiza a entidade no contexto de persistência ("reúne" as alterações). Entidades associadas em relacionamentos com o tipo *MERGE* ou *ALL* também são persistidas/atualizadas;

c) *remove*: Remove uma instância gerenciada. Se esta operação for executada em instâncias com o estado novo ou removido, é ignorada. Se o estado for desanexado, é lançada uma *exception*. Entidades associadas em relacionamentos com o tipo *REMOVE* ou *ALL* também são persistidas/atualizadas;

d) *find*: Busca uma entidade pela chave primária. Retorna nulo se a entidade não existir;

e) *createQuery*: Cria uma instância de *javax.persistence.Query,* interface utilizada para realização de consultas ao contexto de persistência. Os formatos possíveis são *Java Persistence Query Language* – JPQL, uma linguagem simples, semelhante à linguagem SQL, ou em *Criteria*, onde a consulta é montada em código Java.

4.2.2 EntityTransaction

Interface que gerencia uma transação de banco de dados, obtida pelo método *getTransaction* do *EntityManager*. Utilizada para se realizar uma operação num contexto de transação, geralmente conjuntamente com o método *persist*, *merge* e *remove*. Os métodos principais de uma *EntityTransaction* são (KEITH; HALEY; SCHINCARIOL, 2006, tradução nossa):

a) begin: inicia uma transação;

---

[8] Erro lançado em tempo de execução, que geralmente não é fatal (permite o programa continuar sua execução). (GOSLING et al 2005)

b) commit: grava a transação;

c) rollback: desfaz a transação.

4.2.3 O arquivo persistence.xml

O arquivo *persistence.xml*, por padrão contido no pacote *META-INF*, armazena as configurações do mapeamento objeto-relacional. Pode ser composto por uma ou mais unidades de persistência (KEITH; HALEY; SCHINCARIOL, 2006, tradução nossa; ORACLE, 2012, tradução nossa).

As unidades de persistência definem as entidades gerenciadas pelo *EntityManager*. Cada unidade possui um identificador único, um nome, que serve para referenciá-la. Uma das utilizações desse nome pode ser para a obtenção de uma *EntityManagerFactory*, que é um objeto que é utilizado para a criação de *EntityManagers*. As unidades podem ser de implementações diferentes da JPA, e também podem definir um conjunto de propriedades específicas (SUN, 2009, tradução nossa).

Figura 7 – Exemplo de arquivo *persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="tcc" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/tcc</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <validation-mode>NONE</validation-mode>
        <properties>
            <property name="hibernate.cache.use_second_level_cache" value="false"/>
            <property name="hibernate.cache.use_query_cache" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Fonte: Do autor

4.3 JAVA PERSISTENCE QUERY LANGUAGE – JPQL

Conforme Keith e Schincariol (2009, tradução nossa), a *Java Persistence Query Language* – JPQL é uma linguagem de consulta independente de banco de dados, que opera no modelo lógico de entidades, ao invés de operar no modelo físico de dados. Enquanto que numa query SQL são utilizadas tabelas do banco de dados e colunas das mesmas, em JPQL, respectivamente, utilizam-se entidades e atributos das mesmas.

As consultas em JPQL são portáveis, possíveis de conversão na maioria dos bancos de dados atuais, e com uma sintaxe semelhante à linguagem SQL, o que diminui sua

curva de aprendizado. Suporta uma grande gama de recursos, tais como expressões de agregação, junções internas e externas, *subqueries*, cláusulas *group by, having*, funções padrão, e outros (POTHU, 2008, tradução nossa).

4.3.1 Criação de uma consulta

Consultas JPQL são criadas através dos métodos *createQuery* e *createNamedQuery* do *EntityManager*. Uma consulta pode ser de dois tipos (SUN, 2009, tradução nossa):

a) **consultas dinâmicas**: criadas através do método *createQuery*, geralmente passando a consulta como argumento. Essas consultas são definidas na lógica de negócio da aplicação, podendo ser alteradas em tempo de execução, conforme a necessidade.

b) **consultas estáticas**: criadas através do método *createNamedQuery*. Passa-se o nome da consulta como argumento. Essas consultas, diferentemente da forma dinâmica, são definidas na anotação de nível de classe @NamedQuery, e não podem ser alteradas em tempo de execução.

Figura 8 – Exemplo de criação de consulta JPQL

```
public List<Fatura> faturasPorNfe(Nfe nfe)
{
    return getEntityManager()
            .createQuery("select f from Fatura f where f.nfe = :nfe")
            .setParameter("nfe", nfe).getResultList();
}
```

Fonte: Do autor

4.3.2 Cláusula UPDATE

A JPQL provê uma construção semelhante ao UPDATE do SQL, servindo para a realização de atualizações simples e em massa (SUN, 2009, tradução nossa; YOUSAF, 2012, tradução nossa). Consiste na expressão UPDATE, seguida de um nome de entidade, que pode utilizar um alias opcional, seguida de uma instrução SET, onde são listadas, separadas por vírgula, atributos univalorados da entidade, seguidas pelo operador de atribuição (=) e o valor que se deseja atribuir, que pode ser um parâmetro nomeado, e deve ser de um tipo compatível com o tipo do atributo. Opcionalmente, pode ter uma cláusula WHERE para filtrar os

resultados que devem ser atualizados, possuindo todas as funcionalidades presentes da cláusula quando em um SELECT (ORACLE, 2012, tradução nossa).

Figura 9 – Exemplo de atualização em massa com JPQL

```
public void updateAll()
{
    getEntityManager().createQuery("update Nfe n set n.baseCalculoIcms = 0")
            .executeUpdate();
}
```

Fonte: Do autor

4.3.3 Cláusula DELETE

Semelhante ao DELETE provido na linguagem SQL, permite a remoção simples e em massa de registros. Consiste na expressão DELETE FROM, seguida do nome da entidade, com opção de informar alias, e uma cláusula WHERE opcional, com todas funcionalidades presentas na cláusula num SELECT, para restringir o conjunto de resultados a serem removidos (MAKI, 2007, tradução nossa; KEITH; HALEY; SCHINCARIOL, 2006, tradução nossa).

Figura 10 – Exemplo de remoção com JPQL

```
public Integer removeAll()
{
    return getEntityManager().createQuery("delete from Nfe")
            .executeUpdate();
}
```

Fonte: Do autor

4.4 FERRAMENTAS DE MAPEAMENTO OBJETO-RELACIONAL

4.4.1 Hibernate

Criado em 2001 por Gavin King, como uma alternativa para o uso de beans de entidade EJB2. King pretendia oferecer melhores capacidades de persistência que o EJB2, simplificando o uso e permitindo a utilização de características ausentes nessa ferramenta

(JBOSS, 2012, tradução nossa). É uma solução madura, com grande quantidade de documentação e comunidade ativa.

Em 2003, foi lançada a versão 2, com melhorias significativas em relação a primeira versão. Com isso, o Hibernate começou a tornar-se a solução padrão de persistência em Java. A empresa JBoss contratou os desenvolvedores do framework, para dar continuidade ao projeto. Atualmente, encontra-se na versão 4.1.6, lançada em 08/08/2012.

O framework implementa as interfaces de programação e regras de ciclo de vida definidas na especificação 2.0 da JPA. O Hibernate pode ser utilizado com ou sem JPA, ou ainda, utilizando características próprias e da JPA de forma conjunta. Muitas características presentes na JPA foram inspiradas pelo Hibernate, como o modelo de objetos, gerenciamento de entidades, linguagens de busca, entre outros (BAUER; KING, 2007, tradução nossa).

4.4.2 EclipseLink

EclipseLink é um framework de persistência baseado no framework Oracle Toplink. Implementa a JPA 2.0, sendo considerado como a implementação de referência da mesma, implementando todos os recursos obrigatórios, boa parte das características opcionais e muitas extensões. Alguns exemplos de extensões são cache em nível de objeto, variadas opções de tuning, suporte a bases de dados NoSQL, mapeamentos avançados, suporte avançado a banco de dados Oracle, entre outros.

5 TRABALHOS CORRELATOS

Esta seção relaciona alguns dos trabalhos relacionados com o conteúdo semelhante a essa fundamentação teórica, utilizados no desenvolvimento dessa pesquisa.

5.1 PERFORMANCE EVALUATION OF JAVA OBJECT-RELATIONAL MAPPING TOOLS

Dissertação de Mestrado apresentada por Haseeb Yousaf para a Universidade de Geórgia, para obtenção do grau de Mestre de Ciência, nos Estados Unidos, em 2012.

Esse trabalho consiste em uma comparação de desempenho entre *frameworks* de mapeamento objeto-relacional Java de grande utilização, que são o Hibernate, EclipseLink, OpenJPA e Ebean, e uma ferramenta de persistência desenvolvida, chamada de GlycoVault.

O trabalho discorre sobre os trabalhos correlatos, mapeamento objeto-relacional e problemas iniciais enfrentados, ferramentas utilizadas e suas características, como linguagens de *query*, gerenciamento de transações e cache.

O estudo conclui que cada sistema possui seus prós e contras, e que a linguagem de *query*, estratégias de herança e mecanismos de cache podem determinar o que precisa ser feito para melhorar a desempenho. São sugeridas possíveis melhorias no *framework* desenvolvido, o GlycoVault, como implementação de uma linguagem de *query* completa e implementação de cache de segundo nível.

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

Trabalho de Conclusão de Curso de Cristina da Silva Matos para a Universidade do Extremo Sul Catarinense, para a obtenção do grau de Bacharel em Ciência da Computação, Criciúma, 2011.

Esse trabalho teve como objetivo realizar comparações entre os métodos de persistência utilizados para evitar o problema da Impedância Objeto-Relacional, o JDBC e o Hibernate, considerando fatores como facilidade de codificação, legibilidade, tamanho de código-fonte e tempo de acesso. Para esse fim, foram desenvolvidos dois protótipos de aplicação.

No trabalho, foram abordados os assuntos de Orientação a Objetos, Banco de Dados e framework Hibernate, bem como discussão sobre trabalhos correlatos.

O estudo conclui que a utilização da ferramenta de ORM Hibernate é mais vantajosa, visto que possibilita um código-fonte menor, mais legível e com tempo de realização de operações de persistência reduzido. Foram sugeridos como possíveis trabalhos futuros a utilização de uma aplicação legada, de outros métodos de persistência, e de programação orientada a aspectos.

5.3 ESTUDO COMPARATIVO ENTRE OS FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL HIBERNATE E TOPLINK

Monografia de Jader dos Santos Teles Cordeiro, apresentada para a Universidade Estadual de Maringá, para a obtenção do grau de Especialista, em 2011.

Esse trabalho visou a realizar comparações entre os frameworks de Mapeamento Objeto-Relacional Hibernate e TopLink, como uma alternativa a utilização de JDBC, que é menos produtivo, com código-fonte maior, e menos performático.

Foram abordados temas como Persistência de Dados, de Objetos, Banco de Dados, Modelo Relacional, frameworks ORM Hibernate, EclipseLink e JPA, bem como realização de *benchmarks*.

O estudo conclui que o Hibernate obteve um melhor desempenho em comparação ao TopLink no conjunto de consultas realizadas. Sugere como trabalhos futuros a utilização de JPA, e utilização de banco de dados Oracle.

6 COMPARAÇÃO DE DESEMPENHO E CONSUMO DE MEMÓRIA ENTRE FRAMEWORKS DE MAPEAMENTO OBJETO-RELACIONAL JAVA HIBERNATE E ECLIPSELINK

Hibernate e EclipseLink são frameworks que têm por objetivo realizar o Mapeamento Objeto-Relacional. Como implementam a especificação JPA, possuem utilização idêntica.

6.1 METODOLOGIA

Para fins da realização da comparação de desempenho e consumo de memória entre os frameworks ORM Hibernate e EclipseLink, estudaram-se as ferramentas e desenvolveram-se dois protótipos de aplicação, focando-se na aplicação dos conceitos e paradigmas discutidos neste trabalho, como Orientação a Objetos, Mapeamento Objeto-Relacional com JPA e Bancos de Dados.

6.2 PROTÓTIPO DE APLICAÇÃO

Para o protótipo de aplicação desenvolvido modelou-se o cenário de uma aplicação Web, para cadastro de Notas Fiscais Eletrônicas. Para cada framework, foram criados dois protótipos, sendo que ambos possuem estrutura e funcionamento idênticos, excetuando-se as bibliotecas de código utilizadas e o arquivo *persistence.xml,* que possuem configurações específicas para cada framework de persistência.

Por ter o objetivo de demonstrar os conceitos discorridos neste trabalho, o desenvolvimento foi simplificado, atendo-se ao estritamente necessário, para fins de clareza.

Utilizou-se no desenvolvimento dos protótipos a linguagem Java, versão 6, no ambiente de desenvolvimento NetBeans, versão 7.2. O SGBD utilizado foi o MySQL, versão 5.5, e as versões utilizadas do Hibernate e EclipseLink foram a 4.1.7 e a 2.4.0, respectivamente.

6.2.1 Concepção e Modelagem do Protótipo

Para a realização do objetivo geral deste trabalho, criou-se um cenário de uma aplicação voltada para a Web, que possui operações encontradas com facilidade em aplicações reais, como inserção, edição, remoção e listagem de registros do banco de dados.

O cenário escolhido foi uma aplicação para cadastro de Notas Fiscais Eletrônicas. O usuário pode inserir, editar, excluir e listar notas fiscais. Essas operações também poderão ser realizadas com os registros de tabelas vinculadas às notas, tais como Cidade, Estado, Produto, Unidade, entre outras.

O princípio que norteou a criação do protótipo foi a utilização e demonstração dos conceitos discorridos neste trabalho, como Orientação a Objetos, Mapeamento Objeto-Relacional e Bancos de Dados. Por isso, foram criados requisitos que atendessem os dois conceitos, demonstrados nas tabelas 1 e 2.

Tabela 1 – Recursos de Orientação a Objetos

Recurso
Classe simples
Classe com tipo complexo
Classe abstrata
Coleção
Herança

Fonte: Do autor

Tabela 2 – Recursos de Banco de Dados

Recurso
Consulta com *inner join*
Consulta com função de agregação (*count*)
Consulta por chave primária
Consulta com critério
Relacionamento um-para-muitos
Relacionamento muitos-para-um

Fonte: Do autor

Para a realização dos testes, não foram considerados muitos requisitos presentes em aplicações reais, como controles de acesso e validações, para que a clareza do protótipo não seja obscurecida, e por tais requisitos serem irrelevantes ao propósito deste trabalho.

6.2.1.1 Fundamentos do Sistema

O sistema funcionará da seguinte forma:

Ao iniciar, será exibida uma página com o menu principal de aplicação, contendo as opções “Cadastros”, “Movimentos”, “Testes” e “Sobre”.

Ao clicar na opção “Cadastros”, será exibido um submenu com as opções de cadastros disponíveis, que são “Cidades”, “Estados”, “Unidades”, “Produtos”, “Operações”, “Empresas”, “Transportadores” e “Remetentes/Destinatários”. Ao clicar em um dessas opções, abrirá a página de cadastro respectiva.

Ao clicar na opção “Movimentos”, será exibido um submenu com a opção “NFe”. Ao clicar nessa opção, abrirá a página de cadastro da Nota Fiscal Eletrônica.

Ao clicar na opção “Testes”, será exibido um submenu com a opção “Testes”. Ao clicar nessa opção, abrirá a página contendo os testes que podem ser executados na aplicação.

Ao clicar na opção “Ajuda”, será exibido um submenu com a opção “Sobre”. Ao clicar nessa opção, abrirá uma página contendo informações sobre as ferramentas utilizadas no protótipo e suas versões.

Figura 11 – Página inicial do protótipo com o menu principal

Fonte: Do autor

A página de cadastro consiste em guias, sendo uma ou mais contendo os campos de entrada para inserir as informações, tendo o título de “Dados”. Se houver mais de uma guia de “Dados”, serão numeradas, como “Dados 1”, “Dados 2”, e assim em diante.

Outra guia é a de “Consulta”, que possui uma tabela listando todos os registros do banco de dados para aquele cadastro. Podem ser filtrados os registros por meio de campos de entrada situados no cabeçalho da mesma. Clicando no botão “Editar”, são preenchidos os campos de entrada das guias de dados com as informações do registro selecionado, possibilitando a alteração das mesmas, e ao clicar no botão “Excluir”, o registro atual é removido, após ser confirmada sua exclusão.

O botão “Novo” limpa os campos de entrada, e o botão “Salvar” insere os dados de um registro novo, ou persiste as edições realizadas. Os botões “Volumes”, “Faturas” e “Estoques” estão presentes somente na página de cadastro da Nota Fiscal Eletrônica, e ficam visíveis somente ao editar uma nota. Clicando em um desses botões, o usuário é redirecionado para as páginas de cadastro de Volumes Transportados, Faturas e Produtos dessa nota, respectivamente.

Figura 12 – Guia Dados da Nota Fiscal Eletrônica

Fonte: Do autor

Figura 13 – Guia Consulta da Nota Fiscal Eletrônica

Fonte: Do autor

6.2.1.2 Análise e levantamento de requisitos

Após serem obtidos os dados necessários, converteram-se os mesmos em informações, e foi realizado o levantamento de requisitos da aplicação. A tabela 3 apresenta os requisitos para cadastro das Notas Fiscais Eletrônicas. Esses podem ser aplicados para todas as outras tabelas do banco.

Tabela 3 – Requisitos funcionais da aplicação

Requisito	Descrição
F01: Inserir uma Nota Fiscal Eletrônica	Cadastrar uma nova nota fiscal eletrônica no sistema
F02: Editar uma Nota Fiscal Eletrônica	Alterar os dados de uma nota fiscal eletrônica existente
F03: Excluir uma Nota Fiscal Eletrônica	Remover do banco de dados do sistema uma nota fiscal eletrônica
F04: Buscar uma Nota Fiscal Eletrônica	Obtenção de uma nota fiscal eletrônica pela chave primária e por outros campos
F05: Listar as notas fiscais eletrônicas existentes	Lista todas as notas fiscais eletrônicas existentes no banco de dados

Fonte: Do autor

6.2.1.3 Casos de uso

Na tabela 4, são apresentados os casos de uso mais comuns. Os mesmos podem ser aplicados a todas as tabelas do banco de dados.

Tabela 4 – Casos de uso da aplicação

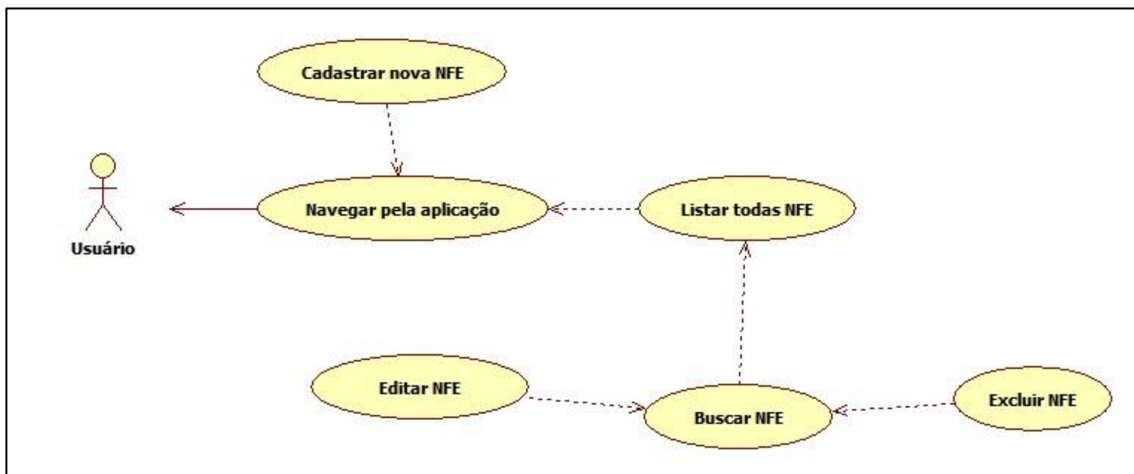
Nome	Descrição
C01	Navegar pela aplicação
C02	Cadastrar uma nova nota fiscal eletrônica
C03	Editar uma nota fiscal eletrônica
C04	Buscar uma nota fiscal eletrônica
C05	Listar todas as notas fiscais eletrônicas existentes
C06	Excluir uma nota fiscal eletrônica

Fonte: Do autor

6.2.1.4 Diagramas de casos de uso

Os casos de uso relevantes, referentes à Nota Fiscal Eletrônica, descobertos no processo de análise e modelagem são demonstrados na figura 14.

Figura 14 – Casos de Uso da Nota Fiscal Eletrônica

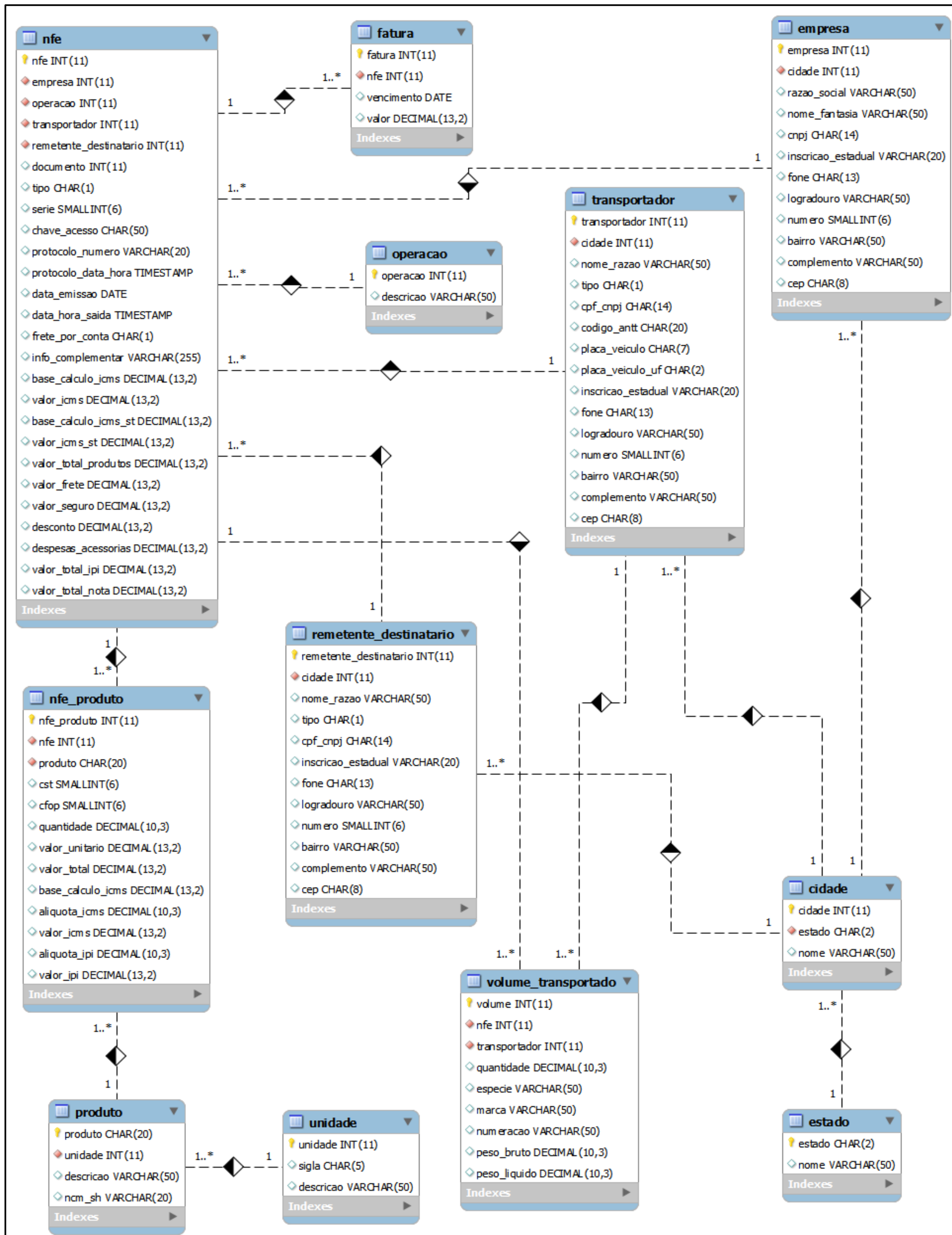


Fonte: Do autor

6.2.1.5 Modelagem do banco de dados

A figura 15 demonstra o diagrama entidade-relacionamento das tabelas do banco de dados utilizado no protótipo.

Figura 15 – Diagrama Entidade-Relacionamento do Banco de Dados

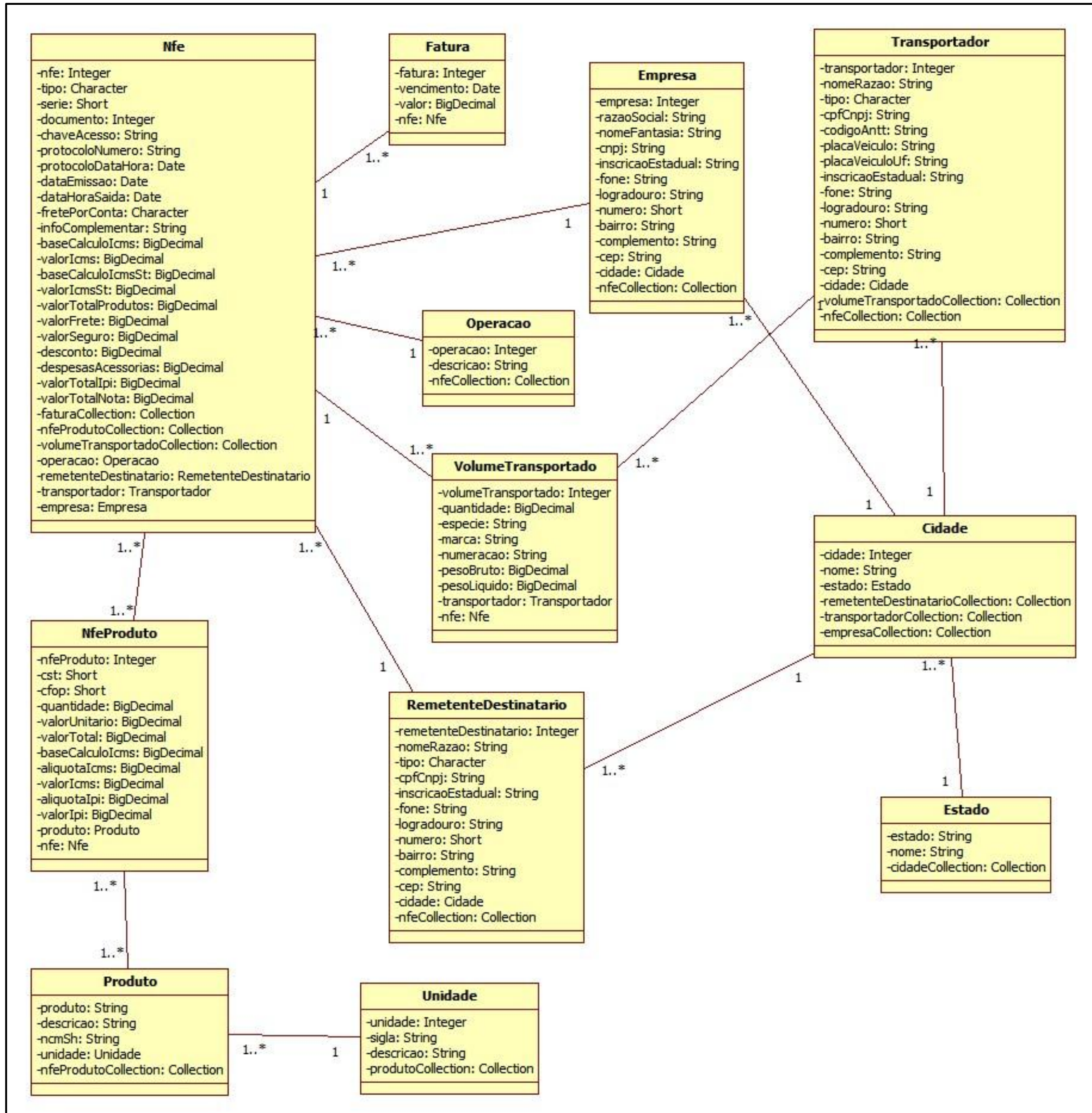


Fonte: Do autor

6.2.1.6 Diagrama de classes do cenário

Na figura 16, é possível visualizar todas as classes do sistema e seus relacionamentos.

Figura 16 – Diagrama das classes do protótipo



Fonte: Do autor

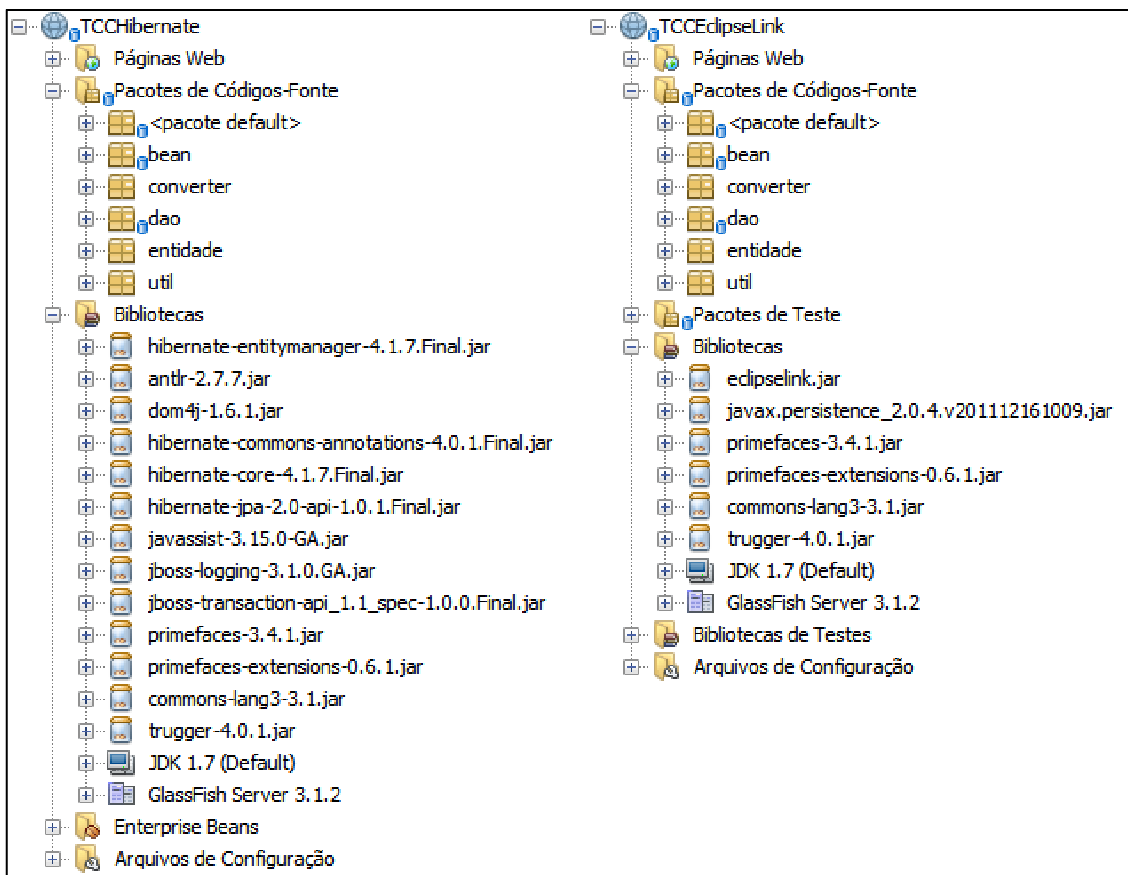
6.2.2 Desenvolvimento do protótipo

Após a análise das informações levantadas e da modelagem, passou-se para a fase de desenvolvimento do protótipo. Foi desenvolvida uma aplicação Web no ambiente de

desenvolvimento NetBeans, utilizando-se os frameworks JSF - *JavaServerFaces*[9] e PrimeFaces[10].

Conforme figura 17, pode-se verificar a estrutura de cada protótipo:

Figura 17 – Estrutura dos protótipos com Hibernate e EclipseLink



Fonte: Do autor

A estrutura do projeto é descrita na tabela 5.

Tabela 5 – Pacotes dos protótipos

Pacote	Nome
bean	Contém os *managed beans*[11] da aplicação.
converter	Contém conversores de tipos.
dao	Contém os DAOs[12], sendo um para cada

[9] Framework do lado do servidor de componentes para desenvolvimento web, baseados em Java (ORACLE, 2012, tradução nossa).
[10] Framework auxiliar para JSF. Contém componentes visuais que estendem os componentes padrão.
[11] POJOs gerenciados pelo container de servidor de aplicação. Providenciam a lógica da aplicação, tendo ligação com os componentes da página. Em geral, cada página possui um *managed bean* (SUN, 2009, tradução nossa).

	entidade.
entidade	Contém as classes de entidade que fazem o mapeamento objeto-relacional das tabelas do banco de dados.
util	Contém classes com métodos utilitários.

Fonte: Do autor

Os projetos diferenciam-se entre si pelas bibliotecas, evidenciadas na imagem 7, e pelo arquivo de configuração *persistence.xml*, situado na pasta “Arquivos de configuração”.

6.2.2.1 Mapeamento

As tabelas do banco de dados presentes na figura 15 foram mapeadas para as classes demonstradas na figura 16, com a utilização das anotações demonstradas no capítulo sobre JPA. A figura 18 mostra o mapeamento das colunas e relacionamentos da tabela cidade para a classe Cidade.

Figura 18 – Exemplo de mapeamento para a classe Cidade

```
@Entity
@Table(name = "cidade")
public class Cidade implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Id
    private Integer cidade;

    private String nome;

    @JoinColumn(name = "estado", referencedColumnName = "estado")
    @ManyToOne(optional = false)
    private Estado estado;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "cidade")
    private Collection<RemetenteDestinatario> remetenteDestinatarioCollection;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "cidade")
    private Collection<Transportador> transportadorCollection;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "cidade")
    private Collection<Empresa> empresaCollection;
```

Fonte: Do autor

[12] *DataAcessObjects* – DAO: padrão de projetos, encapsula a lógica de conexão e operações com o banco de dados em um aplicação (KEITH; SCHINCARIOL, 2009, tradução nossa).

6.2.2.2 Gerenciamento das entidades

Para realizar o gerenciamento das entidades, foi criada uma classe chamada *AbstractDao,* que implementa o padrão de projeto DAO, e contém métodos que encapsulam as principais operações do *EntityManager*.

Figura 19 – Classe AbstractDao com os métodos CRUD principais do EntityManager

```
public abstract class AbstractDao<T>
{
    private Class<T> entityClass;

    protected abstract EntityManager getEntityManager();

    public AbstractDao(Class<T> entityClass)
    {
        this.entityClass = entityClass;
    }

    public void create(T entidade)
    {
        getEntityManager().persist(entidade);
    }

    public void edit(T entidade)
    {
        getEntityManager().merge(entidade);
    }

    public void remove(Object id)
    {
        getEntityManager().remove(this.find(id));
    }

    public T find(Object id)
    {
        return getEntityManager().find(entityClass, id);
    }
```

Fonte: Do autor

Foram criadas classes filhas de *AbstractDao*, sendo uma para cada entidade, e essas são utilizadas nos *managed beans* de cada página de cadastro de entidade para realização da operações com o banco de dados.

Figura 20 – Classe CidadeDao filha de AbstractDao

```
@Stateless
public class CidadeDao extends AbstractDao<Cidade>
{
    @PersistenceContext(unitName = "tcc")
    private EntityManager em;

    public CidadeDao()
    {
        super(Cidade.class);
    }

    @Override
    protected EntityManager getEntityManager()
    {
        return em;
    }
}
```

Fonte: Do autor

6.3 FERRAMENTA PARA REALIZAÇÃO DAS COMPARAÇÕES

Para a realização das comparações, utilizou-se a ferramenta de *profiling* presente no ambiente de desenvolvimento NetBeans, o qual foi utilizado para o desenvolvimento dos protótipos. A ferramenta permite a realização dos seguintes testes:

a) monitor de threads: permite que sejam monitoradas as threads da aplicação em execução;
b) CPU: Para a realização de testes de desempenho;
c) memória: Para acompanhar o consumo de memória da aplicação.

Foram utilizados somente os dois últimos testes supracitados, por estarem de acordo com o objetivo deste trabalho.

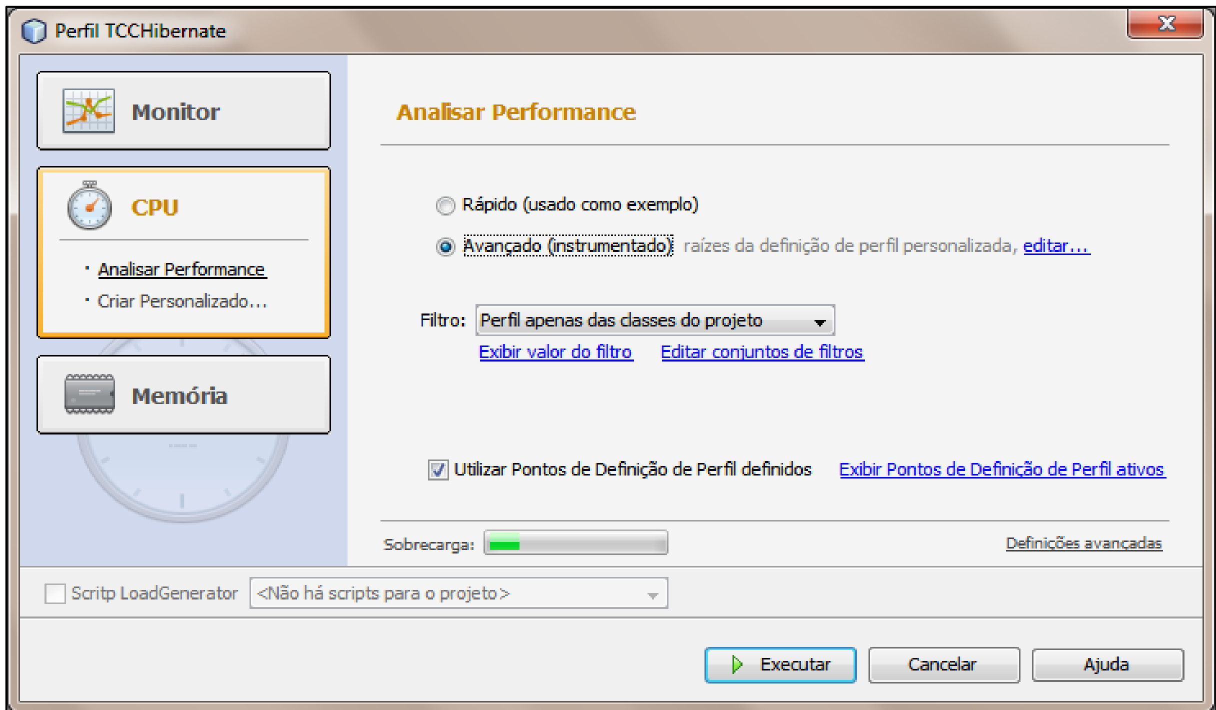
6.3.1 Desempenho

Ao selecionar o modo de teste CPU, é mensurado o desempenho da aplicação em nível de métodos, ou seja, quanto tempo cada um levou para executar. Existem dois modos para esse teste:

a) rápido (usado como exemplo): Verifica a aplicação como um todo. É menos precisa que o modo Avançado (instrumentado), mas possui uma sobrecarga menor.

b) avançado (instrumentado): Permite a seleção de métodos específicos para monitoramento. Pode-se definir um filtro, que vai desde todos os métodos da aplicação até um simples fragmento de código. A sobrecarga desse modo é diretamente proporcional à quantidade de métodos monitorados.

Figura 21 – Tela principal do NetBeans Profiler - Teste de Desempenho



Fonte: Do autor

6.3.2 Memória

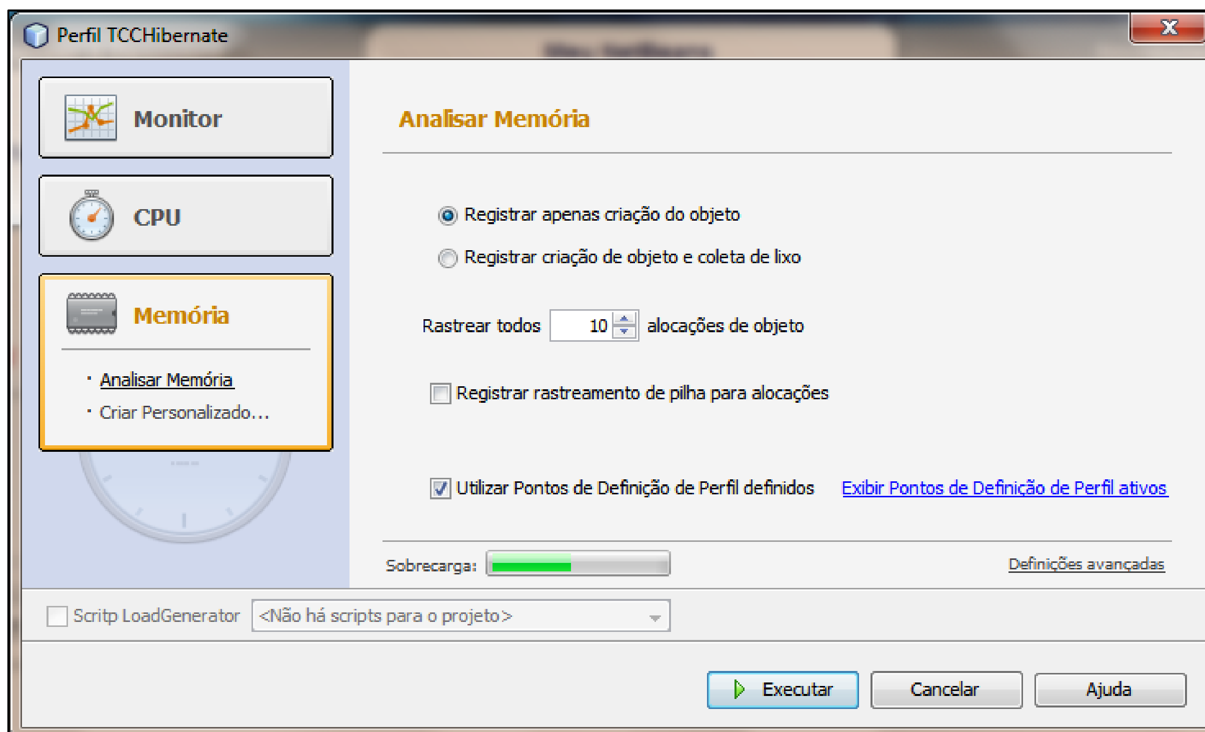
Esse modo de teste permite verificar o consumo de memória pela aplicação, demonstrando informações agrupadas por tipo de objeto, tais como quantidade de alocações, total de bits alocados, total de objetos que não foram recolhidos pelo coletor de lixo[13], tempo de vida médio, entre outros. Existem dois modos para esse teste:

a) Registrar apenas criação do objeto: todas as classes carregadas pela *Java Virtual Machine* – JVM, bem como novas classes carregadas, serão monitoradas;

[13] Programa executado pela JVM, que libera a memória utilizada por um objeto que não está mais sendo utilizado em nenhum ponto da aplicação.

b) Registrar criação do objeto e coleta de lixo: monitora o ciclo de vida de um objeto, que corresponde desde a criação do mesmo até seu recolhimento pelo coletor de lixo.

Figura 22 – Tela principal do NetBeans Profiler – Teste de Memória



Fonte: Do autor

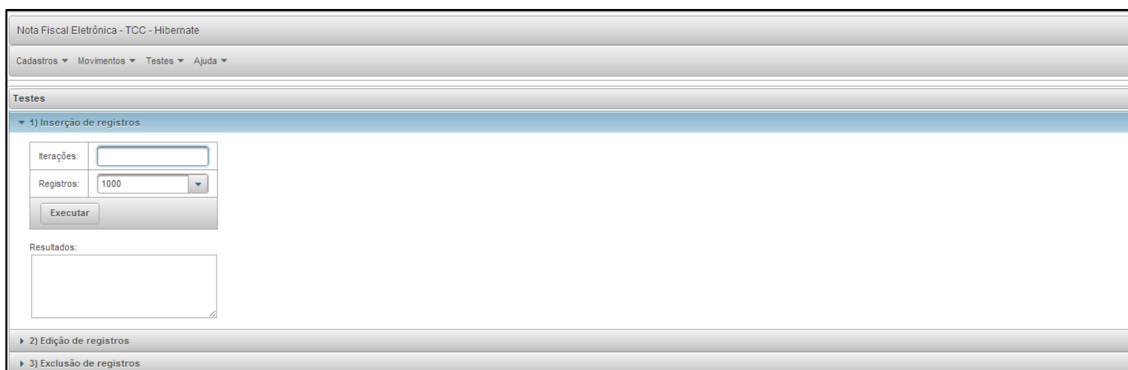
6.4 REALIZAÇÃO DAS COMPARAÇÕES

Para a realização das comparações, foram executadas e mensuradas operações de persistência e consultas com a entidade Nfe, que representa uma nota fiscal eletrônica. As operações de persistência realizadas consistiram na inserção, edição e remoção de mil e dez mil registros. A operação de consulta consistiu na realização de uma consulta com junções, agrupamento e funções de agregação, com dez mil notas fiscais eletrônicas (Nfe), sendo que cada uma possuía dez produtos (NfeProduto).

Para a execução dos testes, foi desenvolvida uma página, que pode ser acessada no menu principal, opção “Testes”, item “Testes”, onde serão exibidos os critérios disponíveis. Ao clicar em um dos critérios, será exibido um formulário, conforme visto na figura 23.

Para realizar um teste mais igualitário entre os frameworks, não se utilizou o cache de segundo nível[14], por não haver a mesma biblioteca para essa função presente no EclipseLink e Hibernate, o que poderia ocasionar discrepâncias.

Figura 23 – Página de automação dos testes



Fonte: Do autor

A função dos componentes exibidos nessa página é demonstrada na tabela 6.

Tabela 6 – Função dos componentes da página de teste

Componente	Função
Campo iterações	Permite informar o número de vezes que o teste irá executar.
Combobox Registros	Permite informar a quantidade de registros que serão adicionados, editados ou removidos. Estão disponíveis as opções 1000 e 10000.
Botão Executar	Executa o teste.

Fonte: Do autor

Para essa página, há um *managed bean* chamado "TesteBean", situado no pacote "bean", que gerencia a preparação e realização dos testes. Foram os métodos desse objeto que foram monitorados pelo NetBeans Profiler, para medição do desempenho e consumo de memória.

[14] O cache de segundo nível é um armazenamento local de entidades, gerenciado pelo provedor de persistência, para melhorar a performance da aplicação (SUN, 2009, tradução nossa).

Nos testes de desempenho, foi selecionado o teste “CPU”, e filtrado um método específico da classe TesteBean para ser monitorado, variando conforme o tipo de operação de persistência.

Já para os testes de memória, foi escolhido o teste “Memória”. Foi selecionada a opção “Registrar criação do objeto e coleta de lixo”, para monitorar o ciclo de vida dos objetos alocados.

6.4.1 Teste de Inserções

Para o teste de inserções, foi monitorado o tempo de execução do método executaTesteInsercao do *managed bean* TesteBean. O método é demonstrado na figura 24:

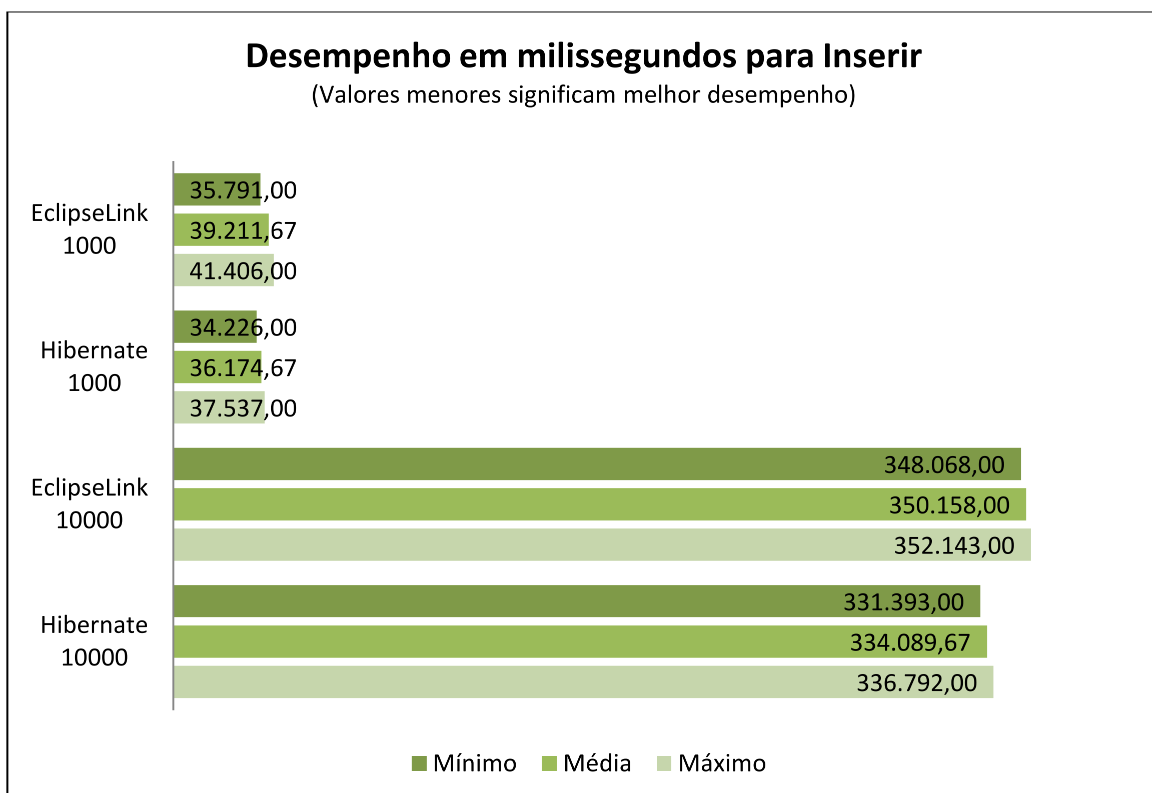
Figura 24 – Método que executa o teste de inserção

```
private void executaTesteInsercao()
{
    for (int j = 1; j <= registros; j++)
    {
        Nfe nfe = this.criaNfeInsercao();
        dao.create(nfe);

        if (j % 20 == 0)
        {
            dao.flushAndClear();
        }
    }
}
```

Fonte: Do autor

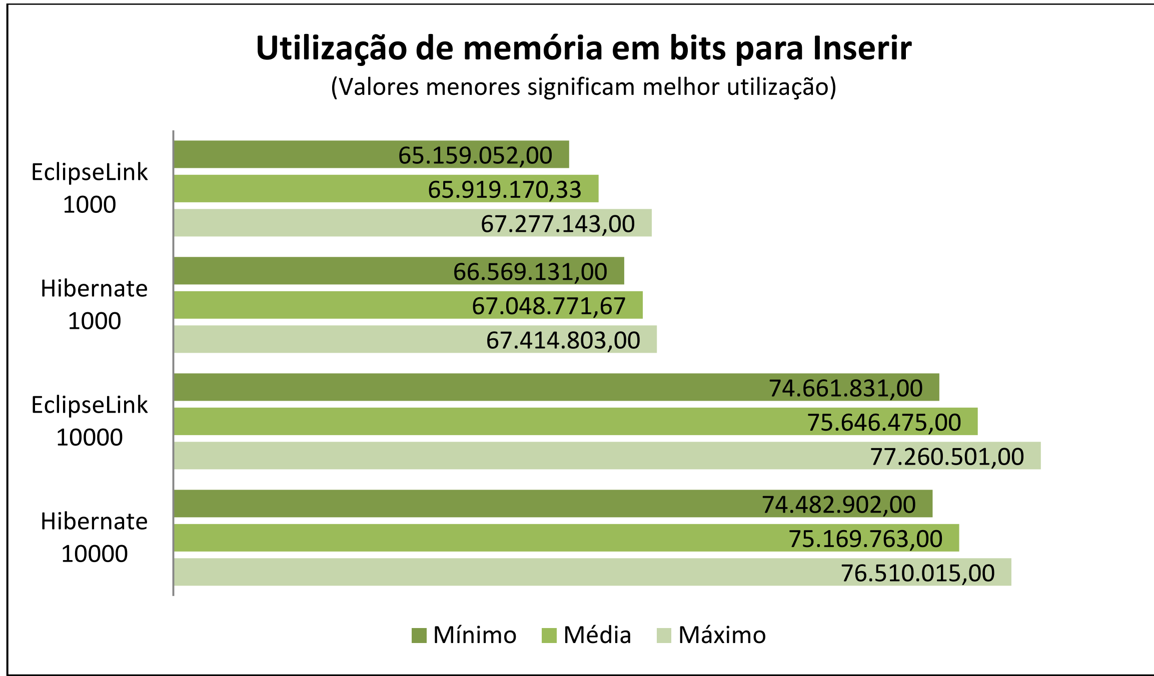
Figura 25 – Resultados dos testes de inserção – Desempenho



Fonte: Do autor

Conforme a figura 25, verificou-se que o Hibernate foi um pouco mais rápido que o EclipseLink, ao inserir mil registros, ampliando a diferença com dez mil registros.

Figura 26 – Resultados dos testes de inserção – Memória



Fonte: Do autor

Nos resultados do teste de memória, a situação foi diferente, com o EclipseLink consumindo menor quantidade de memória ao inserir mil registros, mas perdendo para o Hibernate com dez mil registros.

6.4.2 Teste de Edições

Para o teste com operações de edição de dados, foi monitorado o tempo de execução do método executaTesteEdicao do *managed bean* TesteBean. O método é demonstrado na figura 27:

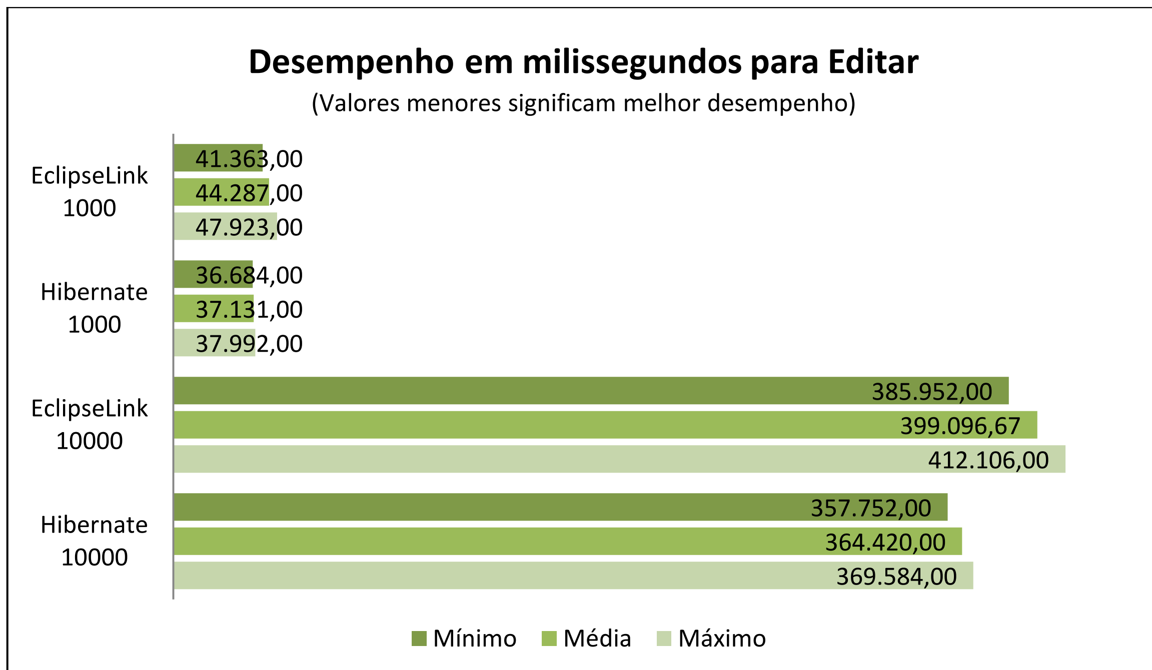
Figura 27 – Método que executa o teste de remoção

```
private void executaTesteEdicao()
{
    for (int j = 0; j < registros; j++)
    {
        Nfe nfe = nfes.get(j);
        this.alteraNfeEdicao(nfe);
        dao.edit(nfe);

        if (j % 20 == 0)
        {
            dao.flushAndClear();
        }
    }
}
```

Fonte: Do autor

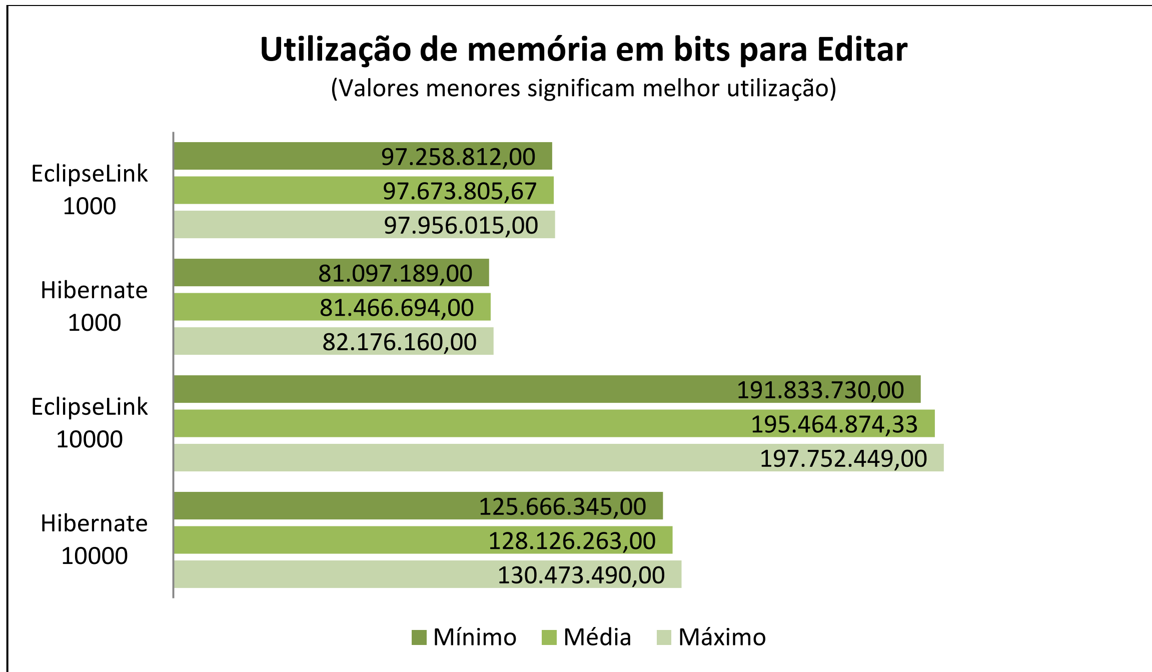
Figura 28 – Resultados dos testes de edição – Desempenho



Fonte: Do autor

Para esse teste, novamente o Hibernate superou o EclipseLink em desempenho, com uma diferença superior de tempo do que nas operações de inserção, para mil e dez mil registros.

Figura 29 – Resultados dos testes de edição – Memória



Fonte: Do autor

Para o quesito Memória, EclipseLink teve um consumo maior em mil e dez mil registros, destacando-se no último, onde registrou os maiores valores dentre as três operações de persistência testadas, e a operação de consulta.

6.4.3 Teste de Remoções

Para o teste de remoções, foi monitorado o tempo de execução do método executaTesteRemocao do *managed bean* TesteBean. O método é demonstrado na figura 30:

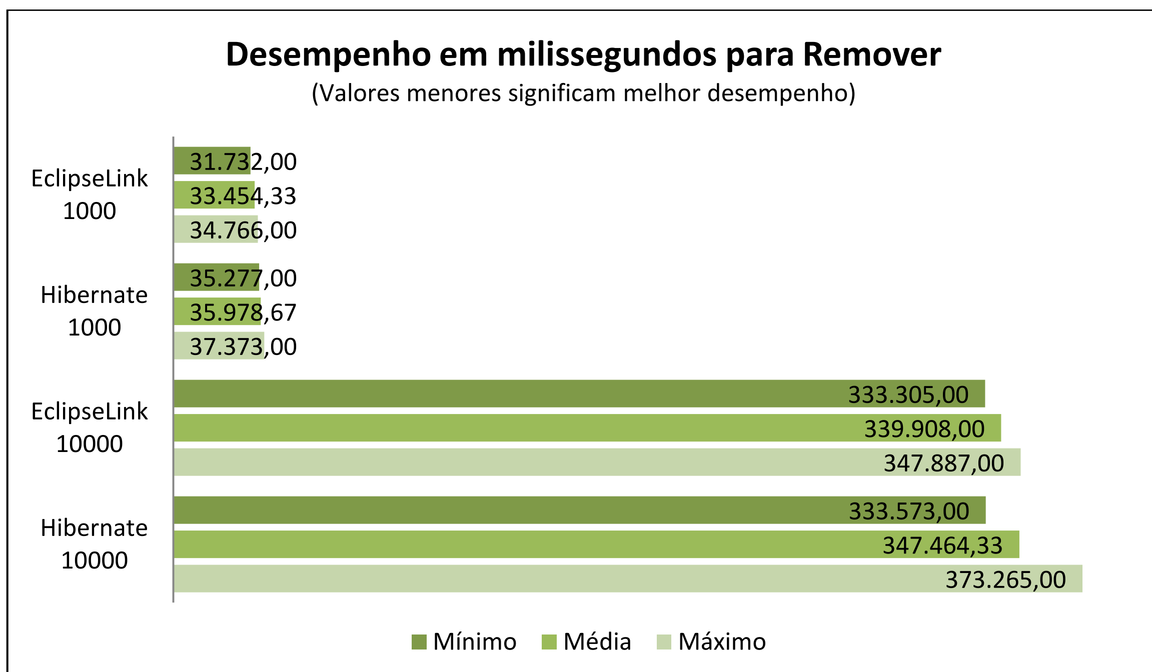
Figura 30 – Método que executa o teste de remoção

```
private void executaTesteExclusao()
{
    for (int j = 0; j < registros; j++)
    {
        dao.remove(nfes.get(j).getNfe());

        if (j % 20 == 0)
        {
            dao.flushAndClear();
        }
    }
}
```

Fonte: Do autor

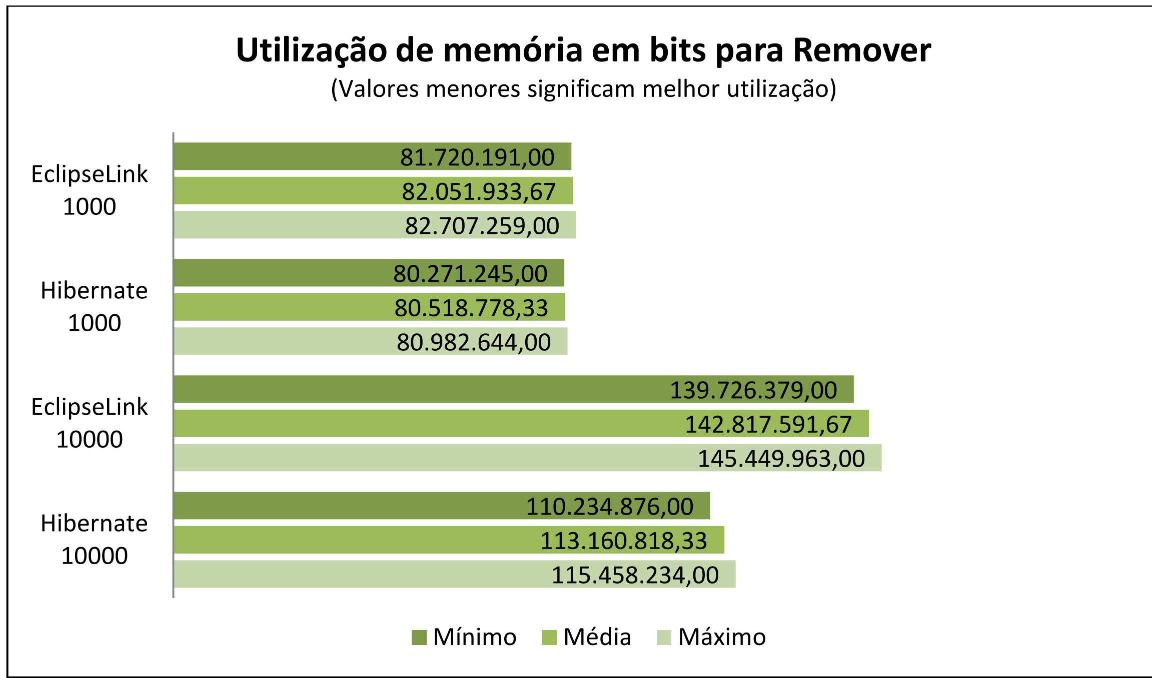
Figura 31 – Resultados dos testes de remoção – Desempenho



Fonte: Do autor

No teste de remoções, EclipseLink teve um desempenho superior em ambos os casos.

Figura 32 – Resultados dos testes de remoção – Memória



Fonte: Do autor

O EclipseLink teve um maior consumo de memória que o Hibernate com mil e dez mil registros, conforme figura 32, sendo uma diferença mínima com mil, e muito maior com dez mil.

6.4.4 Teste de Consultas

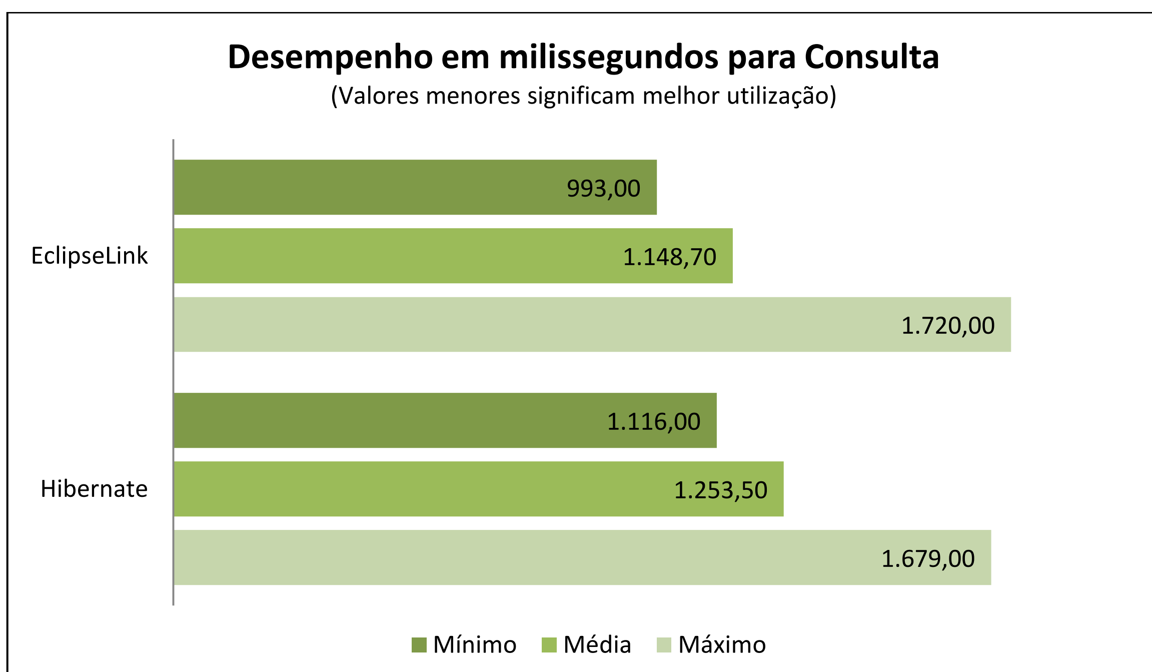
Para o teste de consultas, foi monitorado o tempo de execução do método executaTesteConsulta do *managed bean* TesteBean. O método executa a consulta demonstrada na figura 33:

Figura 33 – Consulta realizada para o teste

```
    SELECT n.nfe,
           n.documento,
           n.tipo,
           n.dataEmissao,
           n.dataHoraSaida,
           n.valorTotalNota,
           n.valorTotalProdutos,
           t.transportador,
           t.nomeRazao,
           r.remetenteDestinatario,
           r.nomeRazao,
           e.empresa,
           e.razaoSocial,
           o.operacao,
           MAX(np.aliquotaIcms),
           MAX(np.valorIpi),
           SUM(np.valorIcms),
           SUM(np.valorIpi),
           SUM(np.quantidade)
    FROM Nfe AS n JOIN
           n.nfeProdutoCollection AS np JOIN
           n.transportador AS t JOIN
           n.remetenteDestinatario AS r JOIN
           n.empresa AS e JOIN
           n.operacao AS o
GROUP BY n.nfe, n.documento, n.tipo, n.dataEmissao, n.dataHoraSaida, n.valorTotalNota, n.valorTotalProdutos,
           t.transportador, t.nomeRazao, r.remetenteDestinatario, r.nomeRazao, e.empresa, e.razaoSocial, o.operacao
ORDER BY n.nfe
```

Fonte: Do autor

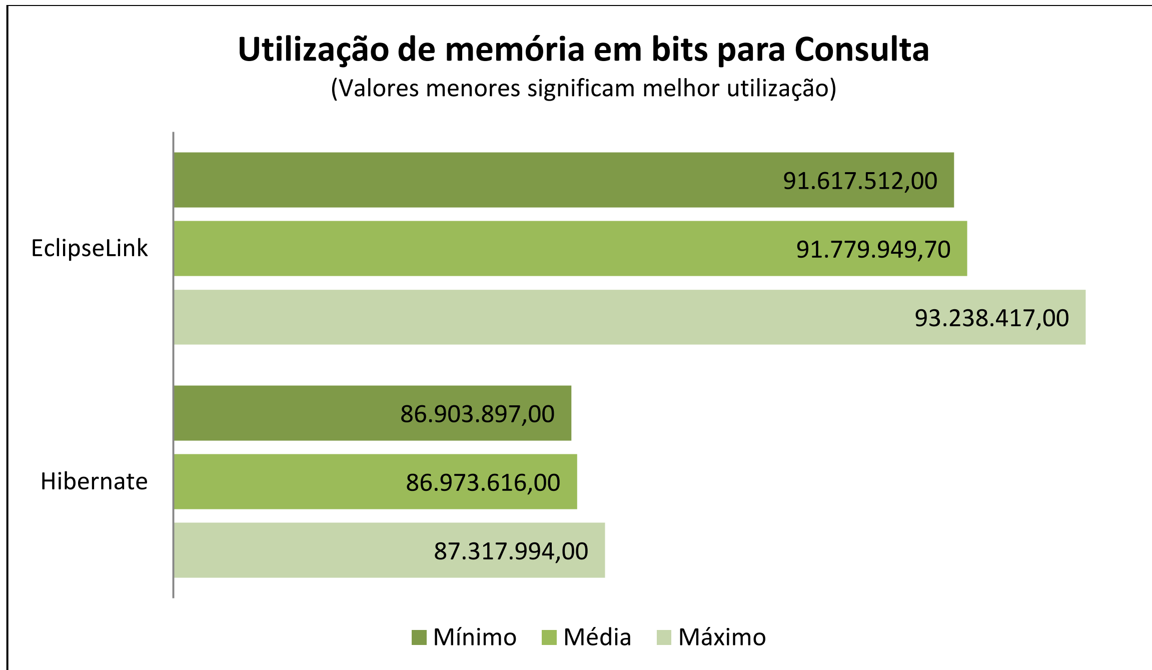
Figura 34 – Resultado do teste de consulta – Desempenho



Fonte: Do autor

Para o teste de consultas, em relação ao critério de desempenho, conforme a figura 34, observou-se que o EclipseLink teve um desempenho maior que o Hibernate, tanto no melhor quanto no médio caso, mas pior no pior caso.

Figura 35 – Resultado do teste de consulta – Memória



Fonte: Do autor

Já em relação ao consumo de memória, demonstrado na figura 35, apesar do EclipseLink possuir melhor desempenho, possui consumo de memória superior ao Hibernate para a realização da consulta proposta.

Observou-se que, após a realização dos testes de desempenho, o Hibernate demonstrou-se mais performático nas operações de inclusão e edição, e foi superado nas operações de remoção e consulta pelo EclipseLink. Para os testes de memória, o Hibernate teve menor consumo, para todos os testes, exceto o teste de edição com mil registros, no qual o EclipseLink consumiu menos memória.

7 CONCLUSÃO

Nessa pesquisa, foi realizada a comparação de desempenho e consumo de memória entre os frameworks de Mapeamento Objeto-Relacional Hibernate e EclipseLink. Para a realização das comparações partiu-se da pesquisa e desenvolvimento de dois protótipos de aplicação idênticos, um para cada framework, sendo que ambos divergem somente nas bibliotecas utilizadas.

Após as comparações, constatou-se que o Hibernate teve um desempenho melhor que o EclipseLink nos testes de inserções e edições, com mil e dez mil registros, perdendo nos testes de remoções com mil e dez mil registros, e no teste de consulta. Nos testes de memória, o Hibernate teve menor consumo em todos os testes, exceto para mil registros no teste de edições.

Nos cenários criados, o Hibernate mostrou-se mais vantajoso, por possuir um desempenho superior na metade das operações, combinado com menor consumo de memória na maioria das operações, o que lhe proporciona um custo-benefício (consumo de memória/desempenho) superior ao do EclipseLink. Outras vantagens do Hibernate consistem na ampla documentação presente, bem como maturidade e vasta comunidade de usuários ativos.

Apesar dos resultados obtidos, não se deve desmerecer o framework EclipseLink, que, mesmo sendo uma solução de persistência mais recente, possui uma grande comunidade que o mantém (Eclipse Foundation), e foi escolhida como a implementação de referência da JPA 2.0. Como o Hibernate, também existe a opção de cache de segundo nível, podendo acarretar melhoras perceptíveis no desempenho e consumo de memória para operações de persistência com o banco de dados.

As dificuldades do projeto consistiram no aprendizado e desenvolvimento da aplicação Web, com JSF e PrimeFaces, e utilizando Hibernate e EclipseLink para a persistência dos dados, bem como a utilização e análise dos dados com o NetBeans Profiler. Graças à ampla documentação disponível em livros, Web e auxílio de fóruns de usuários, essas dificuldades iniciais foram superadas com facilidade.

Os objetivos gerais e específicos foram atingidos com sucesso, e os resultados documentados podem servir de auxílio no processo de tomada de decisão de um framework de Mapeamento Objeto-Relacional em Java.

Para a realização de trabalhos futuros, sugere-se:

a) Utilização de uma aplicação de porte empresarial para execução dos testes;

b) Comparação com outros frameworks ORM que implementam a JPA;
c) Utilização de cache de segundo nível na realização dos testes.

REFERÊNCIAS

AMBLER, Scott W.. **The Object-Relational Impedance Mismatch.** Disponível em: <http://www.agiledata.org/essays/impedanceMismatch.html>. Acesso em: 4 ago. 2012.

BARROS, Bruno Alberth Silva; BARROS, Raul Silva; CORTES, Omar Andres Carmona. Um Estudo Comparativo entre APIs de Persistência Utilizando Grandes Volumes de Dados. In: CONGRESSO DE PESQUISA E INOVAÇÃO DA REDE NORTE E NORDESTE DE EDUCAÇÃO TECNOLÓGICA, 4., 2009, Belém - Pa. **Um Estudo Comparativo entre APIs de Persistência Utilizando Grandes Volumes de Dados.** Belém - Pa: Rede Norte e Nordeste de Educação Tecnológica, 2009. p. 1 - 8.

BAUER, Christian; KING, Gavin. **Hibernate in Action.** Greenwich: Manning, 2005. 431 p.

CAMPOS, Raphael Barreto Palhares de. **Análise Comparativa de Frameworks de Persistência.** 2010. 64 f. Monografia (Graduação) - Universidade Federal de Lavras, Lavras, 2010.

CORDEIRO, Jader Dos Santos Teles. **Estudo Comparativo entre os frameworks de mapeamento objeto-relacional Hibernate e Toplink.** 2011. 55 f. Monografia (Especialização) - Universidade Estadual de Maringá, Maringá, 2011.

CORONEL, Carlos; MORRIS, Steven; ROB, Peter. **Database Systems:** Design, Implementation, and Management. 9. ed. Boston: Cengage Learning, 2011.

DATE, C. J.. **Introdução a sistemas de bancos de dados.** 8. ed. São Paulo: Elsevier, 2004.

DEITEL, Paul J.; DEITEL, Harvey M.. **Java Como Programar.** 9. ed. São Paulo: Prentice Hall, 2012.

ECKEL, Bruce. **Thinking in Java.** 4. ed. Stoughton: Prentice Hall, 2006.

ELMASRI, Ramez; NAVATHE, Shamkant B..**Sistema de Banco de Dados.** 6. ed. São Paulo: Addison-wesley, 2011.

GONCALVES, Antonio. **Beginning Java EE 6 Platform with Glassfish 3.** 2. ed. New York: Apress, 2010. 536 p.

GOSLING, James et al. **The Java Language Specification.** 3. ed. New York: Addison-wesley, 2005.

JBOSS. **HIBERNATE - Relational Persistence for Idiomatic Java.** Disponível em: <http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html/>. Acesso em: 14 ago. 2012.

KEITH, Mike; HALEY, Jason; SCHINCARIOL, Merrick. **Pro EJB 3:** Persistence. New York: Apress, 2006. 480 p.

KEITH, Mike; SCHINCARIOL, Merrick. **Pro JPA 2:** Mastering the Java™ Persistence API. New York: Apress, 2009. 500 p.

MAKI, Christopher. **Jpa 101:** Java Persistence Explained. Highlands Ranch: Sourcebeat, 2007. 536 p.

MATOS, Cristina da Silva. **Comparação entre o uso de JDBC e Hibernate para persistência de dados.** 2011. 99 f. Monografia (Graduação) - Universidade do Extremo Sul Catarinense, Criciúma, 2011.

MICROSYSTEMS, Sun. **JSR 317 - Java Persistence API, Version 2.0.** Disponível em: <http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-eval-oth-JSpec/>. Acesso em: 14 ago. 2012.

ORACLE. **The Java EE 6 Tutorial.** Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/>. Acesso em: 14 ago. 2012.

POTHU, Sudhakar. **A Comparative Analysis of Object-relational mappings for Java.** 2008. 108 f. Dissertação (Mestrado) - University Of Applied Sciences, Braunschweig, 2008.

RICARTE, Ivan Luiz Marques.**Programação Orientada a Objetos:**Uma abordagem com Java. 2001. 118 f. Monografia (Graduação) - Universidade Estadual de Campinas, Campinas, 2001.

TIOBE. **TIOBE Programming Community Index for July 2012.** Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 4 ago. 2012.

YOUSAF, Haseeb. **Performance Evaluation of Java Object-Relational Mapping Tools.** 2012. 121 f. Dissertação (Mestrado) - University Of Georgia, Athens, 2012.

Comparação de Desempenho e Consumo de Memória entre Frameworks de Mapeamento Objeto-Relacional Java Hibernate E EclipseLink

Juliano Marques[1], Paulo João Martins[2]

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

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

julianomqs@gmail.com, pjm@unesc.net

Abstract*. With the arrival of several tools that perform object-relational mapping to solve the problem of Impedance Data Mismatch, there were also questions about which tools would be the best, what influences the choice of tool to use. Two of the main criteria guiding this choice would be performance and memory consumption. Based on these criteria, was performed a comparison between two object-relational mapping frameworks in Java that have widespread use in the market, Hibernate and EclipseLink, in routine operations such as inserting, editing, removing, and querying data, to check tool which has the best cost-benefit, which cost was memory consumption, and performance was benefit.*

Resumo*. Com o surgimento de diversas ferramentas que realizam o mapeamento objeto-relacional para resolver o problema da Impedância de Dados, surgiram também dúvidas sobre qual das ferramentas seria a melhor, o que influencia na escolha da ferramenta para utilização. Dois dos principais critérios que norteiam essa escolha seriam o desempenho e consumo de memória. Com base nesses critérios, realizou-se uma comparação entre dois frameworks de mapeamento objeto-relacional da linguagem Java que possuem grande utilização no mercado, o Hibernate e o EclipseLink, em operações rotineiras como inserção, edição, remoção e consulta de dados, para verificação de qual ferramenta possui o melhor custo-benefício, onde custo seria o consumo de memória, e benefício seria o desempenho.*

1. Introdução

Com a popularização do mapeamento objeto-relacional (ORM – Object-Relational Mapping) como solução mais viável do problema da Impedância de Dados, surgiram diversas ferramentas que realizam essa tarefa (BAUER; KING, 2005). Isso acarretou dúvidas nos desenvolvedores de aplicações, principalmente as corporativas, sobre qual das ferramentas disponíveis seria a melhor, em diversos critérios, tais como:

a) Qual das ferramentas possui o melhor desempenho?
b) Qual das ferramentas consome menor quantidade de memória?

Na linguagem Java, uma das mais utilizadas no mundo, o mapeamento objeto-relacional é utilizado a um bom tempo. Por padrão, costuma-se utilizar o framework Hibernate, que é uma ferramenta open-source, mantida pela JBoss, que realiza o mapeamento

através da utilização de anotações ou arquivos XML. Com ampla adoção na comunidade de desenvolvedores e vasta documentação, o Hibernate tornou-se um padrão em Java para mapear tabelas do banco de dados para classes, inspirando até a criação de uma especificação para essa tarefa, a Java Persistence API – JPA (JBOSS, 2012).

Mas, surgiu recentemente no mercado dos frameworks ORM, a solução EclipseLink, também open-source, mantida pela Eclipse Foundation. Essa ferramenta teve grande adoção, e foi escolhida como implementação de referência da versão 2.0 da JPA. Pelo fato de tanto o Hibernate quanto o EclipseLink implementarem a JPA, eles tem funcionamento idêntico.

Para sanar as duas dúvidas levantadas no primeiro parágrafo, que influenciam na tomada de decisão entre uma ferramenta de mapeamento objeto-relacional ou outra, foi realizado um comparativo, entre o Hibernate e o EclipseLink, que possuem alta visibilidade no mercado, nos critérios de desempenho e consumo de memória, em operações rotineiras como inserções, remoções, edições e consulta de dados. Foram desenvolvidos dois protótipos que simulam aplicações corporativas reais para maior realismo das comparações.

Com o desenvolvimento dos protótipos de exemplo e a realização das comparações, as duas questões levantadas no primeiro parágrafo foram respondidas apropriadamente, demonstrando qual das ferramentas possui o melhor custo-benefício, configurando-se custo como consumo de memória e benefício como desempenho obtido, dessa forma, auxiliando na escolha de uma ferramenta de mapeamento objeto-relacional para desenvolvedores de aplicações.

2. Definição do Problema

Com o surgimento e a popularização de computadores, e com sua adoção expressiva em corporações, surgiu a necessidade de se manter os dados obtidos pelas aplicações após o desligamento da máquina. Após a utilização de sistemas de arquivos computadorizados, procurou-se uma nova solução de armazenamento mais simples, segura, rápida e confiável, e aí surgiu o conceito de banco de dados, que é uma estrutura integrada e compartilhada que armazena uma coleção de dados do interesse do usuário e metadados, ou seja, dados sobre os dados armazenados (CORONEL; MORRIS; ROB, 2011, tradução nossa).

Os bancos de dados tiveram uma adoção massiva com o modelo relacional, criado na década de 1970 por Codd, e implementado comercialmente a partir da década de 1980 (ELMASRI; NAVATHE, 2011). Isso se deve a recursos fornecidos como maior segurança, melhor integração, aumento de produtividade e acesso facilitado à informação.

Para lidar com esses bancos de dados, as empresas adotaram linguagens de programação que utilizavam o paradigma da orientação a objeto. O mesmo visa à modelagem de domínio de problemas em objetos, componentes compostos de estados e comportamentos. O paradigma diferencia-se por recursos como reutilização, redução e menor escrita de código, flexibilidade e manutenibilidade (BAUER, KING, 2005, tradução nossa).

Uma das linguagens que utilizam esse paradigma é a Java, criada por James Gosling, da Sun Microsystems. Conforme Oracle (2012, tradução nossa), possui uma sintaxe amigável, segura, orientada a objeto e portável, foi utilizada em larga escala para aplicativos empresariais, tanto nas plataformas desktop, web e móvel.

Porém, a utilização de linguagens orientadas a objeto, como Java, em conjunto com bancos de dados relacionais, levou ao surgimento do problema conhecido como Impedância Objeto-Relacional, caracterizado pela não integração dos paradigmas orientados a objeto e relacional, por possuírem princípios diferentes (AMBLER, 2012, tradução nossa). Com isso, buscaram-se soluções para o mesmo, e a mais viável foi o mapeamento objeto-relacional.

Com uma premissa simples, de mapear tabelas do banco de dados para classes, e pela eficácia da mesma, tornou-se muito utilizada e considerada, culminando na criação de uma

especificação na linguagem Java para realização de persistência, a Java Persistence API - JPA, que segue este modelo.
Com isso, surgiram inúmeros frameworks seguindo esse preceito, acarretando dúvidas nas corporações, levando a questionamentos, tais como:

a) Qual ferramenta possui o melhor desempenho em operações com o banco de dados?
b) Qual ferramenta consome menor quantidade de memória RAM?

Para esse fim, será desenvolvido um estudo, que pretende realizar um comparativo entre os frameworks com maior visibilidade no mercado, o Hibernate e o EclipseLink, com base no desempenho e utilização de memória RAM em operações com o banco de dados. Considerando a importância dos quesitos da celeridade no processamento de informações e utilização otimizada dos recursos da máquina, valida-se este estudo, como auxílio na tomada de decisão de arquitetos e engenheiros de software sobre a ferramenta de mapeamento objeto-relacional entre as elencadas que possui um melhor custo-benefício, sendo o custo o consumo de memória e benefício como o desempenho na execução de operações.

3. Comparação de Desempenho e Consumo de Memória entre Frameworks de Mapeamento Objeto-Relacional Java Hibernate e EclipseLink

Para a realização da comparação de desempenho e consumo de memória entre os frameworks ORM Hibernate e EclipseLink, estudaram-se as ferramentas e desenvolveram-se dois protótipos de aplicação, sendo um para cada framework.

Para os protótipos de aplicação desenvolvidos modelou-se o cenário de uma aplicação Web, para cadastro de Notas Fiscais Eletrônicas, permitindo operações de inserção, edição, remoção e consulta de notas fiscais e tabelas associadas. Ambos os protótipos possuem estrutura e funcionamento idênticos, excetuando-se as bibliotecas de código utilizadas e o arquivo *persistence.xml*, que possuem configurações específicas para cada framework de persistência.

Utilizou-se no desenvolvimento a linguagem Java, versão 6, no ambiente de desenvolvimento NetBeans, versão 7.2. O SGBD utilizado foi o MySQL, versão 5.5, e as versões utilizadas do Hibernate e EclipseLink foram a 4.1.7 e a 2.4.0, respectivamente.

O princípio que norteou a criação dos protótipos foi a utilização e demonstração dos conceitos como Orientação a Objetos, Mapeamento Objeto-Relacional e Bancos de Dados. Por isso, foram criados requisitos que atendessem os dois conceitos, demonstrados na tabela 1.

Tabela 7 – Recursos de Orientação a Objetos e de Banco de Dados utilizados

Recursos de OO	Recursos de Bancos de Dados
Classe simples	Consulta com *inner join*
Classe com tipo complexo	Consulta com função de agregação
Classe abstrata	Consulta por chave primária
Coleção	Consulta com critério
Herança	Relacionamento um-para-muitos
	Relacionamento muitos-para-um

3.1. Ferramenta para realização das comparações

Para a realização das comparações, utilizou-se a ferramenta de profiling presente no ambiente de desenvolvimento NetBeans, o qual foi utilizado para o desenvolvimento dos protótipos. A ferramenta permite a realização dos seguintes testes:

a) monitor de threads: permite que sejam monitoradas as threads da aplicação em execução;

b) CPU: Para a realização de testes de desempenho;

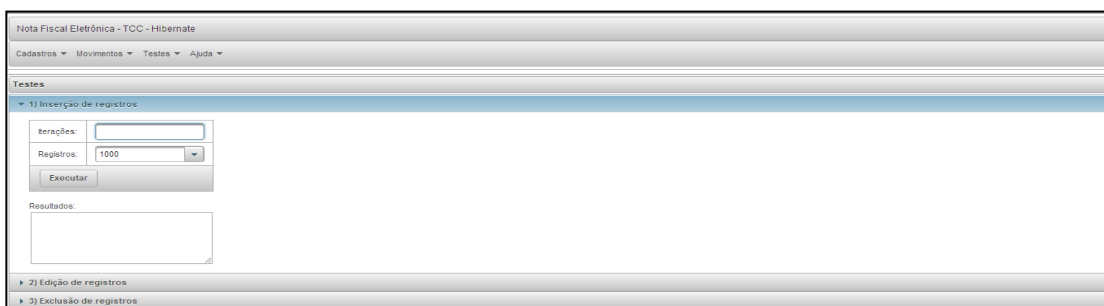
c) memória: Para acompanhar o consumo de memória da aplicação.

Foram utilizados somente os dois últimos testes supracitados, por estarem de acordo com o objetivo deste trabalho.

3.2 Realização das comparações

Para a realização das comparações, foram executadas e mensuradas operações de persistência e consultas com a entidade Nfe, que representa uma nota fiscal eletrônica. As operações de persistência realizadas consistiram na inserção, edição e remoção de mil e dez mil registros. A operação de consulta consistiu na realização de uma consulta com junções, agrupamento e funções de agregação, com dez mil notas fiscais eletrônicas (Nfe), sendo que cada uma possuía dez produtos (NfeProduto).

Figura 36 – Página de automação dos testes



Conforme a figura 1, para a automação dos testes, foi desenvolvida uma página, que pode ser acessada no menu principal, opção "Testes", item "Testes", onde serão exibidos os critérios disponíveis.

A função dos componentes exibidos nessa página é demonstrada na tabela 2.

Tabela 8 – Função dos componentes da página de teste

Componente	Função
Campo iterações	Permite informar o número de vezes que o teste irá executar.
Combobox Registros	Permite informar a quantidade de registros que serão adicionados, editados ou removidos. Estão disponíveis as opções 1000 e 10000.
Botão Executar	Executa o teste.

Para essa página, há um managed bean[15] chamado “TesteBean”, situado no pacote “bean”, que gerencia a preparação e realização dos testes. Foram os métodos desse objeto que foram monitorados pelo NetBeans Profiler, para medição do desempenho e consumo de memória.

Nos testes de desempenho, foi selecionado o teste “CPU”, e filtrado um método específico da classe TesteBean para ser monitorado, variando conforme o tipo de operação.

Já para os testes de memória, foi escolhido o teste “Memória”. Foi selecionada a opção “Registrar criação do objeto e coleta de lixo”, para monitorar o ciclo de vida dos objetos alocados.

Para realizar um teste mais igualitário entre os frameworks, não se utilizou o cache de segundo nível[16], por não haver a mesma biblioteca para essa função presente no EclipseLink e Hibernate, o que poderia ocasionar discrepâncias.

[15] Classes gerenciados pelo container de servidor de aplicação. Providenciam a lógica da aplicação, tendo ligação com os componentes da página. Em geral, cada página possui um *managed bean* (SUN, 2009, tradução nossa).
[16] O cache de segundo nível é um armazenamento local de entidades, gerenciado pelo provedor de persistência, para melhorar a performance da aplicação (SUN, 2009, tradução nossa).

3.3 Análise dos Tempos

Figura 37 – Resultados dos testes de inserção – Desempenho

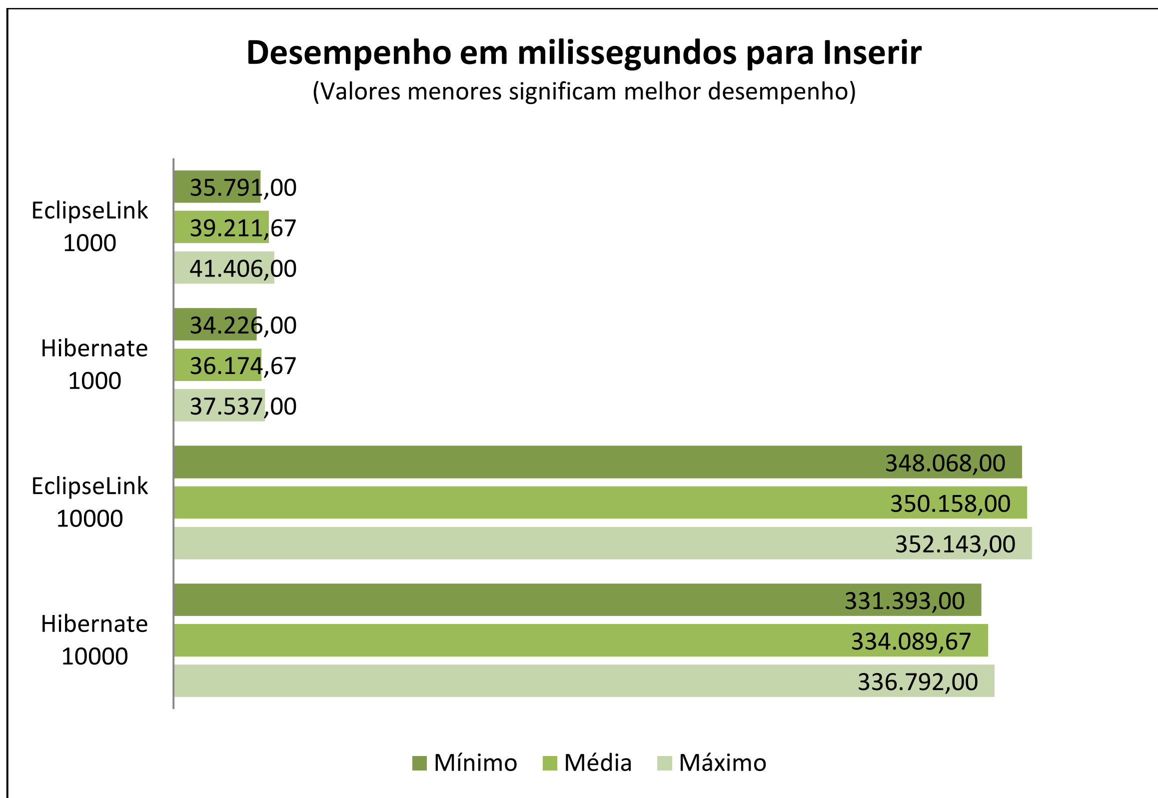


Figura 38 – Resultados dos testes de inserção – Memória

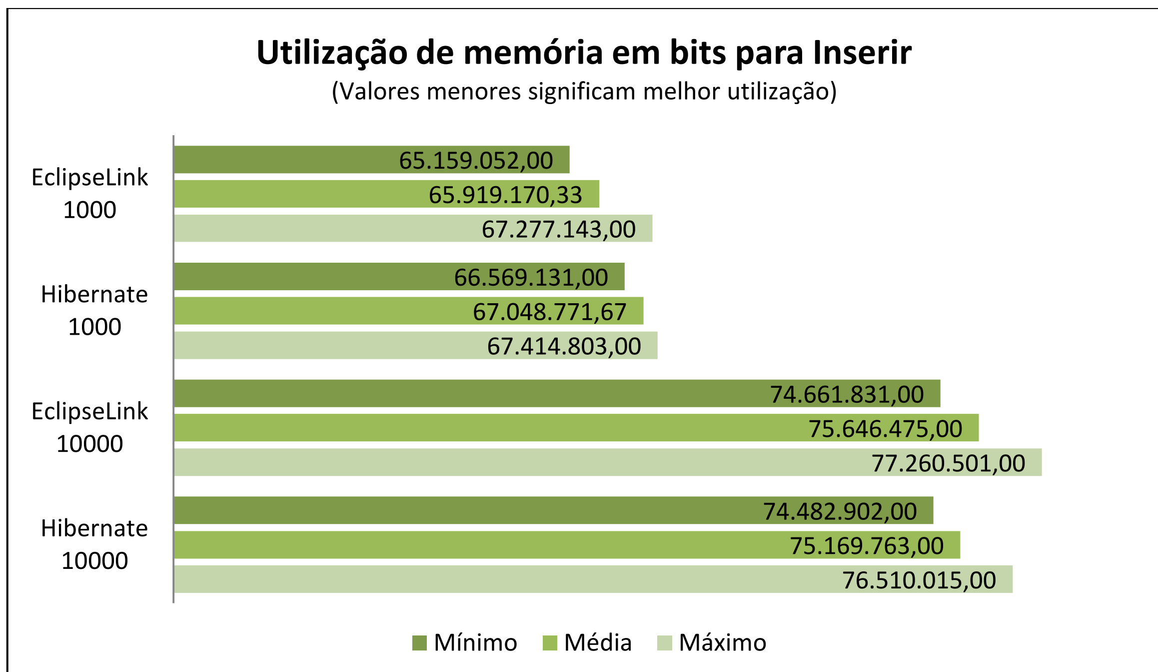


Figura 39 – Resultados dos testes de edição – Desempenho

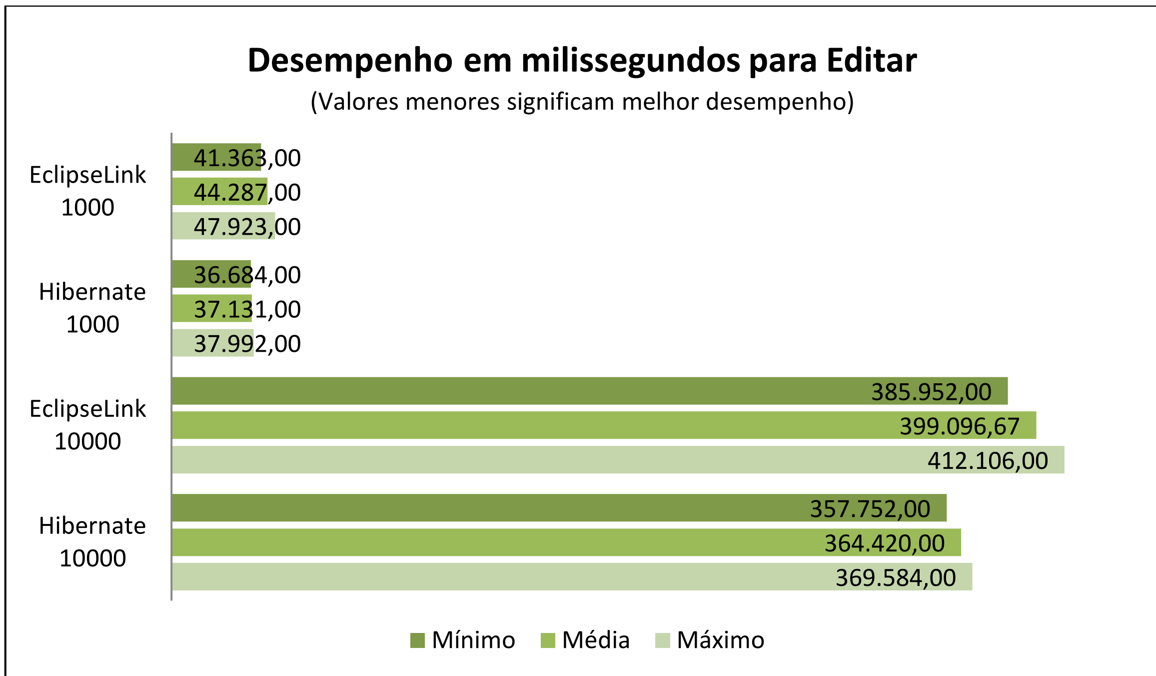


Figura 40 – Resultados dos testes de edição – Memória

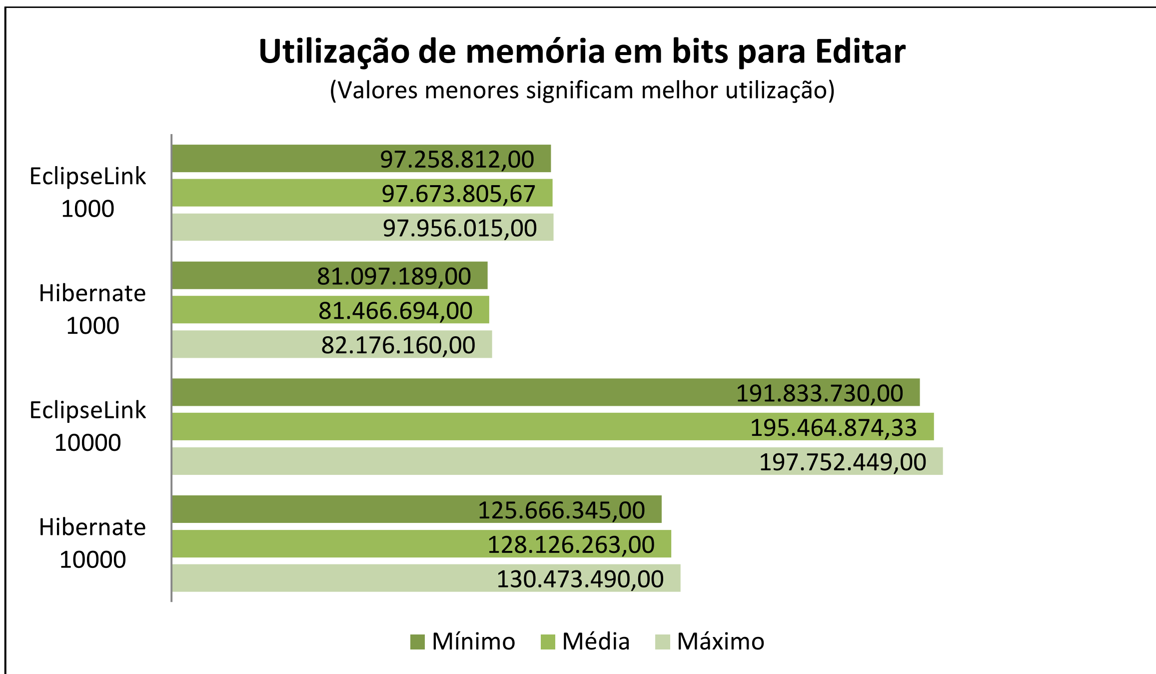


Figura 41 – Resultados dos testes de remoção – Desempenho

Figura 42 – Resultados dos testes de remoção – Memória

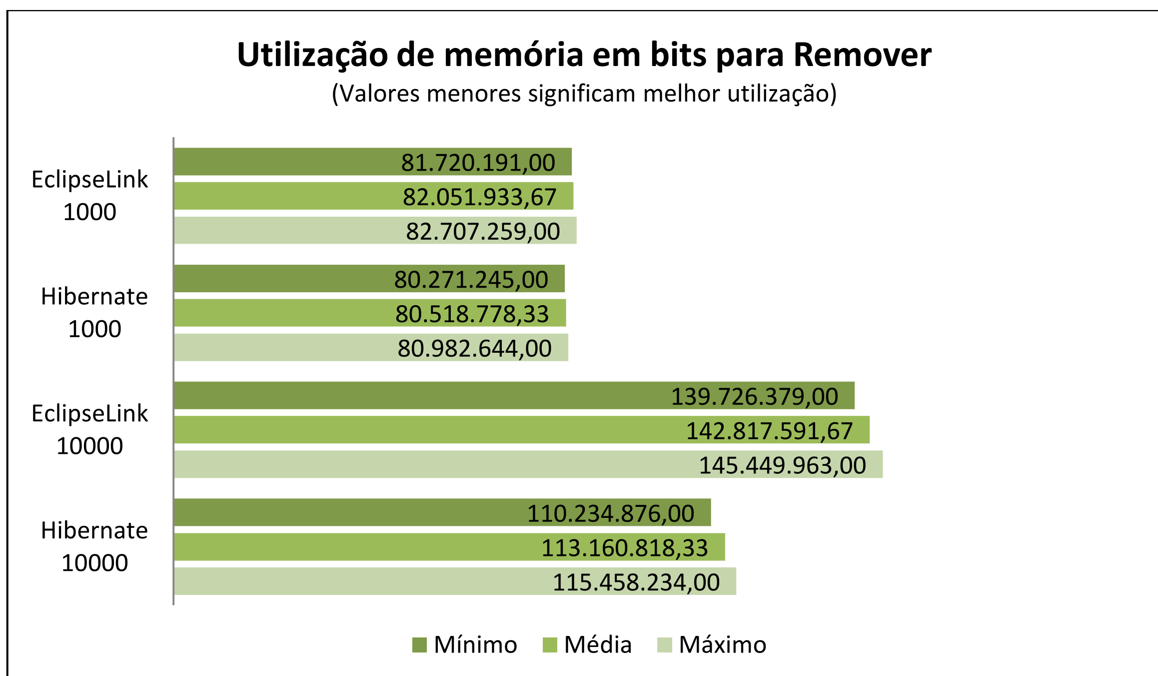


Figura 43 – Resultado do teste de consulta – Desempenho

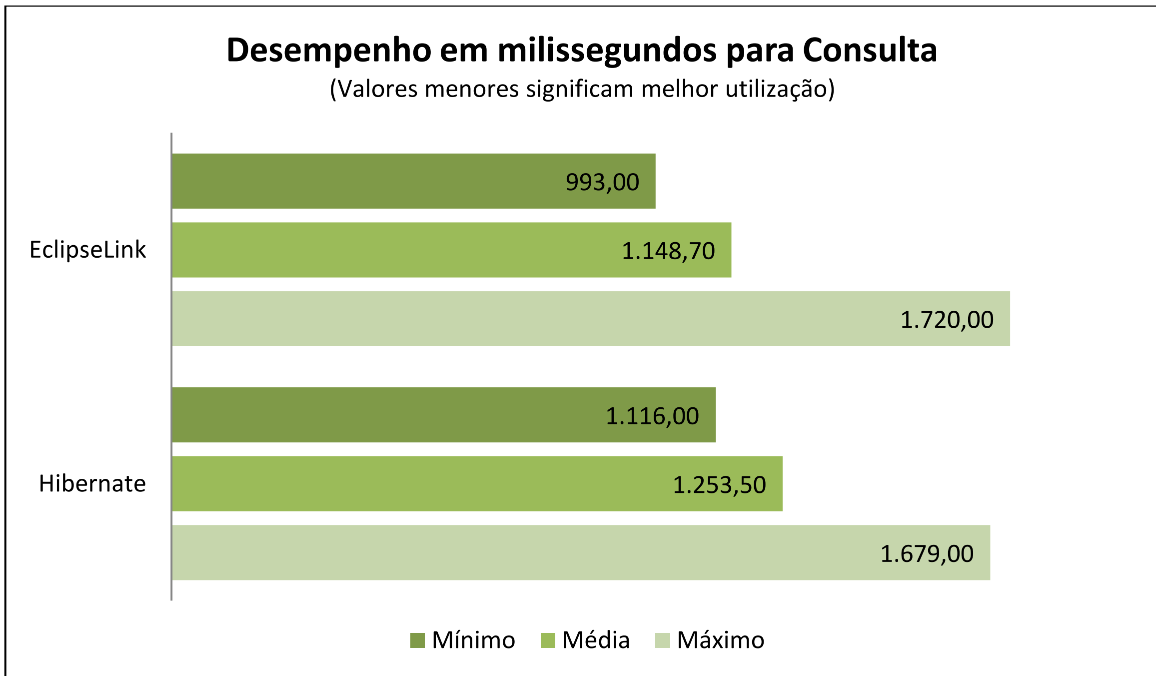
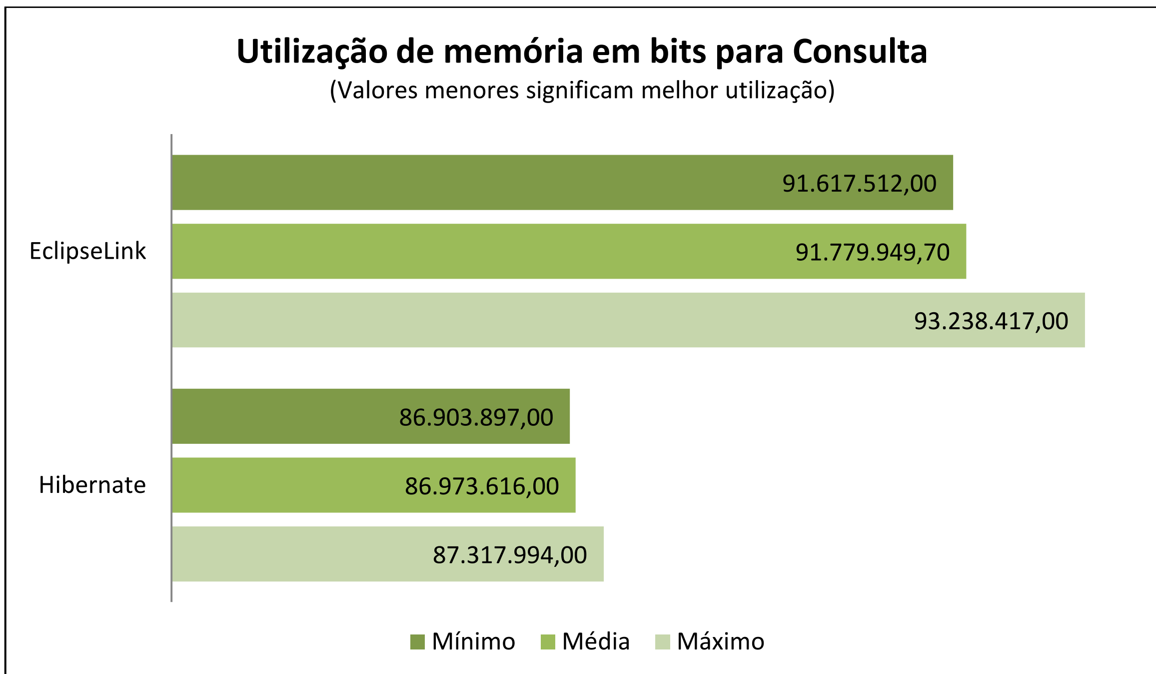


Figura 44 – Resultado do teste de consulta – Memória



Observou-se que, após a realização dos testes de desempenho, o Hibernate demonstrou-se mais performático nas operações de inclusão e edição, e foi superado nas operações de remoção e consulta pelo EclipseLink. Para os testes de memória, o Hibernate teve menor consumo, para todos os testes, exceto o teste de edição com mil registros, no qual o EclipseLink consumiu menos memória.

4. Conclusão

Nessa pesquisa, foi realizada a comparação de desempenho e consumo de memória entre os frameworks de Mapeamento Objeto-Relacional Hibernate e EclipseLink, em operações de inserção, edição, remoção e consulta de registros em dois protótipos de aplicação desenvolvidos.

Após as comparações, constatou-se que o Hibernate teve um desempenho superior do que o EclipseLink nos testes de inserções e edições, com mil e dez mil registros, perdendo nos testes de remoções com mil e dez mil registros, e no teste de consulta. Nos testes de memória, o Hibernate teve menor consumo em todos os testes, exceto para mil registros no teste de edições. Logo, o Hibernate mostrou-se mais vantajoso, possuindo um custo-benefício (consumo de memória/desempenho) superior ao do EclipseLink.

Apesar dos resultados obtidos, não se deve desmerecer o framework EclipseLink, que, mesmo sendo uma solução de persistência mais recente, possui uma grande comunidade que o mantém (Eclipse Foundation), e foi escolhida como a implementação de referência da JPA 2.0. Como o Hibernate, também existe a opção de cache de segundo nível, podendo acarretar melhoras perceptíveis no desempenho e consumo de memória para operações de persistência com o banco de dados.

As dificuldades do projeto consistiram no aprendizado e desenvolvimento da aplicação Web, com JSF e PrimeFaces, e utilizando Hibernate e EclipseLink para a persistência dos dados, bem como a utilização e análise dos dados com o NetBeans Profiler. Graças à ampla documentação disponível em livros, Web e auxílio de fóruns de usuários, essas dificuldades iniciais foram superadas com facilidade.

O problema descrito foi solucionado com sucesso, e os resultados documentados podem servir de auxílio no processo de tomada de decisão de um framework de Mapeamento Objeto-Relacional em Java.

5. Referências

AMBLER, Scott W.. The Object-Relational Impedance Mismatch. Disponível em: <http://www.agiledata.org/essays/impedanceMismatch.html>. Acesso em: 4 ago. 2012.

BAUER, Christian; KING, Gavin. Hibernate in Action. Greenwich: Manning, 2005. 431 p.

CORONEL, Carlos; MORRIS, Steven; ROB, Peter. Database Systems: Design, Implementation, and Management. 9. ed. Boston: Cengage Learning, 2011.

ELMASRI, Ramez; NAVATHE, Shamkant B..Sistema de Banco de Dados. 6. ed. São Paulo: Addison-wesley, 2011.

JBOSS. HIBERNATE - Relational Persistence for Idiomatic Java. Disponível em: <http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html/>. Acesso em: 14 ago. 2012.

ORACLE. The Java EE 6 Tutorial. Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/>. Acesso em: 14 ago. 2012.