**EDUARDO IVAN PICHLER**

SISTEMA DISTRIBUÍDO PARA DISTRIBUIDORAS DE ATACADO

**PATO BRANCO-PR**

**2006**

**EDUARDO IVAN PICHLER**

SISTEMA DISTRIBUÍDO PARA DISTRIBUIDORAS DE ATACADO

Monografia apresentada como requisito parcial para obtenção do título tecnólogo no curso de Tecnologia em Sistemas de Informação da Universidade Tecnológica Federal do Paraná , Campus Pato Branco.

Orientador: Prof. Msc. Ademir Roberto Freddo

**PATO BRANCO-PR**

**2006**

AGRADECIMENTOS

*Sobre tudo, ao meu pai e minha mãe por tudo o que representam para mim e mais do que ninguém, me amam e acreditam fielmente em mim. À minha irmã, meu ídolo, da qual sempre recebo apoio. Ao professor Omero que me ensinou os primeiros algoritmos, à professora Beatriz que foi meu anjo da guarda, ao Robison que me ajudou a escolher a tecnologia certa.*

*À professora Elizabete Paludo, que antes que tudo isso acontecesse, tornou possível meu ingresso à Universidade. E à aqueles professores que se preocuparam com minha formação, realmente meus amigos, me deram acesso ao conhecimento, me inspiraram e me ajudaram à fazer minhas escolhas e a tomar sempre a decisão certa.*

*Ao meu primo Eldoney que, talvez sem querer, fez a minha inclusão tecnológica me emprestando um video-game e aos meus amigos que foram fundamentais não atrapalhando em nada.*

"Aqueles que se enamoram somente da prática, sem cuidar da teoria, ou melhor, dizendo, da ciência, são como o piloto que embarca sem timão nem bússola. A prática deve alicerçar-se sobre uma boa teoria, à qual serve de guia a perspectiva; e em não entrando por esta porta, nunca se poderá fazer coisa perfeita nem na pintura, nem em nenhuma outra profissão".

SUMÁRIO

[SISTEMA DISTRIBUÍDO PARA DISTRIBUIDORAS DE ATACADO 1](#_Toc270320236)

[SISTEMA DISTRIBUÍDO PARA DISTRIBUIDORAS DE ATACADO i](#_Toc270320237)

[AGRADECIMENTOS iii](#_Toc270320238)

[SUMÁRIO vi](#_Toc270320239)

[RESUMO x](#_Toc270320240)

[1. INTRODUÇÃO 1](#_Toc270320241)

[1.1 Justificativa 1](#_Toc270320242)

[1.2 Objetivos 2](#_Toc270320243)

[1.2.1 Geral 2](#_Toc270320244)

[1.2.2 Específicos 2](#_Toc270320245)

[2. TECNOLOGIAS 2](#_Toc270320246)

[2.1 Java 2](#_Toc270320247)

[2.1.1 Características da linguagem 4](#_Toc270320248)

[2.2 JDeveloper 10g 9](#_Toc270320249)

[2.2.1 Desenvolvimento colaborativo em equipe 10](#_Toc270320250)

[2.2.2 Codificação e Navegação 10](#_Toc270320251)

[2.3 12](#_Toc270320252)

[2.3.1 Integração com tecnologias](#_Toc270320253) *[Open Source](#_Toc270320253)* [12](#_Toc270320253)

[2.3.2 Desenvolvimento de Extensões 12](#_Toc270320254)

[2.3.3 13](#_Toc270320255)

[2.3.4 Desenvolvimento J2EE 13](#_Toc270320256)

[2.3.5 Modelagem 13](#_Toc270320257)

[2.3.6 Desenvolvimento do banco de dados 13](#_Toc270320258)

[2.4 Web services 14](#_Toc270320259)

[2.4.1 O Protocolo SOAP 14](#_Toc270320260)

[2.4.2 WSDL, o arquivo descritor 14](#_Toc270320261)

[2.4.3 UDDI 15](#_Toc270320262)

[2.5 Eclipse BIRT 16](#_Toc270320263)

[2.5.1 O que é 16](#_Toc270320264)

[2.5.2 A anatomia de um relatório 17](#_Toc270320265)

[2.6 JUnit 18](#_Toc270320266)

[2.6.1 Testes unitários 18](#_Toc270320267)

[2.7 Log4j 18](#_Toc270320268)

[2.7.1 Estrutura 19](#_Toc270320269)

[2.7.2 Logger 19](#_Toc270320270)

[2.7.3 Appender 19](#_Toc270320271)

[2.7.4 Layout 19](#_Toc270320272)

[2.8 Oracle XE 10](#_Toc270320273)*[g](#_Toc270320273)* [19](#_Toc270320273)

[2.9 Hibernate 21](#_Toc270320274)

[2.9.1 Mapeamento Objeto-Relacional 21](#_Toc270320275)

[2.9.2 Por que usar um framework ORM 22](#_Toc270320276)

[2.10 Commons Configuration 24](#_Toc270320277)

[2.11 Design Pattern Singleton 24](#_Toc270320278)

[3. ANÁLISE UML 26](#_Toc270320279)

[3.1 Diagrama de classes 27](#_Toc270320280)

[3.1.1 Cadastros 27](#_Toc270320281)

[3.2 Movimentos de mercadorias 30](#_Toc270320282)

[3.3 Contas 31](#_Toc270320283)

[3.4 Usuários 32](#_Toc270320284)

[4. UTILIZAÇÃO DAS TECNOLOGIAS NO DESENVOLVIMENTO DO SISTEMA 34](#_Toc270320285)

[4.1 Instalação e configuração das ferramentas 34](#_Toc270320286)

[4.2 Banco de dados Oracle XE 34](#_Toc270320287)

[4.3 Hibernate, e mecanismo de persistência 36](#_Toc270320288)

[4.4 O Design Pattern Singleton 41](#_Toc270320289)

[4.5 Troca de mensagens e separação de classes distintas 43](#_Toc270320290)

[4.6 47](#_Toc270320291)

[4.7 Representando dados tabulares 47](#_Toc270320292)

[4.8 Inserção, edição e validação dos dados 50](#_Toc270320293)

[4.9 Criação de logs 53](#_Toc270320294)

[4.10 Testes unitários com JUnit 55](#_Toc270320295)

[4.11 58](#_Toc270320296)

[4.12 Eclipse BIRT 58](#_Toc270320297)

[4.13 58](#_Toc270320298)

[5. SISTEMA 60](#_Toc270320299)

[5.1 Cadastro de Pessoas 60](#_Toc270320300)

[5.2 Cadastro de produtos 63](#_Toc270320301)

[5.3 Vendas 64](#_Toc270320302)

[5.4 Entrada de estoque 65](#_Toc270320303)

[5.5 Gerenciamento de usuários 65](#_Toc270320304)

[5.6 Relatórios 66](#_Toc270320305)

[5.7 Web services 68](#_Toc270320306)

[6. CONCLUSÃO 70](#_Toc270320307)

[7. Bibliografia 72](#_Toc270320308)

Nenhuma entrada de índice de ilustrações foi encontrada.

Lista de ilustrações, deve ser intitulada de acordo com o tipo de ilustração que contém: LISTA DE TABELAS, LISTA DE QUADROS, LISTA DE FIGURAS. Criar listas específicas desde que haja no mínimo dois itens para o respectivo tipo. Se específicas e extensas devem ser colocadas em folhas distintas

RESUMO

Este trabalho apresenta uma solução para uma empresa do ramo de distribuição de mercadorias. Através do software será possível gerenciar clientes, produtos, fornecedores, estoque, caixa(contas à pagar e receber), extrair relatórios, usuários do sistema e vendas. Para o desenvolvimento do software foram utilizadas as seguintes tecnologias: Oracle XE 10*g*, Java, Web services e Hibernate.

# INTRODUÇÃO

Todas as empresas têm, ou deveriam ter objetivos de crescer e expandirem seus negócios, para tanto, enquanto ela caminha em direção ao crescimento aumenta-se também a complexidade do negócio, quantidade de pessoas envolvidas, transações efetuadas e fluxo de mercadorias, até chegar à um ponto onde a tecnologia é indispensável para o controle do sistema da empresa.

Para tomar decisões corretas, o administrador necessita ter em mãos relatórios consistentes e gráficos que o auxiliem com segurança ajudando em seu trabalho na escolha da estratégia correta.

Para a expansão dos negócios, as empresas do tipo distribuidoras, possuem vendedores que vão em busca de clientes para negociar os seus produtos. O software interno da empresa precisa estar preparado para eventuais implementações que melhorem-no e o torne-o mais integrado sem jogar fora todos os sistemas que já existem funcionando na empresa.

## Justificativa

Uma empresa com grande quantidade de transações torna-se demasiadamente complexa para crescer com velocidade caso não controle suas informações de forma eficaz. Um sistema que controle o fluxo da informação entre departamentos deve armazenar históricos de movimentações em suas bases de dados e deve gerar relatórios com informações eficazes e personalizadas. Assim, auxiliaria os administradores da empresa à tomada de decisões coerentes, pois a qualquer momento poderiam ter em suas mão os relatórios e gráficos. Custos seriam reduzidos e recursos humanos poderiam ser remanejados. A empresa poderia se concentrar em seu principal enfoque, que é vender e oferecer qualidade, pois a tecnologia estaria integrando todos seus funcionários e cuidando dos problemas de comunicação e do negócio. Com o sistema trabalhando de forma integrada, tanto os vendedores como administradores da empresa poderiam estar cientes de seu desempenho através de informações úteis que seriam gerados através de relatórios.

## Objetivos

### Geral

Desenvolver um sistema para distribuidoras que atenda grande parte de suas necessidades comerciais utilizando a tecnologia Java, e que possa ser facilmente integrado com o uso de métodos disponibilizados através de *Web services*.

### Específicos

Para o desenvolvimento do sistema será necessário os seguintes objetivos específicos:

* Aprender Java, Web services, Oracle XE, Hibernate, UML, Eclipse BIRT e JUnit;
* Fazer o gerenciamento de produtos, clientes, vendedores, usuários, etc;
* Gerar relatórios e gráficos;
* Efetuar vendas;
* Fazer o controlar as movimentações de produtos;
* Controlar entrada de mercadorias;

Controlar o fluxo de caixa(contas à receber e pagar);

# TECNOLOGIAS

As seguintes tecnologias foram utilizadas para o desenvolvimento do trabalho: JDeveloper 10*g*, Java, *Web* *services* , Eclipse BIRT, JUnit, Log4j, Hibernate, Oracle XE 10*g*, *Commons* *Configuration* e Log4j. Ainda foi necessário o uso do *Design* *Patter* *Singleton*.

## Java

A tecnologia Java foi criada como uma ferramenta de programação em computação, parte de um pequeno trabalho anônimo e secreto chamado *"the Green Project"* da Sun Microsystems em 1991.

A equipe *"Green Team"*, um grupo secreto formado por 13 pessoas e liderado por James Gosling, trancou-se em um escritório anônimo na *Sand Hill Road* em *Menlo Park* e trabalhou dia e noite por 18 meses, isolado de toda a comunicação com a Sun.

Eles estavam tentando antever e planejar a "nova onda" na computação. Sua primeira conclusão foi que ao menos uma tendência significativa seria a convergência de computadores e de dispositivos controlados digitalmente.

O resultado foi uma linguagem de programação não atrelada a dispositivos, apelidada de "Oak".

Para demonstrar como essa nova linguagem poderia impulsionar o futuro dos dispositivos digitais, a equipe *Green Team* desenvolveu um controlador portátil para sistemas de entretenimento doméstico voltado para o setor de televisão digital. A idéia estava muito à frente de seu tempo e o setor de TV digital não estava preparado para o incrível avanço oferecido pela tecnologia Java.

Mas a Internet estava pronta para essa tecnologia e, bem a tempo para sua apresentação pública em 1995, a equipe pôde anunciar que o navegador *Netscape Navigator* passaria a incorporar a tecnologia Java.

Em 2005, a plataforma Java já tinha atraído mais de 4 milhões de desenvolvedores de software, é usada em todos os principais setores no mundo inteiro e está presente em uma ampla gama de dispositivos, computadores e redes de todas as tecnologias de programação.

Sua versatilidade, eficiência, portabilidade de plataforma e segurança fazem dela a tecnologia ideal para a computação em rede, de modo que, hoje, a tecnologia Java está presente em mais de dois bilhões e meio de dispositivos:

* mais de 700 milhões de *Pcs*;
* 708 milhões de telefones celulares e outros dispositivos portáteis;
* 1 bilhão de *smart cards*;
* além de *set-top boxes*, impressoras, *webcams*, jogos, sistemas de navegação para automóveis, terminais lotéricos, dispositivos médicos, estações de pagamento de estacionamento, etc;

Atualmente, você pode encontrar a tecnologia Java em redes e dispositivos que vão desde a Internet e supercomputadores científicos a *laptops* e telefones celulares, de simuladores de mercado da *Wall Street* a dispositivos para jogos e cartões de crédito -- simplesmente em todo lugar.

A linguagem de programação Java foi completamente refinada, ampliada, testada e experimentada por uma comunidade ativa de mais de quatro milhões de desenvolvedores de software.

Madura, extremamente robusta e surpreendentemente versátil, a tecnologia Java adquiriu um valor inestimável e permite aos desenvolvedores:

* Gravar um software em uma plataforma e executá-lo em praticamente qualquer outra
* Criar programas para execução em navegadores e serviços da *Web*;
* Desenvolver aplicativos no lado do servidor usados tanto em fóruns, lojas e pesquisas *on-line*, como no processamento de formulários HTML, e muito mais;
* Combinar aplicativos ou serviços com tecnologia Java para criar aplicativos ou serviços altamente personalizados;
* Escrever aplicativos potentes e eficazes para telefones celulares, processadores remotos, produtos de consumo de baixo custo e praticamente qualquer dispositivo com tecnologia digital;[1]



Figura 1: Logotipo da tecnologia Java

### Características da linguagem

#### Orientação a Objetos

Java é uma linguagem orientada a objetos e, com isso, não é possível desenvolver nenhum programa sem seguir tal paradigma. Um sistema orientado a objetos é composto por um conjunto de classes e objetos bem definidos que interagem entre si, de modo a gerar o resultado esperado.

A Orientação à Objetos modela o mundo real com classes e instâncias. Cada classe é a estrutura de uma variável, ou seja, um tipo de dado. Nela, são declarados atributos e métodos que poderão ser executados ou acessados nas instâncias da mesma classe. As classes possuem uma função muito importante na modelagem orientada a objetos, elas dividem o problema, modularizam a aplicação e baixam o nível de acoplamento do software.

Variáveis de uma classe são chamadas de instâncias de classe. Por exemplo: Se existe uma classe Pessoa existe a instância (variável) pessoa que é do tipo Pessoa. Um Objeto, ou instância, é uma entidade cujas informações podem incluir desde suas características até suas atividades, ou seja, uma abstração de um domínio de um problema.

Abaixo há uma série de definições sobre a arquitetura de orientação a objetos em Java.

* Pacote: Conjunto de classes e demais arquivos que possuem interesses comuns ou atuam com dependências entre si. Fisicamente são pastas do sistema operacional.
* Instância, objeto: Uma variável do tipo de uma classe.
* Construtor: Responsável por iniciar a criação e inicialização de uma instância de classe.
* Método: Funções em uma classe.
* Modificador de acesso: Descreve que outras classes podem ter acesso a classe que está se criando. Também é usado para indicar que uma classe pode ser acessada de fora de seu pacote.
* Hierarquia de classes: Um grupo de classes que estão relacionadas por herança.
* Superclasse: É a classe que é estendida por uma determinada classe.
* Subclasse: É a classe que estende determinada classe.
* Classe base: A classe de determinada hierarquia que é uma superclasse de todas as outras classes. A classe ” pai de todas ”.

#### Métodos

Um método em Java é uma sub-rotina semelhante as funções de C e Pascal. Os métodos têm um retorno e podem conter parâmetros.

Ao declarar um método é possível adicionar modificadores. O modificador estático declara que o método terá um determinado tipo de comportamento dentro do programa orientado a objetos. Na declaração de um método os modificadores precedem o tipo de retorno, que precede o nome do método e a lista de parâmetros. O corpo do método é colocado entre chaves.

Buscando o exemplo da classe pessoa:

public void testeMetodo (double valor) {

double dobroValor = valor + valor;

}

O corpo do método pode conter declarações de variáveis e comandos. As variáveis não ficam restritas a declaração somente antes dos comandos, como acontece em C ou em um local pré-determinado como em Pascal, ao contrário, podem ser declaradas em qualquer lugar. O tipo de retorno de um método pode ser primitivo ou de referência, ou como no método main, pode ser void, ou seja, não retorna nada.

Em Java todos os argumentos são passados por valor. Não existe passagem por referência. Se for variável primitiva o valor da variável é passada no parâmetro e pode ser alterada a vontade dentro da função que nada sofrerá a variável externa. Com objetos o que muda é que o valor passado no parâmetro da função é a referência para o objeto, ou seja, se um objeto for alterado dentro da função, o objeto externo também será alterado, exceto nas instruções objeto = null;

#### Interfaces

Interfaces foram concebidas para criar um modelo de implementação aumentando o baixo acoplamento entre determinadas partes de um software. Uma interface não pode possuir atributos de instância e nem métodos com implementação, mas pode possuir atributos de estáticos (de classe) e cabeçalhos de métodos que deverão ser desenvolvidos nas classes que implementarão a interface.

Muitas vezes as interfaces representam ações ou papéis para as classes. Um exemplo comum de interface é a *Serializable* que se encontra dentro do pacote java.io e é muito utilizada em aplicações corporativas. Quando uma classe implementa *Serializable* ela não precisa implementar nenhum método definido na interface, mas com esta ação o programador indica ao java que a classe pode ser serializada, transformada em um conjunto de bits para serem armazenados ou transmitidos. A serialização de uma instância armazena todos os seus dados e consegue criar um objeto semelhante na desserialização.

Para implementar uma interface, usa-se a palavra-chave *implements* seguida do nome da interface, ou interfaces, visto que não há limite para quantidade de classes em implementação.

#### Classes Abstratas

Uma classe abstrata é quase uma interface, com a exceção que ela pode conter implementações. Uma classe abstrata é usada para manter alguma programação nas classes ” Pai ”, porém não pode ser instanciada. Para ser utilizada como instância, uma classe abstrata deve ser estendida por uma classe não abstrata.

Para criar uma extensão de qualquer classe, usa-se a palavra-chave *extends*. O Java não permite a herança múltipla, como em C, portanto não é possível estender de duas ou mais classes.

#### Sobrecarga de métodos

O Java permite que você tenha métodos com o mesmo nome, mas com assinaturas diferentes O interpretador determinará qual método deve ser invocado pelo tipo de parâmetro passado. Os trecho abaixo é válido para uma compilação Java.

public void print (int i) {...}

public void print (float f) {...}

public void print (String s) {...}

Quando é escrito o código para chamar um desses métodos, a chamada deverá coincidir com tipos de dados da lista de parâmetros de algum dos métodos.

Diferente da sobrecarga, a sobrescrita acontece quando um método existe em uma classe ” pai ” e é reescrito na classe ” filha ” para alterar o comportamento. A assinatura do método deve ser igual na classe ” pai ” e na classe ” filha ”. Como um exemplo abaixo está o código desenvolvido na seção this e super.

public class Pai {

public int lançar (int numero) {

return 6 \ % numero;

}

}

public class Filho extends Pai {

public int lançar (int numero) {

if (numero = = 0) {

return this.numero;

}

super.lançar (numero);

}

}

O método public int lançar (int numero) existe na classe filha e na pai, o primeiro a ser invocado é o Filho, e, via a palavra chave super, o interpretador chamará o método pai. Caso não houver o comando super.lançar (numero); o método pai não seria invocado.

#### Tratamento de Exceções

As exceções em Java são classes que herdam de *java.lang.Throwable*. Basicamente existem três tipos de exceções que são herdadas de *Throwable*:

* *Error*: Exceção grave que normalmente não pode ser tratado, como falhas físicas e condições anormais de funcionamento.
* *Exception*: Exceções genéricas que devem ser tratadas utilizando declarações *throws* ou criar o tratamento para a exceção com *try catch.*
* *RuntimeException*: São exceções que podem ser lançadas durante a execução da JVM. Podem ser tratadas, porém não precisam ser tratadas com *throws* ou com *try* *catch*

Em Java pode-se escolher onde se quer tratar a exceção. Se for para tratar no mesmo método que ela ocorreu, usa-se a instrução:

try {

}catch () {

} catch () {

} finally {

}

A instrução try, como seu nome já fala, tenta executar um bloco de comandos. Ao ocorrer qualquer erro a JVM passará o controle para um dos *catch*, o que mais se apropriar para a exceção lançada.

Se a JVM não encontrar um *catch* apropriado e se houver uma cláusula *finally* ela é executada, se não houver um *finally* o Java repassa a exceção para o método que chamou este, e assim continua até que a exceção seja simplesmente lançada na tela para o usuário. [2]

## JDeveloper 10g

Oracle JDeveloper 10*g* é um ambiente completo e integrado de desenvolvimento de aplicações baseados em padrões Orientados à Serviço (SOA). O JDeveloper suporta os últimos padrões Java (J2SE 5.0, J2EE 1.4, EJB 3.0), e o desenvolvimento visual de JavaServer Faces.

O JDeveloper tornou-se gratuito em 2005. Entre as características da ferramenta, pode ser citado:

### Desenvolvimento colaborativo em equipe

* Projetos dinâmicos: Projetos são sempre dinâmicos, o JDeveloper guarda guarda uma lista de diretórios contento fontes e não uma lista de fontes apenas.
* Área de trabalho: permite que o navegador seja configurado para mostrar os subconjuntos de arquivos do projeto, muito útil em grandes projetos.
* Propriedades e preferências de usuários compartilháveis: É possível ter controle sobre propriedades do projeto para que as tornem compartilháveis, permitindo que usuários possam sobrescrever configurações sem causar impacto significativo no resto da equipe.
* Lista global de exclusão: Permite que arquivos com certo tipo de extensão não sejam considerados parte do projeto.
* Gerenciamento de bibliotecas: Bibliotecas podem ser guardadas em qualquer lugar do disco.
* Filtros de histórico customizáveis: O navegador dos fontes que estão sob controle de versão suportam customização da navegação através de filtros.

### Codificação e Navegação

#### Refactoring

* O poderoso framework de *refactoring* permite que antes de ser feito a modificação, um *preview* de tudo que irá acontecer seja visualizado, com os fontes que serão atualizados. É dado a escolha de prosseguir ou não. Pode também ser desfeito qualquer modificação se for necessário.
* O mecanismo de *refactoring* é integrado com o software de controle de versões. Se uma classe que ainda é referenciada for renomeada, o JDeveloper fará o *chekout* do arquivo para manterem sincronizados.

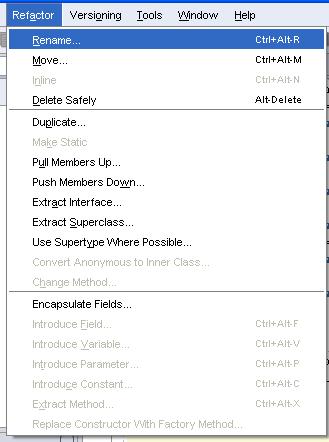


Figura 3: Menu de refactoring o Oracle JDeveloper

#### Java code navigation

* Find Usages: complementa o refactoring permitindo que seja localizado uma string no meio de todo um projeto. Os resultados são visualizados na janela de Logs. JDeveloper pode procurar o uso de classes, métodos, campos, variáveis e parâmetros.
* Auto-completar: JDeveloper possui um mecanismo inteligente de aceleradores e da característica auto-completar do código.

## 

Figura 4: Recurso auto-completar do Oracle JDeveloper

### Integração com tecnologias *Open Source*

JDeveloper possui um suporte completo à tecnologias open source tais como:

* Ant: ferramenta de fazer *builds.*
* JUnit: testes unitários em Java.
* CVS e SVN: controle de versões e desenvolvimento colaborativo.
* Struts: *Framework* para desenvolver aplicações *web* em Java.
* XDoclet: geração de código através de programação orientada à atributos.

### Desenvolvimento de Extensões

#### Prévia da especificação JSR-198

O JDeveloper tem uma prévia da implementação da JSR-198 ( Uma API padrão para o desenvolvimento de extensões para ambientes integrados). Essa especificação está ainda em desenvolvimento, e o JDeveloper será uma referência à essa especificação quando for finalizada. A JSR-198 permite que extensões de terceiros usem para o desenvolvimento uma API padrão para integrar funcionalidades com qualquer IDE que suporte a JSR-198.

### 

### Desenvolvimento J2EE

O JDeveloper suporta:

* Enterprise JavaBeans 3.
* *Web* *services (*possui um editor visual de WSDL).
* Jsp/Servlets.
* JavaServer Faces.
* XML.

### Modelagem

O JDeveloper oferece suporte aos diagramas de Seqüência, Atividades, Casos de Uso e de Classes. Suporta exportação dos mesmos para arquivos XMI.

### Desenvolvimento do banco de dados

No desenvolvimento do banco de dados, o JDeveloper suporta:

* Bancos de dados de terceiros: suporta importar e exportar modelos de objetos de MS SQL Server, Sybase, IBM DB2, IBM Informix e MySQL.
* Construtor de *views*: suporta a criação e edição de *views*, tanto com *wizards* como no diagrama do banco, e diretamente no código.
* *Scripts* automáticos: scripts *ALTER SQL* são automaticamente gerados quando solicitados.
* Modelagem do banco: suporta criação, geração, captura e visualização dos seguintes objetos( em adição à tabelas, colunas e chaves): *Views, Sequences, Synonyms, Indexes, PLSQL Packages, Procedures, Function, User-Defined Types, XML, Spatial, e Media System-Defined Types.*
* Comparação gráfica: oferece um ambiente visual de comparação entre objetos do banco de dados com esquema *on*-*line*, permitindo modificações seletivas para a reconciliação.Edição dos dados: A aba Data no editor de tabelas permite editar registros no banco de dados e fazer *commit* *ou rollback* dos mesmos.[3]

## Web services

*Web service* é um sistema de software designado para suportar interoperabilidade entre máquinas interagindo sobre uma *rede.* Ele *tem uma* interface que é descrita em um arquivo no formato WSDL. Outros sistemas interagem com o webservice em maneira prescrita pela sua interface através de mensagens. Essas mensagens podem ser tipicamente capturadas usando HTTP. Softwares escritos em várias linguagens e rodando sobre várias plataformas podem usar *Web* *services* para processar dados através de redes de computadores como a Internet. Esta interoperabilidade é alcançada através do uso de padrões abertos como o SOAP[4]. Um grupo de *Web* *services* interagindo juntos define uma aplicação em uma arquitetura orientada a serviços.

*Web* *services* usam XML que podem descrever todo e qualquer dado em uma maneira verdadeiramente independente de plataforma, tornando as aplicações de baixo acoplamento. Em nível técnico, *Web* *services* podem lidar com dados de forma muito mais fácil e permite os softwares comunicarem mais livremente.[5]

Os XMLs de serviços de um webservice trafegam através de HTTP e podem seguir padrões abertos, como o SOAP.

### O Protocolo SOAP

SOAP é um protocolo leve para troca de informações descentralizadas, em ambiente distribuindo Ele é baseado em XML e consiste em três partes: um envelope que define um framework para descrever qual mensagem e como processá-la; um conjunto de regras de codificação para expressar os tipos de dados de uma aplicação definida; e uma convenção para representar chamadas remotas de procedimentos e respostas. [6]

### WSDL, o arquivo descritor

*Web* *Service* *Description* *Language* (WSDL) é um documento XML que contém informações sobre a interface, semântica e administrativa de uma chamada para um webservice.

Uma vez desenvolvido o *Web* *service*, deve ser publicado seu descritor um um repositório UDDI (Universal *Description*, *Discovery* *and* *Integration*) para que usuários em potencial o usem. Quando alguém quiser usar o *Web* *service*, ele verifica o WSDL e descobre como acessá-lo.

#### O padrão WSDL

O padrão WSDL é construído pelo W3C (*World* *Wide* Wieb *Consortium*). O W3C o define como “um padrão de formatação XML para descrever serviços de uma rede ou um conjunto de pontos operantes em mensagens contendo informações, orientadas a documentos ou orientadas à procedimentos. Essas operações e mensagens são descritas de forma abstrata, então colocadas em um protocolo concreto de rede em formato de mensagem para definir um ponto. WSDL é extensível para permitir a descrição de pontos e suas mensagens de quais formatos ou protocolos de rede são usados para a comunicação.”[7]

### UDDI

O protocolo de Descrição Universal, Descoberta e Integração(UDDI) é uma das especificações mais requisitadas para o sucesso dos *Web* *services*. UDDI cria um padrão interoperável que habilita companhias e aplicações para que de forma rápida, fácil, e dinâmica procurem e usem *Web* *services* através da internet. UDDI também permite registros operacionais para serem mantido por diferentes propostas em diferentes contextos.

O projeto UDDI usa padrões do W3C e do *Engineering* *Task* *Force* (IETF) como os protocolos XML, HTTP e DNS. Então, características multi-plataformas são garantidas pela adoção de propostas como o SOAP, conhecido como especificação de protocolo de mensagens encontrado no site do W3C.

##### Benefícios do UDDI

Negócios de todos os tamanhos podem se beneficiar do UDDI, devido as compreensivas especificações que resolvem problemas que limitam o crescimento e a sinergia de comércio *Bussines*-*to*-*Bussines* e *Web ser*vices.

O projeto UDDI não é específico à uma indústria. Qualquer indústria, no mundo, oferecendo produtos e serviços podem se beneficiar dessa iniciativa.

Problemas que a especificação pode ajudar a resolver:

* tornar possível organizações descobrir rapidamente milhões de negócios *on­line*.
* definindo como habilitar o comércio para que seja conduzido uma vez que o negócio interessante é descoberto
  + - Benefícios imediatos que o projeto UDDI traz inclui:
* descoberta de novos clientes
* expansão das ofertas
* expansão do mercado
* remove barreiras necessárias para permitir uma rápida participação no economia global da internet
* descreve seus serviços e processos de negócio programaticamente em um simples, aberto, e seguro ambiente.
* usando um conjunto de protocolos que habilitam negócios invocarem serviços através da internet para agregar valor aos clientes.[8]

## Eclipse BIRT

### O que é

*Bussines Inteligence and Reporting Tools*(*BIRT*), é a proposta do projeto BIRT da fundação Eclipse. O projeto BIRT é um open source, sistema de relatórios baseado no eclipse, que interage com outras aplicações, especialmente aquelas baseadas em Java e J2EE , produzindo relatórios tanto para web quando em PDF.

BIRT fornece funcionalidades de *engine* de relatórios como *layout*, acesso à dados e *scritpting*.

BIRT tem dois componentes chaves: o *report* *designer* baseado no Eclipse, e um componente de *runtime* que pode ser adicionado no servidor de aplicação. BIRT também oferece uma *engine* de gráficos que permite que gráficos sejam adicionados nas aplicações.

Com BIRT, uma rica variedade de relatórios podem ser incluídos na aplicação:

* Listas: Os mais simples relatórios são as listas de dados. Quando começam a tornar-se longas, pode ser agrupadas para organizar junto os dados relatados. Se o dado dinâmico for numérico, pode ser facilmente adicionado totais, médias entre outras equações.
* Gráficos: Dados numéricos são muito mais fáceis de entender quando apresentados em forma de gráficos. BIRT proporciona gráficos de pizza, linhas e barras, entre outros. Gráficos BIRT podem ser redenizados em SVG e suportar eventos para permitir interação com o usuário.
* Cartas & Documentos: Notas, cartas, e outros documentos de textos são facilmente criados com BIRT. Documentos podem incluir textos, formatação e *design*, listas, gráficos e mais.
* Relatórios compostos: Muitos relatórios precisam ser combinados em um simples documento. Com BIRT isso é possível.

### A anatomia de um relatório

Um relatório BIRT consiste de quatro partes chave: dados, conversores de dados, lógica de negócio e apresentação.

* Dados – Bancos de dados, *Web* *services* , objetos Java, todos podem fornecer dados ao BIRT. Desde a versão 2.0 suporta JDBC e XML, como também possui suporte para uso de código para pegar dados de outros *data sources*. O framework do BIRT, *Open* *Data* *Acess* (*ODA*) , permite que seja construído relatórios e interfaces a partir de qualquer dado tabular. Em resumo, um simples relatório pode incluir dados de um número variável de fontes de dados.
* Conversores de dados – Relatórios apresentam dados ordenados, somados, filtrados e agrupados para fornecer o que o usuário precisa. Enquanto bancos de dados podem fazer alguns destes trabalhos, o BIRT faz tanto para o banco quanto para outras fontes de dados, como objetos Java. O BIRT permite operações sofisticadas como agrupamento de somas, porcentagens, entre outros.
* Lógica de negócio – Dados do mundo real são raramente estruturados exatamente como é requerido num relatório. Muitos relatórios precisam de uma lógica específica de negócio para converter dados comuns em informações úteis ao usuário. Se a lógica é simplesmente para o relatório, pode ser escrita através do suporte à Javascript do BIRT. Se a aplicação já contém a lógica pronta, pode ser chamado em no código java existente.
* Apresentação - Uma vez que os dados estão prontos, existem muitas maneiras de serem representados ao usuário. Tabelas, gráficos, textos entre outros. Um simples conjunto de dados pode aparecer em múltiplas maneiras, e um simples relatório pode representar dados de muitas fontes de conjuntos de dados diferentes.[9]

## JUnit

JUnit é um framework Java para fazer testes unitários. Criado originalmente por Kent Beck e Erich Gamma, é o framework de maior sucesso da família xUnit. JUnit tem gerado seu próprio ecossistema de extensões JUnit

JUnit tem sido portado para outras linguagens, incluindo C# (NUnit), Python (PyUnit), Fortran (fUnit), and C++ (CppUnit). A família de frameworks de testes unitários é referenciada coletivamente como xUnit. Apesar de JUnit estar muito associado com *Extreme* *Programming*, ele precede essa metodologia.

### Testes unitários

Teste unitário é um procedimento usado para validar uma pedaço de código de programação. O procedimento é escrever *test cases* para todas as funções e métodos Assim em eventuais alterações, rodando os testes novamente, erros podem ser facilmente identificados e corrigidos. O ideal é que cada *test case* seja separado dos outros.

A grande vantagem de testes unitários é isolar cada pedaço do programa para que possa ser testado. Permitem que o programador faça *refactoring* de seu código facilmente, sem importar quanto tempo faz que o código não é alterado. Testes unitários ainda os encorajam à mudanças e melhorias, uma vez que rodando os testes criados após as alterações, os possíveis *bugs* são facilmente localizados.[10]

## Log4j

O Log4j é uma ferramenta simples para a exibição de *logs*, tem sua configuração baseada em arquivos, facilitando a alteração de suas características. O log4j pode também proporcionar ganho de performance à aplicação (dependendo da estrutura utilizada). Pode-se também desabilitar a emissão de *logs* quando aplicação não estiver de fato em produção.

### Estrutura

O Log4j tem sua estrutura composta por 3 componentes básicos, o *Logger*, *Appender* e o *Layout*. O *Logger* tem a função de receber as mensagens e incluí-las ao *output* de acordo com a configuração, *Appender* é o *output* do *log*. A forma como o *log* é exibido é definida pelo *Layout*.

### Logger

O *Logger* é responsável por receber a mensagem e torná-la disponível para o *Appender*. Ao utilizar o log4j é necessário configurar o nível hierárquico desejado para a exibição das mensagens, que podem ser *debug*, *info*, *warn*, *error* e *fatal*.

### Appender

O *Appender* é o modo de como será exibido o *log*, como exemplo, um arquivo HTML, ou arquivo de log ou no próprio console. Para definir qual *appender* a utilizar basta escolher um dos disponíveis no pacote org.apache.log4j e *setá*-lo na configuração do Log4j.

### Layout

O *layout* é o formato de como será exibida a mensagem de *log*. O primeiro passo é atribuir o *Layout* ao *appender*. Conforme o *layout* setado ao *Appender*, implicará no tipo da formatação com que os *logs* serão gerados.

## Oracle XE 10*g*

*Oracle Database 10g Express Edition (Oracle Database XE)* é um banco de dados baseado no *Oracle Database 10g Release* *2*. Gratuito e livre para desenvolvimento e distribuição, simples de administrar, voltado especificamente para:

* Desenvolvedores trabalhando com PHP, Java, .Net e outras aplicações open source;
* *DBAs* que precisam de um banco de dados gratuito para treinamento e desenvolvimento;
* Instituições educacionais e estudantes que precisam de uma banco de dados gratuito em seus currículos;

Com Oracle Database XE, é possível de desenvolver aplicações com um banco de dados poderoso e líder de indústria. Permite assim fazer a atualização para versões pagas do Oracle quando necessário sem custos complexos de migração.

Oracle Database XE pode ser instalado em qualquer máquina com qualquer número de processadores(um banco por máquina), mas o XE só irá guardar até 4GB de dados de usuários, e usará apenas 1GB da memória ram, e apenas um processador por máquina.

O Oracle Database XE inclui o Oracle HTML DB, um ambiente gráfico para desenvolvimento de aplicações centradas no banco e administração do mesmo[11].

Figura 5: Interface do Oracle HTML DB

## Hibernate

Hibernate é um poderoso, de alta performance, framework de mapeamento objeto-relacional. Ele permite que seja desenvolvido classes persistentes seguindo o idioma orientado à objetos, incluindo associações, herança, polimorfismo, composição e *collections.* O Hibernate permite que as *queries* sejam expressas em sua própria linguagem portável de SQL, o HQL, com também em SQL nativo, ou com sua API orientado à objetos.

Diferentemente de outras soluções de persistência, o Hibernate não esconde o poder do SQL e garante que seu investimento na tecnologia relacional será sempre válido. Sua licença LGPL permite que o Hibernate seja usado tanto em projetos open source quanto comerciais.[12]

### Mapeamento Objeto-Relacional

Mapeamento Objeto-Relacional é um meio de persistência automático de objetos java para tabelas de um banco de dados, usando meta dados que descrevem o mapeamento entre objetos e o banco de dados.

Isso implica em certa perda de performance. De qualquer maneira, se ORM é implementado como um *middleware*, há muitas oportunidades e maneiras para otimizar que não existiria numa camada de persistência codificada à mão. O preço que se paga com ORM (em tempo de desenvolvimento) é a previsão e gerenciamento dos meta dados que definem o mapeamento. Mas novamente, o custo é menor que o equivalente custo que seria envolvido para manter um código em uma solução que a camada de persistência fosse feita à mão.

Uma solução ORM consiste nas seguintes quatro partes:

* Uma API para fazer operações básicas CRUD[[1]](#footnote-1) em objetos de classes persistentes.
* Uma linguagem ou API para especificar *queries* que referenciam classes e propriedades de classes.
* Uma maneira fácil de especificar mapeamento de meta dados
* Uma técnica para a implementação ORM interagir com objetos transacionais e executar checagem, *lazy association fetching*, e outros mecanismos de otimização e funções.

*Lazy association* é quando é definido que os dados só serão buscados quando forem necessários ou requisitados na aplicação.

### Por que usar um framework ORM

Uma implementação ORM é uma coisa complexa – menos complexa que um servidor de aplicação, porém mais complexo que uma aplicação web feita em um framework como Struts ou Tapestry. Há várias razões para se introduzir um outro elemento complexo em um sistema.

#### Produtividade

Código de persistência de dados pode ser um dos mais tediosos códigos em Aplicações Java. O Hibernate elimina um grande trabalho e deixa o desenvolvedores se concentrar no problema do negócio. Não importa qual estratégia de desenvolvimento de aplicações seja preferido, Hibernate usado junto com as ferramentas apropriadas significa economia de tempo.

#### Manutenabilidade

Poucas linhas de código torna o sistema mais fácil de ser entendido, e enfatiza a lógica de negócios. Um sistema com poucas linhas também é fácil de ser feito o *refactoring* do código. Persistência automática objeto-relacional reduz a quantidade de códigos, e assim, a complexidade do mesmo.

Há outras razões para que com Hibernate, a aplicação torne-se mais fácil de ser feita manutenção. Em sistemas com persistência codificada à mão, é inevitável a tensão que existe entre a representação relacional e o modelo de objetos implementado no domínio da aplicação.

Modificações feita para um sempre envolve modificações à outro. E freqüentemente o design da representação é comprometido para acomodar a existência de outro. ORM fornece uma maneira elegante de usar a orientação à objetos no lado do código java, e quando alterado um modelo, poucas coisas mudam ao outro.

#### Performance

Uma dito comum é que código feito à mão sempre será mais rápido que a persistência automática. Isto é verdade da mesma maneira que código assembly pode ser mais rápido que um código Java escrito à mão, ou outra linguagem qualquer.

Outra interessante questão é quando é considerado o tempo como requisito no desenvolvimento. Ser mais rápido não significa que todos os programas devem ser feitos em *assembly.*

Dado uma tarefa de persistência, muitas otimizações são possíveis. Algumas são muito mais fáceis se forem feitas à mão com SQL/JDBC. Mas muitas otimizações também são muito mais fáceis para serem feitas de forma automática com ORM.

Hibernate permite que muitas otimizações sejam usadas o tempo todo. De qualquer forma, persistência automática aumenta a produtividade do programador, tanto quanto gastaria-se mais tempo otimizando um código feito à mão

Finalmente, as pessoas que implementam ORM em softwares provavelmente tem muito mais tempo para investigar sobre otimizações na performance do que se não usassem ORM.

#### Independência de fornecedor

Um ORM abstrai a aplicação sem precisar a codificação de códigos em sql. Se a ferramenta suporta diferentes bancos de dados, isso ainda oferece certa portabilidade à aplicação.

É muito mais fácil fazer software multi-plataforma usando ORM. Migrações ocorrem de forma mais fácil com menos riscos. Um aplicativo independente de banco de dados ainda ajuda no desenvolvimento de cenários onde os desenvolvedores usam um banco de dados simples e local, mas fazem o deploy para ser instalado em um ambiente de produto com um banco de dados diferente.[13]

## Commons Configuration

O framework *Commons* *Configuration* permite que as propriedades de configuração sejam acessadas de uma variedade de diferentes fontes. Não importa se as propriedades estão armazenadas em arquivos texto, um documento XML, ou uma árvore JNDI, elas podem ser acessadas da mesma maneira através de uma interface genérica *Configuration*.

Outro ponto forte do *Commons* *Configuration* é sua habilidade de agrupar diferentes configurações de fontes heterogêneas e tratá-las com uma simples lógica de configuração, centralizando-as em uma única interface de acesso.[14]

## Design Pattern Singleton

Os *Design* *Patterns* são uma coleção de padrões de desenho de software, que são soluções para problemas conhecidos e recorrentes no desenvolvimento de software.

Um *Pattern* descreve uma solução comprovada para um problema de desenho recorrente, dando ênfase particular no contexto e forçando a aproximação do problema, e as conseqüências e o impacto de sua solução.

*Patterns* são dispositivos que permitem que os programas compartilhem conhecimento sobre o seu desenho. Documentar um padrão (*pattern*) é uma maneira de poder reusar a forma de resolver problemas e a melhor maneira de se resolver um problema de desenho de software.

No sistema, foi utilizado o design *pattern* *Singleton.* Ele é um dos mais simples padrões de objeto da programação[15].

Segundo o Livro *Effective* *Java* “*A* *singleton is simply a class that is instantiated exactly once. Singletons typically* r*epresent some system component that is intrinsically unique, such as a video display or file system*”[16]. Traduzindo, *singleton* é uma simples classe instanciada exatamente uma vez. *Singletons* tipicamente representam um componente intrinsecamente único, como o *display* de vídeo ou o sistema de arquivos.

Utilizando esse *design* *pattern* em uma classe é garantido que:

* Apenas uma instancia dessa classe existe no programa inteiro.
* A classe ficará acessível globalmente de forma organizada e coerente.

# ANÁLISE UML

A análise foi feita com a ferramenta JDeveloper. Após a definição dos principais requisitos para que o software desenvolvido resolva o problema em questão, foi feito o diagrama de casos de usos.

Na figura é mostrado o diagrama de casos de uso das funcionalidades básicas do sistema:

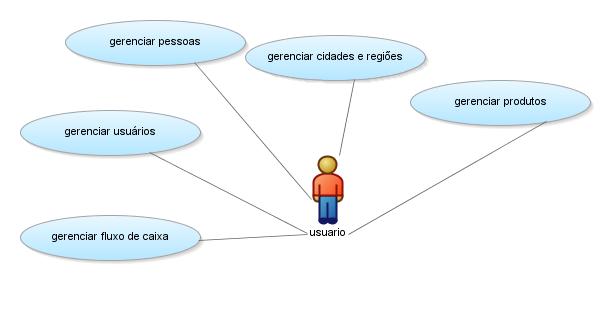


Figura 6: Diagrama de casos de uso com funcionalidades básicas do sistema.

Na figura 6, o usuário representa todos os usuários do sistema sem distinção de grupo. Foi incluído no diagrama as funcionalidades básicas do sistema, das quais, servem para guardar e gerenciar as informações sobre o negócio. O caso de uso “gerenciar produtos” é a inserção, edição e exclusão de produtos que poderão ser negociados. O caso de uso “gerenciar fluxo de caixa” é o controle de contas à receber e à pagar.

Muitos casos de uso triviais estão englobados em outros, pois não seria interessante poluir o diagrama com dezenas de casos de uso sobre inserção, alteração e exclusão de diversos tipos diferentes de objetos.

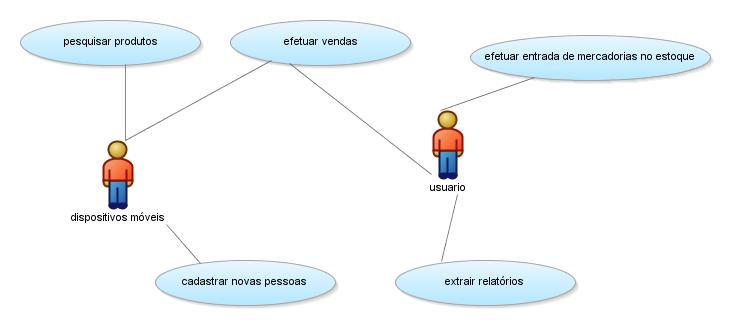


Figura 7: Casos de uso com as funcionalidades do sistema.

Outros casos de uso são mostrados na figura 7. O ator “dispositivos móveis” na verdade representa qualquer meio computacional que tenha tecnologia para acessar e consumir *Web* *services* , assim como fazer a invocação remota dos métodos disponibilizados através do arquivo descritor WSDL do serviço, que estaria instalado no container J2EE.

Como o ator “dispositivos móveis” tem comunicação com um número reduzido de casos de uso, pois ele enxergará apenas acesso aos métodos que serão necessários ser disponibilizados.

Ainda observa-se na figura 6, que o ator Usuário poderá extrair relatórios, como o relatório de “*ranking* de vendas”, que será visualizado em forma de gráficos, para que seja mostrado o desempenho de cada vendedor em uma maneira mais fácil e agradável de visualização. E ambos, usuario e dispostivos móveis poderão efetuar vendas. O ator usuário não está ligado ao caso de uso “cadastrar novas pessoas” no diagrama da figura 7 porque já está definido essa funcionalidade à ele no diagrama mostrado na figura 6, mais especificamente, na comunicação dele com o caso de uso “gerenciar pessoas”.

## Diagrama de classes

### Cadastros

Entre os os principais cadastros do sistema, pode ser citado os cadastros de: pessoas, produtos, contas, regiões movimentos.

A figura 8 representa os cadastros de pessoas:

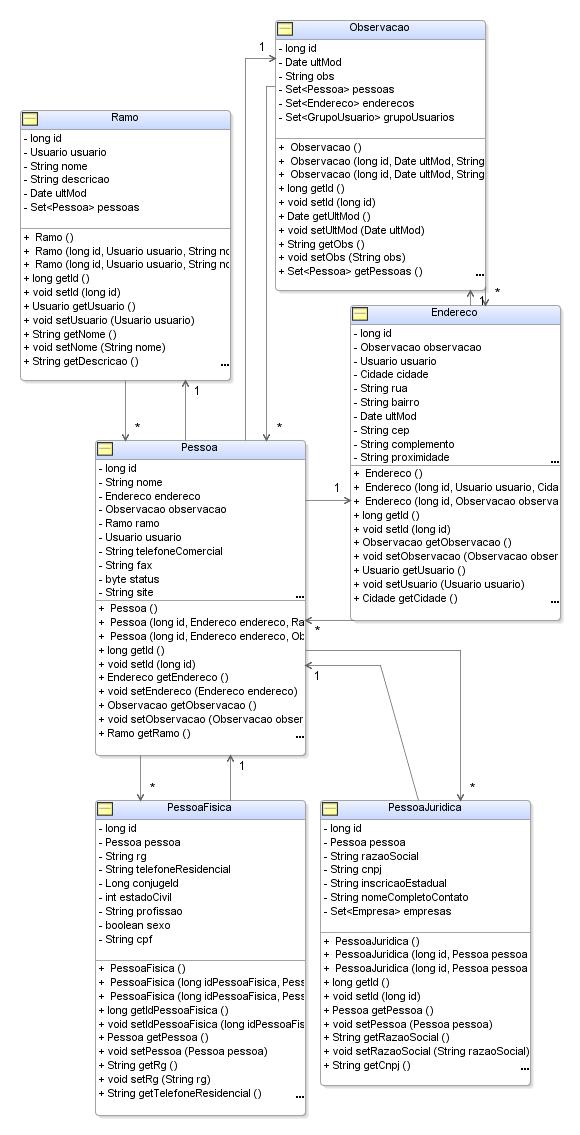


Figura 8: Diagrama de classes do cadastro de pessoas

As pessoas possuem um endereço, do qual podem possuir observações. O endereço sempre pertencerá apenas à uma única pessoa, uma vez que quando uma pessoa for removida, seu endereço também será removido. Um Ramo pode estar relacionado para muitas pessoas. A classe Ramo é o ramo de negócio que a pessoa atua, podendo ser, informática, saúde, desporto, etc.

A figura 9 representa os cadastros de regiões, endereços ou localizações que serão usados nos endereços do sistema:

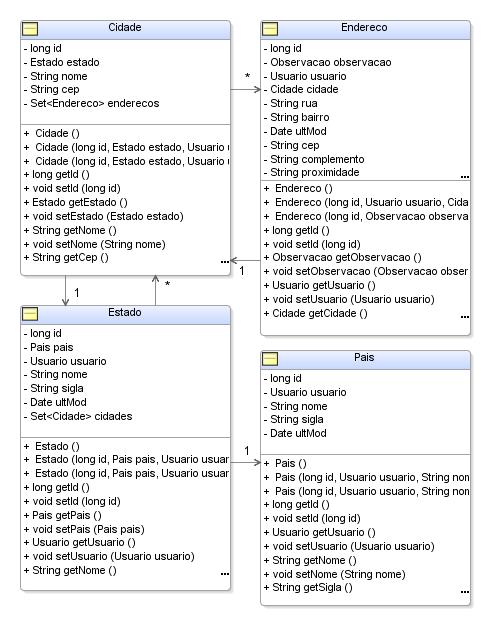


Figura 9: Diagrama de classes das regiões

Um endereço é único por pessoa, ele pertence à uma cidade, que pertence à um estado, que por sua vez pertence à um país.

A figura 10 representa o diagrama de classes dos produtos do sistema:

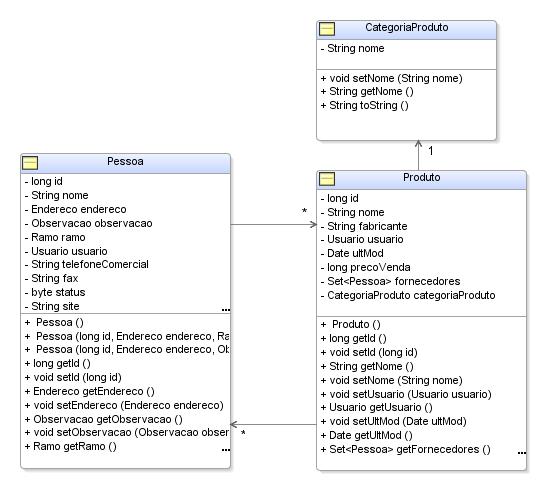


Figura 10: Diagrama de classes dos produtos do sistema

Os fornecedores do produto podem ser qualquer um dos objetos que estão no cadastro de pessoas. Um produto pode ter vários fornecedores e apenas uma categoria de produto. CategoriaProduto é para melhor organizar os vários tipos diferentes de produtos que o sistema visa atender.

## Movimentos de mercadorias

Tanto as vendas quanto as entradas de produtos geram movimentos. Os movimentos são aplicados quando mercadorias ou matérias primas entram no estoque e quando vendas são efetuadas. Na figura 11, o controle dos movimentos de produtos no sistema:

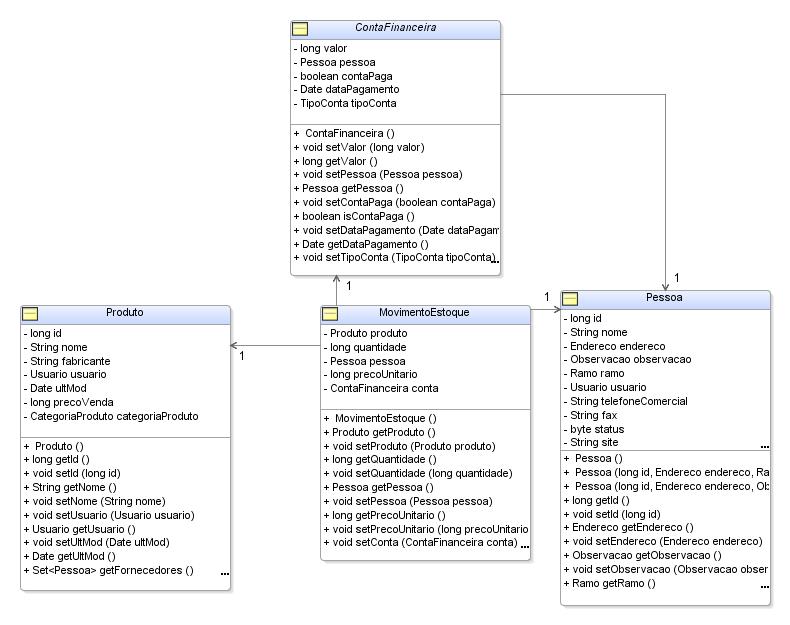


Figura 11: Diagrama de classes dos movimentos de estoque do sistema

Os dois tipos de movimentos podem gerar contas, conforme o tipo de movimento, a conta é à receber ou à pagar. Caso a conta seja à pagar, o atributo do tipo Pessoa é um fornecedor, e caso a conta seja do tipo à receber, Pessoa é um cliente que comprou algum produto.

Para saber a quantidade em estoque de determinado produto, basta somar a quantidade de produtos que saiu e subtrair o valor da quantidade do mesmo que entrou.

Quando um usuário do grupo vendedores efetua uma venda, automaticamente é gerado uma conta à receber, e quando algum produto entra no estoque da empresa, automaticamente é criado uma conta à pagar.

## Contas

As contas são o controle de caixa do sistema, para que seja possível saber quais contas faltam à pagar e quais têm-se para receber. Na figura 12 o controle de contas à receber e pagar no sistema:

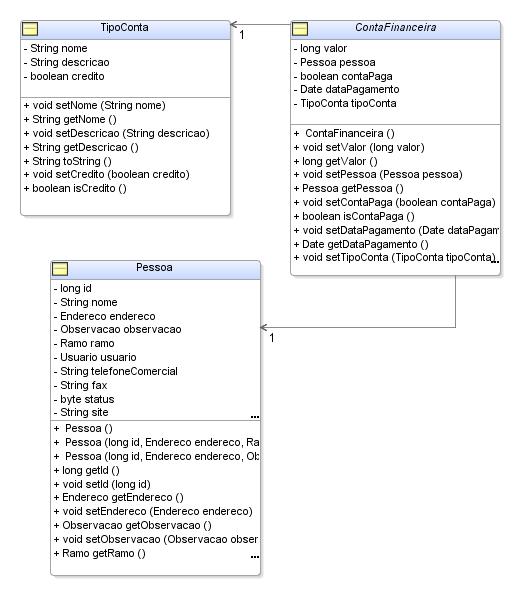


Figura 12: Diagrama de classes das contas financeiras do sistema

Uma pessoa pode ter diversas contas à receber e à pagar caso ela seja fornecedor e cliente ao mesmo tempo. Para ser pesquisado quais contas estão atrasadas, a lógica é feita analisando as informações contidas nos atributos “contaPaga” e “dataPagamento”. Caso a data de pagamento já tenha passado e a conta ainda não tenha sido paga, a pessoa associada à conta está devendo ou a empresa deve à pessoa. Para saber se a conta é a receber ou à pagar, conta financeira possui o atributo “tipoConta” onde sua classe existe o atributo “credito”, caso o valor para esse atributo seja verdadeiro, a conta é de crédito, ou seja, à receber, caso contrário, à pagar.

## Usuários

Os usuários e grupos de usuários sistema são organizados como mostra o diagrama da figura 13:

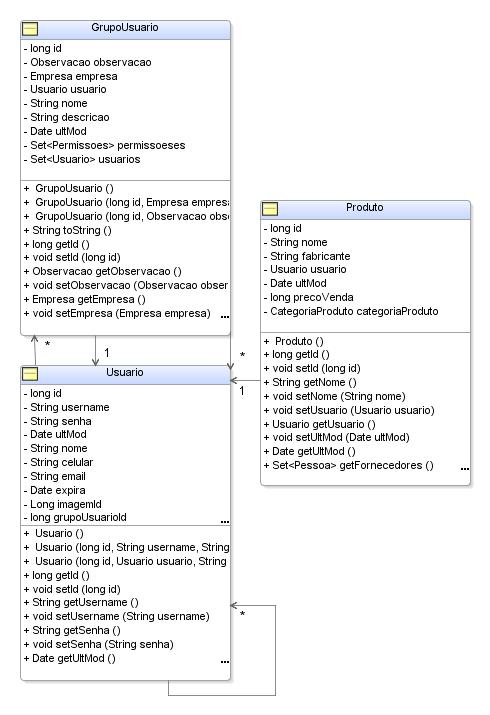


Figura 13: Diagrama de classes mostrando associação entre Usuário e Produto

Usuário, assim como muitas entidades do sistema, possuem um atributo do tipo usuário para definir qual o ultimo usuário que vez uma alteração no objeto. Assim, no caso de usuário existe uma auto-associação, e no classe Produto existe o atributo do tipo Usuário, como observa-se na figura 12. Para saber se o usuário é vendedor, basta verificar de qual grupo ele pertence, uma vez que a classe Usuário é associado diretamente com a classe GrupoUsuário. Um usuário pode pertencer apenas à um grupo de usuários.

# UTILIZAÇÃO DAS TECNOLOGIAS NO DESENVOLVIMENTO DO SISTEMA

Este capítulo apresenta o sistema. Relata-se a instalação e configuração de alguns softwares usados no desenvolvimento e também a utilização dos mesmos nas funcionalidades do sistema.

## Instalação e configuração das ferramentas

Antes de iniciar o desenvolvimento, foi necessário configurar os softwares necessários. Primeiramente o *Subversion* para que o controle de versões seja feito desde as primeiras classes codificadas do projeto, e manter um histórico regular de todas as revisões feitas de cada arquivo, sendo ele do tipo java, figura, arquivo jar, etc. Dando a possibilidade de voltarmos no tempo e ver como estava cada um deles. Apesar de o JDeveloper tem um controle de versões interno, assim como outras ferramentas, como *Visual Studio, Borland Delphi*, etc., o *Subversion* foi usado para manter a segurança, deixando um software à parte para esse tipo de tarefa.

A instalação do Oracle XE para ser o banco de dados, o SqlDeveloper para ser a interface de design e criação das tabelas e estruturas do banco de dados, o *Kit* Java de Desenvolvimento de Software JSDK 1.5 não possuem grandes segredos em sua instalação e não serão explicados detalhadamente.

## Banco de dados Oracle XE

Os sistemas de gerenciamento de bancos de dados Oracle são um dos mais confiáveis do mundo, mas não adianta nada ter tal confiabilidade se o banco de dados que a aplicação for acessar seja mal estruturado.

A Oracle possui ferramentas pagas para o gerenciamento de seus bancos, mas para fazer concorrência com a ferramenta free *Toad* da *Quest Software*, lançou o SqlDeveloper que auxilia muito nas tarefas de design do banco de dados, alteração de tabelas, modificação de registros, triggers, sequences, etc.

Baseado no diagrama de classes, o banco de dados foi construído como SqlDeveloper. Na figura 14 pode ser observado um *screenshot* de uma tela do SqlDeveloper para a edição de uma chave primária de uma tabela:

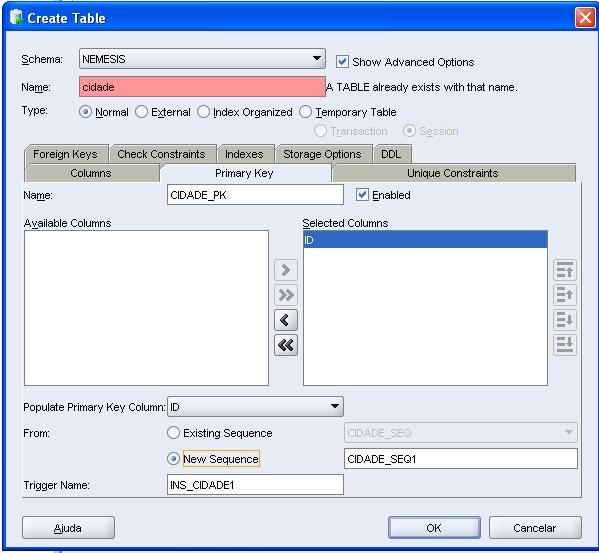


Figura 14: Criação de um campo chave primária no SqlDeveloper

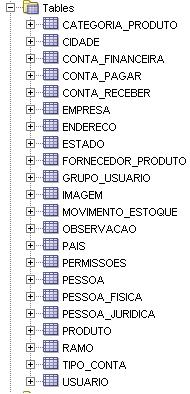


Figura 15: Visualização geral das tabelas do sistema mostradas no SqlDeveloper

Na figura 15 pode-se ser observado a árvore de tabelas que precisaram ser criadas:

No banco de dados do sistema não foi feito nenhuma *trigger* relevante, para manter ao máximo a portabilidade entre bancos. As únicas triggers usadas foram aquelas geradas automaticamente quando se cria um campo com auto-incremento, como mostra na figura 14. Na tela do *screenshot*, a *trigger* que esta sendo criada com o nome INS\_CIDADE1 irá chamar o próximo valor da *sequence* CIDADE\_SEQ1, caso o campo em questão tenha chegado nulo ao banco numa *query* executada contra ele.

Como foi usado Hibernate, para portar de banco de dados seria muito fácil. O Hibernate possui vários dialetos abrangendo grande parte dos bancos de dados mais usados pela comunidade de desenvolvedores.

Sendo que não há triggers de forma significativa, o único empecilho para troca de base de dados seria criar as triggers de auto-incremento no novo banco a ser portado. Dependendo do banco, nem *trigger* precisaria ser criada. Num banco Postgres, por exemplo, bastaria alterar o campo *id* de *number* para serial, que é o tipo de dados numérico do Postgres que faz o auto-incremento automático. No arquivo XML do Hibernate que mapeia a classe com o tal identificador, foi codificado o atributo da seguinte maneira:

<id name="id" type="long">

<column name="ID" precision="10" scale="0" />

<generator class="identity"/>

</id>

O atributo “class” está com o valor “identity”, dizendo ao Hibernate que o banco cuidará do incremento do campo automaticamente, dentro da *tag* XML “generator”.

## Hibernate, e mecanismo de persistência

As classes Java de um projeto são organizadas através de pacotes. No projeto desenvolvido, o pacote que possui as classes POJO e também os XMLs mapeamentos delas em relação as tabelas no banco de dados, ficam dentro do pacote que foi criado com a seguinte estrutura: *com.dudhoo.nemesis.hibernate,*

Seguindo o padrão de desenvolvimento do Java de nomenclatura única, disponíveis no site da Sun nas páginas à respeito de convenções de código da plataforma Java.

Na figura 16 têm-se a ilustração da classe “Cidade” com sua respectiva tabela no banco de dados:

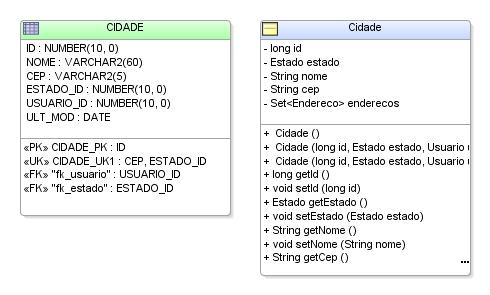


Figura 16: À esquerda a tabela Cidade e sua respectiva classe POJO à direita.

O Mapeamento dessa situação ficou feito da seguinte maneira:

1 <?XML version="1.0"?>

2 <!DOCTYPE Hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

3 "http://hibernate.sourceforge.net/Hibernate-mapping-3.0.dtd">

4

5 <Hibernate-mapping>

6 <class name="com.dudhoo.nemesis.hibernate.Cidade" table="CIDADE"

7 schema="NEMESIS">

8 <id name="id" type="long">

9 <column name="ID" precision="10" scale="0"/>

10 <generator class="identity"/>

11 </id>

12 <many-to-one name="estado" lass="com.dudhoo.nemesis.hibernate.Estado"

13 lazy="false" fetch="select">

14 <column name="ESTADO\_ID" precision="10" scale="0" not-null="true"/>

15 </many-to-one>

16 <many-to-one name="usuario"class="com.dudhoo.nemesis.hibernate.Usuario"

17 fetch="select">

18 <column name="USUARIO\_ID" precision="10" scale="0" not-null="true"/>

19 </many-to-one>

20 <property name="nome" type="string">

21 <column name="NOME" length="60" not-null="true"/>

22 </property>

23 <property name="cep" type="string">

24 <column name="CEP" length="5" not-null="true"/>

25 </property>

26 <property name="ultMod" insert="false" update=”false” type="date">

27 <column name="ULT\_MOD" length="7" not-null="false"/>

28 </property>

29 <set name="enderecos" inverse="true">

30 <key>

31 <column name="CIDADE\_ID" precision="10" scale="0"

32 not-null="true"/>

33 </key>

34 <one-to-many class="com.dudhoo.nemesis.hibernate.Endereco"/>

35 </set>

36 </class>

37 </Hibernate-mapping>

Na linha 6, no atributo *name* ,é informado que está sendo mapeado a classe POJO Cidade, da qual referencia a à tabela CIDADE, como está definido no atributo *table* e pertence ao schema do usuário NEMESIS no banco de dados Oracle.

Na linha 8, têm-se a *tag* *id* com o atributo *name*, que é a coluna da tabela onde está a chave primária, ou identificador de unicidade dos objetos dessa tabela, e logo em seguida, no atributo *type*, significando o seu tipo, que no caso é um tipo primitivo Java, o *long .*

Na linha 10, têm-se uma *tag* filha da *tag* *id*, que define o tipo do gerador de auto-incremento da coluna de identificação, quando houver. Nesse caso, o auto-incremento é gerado automaticamente pelo banco, ou seja o valor do atributo *class* é “*identity*”.

Como temos a relação de muitas cidades para um estado, na linha 12 foi mapeado essa característica A atributo *name* referencia à propriedade da classe java Cidade, que possui o atributo com o nome “estado”, cujo, é do tipo "com.dudhoo.nemesis.hibernate.Estado". O atributo *lazy* espera um valor booleano, que diz aos mecanismos do Hibernate que quando for carregado uma objeto Cidade através de sua sessão ele irá enviar consultas adicionais ao banco de dados e carregar automaticamente o valor da propriedade “estado” do objeto Cidade. Caso *lazy* seja atribuído como *false*, ele irá fazer sempre automaticamente, mesmo que não seja usado, e caso o atributo esteja setado para *true*, ele só irá fazer o novo *select* no banco de dados quando o valor for requisitado, e a sessão do Hibernate precisará estar aberta, senão ocorrerá uma exceção.

Na linha 26 é onde inicia o mapeamento do atributo da propriedade que guarda a última data que o objeto foi modificado. Observa-se que tem os atributo *insert* e *update* setados como false. Isso diz ao Hibernate que ele não irá fazer inserções nem atualizações por conta própria do banco de dados nesse atributo “ultMod”, mesmo que os valores sejam trabalhados e inseridos pelo programador na memória. Configurado dessa maneira, em um ambiente de rede o banco de dados pegaria o *timestamp* do servidor ao invés da máquina cliente, que ocasionalmente poderia estar errado. Para que isso ocorra, ainda foi necessário, na coluna da tabela no banco de dados, setar o valor *default* desse objeto data para *current\_timestamp,* que no caso é a sintaxe em Oracle.

Na linha 29, é o recurso de *set* do Hibernate para *collections*. Ele faz com que eu possa resgatar uma *collection* de objetos associados a um objeto, representado por um atributo do tipo *Set* de Java, nas classes *Pojo*. Na linha 31 é dito que vou buscar os endereços que tenham o valor da coluna “cidade\_id” de mesmo valor que o identificador da classe cidade. E na linha 34, é dito que o objeto é uma instância da classe Endereço.

Apenas criar esses arquivos de mapeamento não basta para que o Hibernate os reconheça, é preciso adicionar os mesmos à *engine*. Quando inicia-se o Hibernate, existem duas formas de adicionar essa classe mapeada para que o Hibernate cuide, uma é através do código, chamando o objeto Configuration da API do Hibernate, e a outra é definindo no arquivo XML de configuração, separado do código, do qual lá também é definido a *string* de conexão com o banco de dados, o dialeto do banco que estamos à usar, no caso Oracle, usuário, senha de acesso ao banco, e demais parâmetros que devem ser passados.

Nas classes java codificadas, no mesmo *package* que se encontra os mapeamentos e classes POJO, encontra-se também a classe responsável pela lógica do negócio, separada de todo o resto do projeto.

O padrão para os nomes das classes de lógica de negócio segue o formato de o nome da classe POJO seguido do sufixo “*Home*”, sendo assim, para a classe Cidade.java ficou CidadeHome.java.

CidadeHome.java assim como todas as classes que possuem regras de negócio herdam da classe pai abstrata HibernateRoot.java. Na Figura 17, o diagrama UML demonstra algumas:

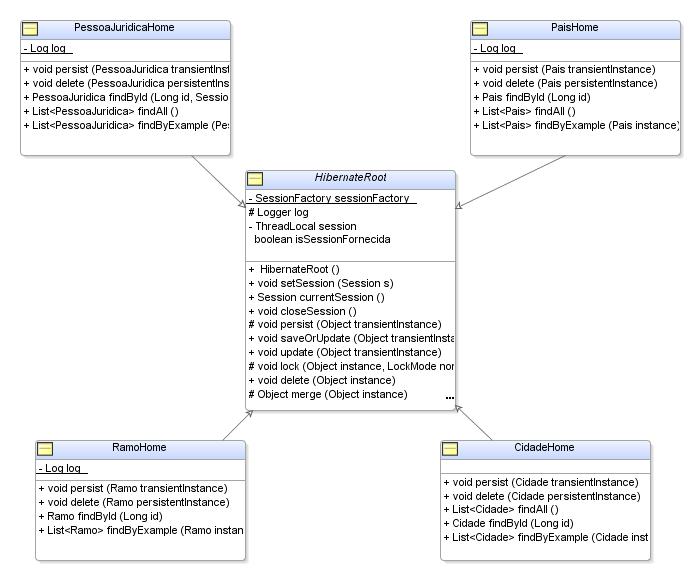


Figura 17: Herança entre as classes de regras de negócio com o mecanismo de persistência.

Os métodos das classes “Home”, podem criar a sessão diretamente de sua classe pai, fazer as devidas operações contra o banco de dados, usufruindo de transações e outras classe se necessário, e em seguida, fechar a conexão, devolvendo os resultados às classes de lógica de interface. Com esse tipo de arquitetura definida, fica fácil fazer a manutenção no código, assim como a resolução de *bugs* relacionados à persistência, pois está tudo centralizado na classe pai. Sem Hibernate, teria de ser codificado toda a parte de acesso à dados, *Strings* sql de inserção, alteração, exclusão, etc, com o *pattern* *Dao*, o tanto é que o papel do Hibernate aqui é deixar transparente toda as *strings* sql que precisariam ser codificadas.

Para tornar um objeto já mapeado persistente, basta ser feito o seguinte:

Cidade city = new Cidade();

....//atribuir os valores a nova cidade

new CidadeHome().saveOrUpdate(city);

A classe CidadeHome irá criar a sessão, persistir o objeto, e então fechar a sessão. Tanto para salvar quando para atualizar um objeto é chamado o mesmo método. O Hibernate sabe se precisa ou não atualizar verificando se o atributo identificador da classe já existe ou não, caso não exista, ele supõe que será preciso fazer um *Insert* no banco de dados, e caso exista, será feito um *update.*  E quando é feito um *insert*, o Hibernate seta automaticamente no objeto persistido o identificador que foi acabado de ser criado no banco de dados.

Como o Log4j foi o framework usado para ser feito os *logs* de toda a aplicação, com a propriedade de configuração identificada pela chave *show\_sql setada* para “*true*”, o Hibernate fica configurado para criar *logs* do que está sendo enviado no banco.

## O Design Pattern Singleton

Ao iniciar a aplicação, um *login* é requerido, tanto por segurança quanto para que posteriormente o DBA poder saber quem fez as alterações nos objetos que possuem o atributo “Usuário”.



Figura 18: Tela de login do sistema

Ao clicar no botão “entrar”, um objeto do tipo UsuarioHome é criado. Esse objeto possui o método “findByLoginAndPass” que retorna o usuário que possui o *login* e senha fornecido. Caso retorne um usuário, o *login* é efetuado, caso o método retorne um objeto nulo, uma mensagem de erro é mostrada.

Figura 19: Tela de erro do Login

Caso o *login* seja válido, é mostrado instantaneamente a tela inicial do sistema ao usuário. Mas ocorrem alguns processos nesse meio tempo. É preciso além de setar qual usuário está logado, carregar algumas configurações de sistema, e se não existirem, serem criadas com os valores *default.*

Um objeto que foi codificado chamado *Application* simula a sessão do usuário logado. Assim que é feito o *login* e o usuario logado é carregado do banco, é guardado para essa classe em memória. Através do objeto *Application* podemos carregar as configurações do sistema. Como atributo dele foi criado um objeto instancia da classe *XMLConfiguration* do framework *CommonsConfiguration*. O objeto *Application* verifica se existe um arquivo de configuração criado por ele no pasta raiz de onde está o programa, e caso não exista, ele cria esse arquivo, e cria todas as configurações na memória para em seguida salvá-las *serializando* o em XML e usando os próprios métodos do objeto *XMLConfiguration*.

Também é setado o usuário para a aplicação no objeto *Application*, que possui o atributo do tipo Usuário.

Então é criado o formulário principal e e setado o nome do usuário logado na barra de título e finalmente, o formulário principal é mostrado.

No entanto, o Hibernate demora alguns segundos para iniciar, conforme o hardware que está o software instalado, pois ele tem que fazer os mapeamentos, a conexão com o banco, fazer *parser* dos XMLs, etc, então, ainda no construtor da classe *Application*, foi criado um thread que chama um método qualquer do Hibernate, para que ele vá sendo iniciado enquanto ainda o usuário está digitando sua senha.

*Application* e *StartFrame*, que é a classe de formulário principal herdada de *JFrame*, ambas utilizam esse design *pattern*, uma vez que somente uma instancia delas deve existir por aplicação, e também devem estar acessíveis de qualquer lugar ou objeto. Para fazer a implementação desse design pattern em um objeto, em java é necessário codificar da seguinte forma:

public class ObjetoUnico{

private static ObjetUnico instance;

private ObjetoUnico(){

}

public static ObjetoUnico getCurrentInstance(){

if(instance == null){

instance = new ObjetoUnico();

}

return instance;

}

}

Usando o atributo do tipo estático, significa que a posição da memória para esse objeto será sempre a mesma, ou seja, não importa quantos objetos sejam criados e chamados, em qualquer parte da aplicação ele será sempre o mesmo. Deve haver muito cuidado ao criar objetos estáticos, pois uma aplicação pode tornar-se um emaranhado de variáveis públicas desorganizadas. Utilizando o *pattern* *singleton* é uma forma segura e correta de acessar objetos públicos sem que isso aconteça.

O método construtor dessa classe está privado com a palavra chave *private*, isso faz com que não seja possível instanciar essa classe apenas chamando seu método construtor, pois métodos privados só podem ser acessados a partir da própria classe, então o programador é forçado à chamar objeto *singleton* para que possa utilizá-lo, assim não há perigo algum de serem criados dois objetos dessa classe. Caso seja tentado fazer a criação do objeto por outro objeto, é disparado um erro de compilação.

## Troca de mensagens e separação de classes distintas

No método construtor de *StartFrame*, chama um método privado configuraMenus(). Cada *JMenuItem* de cada menu que foi criado na barra de menus possui eventos que foram tratados com a criação de objetos que implementam a interface *ActionListener*, e que possuem a lógica toda para que ao passo que sejam clicados, tenham acesso à classe do formulário principal *StartFrame*, do qual possui métodos públicos onde é possível adicionar novos paineis quando for necessário.

Abaixo têm-se uma parte da classe *StartFrame* e mostra-se como é construído os menus do formulário Principal:

24 public class StartFrame extends JFrame {

25 private static StartFrame instance;

26

27 /\*\*

28 \* Cria uma nova instancia de FrameMain

29 \*/

30 private StartFrame(String title) {

31 super(title);

32 Application.getConfiguration();

33 iniciaThreadHibernate();

34 new DailogLogin().setVisible(true);

35

36 configuraMenus();

37 configuraBarraTitulo();

38 }

39

40 public void configuraBarraTitulo() {

41 this.setTitle(this.getTitle() + " - " +

42 Application.getUsuarioLogado().getNome());

43 }

44

45 /\*\*

46 \*Inicia o Hibernate em outra thread.

47 \*/

48 private void iniciaThreadHibernate() {

49 InitHibernate h = new InitHibernate();

50 h.start();

51 }

52

53 public static StartFrame getCurrentInstance() {

54 if (instance == null) {

55 instance =

56 new StartFrame(Application.getConfiguration().

57 getString(Application.APP\_NAME));

58 }

59 return instance;

60 }

61

62 private void configuraMenus() {

63 JMenu menuCadastros =

64 super.addMenu(

65 new MenuCadastroFactory().constroiMenuCadastros());

66 JMenu menuCompras =

67 super.addMenu(

68 new MenuComprasFactory().getMenuCompras());

69 JMenu menuFinancas =

70 super.addMenu(

71 new MenuFinancasFactory().getMenuFinanca());

72 JMenu menuRel =

73 super.addMenu(

74 new MenuRelatoriosFactory().getMenuRelatorios());

75 JMenu menuSeguranca =

76 super.addMenu(

77 new MenuSegurancaFactory().getMenuSeguranca());

78 JMenu menuVendas =

79 super.addMenu(

80 new MenuVendasFactory().getMenuVendas());

81

..........

Na linha 34, e mostrado o *JDialog* de *login*, do qual é *modal*, ou seja, a linha de execução fica parada até que seja feito um *dispose()* nesse *JDialog*. Após o usuário fazer seu *login* correto, irá executar a linha 36, onde irá chamar os métodos que criam os objetos que realmente constroem os menus. Abaixo mostra-se o uma parte da classe que cria o menu de cadastros:

33 public class MenuCadastroFactory {

34 /\*\*

35 \* Creates a new instance of MenuCadastroFactory

36 \*/

37 public MenuCadastroFactory() {

38

39 }

40

41 public JMenu constroiMenuCadastros() {

42 JMenu cadastro = new JMenu();

43 cadastro.setText("Cadastro");

44 cadastro.setToolTipText("Cadastros e " +

45 "pesquisas no banco de dados");

46 cadastro.setMnemonic('C');

47 adicionarCadastroCidades(cadastro);

48 adicionarCadastroComuns(cadastro);

49

50 adicionarCadastroPessoasFisicas(cadastro);

51 adicionarCadastroPessoasJuridicas(cadastro);

52 adicionarCadastroProdutos(cadastro);

53 return cadastro;

54 }

55

56 /\*\*

57 \* Adiciona o cadastro de pessoas

58 \*/

59 private void adicionarCadastroPessoasJuridicas(JMenu menu) {

60 JMenuItem menuItem = menu.add("Pessoas Jurídicas");

61 menuItem.setAccelerator(KeyStroke.getKeyStroke('P',

62 InputEvent.ALT\_MASK + InputEvent.CTRL\_MASK));

63 menuItem.addActionListener(new

64 ListenerCadastrarPessoasJuridicas());

65 }

66

67 /\*\*

68 \* Adiciona o cadastro de produtos

69 \*/

70 private void adicionarCadastroProdutos(JMenu menu) {

71 JMenuItem menuItem = menu.add("Produtos");

72 menuItem.setAccelerator(KeyStroke.getKeyStroke('P',

73 InputEvent.ALT\_MASK + InputEvent.CTRL\_MASK));

74 menuItem.addActionListener(new

75 ListenerCadastroProdutos());

76 }

.........

Na linha 41 têm-se o método que cria o *JMenu* que aparece na barra de menus, ele por sua vez, chama outros métodos privados adicionando *JMenuItems* a ao menu antes de enviá-lo. Como os métodos que estão na linha 59 e 70 são semelhantes em sua arquitetura, será mostrado em detalhes apenas no último, que é a criação do menu de controle das regiões do sistema.

Na linha 72, seta-se o atalho de teclado para acessar o evento de clique desse item do menu, sendo que é preciso segurar as teclas *alt* e *control* . E o *actionListener* setado para o menu, que é a interface implementada que o menu vai chamar ao ser clicado, esta em outra classe, a ListenerCadastroProdutos, que será abordada a seguir:

13 public class ListenerCadastrarProdutos implements ActionListener{

14 private PanelPesquisaGeneric panel = null;

15

16 public void actionPerformed(ActionEvent actionEvent){

17 StartFrame.getCurrentInstance().

18 setCursor(new Cursor(Cursor.WAIT\_CURSOR));

19 panel = new PanelPesquisaProdutoBuilder()

20 .getJPanelPesquisaProduto(true, false);

21

22 StartFrame.getCurrentInstance().addJPanel(panel);

23 StartFrame.getCurrentInstance()

24 .setCursor(new Cursor(Cursor.DEFAULT\_CURSOR));

25 }

26 }

Com um clique no item do menu, surge a tela de pesquisa correspondente, como mostra a figura 20:



Figura 20: No meio da ilustração, a tela de pesquisa construída dinamicamente.

Esta classe chama o formulário principal na linha 17, pega e seta o cursor do *mouse* para ampulheta, simbolizando um estado de espera. Na linha 18, então chama a classe que irá construir um novo *panel*, do qual, em seu meio existe uma tabela para representar os dados tabulares pesquisados no banco. A classe que cria este *panel* é PanelPesquisaProdutoBuilder, que será abordada mais adiante. Na linha 22, pega-e o formulário principal e adiciona-se um *JPanel* ao mesmo, e por fim, na linha 23, seta o cursor do *mouse* para seu estado normal.

## 

## Representando dados tabulares

Como em muitas telas pretendia-se fazer a representação tabular dos dados que tinham no banco conforme o item do menu clicado, foi criado uma classe chamada PanelPesquisaGeneric.

A classe PanelPesquisaProdutoBuilder retorna um PanelPesquisaGeneric, que tem seu *layout* do tipo BorderLayout, da API do java, que divide automaticamente o panel em cinco partes(*south, north, west, leste* e *center*) e organiza automaticamente seus componentes dentro dele. PanelPesquisaGeneric possui uma *toolbar* em cima, com os *JButtons* de inserção, exclusão, e edição, e também possui um objeto *CustomTable*, herdado de *JTable.* Possui também uns componentes para filtragem de dados, e o botão de fechar logo na parte inferior de todos os componentes *swing.*

O objeto *CustomTable* que existe dentro dessa tela de pesquisa genérica além de herdado, foi acrescentado métodos, onde que passando um *FilteredRowSet* populado de uma *String* sql executada contra o banco irá criar as colunas e linhas da tabela conforme o resultado chegado. Na ilustração 19 têm-se o resultado de uma pesquisa onde dinamicamente foi criado o *JTable*.

O objeto C*ustomTable* possui bastante linha e outros objetos como atributo, no entanto, abaixo segue o método que faz a criação dinâmica da tabela:

59 public class EvilTable extends JPanel {

60

61 private JTable jTable1;

62 private FilteredRowSet filteredRowSet;

..........

145 public void setFilteredRowSet(FilteredRowSet crs,

146 Integer indexPk)throws Exception{

147 filteredRowSet = crs;

148 if (filteredRowSet == null) {

149 this.jTable1.setModel(null);

150 } else {

151 try {

152 crs.beforeFirst();

153 int colunasCount = crs.getMetaData().getColumnCount();

154 if ((indexPk != null) &&

155 colunasCount < indexPk) {

156 throw new Exception("A chave primária é a coluna " 157 + indexPk +

158 ". Mas a pesquisa retornou apenas" +

159 colunasCount + " colunas.");

160 }else if((indexPk != null) && indexPk < 1) {

161 indexPk = null;

162 }

163

164 DefaultTableModel model;

165 model = new DefaultTableModel();

166 String columnNames[] = new String[colunasCount];

167 String temp;

168 for (int i = 0; i < colunasCount; i++) {

169 temp = crs.getMetaData().

170 getColumnName(i + 1);

171 columnNames[i] = temp;

172 }

173 temp = null;

174

175 atualizaComboCampos(columnNames);

176 model.setColumnIdentifiers(columnNames);

177 while (crs.next()) {

178 Object rowData[] = new Object[colunasCount];

179 for (int i = 1; i <= colunasCount; i++) {

180 rowData[i - 1] = crs.getObject(i);

181 }

182 model.addRow(rowData);

183 }

184 jTable1.setModel(model);

185

186

187

188 } catch (SQLException e) {

189 throw e;

190 log.error("Erro ao preencher tabela com linhas: " +

191 e.getMessage());

192

193

194 }

195 }

196 }

A linha 145 tem o método mais interno que já espera um *FilteredRowSet* criado por uma pesquisa. *FilteredRowSet* fez parte oficial da API do java a partir da versão 1.5. Na versão anterior, precisava ser feito o *download* à parte. Este objeto mantém um conjunto de linhas estruturadas, como se fosse um espelho das tabelas da consulta feita na memória, e ainda possui métodos para a filtragem de dados.

Na linha 146, o segundo atributo do método é saber qual posição de coluna está a chave primária da tabela, para que no futuro seja possível fazer as exclusões e alterações dos registros dos objetos que foram mostrados.

A linha 153 pega o *metadata[[2]](#footnote-2)* do *RowSet*, que são informações da estrutura da pesquisa que veio do banco de dados, assim têm-se o acesso do número de colunas que precisará ser criado. Na linha 154 é verificado se o número de colunas é menor que a posição da coluna de chave primária, se for, algo está errado e é disparado uma exceção.

Na linha 66 cria-se um *array* de *String* com o tamanho do número de colunas, então na linha 68 iteramos o *metadata* e adicionamos os nomes das colunas ao *array* de *strings*.

Na linha 177 é iterado o *rowset* e adicionado os dados correspondentes à sua coluna. Depois, na linha 184 é setado o novo *DefaultTableModel* à tabela. Com toda a estrutura programada, os dados tabulares podem ser visualizados conforme o FilteredRowSet setado através de uma consulta sql qualquer.

## Inserção, edição e validação dos dados

Para as validações dos dados são feitas antes de os dados serem inseridos no banco. Foi criado um objeto *ValidationException* que herda de *Exception* para ser disparado as exceções.

Assim que a tela de consulta é criada, é setado os eventos nos botões inserir, editar e excluir, com os devidos métodos chamando os devidos *JDialogs* para fazer as operações com os objetos do negócio. No caso comentado, chamaria o *dialog* de alteração e persistência de produtos.

Ao clicar em inserir Produto, é chamado o *dialog* que faz o tratamento. Esse objeto possui um *JPanelProduto* que possui um atributo produto, que é o que será salvo ou atualizado no banco de dados.

Figura 21: Tela de inclusão ou edição de Produto

Assim que for preenchido os valores e clicado em salvar, alguns processos de validação serão feitos. Abaixo o *JPanel* que está dentro do *dialog*, responsável pelo atributo produto, e um objeto reaproveitável.

28 public class JPanelProduto extends JPanel{

29 private Produto produto;

30 private JLabel jLabelProduto = new JLabel();

........

57 public Produto getProduto() throws ValidationException{

58 String nome = this.jTextField1.getText().trim();

59 int qut = 3;

60 if(nome == null || nome.length() < qut){

61 throw new ValidationException("Digite um " +

62 "produto com mais de " + qut + " caracteres.");

63 }

64 if(produto == null){

65 produto = new Produto();

66 }

67 long preco = Long.valueOf(

68 jTextFieldPreco.getText().toString() );

69 produto.setNome(nome);

70 produto.setPrecoVenda(preco);

71 produto.setFabricante(jTextFieldFab.getText().trim());

72 produto.setUsuario(Application.getUsuarioLogado());

73 produto.setFornecedores(panelForn.getFornecedores());

74 produto.setCategoriaProduto(

75 comboCategoriaProduto1.getCategoriaProduto());

76 return produto;

77 }

Na linha 60, caso o valor digitado no nome do produto ter menos de três caracteres, será lançada uma exceção do tipo *ValidationException*. Essa exceção foi criada como filha de *java.lang.Exception*.

Como dito antes, este JPanelProduto está como atributo de uma classe *dialog*, que faz os tratamentos dos erros de validação. Essa classe tem um evento à parte com a mesma funcionalidade que *ActionListener* do Java, porém customizada. Em resumo, o evento dela *doAfterClick* é chamado assim que for clicado em salvar:

16 public class SalvarProdutoListAdapter implements SavePressListener{

17 private JPanelProduto panel;

18 private Logger log = Logger.getLogger(this.getClass());

19 private JDialogProduto dialog;

20 public SalvarProdutoListAdapter(JDialogProduto \_dialog){

21 panel = \_dialog.getPanelProduto();

22 this.dialog = \_dialog;

23 }

24

25 public void doAfterClick(){

26 Produto prod;

27 try{

28 prod = panel.getProduto();

29 ProdutoHome home = new ProdutoHome();

30 home.saveOrUpdate(prod);

31

32 new EvilInformation(panel).

33 showMessage("Salvo com sucesso!");

34 dialog.dispose();

35 } catch(ValidationException e){

36 log.error(e);

37 new EvilError(StartFrame.getCurrentInstance(),

38 true,e).setVisible(true);

39 }

40 }

Na linha 17 têm-se o *panel* do qual valida e é representado graficamente ao usuário o objeto produto. Na linha 18 é o objeto de *logs* criado a partir de uma instancia da classe *Logger* do Log4j, já comentado anteriormente.

O método *doAfterClick* na linha 25 é o que será chamado quando for clicado em salvar no *JDialogProduto.* Esse método, na linha 28, tentará pegar o produto do *panel*, do qual será validado, e caso passe pela validação, a instancia criada na linha 29 de *ProdutoHome* irá persistir esse objeto. Então na linha 32 será mostrado uma mensagem ao usuário.

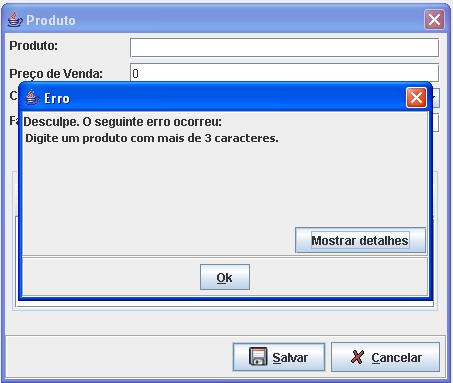


Figura 22: Exemplo da mensagem de erro de validação

Mas caso ocorra erro de validação, será lançado a exceção e a linha de execução do programa irá cair para o tratamento de erro, que está na linha 35. Será criado logs do erro, e mostrado o erro ao usuário utilizando uma tela genérica de erros dos sistema, da qual foi utilizada em toda a aplicação. Essa tela é derivada de *JDialog* e não faz *logs* automaticamente, uma vez que podem existir lugares que seja necessário *customizar* os *logs*, e outros onde é necessário mostrar o erro ao usuário mas nenhum *log* ser criado.

## Criação de logs

A configuração do Log4j pode ser feita em XML, ou em um arquivo nomeado *log4j.properties* que deve ficar no classpath ou na pasta raiz das classes da aplicação.

Abaixo mostra como está configurado atualmente o Log4j para o projeto:

1 log4j.rootCategory=INFO, A1

2 log4j.appender.A1=org.apache.log4j.ConsoleAppender

3 log4j.appender.A1.layout=org.apache.log4j.PatternLayout

4 log4j.appender.A1.layout.ConversionPattern=%-5p - %m%n

A linha um seta a categoria de prioridade raiz para Informação e seu único *appender* para A1. Na linha dois seta o *Appender* para o console do sistema. Na linha três e quatro, configura o padrão com que as mensagens serão mostradas.

Para fazer com que uma classe fique com suporte a este tipo de *logs* basta criar um atributo do tipo org.apache.log4j.Logger, e após criá-la, chamar os métodos que fazem a escrita dos *logs*:

16 public class SalvarProdutoListAdapter implements SavePressListener {

17 private JPanelProduto panel;

18 private static Logger log = Logger.getLogger(this.getClass());

19 private JDialogProduto dialog;

20

21 public SalvarProdutoListAdapter(JDialogProduto \_dialog) {

22 panel = \_dialog.getPanelProduto();

23 this.dialog = \_dialog;

24 }

25

26 public void doAfterClick() {

27 Produto prod;

28 try {

29 prod = panel.getProduto();

30 ProdutoHome home = new ProdutoHome();

31 home.saveOrUpdate(prod);

32 home.closeSession();

33 new EvilInformation(panel).showMessage("Salvo com sucesso!");

34 dialog.dispose();

35 } catch (ValidationException e) {

36 log.error(e);

37 new EvilError(StartFrame.getCurrentInstance(), true,

38 e).setVisible(true);

39 }

40 }

41 }

Na linha é instanciado o *logger* estático para a classe. Estático para que exista apenas um para essa classe em toda a aplicação. Pois a medida que vários objetos instância dessa classe sejam criados, não há necessidade de se ter mais objetos responsáveis pelos *logs*. Ao passo que der uma exceção, a linha 36 fará com que os *logs* sejam escritos em algum lugar, no caso, o console, e então a linha de execução cairá para os métodos seguintes.

## Testes unitários com JUnit

Testes unitários são uma técnica muito empregada em *eXtreme* *Programming*.

Se fosse feito os testes na maneira convencional no software desenvolvido, teria de cada vez fazer *login*, ir para a tela esperada, executar as seqüencias de cliques e digitações, fazendo tudo de maneira manual o teste da parte que o necessita para saber se há *bugs* ou não. E cada vez que fosse alterado alguma fração de código, todos os testes teriam que ser repetidos. Com JUnit, basta rodar novamente a suíte de testes e analisar os resultados.

Abaixo segue parte do TestCase da classe MovimentoEstoque:

26 public class MovimentoEstoqueHomeTest extends TestCase{

27 public MovimentoEstoqueHomeTest(String sTestName){

28 super(sTestName);

29 }

30

31 public void testFindById(){

32 MovimentoEstoqueHome h = new MovimentoEstoqueHome();

33 MovimentoEstoque mov =

34 h.findById(new MovimentoEstoqueHome().

35 findAll().get(0).getId());

36 assertNotNull(mov);

37 }

Na linha 36, espera-se que o valor da variável *mov* não seja nulo, e caso for, não irá passar no teste.

Para rodar esse teste, ele foi colocado na suíte de testes, que depois, passada o *TestRunner* faz com que seja rodado. O JUnit através de *reflection* irá chamar todos os métodos de teste do *test case* que tenham sido declarados com o prefixo “test”.

Abaixo a classe de suíte de testes do projeto, do qual todos os *Test Cases* estão adicionados:

38 public class AllTests{

39

40 public static Test suite(){

41 TestSuite suite;

42 suite = new TestSuite("AllTests");

43 suite.addTestSuite(UsuarioHomeTest.class);

44 suite.addTestSuite(ApplicationTest.class);

45 suite.addTestSuite(StartFrameTest.class);

46 suite.addTestSuite(JPanelAdminUsuariosTest.class);

47 suite.addTestSuite(NewUserActionAdapterTest.class);

48 suite.addTestSuite(HibernateRootTest.class);

49 suite.addTestSuite(ProdutoHomeTest.class);

50 suite.addTestSuite(PessoaHomeTest.class);

51 suite.addTestSuite(PanelPesquisaProdutoBuilderTest.class);

52 suite.addTestSuite(JDialogProdutoTest.class);

53 suite.addTestSuite(ContaReceberHomeTest.class);

54 suite.addTestSuite(ContaPagarHomeTest.class);

55 suite.addTestSuite(MenuFinancasFactoryTest.class);

56 suite.addTestSuite(MenuCadastroFactoryTest.class);

57 suite.addTestSuite(MenuSegurancaFactoryTest.class);

58 suite.addTestSuite(JDialogFindPessoaTest.class);

59 suite.addTestSuite(JDialogContaPagarTest.class);

60 suite.addTestSuite(FindPessoaTest.class);

61 suite.addTestSuite(JDialogContaReceberTest.class);

62 suite.addTestSuite(MovimentoEstoqueHomeTest.class);

63 suite.addTestSuite(BotaoFindPessoaTest.class);

64 suite.addTestSuite(JDialogFindProdutoTest.class);

65 suite.addTestSuite(JDialogMovimentoEstoqueTest.class);

66 suite.addTestSuite(BirtEngineTest.class);

67 suite.addTestSuite(MenuRelatoriosFactoryTest.class);

68 suite.addTestSuite(TipoContaHomeTest.class);

69 suite.addTestSuite(CategoriaProdutoHomeTest.class);

70 return suite;

71 }

72

73 public static void main(String[] args){

74 String args2[] = { "-noloading", "test.AllTests" };

75 TestRunner.main(args2);

76 }

77 }

No método construtor da classe, os testes são adicionados, e no método *main* da mesma, linha 73, é passado qual o nome da classe através do atributo *String* na segunda posição do *array*, para que o *classloader* carregue-a e execute para que os testes sejam iniciados.

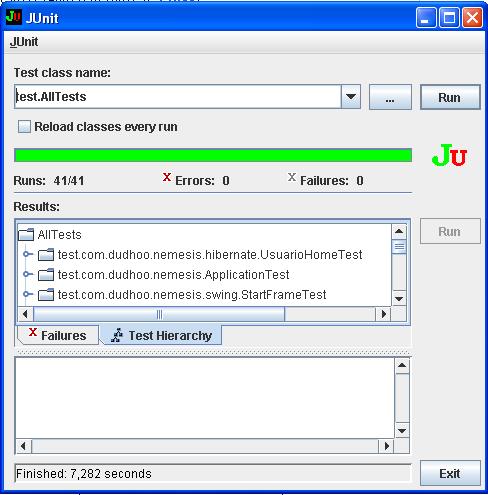


Figura 23: Tela do TestRunner em Swing do JUnit

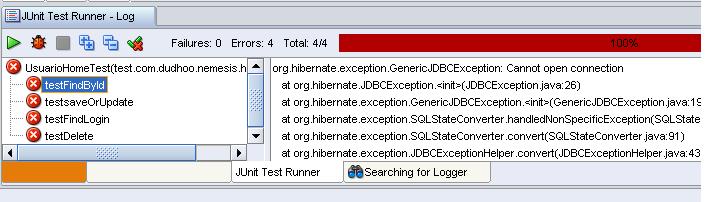
Nota-se que a barra está totalmente verde, logicamente, não houve erros. Se tivesse sido encontrado um erro ou mais, a barra estaria totalmente na cor vermelho, e informações sobre o mesmo seriam mostradas.

O Oracle JDeveloper possui a também a parte de testes de software, e é possível rodar os testes através da interface dele, tanto a suíte quanto cata teste separadamente.

Figura 24: TestRunner de uma extensão oficial do JDeveloper com erros sendo mostrados

Na figura 24 mostra os erros que são encontrados quando o banco de dados é interrompido momentaneamente enquanto o teste é executado.

## 



## Eclipse BIRT

## 

Com a ferramenta *report* *designer* do *BIRT*, foi criado uma biblioteca com a conexão e as consultas. Então os relatórios criados em seguida importavam os dados dessa biblioteca, uma vez que sempre será usada a mesma conexão com o banco e os mesmos conjunto de dados.

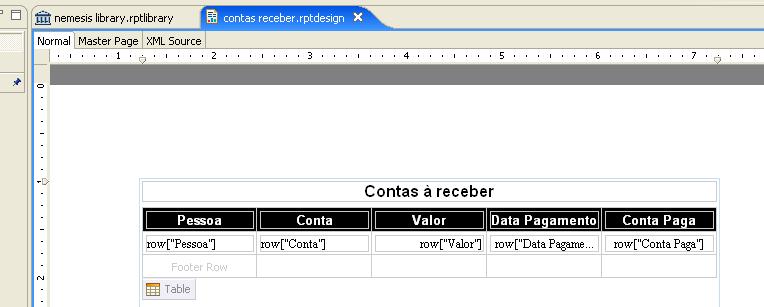


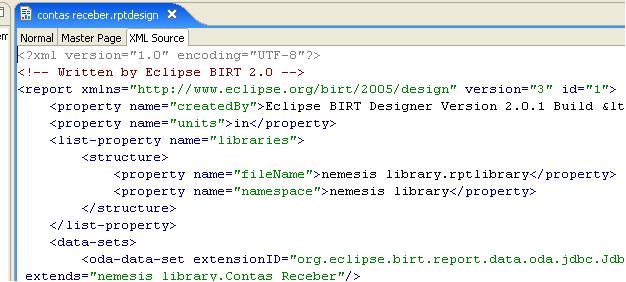
Figura 25: Report Designer do Eclipse Birt

Para a criação da biblioteca, seta-se o driver JDBC, o caminho do banco de dados e um nome para a fonte de dados, para que depois essa conexão JDBC possa ser referenciada. Após a criação da fonte de dados, deve ser criado os *DataSet*, ou conjunto de dados. Para criar um *DataSet* pega*-*se uma consulta sql do banco de dados que traga os dados que o relatório precise mostrar.

Então cria-se um relatório propriamente dito, arrastando com o *mouse* o campo do *DataSet* que seja interessante que apareça no relatório, sendo possível depois aplicar corres, estilos gráficos e qualquer formatação que seja necessária. O BIRT também possui uma grande quantidade de funções, inclusive *JavaScripts*, que podem eventualmente serem chamados quando o relatório for gerado.

O *report* *designer* na verdade é apenas uma interface para a construção do relatórios dos quais as estruturas baseiam-se em XML. Sem o *design reporter* é totalmente possível fazer os *layouts*, mas é necessário ter conhecimento das *tags* XML, do qual é bem difícil devido a quantidade de *tags* e atributos disponíveis.

Figura 24: Tela do Birt Report Designer

Figura 25: XML que é o source do relatório que foi feito através do Birt Designer

Para colocar disponíveis relatórios na aplicação, foi necessário criar uma pasta onde ficariam todas as bibliotecas para que a *engine* do BIRT às encontre. Ao iniciar a engine de relatórios, o BIRT é configurado, e então pode ser chamado os métodos para que os relatórios sejam assim criados. No caso do aplicativo desenvolvido, pode ser tanto no formato *pdf* ou *html*.

O BIIRT também possui um *web viewer* para visualizar os relatórios tanto em J2EE quanto J2SE. Nesse caso para visualizar é um pouco diferente. Precisaria acessar a URL passando como parâmetro o caminho do arquivo de relatório. Com essa alternativa de *web viewer*, poderia ser colocado-o em um servidor J2EE para fazer o serviço de processamento dos relatórios, e os clientes apenas acessariam a URL e seria passado por parâmetros seus relatórios para serem visualizados.

# SISTEMA

Neste capítulo será apresentado algumas das funcionalidades do sistema com suas respectivas telas.

## Cadastro de Pessoas

A tela de cadastro de pessoas é dividida em abas. Caso o cadastro da pessoa seja direcionado à uma pessoa jurídica, a primeira aba é construída voltada a este tipo de pessoa, caso contrário, é voltada à pessoa física.

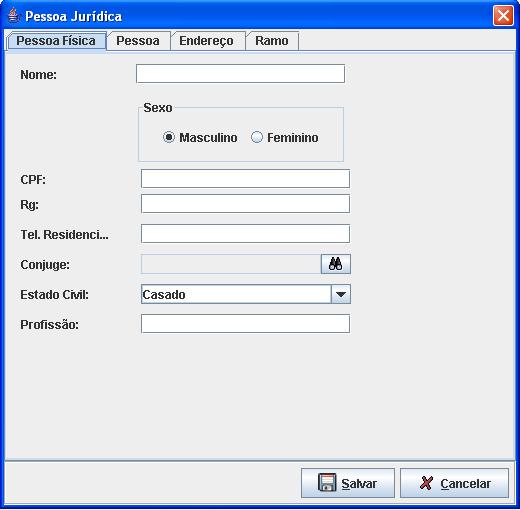


Figura 26: Tela de inclusão de pessoas

Na figura 26 observa-se a primeira aba da tela de inclusão de pessoas físicas. A primeira aba contém todos os atributos específicos à pessoa física como sexo, profissão, cônjuge, que é procurado através de outra tela quando clicado no ícone de pesquisa à direita do campo.

Quando o cadastro é de uma pessoa jurídica, a primeira aba da tela muda, conforme mostrado na figura 27.

Figura 27: Inclusão de pessoa jurídica

As outras abas são iguais nos dois tipos de objetos, mostrados nas imagens 28, 29 e 30.

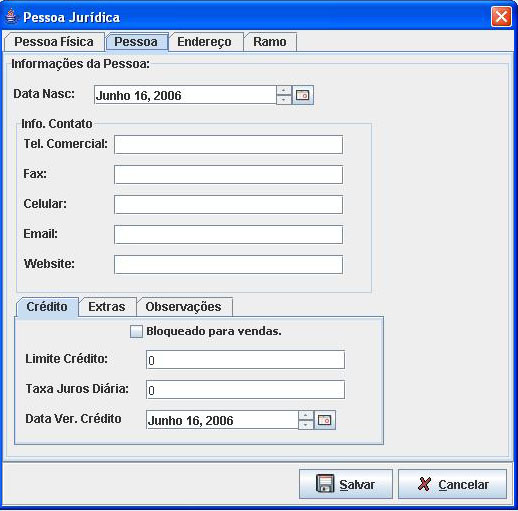


Figura 28: Aba “Pessoa” da tela de cadastros

A aba pessoa contém informações comuns entre os objetos PessoaFisica e PessoaJurídica. Basicamente representa o objeto Pessoa do qual os outros dois objetos herdam.

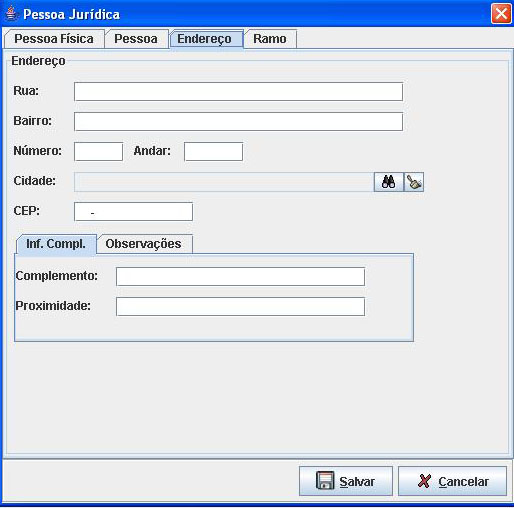


Figura 29: Aba “Endereço” da tela de cadastros

A aba endereço também contém atributos que PessoaFísica e PessoaJurídica possuem indiretamente, pois o Endereço é uma propriedade ou um atributo da classe pai Pessoa. Para setar a cidade que pertencerá o endereço, é necessário que seja clicado no ícone “binóculo” à direita do campo cidade. Dessa maneira, não ficaria lento o carregamento da tela, o que aconteceria se fosse um componente *combo* populado, e assim, à medida que o sistema crescesse e o número de cidades disponíveis aumentasse vertiginosamente.



Figura 30: Aba “Ramo” da tela de cadastros

E sendo concluído a tela de cadastro de pessoas, têm-se a aba Ramo, que representa o ramo de negócio de atuação da pessoa cadastrada.

Observa-se na figura 30 que para selecionar o ramo da pessoa, antes é necessário procurar clicando no ícone à direita do campo. Existe um botão mais à direita do campo, com um ícone “pincel”, do qual tem a função de limpar o campo quando clicado. A mesma função que possui o botão da figura 29.

## Cadastro de produtos

O cadastro e edição de produto é mostrado na figura 31:

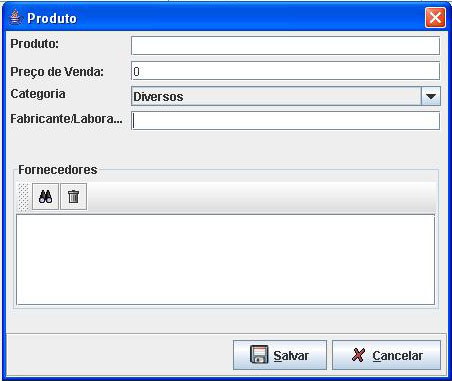


Figura 31: Tela de cadastro e edição de produto

Como mostra a figura 31, na tela de cadastro de produto, pode ser escolhido a categoria em um componente do tipo *combo*. A categoria deve ser antes cadastrada para que aqui possa ser selecionada.

Fornecedores também podem ser adicionados para referenciar o produto, clicando no botão com ícone “binóculo”, acima do componente do tipo lista. Ao clicar no botão, uma janela *modal* do tipo pesquisa irá aparecer e será possível selecionar o fornecedor entre o cadastro de pessoas.

## Vendas

Para efetuar uma venda, ao clicar no menu, uma tela aparecerá conforme mostra a figura 32:

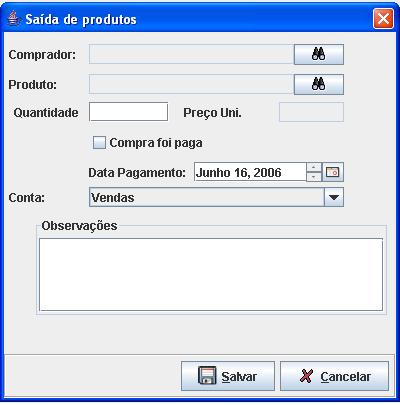


Figura 32: Tela de venda de um produto

Na figura 32, tem-se a tela de venda de produtos, sendo possível escolher o comprador, o produto e a quantidade. O valor do preço unitário será automaticamente incluso conforme o produto foi cadastrado. Escolhe-se a data do pagamento, seta-se no componente do tipo *check box* se a compra foi paga à vista, e a categoria da conta que vai ser gerada. A conta deve já constar no cadastro de contas, e somente aparecerão no *combo* as contas de tipo crédito, ou seja, há uma entrada no caixa. Ao clicar em salvar, será efetuado a venda e uma conta à receber será automaticamente gerada. Cada venda efetuada será atribuída ao usuário que está logado no momento.

## Entrada de estoque

A entrada de mercadorias para venda é feita conforme mostra a figura 33:



Figura 33: Tela de compras (entrada de mercadorias)

Observa-se na figura 33 que pode ser informado o fornecedor da mercadoria, o produto, a quantidade e o preço unitário da mesma. Ainda é possível informar dados sobre a transação financeira, como se a compra foi paga à vista, a data do pagamento e o tipo da conta para que seja gerado no contas à pagar corretamente.

## Gerenciamento de usuários

O gerenciamento de usuários é feito através da tela mostrada na figura 34:

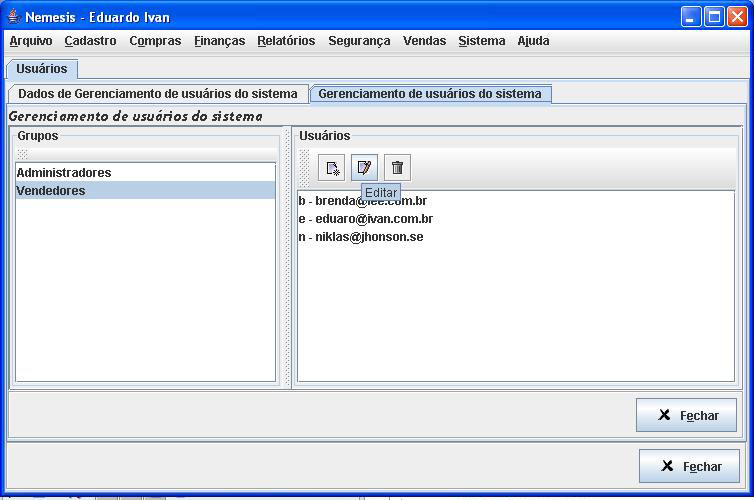


Figura 34: Tela de Visualização e gerenciamento de usuários e grupos

Conforme a figura 34 mostra, é possível criar, editar e excluir informações de *login* e pessoais de um usuário relacionado ao grupo selecionado à esquerda.

## Relatórios

Os relatórios podem ser feitos em HTML ou PDF, e são abertos em uma aplicação externa, podendo ser um *browser*, ou o Adobe *Reader*. Para visualizar um relatório basta clicar no respectivo menu demonstrado na figura 35:

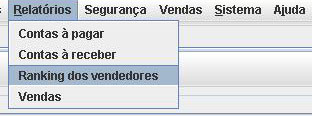


Figura 35: Menu de relatórios

Ao clicar em um dos itens do menu mostrado na figura 35, será visualizado um relatório como mostra a figura 36 e 37.



Figura 36: Relatório com gráficos representando o ranking dos vendedores

Na figura 36 é possível observar o desempenho dos vendedores, primeiramente em forma de gráfico de barras e logo abaixo em forma de tabela. Cada vendedor é um usuário distinto no sistema.

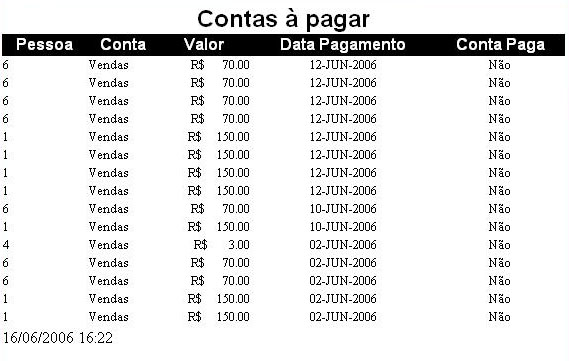


Figura 37: Relatório de contas à pagar

A figura 37 mostra o relatório em formato HTML de contas à pagar em forma de tabela. Gerado após o clique no item “contas à pagar” do menu relatórios que foi mostrado na figura 35.

## Web services

O Oracle JDeveloper tem o Oracle *Application* Server 10*g[[3]](#footnote-3)* embutido na IDE[[4]](#footnote-4), do qual *possui* interfaces *web* para que os *Web* *services* sejam visualizados e também para que métodos seja chamados, mostrando os objetos retornados em SOAP.



Figura 38: Métodos expostos pelo webservice no Oracle Application Server 10g

Na figura 38 observa-se alguns métodos expostos como serviço na *web*. Quando clicado no método *getAllProdutos* é mostrado a tela da figura 39.

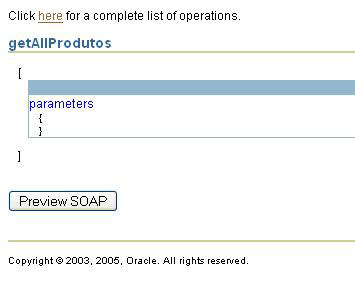


Figura 39: Método getAllProdutos no Oracle Application Server 10g

Quando clicado no botão “*Preview* *SOAP*”, mostrado na figura 39, o serviço será chamado e mostrado outra tela como mostra a figura 40.

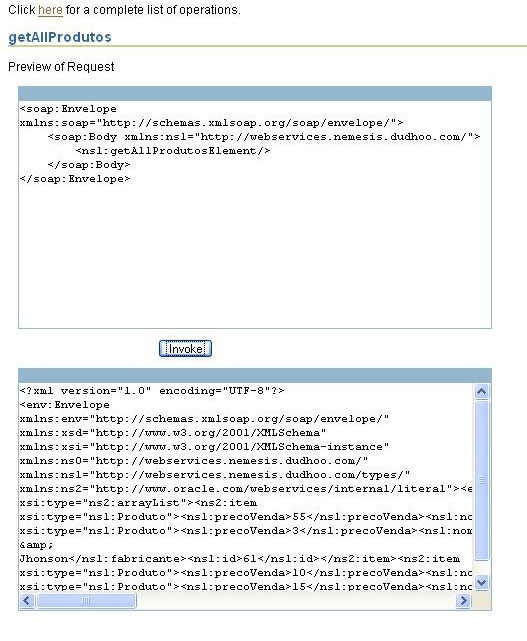


Figura 40: Objetos retornados em formato SOAP

A figura 40 mostra os objetos em formato SOAP na área de texto mais abaixo, sendo retornados após o botão “*Invoke*” ser clicado. O serviço *web* foi executado.

# CONCLUSÃO

Quando foi iniciado o projeto, não tinha-se a intenção nenhuma de se fazer algum software sem custos de licenças. O resultado mostrou que é possível desenvolver software avançado de qualidade apenas usando tecnologias sem custos com licenças. A única exceção seria o banco de dados Oracle, mas como foi dito, a versão suporte até 4 gigabytes sem custo, o que abrangeria uma empresa relativamente com condições financeiras para comprar uma licença caso fosse necessário, além do mais, com o Hibernate torna-se fácil migrar tudo o que existe para um banco de dados confiável, como o Postgres por exemplo.

Java deu muita possibilidade para a escolha de frameworks, que as vezes se tornava até uma preocupação, pois sempre era pensado em usar o melhor framework possível, e com uma gama tão rica de possibilidades, as vezes tornava a escolha muito difícil. Enquanto ao longo dos meses o software é construído, sai muitos rumores e tecnologias novas que de certa forma as o que está sendo usado no desenvolvimento do software torna-se ultrapassado, mas isso não significa que tenha que ser voltado atrás, pois sempre surgirão tecnologias novas e se for sempre mudando, o software nunca chegará ao final.

O JDeveloper mostrou-se uma ferramenta fantástica, difícil até de compreender que algo tão bom e poderoso seja gratuito. O tanto é que o projeto que foi iniciado com o desenvolvimento no Ja*va Studio Enter*prise, e após centenas de classes codificadas, foi constatado que valeria a pena migrar, pela qualidade que a IDE da Oracle oferecia. *NetBeans* e *Java Studio Ent*erprise se apresentaram bons e gratuitos, mas o JDeveloper superou-os pela quantidade insignificante de bugs, boa velocidade na parte de design gráfico para fazer as telas do sistema, e pelo ciclo completo do desenvolvimento de software que ele faz.

Os design patterns são muito bons para ser resolver problemas, uma vez que é dado a certeza que se está programando da forma mais elegante e testada possível.

Com o software totalizado e o conhecimento completo de sua estrutura, da-se a falsa impressão de que pouca coisa foi feita ao longo do projeto, mas sendo pegado os números e analisado, constata-se que bastante coisa foi produzida: 300 classes codificadas, logicamente com quantidades variáveis de linhas. Esse número teria sido muito maior sem o uso do Hibernate e das 83 bibliotecas jar de terceiros, livres ou não, direta e indiretamente.

A conclusão final é que com Java, uma boa IDE, e tecnologias como Web services, ORM, e outros *frameworks* disponíveis, é possível fazer softwares avançados de boa qualidade. Mas antes é necessário escolher rapidamente quais *frameworks* usar e fazer o planejamento coerente da arquitetura envolvida no software à ser projetado.

# Bibliografia

* **[1] Java.com – Hot Games, Cool Apps! -** <http://www.java.com/pt_BR/about/> **-** Acessado em janeiro de 2006.
* **[2] Javafree.org - A voz java do brasil!-** [http://www.javafree.org/content/view.jf?idContent=86](http://www.java.com/pt_BR/about/) **-** Acessado em dezembro de 2005.
* **[3] JDeveloper - Offical Home Page -** [http://www.oracle.com/technology/products/jdev/index.html](http://www.java.com/pt_BR/about/) **-** Acessado em junho 2006.
* **[4] Wiki do World Wide Web Consortion W3C -** [http://esw.w3.org/topic/Web services](http://www.java.com/pt_BR/about/)  **-** Acessado em dezembro 2005.
* **[5] IBM – developerWorks, SOA and Web services** - <http://www-128.ibm.com/developerworks/webservices/> - Acessado em dezembro de 2005.
* **[6] Java Developers Network – Web services Essentials -** <http://developers.sun.com/techtopics/webservices/> - Acessado em junho de 2006.
* **[7] Java Developers Network – Overview About WSDL –** [http://developers.sun.com/sw/building/tech\_article/](http://developers.sun.com/techtopics/webservices/) - Acessado em junho de 2006.
* **[8] UDDI.org –** <http://uddi.org/> - Acessado em junho de 2006.
* **[9] Eclipse BIRT Home –** <http://www.eclipse.org/birt/> - Acessado em abril de 2006.
* **[10] JUnit.org - [http://www.junit.org](http://www.junit.org/)** - Acessado em abril de 2006.
* **[11]**  **Oracle Database 10g Express Edition** - <http://www.oracle.com/technology/products/database> - Acessado em fevereiro de 2006.
* **[12] Hibernate.org -** <http://www.hibernate.org/> **-** Acessado em fevereiro de 2006.
* **[13] BAUER, Christian; KING, Gavin.** “*Hibernate in Action”***.** Manning. Greenwich, 2005.
* **[14] Jakarta Commons** - <http://jakarta.apache.org/commons/index.html> – Acessado em maio de 2006.
* **[15] Java World - Simply Singleton** - <http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html> – Acessado em abril de 2006.
* **[15] BLOCH, Joshua.** **Effective Java: Programming Language Guide.** Addison Wesley, Canadá, 2001.

1. CRUD é um acrônimo para as tarefas de  *Create, Read, Update,* e *Delete..* [↑](#footnote-ref-1)
2. Metadata são meta dados. Ao pé da letra dados representando dados. Em Java são as informações sobre o banco de dados, sua estrutura, colunas e relacionamentos. [↑](#footnote-ref-2)
3. Servidor de aplicações pago e certificado e compatível com J2EE. [↑](#footnote-ref-3)
4. Acrônimo de *Integrated Development Environme*nt, em português Ambiente Integrado de desenvolvimento. [↑](#footnote-ref-4)