

**LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE**

Josinai de Moraes Barbosa

**SISTEMA DE FOLHA DE PAGAMENTO:
UMA IMPLEMENTAÇÃO COM AS TECNOLOGIAS JAVA E HIBERNATE**

**NATAL
2008**

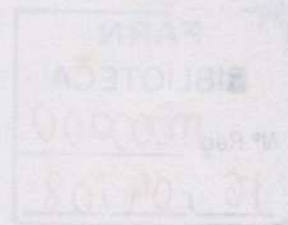
JOSINAI DE MORAIS BARBOSA

**SISTEMA DE FOLHA DE PAGAMENTO:
UMA IMPLEMENTAÇÃO COM AS TECNOLOGIAS JAVA E HIBERNATE**

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do grau de Especialista.

Orientador:
Ricardo Wendell, MSc.

Natal
2008



UMA IMPLANTAÇÃO COM AS TECNOLOGIAS JAVA E HIBERNATE
SISTEMA DE FOLHA DE PAGAMENTO

Monografia apresentada ao Curso de
Especialização em Desenvolvimento de
Sistemas Corporativos da Faculdade
Natalense para o Desenvolvimento do Rio
Grande do Norte como requisito
obrigatório para obtenção do grau de

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Barbosa, Josinai de Moraes

Sistema de folha de pagamento: uma implantação com as tecnologias
Java e Hibernate / Josinai de Moraes Barbosa. – Natal: FARN, 2007.
72f.

Orientador: Ricardo Wendell (M.Sc.)

Monografia (Especialização em Desenvolvimento de Sistemas
Corporativos) – Faculdade Natalense para o Desenvolvimento do Rio
Grande do Norte.

1. Linguagem de Programação – Monografia. 2. Desenvolvimento de
Sistema – Monografia. 3. Unified Modeling Language (UML) – Monografia.
4. Java – Monografia. I. Título.

RN/FARN/BC

CDU 004.43

00443
B198a
2008
ex 01

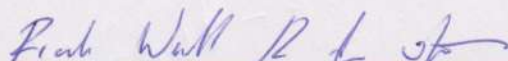


JOSINAI DE MORAIS BARBOSA

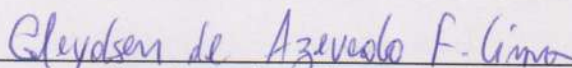
SISTEMA DE FOLHA DE PAGAMENTO:
UMA IMPLEMENTAÇÃO COM AS TECNOLOGIAS JAVA E HIBERNATE

Palavras-chave: Folha de Pagamento, Análise de sistemas, JAVA, Spring, Hibernate, UML

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, aprovada em 03 de março de 2008, pela banca examinadora composta pelos seguintes membros:



Ricardo Wendell, MSc.
Orientador



Gleydson de A. F. Lima, MSc.
Membro Examinador

RESUMO

Visando a conclusão do curso de Especialização em Desenvolvimento de Sistemas Corporativos, a presente monografia faz uma análise e desenvolvimento de um sistema de folha de pagamento, que é um sistema muito útil e necessário em todos os escritórios de contabilidade e nos departamentos de setor pessoal de grandes empresas. Para tanto, foi feito um estudo sobre as etapas e os cálculos necessários para a realização de uma folha de pagamento, e foram empregados métodos e tecnologias ensinadas durante todo o curso de especialização, para a sua modelagem e implementação. Dentre as tecnologias disponíveis, são analisados o emprego da linguagem UML para a modelagem do sistema, a plataforma Java com o uso do *framework Swing* para o desenvolvimento de interfaces gráficas para *desktop* e o mapeamento e persistência dos dados através do *framework Hibernate*. O texto se inicia com uma explicação sobre essas tecnologias, depois fala sobre a análise e modelagem do sistema para enfim concluir com a explicação de como foi implementado o programa.

Palavras-chave: Folha de Pagamento. Análise de sistemas. JAVA. Swing. Hibernate. UML.

LISTA DE FIGURAS ABSTRACT

Aiming the conclusion of the Enterprise Systems Development Specialization Course, this paper presents an analysis and development of a payroll system, which is a system very useful and necessary in all accounting offices and personnel department of large companies. Therefore, a study was done on the steps and calculations required for the creation of a payroll, and were used methods and technologies, taught during the course, for their modeling and implementation. Among the technologies available, are analysed the use of UML language for system modeling, the Java platform using the Swing framework for the development of desktop graphical interfaces and the mapping and persistence of data through the framework Hibernate. The text begins with an explanation of these technologies, and after that focus on the analysis and modeling of the system to finally conclude with the explanation on how the system was implemented.

Keywords: Payroll. System Analysis. JAVA. Swing. Hibernate. UML.

Figura 16 – Exemplo de um refinamento	21
Figura 17 – Exemplo de compilação	23
Figura 18 – Hierarquia de classes Drawing	26
Figura 19 – Arquitetura do MS-Excel	30
Figura 20 – Tabela de aliquotas de INSS e salário família	36
Figura 21 – Tabela de aliquotas do IRPF	37
Figura 22 – Diagrama de classes do sistema	39
Figura 23 – Diagrama de casos de uso do sistema	40
Figura 24 – Tela principal do sistema	41
Figura 25 – Tela de cadastro de empresas	41
Figura 26 – Tela para cadastro de funcionários	42
Figura 27 – Tela para cadastrar folha	42
Figura 28 – Tela para cadastrar lançamentos	43
Figura 29 – Tela para visualizar contra-cheque	43
Figura 30 – DER do sistema	45
Figura 31 – Tela para listar empresas	57
Figura 32 – Tela para excluir folhas	59

LISTA DE ILUSTRAÇÕES

Figura 1 – Notação do diagrama de casos de uso	11
Figura 2 – Exemplo de diagrama de casos de uso	11
Figura 3 – Exemplo de diagrama classes	12
Figura 4 – Exemplo de classe	14
Figura 5 – Exemplo de associação normal	15
Figura 6 – Exemplo de associação recursiva	15
Figura 7 – Exemplo de associação de classe	16
Figura 8 – Exemplo de associação ternária	16
Figura 9 – Exemplo de agregação	17
Figura 10 – Exemplo de agregação compartilhada	17
Figura 11 – Exemplo de agregação de composição	18
Figura 12 – Exemplo de generalização normal	18
Figura 13 – Exemplo de uma generalização de sobreposição	19
Figura 14 – Exemplo de generalização completa	20
Figura 15 – Exemplo de uma dependência entre classes	20
Figura 16 – Exemplo de um refinamento	21
Figura 17 – Exemplo de compilação	23
Figura 18 – Hierarquia de classes do swing	26
Figura 19 – Arquitetura do <i>Hibernate</i>	30
Figura 20 – Tabela de alíquotas do INSS e salário família	36
Figura 21 – Tabela de alíquotas do IRRF	37
Figura 22 – Diagrama de classes do sistema	39
Figura 23 – Diagrama de casos de uso do sistema	40
Figura 24 – Tela principal do sistema	41
Figura 25 – Tela de cadastro de empresas	41
Figura 26 – Tela para cadastro de funcionários	42
Figura 27 – Tela para cadastrar folha	42
Figura 28 – Tela para cadastrar lançamentos	43
Figura 29 – Tela para visualizar contra-cheque	43
Figura 30 – DER do sistema	45
Figura 31 – Tela para listar empresas	57
Figura 32 – Tela para excluir folhas	59

LISTA DE CÓDIGOS

1 INTRODUÇÃO	8
2 REFERENCIAL TEÓRICO	9
2.1 UNIFIED MODELING LANGUAGE (UML)	9
Listagem 1 – Criação de tabela exemplo	31
Listagem 2 – Código da classe Amigo	31
Listagem 3 – Código da classe Amigo com <i>Annotations</i>	32
Listagem 4 – Exemplo de arquivo XML de configuração do <i>Hibernate</i>	33
Listagem 5 – Exemplo de arquivo XML para mapeamento de classes	33
Listagem 6 – Exemplo de uma classe DAO	34
Listagem 7 – Mapeamento da classe Empresa	46
Listagem 8 – Mapeamento da classe Folha	47
Listagem 9 – Mapeamento da classe Funcionário	48
Listagem 10 – Mapeamento da classe FolhaFuncionario	49
Listagem 11 – Mapeamento da classe Folha	49
Listagem 12 – Código do botão de cadastro de funcionários	53
Listagem 13 – Código do botão de inclusão de folha	54
Listagem 14 – Código do botão adicionar da tela de lançamentos	55
Listagem 15 – Código do botão gravar da tela de lançamentos	56
Listagem 16 – Código do botão da busca de empresas	58
Listagem 17 – Código do botão de exclusão de folha	59
3 Mapeamento das classes	60
3.1 DAO	60
4 IMPLEMENTAÇÃO DOS CASOS DE USO	62
4.1 CADASTROS	62
4.2 CONSULTAS	67
4.3 EXCLUSÕES	68
4.4 INCLUSÕES	69
5 REFERÊNCIAS	71
6 ANEXOS	83

SUMÁRIO

1 INTRODUÇÃO	8
2 REFERENCIAL TEÓRICO	9
2.1 <i>UNIFIED MODELING LANGUAGE (UML)</i>	9
2.1.1 Diagramas	10
2.1.2 Diagrama de casos de uso	10
2.1.3 Diagrama de classes	12
2.1.4 Classes e relacionamentos	13
2.2 JAVA	21
2.2.1 A Linguagem Java	22
2.3 DESENVOLVIMENTO DE INTERFACES COM SWING	25
2.4 HIBERNATE	28
2.4.1 Arquitetura do <i>Hibernate</i>	29
2.4.2 Utilizando o <i>Hibernate</i>	31
3 MODELAGEM DO SISTEMA	35
3.1 CÁLCULOS DA FOLHA DE PAGAMENTO	35
3.2 DIAGRAMA DE CLASSES	38
3.3 DIAGRAMA DE CASOS DE USO	40
3.4 PROTÓTIPOS DE INTERFACE	41
4 MAPEAMENTO OBJETO RELACIONAL	44
4.1 BANCO DE DADOS	44
4.2 MAPEAMENTO DAS CLASSES	46
4.3 DAO	50
5 IMPLEMENTAÇÃO DOS CASOS DE USO	52
5.1 CADASTROS	52
5.2 CONSULTAS	57
5.3 EXCLUSÕES	58
6 CONCLUSÕES	60
REFERÊNCIAS	61
APÊNDICES	63

1 INTRODUÇÃO TEÓRICO

Visando a conclusão do curso de especialização de sistemas corporativos, foi realizada uma análise, desenvolvimento e implementação de um sistema de folha de pagamento simplificado. O problema da elaboração de uma folha de pagamento em si é que devido ao grande número de cálculos que devem ser feitos para se poder preencher todos os formulários da área trabalhista, fica muito custoso e demorado fazer tudo à mão. Além do fato de que a legislação do trabalho é farta e é alterada continuamente, dificultando ainda mais a realização desta tarefa. Adicione a estas dificuldades a necessidade da realização desta folha de pagamento todo o mês.

Temos então um problema que pode ser resolvido com a implementação de um sistema informatizado, que faria todos os cálculos necessários e evitaria a repetição das mesmas operações todo mês. Para realizar o desenvolvimento deste sistema, foram utilizadas diversas técnicas e metodologias adquiridas durante a pós-graduação.

Inicialmente foi feito um estudo do sistema a ser desenvolvido e em seguida uma análise orientada a objetos do mesmo, sendo documentada com diagramas em *Unified Modeling Language (UML)*. A seguir modelou-se o banco de dados para o armazenamento persistente das informações do sistema e fazendo-se do uso da tecnologia *Hibernate* foi feito o mapeamento objeto-relacional do sistema. Após esse mapeamento, implementou-se as regras do negócio utilizando a linguagem de programação *JAVA* numa plataforma para *desktop* obtendo o sistema idealizado.

No próximo capítulo teremos o referencial teórico utilizado para a realização deste projeto, e logo a seguir uma explicitação e modelagem do sistema a ser desenvolvido. No quarto capítulo teremos uma melhor explicitação do mapeamento objeto-relacional utilizado. O quinto capítulo traz a como a implementação realizada e por último as conclusões deste relatório.

2 REFERENCIAL TEÓRICO

Neste capítulo abordaremos algumas das tecnologias utilizadas para a realização do projeto.

2.1 UNIFIED MODELING LANGUAGE (UML)

O desenvolvimento de sistemas de software de grande porte é suportado por métodos de análise e projeto que modelam esse sistema de modo a fornecer para toda a equipe envolvida (cliente, analista, programador, etc.) uma compreensão única do projeto.

A UML (*Unified Modeling Language*) é o sucessor de um conjunto de métodos de análise e projeto orientados a objeto (OOA&D).

É um modelo de linguagem, não um método. O modelo de linguagem é a notação que o método usa para descrever o projeto. Corresponde ao ponto principal da comunicação. Se uma pessoa quer conversar sobre o projeto, como outra pessoa, é através do modelo de linguagem que elas se entendem.

Ela define uma notação e um meta-modelo. A notação são todos os elementos de representação gráfica vistos no modelo (retângulo, setas, o texto, etc.), é a sintaxe do modelo de linguagem. A notação do diagrama de classe define a representação de itens e conceitos tais como: classe, associação e multiplicidade. Um meta-modelo é um diagrama de classe que define de maneira mais rigorosa a notação.

A seguir abordaremos algumas de suas notações.

2.1.1 Diagramas

Os diagramas utilizados pela UML são compostos de nove tipos: diagrama de casos de uso, de classes, de objeto, de estado, de seqüência, de colaboração, de atividade, de componente e o de execução.

Todos os sistemas possuem uma estrutura estática e um comportamento dinâmico. A UML suporta modelos estáticos (estrutura estática), dinâmicos (comportamento dinâmico) e funcional. A Modelagem estática é suportada pelo diagrama de classes e de objetos, que consiste nas classes e seus relacionamentos. Os relacionamentos podem ser de associações, herança (generalização), dependência ou refinamentos. Os modelamentos dinâmicos são suportados pelos diagramas de estado, seqüência, colaboração e atividade. E a modelagem funcional é suportada pelos diagramas de componente e execução (UML Linguagem de Modelagem Unificada, 2008).

Abordaremos apenas os dois tipos de diagramas utilizados durante o projeto: diagrama de casos de uso e o diagrama de classes.

2.1.2 Diagrama de casos de uso

A modelagem de um diagrama de casos de uso é uma técnica usada para descrever e definir os requisitos funcionais de um sistema. Eles são escritos em termos de atores externos, casos de uso e o sistema modelado. Os atores representam o papel de uma entidade externa ao sistema como um usuário, um hardware, ou outro sistema que interage com o sistema modelado. Os atores iniciam a comunicação com o sistema através dos casos de uso, onde o caso de uso representa uma seqüência de ações executadas pelo sistema e recebe do ator que lhe utiliza dados tangíveis de um tipo ou formato já conhecido, e o valor de resposta da execução de um caso de uso (conteúdo) também já é de um tipo conhecido, tudo isso é definido juntamente com o caso de uso através de texto de documentação (UML Linguagem de Modelagem Unificada, 2008).

A notação usada pelo Diagrama de Casos de Uso é visto na figura 1.



Figura 1 – Notação do diagrama de casos de uso
Fonte: UML Linguagem de Modelagem Unificada (2008)

Na figura 2 temos um diagrama de casos de uso que demonstra as funções de um ator externo de um sistema de controle bancário de um banco fictício. O diagrama especifica que funções o administrador do banco poderá desempenhar. Pode-se perceber que não existe nenhuma preocupação com a implementação de cada uma destas funções, já que este diagrama apenas se resume a determinar que funções deverão ser suportadas pelo sistema modelado.

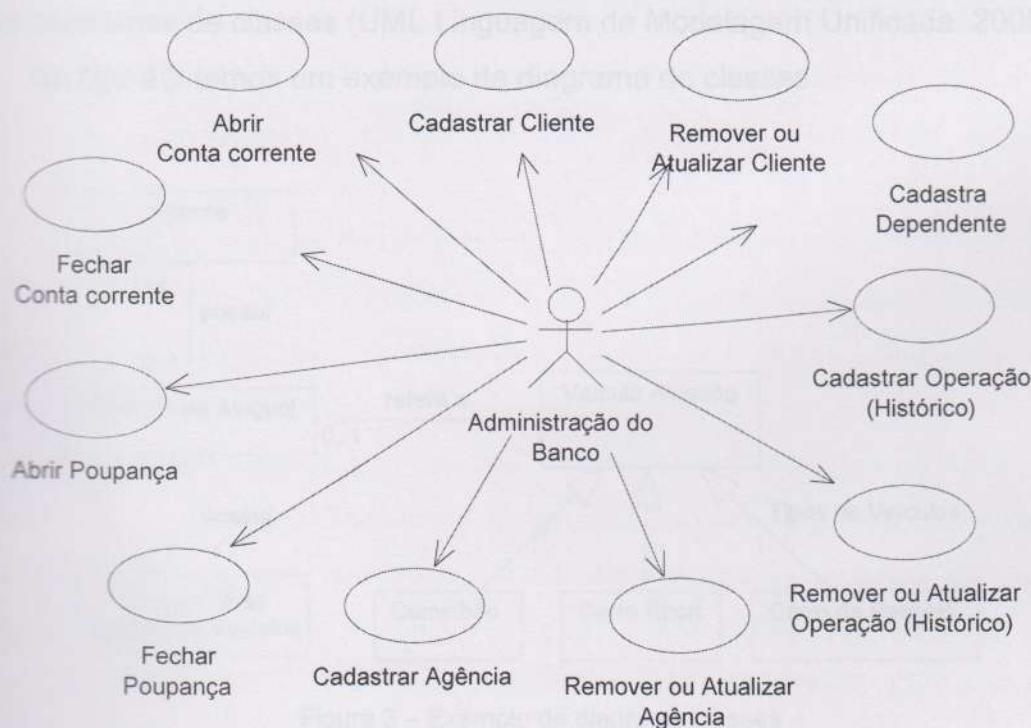


Figura 2 – Exemplo de diagrama de casos de uso
Fonte: UML Linguagem de Modelagem Unificada (2008)

2.1.3 Diagrama de Classes

O diagrama de classes demonstra a estrutura estática das classes de um sistema onde estas representam as "coisas" que são gerenciadas pela aplicação modelada. Classes podem se relacionar com outras através de diversas maneiras: associação (conectadas entre si), dependência (uma classe depende ou usa outra classe), especialização (uma classe é uma especialização de outra classe), ou em pacotes (classes agrupadas por características similares). Todos estes relacionamentos são mostrados no diagrama de classes juntamente com as suas estruturas internas, que são os atributos e operações.

O diagrama de classes é considerado estático já que a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema. Um sistema normalmente possui alguns diagramas de classes, já que não são todas as classes que estão inseridas em um único diagrama e uma certa classe pode participar de vários diagramas de classes (UML Linguagem de Modelagem Unificada, 2008).

Na figura 3 temos um exemplo de diagrama de classes.



Figura 3 – Exemplo de diagrama classes
Fonte: UML Linguagem de Modelagem Unificada (2008)

No exemplo de diagrama de classes da figura 3 temos modelado as classes de um sistema de uma locadora de veículos, onde a companhia possui vários contratos de aluguéis com vários clientes, e cada contrato de aluguel se refere a um

veículo específico, que por sua vez pode ser do tipo caminhão, carro sport ou carro de passeio.

2.1.4 Classes e Relacionamentos

Uma classe é a descrição de um tipo de objeto. Todos os objetos são instâncias de classes, onde a classe descreve as propriedades e comportamentos daquele objeto. Objetos só podem ser instanciados de classes. Usam-se classes para classificar os objetos que identificamos no mundo real. Tomando como exemplo Charles Darwin, que usou classes para classificar os animais conhecidos, e combinou suas classes por herança para descrever a “Teoria da Evolução”. A técnica de herança entre classes é também usada em orientação a objetos.

Uma classe pode ser a descrição de um objeto em qualquer tipo de sistema – sistemas de informação, técnicos, integrados real-time, distribuídos, software etc. Num sistema de software, por exemplo, existem classes que representam entidades de software num sistema operacional como arquivos, programas executáveis, janelas, barras de rolagem, etc.

Identificar as classes de um sistema pode ser complicado, e deve ser feito por especialistas no domínio do problema a que o software modelado se baseia. As classes devem ser retiradas do domínio do problema e serem nomeadas pelo que elas representam no sistema.

Em UML as classes são representadas por um retângulo dividido em três compartimentos: o compartimento de nome, que conterá apenas o nome da classe modelada, o de atributos, que possuirá a relação de atributos que a classe possui em sua estrutura interna, e o compartimento de operações, que serão os métodos de manipulação de dados e de comunicação de uma classe com outras do sistema. A sintaxe usada em cada um destes compartimentos é independente de qualquer linguagem de programação, embora podem ser usadas outras sintaxes como a do C++, Java, e etc (UML Linguagem de Modelagem Unificada, 2008).

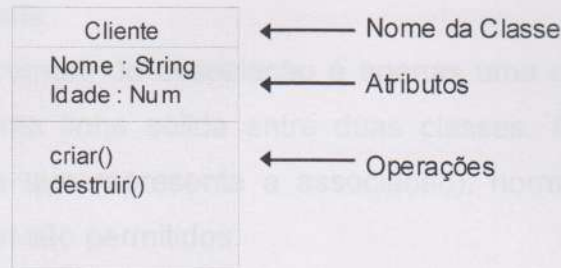


Figura 4 – Exemplo de classe
Fonte: UML Linguagem de Modelagem Unificada (2008)

Os relacionamentos ligam as classes/objetos entre si criando relações lógicas entre estas entidades. Os relacionamentos podem ser dos seguintes tipos:

- **Associação:** É uma conexão entre classes, e também significa que é uma conexão entre objetos daquelas classes. Em UML, uma associação é definida com um relacionamento que descreve uma série de ligações, onde a ligação é definida como a semântica entre as duplas de objetos ligados.
- **Generalização:** É um relacionamento de um elemento mais geral e outro mais específico. O elemento mais específico pode conter apenas informações adicionais. Uma instância (um objeto é uma instância de uma classe) do elemento mais específico pode ser usada onde o elemento mais geral seja permitido.
- **Dependência e Refinamentos:** Dependência é um relacionamento entre elementos, um independente e outro dependente. Uma modificação é um elemento independente afetará diretamente elementos dependentes do anterior. Refinamento é um relacionamento entre duas descrições de uma mesma entidade, mas em níveis diferentes de abstração.

Abordaremos agora os tipos de relacionamento e suas respectivas subdivisões.

Associações

Uma associação representa que duas classes possuem uma ligação (link) entre elas, significando, por exemplo, que elas “conhecem uma a outra”, “estão conectadas com”, “para cada X existe um Y” e assim por diante. Classes e associações são muito poderosas quando modeladas em sistemas complexos.

Associações Normais

O tipo mais comum de associação é apenas uma conexão entre classes. É representada por uma linha sólida entre duas classes. A associação possui um nome (junto à linha que representa a associação), normalmente um verbo, mas substantivos também são permitidos.

Pode-se também colocar uma seta no final da associação indicando que esta só pode ser usada para o lado onde a seta aponta. Mas associações também podem possuir dois nomes, significando um nome para cada sentido da associação.

Para expressar a multiplicidade entre os relacionamentos, um intervalo indica quantos objetos estão relacionados no link. O intervalo pode ser de zero para um (0..1), zero para vários (0..* ou apenas *), um para vários (1..*), dois (2), cinco para 11 (5..11) e assim por diante. É também possível expressar uma série de números como (1, 4, 6..12). Se não for descrita nenhuma multiplicidade, então é considerado o padrão de um para um (1..1 ou apenas 1).

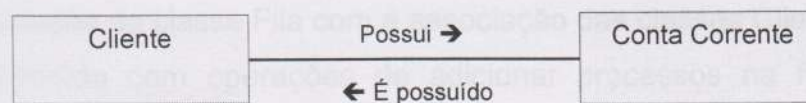


Figura 5 – Exemplo de associação normal
Fonte: UML Linguagem de Modelagem Unificada (2008)

Associação Recursiva

É possível conectar uma classe a ela mesma através de uma associação e que ainda representa semanticamente a conexão entre dois objetos, mas os objetos conectados são da mesma classe. Uma associação deste tipo é chamada de associação recursiva.



Figura 6 – Exemplo de associação recursiva
Fonte: UML Linguagem de Modelagem Unificada (2008)

Associação de Classe

Uma classe pode ser associada a uma outra associação. Este tipo de associação não é conectado a nenhuma das extremidades da associação já existente, mas na própria linha da associação. Esta associação serve para se adicionar informações extras a associação já existente.

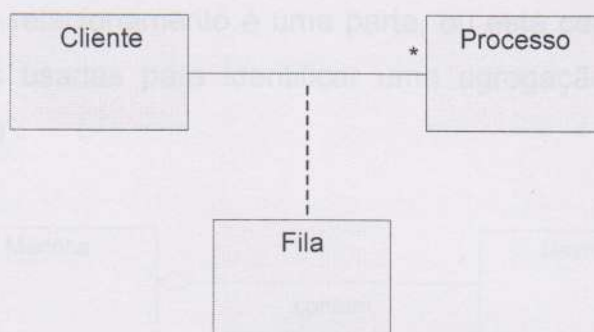


Figura 7 – Exemplo de associação de classe
Fonte: UML Linguagem de Modelagem Unificada (2008)

A associação da classe Fila com a associação das classes Cliente e Processo pode ser estendida com operações de adicionar processos na fila, para ler e remover da fila e de ler o seu tamanho. Se operações ou atributos são adicionados à associação, ela deve ser mostrada como uma classe.

Associação Ternária

Mais de duas classes podem ser associadas entre si, a associação ternária associa três classes. Ela é mostrada como uma grade losango (diamante) e ainda suporta uma associação de classe ligada a ela, traçaria-se, então, uma linha tracejada a partir do losango para a classe onde seria feita a associação ternária.

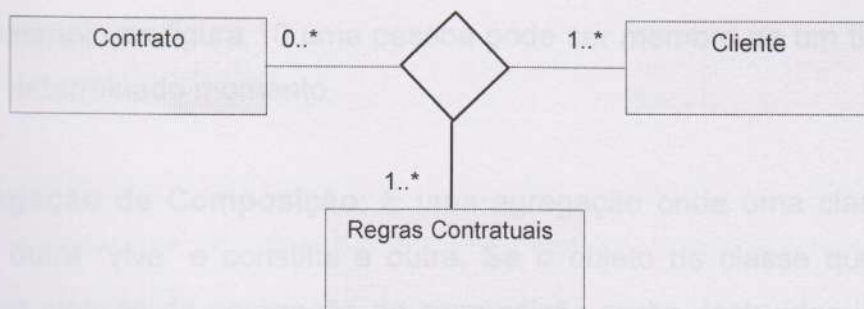


Figura 8 – Exemplo de associação ternária
Fonte: UML Linguagem de Modelagem Unificada (2008)

No exemplo da figura 8 a associação ternária especifica que um cliente poderá possuir 1 ou mais contratos e cada contrato será composto de 1 ou várias regras contratuais.

Agregação

A agregação é um caso particular da associação. A agregação indica que uma das classes do relacionamento é uma parte, ou está contida em outra classe. As palavras chaves usadas para identificar uma agregação são: “consiste em”, “contém”, “é parte de”.

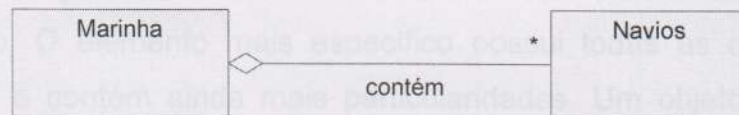


Figura 9 – Exemplo de agregação
Fonte: UML Linguagem de Modelagem Unificada (2008)

Existem tipos especiais de agregação que são as agregações compartilhadas e as compostas.

Agregação Compartilhada: É dita compartilhada quando uma das classes é uma parte, ou está contida na outra, mas esta parte pode estar contida na outra várias vezes em um mesmo momento.



Figura 10 – Exemplo de agregação compartilhada
Fonte: UML Linguagem de Modelagem Unificada (2008)

No exemplo da figura 10 uma pessoa pode ser membro de um time ou vários times e em determinado momento.

Agregação de Composição: É uma agregação onde uma classe que está contida na outra “vive” e constitui a outra. Se o objeto da classe que contém for destruído, as classes da agregação de composição serão destruídas juntamente já que as mesmas fazem parte da outra.

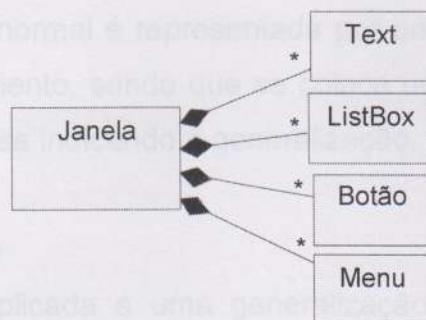


Figura 11 – Exemplo de agregação de composição
 Fonte: UML Linguagem de Modelagem Unificada (2008)

Generalizações

A generalização é um relacionamento entre um elemento geral e um outro mais específico. O elemento mais específico possui todas as características do elemento geral e contém ainda mais particularidades. Um objeto mais específico pode ser usado como uma instância do elemento mais geral. A generalização, também chamada de herança, permite a criação de elementos especializados em outros.

Existem alguns tipos de generalizações que variam em sua utilização a partir da situação. São elas: generalização normal e restrita. As generalizações restritas se dividem em generalização de sobreposição, disjuntiva, completa e incompleta.

Generalização Normal

Na generalização normal a classe mais específica, chamada de subclasse, herda tudo da classe mais geral, chamada de superclasse. Os atributos, operações e todas as associações são herdados.

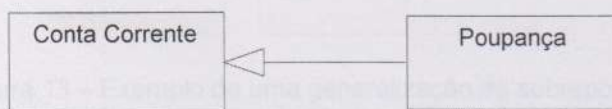


Figura 12 – Exemplo de generalização normal
 Fonte: UML Linguagem de Modelagem Unificada (2008)

Uma classe pode ser tanto uma subclasse quanto uma superclasse, se ela estiver numa hierarquia de classes que é um gráfico onde as classes estão ligadas através de generalizações.

A generalização normal é representada por uma linha entre as duas classes que fazem o relacionamento, sendo que se coloca uma seta no lado da linha onde se encontra a superclasse indicando a generalização.

Generalização Restrita

Uma restrição aplicada a uma generalização especifica informações mais precisas sobre como a generalização deve ser usada e estendida no futuro. As restrições a seguir definem as generalizações restritas com mais de uma subclasse:

Generalizações de Sobreposição e Disjuntiva: Generalização de sobreposição significa que quando subclasses herdarem de uma superclasse por sobreposição, novas subclasses destas podem herdar de mais de uma subclasse. A generalização disjuntiva é exatamente o contrário da sobreposição e a generalização é utilizada como padrão.

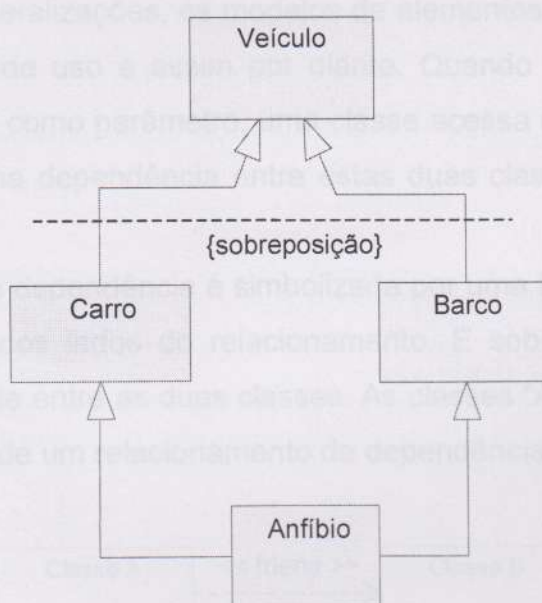


Figura 13 – Exemplo de uma generalização de sobreposição
Fonte: UML Linguagem de Modelagem Unificada (2008)

Generalizações Completa e Incompleta: Uma restrição simbolizando que uma generalização é completa significa que todas as subclasses já foram especificadas, e não existe mais possibilidade de outra generalização a partir daquele ponto. A generalização incompleta é exatamente o contrário da completa e é assumida como padrão da linguagem.

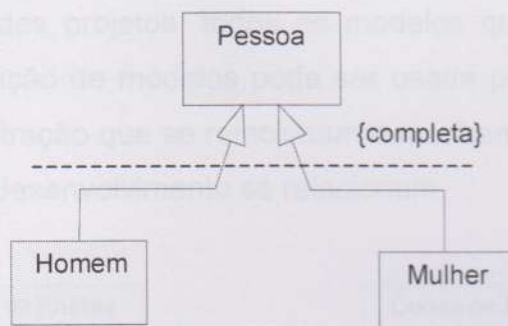


Figura 14 – Exemplo de generalização completa
Fonte: UML Linguagem de Modelagem Unificada (2008)

Dependência e Refinamentos

Além das associações e generalizações, existem ainda dois tipos de relacionamentos em UML. O relacionamento de dependência é uma conexão semântica entre dois modelos de elementos, um independente e outro dependente. Uma mudança no elemento independente irá afetar o modelo dependente. Como no caso anterior com generalizações, os modelos de elementos podem ser uma classe, um pacote, um caso de uso e assim por diante. Quando uma classe recebe um objeto de outra classe como parâmetro, uma classe acessa o objeto global da outra. Nesse caso existe uma dependência entre estas duas classes, apesar de não ser explícita.

Uma relação de dependência é simbolizada por uma linha tracejada com uma seta no final de um dos lados do relacionamento. E sobre essa linha o tipo de dependência que existe entre as duas classes. As classes “Amigas” provenientes do C++ são um exemplo de um relacionamento de dependência.

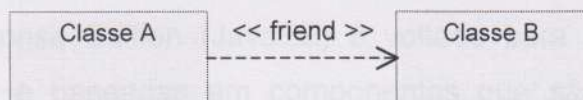


Figura 15 – Exemplo de uma dependência entre classes
Fonte: UML Linguagem de Modelagem Unificada (2008)

Os refinamentos são um tipo de relacionamento entre duas descrições de uma mesma coisa, mas em níveis de abstração diferentes, e podem ser usados para modelar diferentes implementações de uma mesma coisa (uma implementação simples e outra mais complexa, mas também mais eficiente).

Os refinamentos são simbolizados por uma linha tracejada com um triângulo no final de um dos lados do relacionamento e são usados em modelos de

coordenação. Em grandes projetos, todos os modelos que são feitos devem ser coordenados. Coordenação de modelos pode ser usada para mostrar modelos em diferentes níveis de abstração que se relacionam e mostram também como modelos em diferentes fases de desenvolvimento se relacionam.

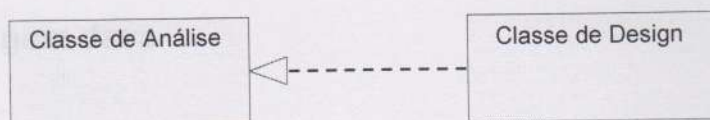


Figura 16 – Exemplo de um refinamento
Fonte: UML Linguagem de Modelagem Unificada (2008)

2.2 JAVA

Java é o nome do ambiente computacional, ou plataforma, desenvolvido pela empresa norte-americana *Sun Microsystems*. A plataforma permite o desenvolvimento de vários tipos de aplicativos e tem como base a linguagem padrão que leva o seu próprio nome.

A plataforma Java abrange um grande conjunto de tecnologias, dentre as quais podemos destacar a própria linguagem, JavaSE, JavaEE e JavaME.

O Java Standard Edition (JavaSE) é uma tecnologia de desenvolvimento para a plataforma Java. Ela contém todo o ambiente necessário para a criação e execução de aplicações Java, incluindo a máquina virtual Java (JVM), o compilador Java, as APIs do Java e outras ferramentas utilitárias.

O Java Enterprise Edition (JavaEE) é voltado para aplicações de grande porte, multicamadas e baseadas em componentes que são executados em um servidor de aplicações. A tecnologia JavaEE é considerada um padrão de desenvolvimento já que o fornecedor de software nesta plataforma deve seguir determinadas regras se quiser declarar os seus produtos compatíveis com o JavaEE. Ela contém bibliotecas desenvolvidas para o acesso a base de dados, RPC, CORBA, etc. Devido a essas características a plataforma é utilizada principalmente para o desenvolvimento de aplicações corporativas.

Java Micro Edition (JavaME) possibilita o desenvolvimento de software para sistemas e aplicações embarcadas, ou seja, toda aquela que roda em um dispositivo

de propósito específico, desempenhando alguma tarefa que seja útil para o dispositivo. É a tecnologia Java para dispositivos compactos, como celulares, PDA's, controles remotos, e uma outra gama de dispositivos.

2.2.1 A Linguagem Java

Java é uma linguagem de programação orientada a objetos desenvolvida na década de 90. É resultado de uma busca por uma linguagem de programação que pudesse fornecer uma ligação com o C++, mas com segurança. A seguir temos algumas das suas principais características.

JVM - Java Virtual Machine

O coração da plataforma e da linguagem Java é o conceito de uma máquina virtual: a JVM. Um programa em Java é compilado para o chamado *byte-code*, que são instruções para a JVM. Este *byte-code* é independentemente do *hardware* ou sistema operacional em que o programa será executado. A plataforma Java disponibiliza a JVM, que interpreta, em tempo de execução, o *byte-code* para instruções nativas do sistema operacional instalado. Isto permite que uma mesma aplicação seja executada em qualquer plataforma computacional que possua uma implementação da máquina virtual java.

Desde a versão 1.2 da JRE, a implementação da Sun da JVM inclui um compilador *just-in-time* (JIT). Com este compilador, todo o *byte-code* de um programa é transformado em instruções nativas e carregado na máquina virtual em uma só operação, permitindo um ganho de desempenho muito grande em comparação com a implementação anterior, onde as instruções em *byte-code* eram interpretadas uma por vez. O compilador JIT pode ser projetado de acordo com a plataforma ou *hardware* de destino, e o código que ele gera pode ser otimizado com base na observação de padrões de comportamento dos programas.

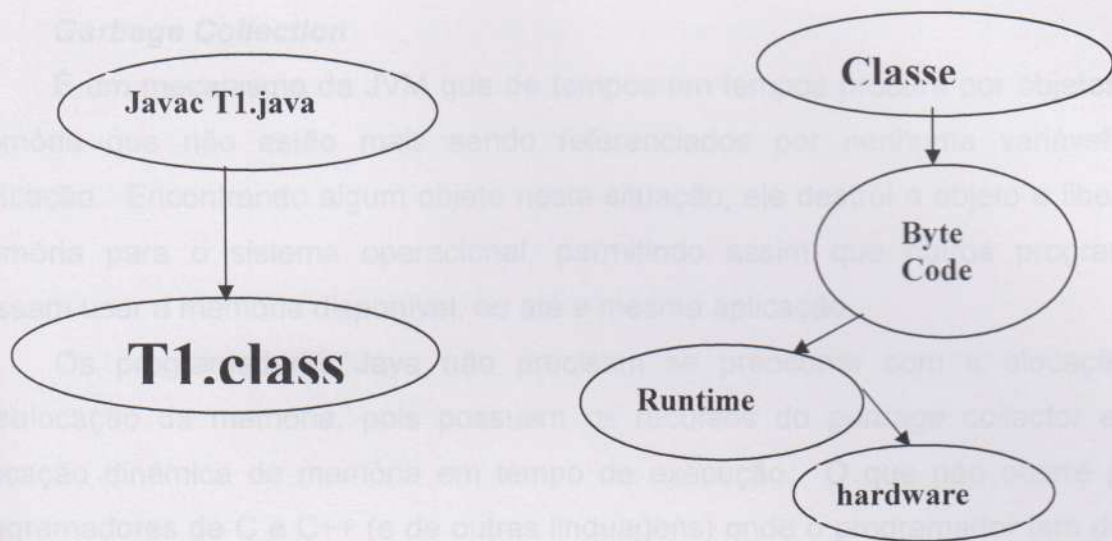


Figura 17 – Exemplo de compilação
Fonte: Apostila de Java (2008)

Parecida com C, C++

Java tem a aparência de C ou de C++, embora a filosofia da linguagem seja diferente. Quem programa em qualquer uma delas, ou em uma linguagem orientada a objetos, se sentirá mais a vontade e se tornará um bom programador Java em menos tempo.

Portável

Java foi criada para ser portátil. Dessa forma, o *byte-code* gerado pelo compilador para a sua aplicação específica pode ser transportado entre plataformas distintas que suportem a JVM. Não é necessário recompilar um programa para que ele rode numa máquina e sistema diferente, ao contrário do que acontece, por exemplo, com programas escritos em C e outras linguagens.

Esta portabilidade é importante para a criação de aplicações para a heterogênea internet.

Orientada a Objetos:

Java é uma linguagem orientada a objetos que suporta herança, mas não herança múltipla. A ausência de herança múltipla pode ser compensada pelo uso de herança e interfaces, onde uma classe herda o comportamento de sua superclasse além de oferecer uma implementação para uma ou mais interfaces. Java também permite a criação de classes abstratas.

Garbage Collection

É um mecanismo da JVM que de tempos em tempos procura por objetos na memória que não estão mais sendo referenciados por nenhuma variável da aplicação. Encontrando algum objeto nesta situação, ele destrói o objeto e libera a memória para o sistema operacional, permitindo assim que outros programas possam usar a memória disponível, ou até a mesma aplicação.

Os programadores Java não precisam se preocupar com a alocação e desalocação da memória, pois possuem os recursos do *garbage collector* e da alocação dinâmica de memória em tempo de execução. O que não ocorre para programadores de C e C++ (e de outras linguagens) onde o programador tem de se preocupar com a alocação e desalocação da memória.

Segura

A presença de coleta automática de lixo, evita erros comuns que os programadores cometem quando são obrigados a gerenciar diretamente a memória. A eliminação do uso de ponteiros, em favor do uso de vetores, objetos e outras estruturas substitutivas traz benefícios em termos de segurança. O programador é proibido de obter acesso a memória que não pertence ao seu programa, além de não ter chances de cometer erros comuns tais como o uso indevido de aritmética de ponteiros. Estas medidas são particularmente úteis quando pensarmos em aplicações comerciais desenvolvidas para a internet.

Ser fortemente tipada também é uma vantagem em termos de segurança, que está aliada a eliminação de conversões implícitas de tipos. A presença de mecanismos de tratamento de exceções torna as aplicações mais robustas, não permitindo que elas encerrem, mesmo quando rodando sob condições anormais.

Suporte a concorrência

A linguagem permite a criação, de maneira fácil, de várias *threads* de execução dentro de um mesmo programa. Esta característica facilita a criação de programas distribuídos e multitarefa.

Eficiente

Como Java foi criada para ser usada em computadores pequenos, ela exige pouco espaço e pouca memória. Java é muito mais eficiente que grande parte das

linguagens de *script* existentes, embora seja mais lenta que C. Mas com a evolução da linguagem, estão sendo criados geradores de *byte-code* cada vez mais otimizados que trazem as marcas de performance da linguagem cada vez mais próximas das de C++ e C.

Suporte para programação de sistemas distribuídos

Java fornece facilidades para programação de sistemas distribuídos, tais como *sockets*, *remote method call*, *tcp-ip*, *threads*, dentre outros.

2.3 DESENVOLVIMENTO DE INTERFACES COM SWING

Nas primeiras versões do Java a única forma de fazer programas gráficos era através da AWT, uma biblioteca de baixo-nível que dependia de código nativo da plataforma onde rodava. Ela traz alguns problemas de compatibilidade entre as plataformas, fazendo que nem sempre o programa tinha aparência desejada em todos os sistemas operacionais.

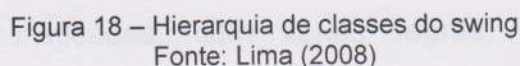
Para suprir as necessidades cada vez mais freqüentes de uma API mais estável e fácil de usar, o *Framework Swing* foi criado como uma extensão do Java a partir da versão 1.2. *Swing* fornece componentes de mais alto nível, possibilitando assim uma melhor compatibilidade entre os vários sistemas onde Java roda.

Ao contrário da AWT, *Swing* não contém uma única linha de código nativo, e permite que as aplicações tenham diferentes tipos de visuais (*skins*), os chamados "*Look and Feel*". Já com AWT isso não é possível, tendo todos os programas a aparência da plataforma onde estão rodando.

Apesar da AWT ainda estar disponível no Java, é altamente recomendável que seja usado *Swing*, pelas razões descritas aqui e por várias outras. Componentes *Swing* contém um "J" na frente, como em *JButton* por exemplo. Componentes AWT não contém inicial alguma ("*Button*" no caso).

Além disso tudo, as vantagens do *Swing* não param por aí. Há uma enorme gama de controles extras disponíveis, tais como áreas de texto que nativamente podem mostrar conteúdo RTF ou HTML, botões com suporte a imagens, *sliders*, selecionadores de cores etc. É também possível alterar o tipo de borda para a maior

A figura 18 ilustra a hierarquia de classes do *Swing*.



Cada programa com uma interface gráfica do *Swing* deve ter pelo menos um recipiente *top-level* do *Swing*. Um recipiente *top-level* do *Swing* fornece o suporte que os componentes do *Swing* necessitam para desenhar e manipular os eventos. Há quatro tipos de recipientes *top-level* geralmente utilizados do *Swing*: *JFrame*, *JDialog*, *JWindow* e (para applet) *JApplet*. Cada objeto de *JFrame* executa uma única janela principal, e cada *JDialog* uma janela secundária (um dependente da janela em uma outra janela). Cada objeto de *JApplet* executa a área de exposição de um *applet* dentro de uma janela de navegador.

Executado como um exemplo da classe de `JFrame`, um frame é uma janela que, por padrão, tem decorações tais como uma borda, um título, e botões para minimizar e fechar a janela. As aplicações com um GUI (Interface Gráfica de Usuário) utilizam tipicamente pelo menos um frame.

À exceção dos recipientes *top-level*, tais como `JFrame`, todos os componentes do *Swing* descendem da classe `JComponent`. Cada recipiente *top-level* tem um *content panel* que contém, diretamente ou indiretamente, todos os componentes visíveis (à exceção dos menus e das decorações da janela) no recipiente *top-level*. Então para montar a sua interface gráfica, basta ir adicionando ao *content panel* os componentes visuais desejados, como botões, caixas de texto, listas, e etc.

Quando montamos nossa interface definindo coordenadas para o posicionamento de cada componente, podemos está ocasionando alguns problemas como o não reposicionamento caso a janela seja redimensionada, e quando a resolução da tela não é a mesma em que o programa foi desenvolvido. Para solucionar estes problemas fazemos o uso dos gerenciadores de *layout*. Estes gerenciadores posicionam os componentes segundo uma regra e quando ocorre o redimensionamento da tela ou a mudança de resolução, estes gerenciadores automaticamente corrigem as posições e dimensões dos componentes na tela.

Temos seis tipos de gerenciadores: `FlowLayout` onde os componentes são adicionados da esquerda para a direita; `BorderLayout` onde a tela é dividida em cinco regiões, e os componente são adicionados em uma dessas regiões; `GridLayout` a tela é dividida em regiões de tamanhos iguais; `CardLayout` que permite que componentes sejam exibidos usando-se abas; `GridBagLayout` permite a criação de interfaces mais sofisticadas, com várias regras; `FreeDesign` permite o posicionamento livre de componentes.

Para o tratamento de eventos, o *Framework Swing* usa o padrão de projeto *Observer*, que define uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente. Assim temos classes que ficam escutando o evento e realizam alguma ação quando ele ocorre, são as chamadas classes *Listeners*. Existem vários tipos de eventos que podem ser monitorados, como os de ação, os do mouse, os de tecla, entre outros... Todo evento recebe um argumento do tipo

EventObject, e é através deste objeto que possível obter as informações do evento gerado e que objeto disparou o evento.

2.4 HIBERNATE

Trabalhar com desenvolvimento orientado a objetos e banco de dados relacionais pode ser complexo e muito demorado. Pois fica a cargo do programador misturar os comandos SQL para a persistência dos dados e consultas no banco de dados relacional, com o código Java mapeando os resultados para os seus objetos.

Hibernate é um mecanismo bem simples e poderoso que permite fazer o mapeamento de objetos em banco de dados relacionais, tudo de maneira transparente e para qualquer tipo de aplicação Java, seja ela uma aplicação para Internet ou uma aplicação *desktop*. Ou seja, ao invés de perder tempo escrevendo aquele código SQL, com o *Hibernate* você vai precisar se preocupar apenas com suas classes.

O *Hibernate* oferece uma grande variedade de mapeamentos para coleções e objetos dependentes, além do suporte a herança, polimorfismo e agregação. Não gera código extra na compilação dos códigos fonte para *byte-code*, tem uma alta performance com sua arquitetura com dois *caches* e pode ser usado em *cluster*. Soluciona não somente o envio de dados ao banco, mas também a busca dos mesmos através de várias maneiras, tais como o *Hibernate Query Language* (HQL), *Criteria*, *Query By Example* e SQL Nativo.

O HQL é uma linguagem de consulta muito parecida com o SQL, só que totalmente orientada a objetos, podendo usar herança, polimorfismo e associações. *Criteria* é uma API extensível e intuitiva provida pelo *Hibernate* que faz consultas por critérios, onde uma classe é selecionada e os critérios de seleção são adicionados a busca para a obtenção dos resultados. *Query By Example* é semelhante ao *Criteria* sendo que o critério de seleção utilizado são objetos *Hibernate* da classe *Example*. E o SQL nativo, como o próprio nome diz, é a utilização de consultas no dialeto SQL nativo do banco de dados instalado.

Adicionalmente, o *Hibernate* possui suporte a transações a nível de aplicação suportando contextos de persistência de longa duração, executando *lock* otimista automaticamente. É *open source* licenciado sobre a licença do tipo LGPL e possui compatibilidade com a API Java JCA garantindo a conexão com banco de dados legados de forma transparente.

O processo de desenvolvimento usando *Hibernate* é muito mais simples do que usar JDBC puro. Ao utilizá-lo, geralmente são necessários apenas a implementação de poucos passos:

- criar a tabela no seu banco de dados onde os objetos vão persistir;
- criar o objeto cujo estado vai ser persistido;
- criar um XML ou utilizar *Annotations* em suas classes para relacionar as propriedades do objeto aos campos na tabela;
- criar um arquivo contendo as propriedades para que o *Hibernate* se conecte ao banco de dados
- criar a classe DAO que vai persistir seu objeto;

2.4.1 Arquitetura do *Hibernate*

O *Hibernate* faz a comunicação da aplicação com o banco de dados, ele usa as informações da configuração para prover serviços de persistência (e objetos persistentes) à aplicação (veja a parte esquerda da figura 19). E detalhando a sua arquitetura (parte a direita da figura 19) vemos um conjunto de interfaces que o formam.

De acordo com a Figura 19, as interfaces são classificadas como:

- Interfaces responsáveis por executar operações de criação, deleção, consulta e atualização no banco de dados: *Session*, *Transaction* e *Query*;
- Interface utilizada pela aplicação para configurar o *Hibernate*: *Configuration*;
- Interfaces responsáveis por realizar a interação entre os eventos do *Hibernate* e a aplicação: *Interceptor*, *Lifecycle* e *Validatable*.
- Interfaces que permitem a extensão das funcionalidades de mapeamento do *Hibernate*: *UserType*, *CompositeUserType*, *IdentifierGenerator*.

O *Hibernate* também interage com APIs já existentes do Java: JTA, JNDI e JDBC (GALHARDO; LIMA, 2008).

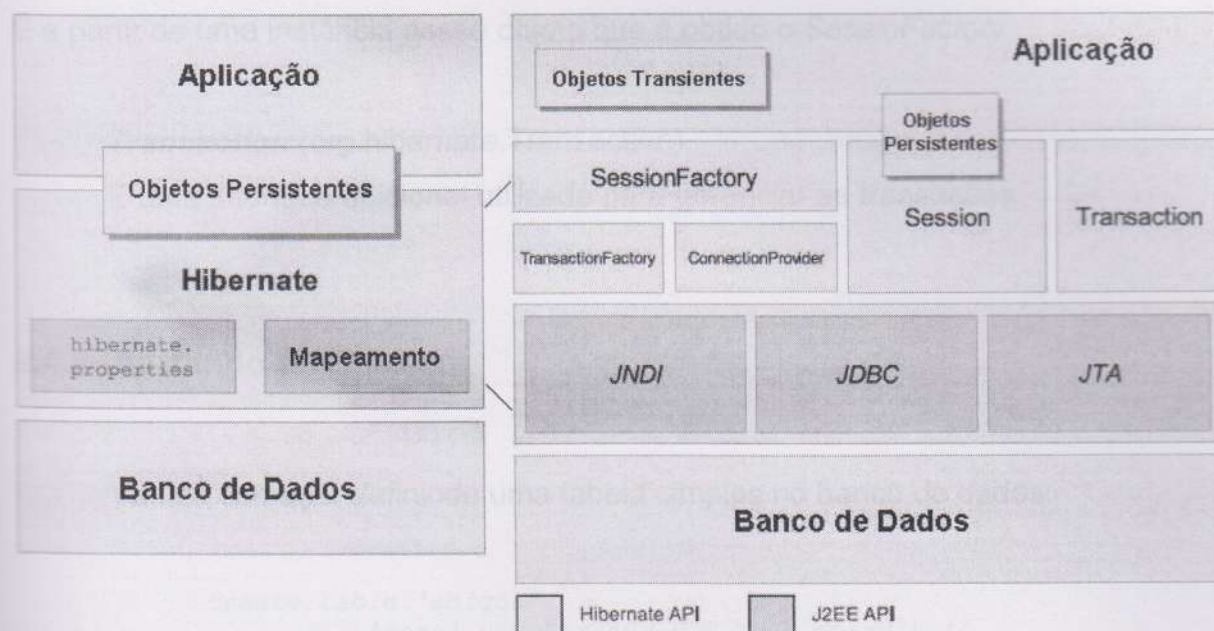


Figura 19 – Arquitetura do Hibernate
Fonte: Dados da pesquisa (2008)

Algumas definições sobre os objetos da figura 19:

Session (org.hibernate.Session)

É uma interface leve e não custosa para criar e destruir, que possibilita a criação, remoção, busca e atualização de objetos. Não é *threadsafe*, ou seja, duas *threads* não podem usar um mesmo *Session* sem sincronização e sua noção é algo entre a conexão e a transação. Faz a comunicação entre a aplicação e a persistência dos dados.

SessionFactory (org.hibernate.SessionFactory)

Representa a fábrica de objetos *Session*, sendo um objeto pesado com tipicamente uma instância por aplicação. Ele faz cache dos comandos SQL gerados pelo *Hibernate*.

Configuration (org.hibernate.Configuration) (hibernate.properties)

Um objeto *Configuration* é utilizado para realizar as configurações de inicialização do *Hibernate*. Com ele, define-se diversas configurações do *Hibernate*. É a partir de uma instância desse objeto que é obtido o *SessionFactory*.

Transaction (org.hibernate.Transaction)

É uma interface opcional utilizada para gerenciar as transações.

2.4.2 Utilizando o *Hibernate*

Vamos começar definindo uma tabela simples no banco de dados.

```
create table 'amigos'(  
    'nome' varchar(40) NOT NULL default '',  
    'nascimento' datetime default null,  
    PRIMARY KEY (`nome`)  
);
```

Listagem 1 – Criação de tabela exemplo
Fonte: Oliveira (2008)

Agora vamos para o próximo passo, que é a representação disso em forma de objeto em Java. Como o *Hibernate* trabalha através de reflexão para fazer o mapeamento, temos que colocar os métodos *getters* e *setters* no nosso objeto. Assim:

```
public class Amigo {  
    private String nome;  
    private java.util.Date nascimento;  
  
    public Amigo() {  
    }  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    public java.util.Date getNascimento() {  
        return nascimento;  
    }  
    public void setNascimento(java.util.Date nascimento) {  
        this.nascimento = nascimento;  
    }  
}
```

Listagem 2 – Código da classe Amigo
Fonte: Oliveira (2008)

Criamos uma tabela e uma classe que "representa" o esquema desta nossa tabela. Então nos falta informar ao *Hibernate* como juntar os dois.

O mapeamento do objeto com o banco é através de *Annotations*, que são uma novidade do Java 5. Eles permitem adicionar configurações diretamente no fonte sem precisar de manutenção de arquivos XML separados. O *Hibernate Annotations* permite configurar as classes de mapeamento através de anotações.

O mapeamento então seria feito da seguinte maneira:

```
@Entity
public class Amigo {
    @Id
    private String nome;
    @Column
    private java.util.Date nascimento;
    ...
}
```

Listagem 3 – Código da classe Amigo com Annotations
Fonte: Dados da pesquisa (2008)

A anotação *@Entity* indica que a classe devera ser persistida e possui uma tabela correspondente de mesmo nome no banco de dados. A anotação *@Id* indica que aquele atributo é a chave primeira na tabela e o *@Column* indica que o atributo é uma coluna da tabela, como não especificado os nomes dos campos na tabela, o *Hibernate* assume que são os mesmos nomes dos atributos da classe.

Agora, vamos criar o arquivo de configuração para que o *Hibernate* possa se conectar ao banco de dados. Este pode ser um arquivo XML ou um arquivo de propriedades bem comum, vejamos o exemplo na listagem 4.

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
  <session-factory name="java:comp/env/hibernate/SessionFactory">
    <property name="hibernate.connection.driver_class">
      org.postgresql.Driver
    </property>
    <property name="hibernate.connection.url">
      jdbc:postgresql://localhost:5432/FolhaPagamento
    </property>
    <property name="hibernate.connection.username">
      postgres
    </property>
    <property name="hibernate.connection.password">
      postgres
    </property>
    <property name="hibernate.dialect">
      org.hibernate.dialect.PostgreSQLDialect
    </property>
    <mapping class="Amigo" />
  </session-factory>
</hibernate-configuration>

```

Listagem 4 – Exemplo de arquivo XML de configuração do *Hibernate*
 Fonte: Dados da pesquisa (2008)

Dê a ele o nome que você quiser (agenda.properties, por exemplo). Porém, se você der a este arquivo o nome de hibernate.properties e colocá-lo no seu classpath, o *Hibernate* irá carregá-lo automaticamente durante a início do programa.

Outra maneira de fazer o mapeamento das classes é criando um arquivo, cujo nome deve ser sempre "NomeDoObjeto.hbm.xml" (no nosso caso, "Amigo.hbm.xml"); cada objeto a ser persistido tem seu próprio arquivo de mapeamento.

O modelo para o nosso Amigo.hbm.xml segue abaixo:

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping.dtd">
<hibernate-mapping>
  <class name="Amigo" table="amigos">
    <id name="nome" column="nome" type="string">
      <generator class="assigned"/>
    </id>
    <property name="nascimento" type="date"/>
  </class>
</hibernate-mapping>

```

Listagem 5 – Exemplo de arquivo XML para mapeamento de classes
 Fonte: Oliveira (2008)

Explicando a listagem 5:

O atributo "name", da tag "class" - o valor de "name" inclui o nome completo do objeto, incluindo seu package (ou seja, x.y.z.MeuObjeto). É nesta tag que dizemos qual tabela se relaciona com qual objeto (através do atributo "table");

A tag "property" - o atributo "name" se refere ao atributo da classe, enquanto a propriedade "column" se refere à coluna no nosso banco de dados (este última é opcional: se você deixá-la em branco, o Hibernate vai procurar por uma coluna cujo nome seja igual ao da propriedade do seu bean). O atributo "type" também é opcional e se refere ao tipo de dado do atributo da sua classe;

A tag ID – faz o mapeamento da chave primária da sua tabela. E o mapeamento é semelhante ao feito nas tags "property";

E finalmente para poder persistir os objetos precisamos de uma classe DAO:

```
import cirrus.hibernate.*;
import java.util.Date;
public class AmigoDAO{

    private SessionFactory factory;

    public AmigoDAO() throws Exception{
        Datastore datastore = Hibernate.createDatastore();
        datastore.storeClass(Amigo.class);
        factory = datastore.buildSessionFactory();
    }

    public void insert(Amigo amigo) throws Exception{
        Session session = factory.openSession();
        session.save(amigo);
        session.flush();
        session.close();
    }

    public Amigo retrieve(String pk) throws Exception{
        Session session = factory.openSession();
        Amigo amigo = (Amigo)session.load(Amigo.class, pk);
        session.flush();
        session.close();
        return amigo;
    }

    public void delete(Amigo amigo) throws Exception{
        Session session = factory.openSession();
        session.delete(amigo);
        session.flush();
        session.close();
    }
}
```

Listagem 6 – Exemplo de uma classe DAO

Fonte: Oliveira (2008)

A partir daí podemos persistir nossos objetos da classe Amigo através do DAO.

3 MODELAGEM DO SISTEMA

O sistema modelado é um sistema de folha de pagamento, bastante utilizado hoje em escritórios de contabilidade ou no departamento pessoal de várias empresas. É um sistema útil e imprescindível, pois devido ao grande número de cálculos que devem ser feitos para se poder preencher todos os formulários da área trabalhista, fica muito custoso e demorado fazer tudo à mão. Sem mencionar o fato de que a legislação do trabalho é farta e é alterada continuamente, dificultando ainda mais a realização desta tarefa.

A folha de pagamento é um documento de emissão obrigatória para efeito de fiscalização trabalhista e previdenciária. A empresa é obrigada a preparar a folha de pagamento da remuneração paga, devida ou creditada a todos os empregados a seu serviço.

Para sua elaboração não existe modelo oficial, ou seja, podem ser adotados critérios que melhor atendam as necessidades de cada empresa. Uma folha de pagamento, por mais simples que seja, apresenta pelo menos os seguintes elementos:

- Discriminação do nome dos empregados (segurados), indicando cargo, função ou serviço prestado; Valor bruto dos salários;
- Valor da contribuição de Previdência, descontado dos salários;
- Valor líquido que os empregados receberão.

Da folha de pagamento origina-se o recibo de pagamento, que indica os dados que constaram da folha relativamente a cada um dos empregados e a estes é entregue (OLIVEIRA, 2008).

3.1 CÁLCULOS DA FOLHA DE PAGAMENTO

Despesas da empresa

Valor bruto dos salários é o valor, considerado, para a empresa como despesa total de salários. Outra despesa para a empresa é a contribuição de

Previdência da parte da empresa que é calculada como 21% do valor total dos salários de seus empregados. E tem também o Fundo de Garantia do Tempo de Serviço (FGTS) que corresponde a 8% do valor recebido no mês pelos empregados.

Exemplos: Valor total dos salários pagos: 1.000,00

Valor a ser pago ao INSS (21%): 210,00

Valor a ser pago ao FGTS (8%): 80,00

Previdência Social - INSS

De acordo com a legislação atual, todo empregado assalariado, regido pela Consolidação das Leis do Trabalho (CLT), está obrigado a contribuir com a Previdência Social. Essa contribuição é descontada do empregado em folha de pagamento, ela varia de acordo com a faixa salarial de cada empregado e é calculada mediante aplicação de um percentual sobre o salário de contribuição.

Atualmente, o cálculo é feito com base na tabela apresentada na figura 20.

MES/ANO.....: 07/07			
FAIXAS DE DESCONTOS	SALARIO CONTRIBUICAO	PERC.	
FAIXA 1.....:	868,29	7,65%	
FAIXA 2.....:	1.140,00	8,65%	
FAIXA 3.....:	1.447,14	9,00%	
FAIXA 4.....:	2.894,28	11,00%	
FAIXA 5.....:	0,00	0,00%	
FAIXA 6.....:	0,00	0,00%	
SAL.FAMILIA VALOR ATE...:	449,93	VALOR PAGO...:	23,08
SAL.FAMILIA VALOR ENTRE:	449,94		
E:	676,27	VALOR PAGO...:	16,26
PERC.DE AJUSTE DO CPMF.: 0,3500%			
TETO MAXIMO PARA PAGAMENTO DO SALARIO MATERNIDADE..:	13.165,20		

Figura 20 – Tabela de alíquotas do INSS e salário família

Fonte: Oliveira (2008)

O salário de contribuição é um valor fixado pela Previdência que serve de base para o cálculo das contribuições previdenciárias. Vale lembrar que o teto de contribuição é de 11% de 2.894,28 que corresponde a R\$ 318,37.

Exemplos:

Quem recebe R\$ 1.000,00 tem pela tabela alíquota de 8,65%, logo contribuirá com R\$ 86,5 a previdência.

Quem recebe R\$ 2.000,00 tem pela tabela alíquota de 11%, logo contribuirá com R\$ 220 a previdência.

Quem recebe R\$ 3.000,00 recebe acima do teto e paga portanto R\$ 318,37 a previdência.

Fundo de Garantia do Tempo de Serviço (FGTS)

Corresponde a 8% sobre o valor bruto da folha de pagamento, o qual será recolhido na Caixa Econômica Federal, em nome dos empregados. Constitui uma despesa paga pela empresa aos empregados.

Imposto de Renda na Fonte (IRF)

É um imposto variável de acordo com o valor do salário. Sobre o valor bruto do salário aplica-se a tabela progressiva da figura 21.

Tabela de IRRF de 07/2007				
MES/ANO DA TABELA DO IR.....: 07/07				
ATE		ISENTO	PERC.	VALOR DEDUZIR
DE	1.313,69			
DE	1.313,70 A	2.625,12	15,00%	197,05
DE	2.625,13 A	999.999.999.999,99	27,50%	525,19
ACIMA DE	999.999.999.999,99		27,50%	525,19
VALOR A DEDUZIR POR DEPENDENTE.....:				132,05
ISENCAO FUNCIONARIO 65 ANOS.....:				0,00
VALOR A DEDUZIR (MP N°202 DE 23/07/2004).....:				0,00

Figura 21 – Tabela de alíquotas do IRRF

Fonte: Oliveira (2008)

O cálculo do imposto de renda devido já é um pouco mais complicado. Primeiro pega-se o valor da remuneração obtida no período e aplica-se as deduções se cabível (desconta-se o valor do INSS do mês e um valor por dependente), do resultado obtido encontramos a alíquota na tabela e a multiplicamos pelo valor menos as deduções. Em seguida deduzimos o valor da parcela a deduzir da tabela, se o resultado destas operações for maior que R\$ 10, será esse o valor descontado do empregado como IRRF.

Exemplo:

Quem recebe R\$ 2.000,00 e tem um (1) dependente:

Valor do salário menos deduções = $2000 - 220 \text{ (INSS)} - 132,50 \text{ (1 dependente} \times 132,05)$

Valor tributável = R\$ 1.647,50

Para esse valor temos uma alíquota de 15% logo:

Valor apurado = R\$ 247,13

Diminuindo a parcela a deduzir temos o imposto devido:

IRRF = $247,13 - 197,05 = \text{R\$ } 50,08$

Salário Família

É um auxílio da Previdência Social aos trabalhadores de baixa renda que possuem filhos menores de 14 anos de idade. A empresa paga este valor para o empregado e é reembolsada pela Previdência Social quando efetua os recolhimentos correspondentes a folha de pagamento. Para o cálculo do benefício é utilizada uma tabela como a que consta na figura 20.

Exemplos:

Quem recebe até R\$ 449,93 e tem dois (2) filhos menores de 14 anos, vai receber de salário família o total de R\$ 46,16 ($2 \times \text{R\$ } 23,08$).

Quem recebe até R\$ 676,27 e tem três (3) filhos menores de 14 anos, vai receber de salário família o total de R\$ 48,78 ($3 \times \text{R\$ } 16,26$).

Quem recebe mais de R\$ 676,27 não recebe o benefício mesmo tendo filhos menores de 14 anos.

Esses são os cálculos básicos de uma folha de pagamento hoje em dia.

3.2 DIAGRAMA DE CLASSES

Começaremos agora a modelar as classes que poderiam implementar um sistema de folha de pagamento simplificado.

O diagrama de classes elaborado é mostrado na figura 22.

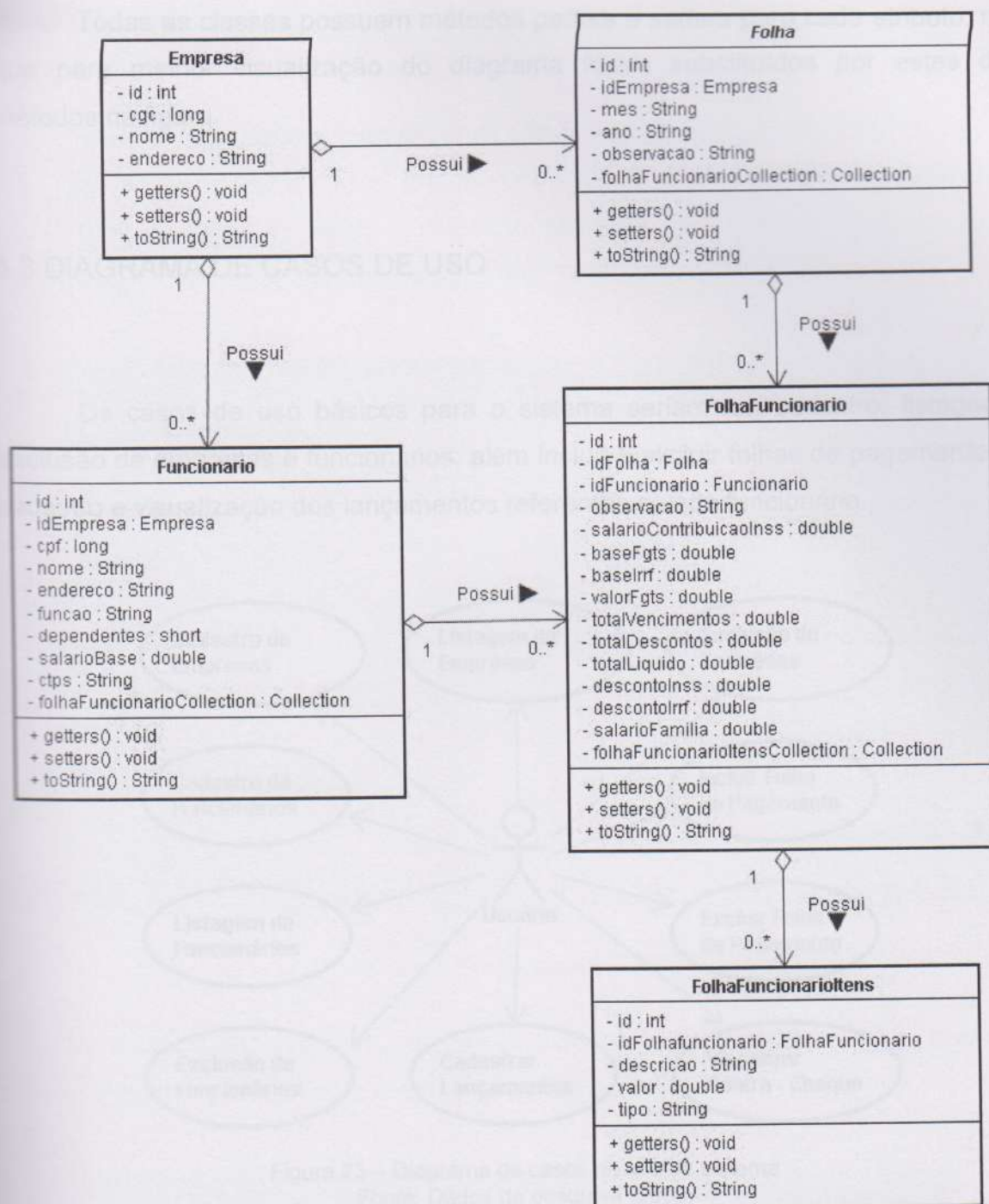


Figura 22 – Diagrama de classes do sistema

Fonte: Dados da pesquisa (2008)

Temos modelado na figura 22 uma classe para a empresa que possui vários funcionários e várias folhas de pagamento. A classe *FolhaFuncionario* relaciona cada funcionário com a folha de pagamento mensal e pode possuir vários itens.

Todas as classes possuem métodos *getters* e *setters* para cada atributo, mas que para melhor visualização do diagrama foram substituídos por estes dois métodos na figura.

3.3 DIAGRAMA DE CASOS DE USO

Os casos de uso básicos para o sistema seriam um cadastro, listagem e exclusão de empresas e funcionários, além incluir e excluir folhas de pagamento e o cadastro e visualização dos lançamentos referentes a cada funcionário.

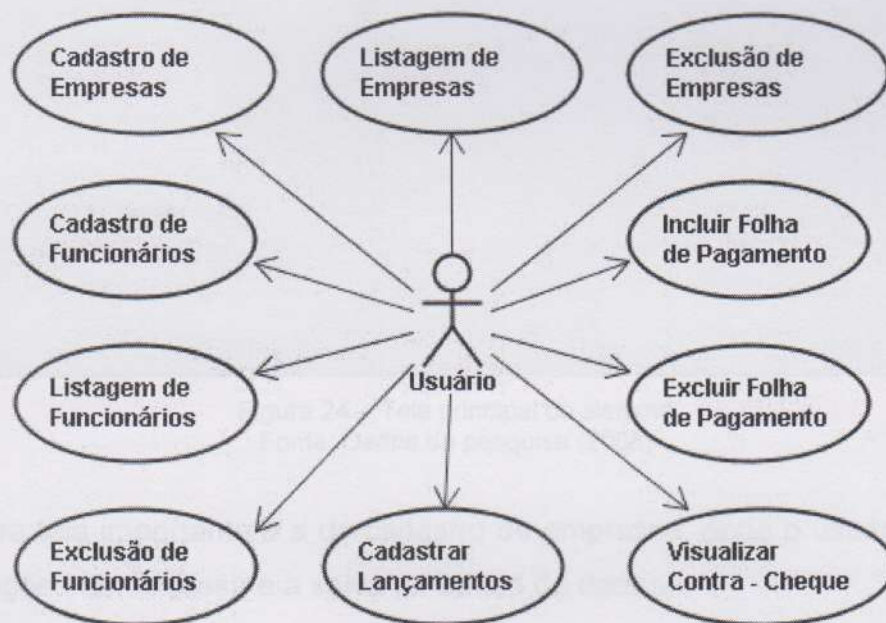


Figura 23 – Diagrama de casos de uso do sistema
Fonte: Dados da pesquisa (2008)

A primeira coisa que o usuário deve fazer é cadastrar uma empresa, após isso, o cadastro de funcionários daquela empresa. Tendo a empresa e os funcionários cadastrados, basta incluir a folha de pagamento para o mês em questão e, caso necessário, cadastrar algum lançamento específico dos funcionários que necessitarem. Depois de tudo isso feito, o usuário do sistema terá sua folha de pagamento para o mês cadastrado pronta.

3.4 – Protótipos de interface

Mostraremos nesta seção algumas das principais telas do sistema.

Na figura 24 temos a tela principal do programa com os menus e uma barra de ferramentas para o rápido acesso ao cadastro de funcionários, cadastro de lançamentos e para o fechamento do sistema.

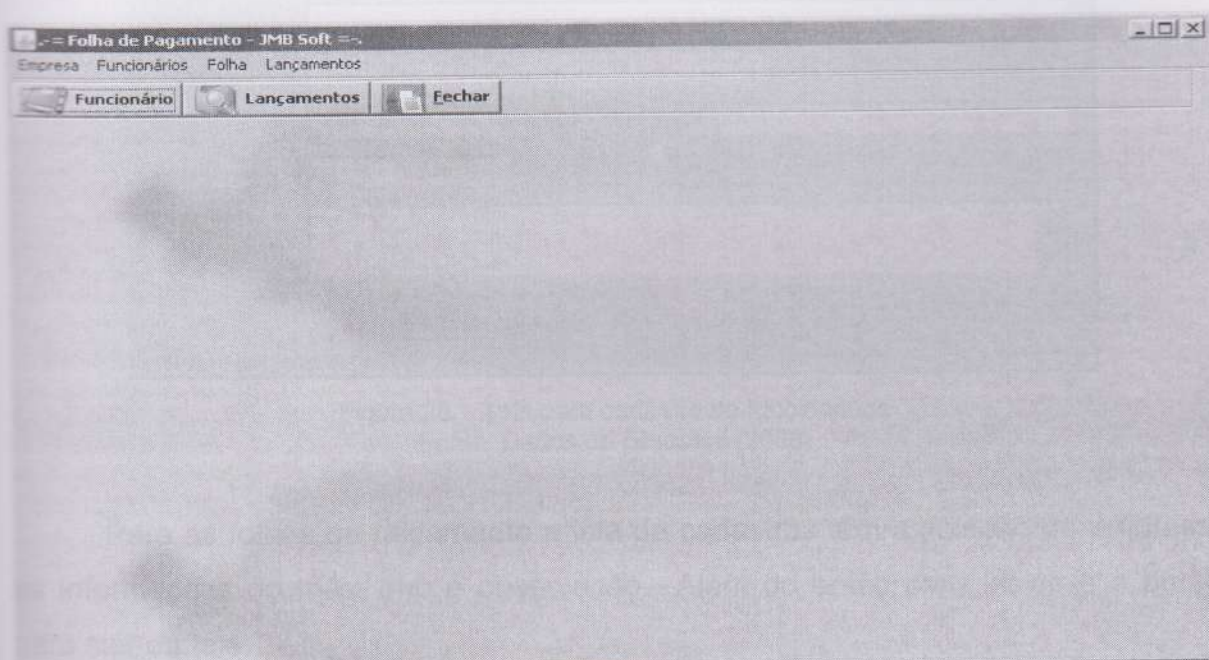


Figura 24 – Tela principal do sistema
Fonte: Dados da pesquisa (2008)

Outra tela importante é a de cadastro de empresas, onde o usuário preenche as informações da empresa e a salva no banco de dados.

Figura 25 – Tela de cadastro de empresas
Fonte: Dados da pesquisa (2008)

A tela de cadastro de funcionários tem campos texto para o usuário inserir os dados do funcionário, uma caixa de seleção para se escolher a empresa do mesmo e os botões para salvar e sair da tela.

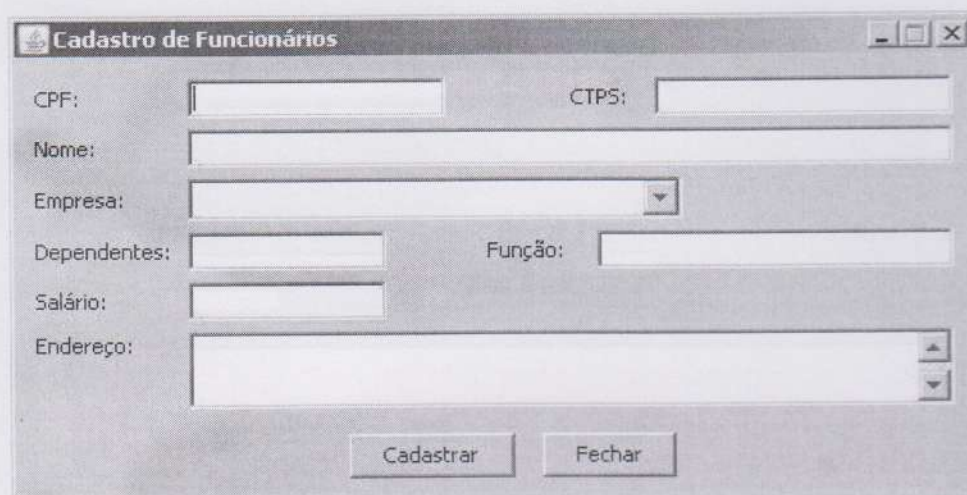
A imagem mostra uma janela de software intitulada "Cadastro de Funcionários". Ela contém vários campos de entrada: "CPF:" e "CTPS:" no topo; "Nome:" logo abaixo; "Empresa:" com uma caixa de seleção; "Dependentes:" e "Função:" no meio; "Salário:" abaixo de Dependentes; e "Endereço:" no fundo com uma barra de rolagem. Na base da janela, há dois botões: "Cadastrar" e "Fechar".

Figura 26 – Tela para cadastro de funcionários
Fonte: Dados da pesquisa (2008)

Para as folhas de pagamento a tela de cadastros tem a seleção da empresa, as informações do mês, ano e observação. Além do botão para incluir e o botão para sair da tela.

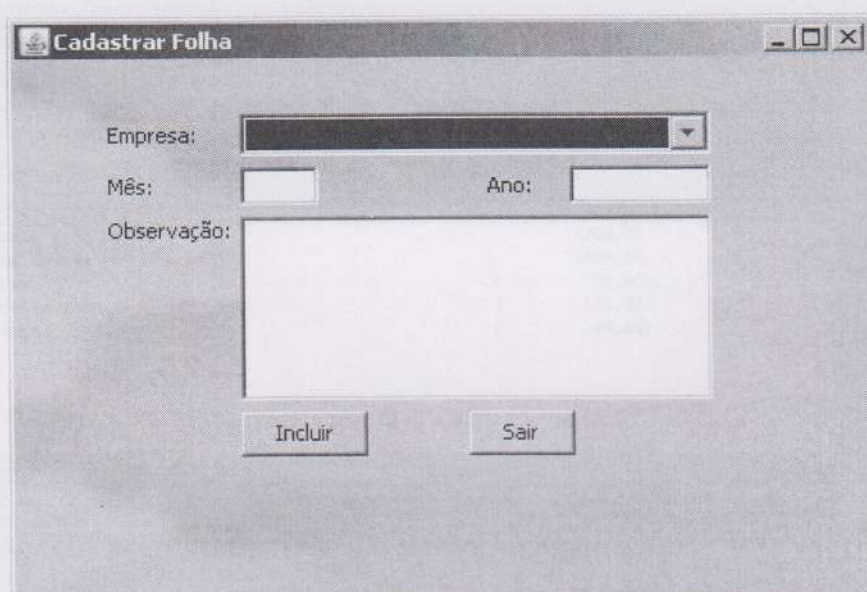
A imagem mostra uma janela de software intitulada "Cadastrar Folha". Ela contém: "Empresa:" com uma caixa de seleção; "Mês:" e "Ano:" no topo; e "Observação:" com uma área de texto grande no meio. Na base da janela, há dois botões: "Incluir" e "Sair".

Figura 27 – Tela para cadastrar folha
Fonte: Dados da pesquisa (2008)

E finalmente as telas para cadastrar os lançamentos individuais e visualizar os contra-cheques.

Figura 28 – Tela para cadastrar lançamentos
Fonte: Dados da pesquisa (2008)

Descrição	Valor
Salário Base	1200,00
Horas Extras	1000,00
Faltas	(82,30)
Desconto INSS	(232,95)
Desconto IRRF	(85,66)

Salário Base: 1200,00 Base Cálculo FGTS: 2147,70 Total Vencimentos: 2200,00

Salário Contribuição INSS: 2147,70 Valor FGTS: 160,42 Total Descontos: 400,91

Base Cálculo IRRF: 1894,75 Total Líquido a Receber: 1799,09

Figura 29 – Tela para visualizar contra-cheque
Fonte: Dados da pesquisa (2008)

4 MAPEAMENTO OBJETO RELACIONAL

Neste capítulo abordaremos como foi feito o mapeamento objeto relacional do sistema através do *Hibernate*. Apresentando as tabelas no banco de dados, o mapeamento das classes, a configuração do *Hibernate* e o DAO.

4.1 BANCO DE DADOS

O sistema foi desenvolvido utilizando-se do banco de dados relacional *PostgreSQL 8.1* e teve uma estrutura de tabelas bastante parecidas com o já apresentado diagrama de classes como demonstra a figura 30.

Figura 30 – DER do sistema
Fonte: Dados da pesquisa (2008)

Todas as tabelas do banco possuem como chave primária o campo *id* e todos os campos do tipo inteiro e possuem auto-incremento, no caso do nosso banco de dados gerados por seqüências. As tabelas de funcionários e de folhas têm uma chave estrangeira de referência com a tabela de empresas que faz referência a chave primária da tabela de empresas. A tabela *folhafuncionario* tem duas chaves estrangeiras: a chave que se relaciona com a tabela de folha, e a chave que se relaciona com a tabela de funcionários. E por fim temos uma chave estrangeira na tabela *folhafuncionario* no campo *id_empresa* que se relaciona com a chave primária da tabela de empresas.

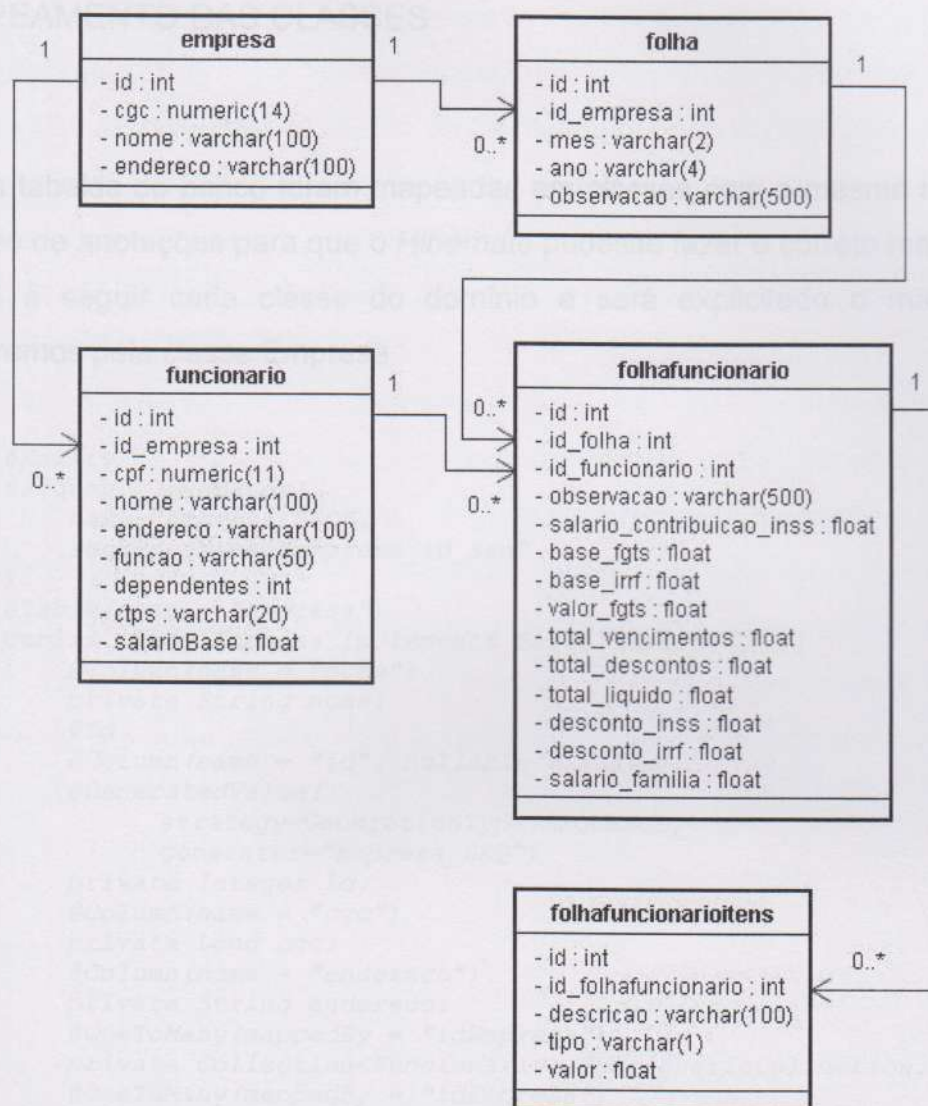


Figura 30 – DER do sistema
 Fonte: Dados da pesquisa (2008)

Todas as tabelas do banco possuem como chave primária o campo *id* e todos eles são do tipo inteiro e possuem auto-incremento, no caso do nosso banco de dados, gerados por seqüências. As tabelas de funcionários e de folhas têm uma chave estrangeira *id_empresa* que faz referencia a chave primária da tabela de empresas. A tabela *folhafuncionario* por sua vez possui duas chaves estrangeiras: *id_folha* que se relaciona com a tabela de folha, e *id_funcionario* que se relaciona com a tabela funcionário. E por fim temos uma chave estrangeira na tabela *folhafuncionarioitens* no campo *id_folhafuncionario* que se refere a chave primária da tabela *folhafuncionario*.

4.2 MAPEAMENTO DAS CLASSES

As tabelas do banco foram mapeadas em classes com o mesmo nome, e foi feito o uso de anotações para que o *Hibernate* pudesse fazer o correto mapeamento. Veremos a seguir cada classe do domínio e será explicitado o mapeamento. Começaremos pela classe *Empresa*.

```
@Entity
@SequenceGenerator(
    name="Empresa_SEQ",
    sequenceName="empresa_id_seq"
)
@Table(name = "empresa")
public class Empresa implements Serializable {
    @Column(name = "nome")
    private String nome;
    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue(
        strategy=GenerationType.SEQUENCE,
        generator="Empresa_SEQ")
    private Integer id;
    @Column(name = "cgc")
    private Long cgc;
    @Column(name = "endereco")
    private String endereco;
    @OneToMany(mappedBy = "idEmpresa")
    private Collection<Funcionario> funcionarioCollection;
    @OneToMany(mappedBy = "idEmpresa")
    private Collection<Folha> folhaCollection;
    ...
}
```

Listagem 7 – Mapeamento da classe *Empresa*
Fonte: Dados da pesquisa (2008)

Começamos com algumas anotações que são inseridas antes da definição da classe, fazendo referência à classe em si, começando com a anotação *@Entity* que indica que a classe será persistida no banco de dados. A seguir temos anotação *@SequenceGenerator* que define uma *sequence* do banco de dados para poder ser associada ao campo *id* da classe, esta anotação tem dois parâmetros definidos, o primeiro (*name*) é o nome que este *@SequenceGenerator* será identificado dentro da classe e o segundo (*sequenceName*) faz referência a uma sequência que existe no banco de dados. A próxima anotação é *@Table* que nos diz que a classe se referenciará a uma tabela no banco de dados que for indicada pelo parâmetro *name*.

Depois do `@Table` as anotações são inseridas antes de cada atributo da classe e se referem a cada atributo especificamente, como a anotação `@Column` que indica que o atributo é um campo da tabela indicado pelo parâmetro *name*. `@Id` indica que o atributo em questão é a chave primária da tabela, `@GeneratedValue` informa que o valor do campo ou atributo será gerado automaticamente, no caso por uma seqüência (definido pelo parâmetro *strategy*) declarada no começo da classe (definido pelo parâmetro *generator*).

A classe empresa possui dois relacionamentos, um com Funcionario e outro com Folha, ou seja, podemos ter um conjunto de funcionários daquela empresa e um conjunto de folhas daquela empresa. Estes relacionamentos estão representados na classe por duas coleções: *funcionarioCollection* e *folhaCollection*. E para que o *Hibernate* possa recuperar automaticamente estes conjuntos do banco de dados, usamos a anotação `@OneToMany` indicando que aquele atributo se refere a vários de outra classe, com o parâmetro *mappedBy* indicando qual atributo na classe “filha” que faz a ligação entre as classes.

A próxima classe a ser detalhada será a Folha.

```
@Entity
@SequenceGenerator(
    name="Folha_SEQ",
    sequenceName="folha_id_seq" )
@Table(name = "folha")
public class Folha implements Serializable {
    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue(
        strategy=GenerationType.SEQUENCE,
        generator="Folha_SEQ")
    private Integer id;

    @Column(name = "mes")
    private String mes;
    @Column(name = "ano")
    private String ano;
    @Column(name = "observacao")
    private String observacao;
    @OneToMany(mappedBy = "idFolha")
    private Collection<FolhaFuncionario>
        folhaFuncionarioCollection;

    @ManyToOne
    @JoinColumn(name = "id_empresa", referencedColumnName = "id")
    private Empresa idEmpresa;
    ...
}
```

Listagem 8 – Mapeamento da classe Folha
Fonte: Dados da pesquisa (2008)

Assim como na classe anterior, temos as anotações: `@Entity`, `@SequenceGenerator`, `@Table`, `@Column`, `@Id`, `@GeneratedValue`, `@OneToMany`. As únicas novidades são as anotações `@ManyToOne` e `@JoinColumn`. Essas anotações indicam que o atributo em questão é uma chave estrangeira de outra tabela no banco de dados, ou seja, aquele campo vai se referir a apenas um registro na tabela "pai". O parâmetro *name* indica qual o campo da tabela que é a chave estrangeira, e o parâmetro *referencedColumnName* indica o nome do campo que é referenciado na tabela "pai".

E semelhante as classes anteriores temos os mapeamentos das classes restantes. Como a classe `Funcionario`.

```
@Entity
@javax.persistence.SequenceGenerator(
    name="Funcionario_SEQ",
    sequenceName="funcionario_id_seq"
)
@Table(name = "funcionario")
public class Funcionario implements Serializable {
    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue( strategy=GenerationType.SEQUENCE,
                    generator="Funcionario_SEQ")
    private Integer id;
    ...

    @ManyToOne
    @JoinColumn(name = "id_empresa", referencedColumnName = "id")
    private Empresa idEmpresa;
    @OneToMany(mappedBy = "idFuncionario")
    private Collection<FolhaFuncionario>
        folhaFuncionarioCollection;
    ...
}
```

Listagem 9 – Mapeamento da classe `Funcionario`

Fonte: Dados da pesquisa (2008)

A classe `FolhaFuncionario`.

```

@Entity
@GeneratedValue(
    name="FolhaFuncionario_SEQ",
    sequenceName="folhafuncionario_id_seq"
)
@Table(name = "folhafuncionario")
public class FolhaFuncionario implements Serializable {
    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue( strategy=GenerationType.SEQUENCE,
                    generator="FolhaFuncionario_SEQ")
    private Integer id;

    ...

    @OneToMany( mappedBy = "idFolhafuncionario" )
    private Collection<FolhaFuncionarioItens>
        folhaFuncionarioItensCollection;

    @ManyToOne
    @JoinColumn(name = "id_folha", referencedColumnName = "id")
    private Folha idFolha;
    @ManyToOne
    @JoinColumn(
        name = "id_funcionario",
        referencedColumnName = "id")
    private Funcionario idFuncionario;

    ...
}

```

Listagem 10 – Mapeamento da classe FolhaFuncionario

Fonte: Dados da pesquisa (2008)

E a última classe FolhaFuncionarioItens.

```

@Entity
@GeneratedValue(
    name="FolhaFuncionarioItens_SEQ",
    sequenceName="folhafuncionarioitens_id_seq"
)
@Table(name = "folhafuncionarioitens")
public class FolhaFuncionarioItens implements Serializable {
    @Id
    @Column(name = "id", nullable = false)
    @GeneratedValue( strategy=GenerationType.SEQUENCE,
                    generator="FolhaFuncionarioItens_SEQ")
    private Integer id;

    ...

    @ManyToOne
    @JoinColumn(
        name = "id_folhafuncionario",
        referencedColumnName = "id")
    private FolhaFuncionario idFolhafuncionario;

    ...
}

```

Listagem 11 – Mapeamento da classe Folha

Fonte: Dados da pesquisa (2008)

4.3 DAO

Agora falaremos um pouco do arquivo XML de configuração do *Hibernate* e da classe de persistência elaborada, o DAO.

O arquivo de configuração (veja o apêndice A) do tem o nome de *hibernate.cfg.xml* e possui a seguinte estrutura.

Declaração que é um arquivo de configuração com a *session-factory* padrão e definimos algumas propriedades da sessão. A primeira propriedade "*hibernate.bytecode.use_reflection_optimizer*" é setada como falso fazendo com que não seja usado um otimizador de reflexão. "*hibernate.connection.driver_class*" é a propriedade que diz qual o driver de acesso a banco que será utilizado. "*hibernate.connection.pool_size*" informa qual o tamanho do pool utilizado. "*hibernate.connection.url*" define a string de conexão utilizada para poder acessar o banco de dados. "*hibernate.connection.username*" informa o usuário de conexão com o banco. "*hibernate.connection.password*" a senha deste usuário. "*hibernate.dialect*" essa propriedade se refere ao dialeto do banco de dados. "*hibernate.show_sql*" colocando essa propriedade como verdadeiro fazemos com que sejam mostradas as instruções sql passadas ao banco. As ultimas linhas do arquivo informam ao *Hibernate* que temos tabelas mapeadas através de anotações nas classes especificadas.

Criamos no sistema uma classe que é responsável pela a conexão com o banco de dados, que é a classe *Database*. Nesta classe temos um atributo *SessionFactory* que será inicializado com as configurações definidas no arquivo XML. E para se poder fazer qualquer coisa no banco, temos primeiro que pegar uma sessão desta classe. Vemos o seu código no apêndice B.

O nosso DAO possui três atributos, uma transação, uma sessão e um atributo auto-comit que por padrão é verdadeiro.

Ao ser criado, o nosso DAO pega a sessão da nossa classe *Database*. Os métodos *beginTransaction()*, *commitTx()* e *rollbackTx()* manipulam a transação com o banco, iniciando, finalizando ou desfazendo as alterações. O método *close()* fecha a sessão aberta. Os métodos *create()*, *delete()* e *update()* servem para persistir as classes no banco de dados, basta escolher a ação desejada e passar o objeto como

parâmetro. E na parte de consulta temos quatro opções já definidas no nosso DAO, que são: `findAll()` que nos retorna todos os objetos de uma classe que estão armazenados no banco de dados; `findQuery()` que nos permite passar uma consulta em HQL como parâmetro e seu resultado é trazido; `findByPrimaryKey()` nos trás o objeto da classe requerida que possui o id informado; `findLike()` é uma consulta que se obtém os elementos de uma classe que possui um campo com o valor especificado.

A utilização deste DAO facilita e simplifica consideravelmente os códigos necessários nas telas do sistema, como veremos no capítulo cinco.

2.3.3 O código da GUI

Como nosso sistema está utilizando o mapeamento objeto-relacional, as ações de cadastro, exclusão e alteração estão bastante simples, como veremos a seguir. Comentaremos agora o código das implementações das telas do sistema.

2.4 CADASTROS

As telas de cadastro do sistema possuem layout e funcionalidade bastante semelhantes. Elas têm as opções de texto para a inclusão das informações, campos de seleção para a escolha de empresas, telas ou funcionalidades. E dois botões, um para o cadastro e outro para a saída da tela (ver figuras 25, 26 e 27). A implementação dessas telas se resume basicamente ao código do botão de inclusão, onde é criado um objeto da classe e são atribuídos seus atributos com os valores informados pelo usuário na tela para então se passar o objeto para o nosso DAO, que cria o objeto no banco de dados. Segue como exemplo o código de inclusão de funcionários na imagem 12.

5 IMPLEMENTAÇÃO DOS CASOS DE USO

Todas as telas do sistema foram feitas utilizando-se do *framework swing* e são filhas de uma classe abstrata: *AbstractGUI*. Esta classe possui alguns métodos genéricos que são bastante úteis e utilizados nas demais telas do sistema. Os métodos *findAll()*, *findQuery()* e *findLike()* utilizam o *DAO* que chama os seus métodos de mesmo nome que foram explicitados no capítulo anterior. Os métodos *showQuestion()*, *showError()* e *showMessage()* utilizam telas de mensagens para interação com o usuário. E o método *getTableModel()* é utilizado nas consultas do sistema para a formatação das tabelas de resultado. Vemos o código da classe *AbstractGUI* no apêndice C.

Como nosso sistema está utilizando o mapeamento objeto relacional, os códigos de cadastro, exclusão e alteração estão bastante simples, como veremos adiante. Comentaremos agora o código das implementações das telas do sistema.

5.1 CADASTROS

As telas de cadastro do sistema possuem layout e funcionalidade bastante semelhantes. Elas têm as caixas de texto para a inclusão das informações, caixas de seleção para a escolha de empresas, folhas ou funcionários. E dois botões, um para o cadastro e outro para a saída da tela (ver figuras 25, 26 e 27). A implementação destas telas se resume basicamente ao código do botão de cadastro, onde é criado um objeto da classe e são setados seus atributos com os valores informados pelo usuário na tela para então é passar o objeto para o nosso *DAO*, que cria o objeto no banco de dados. Segue como exemplo, o código do botão de inclusão de funcionários na listagem 12.

```

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    Funcionario f = new Funcionario();
    f.setCpf( Long.valueOf(txtCPF.getText()));
    f.setCtps( txtCtps.getText() );
    f.setDependentes( Short.valueOf(txtDependentes.getText()) );
    f.setEndereco( txtEndereco.getText());
    f.setFuncao(txtFuncao.getText());
    Empresa e = (Empresa) cmbEmpresas.getSelectedItem();
    f.setIdEmpresa(e);
    f.setNome(txtNome.getText());
    f.setSalarioBase( Double.valueOf(txtSalario.getText()));
    GenericDAO dao = new GenericDAO();
    try {
        dao.create(f);
        txtCPF.setText("");
        txtCtps.setText("");
        txtDependentes.setText("");
        txtEndereco.setText("");
        txtFuncao.setText("");
        txtNome.setText("");
        txtSalario.setText("");
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(null, ex.getMessage());
    }
    dao.close();
}

```

Listagem 12 – Código do botão de cadastro de funcionários
 Fonte: Dados da pesquisa (2008)

Temos uma pequena diferença no cadastro de folhas de pagamento, onde a inclusão de uma nova folha de pagamento no sistema faz com que cada funcionário dessa empresa seja adicionado a esta folha, com os lançamentos padrões já feitos. Assim basta cadastrar os lançamentos específicos dos funcionários que por ventura tenham particularidades no mês e se terá a folha pronta.

Isso faz com que a implementação do botão de cadastro desta tela seja um pouco diferente do código da listagem 12, devido a rotina de adição dos funcionários à folha.

```

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    Folha f = new Folha();
    Empresa e = (Empresa) cmbEmpresas.getSelectedItemAt();
    f.setIdEmpresa(e);
    f.setMes(txtMes.getText());
    f.setAno(txtAno.getText());
    f.setObservacao(txtObservacao.getText());
    GenericDAO dao = new GenericDAO();
    try {
        dao.create(f);
        Session session = Database.getInstance().getSession();
        session.lock(e, LockMode.NONE);
        for (Funcionario func : e.getFuncionarioCollection()) {
            FolhaFuncionario folhaFunc = new FolhaFuncionario();
            folhaFunc.setIdFolha(f);
            folhaFunc.setIdFuncionario(func);
            folhaFunc.atualizar();
            dao.create(folhaFunc);
        }
    } catch (Exception ex) {
        showError("Erro ao gravar a folha!");
    }
    txtMes.setText("");
    txtAno.setText("");
    txtObservacao.setText("");
    dao.close();
}

```

Listagem 13 – Código do botão de inclusão de folha

Fonte: Dados da pesquisa (2008)

O cadastro de lançamentos já é um pouco mais diferente e complicado do que os anteriores. A tela de cadastro de lançamentos é a maior do sistema, nela o usuário vai informar os lançamentos específicos para os funcionários que necessitarem. Pode-se informar, por exemplo, as faltas incorridas no mês, adiantamentos recebidos, horas extras trabalhadas e etc.

Para sua correta utilização, o usuário primeiro escolhe uma empresa, depois a folha que ele deseja e o funcionário que teve a movimentação. Selecionado isso, o usuário informa o tipo de lançamento, se é desconto no pagamento do funcionário ou se é vencimentos a receber, a descrição do lançamento e o seu valor. O usuário deve repetir a operação para tantos quantos lançamentos o funcionário tiver. Os únicos lançamentos que não precisam ser informados são: o salário que já vem informado pelo cadastro do funcionário, o salário família, o desconto da previdência e o desconto do imposto de renda, estes três últimos são calculados pelo sistema. Depois de incluído os lançamentos, que apareceram na tela, basta clicar no botão gravar para salvar as informações no banco de dados. A tela que faz esta operação pode ser vista na figura 28.

A tela possui uma aba para a visualização das informações do contra-cheque do funcionário, quando o mesmo é selecionado pode-se observar os seus valores na aba correspondente como mostra a figura 29.

Falando mais agora de sua implementação em si, quando se seleciona um funcionário, vamos buscar no banco os lançamentos específicos dele para poder carregar na lista de movimentações, e atualizamos a visualização do seu contra-cheque. Para tanto, utilizamos a rotina `atualizaContraCheque()` (veja no apêndice D). E funciona da seguinte maneira: se for passado um objeto vazio, a função limpa os campos da tela, se um objeto for passado como parâmetro, é recuperadas as informações do contra-cheque e colocado nas suas caixas de texto específicas. Em seguida a lista com os lançamentos daquele funcionário é atualizada colocando inclusive os lançamentos padrões (salário, salário família, desconto do INSS e IRRF), separando o que é vencimento para a parte de cima da lista e o que é débito para a parte de baixo.

Tendo selecionado a folha do funcionário, o usuário pode adicionar ou remover itens de seus lançamentos. Querendo adicionar basta informar o tipo de lançamento, a descrição e o valor e apertar no botão adicionar.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    FolhaFuncionarioItens ffi = new FolhaFuncionarioItens();
    ffi.setDescricao(txtDescricao.getText());
    ffi.setValor( Double.valueOf(txtValor.getText()) );
    if (rdbVencimentos.isSelected()){
        ffi.setTipo("V");
    }else{
        ffi.setTipo("D");
    }
    if ( itemJaInserido(ffi) ) {
        showError("Item já inserido");
    } else {
        itensInseridos.add(ffi);
        lstItens.updateUI();
    }
}
```

Listagem 14 – Código do botão adicionar da tela de lançamentos

Fonte: Dados da pesquisa (2008)

O código do botão faz apenas a criação de um objeto da classe `FolhaFuncionarioItens`, seta seus atributos, verifica se o mesmo já não está na lista, caso não estando inclui no vetor de `itensInseridos`.

O botão de remover possui código bastante simples, apenas remove o item selecionado do vetor de itens inseridos e atualiza a exibição dos itens na lista.

Depois de atualizada a lista de movimentações, o usuário clica no botão gravar para persistir os dados no banco, o que faz chamar o método da listagem 15.

```
private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    FolhaFuncionario ff =
        (FolhaFuncionario) cmbFuncionario.getSelectedItem();
    GenericDAO dao = new GenericDAO();
    dao.beginTransaction();
    try {
        for (FolhaFuncionarioItens aux :
            ff.getFolhaFuncionarioItensCollection()) {
            dao.delete(aux);
        }
        ff.getFolhaFuncionarioItensCollection().clear();
        for (int i = 0; i < itensInseridos.size(); i++) {
            FolhaFuncionarioItens ffi =
                (FolhaFuncionarioItens) itensInseridos.get(i);
            ffi.setIdFolhafuncionario(ff);
            dao.create(ffi);
            ff.getFolhaFuncionarioItensCollection().add(ffi);
        }
        ff.atualizar();
        dao.update(ff);
        showMessage("Alterações realizadas com sucesso!");
    } catch (Exception ex) {
        dao.rollbackTx();
        showError("Erro ao gravar os lançamentos!");
    } finally {
        dao.close();
    }
    atualizaContraCheque(ff);
}
```

Listagem 15 – Código do botão gravar da tela de lançamentos
Fonte: Dados da pesquisa (2008)

Primeiro apagamos todos os lançamentos que o funcionário tinha para aquela folha, fazendo um laço e apagando um a um. Em seguida varremos o vetor que tem os novos lançamentos informados pelo usuário, criamos cada um deles no banco de dados e adicionamos a lista dos lançamentos da folha do funcionário. Depois de percorrido todo o vetor com os lançamentos corretos, utilizamos o método atualizar() da classe FolhaFuncionario para que, de acordo com as novas movimentações, seja recalculado todos descontos e vencimentos, para podermos assim atualizar no banco de dados a folha do funcionário. Para finalizar o método, chamamos a rotina que faz a visualização na tela das alterações realizadas.

Todo os cálculos trabalhistas são realizados no método atualizar() da classe FolhaFuncionario, que está especificado no apêndice E. Lá é encontrado o salário de contribuição para o INSS que é o salário contratual mais os vencimentos

informados nos lançamentos. O FGTS é calculado a base de 8% do salário de contribuição. O desconto do INSS é calculado de acordo com a tabela e método abordados no terceiro capítulo. Assim como o desconto do IRRF foi explicitado na modelagem do sistema, é feito o seu cálculo. E por último é calculado o salário família de acordo com a tabela vigente. Cálculos realizados, atributos corrigidos, agora o objeto está atualizado.

5.2 CONSULTAS

As telas de listagem e consulta de possuem duas opções de consulta: consulta todos os registros; consulta a por nome. Tem um botão para a escolha do tipo de consulta, uma caixa de texto para se incluir o nome a ser pesquisado, uma tabela que mostrará os resultados obtidos, um botão para consultar e um botão para sair da tela, como exemplo temos a figura 31.

A interface 'Lista de Empresas' apresenta uma barra de busca no topo com o rótulo 'Buscar por'. Abaixo dela, há duas opções de seleção: 'Nome da Empresa' (selecionada com um botão de opção) e 'Todos' (com um botão de opção). Uma caixa de texto para entrada de texto está ao lado de 'Nome da Empresa'. Abaixo das opções, há dois botões: 'Buscar' e 'Sair'. Na parte inferior da interface, há uma tabela com duas colunas: 'Razão Social' e 'Endereço'. A tabela está atualmente vazia.

Figura 31 – Tela para listar empresas
Fonte: Dados da pesquisa (2008)

A implementação destas telas está no botão de consultar que primeiramente define o modelo da tabela que será apresentada mostrando os resultados, e se o usuário tiver escolhido a consulta geral, ira usar o método *findAll()* para recuperar todos os objetos do banco de dados. Caso o usuário escolha a consulta por nome,

é usado o método *findLike()* para se recuperar apenas os objetos com o nome requerido. Tendo o resultado da consulta numa lista, é feito um laço para colocar um a um os objetos na tabela. Veja como exemplo o código da listagem 16.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

    DefaultTableModel model = new DefaultTableModel();
    model.addColumn("Nome");
    model.addColumn("CGC");
    model.addColumn("Endereço");
    List<Empresa> empresas = null;
    if ( rdoTodos.isSelected() ) {
        empresas = findAll(Empresa.class);
    } else {
        empresas = findLike(    "Empresa", "nome",
                               txtRazaoSocial.getText());
    }

    for ( Empresa e : empresas ) {
        String[] linha = new String[3];
        linha[0] = e.getNome();
        linha[1] = String.valueOf( e.getCgc() ) ;
        linha[2] = e.getEndereco();
        model.addRow(linha);
    }
    tblEmpresas.setModel(model);
}
```

Listagem 16 – Código do botão da busca de empresas
Fonte: Dados da pesquisa (2008)

5.3 EXCLUSÕES

Para as telas de exclusão é utilizado caixas de seleção para se escolher o objeto a ser excluído, e dois botões: um para excluir a folha e outro para sair do sistema. Veja a figura 32.

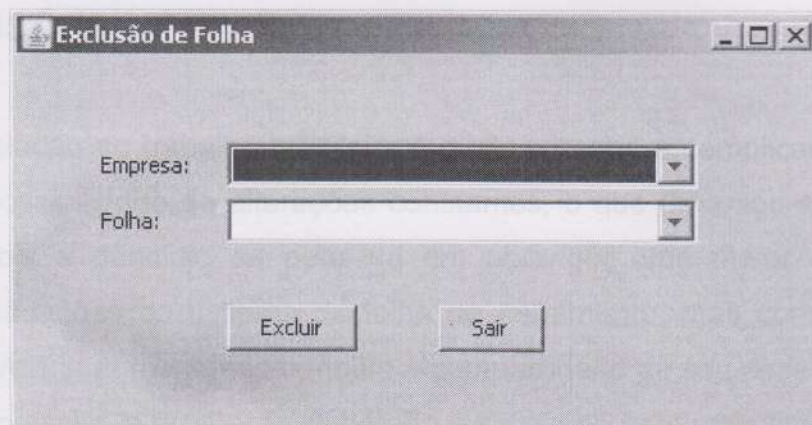


Figura 32 – Tela para excluir folhas
Fonte: Dados da pesquisa (2008)

Depois de selecionado o objeto a ser excluído, basta passá-lo para o DAO que faz a sua exclusão, caso o usuário confirme a operação.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    Folha f = (Folha) cmbFolhas.getSelectedItem();
    Empresa e = (Empresa) cmbEmpresas.getSelectedItem();
    if ( showQuestion("Deseja realmente
                    apagar esta Folha de Pagamento?")) {
        GenericDAO dao = new GenericDAO();
        try {
            dao.delete(f);
            DefaultComboBoxModel model = new
                DefaultComboBoxModel(findQuery("from Folha obj
                    where obj.idEmpresa = " + e.getId()).toArray());
            cmbFolhas.setModel(model);
            cmbFolhas.setSelectedIndex(-1);
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
        dao.close();
    }
}
```

Listagem 17 – Código do botão de exclusão de folha
Fonte: Dados da pesquisa (2008)

Após a exclusão a caixa de seleção com os objetos remanescentes é atualizada.

6 CONCLUSÕES

A legislação do trabalho brasileira é muito extensa e complicada, com várias minúcias e possibilidade de alterações constantes, o que dificultou a realização do programa, pois a princípio se pensava em abranger uma maior quantidade de variáveis e situações do sistema de folha de pagamento, mas com o incremento dessas variáveis ia aumentando também a complexidade de seu desenvolvimento.

Mesmo assim, o programa elaborado é suficiente para um sistema de folha de pagamento simplificado. E possui várias perspectivas de melhorias que ainda podem ser feitas com o devido tempo hábil, como a informação pelo usuário das tabelas de imposto de renda e previdência social, as informações automáticas de férias e rescisões contratuais, e assim por diante. Podendo até um dia fazer tudo o que sistemas bem mais completos existentes comercialmente hoje já fazem, como o controle de ponto e emissão de arquivos exigidos.

A realização deste sistema englobou vários conhecimentos adquiridos durante o curso de Especialização em Desenvolvimento de Sistemas Corporativos demonstrando o bom aprendizado e qualificação de seu autor, servindo, assim, seu propósito.

LIMA, Gleydon A. F. de. Material didático. 2007. Material disponibilizado em aula.

OLIVEIRA, Anísio de. Cálculos Trabalhistas. São Paulo: Atlas. 2004.

OLIVEIRA, Daniel Quirino. Livre-se do SQL: uma introdução ao Hibernate. Disponível em: <<http://www.guj.com.br/java/tutorial/artigo.125.1.guj>>. Acesso em: 10 de fev. de 2008.

STEIL, Rafael. Introdução à programação gráfica em Java com Swing. Disponível em: <<http://www.guj.com.br/java/tutorial/artigo.38.1.guj>>. Acesso em: 10 de fev. de 2008.

UML. Linguagem de Modelagem Unificada. Disponível em: <<http://www.esportilando.com/download.php?cod=2038&categoria=Outras%20Linguagens>>. Acesso em: 10 de fev. de 2008.

REFERÊNCIAS

UML, Resource Page. Disponível em: <<http://www.uml.org>>. Acesso em: 10 de fev. de 2008.

APOSTILA de JAVA. Disponível em:
<<http://www.apostilando.com/download.php?cod=35&categoria=Java>>. Acesso em:
10 de fev. de 2008.

BITENCOURT, Tatyana. **Introdução a Orientação de Objetos**. Disponível em:
<<http://www.guj.com.br/java/tutorial/artigo.149.1.guj>>. Acesso em: 10 de fev. de
2008.

GALHARDO, Raphaela; LIMA, Gleydson A. F. de. **Introdução ao Hibernate**.
Disponível em: <<http://www.j2eebrasil.com.br/mostrar/4>>. Acesso em: 10 de fev. de
2008.

HIBERNATE – Relational Persistence for Java and .NET. Disponível em:
<<http://www.hibernate.org>>. Acesso em: 10 de fev. de 2008.

HIBERNATE Annotations reference guide. Disponível em:
< http://www.hibernate.org/hib_docs/annotations/reference/en/html_single/>. Acesso
em: 10 de fev. de 2008.

JAVA Swing Tutorial. Disponível em: <<http://www.javabeginner.com/java-swing-tutorial.htm>>. Acesso em: 10 de fev. de 2008.

LIMA, Gleydson A. F. de. **Material didático**, 2007. Material disponibilizado em aula.

OLIVEIRA, Aristeu de. **Cálculos Trabalhistas**. São Paulo: Atlas, 2004.

OLIVEIRA, Daniel Quirino. **Livre-se do SQL**: uma introdução ao Hibernate.
Disponível em: <<http://www.guj.com.br/java/tutorial/artigo.125.1.guj>>. Acesso em: 10
de fev. de 2008.

STEIL, Rafael. **Introdução a programação gráfica em Java com Swing**.
Disponível em: <<http://www.guj.com.br/java/tutorial/artigo.38.1.guj>>. Acesso em: 10
de fev. de 2008.

UML Linguagem de Modelagem Unificada. Disponível em:
<<http://www.apostilando.com/download.php?cod=2038&categoria=Outras%20Apostilas>>. Acesso em: 10 de fev. de 2008.

UML Resource Page. Disponível em: <<http://www.uml.org>>. Acesso em: 10 de fev. de 2008.

APÊNDICES

APÊNDICE A – Listagem do arquivo de configuração do *Hibernate*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
  <session-factory name="java:comp/env/hibernate/SessionFactory">
    <property
      name="hibernate.bytecode.use_reflection_optimizer">
        false
      </property>

    <property
      name="hibernate.connection.driver_class">
        org.postgresql.Driver
      </property>

    <property
      name="hibernate.connection.pool_size">
        10
      </property>

    <property
      name="hibernate.connection.url">
        jdbc:postgresql://localhost:5432/FolhaPagamento
      </property>

    <property
      name="hibernate.connection.username">
        postgres
      </property>

    <property
      name="hibernate.connection.password">
        postgres
      </property>

    <property
      name="hibernate.dialect">
        org.hibernate.dialect.PostgreSQLDialect
      </property>

    <property
      name="hibernate.show_sql">
        true
      </property>

    <mapping class="br.curso.gui.dominio.Empresa" />
    <mapping class="br.curso.gui.dominio.Folha" />
    <mapping class="br.curso.gui.dominio.FolhaFuncionario" />
    <mapping class="br.curso.gui.dominio.FolhaFuncionarioItens" />
    <mapping class="br.curso.gui.dominio.Funcionario" />
  </session-factory>
</hibernate-configuration>
```

APÊNDICE B – Listagem da classe DAO

```

public class GenericDAO {

    protected boolean autocommit = true;

    private Transaction tx;

    private Session session;

    public GenericDAO(boolean autocommit) {
        session = getSession();
        this.autocommit = autocommit;
    }

    public GenericDAO() {
        session = getSession();
    }

    protected Session getSession() {
        if (session == null) {
            session = Database.getInstance().getSession();
        }
        return session;
    }

    public void beginTx() {
        tx = getSession().beginTransaction();
    }

    public void commitTx() {
        tx.commit();
    }

    public void rollbackTx() {
        tx.rollback();
    }

    public void close() {
        if (session != null)
            session.close();
    }

    private void change(Object o, char t) throws Exception {
        try {
            if (autocommit)
                beginTx();

            if (t == 'C') {
                session.save(o);
            } else if (t == 'D') {
                session.delete(o);
            } else if (t == 'U') {
                session.saveOrUpdate(o);
            } else {
                throw new Exception("Operação desconhecida");
            }

            if (autocommit)
                commitTx();
        } catch (Exception e) {
            e.printStackTrace();
            throw e;
        }
    }
}

```

```

    }

    public void create(Object o) throws Exception {
        change(o, 'C');
    }

    public void delete(Object o) throws Exception {
        change(o, 'D');
    }

    public void update(Object o) throws Exception {
        change(o, 'U');
    }

    public Collection findAll(Class c) {
        // Session session = Database.getInstance().getSession();
        Collection col = session.createCriteria(c).list();
        // session.close();
        return col;
    }

    public Collection findAll(String entity) {
        Query q = session.createQuery("from " + entity);
        return q.list();
    }

    public Collection findQuery(String query) {
        Query q = session.createQuery(query);
        return q.list();
    }

    @SuppressWarnings("unchecked")
    public Collection findAllIterate(String entidade) {
        // Session session = Database.getInstance().getSession();
        Query q = session.createQuery("from " + entidade);
        Iterator it = q.iterator();
        Collection col = new ArrayList();
        while (it.hasNext()) {
            col.add(it.next());
        }
        // session.close();
        return col;
    }

    public Object findByPrimaryKey(Class c, Serializable id) {
        // Session session = Database.getInstance().getSession();
        Object o = session.get(c, id);
        // session.close();
        return o;
    }

    public Object findByPrimaryKeyLock(Class c, Serializable id) {
        // Session session = Database.getInstance().getSession();
        Object o = session.get(c, id, LockMode.UPGRADE);
        // session.close();
        return o;
    }

    public List findLike(String classe, String field, String valor) {
        Query q = session.createQuery(
            "from " + classe + " obj " +
            " where obj." + field +
            " like '" + valor + "%'");
        return q.list();
    }
}

```

APÊNDICE C – Listagem da classe AbstractGUI

[illegible]

```
public DefaultTableModel getTableModel(String[] colunas,  
                                     String[] fields, Class entity ) {
```

```
    Collection col = findAll(entity);  
    DefaultTableModel model = new DefaultTableModel();  
    for (int i = 0; i < colunas.length; i++) {  
        model.addColumn(colunas[i]);  
    }  
    Iterator it = col.iterator();  
    while ( it.hasNext() ) {  
        Object item = it.next();  
        Vector linha = new Vector();  
        for (int i = 0; i < fields.length; i++) {  
            linha.add(evalProperty(item,fields[i]));  
        }  
        model.addRow(linha);  
    }  
  
    return model;
```

```
    }  
    ...  
}
```

APÊNDICE D – Listagem do método atualizaContraCheque()

```

public void atualizaContraCheque (FolhaFuncionario folhaFunc){
    if (folhaFunc == null){
        lblBaseCalculoFgts.setText("");
        lblBaseCalculoIrrf.setText("");
        lblCgc.setText("");
        lblEmpresa.setText("");
        lblEndereco.setText("");
        lblFolha.setText("");
        lblFuncao.setText("");
        lblFuncionario.setText("");
        lblSalarioBase.setText("");
        lblSalarioContribuicaoInss.setText("");
        lblTotalDescontos.setText("");
        lblTotalLiquido.setText("");
        lblTotalVencimentos.setText("");
        lblValorFgts.setText("");

        itensContraCheque.clear();
        lstContraCheque.updateUI();
    }else{
        Session session = Database.getInstance().getSession();
        session.lock(folhaFunc, LockMode.NONE);
        DecimalFormat duasCasas = new DecimalFormat("0.00");
        lblBaseCalculoFgts.setText( duasCasas.format(
            folhaFunc.getBaseFgts()));
        lblBaseCalculoIrrf.setText( duasCasas.format(
            folhaFunc.getBaseIrrf()));
        lblCgc.setText("Empresa: " + folhaFunc
            .getIdFuncionario().getIdEmpresa()
            .getCgc().toString() + " - ");
        lblEmpresa.setText(folhaFunc.getIdFuncionario()
            .getIdEmpresa().getNome());
        lblEndereco.setText("Endereco: " + folhaFunc
            .getIdFuncionario().getIdEmpresa().getEndereco());
        lblFolha.setText("Pagamento Referente ao mês: " +
            folhaFunc.getIdFolha().getMes() + "/" +
            folhaFunc.getIdFolha().getAno());
        lblFuncao.setText(folhaFunc.getIdFuncionario().getFuncao());
        lblFuncionario.setText("Funcionário: " +
            folhaFunc.getIdFuncionario().getNome() + " (id = "
            + folhaFunc.getIdFuncionario().getId().toString()
            + ")");
        lblSalarioBase.setText( duasCasas.format(
            folhaFunc.getIdFuncionario().getSalarioBase()));
        lblSalarioContribuicaoInss.setText( duasCasas.format(
            folhaFunc.getSalarioContribuicaoInss()));
        lblTotalDescontos.setText( duasCasas.format(
            folhaFunc.getTotalDescontos()));
        lblTotalLiquido.setText( duasCasas.format(
            folhaFunc.getTotalLiquido()));
        lblTotalVencimentos.setText( duasCasas.format(
            folhaFunc.getTotalVencimentos()));
        lblValorFgts.setText( duasCasas.format(
            folhaFunc.getValorFgts()));
    }
}

```

```

itensContraCheque.clear();
FolhaFuncionarioItens aux = new FolhaFuncionarioItens();
aux.setDescricao("Salário Base");
aux.setTipo("V");
aux.setValor(folhaFunc.getIdFuncionario().getSalarioBase());
itensContraCheque.add(aux);
if (folhaFunc.getSalarioFamilia() > 0.0) {
    aux = new FolhaFuncionarioItens();
    aux.setDescricao("Salário Família");
    aux.setTipo("V");
    aux.setValor(folhaFunc.getSalarioFamilia());
    itensContraCheque.add(aux);
}
if (folhaFunc.getFolhaFuncionarioItensCollection() != null) {
    for (FolhaFuncionarioItens ffi : folhaFunc
        .getFolhaFuncionarioItensCollection()) {
        if(ffi.getTipo().equalsIgnoreCase("V")){
            itensContraCheque.add(1,ffi);
        }else{
            itensContraCheque.add(ffi);
        }
    }
}

if (folhaFunc.getDescontoInss() > 0.0) {
    aux = new FolhaFuncionarioItens();
    aux.setDescricao("Desconto INSS");
    aux.setTipo("D");
    aux.setValor(folhaFunc.getDescontoInss());
    itensContraCheque.add(aux);
}

if (folhaFunc.getDescontoIrrf() > 0.0) {
    aux = new FolhaFuncionarioItens();
    aux.setDescricao("Desconto IRRF");
    aux.setTipo("D");
    aux.setValor(folhaFunc.getDescontoIrrf());
    itensContraCheque.add(aux);
}

lstContraCheque.updateUI();
session.close();

```

APÊNDICE E – Listagem do método atualizar()

```

public void atualizar(){

    // Primeiro calcula-se o salario de contribuicao
    this.salarioContribuicaoInss = this.idFuncionario.getSalarioBase();
    this.totalVencimentos = this.idFuncionario.getSalarioBase();
    this.totalDescontos = 0.0;

    if (this.folhaFuncionarioItensCollection != null){
        for (FolhaFuncionarioItens ffi :
            this.folhaFuncionarioItensCollection) {
            if (ffi.getTipo().equalsIgnoreCase("V")){
                this.salarioContribuicaoInss += ffi.getValor();
                this.totalVencimentos += ffi.getValor();
            }else{
                this.salarioContribuicaoInss -= ffi.getValor();
                this.totalDescontos += ffi.getValor();
            }
        }
    }

    // Calculo do FGTS
    this.baseFgts = this.salarioContribuicaoInss;
    this.valorFgts = this.baseFgts * 0.08;
    // Calculo do desconto do INSS
    if (    this.salarioContribuicaoInss > 0 &&
        this.salarioContribuicaoInss <= 868.29){
        // Percentual = 7.65%
        this.descontoInss = this.salarioContribuicaoInss * 0.0765;
    }
    if (    this.salarioContribuicaoInss > 868.29 &&
        this.salarioContribuicaoInss <= 1140.0){
        // Percentual = 8.65%
        this.descontoInss = this.salarioContribuicaoInss * 0.0865;
    }
    if (    this.salarioContribuicaoInss > 1140.0 &&
        this.salarioContribuicaoInss <= 1447.14){
        // Percentual = 9%
        this.descontoInss = this.salarioContribuicaoInss * 0.09;
    }
    if (    this.salarioContribuicaoInss > 1447.14 &&
        this.salarioContribuicaoInss <= 2894.28){
        // Percentual = 11%
        this.descontoInss = this.salarioContribuicaoInss * 0.11;
    }
    if (this.salarioContribuicaoInss > 2894.28 ){
        // Percentual = 11%
        this.descontoInss = 2894.28 * 0.11;
    }

    // Calculo do desconto do IRRF
    this.baseIrrf = this.salarioContribuicaoInss - this.descontoInss;
    this.baseIrrf -= (132.05 * this.idFuncionario.getDependentes());

    if (this.baseIrrf <= 1313.69){
        // Percentual = 0%
        this.descontoIrrf = 0.0;
    }
    if (this.baseIrrf > 1313.69 && this.baseIrrf <= 2625.12){

```

```
// Percentual = 15%
this.descontoIrrf = (this.baseIrrf * 0.15) - 197.05;
if (this.descontoIrrf <= 10.0){
    this.descontoIrrf = 0.0;
}
}

if (this.baseIrrf > 2625.12 ){
    // Percentual = 27.5%
    this.descontoIrrf = (this.baseIrrf * 0.275) - 525.19;
    if (this.descontoIrrf <= 10.0){
        this.descontoIrrf = 0.0;
    }
}

// Calculo do salario familia
this.salarioFamilia = 0.0;
if (    this.salarioContribuicaoInss > 0 &&
        this.salarioContribuicaoInss <= 449.93){
    // Valor = 23.08
    this.salarioFamilia = 23.08 *
        this.idFuncionario.getDependentes();
}
if (    this.salarioContribuicaoInss > 449.93 &&
        this.salarioContribuicaoInss <= 676.27){
    // Valor = 16.26
    this.salarioFamilia = 16.26 *
        this.idFuncionario.getDependentes();
}

this.totalVencimentos += this.salarioFamilia;
this.totalDescontos += (this.descontoInss + this.descontoIrrf);
this.totalLiquido = this.totalVencimentos - this.totalDescontos;
}
```