UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOSÉ FRANCISCO VIANA PINHEIRO

UM FRAMEWORK PARA PERSISTÊNCIA DE OBJETOS EM BANCO DE DADOS RELACIONAIS

NITERÓI
2005

JOSÉ FRANCISCO VIANA PINHEIRO

UM FRAMEWORK PARA PERSISTÊNCIA DE OBJETOS EM BANCO DE DADOS RELACIONAIS

Dissertação apresentada ao Curso de Pós-Graduação em Computação da Universidade Federal Fluminense, como requesito parcial para obtenção do Grau de Mestre. Área de Concentração: Processamento Distribuído e Paralelo.

Orientadora: Prof[a] Dr[a] MARIA LUIZA D'ALMEIDA SANCHEZ

Niterói
2005

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

P654 Pinheiro, José Francisco Viana
Um framework para persistência de objetos em banco de dados relacionais / José Francisco Viana Pinheiro. – Niterói, RJ : [s.n.], 2005.
207 f.

Orientador: Maria Luiza D'Almeida Sanchez.
Dissertação (Mestrado em Ciência da Computação) – Universidade Federal Fluminense, 2005.

1. Banco de dados relacionais. 2. Framework orientado a objetos. 3. Metadados. 4. Programação orientada a objetos (Computação). I. Título.

CDD 005.756

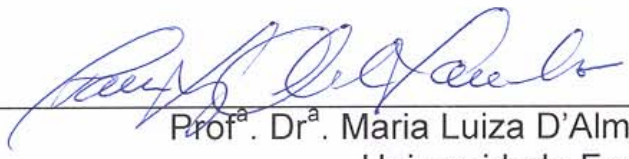
JOSÉ FRANCISCO VIANA PINHEIRO

UM FRAMEWORK PARA PERSISTÊNCIA DE OBJETOS EM BANCO DE DADOS RELACIONAIS

Dissertação apresentada ao Curso de Pós-Graduação em Computação da Universidade Federal Fluminense, como requesito parcial para obtenção do Grau de Mestre. Área de Concentração: Processamento Distribuído e Paralelo.

Aprovada em Janeiro de 2005

BANCA EXAMINADORA



Prof[a]. Dr[a]. Maria Luiza D'Almeida Sanchez – Orientadora
Universidade Federal Fluminense

Prof[a]. Dr[a]. Profa. Débora Christina Muchaluat Saade
Universidade Federal Fluminense

Prof. Dr. Alcione de Paiva Oliveira
Universidade Federal de Viçosa

Niterói
2005

À minha esposa Ana Lea, pelo incentivo, compreensão e amor.

À minha filha Júlia, apesar de estar ainda no ventre de sua mãe, é fonte de inspiração na realização deste trabalho.

AGRADECIMENTOS

À professora Maria Luiza D'Almeida Sanchez, minha orientadora, pela confiança em mim depositada e pela perspicácia de suas observações.

Ao Vinícius Valente Maciel pela sua valiosa contribuição.

Ao inestimável amigo e colega da INFOR/PREVI, Marcelo Pires Augusto, pela amizade e suporte durante os meus afastamentos em função deste trabalho.

Ao amigo Marco Antônio Allemão pelo seu incentivo e apoio através de sua experiência acadêmica.

À minha família, por compreender minha ausência no decorrer deste curso e me apoiarem nos momentos mais difíceis.

Ao Wilson Pumar de Paula, Osmar Agostinho Chagas Fernandes, gerentes da área de informática e Marcos dos Anjos Teixeira, gerente de equipe da INFOR/PREVI durante esse meu esforço, por terem viabilizado a realização deste curso.

Finalmente agradeço a Deus por permitir mais esta realização na minha vida.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

RESUMO

O uso da metodologia de orientação a objetos se proliferou no desenvolvimento de software, provocando uma mudança na estruturação e organização da informação. Contudo, a maioria das aplicações demanda o armazenamento e a recuperação de informações em um mecanismo de persistência. Devido à prevalência do banco de dados relacionais no gerenciamento de dados, seu uso é freqüentemente exigido, em vez dos bancos de dados de objetos, pois são perceptível a maturidade e confiabilidade dos SGBDs adquirida após anos de desenvolvimento e ajustes de desempenho.

Uma aplicação orientada a objetos que utilize um banco de dados relacional deve ter a capacidade de recuperar seus dados para a memória local e retorná-los, em tempo de execução. Várias questões surgem neste trânsito devido à incompatibilidade entre as representações orientadas a registros e orientadas a objetos, tais como relacionamentos, concorrência, transparência e acoplamento. As propostas de solução para este desencontro tecnológico convergem para o conceito de uma camada de abstração de acesso a dados, diminuindo o acoplamento da aplicação em relação ao mecanismo de armazenamento de dados.

Este trabalho propõe um framework que realiza a persistência de objetos em um banco de dados relacional, que trata a incompatibilidade entre estas tecnologias através de um modelo de mapeamento objeto relacional. A sua utilização é transparente, dispensando o programador do aprendizado da sintaxe de comandos SQL e da tecnologia de armazenamento de dados através da utilização de conceitos de reflexão e manipulação de *bytecodes*.

Palavra-chave: Persistência de Objetos, Mapeamento Objeto Relacional, Banco de Dados Relacionais, Metadados, Framework, Linguagem de Programação Orientada a Objetos.

ABSTRACT

The use of orientation to the methodology of objects has proliferated in the development of software causing a change in the information structure and organization. However, most applications demand storage and retrieval of information in a mechanism of persistence. Due to prevalence of relational database on data management, its use is frequently demanded instead of an object database since maturity and reliability of SGBDs are perceivable and acquired after years of development and performance adjustments.

An object oriented application using a relational database must have the capacity to retrieve its data to local memory and return them on time of execution. Several issues arise in this road due to incompatibility among record and object oriented representations such as relationships, competition, transparency and coupling. Solution proposals for this technological clash converge to the concept of a data access abstraction layer, decreasing the application coupling with reference to the data storage mechanism.

This work proposes a framework that performs the object persistence in a relational database which treats incompatibility among these technologies by means of a relational mapping model of objects. Its use is transparent, exempting the programmer from the SQL command syntax learning process and from the data storage technology by using reflection and *bytecodes* manipulation concepts.

Key-word: Object Persistence, Object to Relational Mapping, Relational Database, Metadata, Framework, Object Oriented Programming Language.

1. INTRODUÇÃO

O paradigma de Orientação a Objetos (OO) é cada vez mais empregado nas etapas de análise, projeto e implementação de sistemas de informação. Utiliza conceitos tradicionais da engenharia de software como: baixo acoplamento, alta coesão, encapsulamento da informação, herança e polimorfismo que aumentam o grau de manutenibilidade do software. O modelo do sistema concebido com base no paradigma OO organiza o software como uma coleção de objetos que incorporam tanto a estrutura quanto o comportamento dos dados (RUMBAUGH,1994).

O modelo relacional vem sendo utilizado para implementação de persistência dos dados de uma aplicação. Por definição, no modelo relacional, os dados são organizados segundo um conjunto de tabelas e as operações sobre estas utilizam uma linguagem não procedural – SQL - que manipula a álgebra relacional (teoria dos conjuntos), ou seja, manipula conjuntos de uma só vez (MACHADO,1995). Este modelo é utilizado para o gerenciamento de grandes volumes de dados e oferece pesquisas rápidas, integridade referencial, compartilhamento de informações de forma segura e o gerenciamento de acessos. Isto determina uma predominância de gerenciadores de bancos de dados relacionais.

Com o uso cada vez mais freqüente da modelagem orientada a objetos, surge uma série de questões decorrentes do desencontro entre a representação OO e o modelo relacional utilizado para o armazenamento das informações. O modelo relacional se preocupa em moldar a complexidade do sistema em tabelas,

relacionamentos e formas normais, ao tempo que o paradigma OO divide o problema em classes de objetos.

A lógica da aplicação, que representa os processos de negócio, é projetada e implementada utilizando ferramentas orientadas a objetos. Já a informação tratada pelos processos de negócio utiliza SQL para armazenar, recuperar e manipular dados em uma base de dados relacional. Portanto, cada informação recuperada deve passar por um processo de tradução de sua representação original para sua representação no modelo OO. Inversamente, para dados representados no modelo OO que devem ser persistentes, isto é, neste caso salvos no banco de dados, deverá ocorrer à tradução da informação da representação OO para a representação relacional.

Na busca da solução para a redução das distâncias destas duas tecnologias surgem propostas de padrões de projeto/programação e *frameworks*. (HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1, 2003; COCOBASE O/R MAPPING – BASIC CONCEPTS AND QUICK START GUIDE, 2001; LARMAN, 2000; ZIMBRÃO, 2003; KELLER, 1997 e LERMEN, 1998). Um framework é um subsistema extensível para um conjunto de serviços relacionados, como por exemplo, frameworks para interfaces gráficas de usuário ou frameworks para persistência (LARMAN,2000). Um framework para persistência é um conjunto de classes reutilizáveis – e usualmente extensíveis – que fornecem serviços para objetos persistentes. Estes serviços incorporam a tradução de objetos para registros, quando salvos em banco de dados e a conversão de registros para objetos, quando recuperados de banco de dados. O framework proporciona um alto nível de reutilização de software, evitando assim a replicação de códigos relacionados à persistência de objetos por toda aplicação.

O objetivo deste trabalho é a definição de um framework que realize a persistência de objetos em um banco de dados relacional, tratando este "descasamento de impedância" (*impedance mismatch*) existente entre o modelo Relacional e o modelo de Orientação a Objetos (ZIMBRÃO,2003). A sua principal contribuição é a transparência proporcionada aos programadores na manipulação de informações de objetos em um mecanismo de persistência.

1.1 MOTIVAÇÕES

O desafio de unir a tecnologia de desenvolvimento de softwares baseados na orientação a objetos com a tecnologia de armazenamento de dados relacional de forma transparente para o desenvolvimento de sistemas é a grande motivação deste trabalho. A impedância entre estas tecnologias nos traz problemas já consagrados na engenharia de software e que devem ser implementados de forma transparente, sem comprometer a individualidade das tecnologias.

A diferença de impedância entre as tecnologias é evidenciada na necessidade dos sistemas de guardar as informações de objetos em um meio de armazenamento duradouro. Persistência, neste trabalho, é o processo pelo qual a informação, após a sua criação ou processamento, é salva em um mecanismo de guarda prolongado e durável. Desta forma, seu conteúdo é mantido por diversas execuções da aplicação.

Um mecanismo de persistência é qualquer tecnologia usada para armazenar permanentemente objetos para posterior atualização, recuperação, e/ou exclusão. Mecanismos de persistência podem ser arquivos, banco de dados relacionais, banco de dados objeto relacionais, banco de dados hierárquicos, etc. Mudanças de versões, migrações de fornecedores e troca de mecanismos de persistência ocorrem em aplicações ao longo de sua evolução. Esta diversidade de tipos de mecanismos de persistência torna aplicações orientadas a objeto dependentes de uma tecnologia de persistência, prejudicando uma eventual migração de tecnologia de armazenamento de objetos.

A disseminação da tecnologia da orientação a objetos esbarra na predominância atual dos gerenciadores de banco de dados relacionais. A transição perfeita do projeto para a implementação de um sistema orientado a objetos seria uma realidade se a linguagem e o banco de dados fossem orientados a objetos.

É considerável a quantidade de bases de dados que estão implementadas e consolidadas em modelos relacionais e são utilizadas por sistemas desenvolvidos em diferentes metodologias de desenvolvimento. O alto custo de conversão dos dados existentes para um banco de dados orientados a objetos aliado ao risco de

prejudicar o funcionamento dos sistemas atuais inviabiliza o abandono definitivo da tecnologia de banco de dados relacionais. As principais vantagens da tecnologia de banco de dados relacional são sua maturidade, diversidade de fornecedores e suporte disponíveis no mercado e a possibilidade das aplicações legadas continuarem trabalhando nos mesmos dados que são acessados pelas novas aplicações orientadas a objetos (AMBLER,1998;LERMEN,1998;WHITENACK,2002).

O desenvolvimento de sistemas seguindo a tecnologia orientada a objetos e o armazenamento de informações em bancos de dados relacionais requer que profissionais da tecnologia da orientação a objetos dominem a tecnologia de banco de dados e resolvam impedâncias entre estas tecnologias. Este fato eleva o custo do projeto de desenvolvimento de aplicações em função do custo mais elevado de mão de obra especializada e da prorrogação do prazo na resolução de problemas de impedância.

A exemplo de banco de dados relacionais, os dispositivos de armazenamento de dados têm profissionais responsáveis por sua administração. No ambiente de banco de dados temos o administrador de banco de dados responsável pela otimização e administração do banco de dados. Os administradores de banco de dados podem adicionar e alterar estruturas no banco de dados seja por motivos de manutenção, otimização ou atualização de versão. Estas alterações podem afetar estruturas usadas por aplicações orientadas a objeto, obrigando a realização de manutenções desnecessárias no ponto de vista do desenvolvedor OO, uma vez que a origem da manutenção não está no escopo do negócio do sistema ou da tecnologia utilizada na sua implementação.

Atualmente as organizações têm várias opções de arquitetura utilizadas em sua área de informática que vão desde *mainframes* centralizados a arquiteturas cliente/servidor de duas camadas ou uma arquitetura de n camadas com objetos distribuídos. Similar a migrações de tecnologias de armazenamento de dados, as migrações de arquitetura de ambientes computacionais também são comuns no setor de informática. A camada de persistência deve ser um componente que não prejudique este tipo de migração.

1.2 SOLUÇÕES EXISTENTES

A elaboração de um framework como solução para a persistência de objetos exige a pesquisa de tecnologias existentes destinadas à mesma finalidade. A percepção de carências tecnológicas nas soluções atuais conduz a contemplação de aspectos que enriquecerão o caráter inovador do novo framework.

1.2.1 Mecanismos de Persistência de Objetos

Segundo Srinivasan (1997) os mecanismos de persistência de objetos são classificados em três classes:

- GOP (Gateway-based Object Persistent) adiciona dados persistentes a programas orientados a objetos usando formas tradicionais de armazenamento de dados não orientados a objetos como banco de dados relacionais, banco de dados hierárquicos e arquivos.
- ORDBMS (Object-Relational DBMS) amplia o popular modelo de dados relacional, adicionando características do modelo orientado a objetos.
- OODBMS (Object-Oriented DBMS) adiciona suporte a persistência de objetos em uma linguagem programação orientada a objetos como Smaltalk ou C++.

1.2.1.1 GOP

Este sistema é usado para suportar um modelo de programação orientada a objetos que usa armazenamento de dados não orientados a objetos para armazenar informações de objetos. Para tanto, os sistemas GOP fazem um mapeamento entre o ambiente OO na camada de aplicação e o ambiente de armazenamento de dados. Em tempo de execução estes sistemas traduzem os objetos de sua representação usada no ambiente de armazenamento de dados para a sua representação no ambiente da aplicação e vice-versa. Para facilitar o uso, o sistema GOP faz este

processo de tradução de forma transparente para o programador da aplicação (exceto durante o processo de mapeamento que é feito pelo programador). GOP é usado por vários produtos como VisualAge C++ Data Access Builder, SMRC, ObjectStore Gateway, Persistence, UniSQL/M, Gemstone/Gateway e Subtleware/SQL. Esta solução é essencialmente uma camada intermediária, sendo que a aplicação e os dados permanecem independentes (SRINIVASAN,1997). A padronização deste tipo de persistência tem sido feito pela OMG.

O GOP é uma abordagem *middleware* adequada à integração de diversos sistemas de informação empresariais e oferece um framework comum para a elaboração de aplicações OO.

O gerenciamento do compartilhamento, distribuição, heterogeneidade e neutralidade da linguagem de persistência de objetos de negócio é outra característica do GOP. A principal vantagem de elaborar uma aplicação GOP é que as aplicações legadas continuam trabalhando sobre os mesmos dados acessados pelas novas aplicações. O GOP oferece acesso orientado a objeto para dados não orientados a objetos, mas em contrapartida, tem dificuldades no armazenamento de objetos complexos em um sistema de banco de dados legado. Objetos complexos são objetos que contém atributos que são objetos da mesma ou de outra classe (SRINIVASAN,1997). As desvantagens no mapeamento de modelos orientados a objetos em um banco de dados não orientados a objetos entre outras são: desempenho ruim e lógica complexa. Aplicações GOP, porém, têm a flexibilidade de acessar outros OODBMSs para o armazenamento de objetos complexos nativamente enquanto acessa e atualiza banco de dados legados.

1.2.1.2 ORDBMS

A premissa do ORDBMS é que a melhor solução para a persistência de objetos consiste em estender o modelo relacional. Este tipo de abordagem estende tanto a modelagem de banco de dados relacional quanto a linguagem SQL, deixando a tecnologia de banco de dados atual intacta. Existem duas classes de

banco de dados objeto relacional: aqueles que foram feitos do zero e aqueles que foram feitos a partir de um banco de dados relacional já existente.

O X3H2 (comitê americano responsável pela especificação dos padrões SQL) tem trabalhado na extensão de objetos em SQL desde de 1991. Estas extensões fazem parte do rascunho do padrão SQL denominado SQL3 (SRINIVASAN, 1997).

ORDBMS é uma abordagem *bottom-up*, sendo dados (ou banco de dados) o centro. O problema de impedância no DBMS relacional é resolvido oferecendo suporte para as características de modelo de dados das principais linguagens orientadas a objeto. Porém, o modelo de dados que é usado pelas aplicações pode ser parecido, mas não é idêntico ao modelo de dados usado no DBMS para armazenar dados das aplicações. ORDBMSs supera estes problemas oferecendo a capacidade de executar consultas complexas e um rico suporte para executar porções da aplicação no servidor do banco de dados. Adicionalmente, oferece melhor robustez, concorrência e recuperação de erros dentre as três abordagens (SRINIVASAN, 1997).

1.2.1.3 OODBMS

Object-oriented DBMSs(OODBMSs) foi elaborado com o princípio de que o melhor caminho de adicionar persistência a objetos é fazer objetos persistentes usando uma linguagem de programação orientada a objetos(OOPL) como o C++ ou Smalltalk. Em função de OODBMS ter seus princípios em linguagens orientadas a objeto, estes são freqüentemente referenciados como um sistema de linguagem de programação persistente. OODBMSs, porém, vai muito além de simplesmente adicionar persistência a qualquer linguagem de programação orientada a objetos. Por isso, historicamente, muitos OODBMSs são feitos para servir o mercado de aplicações computer-aided design/manufacturing(CAD/CAM) cujos recursos como navegação de acesso, versões e transações longas são extremamente importantes. OODBMS, porém, suporta aplicações avançadas de banco de dados orientado a objeto com recursos como suporte a objetos persistentes de mais de uma linguagem de programação, distribuição de dados, modelos de transação avançados, versões,

evolução de schemas e geração dinâmica de novos tipos. Ainda que muitos destes recursos tenham pouco a ver com orientação a objeto, OODBMS enfatiza-os em seus sistemas e aplicações. OODBMSs têm sido padronizados pela ODMG(Object Database Management Group), um consórcio dos principais fornecedores de OODBMS (SRINIVASAN, 1997).

O OODBMS é uma abordagem *top-down*, sendo a aplicação (ou linguagem de programação) o fator central. É a melhor abordagem para o armazenamento de aplicações orientadas a objetos, possibilitando uma persistência sem emendas do ponto de vista da linguagem de programação. OODBMS evita o problema da impedância através de extensivo suporte às características de modelo de dados para uma ou mais linguagens de programação orientada a objetos. Portanto, o modelo de dados usado na aplicação é idêntico ao implementado no DBMS. Porém, OODBMS não oferece recursos de consultas tão sofisticados quanto o ORDBMS. Testes de desempenho mostram que as taxas de transação obtidas no OODBMSs não se aproximam às altas taxas conseguidas pelo DBMS relacional no processamento de transações padrão. Aplicações que necessitem de excelente desempenho de navegação, que não tenham consultas complexas e aspectos de integridade e segurança que não são críticos, são melhores em OODBMSs.

1.2.2 Tipos de Implementações de Persistência de Objetos

Scott W. Ambler em seu trabalho (AMBLER,2000b) discute os diferentes tipos de implementações de persistência de objetos atualmente utilizados.

1.2.2.1 Codificação SQL nas Classes da Aplicação

Este tipo de implementação é a mais comum e atraente para os programadores, na qual as operações de persistência são elaboradas através de códigos SQL embutidos diretamente nos códigos fontes das classes da aplicação. Esta abordagem é viável para pequenas aplicações e protótipos, tendo em vista a rapidez obtida no desenvolvimento do aplicativo. Porém, a desvantagem deste tipo

de implementação é a dependência das classes com códigos SQL embutido em relação à estrutura do banco de dados no modelo relacional e vice-versa. Qualquer manutenção realizada na estrutura dos objetos do banco de dados relacional proporciona um esforço de uma nova codificação nos códigos SQL embutidos nas classes.

1.2.2.2 Codificação SQL em Classes de Dados

Esta é uma abordagem levemente melhor em relação a anterior na qual comandos SQL são encapsulados em uma ou mais “classes de dados”. Novamente, esta abordagem é mais apropriada para protótipos e pequenos sistemas com menos de 40 ou 50 classes, pois continua tendo o problema do impacto de alterações na estrutura de bancos de dados relacionais provocar manutenções na codificação das classes, só que neste caso apenas nas classes de dados. O avanço que podemos observar nesta nova abordagem é o código referente à persistência dos objetos encapsulado em um lugar: nas classes de dados.

1.2.2.3 Camada de Persistência

Esta estratégia de persistência parte do conceito do desenvolvimento de aplicações em camadas como demonstrado na Figura 1.

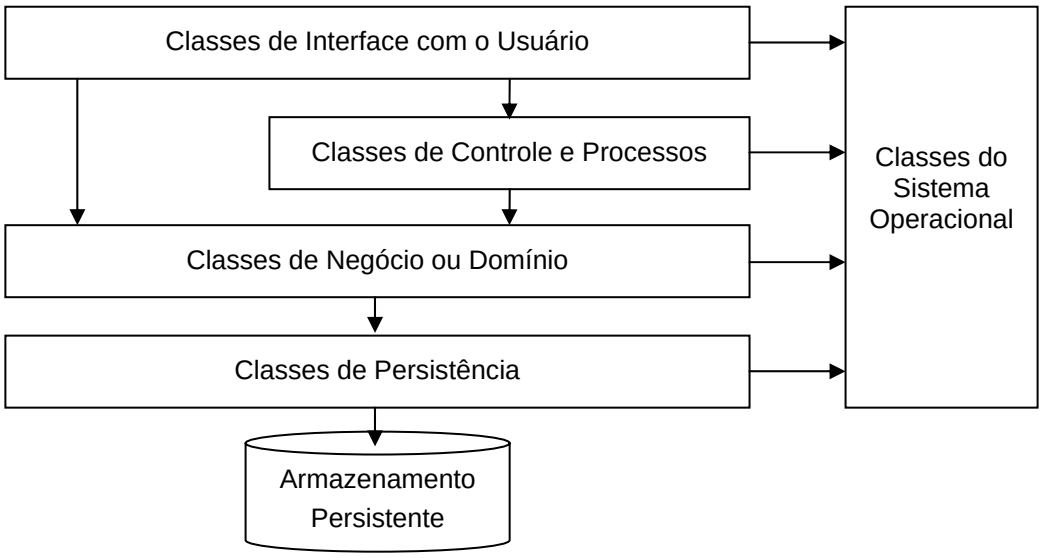


Figura 1 - Arquitetura em camadas

A arquitetura apresentada na figura anterior separa as classes de uma aplicação em cinco tipos: Classes de Interface com o Usuário, Classes de Controle e Processos, Classes de Negócio ou Domínio e Classes de Persistência (AMBLER2000b;AMBLER,1998).

A idéia básica é que a classe de uma camada pode interagir com outras classes da mesma camada ou com classes da camada adjacente. A codificação da aplicação nesta arquitetura facilita manutenções e otimizações por causa da grande redução do acoplamento das funcionalidades da aplicação.

1.2.2.3.1 Classes de Interface

A camada de interface com usuários é formada por classes que implementam telas e relatórios. Classes de Interface de Usuários podem mandar mensagens para a camada de negócio ou domínio, para a camada de controle ou processo ou para a camada de sistema Operacional.

1.2.2.3.2 Classes de Negócio ou Domínio

A camada de negócio ou domínio implementa classes que representam elementos do negócio ou domínio da empresa. Por exemplo, a camada de negócio de uma empresa de telecomunicações pode conter classes como Consumidor e Telefonema.

1.2.2.3.3 Classes de Controle e Processos

A camada de controle e processos implementa classes responsáveis pela lógica do negócio que envolve a colaboração de diversas classes de negócio ou domínio ou até outras classes de controle e processos. Um exemplo de classes de Controle e Processos pode ser o gerenciamento de contas de telefones que pode interagir com instâncias da classe Consumidor, Telefonema e Plano de Ligações.

1.2.2.3.4 Classes de Sistema Operacional

A camada de Sistema Operacional implementa classes que fornecem interações com funcionalidades do sistema operacional como impressão e correio eletrônico.

1.2.2.3.5 Classes de Persistência

O nível de persistência reúne todo o comportamento relacionado ao armazenamento de objetos no mecanismo de persistência que pode ser, por exemplo, arquivos ou banco de dados relacionais.

Esta estratégia proporciona uma organização das classes que reduz o acoplamento das classes de negócio e de interface em relação à persistência. O isolamento de classes responsáveis pelo armazenamento de objetos possibilita a

reorganização do mecanismo de persistência sem precisar alterar o código fonte de outras camadas.

1.3 OBJETIVO

O objetivo deste trabalho é elaborar um framework para persistência de objetos que realize o armazenamento e recuperação de objetos em um meio de armazenamento de banco de dados relacional. Deve ser de fácil utilização, transparente e minimizar a influência da persistência na codificação de programas.

O framework, seguindo uma arquitetura em camadas, será um conjunto de classes na camada de persistência com o objetivo de oferecer serviços de persistência à camada de negócio para os objetos que, por sua natureza, devem ser armazenados. Sendo uma camada independente, possibilita eventuais migrações ou mudanças no dispositivo de armazenamento de dados, sem provocar mudanças nas demais camadas da aplicação. Esta ferramenta dispensa o programador do aprendizado da tecnologia de armazenamento de dados, diminuindo o prazo e os custos no desenvolvimento de sistemas.

A seguir, detalharemos alguns aspectos a serem resolvidos na implementação do framework proposto.

1.3.1 Persistência no Modelo Relacional

Banco de dados relacionais é uma tecnologia bem madura no gerenciamento de grandes volumes de dados com bom desempenho e alta confiabilidade. A adoção de banco de dados puramente orientados a objetos tem como desvantagens o pequeno mercado para o fornecimento deste tipo de banco de dados, não suportar transações relacionais de sistemas legados e a impossibilidade de atender aplicações baseadas em SQL, como sofisticadas ferramentas de suporte a decisões (*business inteligence*) que têm aumentado seu espaço nas grandes organizações.

Capacidade de recuperação e armazenamento seguro de grandes volumes de dados, algoritmos de otimização de desempenho de consulta, dispositivos de controle de integridade de dados e controle de concorrência de acesso de dados são alguns dos serviços já disponíveis nos gerenciadores de banco de dados relacionais. A utilização de uma base de dados relacional permite transferir para o servidor de banco de dados a responsabilidade pelos serviços acima citados. Algoritmos de implementação destes serviços não são necessários na camada de persistência de dados, uma vez que o gerenciador de banco de dados já realiza tais tarefas com sucesso comprovado. A adoção desta opção de armazenamento de dados reduz o tamanho da camada de persistência.

1.3.2 Mapeamento Objeto-Relacional

Uma arquitetura de persistência de objetos que adota um banco de dados relacional como o meio de armazenamento de dados deverá ter uma correlação que traduza o modelo orientado a objetos para um modelo relacional devido às diferenças de representação destes modelos.

A recuperação do conteúdo das propriedades de um objeto só é possível com a existência de uma indicação do caminho que indique sua correspondência na representação relacional. A existência deste caminho é no mapeamento.

Um mecanismo de persistência de objetos se baseia nesta correspondência para a recuperação e manipulação das propriedades do objeto.

Este processo de tradução objeto-relacional deve ser flexível e prever que atributos de classe podem ser relacionados a colunas de diferentes tabelas, por exemplo. Em contrapartida, segue convenções e regras que evitem um alto grau de complexidade na compatibilização dos diferentes modelos.

O mapeamento objeto-relacional resolve características da modelagem orientada a objetos e adapta para o modelo relacional, como por exemplo: classes, atributos, hierarquias, objetos complexos e agregações devem ser resolvidos com tabelas, visões, colunas e relacionamentos no modelo relacional. O framework deve

ser capaz de interpretar e resolver diferentes situações de impedância através de *joins* e outros artifícios disponíveis.

A implementação do mapeamento Objeto-Relacional será em uma camada à parte e documentado em um padrão acessível a outras tecnologias de framework. Esta característica, além de diminuir o esforço de construção do sistema, obriga os profissionais de TI a documentar o modelo OO e relacional em um formato padrão e compreensível a outras tecnologias.

1.3.3 Transparência

Transparência é a qualidade da implementação de persistência de dados que caracteriza a sua utilização como silenciosa, isto é, o programador realiza as operações de manipulação de dados de objetos em sua programação sem perceber que por trás destas manipulações existe uma arquitetura provendo estas funcionalidades.

A persistência do objeto é mais transparente quando as funções são embutidas no objeto, sem a necessidade de poluir o código fonte do objeto com mapeamentos e outras funções não pertinentes ao seu escopo. As funções de persistência devem se resumir a operações de inclusão, atualização, consulta e exclusão de determinado(s) objeto(s). Critérios de consulta, atualização e exclusão também farão parte deste conjunto de funcionalidades.

1.3.4 Transparência no Tratamento de Coleções

A transparência na manipulação individual ou coletiva de objetos persistentes é uma característica a ser alcançada em um *framework*. A indiferença no tratamento e recuperação única ou coletiva de objetos persistentes liberta o programador da preocupação da escolha do método a ser utilizado para manipular o(s) objeto(s) em tempo de codificação.

Muitas soluções tratam a recuperação e manipulação individual e coletiva de objetos de forma distinta obrigando o programador a conhecer formas de manipulação e individual e coletiva de objetos e julgar a implementação mais adequada a cada trecho de sua aplicação. Um formato único e transparente de tratamento deste aspecto da persistência é uma qualidade a ser alcançada no *framework* proposto.

1.3.5 Controle de Estados

O estado do objeto na persistência reflete a relação entre a representação do objeto em memória e sua representação no meio persistente. O framework deve apoiar o desenvolvedor no controle do estado do objeto para garantir ao programador o tratamento adequado ao objeto. Como por exemplo, quando um objeto é recuperado e alterado. Se o desenvolvedor possibilitar em sua programação o abandono das alterações do objeto sem que estas sejam efetivadas no meio persistente, o framework alertará a respeito do estado do objeto em relação a sua representação no meio persistente, impedindo a perda das alterações efetuadas.

1.3.6 Concorrência

O framework adotará uma estratégia de controle de concorrência de acessos ao meio persistente que garanta a diversas pessoas trabalharem simultaneamente com um objeto, sem ter o risco destes usuários sobrescreverem o trabalho de outros.

Esta estratégia envolverá o controle de colisões, no qual perceberá que um objeto que está sendo editado foi modificado por outro processo, informando tal ocorrência.

1.3.7 Independência Arquitetônica

O framework terá um modelo que contempla um baixo nível de acoplamento em relação à aplicação usuária de seus serviços. Quanto menor for a influência dos serviços oferecidos pelo framework sobre a codificação da aplicação orientada a objetos, menor será o impacto das evoluções e alterações de tecnologias do framework sobre a aplicação. Esta característica vislumbra também a possibilidade da adaptação de outras classes de sistemas de persistência de objetos além do GOP, como o ORDBMS e OODBMS, sem causar impactos na implementação da aplicação.

1.3.8 Acoplamento

A evolução tecnológica da computação exige uma flexibilidade cada vez maior das arquiteturas computacionais. Soluções que exigem padronizações rígidas e de grande influência sobre a implementação submetem a aplicação a uma alta dependência tecnológica. Quando a aplicação migrar para uma tecnologia mais recente, impactos sobre a decodificação são inevitáveis.

O mesmo acontece em um framework de persistência. Framework fortemente acoplado ao aplicativo significa uma dependência estratégica do cliente em relação ao fornecedor ou mantenedores do framework. Caso o framework seja descontinuado ou imponha severas alterações em uma nova versão, a codificação do aplicativo que o utiliza estará comprometida.

1.4 MÉTODO DE TRABALHO

O projeto de elaboração do framework foi organizado em quatro etapas, iniciado pelo aprendizado das tecnologias envolvidas, explorando suas características e propondo um modelo de compatibilização. Um modelo inicial é criado e aplicado a um estudo de caso. Durante a aplicação do modelo inicial ao estudo de caso, surgem mudanças e adaptações, resultando na versão final do framework.

1.4.1 Estudo de Caso

Elaborar uma aplicação com um modelo de dados relacional e um modelo UML que contemple os vários aspectos de impedância entre as tecnologias. Estudar propostas e soluções para cada aspecto da impedância objeto-relacional percebida, chegando a um modelo inicial de framework.

1.4.2 Framework

Formalizar a proposta de um novo framework através da documentação do seu modelo de classes e definição dos serviços oferecidos. Estudar e implementar soluções para a incorporação das funcionalidades do framework nos objetos persistentes sem a intervenção do desenvolvedor.

1.4.3 Aplicar o Framework ao Estudo de Caso

O modelo inicial do framework é aplicado ao estudo de caso, surgindo mudanças e adaptações durante o processo. O modelo inicial é alterado conforme as novas situações forem surgindo e sendo resolvidas.

1.4.4 Formalização da Tese

Organizar nos demais capítulos desta tese as anotações feitas durante as etapas anteriores e os conceitos pesquisados. Complementar com as conclusões finais e propor possíveis melhoramentos do framework proposto e sugerir trabalhos futuros.

1.5 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação é dividida em seis capítulos.

O capítulo 1 dá uma introdução ao tema, apresentando as motivações e objetivos desta dissertação e descrevendo o método de trabalho adotado.

O capítulo 2 expõe conceitos utilizados na elaboração deste trabalho, abrangendo inclusive comentários a respeito das soluções de mercados que tratam a persistência de objetos.

O capítulo 3 explana a proposta do framework, descrevendo o seu modelo e funcionamento. Este capítulo é complementado por uma descrição dos processos de desenvolvimento de sistemas, utilizando o framework de persistência de objetos.

O capítulo 4 relata o estudo de caso ao qual o framework foi submetido, bem como a sua avaliação sob diversos aspectos.

Finalmente, os capítulos 5 e 6 apresenta as conclusões e sugestões para trabalhos futuros respectivamente.

2 CONCEITO

Este capítulo apresenta uma breve revisão de alguns conceitos já consagrados da orientação a objetos e do modelo relacional. Após esta revisão, serão apresentadas soluções atualmente utilizadas para o mapeamento objeto-relacional. Em seguida, serão apresentados aspectos da implementação da persistência e finalmente discutiremos as soluções utilizadas no mercado de informática que seguem uma tecnologia semelhante ao framework proposto para a persistência de objetos.

2.1 ORIENTAÇÃO A OBJETO

A orientação a objetos é um paradigma de desenvolvimento de sistemas que abstrai a realidade a ser automatizada em termos de objetos que incorporam tanto informações como procedimentos.

No mundo real identificam-se várias entidades com atributos e operações em comum. A análise orientada a objetos empacota os atributos, operações e relacionamentos das entidades identificadas, criando os objetos. Um objeto é um conceito, uma abstração, algo com limites nítidos e significados no contexto do problema em causa (RUMBAUGH,1994). Os objetos que possuem as mesmas características de atributos, comportamento, relacionamentos e semântica são agrupados e categorizados em Classes.

Os relacionamentos de objetos são classificados como associações, generalização (hierarquia), agregação e composição. As características do relacionamento entre objetos são: opcionalidade e cardinalidade.

2.2 MODELO RELACIONAL

O conceito do modelo entidade relacionamento destina-se prioritariamente ao projeto de banco de dados, no desenvolvimento de estruturas de dados (MACHADO,1995).

O objetivo da modelagem de dados é possibilitar a apresentação de uma visão única, não redundante e resumida dos dados de uma aplicação. No desenvolvimento de aplicações em banco de dados, o modelo relacional é o mais usado para a representação e entendimento dos dados que compõem a essência de um sistema de informações.

Peter Chen formulou a proposta do modelo de Entidade Relacionamento destacando a importância de reconhecer os objetos que compõem este negócio, independentemente de preocupar-se com formas de tratamento das informações, procedimentos, programas e outros (MACHADO,1995). Estes objetos a serem descobertos e modelados em um sistema foram classificados em Entidade e Relacionamentos.

Define-se entidade como um objeto que existe no mundo real com uma identificação distinta e com um significado próprio e um conjunto de informações com a mesma característica.

Toda entidade é descrita com propriedades que são descritas por atributos e valores. Os atributos e seus valores, juntos, descrevem as instâncias de uma entidade.

As entidades não estão soltas, isoladas uma das outras, mas relacionadas de forma a mostrar a realidade em um contexto lógico.

Define-se relacionamento na modelagem relacional como o fato ou acontecimento que liga duas entidades existentes no mundo real. Similar ao relacionamento da orientação a objetos, suas características são: obrigatoriedade e cardinalidade (um para um, um para muitos e muitos para muitos)

No ambiente de banco de dados implementamos as entidades como tabelas, suas instâncias como registros e atributos como colunas. Os relacionamentos são mapeados em regras de integridade implementadas através de *constraints* de chaves estrangeira no banco de dados. A integridade referencial deve garantir que quando um registro se relaciona, este deve ter uma referência válida para outro registro em outra tabela. Esta referência é implementada por cópias de colunas implementadas na tabela cujo único objetivo é referenciar a chave primária de outro registro em outra tabela.

2.3 MAPEAMENTO OBJETO-RELACIONAL

O mapeamento é o ato de se determinar como os objetos e seus relacionamentos são persistidos em um mecanismo de armazenamento de dados permanente seguindo um modelo relacional (AMBLER,2000a). Diversos aspectos do ambiente da orientação a objeto e do mecanismo persistente, no caso o banco de dados relacional, devem ser considerados ao se elaborar um mapeamento objeto-relacional.

Os artigos de Zimbrão(2003), Ambler(2000a), Polak(2001) e Keller(1997) discutem a questão do mapeamento objeto-relacional. Um aspecto apresentado é a relação entre as propriedades de uma classe a colunas de tabelas do banco de dados. Geralmente a relação entre classe e tabela, bem como, propriedade e coluna são implementadas de forma direta.

Os relacionamentos entre objetos devem ser implementados como chaves estrangeiras no ambiente de banco de dados, com peculiaridades em relação à cardinalidade. Estruturas hierárquicas têm alternativas de mapeamento que dependem da situação do modelo orientado a objetos que se deseja implementar.

Discutiremos a seguir propostas de soluções de mapeamento de relacionamentos no modelo orientado a objeto para um modelo relacional de banco de dados.

2.3.1 Estruturas Hierárquicas

Bancos de dados relacionais não suportam hierarquia nativamente. Este princípio faz com que se torne necessário o mapeamento das estruturas hierárquicas presentes na orientação a objeto para o ambiente de banco de dados. A forma de organizar uma hierarquia de classes em um banco de dados prevê três técnicas (AMBLER,1998): mapeamento de toda a estrutura hierárquica em uma única tabela, mapeamento de cada classe concreta em uma tabela correspondente e mapeamento de cada classe em sua tabela.

Tabela Única para toda a Hierarquia

Nesta técnica todos os atributos de todas as classes da estrutura hierárquica são mapeados para uma única tabela. Cada linha é um objeto de uma subclasse específica, diferenciada na tabela através de uma coluna extra que indica a qual classe se refere. Esta coluna extra, que denominamos código de tipo, pode implementar o polimorfismo se vislumbramos a possibilidade de acrescentar códigos que representam combinações de classes. Uma linha, por exemplo, pode ser da classe PessoaFisica, da classe Funcionario ou de ambos. Scott Ambler (AMBLER,2000a) sugere nestes casos de combinação de classes a substituição da coluna de códigos de tipos por colunas lógicas. O exemplo anterior pode ser implementado por duas colunas lógicas: isPessoaFisica e isFuncionario.

A vantagem desta abordagem é a simplicidade. Para adicionar classes, só é necessário acrescentar colunas para as novas informações. Como já mencionado, esta técnica implementa o polimorfismo. O fato das informações estarem em uma única tabela proporciona um bom desempenho e facilita a elaboração de consultas ad-hoc.

O forte acoplamento em função de todas as classes estarem implementadas em uma mesma tabela representa uma desvantagem nesta abordagem. Uma mudança em uma classe pode afetar a tabela, abalando outras classes na hierarquia. Existe também o potencial desperdício de espaço no banco de dados, tendo em vista que nem todas as colunas estão preenchidas. A indicação do tipo (classe) para cada linha da tabela pode se tornar complexa quando houver sobreposição de tipos. A tabela pode aumentar demasiadamente no caso de grandes hierarquias.

Esta abordagem é aconselhável para simples hierarquias de classes que não apresentem sobreposição entre os tipos.

Uma Tabela para cada Classe Concreta[1]

Cada classe concreta é implementada na correspondente tabela. A tabela implementa, além dos atributos da classe, os atributos herdados da super classe.

A vantagem desta abordagem é a facilidade de consultas ad-hoc, pois todas as informações necessárias de uma classe estão em uma única tabela. Apresenta bom desempenho no acesso a dados de objetos.

A manutenção é prejudicada pois, ao alterar uma super classe, além de modificar a tabela correspondente a classe, se faz necessária à alteração das tabelas de suas sub classes. Quando um objeto muda de papel (classe), é necessário copiar seus dados para a apropriada tabela e conceder uma nova identificação. O suporte a múltiplos papéis é complexo pois nesta abordagem existe a mesma informação da superclasse é replicada em várias tabelas referentes à classe concreta. Esta replicação requer um esforço extra na manutenção da integridade da informação entre as tabelas.

Esta abordagem é apropriada para aplicações que raramente sofrem mudanças e sobreposição de tipos (classes).

Uma Tabela para cada Classe

---

[1] Classes concretas são aquelas que podem ser instanciadas.(AMBLER,2000a).

Cada classe da estrutura hierárquica é implementada em uma correspondente tabela.

Esta abordagem é fácil de entender pois a correlação entre atributos, classes, colunas e tabelas é direta. Suporta o polimorfismo apropriadamente, tendo em vista que o objeto pode ser armazenado nas tabelas correspondentes aos papéis exercidos, ou melhor classes. A manutenção é simples para a inclusão de novas super ou sub classes, bastando a modificação ou criação da correspondente tabela. Não há desperdício de espaço no banco de dados pois seu espaço aumenta na mesma proporção do número de objetos persistidos.

Esta abordagem proporciona a criação do maior número de tabelas para a mesma hierarquia dentre as três apresentadas e, conseqüentemente, relacionamentos. O desempenho é potencialmente mais lento, tendo em vista que para as operações de consulta e atualização é necessário percorrer um maior número de tabelas. Consultas ad-hoc se tornam mais complexas, a não ser que se crie visões que simulem as tabelas desejadas.

A implementação de uma tabela por classe é adequada quando existem ocorrências de mudança e sobreposição de tipos.

2.3.2 Relacionamentos de Objetos

Na metodologia orientada a objetos, além da hierarquia, existem mais três tipos de relacionamentos entre objetos: associação, agregação e composição. Relacionamentos na tecnologia de orientação a objetos são implementados através de referências a objetos e correspondentes operações de manipulação.

Apesar da existência de peculiaridades entre os tipos de relacionamentos em relação à integridade referencial (AMBLER,2003b), a serem posteriormente discutidas no item 2.4.3, o mapeamento destes é realizado da mesma maneira. Apresentamos a seguir classificações de relacionamento de objetos que realmente influenciam o mapeamento objeto-relacional.

A primeira classificação se refere à cardinalidade do relacionamento entre objetos (AMBLER,2003c):

- **relacionamento um para um** - este é um relacionamento onde a cardinalidade de cada um dos componentes é no máximo um. Esta cardinalidade é implementada na orientação a objetos com uma referência, uma operação "get" e uma operação "set".

- **relacionamento um para muitos** - este é um relacionamento onde a cardinalidade do objeto corrente é no máximo um e a cardinalidade do outro objeto pode ser um ou mais. Esta cardinalidade através de um atributo do tipo coleção e operações para a manipulação de seus elementos integrantes.

- **relacionamento muitos para muitos** – este é um relacionamento onde a cardinalidade de cada um dos componentes é um ou mais. Sua implementação na orientação a objetos é idêntica a cardinalidade um para muitos.

A segunda classificação se refere ao direcionamento do relacionamento de objetos (AMBLER,2003c):

- **relacionamentos unilaterais** - um relacionamento unilateral é quando o objeto tem uma referência a outro objeto, mas este não contém uma referência ao primeiro.

- **relacionamentos bidirecionais** - um relacionamento bidirecional existe quando os objetos envolvidos no relacionamento se referenciam.

Portanto podemos afirmar que existem cinco possibilidades de relacionamentos no ambiente orientado a objetos. Porém um dos aspectos da incompatibilidade entre a tecnologia da orientação a objetos e a modelagem relacional do banco de dados é a inexistência de relacionamentos bidirecionais no modelo relacional. No modelo relacional todos os relacionamentos entre tabelas são unilaterais, pois somente uma das tabela contém uma ou mais colunas que referenciam a chave primária da outra tabela.

Os relacionamentos na tecnologia de banco de dados relacionais são implementados através do uso de chaves estrangeiras. Uma chave estrangeira é um objeto de banco de dados associado a uma tabela que garante a consistência referencial entre colunas de sua tabela em relação a colunas componentes de uma chave primária de outra tabela. Em relacionamentos com cardinalidade 1 para 1, deve-se escolher uma das tabelas para implementar a chave estrangeira. Em relacionamentos com cardinalidade muitos para 1, a chave estrangeira deve ser implementada na tabela que esteja na extremidade cuja cardinalidade seja muitos. Em relacionamentos com cardinalidade muitos para muitos, deve-se criar uma tabela denominada associativa para registrar as associações entre as tabelas. A tabela associativa contém duas chaves estrangeiras, correspondentes às chaves primárias das duas tabelas envolvidas no relacionamento.

O mapeamento do relacionamento de objetos no mundo da modelagem relacional deve seguir a cardinalidade. O mapeamento de um relacionamento de objetos de cardinalidade um para um deve ser implementado através de um relacionamento 1 para 1 no ambiente de banco de dados e na mesma relação para as cardinalidades muitos para um e muitos para muitos.

O mapeamento de relacionamento deve conter as seguintes informações

- Relacionamento do Objeto
- Classe Origem
- Classe Destino
- Cardinalidade
- Regra de Leitura, Gravação e Inclusão
- Tabela que implementa o relacionamento no ambiente de banco de dados
- Coluna que implementa o relacionamento no ambiente de banco de dados
- Atributo da classe origem que implementa o relacionamento

A seguir discutiremos as peculiaridades na implementação do mapeamento de relacionamento entre objetos para cada tipo de cardinalidade.

Cardinalidade Um para Um

Existe um problema na maneira em que este tipo de relacionamento é mapeado no banco de dados. Embora este relacionamento seja de cardinalidade um para um entre os objetos, sua implementação no banco de dados através de chave estrangeira só garante a cardinalidade em uma das extremidades. Isto se deve pelo fato da chave estrangeira ser implementada em uma das tabelas envolvidas no relacionamento.

Cardinalidade Um para Muitos

Não podemos considerar que todas as chaves estrangeiras se baseiam em uma chave primária de outra tabela para manter a integridade referencial. A chave estrangeira pode se basear em uma chave candidata de outra tabela. Neste caso deve-se considerar que a coluna a ser atualizada não corresponde à chave primária da outra tabela e sim em uma coluna integrante de uma chave candidata.

Cardinalidade Muitos para Muitos

Para implementar relacionamentos de objetos com cardinalidade muitos para muitos se deve utilizar o conceito de tabelas associativas que tem por objetivo manter o relacionamento entre duas tabelas. Esta solução faz com que este tipo de cardinalidade tenha a peculiaridade de ter uma entidade extra no ambiente relacional para ser mapeada. Duas classes que tenham este tipo de relacionamento necessitam de três tabelas para persistir, o que acarreta um esforço extra em sua implementação.

Existem ainda os relacionamentos recursivos, também denominados auto-relacionamentos no ambiente relacional do banco de dados, que ocorrem quando a mesma entidade (objetos e tabelas) está nas duas extremidades do relacionamento.

A implementação da persistência neste tipo de relacionamento segue as mesmas características já mencionadas neste documento, dependendo também da cardinalidade.

2.3.3 Propriedades Estáticas

Existem atributos de classes que podem ser implementados de forma a atender não somente a uma instância, mas a qualquer instância de objetos. Em função de este atributo ter um valor constante para todos os objetos desta classe, este deve ser mapeado de forma diferente. Existem três maneiras de mapear este tipo de atributo (AMBLER,2003c):

Uma Tabela por Atributo

O atributo estático é mapeado em uma única tabela contendo uma única coluna e uma única linha. Apesar da vantagem de ser uma solução simples e que proporciona fácil acesso pode resultar na disseminação de várias pequenas tabelas no modelo de banco de dados.

Uma Tabela por Classe

Os atributos estáticos de uma classe são mapeados em uma única tabela contendo uma única linha em correspondentes colunas. Mesmas vantagens e desvantagens da solução anterior, divergindo somente na vantagem de implementar um menor número de pequenas tabelas no modelo de banco de dados.

Uma Tabela para Todas as Classes

Os atributos estáticos de todas as classes são mapeados em uma única tabela contendo uma única linha em correspondentes colunas. Esta solução reduz o número de tabelas a serem criadas no ambiente de banco de dados, mas apresenta potenciais problemas de concorrência no acesso e atualização das informações contidas considerando que desta tabela é acessada em função de várias classes.

Uma Tabela para Todas as Classes com Estrutura Genérica

Os atributos estáticos de todas as classes são mapeados em uma única tabela com uma estrutura genérica, isto é, contendo o nome da classe, o nome da propriedade e o valor e o tipo da propriedade. A estratégia minimiza o número de tabelas a serem criadas no banco de dados e evita o problema de concorrência. Porém, por ser uma estrutura genérica, tem a necessidade da conversão de valores pois o tipo da coluna correspondente ao valor da propriedade é único (geralmente character). A estrutura genérica com estas três colunas pode acarretar um forte acoplamento com o nome das classes e suas propriedades estáticas. Uma estrutura mais normalizada pode diminuir este acoplamento.

O mapeamento objeto-relacional é um componente estático na persistência de um objeto em um modelo de banco de dados relacional. Apesar de fundamental, a persistência de objetos tem aspectos dinâmicos que devem ser resolvidos. A seguir estes aspectos serão apresentados e discutidos.

2.4 IMPLEMENTAÇÃO DA PERSISTÊNCIA DE OBJETOS

Muitos artigos na atualidade discutem aspectos da persistência de objetos e maneiras de implementá-las buscando a forma mais eficiente e prática na realização das etapas necessárias ao controle da recuperação e manutenção das informações pertinentes a um objeto em um meio persistente (PORTO,1998; AMBLER,2000a, 2003a,2003c; KELLER,1997; LERMEN,1998; MACIEL,2001; ZIMBRÃO,2003). Apresentamos nesta sessão comentários a respeito de artigos pesquisados que abordam questões da persistência de objetos.

2.4.1 Sincronismo

A arquitetura de Porto (1998) tem como objetivo diminuir o impacto nas transações de clientes informando, tão cedo possível, a respeito de atualizações realizadas por outras transações que alteram o valor dos objetos quando editados por clientes. Esta funcionalidade pode ser sumarizada em três partes: a identificação das atualizações sob objetos ou tabelas, o registro das atualizações e a

comunicação da atualização através de mensagens. O *overhead* ocasionado pela divulgação de mensagens associado à impossibilidade de se vislumbrar em diversos bancos de dados a comunicação de atualizações através de mensagens torna esta solução pouco provável ou até inviável sob o ponto de vista deste trabalho.

2.4.2 Controle de Concorrência

O controle de concorrência tem por objetivo minimizar a possibilidade de ocorrer problemas provocados no acesso simultâneo a entidades compartilhadas, podendo ser objetos, registros de dados, ou qualquer tipo de representação.

Um importante conceito a ser levantado é que em modernos projetos de desenvolvimento de software o controle de concorrência e transações não está simplesmente no domínio do banco de dados, são pertinentes a todas as camadas de arquitetura.

Para entender como implementar o controle de concorrência deve-se entender os conceitos básicos de colisão para que seja possível evitá-los ou detectá-los e resolvê-los da melhor maneira (AMBLER,2003a).

Uma colisão ocorre quando duas atividades tentam alterar entidades dentro de um sistema de registros e provocam interferência entre si.

Existem três abordagens para se evitar colisões:

- Bloqueio pessimista, no qual se evita colisões mas reduz o desempenho do sistema.
- Bloqueio otimista, no qual se detecta colisões e se resolve o que fazer
- Bloqueio excessivamente otimista, que ignora totalmente a colisão.

Bloqueio Pessimista

Bloqueio pessimista é a abordagem onde uma entidade é bloqueada no banco de dados enquanto sua cópia está sendo usada em memória por uma

aplicação, geralmente sob a forma de um objeto (AMBLER,2003a). Um bloqueio previne que outros usuários possam trabalhar nesta mesma entidade no banco de dados simultaneamente. Existem dois níveis de bloqueio: escrita e leitura. Um bloqueio de escrita indica que o proprietário do bloqueio pretende atualizar a entidade e desabilita a todos de ler, atualizar e excluir esta entidade. Um bloqueio de leitura indica que o proprietário do bloqueio não irá atualizar a entidade durante o bloqueio, permitindo outros usuários a ler a entidade, mas não permite atualizá-la ou excluí-la. O escopo de um bloqueio pode ser sobre todo o banco de dados, uma tabela, uma coleção de linhas ou uma simples linha.

As vantagens do bloqueio pessimista são: a facilidade de implementação; e a garantia que suas alterações no banco de dados serão feitas de forma consistente e segura. A desvantagem primária é que esta solução não é escalonável. Quando um sistema tem vários usuários, ou quando a transação envolve um grande número de entidades, ou quando transações são de longa duração, as chances de haver esperas pela liberação de um bloqueio aumentam. Portanto isto limita o número de usuários simultâneos que o seu sistema suporta, podendo provocar um colapso no uso do sistema.

Bloqueio Otimista

Em sistemas de vários usuários é comum ter situações onde colisões são comuns. Apesar de dois usuários estarem trabalhando com o mesmo objeto, cada um deles pode estar trabalhando em instâncias diferentes deste mesmo objeto, não ocorrendo colisões. Quando ocorre este tipo de fato, o bloqueio otimista se torna uma estratégia de controle de bloqueio viável. A idéia é aceitar a ocorrência freqüente de colisões e como forma de prevenção se escolhe simplesmente detectá-la e resolvê-la quando esta suceder (AMBLER,2003a).

A Figura 2 descreve a lógica de atualização de um objeto quando um se adota um bloqueio otimista.

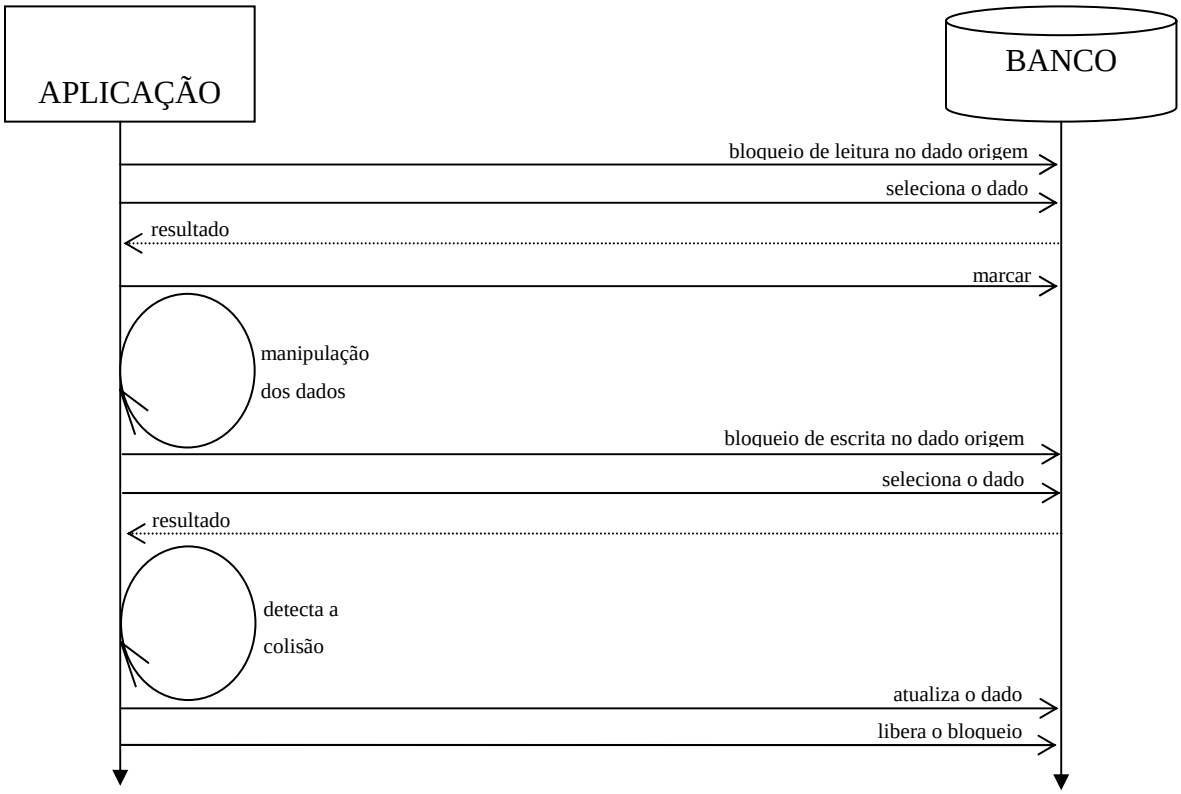


Figura 2 – Bloqueio Otimista

A aplicação lê o objeto em memória. Um bloqueio de leitura é obtido sobre o dado, o dado é lido para a memória, e o bloqueio é liberado. Neste ponto a(s) linha(s) pode(m) ser marcada(s) para facilitar a detecção de uma colisão em um momento posterior. A aplicação então manipula o objeto até o ponto que precisar atualizá-lo, quando então obtém o bloqueio de escrita sobre o dado e lê novamente o registro original verificando se houve colisão. Se não houve colisão então atualiza o registro e libera o bloqueio. Havendo colisão, por exemplo o registro foi atualizado por outro processo após a captura para a memória do original, então a colisão precisa ser resolvida.

Existem duas estratégias básicas para determinar se a colisão ocorreu:

- Marcar a origem com um identificador único.
- Reter uma cópia do original.

A Figura 2 descreve um acesso ingênuo, e de fato existem maneiras de reduzir o número de interações com o banco de dados. As primeiras três requisições ao banco de dados - o bloqueio inicial, marcação do dado original (se for o caso) e o desbloqueio - podem ser feitos em uma única transação. As próximas duas interações, para bloquear e obter uma cópia do dado original, podem ser facilmente combinadas com um simples acesso ao banco de dados. Além disso, a atualização e a liberação do bloqueio podem ser similarmente combinadas. Outra maneira de melhorar este processo é combinar as quatro últimas interações em uma simples transação e simplesmente efetuar a detecção da colisão no servidor de banco de dados ao invés de realizá-la no servidor de aplicação.

Temos cinco estratégias básicas que podem ser aplicadas para resolver colisões no bloqueio otimista (AMBLER,2003a):

- Desistir;
- Mostrar o problema e deixar o usuário decidir;
- Combinar as alterações;
- Registrar o problema para que alguém possa decidir em um momento posterior;
- Ignorar a colisão e sobrescrever.

É importante reconhecer que a granularidade de uma colisão é decisiva. Assuma que dois usuários estejam trabalhando sobre a mesma instância de uma entidade denominada Consumidor. Suponha que o primeiro usuário esteja atualizando o nome do Consumidor e o segundo usuário esteja atualizando o shopping de sua preferência. Efetivamente ocorre uma colisão no nível de entidade, os usuários atualizaram o mesmo consumidor, mas não no nível de atributo. É comum detectar colisões potenciais no nível de entidade, mas seria mais eficiente se esta detecção fosse no nível de atributo.

Algumas abordagens propõem que este controle seja feito através da implementação na base de dados da informação de timestamp (AMBLER2003a).

Quando um dado é recuperado, o timestamp do registro ou, do campo é armazenado em memória. No momento em que a mesma informação é atualizada na base de dados, o timestamp em memória é comparado com o timestamp que consta na base de dados. Caso haja divergência, o sistema notifica o usuário informando que alguém atualizou o dado. Esta solução apresenta um overhead na atualização e recuperação de dados, além de poluir o modelo de dados com a criação de colunas de controle do timestamp.

Bloqueio Excessivamente Otimista

Nesta estratégia não se evita ou detecta colisões, considera-se que colisões jamais ocorrerão. Esta situação é apropriada para sistemas de usuários únicos, onde é garantido que os dados serão acessados por somente um usuário ou processo de cada vez. Estes sistemas são raros, mas existem. Esta abordagem é totalmente imprópria para sistemas que atendem a vários usuários.

2.4.3 Integridade Referencial

Integridade referencial tem por objetivo garantir que se uma entidade se refere à outra, então esta outra deve existir. Em projetos de banco de dados relacionais a regra de integridade referencial estabelece que cada valor não nulo de uma chave estrangeira deve ser igual ao valor de alguma chave primária. Tendo em vista que banco de dados relacional implementa de forma relativamente simples a integridade referencial, a lógica do negócio é implementada no código da aplicação e suas regras de integridade são implementadas no banco de dados (AMBLER,2003b).

O desenvolvimento moderno de software não implementa mais desta maneira a integridade referencial. As linguagens atuais como C++ e Java implementam entidades denominadas objetos. Como resultado, integridade referencial se tornou um item dentro do código da aplicação bem como no banco de dados. Banco de dados relacionais tem se aperfeiçoado drasticamente, suportando linguagens de programação nativas para escrever *stored procedures*, *triggers* e até linguagens

padrão de objetos como o Java. Atualmente é viável implementar a lógica do negócio no banco de dados bem como no código da aplicação. A melhor maneira de visualizar esta nova realidade é o fato de se ter opções de onde implementar a integridade referencial e a lógica do negócio: no banco de dados ou na aplicação.

Discutiremos a seguir aspectos relacionados à integridade referencial a serem considerados na persistência de objetos.

2.4.3.1 Representação Múltipla de Entidades

Uma entidade pode ser representada de diferentes maneiras em uma arquitetura de várias camadas. Por exemplo, dados de consumidor podem ser visualizados em uma página HTML, ser usado para criar um objeto consumidor que reside em um servidor de aplicação e ser armazenado em um banco de dados. Quando se usa a tecnologia de orientação a objetos em conjunto com a tecnologia relacional de banco de dados temos a situação de se implementar estruturas similares em dois lugares: no esquema de objetos como classes que têm inter-relacionamentos e no esquema de banco de dados como tabelas que têm inter-relacionamentos. A existência da questão da integridade referencial em ambos esquemas é evidente.

Quando a mesma entidade é representada em vários esquemas, mesmo que seja esquema de dados ou de objetos, existe o problema de integridade referencial do conteúdo destas representações entre os diferentes esquemas.

O banco de dados que armazena os dados referentes aos objetos é o local “oficial”, onde todas as alterações realizadas em seu conteúdo devem ser consideradas, sobrepondo qualquer informação divergente oriunda das aplicações que utilizam o banco de dados.

2.4.3.2 Controle de Relacionamento entre Objetos

Uma técnica comum de assegurar a integridade referencial é o uso de operações que, a partir de uma ação em uma determinada entidade, propaguem esta mesma ação nas demais entidades relacionadas. Estas operações em cascata são implementadas em *triggers* de banco de dados para a implementação de operações em cascata. Uma operação em cascata no banco de dados ocorre quando uma ação em uma tabela dispara um *trigger* que por sua vez cria uma ação similar em outra tabela, a qual pode por sua vez disparar outro *trigger* que provocará ações semelhantes sucessivamente. Operações em cascata, quando bem implementadas, efetuam de forma automática o gerenciamento de relacionamentos. Existem três tipos comuns de operações em cascata no banco de dados (AMBLER,2003b).

- *Exclusão em Cascata* - a exclusão de linhas em uma determinada tabela provoca a exclusão de linhas de outras tabelas que referenciam esta tabela através de chave estrangeira.

- *Inclusão em Cascata* - a inclusão de uma nova linha em uma determinada tabela provoca a criação de uma linha em outra tabela que referencia esta tabela através de chave estrangeira.

- *Alteração em Cascata* - a atualização de uma linha em uma determinada tabela provoca atualizações em outras tabelas cujos registros sofram alguma interferência de negócio em função da alteração de registros desta tabela.

O conceito de operações em cascata é aplicável a relacionamentos entre objetos, substituindo registros por instâncias de objetos e tabelas por classes, porém com algumas peculiaridades, descritas a seguir (AMBLER,2003b):

- *Exclusão em Cascata em Objetos* - a exclusão de um determinado objeto provoca a exclusão de outros objetos relacionados a este objeto.

- *Leitura em Cascata em Objetos* - quando se recupera um determinado objeto, objetos que são referenciados por seus atributos complexos devem também ser recuperados.

- *Gravação em Cascata em Objetos* - quando um objeto é criado, objetos que são referenciados por seus atributos complexos devem também ser criados automaticamente.

Um framework de persistência pode ser sofisticado o suficiente a ponto de contemplar operações em cascata automáticas, baseando-se na configuração prevista para os relacionamentos no mapeamento objeto relacional.

Existem algumas implicações importantes ao se utilizar operações em cascata (AMBLER,2003b):

- *Estratégia de Implementação* - a partir da existência de um relacionamento deve-se definir se existe a necessidade de uma operação em cascata e, caso exista, onde deverá ser implementada esta operação: no banco de dados ou na camada da aplicação, com os objetos.

- *Precauções com ciclos* - Um ciclo ocorre quando um ciclo de cascata volta ao seu início.

- *Precauções com operações de cascata fora de controle* - Apesar de operações em cascata ser um artifício sofisticado, podem causar graves problemas. Uma operação em cascata de leitura automática implementada pode causar a recuperação de milhares de objetos relacionados a partir da recuperação de um simples objeto.

A agregação e composição geralmente são direcionadas a ter operações em cascata. Para associações, o que determina se haverá operação em cascata é a sua multiplicidade. A Tabela 1 resume as estratégias que podem ser usadas para definir operações em cascata em relacionamentos de objetos (AMBLER,2003b).

Tipo de Relacionamento	Exclusão em Cascata	Leitura em Cascata	Gravação em Cascata
Agregação	Considere excluir as partes automaticamente quando o todo for excluído.	Considere recuperar as partes quando o todo for recuperado	Considere gravar as partes automaticamente quando o todo for gravado
Associação (um para um)	Considere excluir a correspondente entidade na multiplicidade de 0 para 1. Exclua a correspondente entidade na multiplicidade de 1 para 1	Considere recuperar a correspondente entidade.	Considere salvar a correspondente entidade.
Associação (um para muitos)	Considere excluir as várias entidades	Considere ler as várias entidades	Considere gravar as várias entidades
Associação (muitos para um)	Não exclua esta entidade pois existem outras entidades referenciando esta.	Considere ler esta entidade	Considere gravar esta entidade
Associação (muitos para muitos)	Não exclua entidades que sejam referenciadas por outras referenciadas	Não realize a leitura pois se tornará um processo fora de controle	Não realize a gravação pois se tornará um processo fora de controle
Composição	Considere excluir as partes automaticamente quando o todo for excluído	Leia as partes automaticamente quando o todo for recuperado	Grave as partes automaticamente quando o todo for salvo.

Tabela 1 – Estratégia de Operações em Cascata(AMBLER,2003b)

Além do cuidado com operações em cascata, devemos ter o cuidado com referências cruzadas entre objetos. A integridade deste tipo de referência é denominada de princípio das "propriedades correspondentes" que afirma que os valores das propriedades usadas para implementar o relacionamento devem ser mantidos apropriadamente.

2.4.3.3 Estratégia de Implementação da Integridade Referencial

Existem duas correntes filosóficas que definem onde deve ser implementada a integridade referencial (AMBLER,2003b). Os tradicionalistas defendem que as regras de integridade referencial devem ser implementadas no banco de dados, pois

este ambiente fornece modernos mecanismos para esta funcionalidade e efetivamente é o local ideal para se centralizar o esforço deste controle que servirá para todas as aplicações. Os puristas da orientação a objetos defendem que a integridade referencial é uma questão de negócio e deve ser implementada dentro da camada de negócio e não no banco de dados. Esta implementação pode ser dentro dos objetos de negócio ou numa camada de persistência.

As duas opções apresentadas anteriormente apresentam vantagens e desvantagens. Na verdade existem diversas opções disponíveis para a implementação da integridade referencial que podem ser avaliadas e escolhidas de acordo com a estratégia de isolamento a ser adotada. Os tópicos a seguir comparam estas opções e seus contrastes (AMBLER,2003b).

- *Objetos de Negócio* - a integridade referencial é garantida através de operações implementadas nos objetos de negócio da aplicação. Atente a filosofia pura da orientação a objetos e os testes na aplicação são facilitados em função de se ter, em um só local, toda a lógica do negócio implementada. Obriga que toda aplicação deva ser implementada usando os mesmos objetos de negócio e o esforço extra de programação de funcionalidades que são suportadas naturalmente pelo banco de dados.

- *Constraints de Banco de Dados* - Esta abordagem usa a linguagem de definição de dados (DDL-Data Definition Language) para elaborar constraints no banco de dados para garantir as regras de integridade referencial. Garante que a integridade referencial no banco de dados e as constraints podem ser geradas e recuperadas por ferramentas de modelagem de dados. Exige que as aplicações devem ser projetadas a usar o mesmo banco de dados, ou todas as constraints devem ser implementadas em cada banco de dados. As constraints podem prejudicar o desempenho em tabelas com grande volume de dados.

- *Triggers de Banco de Dados* – Similares a constraints de banco de dados sendo disparados por um evento, como por exemplo a exclusão de uma linha para realizar ações que garantam a integridade referencial. Os comentários

feitos a respeito de constraints de banco de dados aplicam-se a este tipo de solução.

- *Framework de Persistência* - as regras de integridade são definidas como parte do mapeamento dos relacionamentos. A multiplicidade (cardinalidade e obrigatoriedade) dos relacionamentos é definida no metadados através de indicadores que informam a necessidade de leituras, atualizações e exclusões em cascata. A integridade referencial é implementada como parte da estratégia de persistência de objetos, sendo centralizada em um simples repositório de metadados. Toda aplicação deve ser projetada a usar o mesmo framework de persistência, ou usar o mesmo mapeamento de relacionamentos. Pode gerar dificuldades nos testes de integridade referencial, tendo em vista que está embutido em um metadados.

- *Visões Atualizáveis* - as regras de integridade são definidas na elaboração da visão de banco de dados. Visões atualizáveis são geralmente problemáticas em banco de dados relacionais devido a implicações na integridade referencial e quando referenciam várias tabelas podem não ser suportadas pelo gerenciador de banco de dados. Todas as aplicações devem usar visões, não mais as tabelas.

Apesar de todas as opções apresentadas, o banco de dados geralmente é a melhor escolha para a implementação da integridade referencial. O crescimento da importância dos serviços WEB e XML aponta para uma tendência onde a lógica das aplicações se torna menos orientada a objeto e mais orientada a processamento de dados, apesar da tecnologia orientada a objetos ser a base do desenvolvimento destes serviços (AMBLER,2003b).

2.4.3.4 Recuperação e Atualização de Objetos com Relacionamentos

Na perspectiva da persistência de objetos de forma isolada, os processos de recuperação e manutenção estão concentrados somente nos dados do objeto em questão. Porém, quando o objeto persistente tem relacionamentos com outros

objetos, o escopo deste processo não se restringe mais em recuperar e manter dados de um objeto, mas também dos objetos correlacionados. Ambler(2000a) sugere uma seqüência lógica de procedimentos em uma transação de recuperação e atualização em um ambiente objeto-relacional para manter a integridade de objetos envolvidos nesta operação. Estes procedimentos variam de acordo com a cardinalidade dos relacionamentos.

Mapeamento Um para Um

A lógica da recuperação de um objeto da *classe origem* segue os seguintes passos:

- O objeto da *classe origem* é lido para a memória
- A referência do relacionamento é automaticamente carregada
- O valor obtido pela correspondente *coluna que implementa o relacionamento no ambiente de banco de dados* é usado para identificar e carregar os dados da *classe destino* em memória
- A *classe destino* é instanciada
- O *atributo da classe origem que implementa o relacionamento* é preenchido com uma referência ao objeto da *classe destino* instanciada.

A lógica do processo de atualização de um objeto da *classe origem* segue os seguintes passos:

- Criar uma transação para manter a integridade referencial.
- Criar comandos de atualização da base de dados para cada objeto afetado pela transação. Cada comando de atualização inclui atributos dos objetos e as colunas que implementam o relacionamento no banco de dados. Tendo em vista que os relacionamentos são implementados através de chaves estrangeiras e que estes valores serão atualizados, o relacionamento efetivamente será persistido.

- Efetive a transação.

Mapeamento Muitos para Um

Os procedimentos de leitura são similares ao discutido no mapeamento um para um, com a diferença de que quando recuperar os objetos da *classe destino* deve-se prever a recuperação de uma coleção de objetos. Caso a *classe origem* esteja no sentido inverso da cardinalidade, isto é, muitos para 1, deve-se ter o cuidado de não recuperar uma coleção de cópias do mesmo objeto. Deve ser adotada uma estratégia que garanta que somente uma cópia do objeto permaneça na memória.

Mapeamento Muitos para Muitos

As etapas de leitura de um objeto que esteja em memória e deseja obter a coleção de objetos que se relacionam são:

- Crie um comando SQL que produza um produto cartesiano entre a tabela associativa e a tabela que implementa a coleção de objetos, filtrando todas as ocorrências da tabela associativa que relacionem com o identificador do registro que representa no banco de dados o objeto em memória.

- Submeta o comando SQL no banco de dados

- Os dados coletados que representam a coleção de objetos desejada são convertidos em uma coleção de objetos. Esta tarefa deve prever a verificação da existência do objeto em memória. Caso o objeto já esteja em memória, suas informações devem ser atualizadas.

- A coleção de objetos criada ou atualizada é incluída no atributo do objeto que estava em memória.

O processo de gravação neste tipo de relacionamento passa pelos seguintes passos:

- Iniciar uma transação

- Criar comandos de atualização para qualquer objeto da coleção que tenha sido atualizado
- Criar comandos de inclusão para qualquer objeto da coleção que tenha sido incluído
- Criar comandos de inclusão na tabela associativa
- Criar comandos de exclusão para qualquer objeto da coleção que tenha sido excluído. Este processo pode não ser necessário se os objetos já tiveram sido excluídos individualmente.
- Criar comandos de exclusão na tabela associativa para qualquer objeto da coleção que não exista mais. Este processo pode não ser necessário se os objetos já tiveram sido excluídos individualmente.
- Criar comandos de exclusão na tabela associativa nos casos em que não exista mais o relacionamento entre o objeto em memória e algum da coleção de objetos.
- Efetive a transação.

2.4.4 Desempenho

Neste capítulo foram discutidas diversas alternativas no mapeamento de relacionamentos e estruturas hierárquicas de objetos, cada qual com suas vantagens e desvantagens. Esta variedade de técnicas possibilita a busca do aprimoramento do desempenho do acesso aos dados, testando e analisando cada uma das escolhas apresentadas. A seguir apresentaremos outras técnicas complementares de aprimoramento do desempenho na persistência de objetos (AMBLER,2003b).

Leituras sob Demanda

Uma importante questão a considerar no desempenho é se o atributo pode ser automaticamente lido quando o objeto é recuperado. Por exemplo, um atributo que seja um objeto complexo pode ter um tamanho extremamente grande enquanto que os demais atributos não alcançam 10% de seu tamanho. Nesta situação, a leitura sob demanda pode prejudicar o desempenho consideravelmente.

Leitura sob demanda é uma técnica comum em aplicações orientadas a objetos onde os valores de atributos são disponibilizados à medida que são solicitados. Esta técnica apresenta um ganho de desempenho pois evita a recuperação desnecessária de objetos que nunca serão utilizados. A adoção desta estratégia é boa quando consideramos que a necessidade de objetos relacionados não é freqüente, mas se torna uma má escolha se considerarmos o aumento no volume de acessos na recuperação de objetos quando a demanda é freqüente.

Cache

Um cache é um local onde cópias de entidades são temporariamente guardadas para reduzir o número de interações com o banco de dados, obtendo-se assim uma melhoria de desempenho. Os acessos a banco de dados consomem a maior parte do tempo de processamento em aplicações de negócio, e a técnica cache pode reduzir consideravelmente a necessidade destes acessos em uma aplicação.

As três principais implementações desta técnica serão descritas a seguir:

- *Cache de Objetos* - nesta abordagem cópias de objetos de negócio são mantidas em memória. Servidores de aplicação podem disponibilizar objetos de negócio em uma área de cache compartilhada, possibilitando que todos os seus usuários trabalhem com a mesma cópia de objetos. Esta técnica reduz o número de interações com o banco de dados pois nesta se pode obter um objeto em uma operação e consolidar as mudanças feitas por vários usuários antes de atualizá-lo no banco de dados.

- *Cache de Banco de Dados* - um servidor de banco de dados mantém dados em memória cache reduzindo o número de acessos a disco.

- *Cache de Dados no Cliente* - as máquinas dos usuários têm pequenas cópias do banco de dados, reduzindo o tráfego de rede e possibilitando trabalhar em modo desconectado. As cópias do banco de dados são sincronizadas a partir de um banco de dados corporativo .

A técnica cache é benéfica quando usada para informações usadas somente para leitura e que não necessitam de atualizações freqüentes. Contudo, esta técnica adiciona considerável complexidade à aplicação em função da lógica necessária para o gerenciamento das informações em memória. O gerenciamento inclui o controle da replicação dos dados para se evitar colisões entre os dados originais e suas cópias. Adicionalmente, o fato de se ter cópias de objetos distribuídos aumenta o risco de se ter problemas de inconsistência e sincronismo entre estas réplicas.

2.4.5 Transações

Transação é uma coleção de operações sendo realizadas sobre vários objetos. Uma transação pode ser composta por uma combinação de operações de atualização, recuperação e exclusão de objetos. O controle de transação deve seguir o seguinte conceito: todas as suas ações devem ser realizadas com sucesso ou canceladas, caso ocorra algum erro durante a efetivação de alguma de suas operações. A duração de uma transação pode ser curta, durando alguns segundos, ou longa, durando dias, semanas ou até meses para ser completada.

Transações possuem quatro propriedades essenciais (TANNEMBAUM,1995):

a) Atômicas - uma transação é indivisível;

b) Consistente – uma transação não viola invariantes do sistema;

c) Isolada – transações concorrentes não interferem umas nas outras;

d) Durável - uma vez realizada, as mudanças decorrentes permanecem.

Primitivas do tipo *begin_transaction, commit_transaction ou rollback_transaction* são necessárias para delimitar o escopo de uma transação. A

primitiva *begin_transaction* delimita o inicio de uma série de operações sob uma mesma transação e as duas últimas finalizam uma transação (com ou sem sucesso respectivamente).

Como as transações são atômicas, isto implica que o conjunto de operações sob uma mesma transação ou acontece totalmente ou não, como se fosse uma única ação instantânea.

A consistência garante que uma transação não pode ferir regras estabelecidas para o sistema, caso isto aconteça durante a execução de uma transação, a transação não ocorre.

Toda transação é isolada ou serializável, isto significa que se duas ou mais transações estão ocorrendo concorrentemente, o resultado final é como se tivessem sido executadas em uma ordem seqüencial qualquer.

2.5 SOLUÇÕES EXISTENTES

No mercado de desenvolvimento de aplicativos, a persistência de objetos é resolvida através de soluções cujo nível de complexidade e eficiência varia com o escopo de sua atuação. Discutiremos algumas das muitas soluções existentes, suas características, sua atuação e implementação em uma aplicação orientada a objetos.

2.5.1 J2EE

JAVA 2 Plataform, Enterprise Edition (J2EE) é um padrão aberto para elaboração e implementação de aplicações corporativas portáteis e altamente escaláveis (PANDA,2003). O J2EE define uma plataforma para desenvolvimento de aplicações em múltiplas camadas. Muitos fornecedores, incluindo BEA, IBM, Oracle, e Sun, suportam J2EE oferecendo servidores de aplicação nos padrões estabelecidos pelo J2EE. Em termos de arquitetura em camadas, um servidor de aplicações atua como uma camada intermediária entre os usuários e o sistema

gerenciador de banco de dados, fornecendo um framework para a elaboração de aplicações corporativas.

J2EE é formado por vários componentes, entre eles temos: JAVA Servlets, JAVAServer Pages, Enterprise JAVABeans (EJB), JAVA Message Services, JAVA Transaction API, JAVA Database Connectivity (JDBC), J2EE Connector Architecture, JAVA Naming e Directory Interface (JNDI), e componentes JAVAMail.

Faremos algumas explanações a respeito dos componentes que mais se relacionam ao tema deste trabalho.

2.5.1.1 JDBC: O ponto de partida

Linguagens Orientadas a Objeto, como o JAVA, podem utilizar uma interface de programação (API) denominada JDBC (JAVA Database Connectivity) para interagir com o banco de dados (PANDA,2003). O JDBC provê uma série de serviços aos desenvolvedores de aplicações OO para armazenar e manipular informações no banco de dados relacionais através de códigos SQL. Este método de persistência direta torna a aplicação extremamente frágil a alterações de estruturas na base de dados. O DBA (Database Administrator) freqüentemente altera os esquemas de banco de dados por motivos de otimização de desempenho ou organização do ambiente. Estas alterações provocam manutenções nas codificações SQL existentes nos códigos JAVA. A melhor forma de implementação da persistência é separando-a em uma camada, tornando eventuais alterações dos esquemas de banco de dados transparente à camada de aplicação.

2.5.1.2 EJB

A tecnologia Enterprise JavaBean tem sido descrita como uma arquitetura de componentes distribuídos em servidor baseado em Java que fornece elementos básicos para sistemas de software corporativos que são melhores nos aspectos de escala, segurança, portabilidade e manutenção (WHITE,2001). A tecnologia EJB é

mais que objetos distribuídos ou componentes de arquitetura, é um framework de aplicação que disponibiliza vários serviços fundamentais a sistemas, tais como transações e persistência. Esta característica liberta desenvolvedores do trabalho de desenvolver elementos básicos, concentrando seus esforços no domínio do problema a ser automatizado. O paradigma EJB também padroniza vários aspectos do ciclo de vida de aplicações corporativas, como o desenvolvimento, depuração e implementação (WHITE,2001).

Container EJB

Um EJB é um componente reutilizável da programação JAVA que roda dentro de um container EJB e pode ser chamado localmente ou remotamente por um programa JAVA ou cliente CORBA (PANDA,2003). Em função de um EJB executar em um container que oferece suporte a transações, persistência e controle de acesso de beans, o desenvolvedor não tem a necessidade de implementar estes tipos de serviços. Existem três tipos de EJB: session beans, entity beans e message-driven beans (MDB's).

Session beans encapsulam regras de negócio que são transientes por natureza e não permanecem após uma falha de máquina ou servidor. Session beans podem interagir com um banco de dados usando JDBC.

Entity beans são persistentes e necessitam de um repositório de dados como um banco de dados. Como alternativas ao uso de entity beans, algumas organizações podem decidir usar uma camada de persistência pronta de um fornecedor (Oracle 9i TopLink) ou desenvolver um código "caseiro" para a persistência no banco de dados - soluções já comentadas neste documento. Um entity bean pode representar, por exemplo, uma linha em uma tabela de consumidores ou um registro de uma tabela de empregados. Entity beans podem também ser compartilhados por vários clientes, como por exemplo, o *entity bean* "empregado" pode ser usado por vários usuários para calcular salário anual ou atualizar endereço. Uma falha no servidor EJB pode resultar em rollback de uma transação, porém sem destruir o entity bean, pois este representa uma informação persistente em um banco de dados. Similar a um registro de uma tabela de banco de dados, entity bean tem um identificador único.

A especificação EJB define dois tipos de entity beans: bean-managed persistence (BMP) beans e container-managed persistence (CMP) beans. No BMP beans, desenvolvedores de bean escrevem o código SQL em seus beans para criar, atualizar e excluir informações do banco de dados. Nesta solução o desenvolvedor tem que ter o conhecimento dos dois paradigmas de programação, SQL e OO, representando um custo de aprendizado e tempo para um projeto de software, como já comentado anteriormente.

No CMP beans o container gera todos os comandos SQL, sendo necessário o mapeamento entre a entity beans e o correspondente schema no banco de dados. Porém nem todos os servidores de aplicação ou containers têm a flexibilidade de mapear um entity bean a uma tabela específica. A maioria dos servidores de aplicação utiliza algoritmos próprios para nomear as tabelas a serem utilizadas na persistência dos dados. Os desenvolvedores devem definir nos beans chaves primárias e mapeá-las para suas colunas correspondentes na base de dados. Caso não se tenha realizado o mapeamento da chave primária, o servidor de aplicação cria e nomeia automaticamente colunas no banco de dados para tal função. Um container J2EE gera valores únicos para as chaves primárias por algoritmos próprios.

Um container-managed relationship (CMR) é onde os desenvolvedores criam relacionamentos entre entidades e o servidor de aplicação J2EE implementa estes relacionamentos no ambiente persistente. Os relacionamentos podem ser de um para um, um para vários ou vários para vários. Alguns containers usam *constraints* de chaves estrangeiras já existentes ou tabelas denominadas associativas para o controle dos relacionamentos. Normalmente, o container se encarrega de gerar e nomear tabelas para a manutenção dos relacionamentos muitos para muitos. Para efetuar consultas em CMP beans os desenvolvedores utilizam a linguagem padrão EJB QL. Esta linguagem é similar a ANSI SQL e quando utilizada o container a converte em comandos SQL em tempo de execução.

White(2001) em seu trabalho enfatiza alguns pontos de projeto a serem considerados ao se utilizar o JAVABeans. O primeiro é a escolha da especificação EJB e o servidor de aplicação EJB a ser utilizado. O seu artigo alerta para as

grandes diferenças nas especificações das versões 1.0 e 2.0, lembrando que a tecnologia EJB está em contínua evolução. Fornecedores se esforçam em manter as especificações e consideram uma vantagem competitiva adicionar funcionalidades não padronizadas para implementar freqüentes requisitos que ainda não são tratados (ou algumas vezes livremente tratados) pela especificação. Outro ponto comentado é a dependência existente entre a estrutura do banco de dados e o projeto de beans e vice-versa. Sem a ajuda de ferramentas sofisticadas e funcionalidades do servidor EJB, entity beans tende a ser mapeada para uma simples tabela. Podemos então concluir que quanto mais normalizado for o modelo de banco de dados, mais granularizado o modelo beans se tornará. Se isto não for seguido, desenvolvedores EJB terão mais trabalho para implementar a persistência através da elaboração de códigos denominados *bean managed persistence*. Este fato nos leva a outra questão levantada no artigo (WHITE,2001) na qual se afirma que em geral peritos em EJB preferem beans menos granularizados, isto é, aqueles que abrangem de forma mais macro as questões de negócio. Esta preferência se dá pelo fato de que beans mais abrangentes demandam com menor freqüência comunicações remotas entre os EJB's e conexões com o banco de dados, melhorando o desempenho do aplicativo.

O artigo (SILVA,2003) retrata o aumento das dificuldades enfrentadas nos projetos J2EE com a existência de um distanciamento entre o servidor de aplicações e os sistemas corporativos a serem desenvolvidos. A maioria das organizações define que a viabilização da arquitetura J2EE só é possível com o desenvolvimento ou aquisição de frameworks. Neste contexto o autor prevê nos próximos anos a disponibilização de várias opções de framework no mercado, fator fundamental para o crescimento e proliferação da arquitetura J2EE no ambiente corporativo.

2.5.2 JDO

A arquitetura JAVA Data Object - JDO define um conjunto de padrões de contrato entre um programador de aplicações e o fornecedor JDO. Estes contratos

enfocam na visão de instâncias JAVA de classes passíveis de persistência (KRUSZELNICKI,2002).

JDO usa o *Connector Architecture* para especificar o contrato entre o fornecedor JDO e um servidor de aplicação. Estes contratos concentram-se em aspectos importantes de integração com sistemas de informações corporativas heterogêneas: *instance management*, *connection management* e *transaction management*.

2.5.2.1 JDO Instance

Um *JDO instance* é uma instância da classe Java que implementa as funcionalidades da aplicação e representa dados de um repositório de dados corporativo. Sem limitações, o dado pode vir de uma simples entidade de armazenamento de dados ou de uma coleção de entidades. Por exemplo, uma entidade pode ser um simples objeto de um banco de dados orientado a objetos, uma simples linha de um banco de dados relacional, o resultado da consulta em um banco de dados relacional que incorpora várias linhas, uma mesclagem de dados de diversas tabelas em um banco de dados relacional, ou o resultado da execução de uma API que retorna informação oriunda de um sistema ERP.

Um JDO instance tem uma limitação importante: esta classe sempre implementa a interface *PersistenceCapable*. Esta interface prevê uma série de especificações que tem por objetivo estabelecer um padrão único de interface nas implementações de persistência desenvolvidas e comercializadas por diversos fornecedores. Uma vantagem desta abordagem é a transparência proporcionada em relação ao tipo de armazenamento de dados que está sendo utilizado (KRUSZELNICKI,2002).

2.5.2.2 JDO Implementation

Um *JDO implementation* é uma coleção de classes as quais implementam os contratos JDO (KRUSZELNICKI,2002). O *JDO implementation* pode ser fornecido por um fornecedor EIS ou por um terceiro fornecedor, conhecido como um fornecedor JDO. Estes terceiros podem fornecer uma implementação que é otimizada para um domínio específico de aplicação, ou pode ser uma ferramenta de propósito geral. (por exemplo, uma ferramenta de mapeamento relacional, um banco de dados de objetos embutido, ou um banco de dados de objetos corporativos).

2.5.2.3 JDO Enhancer

Um *JDO enhancer*, ou otimizador de byte code, é um programa que modifica o *bytecode* de um arquivo de classes JAVA que implementa um componente da aplicação, permitindo a leitura e gravação transparente dos campos das instâncias persistentes.

2.5.2.4 JDO: Modelo de Persistência de Objetos

O ambiente de desenvolvimento JAVA suporta diferentes tipos de classes que são de interesse do desenvolvedor. As classes que modelam a aplicação e o domínio do negócio são o foco primário do JDO. Em uma aplicação típica, suas classes são altamente interconectadas, e o cenário de instâncias destas classes inclui todo o contexto de armazenamento de dados.

Aplicações usualmente tratam um pequeno número de instâncias persistentes ao mesmo tempo, e esta é a função do JDO em permitir a ilusão de que a aplicação pode acessar todo o cenário de instâncias conectadas, enquanto que na realidade somente uns pequenos subgrupos de instâncias precisam ser instanciados no JVM (JAVA Virtual Machine).

O armazenamento dos objetos em repositórios de dados pode ser bem diferente do armazenamento de objetos no JVM. Portanto a arquitetura prevê um mapeamento entre as instâncias JAVA e os objetos contidos no repositório de dados. Este mapeamento é executado pelo *JDO implementation*, usando um metadado que permanece disponibilizado em tempo de execução. O metadado é gerado por uma ferramenta de um fornecedor JDO, conjuntamente com o especialista do sistema. O mapeamento não é padronizado pelo JDO.

2.5.3 Hibernate

Hibernate (HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1,2003) é uma ferramenta de mapeamento objeto relacional para o ambiente JAVA. O termo mapeamento objeto relacional (ORM) refere-se à técnica de mapeamento de uma representação de um modelo de objetos para uma estrutura relacional. O objetivo do Hibernate é aliviar em 95% o esforço de desenvolvimento de tarefas de programação relacionadas à persistência de dados.

O Hibernate possibilita o desenvolvimento de objetos persistentes contemplando características da linguagem JAVA – incluindo associação, herança, polimorfismo entre outros, e rejeita o uso de geração de código ou processos de *bytecode* em tempo de construção. Ao invés disso, a ferramenta utiliza a reflexão e codificação de *bytecode* em tempo de execução e a geração de SQL ocorre na inicialização do sistema. Esta decisão assegura que o Hibernate não impactará os processos de debug e compilação incremental.

O mapeamento objeto/relacional é definido em um documento XML, conforme Figura 3. O documento de mapeamento é projetado para ser legível e editável. A linguagem de mapeamento é orientada a JAVA, isto é, o mapeamento é construído nos moldes da declaração da classe persistente, desconsiderando a estrutura da tabela. A documentação apresentada em seus tutoriais não apresenta a flexibilidade de se ter uma classe implementada em mais de uma tabela, o que limita o seu uso em uma base de dados relacional legada.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping package="eg">
        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat" discriminator-value="D">
                <property name="name" type="string"/>
                </subclass>
        </class>
        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>
</hibernate-mapping>
```

Figura 3 – Exemplo de Mapeamento O/R em XML do Hibernate

A utilização do hibernate na persistência de objetos implica na utilização de classes próprias responsáveis pela efetivação da persistência dos objetos nos códigos JAVA. A incorporação destas classes caracteriza um alto nível de acoplamento da aplicação em relação ao framework resultando em prejuízos já mencionados neste trabalho.

O Hibernate delega o controle de transações para a interface JDBC ou qualquer outro mecanismo que realize esta tarefa. Seu papel se restringe em ser um simples adaptador à interface JDBC, adicionando a semânticas da orientação a objetos.

2.5.4 CocoBase Enterprise O/R

CocoBase Enterprise O/R é um framework de persistência de objetos para o ambiente JAVA cuja proposta é diminuir o custo despendido no desenvolvimento de sistemas no acesso a banco de dados relacionais em até 85%.

Seu funcionamento é baseado no uso de um mapeamento objeto relacional, implementado em um banco de dados e administrado através de uma ferramenta que possui uma interface amigável. Atributos de objetos podem ser mapeados para uma ou mais tabelas simultaneamente. Sua arquitetura torna a aplicação orientada a objetos independente das peculiaridades relacionadas a estrutura e sintaxe SQL de um fornecedor específico de banco de dados.

O CocoBase proporciona uma persistência de objetos transparente através de dois recursos (COCOBASE O/R MAPPING – PROGRAMMERS GUIDE v.1, 2001): (i) um processo automatizado denominado *proxying data objects*, que dispensa o objeto persistente da obrigatoriedade de implementar interfaces de persistência; (ii) a técnica de reflexão da linguagem JAVA que obtém informações de atributos de objetos persistentes. Sua utilização não afeta o código fonte e o *bytecode* de classes persistentes JAVA.

Similar ao Hibernate, o CocoBase necessita da instanciação de classes próprias, responsáveis pela efetivação da persistência dos objetos.

Seu mapeamento objeto relacional contém somente os metadados de tabelas, não fazendo qualquer referência às classes persistentes. Não existe a correlação, por exemplo, de colunas de tabelas a atributos de classes persistentes em seu modelo de mapeamento objeto relacional. Simplesmente é definido um *CocoBase Map* (COCOBASE O/R MAPPING – BASIC CONCEPTS AND QUICK START GUIDE, 2001) contendo as seguintes informações: (i) a tabela a qual se refere, podendo ser até mais de uma; (ii) as colunas envolvidas e suas características; (iii) a cláusula *where* para as operações de consulta, atualização e exclusão. Uma vez definido um *CocoBase Map*, este pode ser utilizado para realizar a persistência de qualquer classe cujos atributos sejam equivalentes às colunas definidas no *CocoBase Map*. Conseqüentemente, a assinatura dos serviços de persistência disponibilizados pelo CocoBase (*select, insert, update, delete, etc*) prevê como parâmetros o objeto e o *CocoBase Map* a ser utilizado.

Esta ferramenta oferece uma camada de controle de transação orientado a objetos, independente da conexão com o banco de dados. Os objetos que se utilizarem desta camada têm seus dados automaticamente sincronizados com o

banco de dados, bem como, suas alterações monitoradas. O monitoramento proporciona um ganho de desempenho tendo em vista que somente os objetos alterados serão atualizados na base de dados. Este recurso é opcional, podendo a aplicação somente se utilizar do controle de transações da camada de banco de dados.

3. PROPOSTA DE UM FRAMEWORK PARA IMPLEMENTAÇÃO DE PERSISTÊNCIA

Este capítulo apresenta um Framework para a persistência de objetos utilizando bancos de dados relacionais – FPOR (**F**ramework para **P**ersistência **O**bjeto **R**elacional), que visa alcançar os objetivos estabelecidos na seção 1.3, em especial, reforça:

- Sob o ponto de vista do código fonte da aplicação, a transparência na utilização do framework. Isto é, o código da aplicação deve sofrer o mínimo de influência em conseqüência da utilização de um framework específico. E o programador de aplicação não deve se preocupar com detalhes de implementação do framework para sua utilização;

- Facilitar o mapeamento entre o modelo orientado a objetos e o modelo relacional, garantindo as características do modelo OO e aproveitando ao máximo as características do modelo relacional já bem resolvidas nas implementações dos SGBD's. Por exemplo, mapear diretamente associações entre classes e ligações (*join)* entre entidades do modelo físico do banco de dados;

- Ser adaptável a diversos gerenciadores de bancos de dados relacionais.

Primeiramente foi elaborado um modelo para o framework proposto, utilizando UML como linguagem de representação, que visa contemplar:

- como serão manipulados os objetos persistentes em uma aplicação;
- como será especificado o mapeamento entre classes/associações e entidades/relacionamentos;
- como será efetuado o acesso ao meio de persistência.

O framework proposto foi implementado em JAVA. As seguintes diretivas nortearam as escolhas de implementação:

- os SGBD's relacionais de mercado são ferramentas estáveis de alto desempenho e já tratam questões como transações atômicas, concorrência e *cache*. Portanto, o FPOR faz uso ao máximo destas características e não as implementa internamente;
- da mesma forma, as restrições de integridade referencial relativas aos relacionamentos são garantidas pelos SGBD's;
- o modelo de concorrência adotado é o bloqueio otimista;
- os serviços de persistência devem ser disponibilizados através das próprias classes persistentes;
- a individualidade e as características do modelo de classes e do modelo relacional devem ser preservadas, sem que uma embase a outra, possibilitando o uso de sistemas legados.

3.1 MODELO

O Modelo do FPOR foi organizado em 3 pacotes:

- Persistência, que focaliza o modelo de objetos persistentes da aplicação;
- Mapeamento, que especifica a relação entre os elementos dos modelos orientado a objetos e entidade relacionamento;

- Mecanismo, que trata a possibilidade de utilização de diversos mecanismos de persistência.

Na Figura 4 é apresentado também um pacote, denominado aplicação, que mostra a interação de classes de uma aplicação com o FPOR.

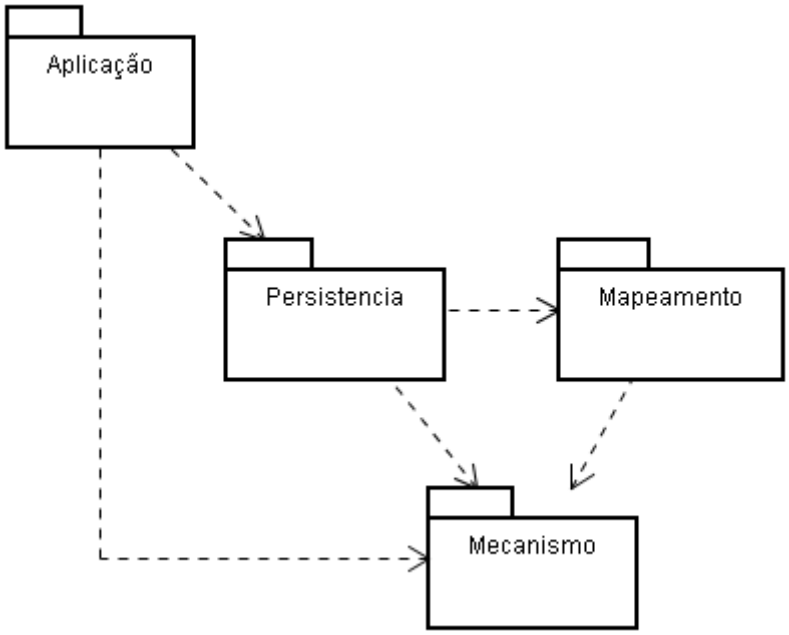


Figura 4 – Diagramas de Pacote do FPOR

O Pacote Persistência é a parte central desta proposta, pois especifica a interação do FPOR com a aplicação. Os outros dois pacotes modelam a dinâmica interna de funcionamento do framework de forma a cumprir os pré-requisitos estabelecidos.

3.1.1 Pacote Persistência

A Figura 5 apresenta o diagrama de classes que descreve o pacote persistência.

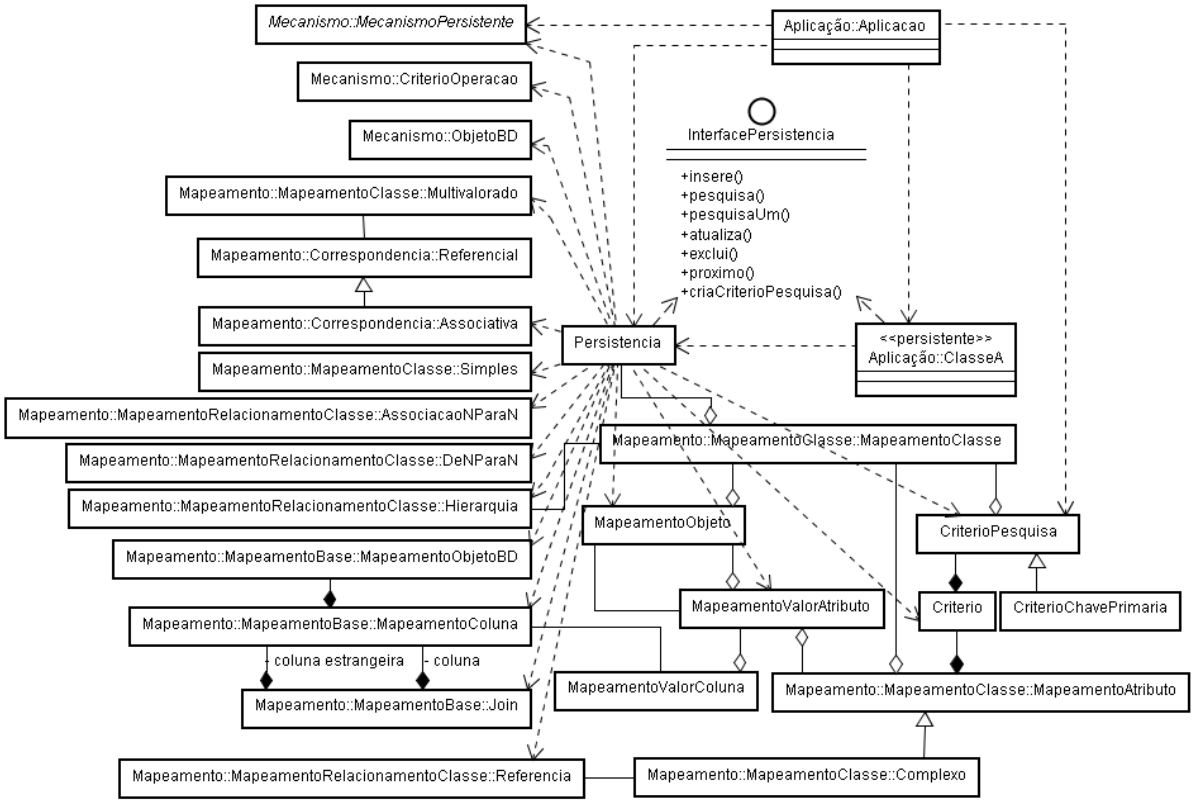


Figura 5 – Pacote Persistencia[1]

A *classeA* representa uma classe persistente qualquer de uma aplicação. Esta deve implementar a *InterfacePersistencia* que estabelece o conjunto de operações específicas que devem ser disponibilizadas por objetos persistentes. A implementação de uma classe persistente utiliza o padrão de projeto Adaptador (LARMAN,2004). Desta forma, o FPOR disponibiliza a classe *Persistencia* que implementa estas operações.

Uma classe persistente será manipulada por outras classes da aplicação que serão responsáveis por:

- instanciar um objeto desta classe: a instanciação de um objeto persistente faz com que seja instanciado um objeto *MapeamentoClasse* e um da classe *Persistencia*. Informações sobre o mapeamento da classe,

[1] A notação <nome do pacote>::<nome da classe> adotada neste diagrama, em função do software de edição gráfica utilizado, é equivalente à < from < nome do pacote>>.

empregadas em *MapeamentoClasse* são fundamentais e caso não estejam disponíveis, ocorre uma exceção;

- durante a utilização do objeto persistente, serão invocados métodos relacionados a operações definidas na *InterfacePersistencia*. A solicitação de um serviço persistente demanda na classe *Persistencia* a instanciação de vários objetos do pacote *Mapeamento*, para a coleta de informações com a finalidade de elaborar os comandos SQL que devem ser submetidos ao Gerenciador de banco de dados por intermédio do pacote *Mecanismo*.

Por exemplo, na Figura 6 temos duas classes de uma aplicação: a *IntPenalidade* que interage com o usuário final disponibilizando assim a criação de novos objetos da classe *Penalidade* e a consulta de objetos existentes.

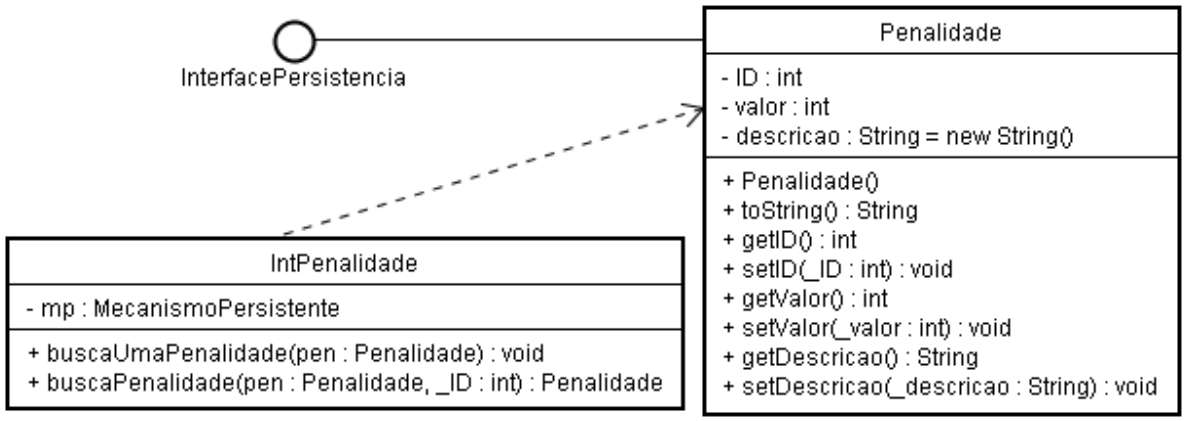


Figura 6 – Exemplo de Uso do FPOR

Ressaltamos que as operações da classe persistente *Penalidade* apresentadas no modelo da Figura 6 são apenas as relacionadas a aplicação. As operações de persistência estão na interface *InterfacePersistencia*.

```
...
    public  IntPenalidade() throws Exception
    {
       MecanismoPersistente mp =
            new MecanismoPersistente(     "usuario",
                                          "manager",
                                          "jdbc:oracle:oci8:@ora");
       Persistencia.defineMapeamento(
            new ParseOracle(  "mapeamento",
                              "manager",
                              "jdbc:oracle:oci8:@ora"));
       super();
    }
    public void novaPenalidade(Penalidade pen)
    {
       pen.insere(mp);
       mp.efetiva;
    }
    public Penalidade buscaPenalidade(Penalidade pen,int _ID)
    {
       CriterioPesquisa cp1 = pen.criaCriterioPesquisa();
       cp1.adicionaCriterio("ID","=",_ID);
       if (pen.pesquisaUm(mp,cp1))
          return pen;
       else
          return null;
    }
...
```

Figura 7 – Trecho Exemplo de Uso do FPOR

No techo de código que consta na figura 7, quando a classe *IntPenalidade* invoca o método *pesquisaUm* do objeto *Penalidade*, este por sua vez submete esta tarefa junto ao pacote *Persistência* que resolve a solicitação através da coleta de informações (utilizando classes no pacote *Mapeamento*) e concretiza as ações necessárias no banco de dados relacional através de classes do pacote *Mecanismo*. O diagrama de atividades da Figura 8 ilustra o processo de funcionamento do FPOR.

A notação utilizada na Figura 8, bem como em outras ao longo deste capítulo, é o diagrama de atividades da UML. Ele pode ser aplicado a qualquer propósito (como a visualização dos passos de um algoritmo de computador), mas é

especialmente útil para visualizar fluxos de trabalho e processos do negócio ou casos de uso (LARMAN ,2004).

Algumas notações de destaque incluem ramificações, raias de piscina (*swimlanes*) e barras de bifurcação e união. Ramificações especificam caminhos alternativos baseados em expressões *booleanas*. Raia de piscina é um elemento opcional e corresponde a uma área de responsabilidade, sendo freqüentemente uma unidade organizacional. Neste trabalho convencionamos que as raias de piscina correspondem às classes de aplicação, classes persistentes, pacotes do FPOR e pacotes Java utilizados na persistência de objetos. Barras de sincronização especificam a união e a bifurcação dos fluxos paralelos de controle.

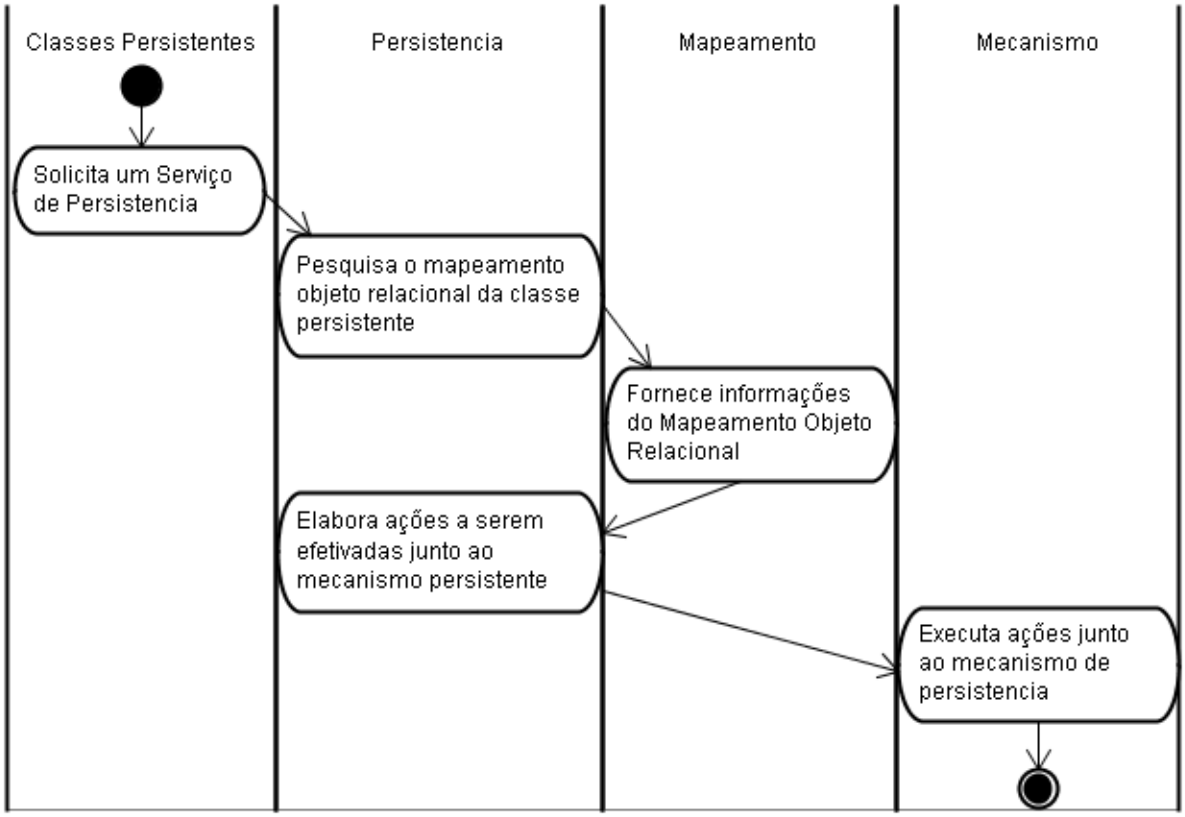


Figura 8 – Diagrama de Atividades Macro do FPOR

Esta organização minimiza o impacto sobre o framework em casos de evoluções e alterações tecnológicas, tendo em vista a possibilidade da substituição ou extensão de qualquer um dos pacotes sem abalar os demais.

No construtor da classe *IntPenalidade* podemos observar que é instanciado um objeto da classe *MecanismoPersistente* e armazenado na variável *mp*. Esta variável é utilizada como parâmetro nos serviços persistentes e na efetivação da inclusão do objeto *Persistencia*.

A classe *MecanismoPersistente* é uma classe do pacote *Mecanismo* que representa o banco de dados relacional com o qual a aplicação está trabalhando. O conceito de transação está nesta classe, pois o FPOR utiliza os controles de transação do SGBD, sem implementar qualquer controle interno em relação a este aspecto da persistência. A classe *MecanismoPersistente* disponibiliza os métodos *efetiva* e *desfaz* que correspondem à primitivas *commit_transaction* e *rollback_transaction* descritas no capítulo 2.

A utilização do *MecanismoPersistente* pode ser feita com ou sem o gerenciamento de conexões. A diferença está no construtor, pois se a aplicação desejar realizar o gerenciamento da conexão, esta deve instanciar o mecanismo persistente com um construtor específico. Os métodos *conecta* e *desconecta* da classe *MecanismoPersistente* possibilitam o controle da conexão junto ao mecanismo persistente.

A classe *MapeamentoObjeto* representa os diversos objetos persistentes instanciados durante a utilização do framework. Esta classe tem como função principal permitir o mapeamento dentro do FPOR dos valores dos atributos do objeto tanto na aplicação quanto na base de dados, através das classes *MapeamentoValorAtributo* e *MapeamentoValorColuna* respectivamente. Este mapeamento é o alicerce do controle de estados, definido no capítulo 1, o qual se baseia na relação entre sua representação em memória e no meio persistente. Estas classes, além disso, são utilizadas no controle de concorrência que indica a ocorrência de colisão (vide seção 2.4.2), quando se encontram divergências entre os valores recuperados do objeto persistente com o que consta no banco de dados, no momento da efetivação de uma alteração, gerando então uma exceção.

A classe *CriterioPesquisa* é utilizada na criação de critérios de pesquisa. Estes critérios são parâmetros obrigatórios na invocação de serviços de recuperação de objetos. Um *CriterioPesquisa* pode conter vários critérios, representados pela

classe *Criterio*. A utilização da classe *CriterioPesquisa* e *Criterio* por uma classe de aplicação está exemplificada na figura 9. A classe *CriterioChavePrimaria* é uma especialização da classe *CriterioPesquisa* destinada à localização de um objeto através da montagem de critérios com atributos que correspondem a chaves identificadoras no meio de persistência.

3.1.1.1 Dinâmica dos Objetos Persistentes

A manipulação e recuperação individual e coletiva de objetos foi resolvida através de métodos específicos de recuperação coletiva, associados a métodos de navegação sobre estas coleções. Os serviços de pesquisa das classes persistentes no FPOR são especializados em duas funcionalidades: a recuperação individual de objetos e a recuperação de mais de um objeto.

A recuperação individual de objetos prevê que a classe de aplicação informe um critério de pesquisa e que seu resultado retorne somente uma instância do objeto persistente. Caso sejam retornadas mais de uma instância, o FPOR provoca uma exceção.

A recuperação coletiva de objetos pode ser feita opcionalmente com um critério de pesquisa. Este tipo de recuperação retorna para a classe de aplicação um cursor, ao invés de uma coleção de objetos. A classe de aplicação pode montar uma coleção de objetos a partir da navegação deste cursor. Exemplificamos esta recuperação através do código que consta na Figura 9.

```
...
    public Penalidade buscaUmaPenalidade(Penalidade pen,int _ID)
    {
       CriterioPesquisa cp1 = pen.criaCriterioPesquisa();

       /* Cria um critério de pesquisa */
       cp1.adicionaCriterio("ID","=",_ID);

       /* Realiza a pesquisa retornando o objeto encontrado ou,
          se não obtiver sucesso, retorna null */
       if (pen.pesquisaUm(mp,cp1))
          return pen;
       else
          return null;
    }

    public Collection buscaPenalidadesMaiores(Penalidade pen,int _ID)
    {
       /* Cria uma coleção para ser preenchida */
       Collection retorno = new java.util.Collection.ArrayList();

       /* Cria um critério de pesquisa */
       CriterioPesquisa cp2 = pen.criaCriterioPesquisa();

       /* Adiciona um critério, no caso todos que tenham
          ID maior que o informado via parâmetro */
       cp2.adicionaCriterio("ID",">",_ID);

       /* Pesquisa e preenche a variável de retorno navegando pelo
          seu resultado. */
       pen.pesquisa(mp,cp2);
       while (pen.proximo(mp))
          retorno.add(pen.clone());

       /* Finaliza retornando a coleção de objetos */
       return retorno;
    }
...
```

Figura 9 – Código Exemplo de Pesquisa no FPOR

O FPOR controla os estados dos objetos persistentes com base na comparação dos valores das propriedades do objeto em relação a seus respectivos valores no banco de dados. O diagrama de estados da Figura 10 apresenta os possíveis estados e transições para um objeto persistente. Para garantir a consistência do objeto em memória com sua representação no meio de armazenamento, foram implementadas no FPOR restrições (ver estados “Persistente Alterado” e “Preenchido”) que não permitem, por exemplo, navegar em uma coleção após ter sido alterado o valor dos atributos de um objeto sem efetuar explicitamente operações de atualizar (persistir na base de dados) ou descartar.

Os estados “Inserindo” e “Excluindo” ocorrem respectivamente durante o processamento de inclusão e exclusão do objeto persistente.

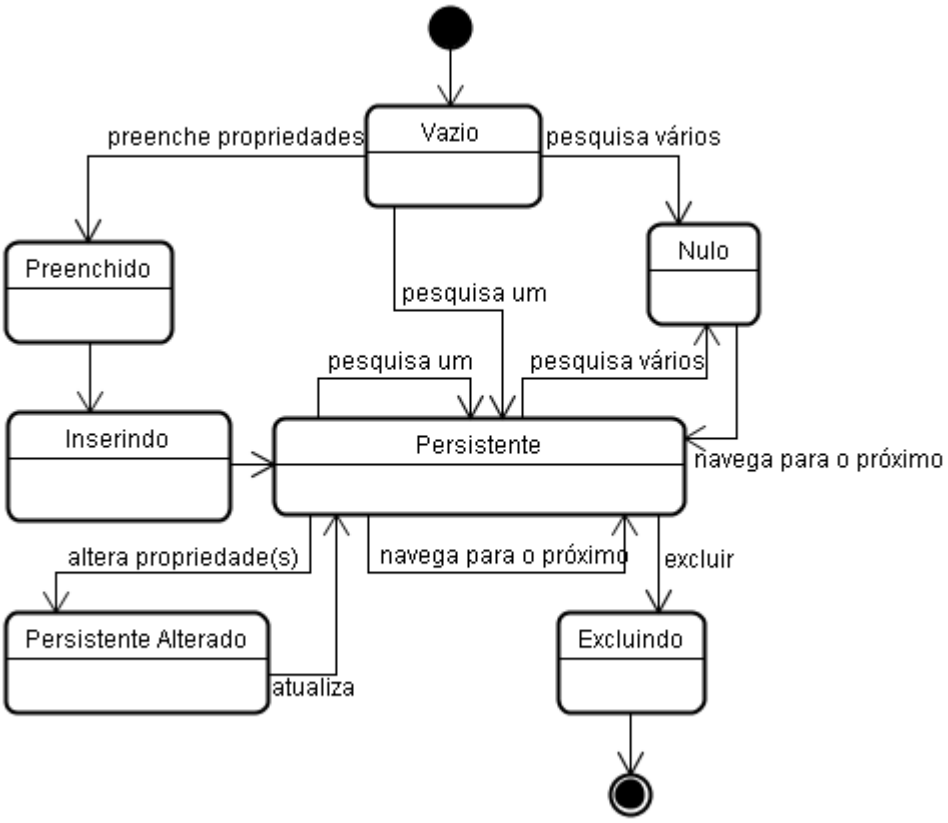


Figura 10 – Diagrama de Estados do Objeto

3.1.1.2 Programando uma Classe Persistente

O processo de transformação de classes persistentes proporciona ao programador total transparência na implementação da persistência com o framework FPOR. Este processo agrega métodos e atributos previstos na interface de persistência em arquivos de classes já compilados. O programador deverá marcar apenas as classes persistentes implementando a interface *InterfacePersistencia* – *implements IntPersistencia*.

A compilação do arquivo fonte com extensão.Java produz um arquivo com extensão.class. Este arquivo é composto por *bytecodes*, cuja manipulação é possível através de bibliotecas de classes específicas. A manipulação de arquivos com *bytecodes* permite incluir, excluir e alterar métodos e atributos diretamente no arquivo de extensão.class, sem ter a necessidade de modificação do código fonte contido no arquivo.Java. Esta técnica é empregada neste trabalho como a solução para facilitar a implementação de classes persistentes, evitando que o programador seja obrigado a implementar cada método da interface, delegando a chamada para o objeto *Persistência* instanciado na classe. Desta forma, o programador fica livre desta tarefa repetitiva e normalmente, geradora de falhas de implementação.

Apesar do framework Hibernate (HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1,2003) rejeitar o processo de manipulação de *bytecode* com a justificativa de evitar prejuízos nos processos de debug e compilação incremental (item 2.5.3), este trabalho não constatou este fato. É possível realizar o *debug* de uma classe persistente ignorando os métodos implementados da interface *InterfacePersistencia*, pois estes não necessitam passar por este processo. Portanto, a implementação de métodos e atributos no código *bytecode* das classes persistentes não acarreta qualquer impacto nos processos de debug ou compilação.

A transformação de classes é feita através de uma biblioteca de classes denominada Javassist (CHIBA,2003;CHIBA;2004;GERBER,2003) que proporciona um nível de linguagem mais alto para a manipulação direta do código *bytecode*. Esta biblioteca oferece uma interface de programação com vocabulário semelhante ao de

um código fonte Java, ao invés do complexo vocábulo de manipulação direta do código bytecode.

O componente do framework que realiza a tarefa de transformação de classes persistentes é a classe denominada *CompiladorJavassist* que recebe como parâmetro o nome de um pacote. O *CompiladorJavassist* inclui os métodos e atributos que devem constar na classe denominada *ModeloPersistencia* no arquivo bytecode de todas as classes que implementarem a I*nterfacePersistencia* do pacote informado. A Figura 11 ilustra melhor este processo.

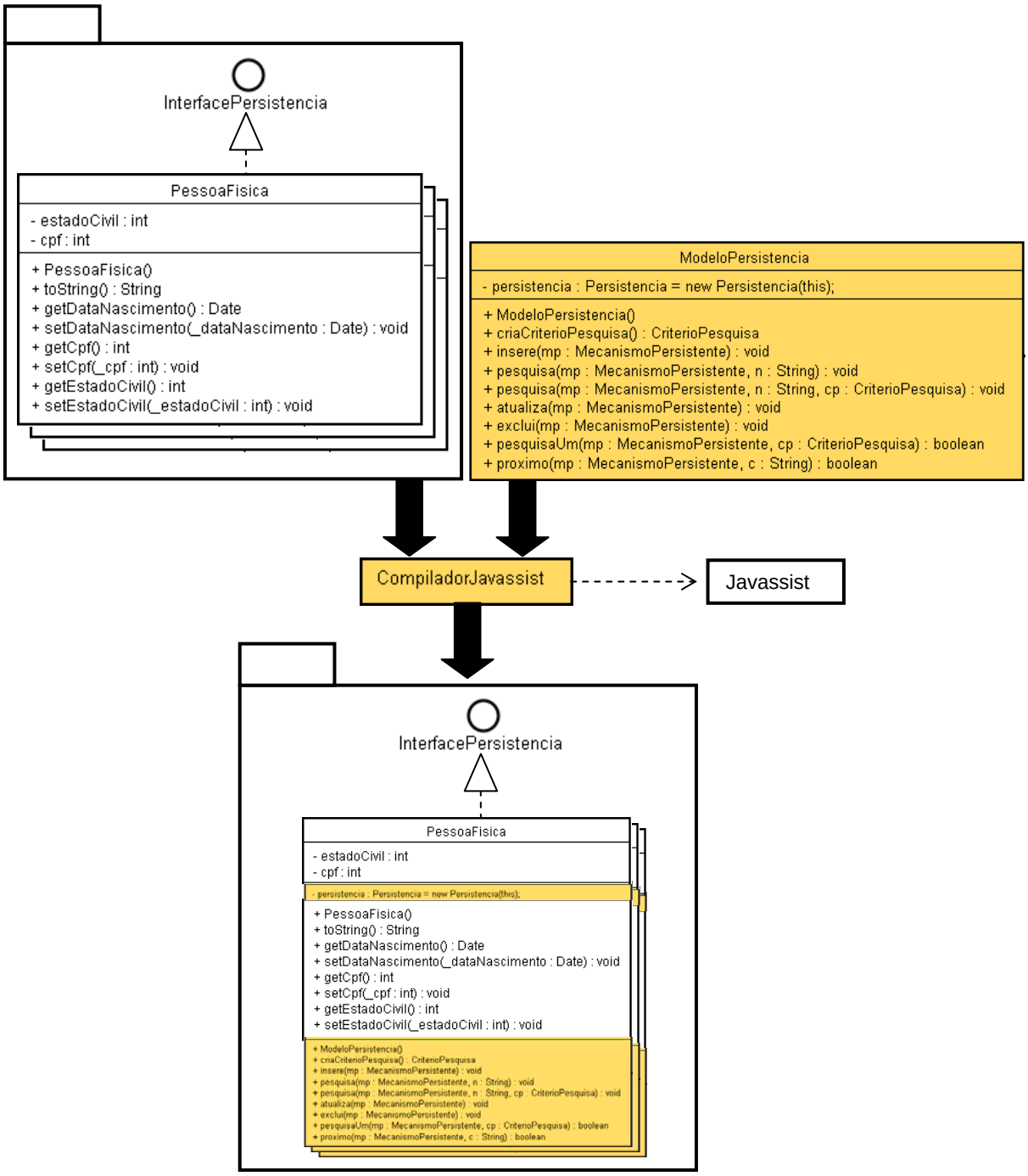


Figura 11 – Transformação de Classes Persistentes

Cabe salientar que todas as classes de Aplicação que invocarem em seu código métodos de persistência de classes persistentes só devem ser compiladas após o processo de transformação das classes persistentes.

Além de implementar métodos e atributos pertinentes à persistência, o CompiladorJavassist incorpora em cada método “get” de propriedade da classe persistente uma chamada a um método do framework destinado a controlar a recuperação de objetos sob demanda, descrito no item 3.2.2.

A recuperação de objetos é baseada no padrão Observador, também conhecida como Publicação-Assinatura ou Modelo de Delegação de Evento (LARMAN,2004) que prevê uma interface “assinante” ou “ouvinte”. Os assinantes implementam esta interface. O publicador pode registrar dinamicamente os assinantes que estejam interessados em um evento e notificá-lo quando um evento ocorrer.

No caso do FPOR, o “ouvinte” é a classe *Persistente* e o publicador são as classes persistentes que notificam quando um atributo complexo ou multivalorado é referenciado através do método “get”. Nesta implementação deste padrão não existem interfaces de ouvintes a serem implementadas e o publicador não registra os assinantes. Estas tarefas ficam a cargo do CompiladorJavassist, que se encarrega de transformar os objetos persistentes em publicadores de eventos para a classe “ouvinte” *Persistencia*.

O pacote *preCompiladorJavassist* é o responsável por realizar a transformação de uma classe em persistente. Conforme Figura 12, o pacote é composto por duas classes e uma interface. A classe *CompiladorJavassist* copia os atributos e métodos da classe *ModeloPersistencia* para a nova classe persistente. A classe *ModeloPersistencia* serve somente de base para a classe *CompiladorJavassist* na transformação de classes persistentes. Portanto, a interface *InterfacePersistencia* não contém métodos a serem implementados, seu papel é servir como indicador para a classe *CompiladorJavassist* no processo de transformação. O *CompiladorJavassist* verifica se a classe a ser transformada implementa a interface *InterfacePersistencia* antes de habilitá-la para a persistência.

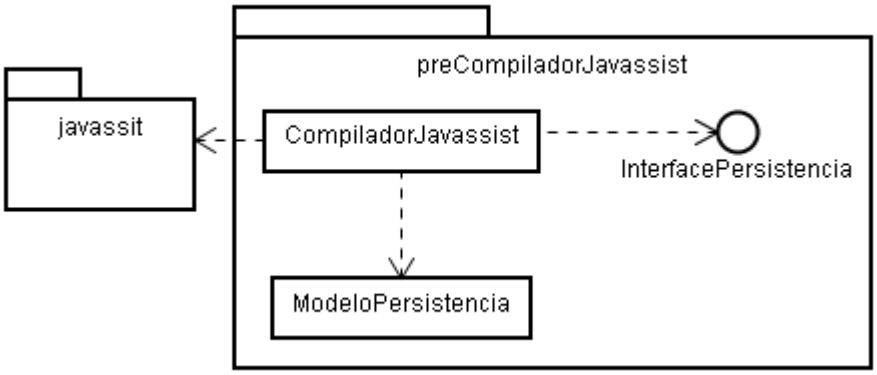


Figura 12 – Pacote preCompiladorJavassist

A classe *ModeloPersistencia* foi criada em função da facilidade de manutenção do pacote *preCompiladorJavassist* na alteração ou implementação de novos serviços persistentes. Neste caso o impacto só será na classe *ModeloPersistencia*, dispensando modificações no código fonte da classe *CompiladorJavassist*, onde é realizada a manipulação do código *bytecode* das classes persistentes.

O mapeamento de uma classe persistente no mapeamento objeto relacional é feito automaticamente no momento de sua instanciação na classe de aplicação. Quando uma classe persistente é instanciada e ainda não consta no mapeamento objeto relacional, o framework realiza o seu cadastramento automático colhendo informações através da reflexão (DEVEGILI,2000) e emite um erro informando que o mapeamento desta classe está incompleto. Esta nova classe fica na condição de cadastrada, aguardando que o administrador de classes e dados complete seu mapeamento e indique que está pronta para uso. Os conceitos de administrador de classes e dados e o funcionamento do mapeamento objeto relacional serão esclarecidos na seção 3.3 e 3.1.5 respectivamente.

3.1.1.3 Fazendo uso de uma Classe Persistente

Para elucidarmos melhor o uso de classes persistentes, elaboramos um roteiro no qual exemplificamos as funcionalidades disponibilizadas pelo FPOR nas classes persistentes, bem como, a forma de utilizá-las e seus requisitos.

Primeiramente a aplicação tem que (i) definir um Mecanismo Persistente e (ii) definir um Mapeamento Objeto Relacional. A aplicação pode tratar objetos persistentes, seguindo os passos abaixo:

a) para criar e persistir um novo objeto:

   I. Instanciar um objeto persistente;

   II. Preencher seus atributos;

   III. Incluir o objeto na base de dados.

b) Para pesquisar um objeto existente, definindo um critério de busca:

   I. Instanciar um objeto persistente;

   II. Instanciar um critério de busca;

   III. Preencher o critério de busca;

   IV. Efetuar a pesquisa sobre o objeto utilizando o critério definido.

c) Pesquisar vários objetos, sem um critério de busca, recuperando todos os objetos existentes na base;

   I. Instanciar um objeto persistente;

   II. Efetuar a pesquisa sobre o objeto;

   III. Navegar pelo resultado da pesquisa;

d) Pesquisar vários objetos, definindo um critério de busca;

   I. Instanciar um objeto persistente;

   II. Instanciar um critério de busca;

   III. Preencher o critério de busca;

   IV. Efetuar a pesquisa sobre o objeto utilizando o critério definido;

V. Navegar pelo resultado da pesquisa;

e) Atualizar um objeto;

I. Instanciar um objeto persistente;

II. Instanciar um critério de busca;

III. Preencher o critério de busca;

IV. Efetuar a pesquisa sobre o objeto utilizando o critério definido.

V. Alterar seus atributos;

VI. Atualizar o objeto na base de dados.

f) Excluir objetos;

I. Instanciar um objeto persistente;

II. Instanciar um critério de busca;

III. Preencher o critério de busca;

IV. Efetuar a pesquisa sobre o objeto utilizando o critério definido.

V. Excluir o objeto na base de dados.

O código fonte listado na Figura 13 é de uma classe de aplicação denominada IntPenalidade que manipula objetos da classe Penalidade que efetua, em ordem, as operações acima descritas.

```
public static void main(String[] args) {
    try
    {
       //
       // Criando um Mecanismo Persistente
       //
       MecanismoPersistente mp = new
              Oracle("usuario","manager","jdbc:oracle:oci8:@ora");
       //
       // Definindo um Mapeamento Objeto Relacional
       //
       Persistencia.defineMapeamento(new
              ParseOracle("mapeamento","manager","jdbc:oracle:oci8:@ora"));
       //
       // Instanciando um novo objeto persistente
       //
       Penalidade pen = new Penalidade();
       //
       // Carregando seus atributos
       //
       pen.setID(1);
       pen.setValor(200);
       pen.setDescricao("Duzentas UFIR's");
       //
       // Realizando a sua inclusão do objeto persistente
       //
       pen.insere(mp);
       mp.efetiva();
       //
       // Realizando uma consulta de um objeto com criterio
       //
       // Criando um critério de pesquisa
       //
       CriterioPesquisa cp1 = pen.criaCriterioPesquisa();
       //
       // Adicionando um critério à pesquisa
       //
       cp1.adicionaCriterio("ID","=","1");
       //
```

```
      // Efetuando a pesquisa, utilizando o critério de pesquisa
      // definido
      if (pen.pesquisaUm(mp,cp1))
      {
         //
         // Caso encontre o objeto, imprime suas propriedades
         //
         System.out.println(pen);
      }
      //
      // Consultando varios objetos, sem criterio
      //
      pen.pesquisa(mp);
      //
      // Navega por todos os objetos recuperados
      //
      while (pen.proximo(mp))
            System.out.println(pen);
      //
      // Consulta varios objetos, com criterio (ID>1)
      //
      CriterioPesquisa cp2 = pen.criaCriterioPesquisa();
      cp2.adicionaCriterio("ID",">","1");
      //
      pen.pesquisa(mp,cp2);
      //
      // Navega por todos os objetos recuperados
      //
      while (pen.proximo(mp))
      {
         System.out.println(pen);
      }
      // Atualizando o objeto:
      //   1. Pesquisa um objeto
      //   2. Atualiza sua descrição para "Atualizado"
      //
      if (pen.pesquisaUm(mp,cp1))
      {
         pen.setDescricao("Atualizado.");
         pen.atualiza(mp);
      }
      mp.efetiva();
```

```
        // Excluindo objetos:
        //   1. Pesquisa os objeto
        //   2. A cada objeto recuperado,
        //      realiza a sua exclusão
        //
        pen.pesquisa(mp);
        while (pen.proximo(mp))
        {
           pen.exclui(mp);
        }
        mp.efetiva();
    }
    catch(Exception ex)
    {
       System.out.println("Problemas..."+ex);
       ex.printStackTrace();
    }
}
```

Figura 13 – Trecho do Código Fonte da Classe IntPenalidade

3.1.2 Pacote *Mecanismo*

O pacote *Mecanismo* define uma classe abstrata *MecanismoPersistente* que proporciona a comunicação do FPOR com os gerenciadores de bancos de dados relacionais. Esta classe deve ser especializada a fim de contemplar um fornecedor específico de SGBD relacional. No diagrama de classes da Figura 14 ilustramos uma especialização desta classe para o ORACLE, o qual produz instruções para o banco de dados relacionais ORACLE.

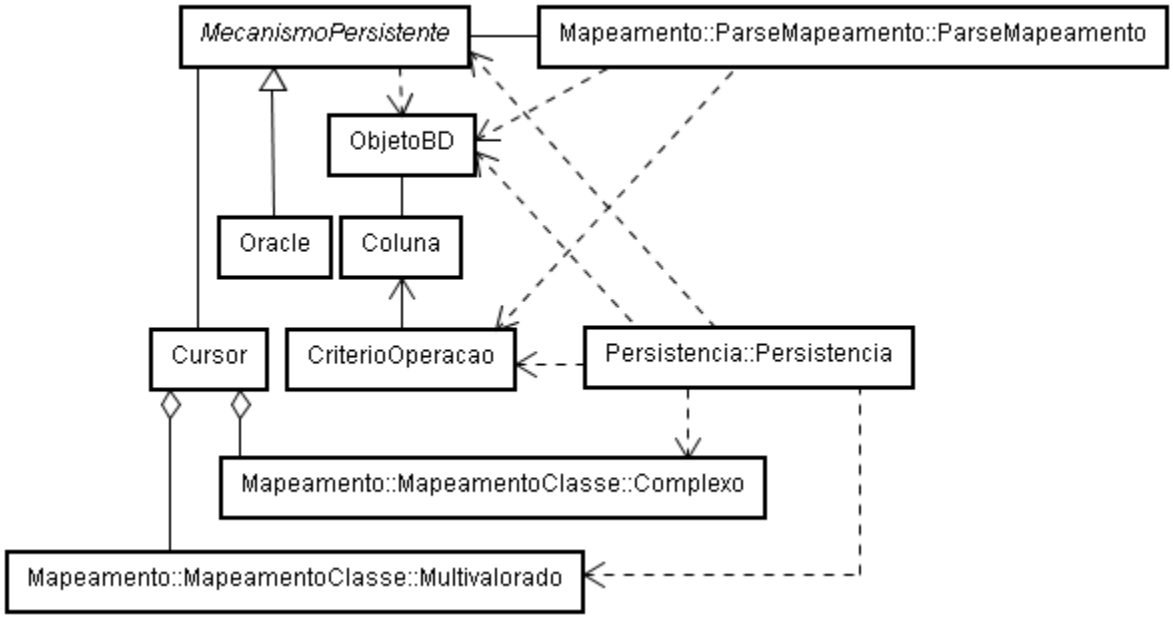


Figura 14 – Pacote Mecanismo

A classe *Cursor* mapeia o conceito de cursor dos SGBD's relacionais. Pesquisas realizadas no meio persistente e manipuladas durante a execução de uma aplicação se tornam uma coleção gerenciada pelo SGBD. O cursor mantém o controle sobre o objeto corrente da coleção e possibilita a navegação para as demais. As classes *Complexo* e *Multivalorado* utilizam a classe *Cursor* na recuperação sob demanda, processo a ser explanado no item 3.2.2.

As classes *ObjetoBD*, *Coluna* e *CriterioOperacao* têm a função de estabelecer uma interface entre a classe *MecanismoPersistente* com as demais classes do FPOR. A preocupação deste trabalho em manter o pacote *Mecanismo* com acoplamento baixo em relação às classes do pacote Mapeamento (LARMAN,2000) levou à criação destas classes. A classe *Persistencia* do pacote *Persistencia* utiliza estas classes na assinatura dos métodos da classe *MecanismoPersistente*.

A classe *ObjetoBD* representa a tabela que sofrerá a ação do mecanismo persistente. As *Coluna's* vinculadas ao *objetoBD* são utilizadas pela classe *MecanismoPersistente* para a montagem do comando. O *CriterioOperacao* é empregado nas operações persistentes que possam ter alguma restrição de filtragem dos objetos que sofrerão a ação. A partir deste conjunto de classes,

associado à ação persistente desejada, a classe *MecanismoPersistencia* tem condições de elaborar as instruções para o meio persistente e assim efetivar os serviços solicitados ao FPOR.

A classe *ParseMapeamento* do pacote *Mapeamento* utiliza a classe *MecanismoPersistencia* em função do repositório do mapeamento objeto relacional estar armazenado em um banco de dados relacional. As características do pacote Mapeamento e o repositório do mapeamento objeto relacional serão elucidadas no item 3.1.3.

3.1.3 Pacote *Mapeamento*

O pacote Mapeamento resolve os desencontros das características da modelagem orientada a objetos em relação ao modelo relacional, colocando elementos como classes, atributos, hierarquias, objetos complexos e agregações em termos de tabelas, visões, colunas e chaves estrangeiras.

As informações contidas neste pacote são recuperadas do repositório do mapeamento objeto relacional do FPOR, implementado em um mecanismo persistente. Os dados pertinentes a uma classe são recuperados para o pacote *Mapeamento* no momento da instanciação do objeto persistente. O pacote *Mapeamento*, então, é utilizado em tempo de execução pelo pacote *Persistencia* para resolver as solicitações de serviços de persistência desta classe persistente.

O modelo e funcionamento do repositório do mapeamento objeto relacional será melhor detalhado neste capítulo no item 3.1.5.

O mapeamento objeto relacional tem como principal característica a flexibilidade, sem impor qualquer limitação em relação à correspondência tecnológica entre a orientação objeto e relacional. Atributos podem ser equivalentes a colunas de diferentes tabelas, diferentes classes podem estar implementadas em uma mesma tabela ou, ao invés, uma classe pode corresponder a mais de uma

tabela, são alguns exemplos das possibilidades permitidas pelo pacote *Mapeamento*. Alguns preceitos que devem ser seguidos a fim de evitar um alto grau de complexidade na compatibilização dos diferentes modelos. As regras de equivalência que norteiam o mapeamento objeto relacional serão elucidados no item 3.1.3.7.

O modelo do pacote *Mapeamento* é composto por 5 pacotes:

- *MapeamentoClasse* que mapeia as informações pertinentes à classe persistente;
- *MapeamentoRelacionamentoClasse* que representa os possíveis relacionamentos entre classes;
- *MapeamentoBase* que retrata os elementos da tecnologia relacional utilizados pelo framework de persistência;
- *Correspondencia* que realiza a correlação entre os pacotes *MapeamentoClasse*, *MapeamentoRelacionamentoClasse* e *MapeamentoBase*;
- *ParseMapeamento* que recupera as informações do repositório do mapeamento objeto relacional.

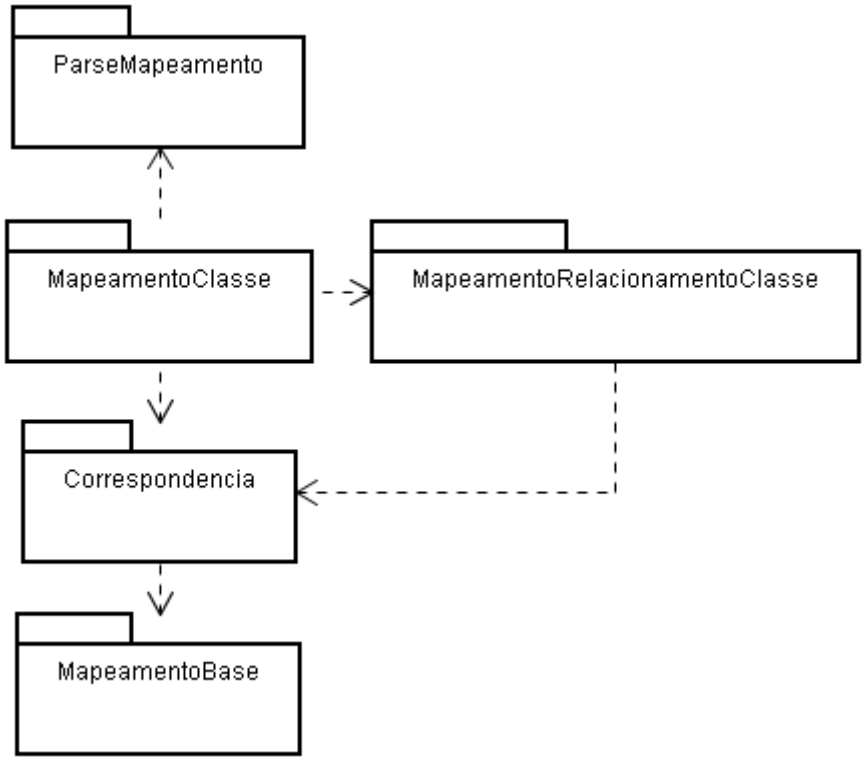


Figura 15 – Pacotes do Mapeamento

3.1.3.1 Pacote *MapeamentoClasse*

O *MapeamentoClasse* corresponde à classe do objeto e o *MapeamentoAtributo* representa seus atributos. Um *MapeamentoAtributo* pode ser especializado em *Primitivo* ou *Complexo*. Um *MapeamentoAtributo* é *Complexo* quando é uma referência a outra classe de objeto persistente. Um *MapeamentoAtributo Primitivo* pode ser *Multivalorado*, quando é um vetor, ou *Simples*, quando for de tipo primitivo da linguagem.

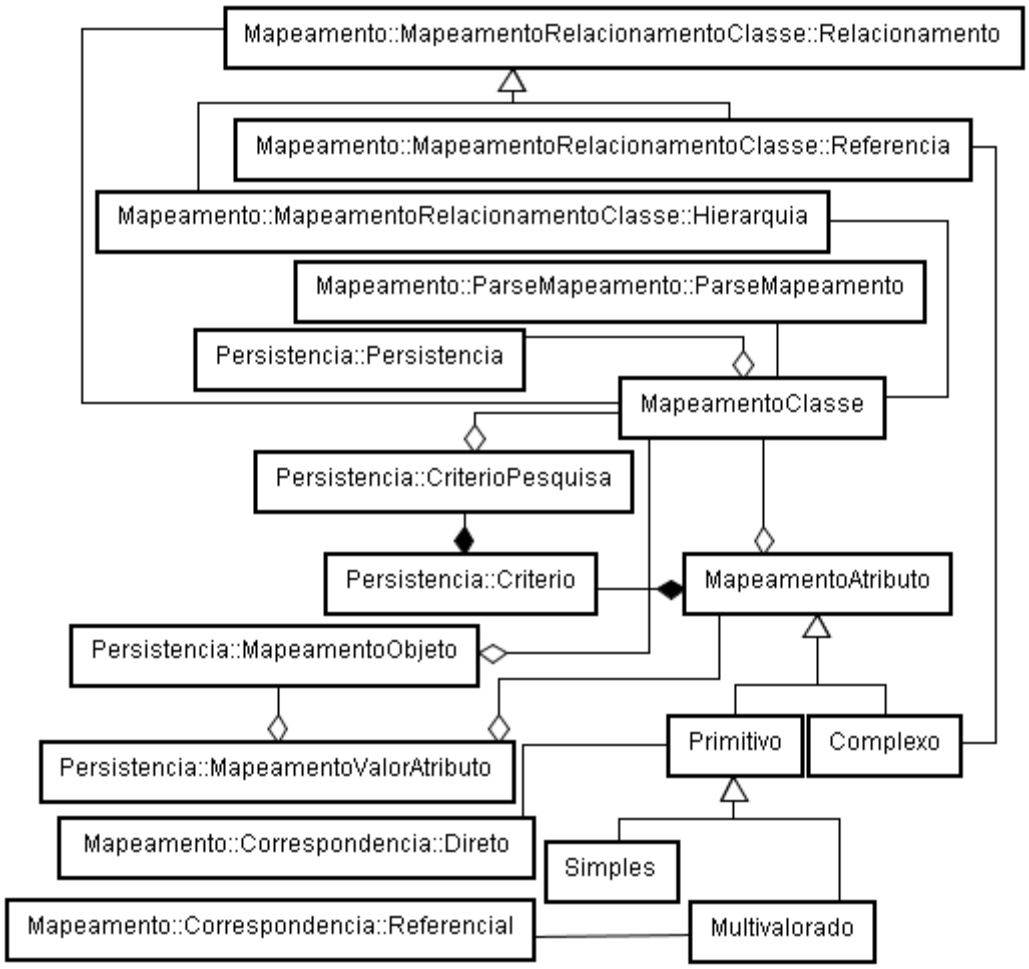


Figura 16 – Pacote MapeamentoClasse

A Figura 17 demonstra o modelo da classe persistente *Veiculo*. A classe *MapeamentoClasse* mapeia as características da classe *Veiculo*. A classe *MapeamentoAtributo* mapeia os atributos da classe Veiculo. Os atributos *renavan*, *placa*, *chassi*, *acessorios*, *modelo* e *cilindradas* são considerados atributos primitivos, pois são do tipo *number*, *string* e *array* da linguagem java. O atributo *acessorios* exemplifica uma especialização da classe *MapeamentoAtributo* denominada *Multivalorado*, pois seu conteúdo prevê uma coleção de *string*. O mapeamento objeto relacional do FPOR dispensa que o atributo acessórios seja resolvido através de classes de implementação com chave e valores, pois prevê sua persistência em uma tabela com esta finalidade, conforme seção 3.1.3.3. O atributo *proprietario* é uma referência à classe *Pessoa*, portanto é mapeado como uma especialização do *MapeamentoAtributo* denominada *Complexo*.

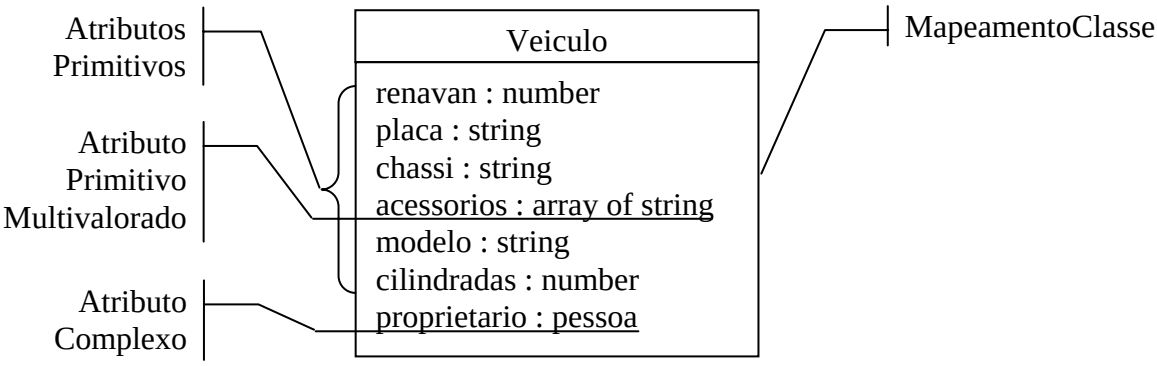


Figura 17 – Exemplo de Mapeamento de uma Classe Persistente

3.1.3.2 Pacote *MapeamentoRelacionamentoClasse*

A classe *Relacionamento* representa os possíveis relacionamentos entre classes. A classe *Relacionamento* pode ser especializada em *Hierarquia* ou em *Referencia* que por sua vez pode ser especializada em *Agregação/Composição* ou uma simples *Associação*. Para cada especialização de *Referencia* existe um conjunto de especializações referentes a cardinalidade do relacionamento representada no modelo da Figura 18.

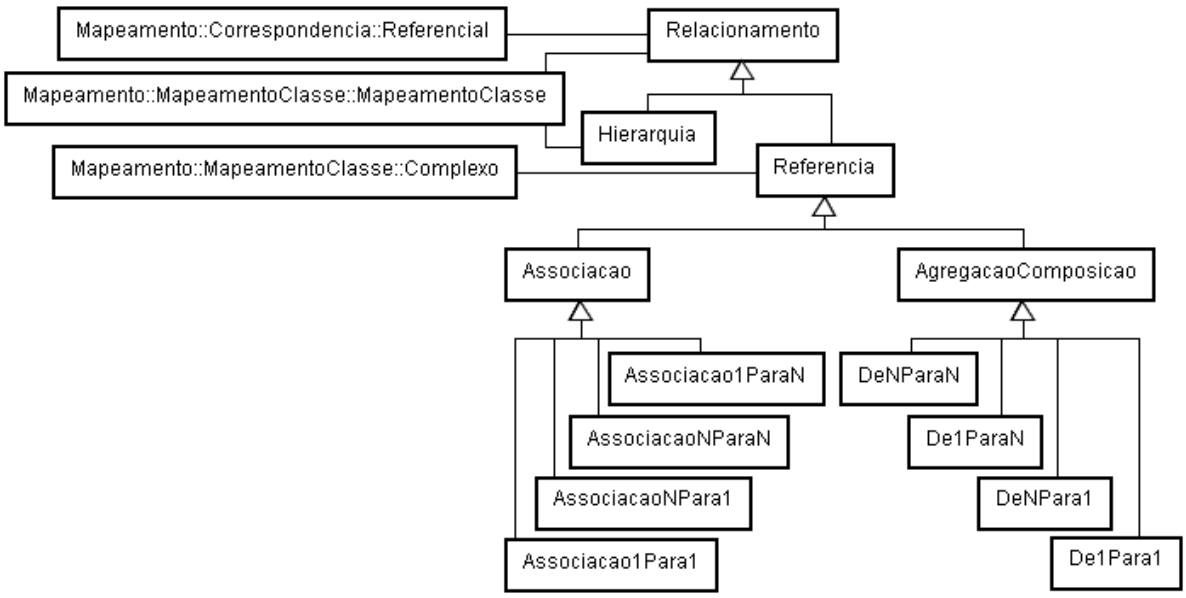


Figura 18 – Pacote MapeamentoRelacionamentoClasse

Um *MapeamentoAtributo Complexo* se refere a um *MapeamentoClasse* e este tipo de relacionamento é representado através da classe *Referencia*. Um *MapeamentoClasse* pode se relacionar a outro *MapeamentoClasse* através de hierarquias de generalização ou especializações. Este outro tipo de relacionamento está representado pela classe *Hierarquia*. Todo relacionamento entre classes, sempre terá um *MapeamentoClasse* em uma das suas extremidades. Caso seja uma *Hierarquia*, haverá outro *MapeamentoClasse* na extremidade oposta. No caso de *Associacao*, a outra extremidade do relacionamento será um *MapeamentoAtributo Complexo*.

3.1.3.3 Pacote *Correspondencia*

A principal classe deste pacote é a *Equivalencia* que pode ser especializada em *Direto* ou *Referencial*, conforme demonstrado na Figura 19. Uma *Equivalencia Direta* é utilizada no mapeamento de atributos primitivos que são diretamente relacionados a colunas de uma tabela ou visão. A *Equivalencia Referencial* associa os objetos *Relacionamento* do pacote *MapeamentoRelacionamentoClasse* com os objetos *ChaveEstrangeira* do pacote *MapeamentoBase*. Conceituamos que todo relacionamento entre classes é implementado no ambiente de banco de dados através de uma chave estrangeira. Nos casos de relacionamentos entre classe com cardinalidade muitos para muitos, temos um tipo de *Equivalencia Referencial* denominada *Associativa*. A *Equivalencia Referencial Associativa* demanda uma chave estrangeira a mais no ambiente de banco de dados.

Atributos primitivos multivalorados, além de serem relacionados diretamente às colunas de tabelas, também são relacionados a uma *Equivalencia Referencial* pelo fato de seu conteúdo estar contido em uma diferente tabela ou visão no banco de dados.

Para cada relacionamento mapeado, deve-se definir no mapeamento objeto relacional o controle de relacionamentos entre objetos, discutido no capítulo 2, no qual indica se as operações de inclusão, alteração ou exclusão devem ser em cascata. O administrador de classes e dados deve ter cautela na definição da

exclusão em cascata. Dados de objetos adjacentes podem estar relacionados a outros registros no banco de dados. Neste caso o banco de dados impedirá a exclusão do objeto persistente informando quebra de integridade referencial com a exclusão de um objeto adjacente.

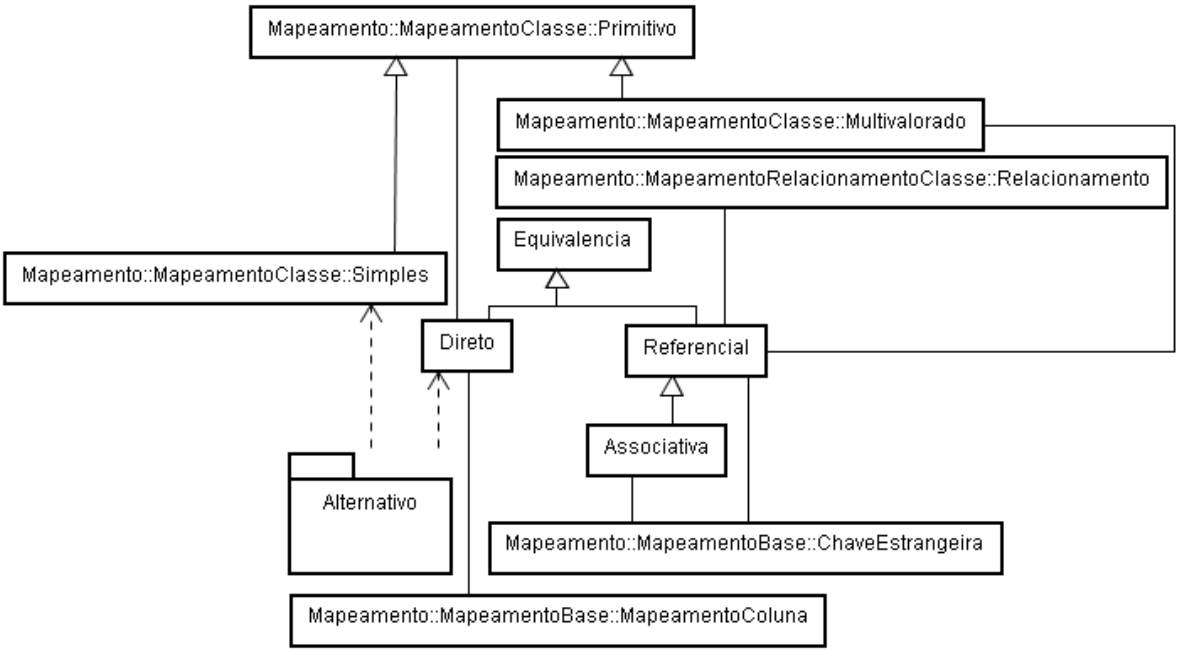


Figura 19 – Pacote Correspondencia

As classes *ViaVisao* e *ComandoSQL* do pacote *Alternativa*, demonstrados na Figura 20, são uma opção de mapeamento de atributo, caso estes não possam ser mapeados diretamente a uma coluna de uma tabela, é possível mapeá-lo a uma coluna de visão que permite o uso do framework a base de dados legadas. Contudo, as operações de persistência ficam prejudicadas, tendo em vista que o FPOR não poderá realizar as operações no meio persistente sobre uma visão. Nestes casos deve ser informado, no mapeamento objeto relacional, o comando SQL que deverá ser feito para as transações de inclusão, alteração e exclusão, na classe *ComandoSQL*. Para cada um destes comandos deverão ser informados os *MapeamentoAtributo's* cujos valores serão utilizados como parâmetros. Apesar de prevista, esta funcionalidade não foi implementada neste trabalho. É um recurso alternativo.

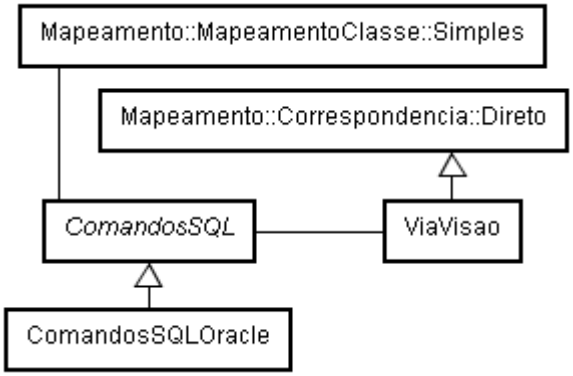


Figura 20 – Pacote Alternativo

3.1.3.4 Pacote MapeamentoBase

A classe *MapeamentoObjetoBD* corresponde a tabelas ou visões e *MapeamentoColuna* representa suas colunas no ambiente de banco de dados, conforme Figura 21.

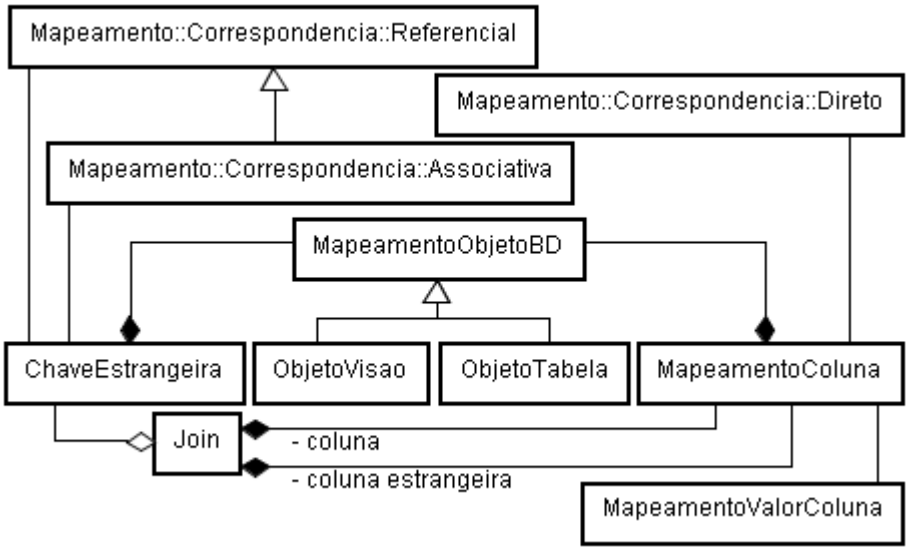


Figura 21 – Pacote MapeamentoBase

A classe *ChaveEstrangeira* corresponde aos objetos de banco de dados responsáveis pela integridade referencial entre tabelas ou visões no banco de dados. A classe *Join* representa pares de *MapeamentoColuna's*, unidas em função

de uma integridade referencial a ser garantida no banco de dados. Uma *ChaveEstrangeira* pode agregar um ou vários *Joins* para garantir uma integridade referencial de tabelas ou visões distintas. O parâmetro utilizado para garantir esta integridade é o par de colunas existentes no *Join*, sendo cada coluna oriunda de tabelas ou visões distintas, porém componentes de um relacionamento, ou melhor, de uma chave estrangeira. Uma das colunas que consta no Join sempre pertence à mesma tabela da chave estrangeira.

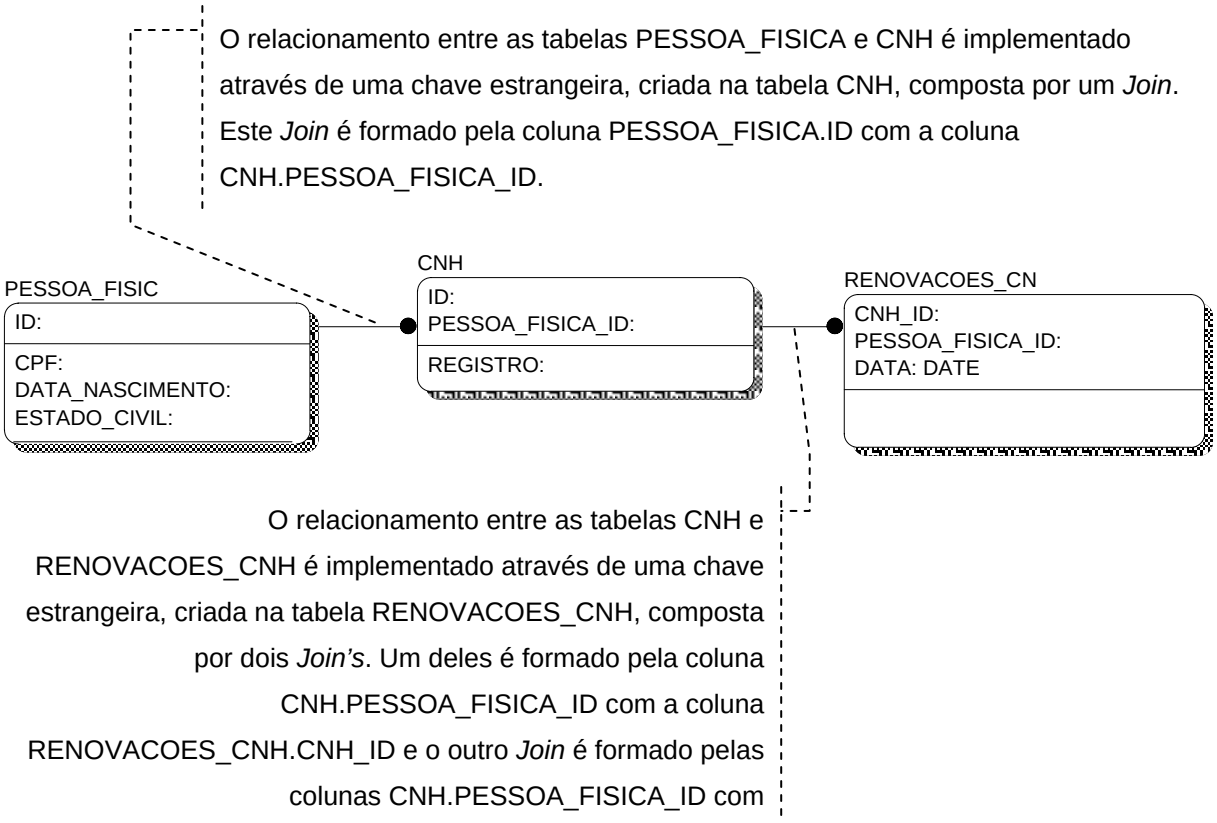


Figura 22 – Chave Estrangeira e Join

O relacionamento denominado *coluna* é caracterizado pelo fato do *MapeamentoColuna* se referir ao mesmo *MapeamentoObjetoBD* associado à *ChaveEstrangeira* do *Join*. O relacionamento denominado *coluna estrangeira* é caracterizado pelo fato do *MapeamentoColuna* se referir a um *MapeamentoObjetoBD* diferente do que estiver associado à *ChaveEstrangeira* do *Join*.

No *Join* do relacionamento entre as tabelas PESSOA_FISICA e CNH, que consta na Figura 22, o MapeamentoColuna que representar a coluna CNH.PESSOA_FISICA_ID será considerado como *coluna*, pois pertence a mesma tabela que implementou a chave estrangeira, isto é, a tabela CNH. Cabe à coluna PESSOA_FISICA.ID o papel de *coluna estrangeira* neste *Join*.

3.1.3.5 Pacote ParseMapeamento

A classe *ParseMapeamento* realiza o intercâmbio do FPOR com o repositório do mapeamento objeto relacional. A classe *MapeamentoClasse* se utiliza da classe *ParseMapeamento* na recuperação das informações pertinentes à classe persistente que mapeia.

Similar à classe *MecanismoPersistente*, esta classe pode ser especializada por novas classes em função de novos mapeamentos objeto relacionais proporcionando uma independência do framework em relação ao modelo do mapeamento objeto relacional. Esta classe utiliza um mecanismo de persistência, tendo em vista que as informações a respeito do mapeamento objeto relacional são armazenadas em um meio persistente.

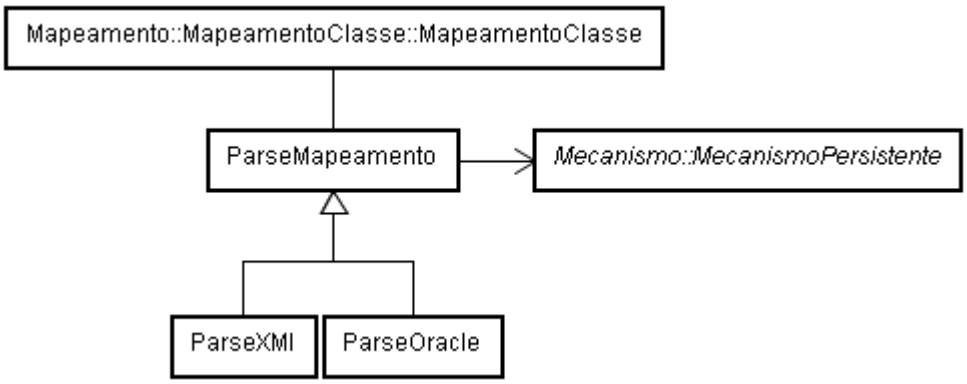


Figura 23 – Pacote ParseMapeamento

3.1.3.6 Características da Implementação

A adoção da tecnologia de banco de dados relacionais resolveu questões de integridade referencial, transação e *cache*, dispensando o FPOR de implementar algoritmos que tratem estes controles, uma vez que o gerenciador de banco de dados realiza estas tarefas com eficiência comprovada.

A técnica do *cache* adicionaria considerável complexidade ao framework em função do controle da replicação de objetos em memória no sentido de se evitar

colisões entre dados originais e cópia. O banco de dados realiza este controle, mantendo dados em memória *cache*, reduzindo o número de acessos a disco.

A integridade referencial através do framework seria centralizada em um simples repositório de mapeamento objeto relacional. Qualquer acesso a dados do objeto teria que ser programado a usar o framework de persistência ou o mesmo mapeamento objeto relacional. É sabido que sistemas implementados em diferentes tecnologias fazem uso da mesma base de dados. Esta escolha de implementação obriga que estes sistemas se adaptem a este framework para que atualizassem de forma segura as suas informações. A concentração deste controle no banco de dados é a alternativa mais natural, pois além do motivo anteriormente citado, a elaboração de *constraints* é feita de forma automática, a partir da definição de um relacionamento, em ferramentas de modelagem de dados de mercado já consagradas.

Adicionar suporte a transações em aplicações orientadas a objetos não é uma tarefa trivial. Qualquer comportamento invocado como parte de uma transação requer suporte às operações de submissão, *commit* e *rollback*. As transações no banco de dados podem ser iniciadas, submetidas e efetivadas através de código SQL. API’s de banco de dados como JDBC e ODBC fornecem classes que suportam as funcionalidades básicas transacionais.

A possibilidade do bloqueio de registros na base de dados é um recurso fundamental no controle de concorrência do framework. Sem esta funcionalidade, o FPOR teria que controlar todos os acessos de dados no mecanismo persistente, o que agregaria uma complexidade que não estaria em seu escopo de atuação. Esta funcionalidade no FPOR traria problemas quando o acesso a informação fosse efetivado sem o uso do framework (sistemas legados, por exemplo).

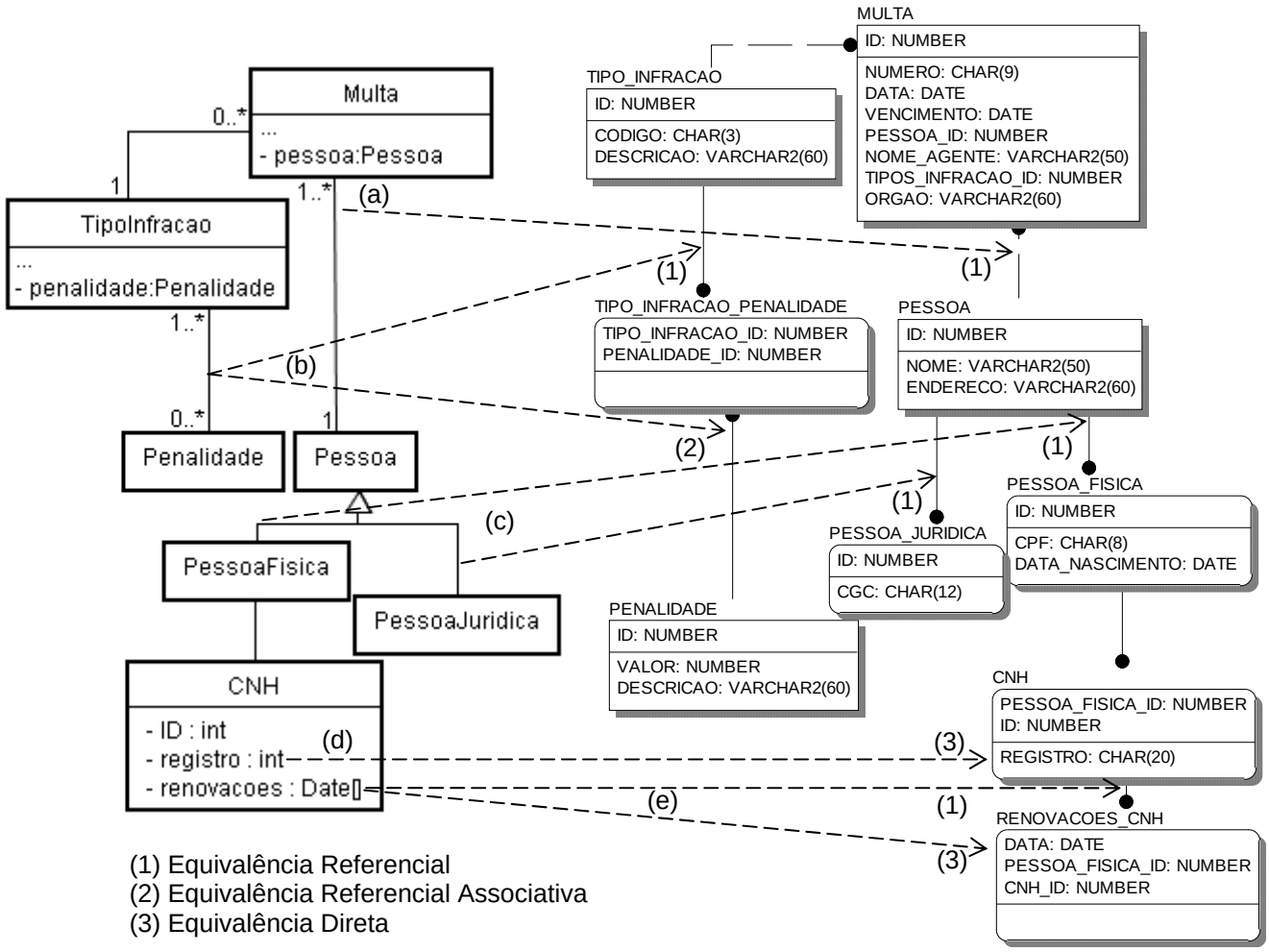
Usamos as vantagens da tecnologia de banco de dados relacional, principalmente as técnicas desenvolvidas ao longo de sua existência no tratamento de questões de desempenho e organização no armazenamento de dados. Esta estratégia dispensa o ambiente de aplicação do tratamento destas questões, podendo se dedicar melhor à manipulação das informações.

3.1.3.7 Elaborando um Mapeamento

A elaboração de um Modelo Objeto relacional tem padrões difundidos em diversas publicações (AMBLER,2000a;AMBLER,1998;KELLER,1997). Refinando as técnicas existentes, colocamos algumas boas práticas que devem ser seguidas na elaboração do mapeamento objeto relacional com a finalidade de se evitar uma complexidade maior na compreensão do sistema de objetos persistentes.

O mapeamento de cada classe, naturalmente, deve ser feito para uma tabela, bem como seus atributos devem ser mapeados a colunas desta tabela de forma direta. Todas as associações, inclusive as que representam hierarquia de generalização ou especialização, devem seguir a premissa anterior, isto é, cada classe deve corresponder a uma tabela no banco de dados, pois proporciona uma facilidade de entendimento. Relacionamentos entre objetos necessariamente devem ter uma correspondente chave estrangeira no banco de dados com a finalidade de garantir a integridade referencial. Relacionamentos de cardinalidade um para um tem a particularidade de sua integridade no banco de dados ser somente garantida em uma das extremidades. Relacionamentos de cardinalidade muitos para muitos pressupõem a criação de uma tabela específica para o registro de suas associações.

Utilizaremos o exemplo ilustrativo da Figura 24 para elucidar melhor a identificação dos elementos do modelo de classe e entidade relacionamento, bem como indicar a correspondência destes elementos.



(1) Equivalência Referencial
(2) Equivalência Referencial Associativa
(3) Equivalência Direta

(a) Atributo Complexo (Associação)
(b) Atributo Complexo (Associação de cardinalidade n para n)
(c) Hierarquia
(d) Atributo Primitivo Simples
(e) Atributo Primitivo Multivalorado

Figura 24 – Exemplo de Mapeamento

A Figura 24 contém à sua esquerda um modelo de classes e à sua direita o modelo de entidade e relacionamento de um mesmo aplicativo. As setas tracejadas simbolizam o mapeamento destes dois modelos.

Os mapeamentos ilustrados estão identificados em letras e estão descritos nos tópicos a seguir:

(a) O mapeamento da associação entre as classes *Multa* e *Pessoa* é feita através de um atributo complexo *pessoa* da classe *Multa* que referencia a classe *Pessoa*. O atributo complexo é correlacionado a chave estrangeira

que garante a integridade das tabelas *MULTA* e *PESSOA*, que implementam as classes *Multa* e *Pessoa* respectivamente. Esta correlação é denominada equivalência referencial.

(b) A associação das classes *TipoInfracao* e *Penalidade* é de cardinalidade muitos para muitos. Este tipo de associação prevê a criação de uma tabela denominada associativa, implementada por TIPO_INFRACAO_PENALIDADE. Portanto, o mapeamento deste tipo de relacionamento prevê a correlação do atributo complexo *penalidade* da classe *TipoPenalidade* com duas chaves estrangeiras. Uma destas é a que garante a integridade das tabelas TIPO_INFRACAO e TIPO_INFRACAO_PENALIDADE e a outra é responsável pelo relacionamento entre as tabelas TIPO_INFRACAO_PENALIDADE e a tabela PENALIDADE. O mapeamento deste tipo de associação origina uma especialização da equivalência referencial denominada associativa.

(c) A classe *Pessoa*, que é especializada nas classes *PessoaFisica* e *PessoaJuridica*, adotou a política de mapeamento de uma tabela para cada classe. Portanto, temos: classe Pessoa relacionada com a classe *PessoaFisica* e a classe *Pessoa* relacionada com *PessoaJuridica*. Cada relacionamento deve corresponder a uma chave estrangeira no ambiente de banco de dados.

(d) Este mapeamento demonstra a simples correlação do atributo primitivo simples denominado *registro* da classe *CNH* a coluna REGISTRO da tabela CNH. O mapeamento deste tipo de atributo origina equivalência direta.

(e) O atributo *renovacoes* da classe *CNH* é uma coleção de datas. Este tipo de atributo primitivo, denominado *multivalorado*, necessita de uma tabela destinada ao armazenamento de seus valores. A tabela empregada para tal fim é denominada RENOVACOES_CNH. Portanto, o seu mapeamento demanda além de uma equivalência referencial, uma equivalência direta que indique a coluna DATA da tabela RENOVACOES_CNH que armazena seus valores. Não se admite que uma coleção contenha

elementos de diferentes tipos de dados, excluindo os casos em que seja diferentes classes da mesma hierarquia.

3.1.4 Gerenciando as Informações de Mapeamento

As opções atualmente utilizadas para o armazenamento das informações pertinentes ao mapeamento objeto relacional nos frameworks de persistência de mercado pesquisados têm sido o XML e um banco de dados (HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1, 2003; COCOBASE O/R MAPPING – BASIC CONCEPTS AND QUICK START GUIDE, 2001; RODRIGUES,2002).

Os documentos XML têm como principal vantagem a portabilidade, pois o XML é um padrão aberto, sem dependência de fornecedores, e estes documentos podem ser criados usando qualquer editor de texto. A estrutura de sintaxe *tag* do XML faz com que seja simples de ler como o HTML, e é claramente superior para transmitir informações estruturadas.

Apesar de todas as vantagens citadas, o XML, quando usado no armazenamento de informações de mapeamento objeto relacional, tem a carência de garantir uma integridade dos dados. Além disso, as API's e modelos de criação e manipulação de arquivos XML são de alta complexidade, demando um razoável tempo de aprendizado e implementação.

O FPOR emprega um repositório de mapeamento objeto relacional implementado diretamente em um banco de dados Oracle, dentro de uma tecnologia relacional. Esta opção foi adotada em função do desempenho para o acesso às informações, garantia de integridade e facilidade de implementação. Contudo, o mapeamento objeto relacional se utiliza do pacote *Mapeamento* (seção 3.1.2) para a persistência de suas informações, possibilitando que seu repositório esteja em qualquer mecanismo persistente suportado pelo pacote *Mecanismo* do FPOR, podendo até ser diferente daquele que contém os objetos persistentes da aplicação.

A recuperação das informações do repositório é sob demanda, isto é, o FPOR recupera as informações pertinentes a classe persistente que tiver algum objeto instanciado pela aplicação. Esta característica proporciona um melhor desempenho ao framework, pois restringe a quantidade de dados a respeito do mapeamento objeto relacional manipulados durante a sua execução.

Com o intuito de se evitar o acoplamento do mapeamento objeto relacional a somente um fornecedor de banco de dados, sugerimos a exportação e importação de sua estrutura e conteúdo em qualquer mecanismo de persistência através de arquivos XML, seguindo um modelo XML Schema. Existem aplicativos disponíveis, como por exemplo o XML Spy v4.3 (http://www.xmlspy.com), que já realiza esta tarefa a partir de um banco de dados, através de arquivos XML.

3.1.4.1 Modelo Físico das Informações do Mapeamento

O modelo físico das informações do mapeamento contém somente as informações que devem ser persistidas para o funcionamento do FPOR.

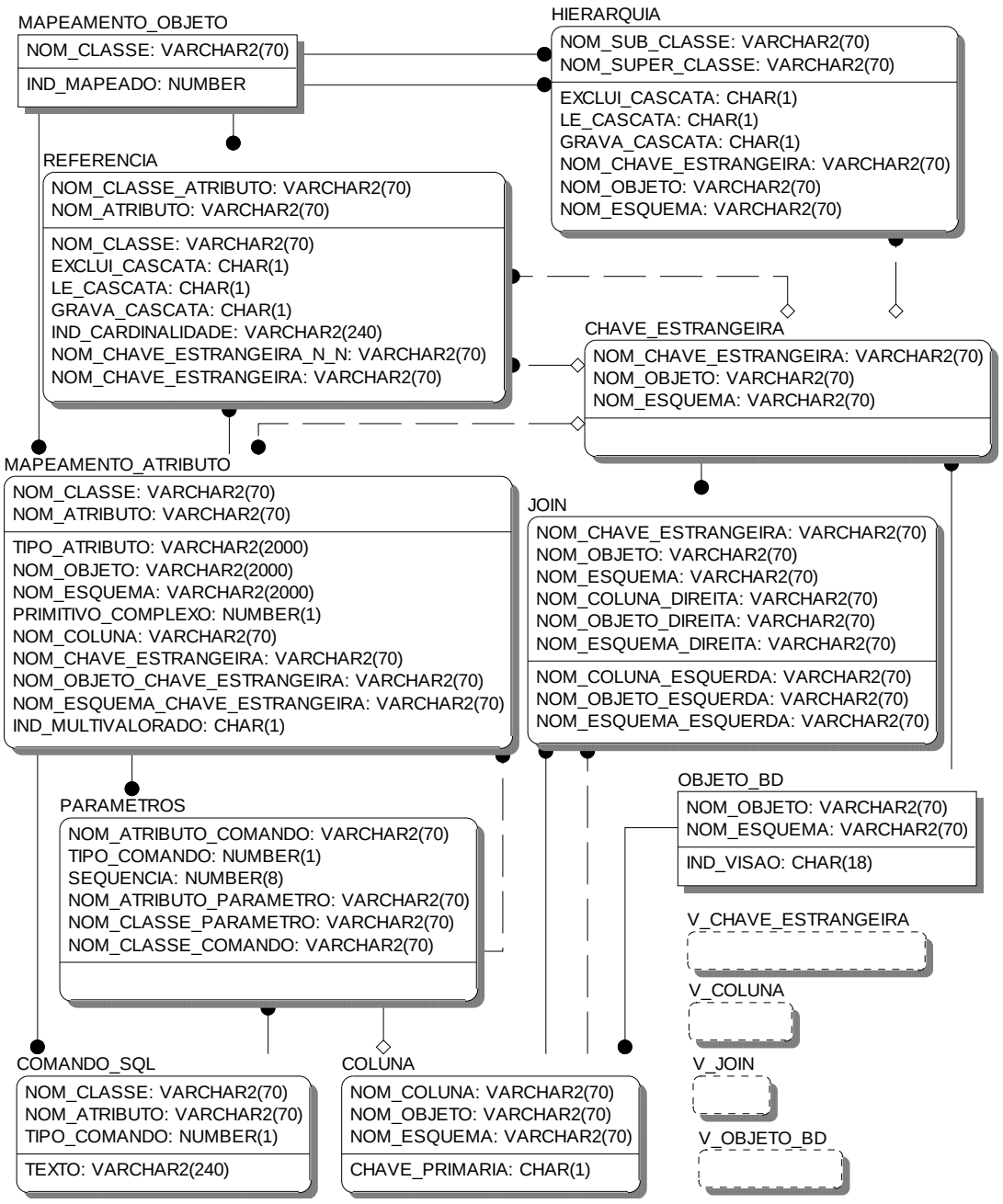


Figura 25 – Modelo de Entidade Relacionamento do Mapeamento Objeto

O modelo de entidade e relacionamento implementa as classes do pacote *Mapeamento*. As visões V_CHAVE_ESTRANGEIRA, V_COLUNA, V_JOIN e

V_OBJETO_BD são utilizadas para a coleta de dados do metadados do banco de dados descrito no item a seguir.

3.1.4.2 Conceitos e Facilidades Empregadas na Implementação

Na implementação do mapeamento objeto relacional utilizamos alguns conceitos e facilidades, já mencionados neste capítulo, disponíveis no ambiente SGBD e da orientação a objetos. Explanaremos nesta seção estes conceitos e facilidades.

Metadados

Metadados são informações a respeito de dados. Definições como classes e tabelas são considerados metadados. Os modelos são metadados, pois descrevem conceitos e coisas que estão sendo modelados. Muitos ambientes disponibilizam metadados para consulta, como o gerenciador de banco de dados através de visões estáticas do dicionário de dados que armazenam definições de objetos de banco de dados. Nas linguagens de programação os metadados são disponibilizados através da reflexão (DEVEGILI,2000).

Dicionário de Dados

Sistemas gerenciadores de banco de dados relacionais utilizam metadados e disponibilizam estas informações através de diversas "metavisões" que fornecem informações a respeito de todos os objetos de um banco de dados, como por exemplo, informações a respeito de objetos que implementam integridades referenciais, estrutura de tabelas, características de colunas e chaves primárias (ORACLE8I SERVER ONLINE DOCUMENTATION,2000).

Estas visões são ditas estáticas em função do seu conteúdo lógico só poder ser alterado quando ocorrem manutenções em objetos de banco de dados, como criação de tabelas, manutenção de visões, e outros.

Os principais usos de Metadados pelo gerenciador de banco de dados relacional são:

- gerenciador do banco de dados acessa o metadados para buscar informações a respeito de usuários, objetos e estruturas de armazenamento;

- gerenciador do banco de dados modifica o metadados toda vez que a estrutura de um objeto de banco de dados é modificada através da submissão de uma DDL - Data Definition Language.

Metadados é fundamental para o funcionamento do gerenciador de banco de dados, porém somente o gerenciador pode atualizar suas informações. Durante suas operações, o gerenciador obtém informações do metadados para assegurar que determinado objeto existe e quais usuários têm acesso a este objeto. São atualizados para refletir as alterações de estruturas e privilégios.

Metadados também é uma ferramenta importante para projetistas de aplicações e administradores de banco de dados. Podem ser acessados somente para consulta através de comandos SQL nas visões estáticas do dicionário de dados.

Reflexão

Segundo Devegili(2000), uma linguagem ou ambiente de programação que possua uma arquitetura reflexiva deve garantir que (MAES,1988):

- as aplicações tenham acesso aos dados que representam aspectos sobre si mesma (auto-representação);

- qualquer mudança nestes dados seja causada por mudanças nos aspectos sobre o sistema, e que mudanças nos aspectos sobre os sistemas sejam refletidas nos dados que os representam (conexão causal).

Neste mesmo trabalho (DEVEGILI,2000), é indicada a persistência como uma das aplicações da reflexão.

Reflexão computacional é toda a atividade de um sistema computacional realizada sobre si mesmo, e de forma separada das computações em curso, com o

objetivo de resolver seus próprios problemas e obter informações sobre suas computações.

Reflexão como técnica de programação consiste em obter informações dinâmicas sobre objetos: classe, superclasse, campos, seus tipos e valores, métodos, construtores, etc. e usar estas informações para fazer transformações durante a execução do programa. Este recurso na linguagem JAVA está disponível através de classes do pacote Java.lang.reflect.

3.1.5 Especificando um Mapeamento no FPOR

O FPOR realiza de forma automática o mapeamento de classes, através da reflexão, no repositório do mapeamento objeto relacional. Um processo similar ocorre em relação à base de dados, no qual um aplicativo de gerenciamento (FPOR-MAP) cadastra os elementos de um determinado esquema de banco de dados através do acesso aos seus metadados. O mapeamento objeto relacional é simplesmente finalizado com a equivalência dos dois modelos, que é feito através do aplicativo de mapeamento objeto relacional.

Quando uma classe persistente é instanciada e ainda não consta no mapeamento objeto relacional, o framework realiza o seu cadastramento automático colhendo informações através da reflexão e emite um erro informando que o mapeamento desta classe está incompleto. Esta nova classe fica na condição de cadastrada, aguardando que o administrador de classes e dados complete seu mapeamento objeto relacional e indique que está pronta para uso.

3.1.5.1 Instanciando o Mapeamento de uma Classe Persistente

O FPOR-MAP é dividido em duas pastas: classe e base. A pasta de classe contém as informações pertinentes às classes de negócio de objetos persistentes extraídas através da reflexão. Estas informações podem ser alteradas ou até preenchidas pelo administrador do aplicativo. Os campos que representam o mapeamento de classes persistentes estão destacadas na Figura 26.

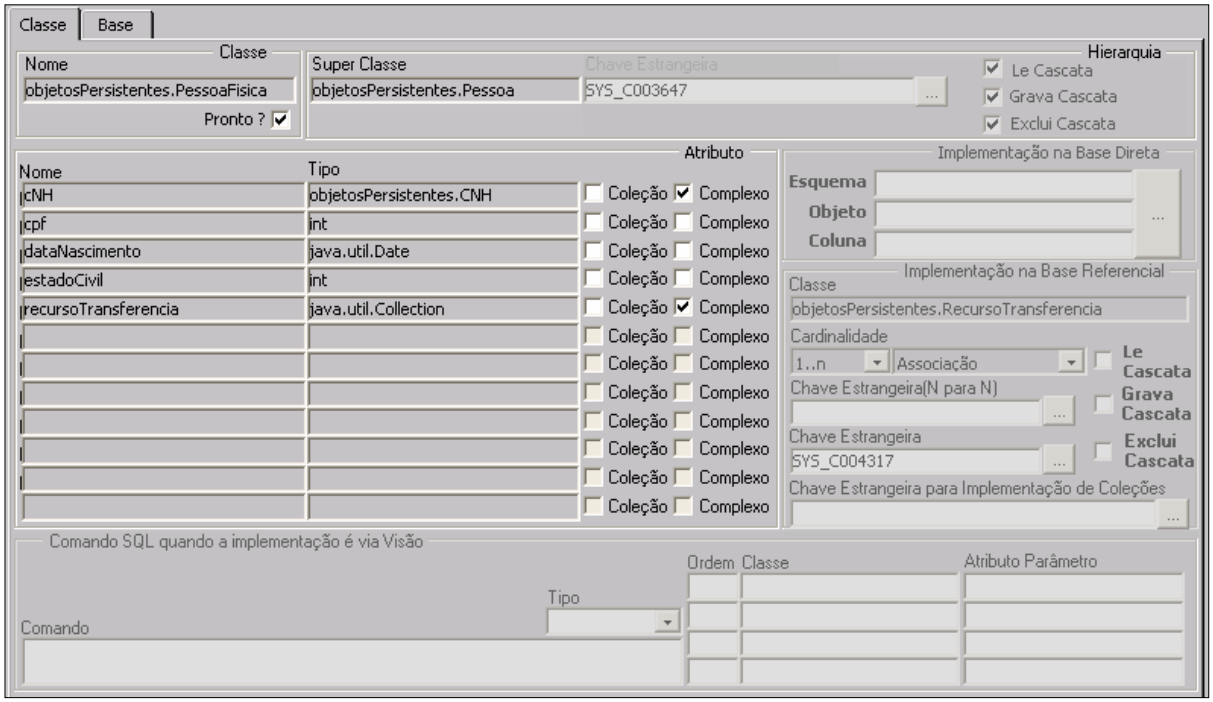


Figura 26 – Aplicativo de Gerenciamento – Classes

3.1.5.2 Recuperando Informações do Modelo Físico da Base de Dados Existente

A pasta Base (Figura 26) tem todas as informações a respeito dos objetos do ambiente de banco de dados. Estas informações podem ser importadas do banco de dados através do botão “Carrega Dados a Respeito do Esquema na Base Oracle”, bastando informar o esquema onde estão implementados os objetos de banco de dados da aplicação.

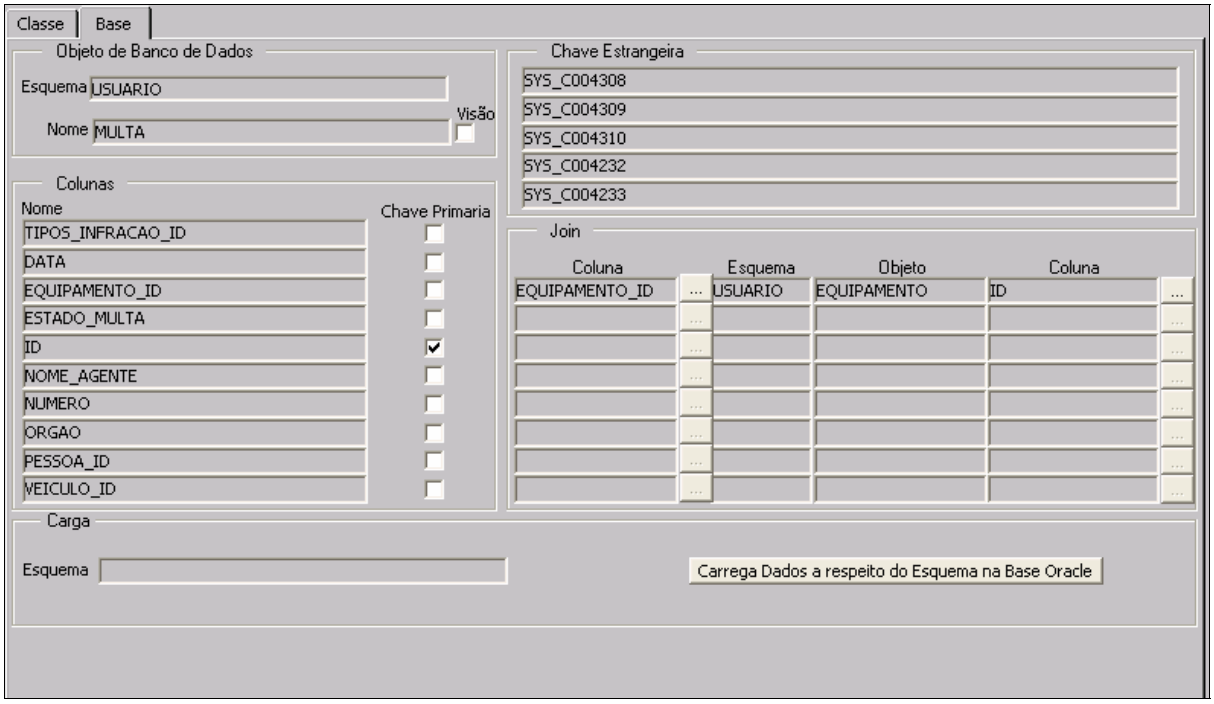


Figura 27 – Aplicativo de Gerenciamento – Base

3.1.5.3 Relacionando Classes e Base de Dados

Na pasta Classe também é informada a correspondência entre os elementos das classes persistentes e sua implementação no ambiente de banco de dados. Os campos de preenchimento estão destacados na Figura 28. Aqueles campos que corresponderem a algum objeto do banco de dados contêm um botão, simbolizado com "...", que possibilita a escolha de um objeto a partir de uma lista em tela.

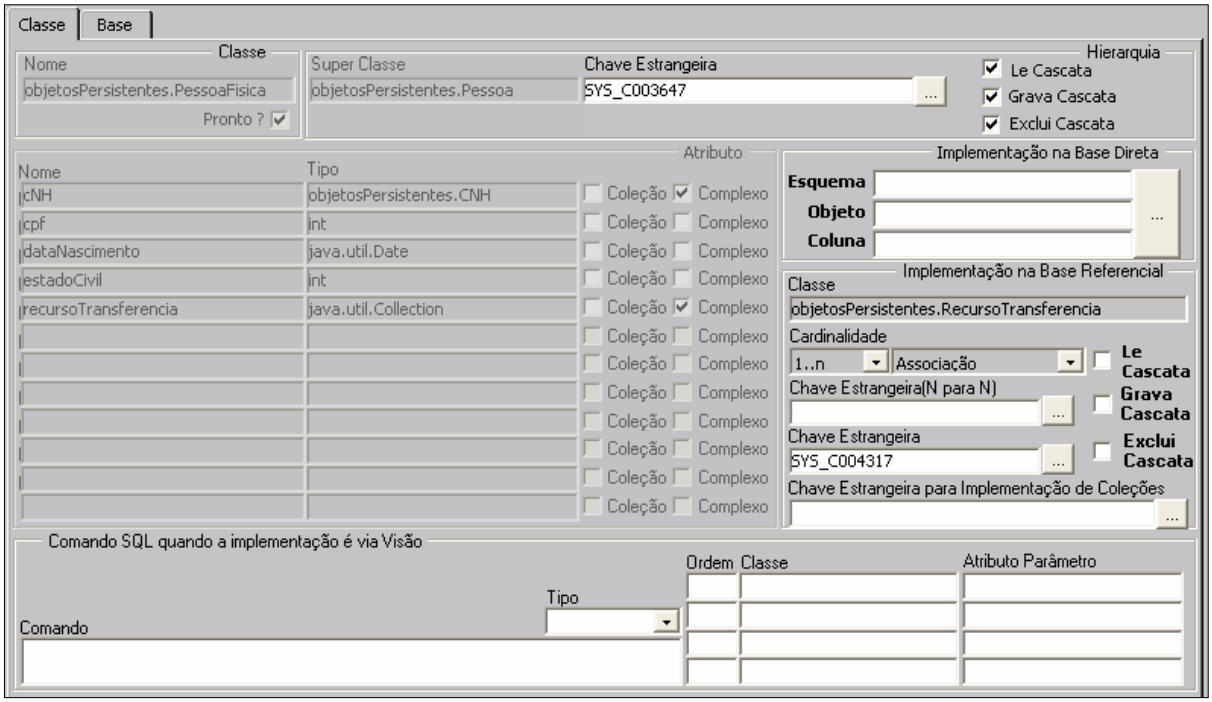


Figura 28 – Aplicativo de Gerenciamento – Classes (Correspondência)

3.2 DINÂMICA DE FUNCIONAMENTO DO FPOR

Paulatinamente explicaremos através de diagramas de atividade o funcionamento do framework, desde uma visão macro até os detalhes de funcionamento de seus componentes. Não temos a pretensão de esgotar todas as dúvidas a respeito da complexidade de seu funcionamento, mas pelo menos dar condições para a compreensão do modelo em questão.

Uma classe persistente é manipulada por outras classes da aplicação através de serviços persistentes. Estes serviços, por sua vez, são repassados para o pacote *Persistencia* que interage com o pacote *Mapeamento* e *Mecanismo* para efetivá-lo junto ao banco de dados relacional. Esta dinâmica está demonstrada na Figura 28.

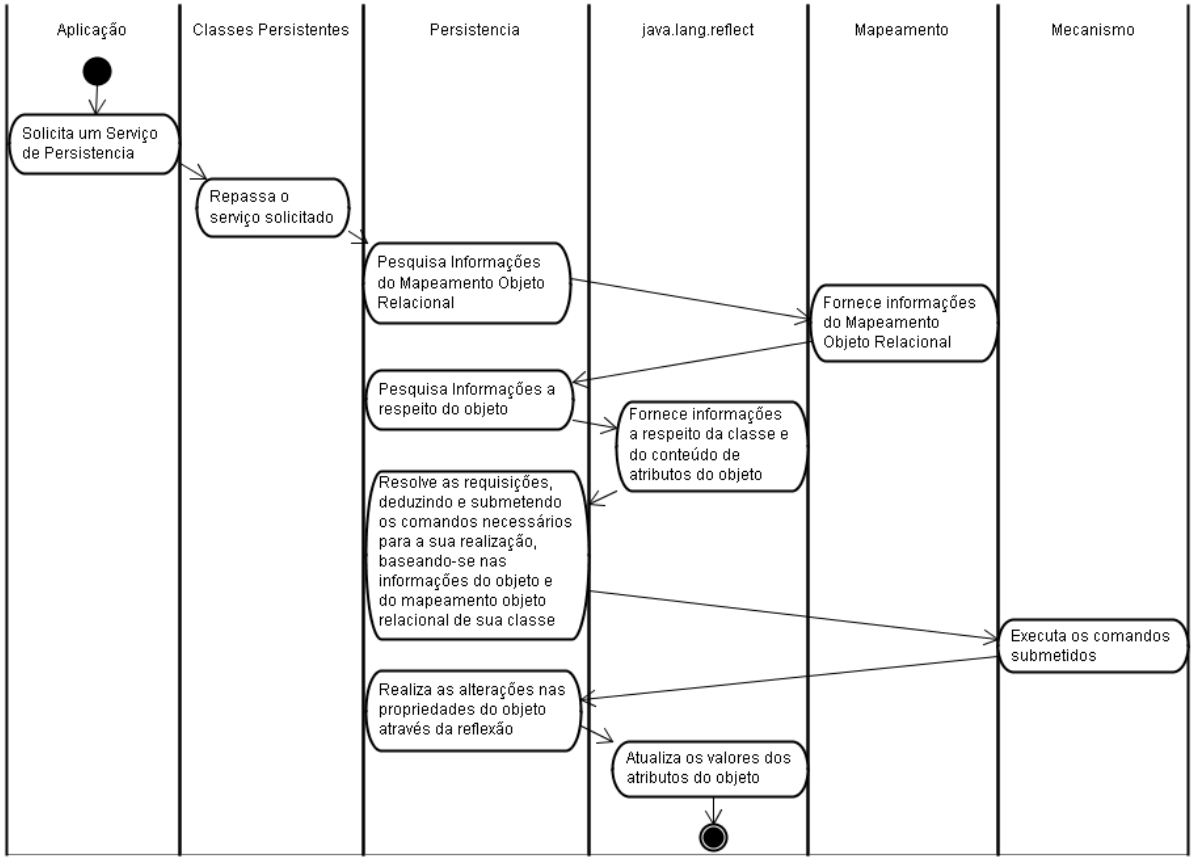


Figura 29 – Funcionamento do FPOR

3.2.1 Serviços de Persistência

O funcionamento do framework começa na aplicação, conforme Figura 30. As classes da aplicação utilizam um conjunto de serviços disponibilizados pelo FPOR, que são invocados a partir das classes persistentes.

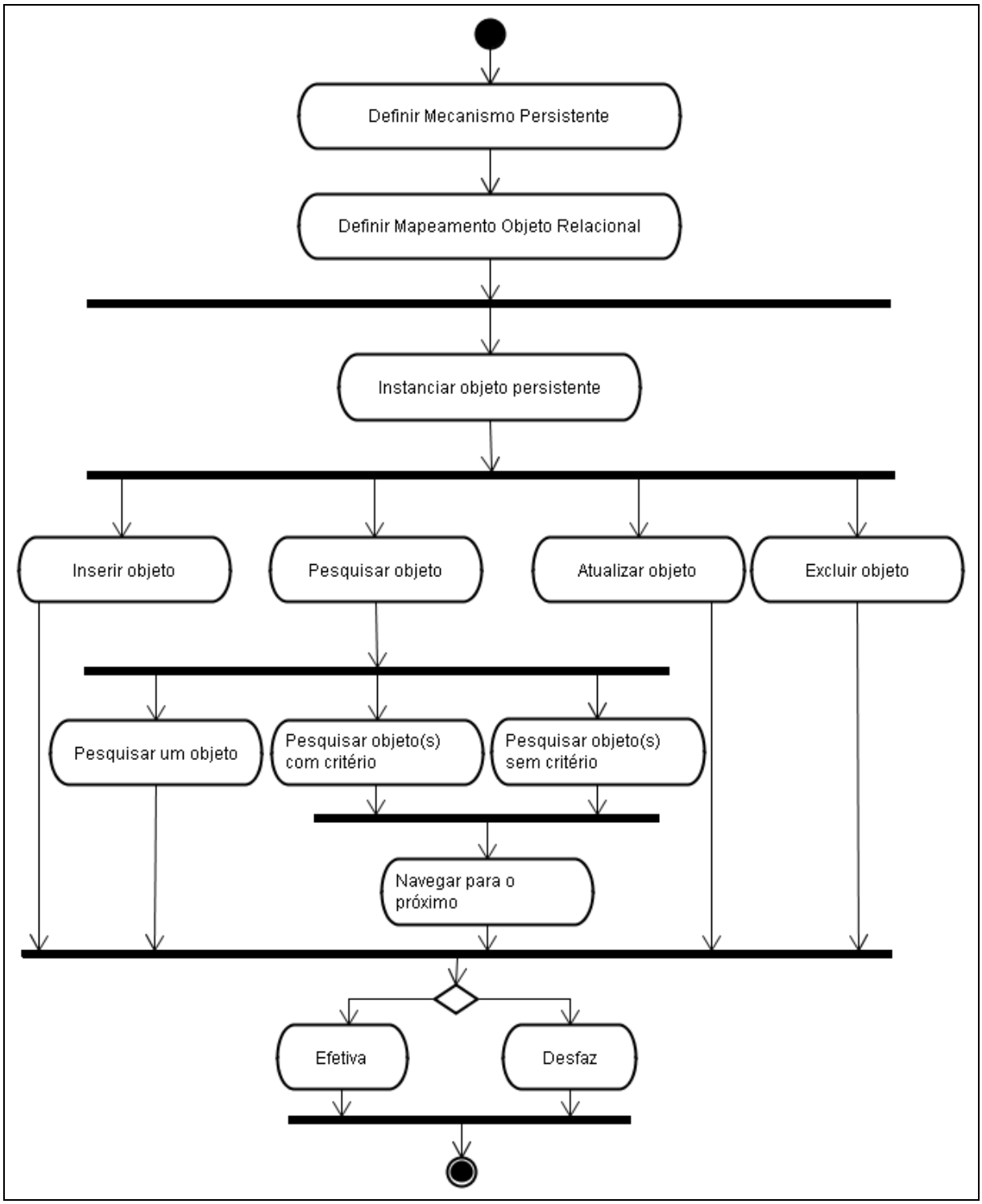


Figura 30 – Serviços Persistentes

Em um primeiro momento, a classe de aplicação deve definir um mecanismo persistente e o mapeamento objeto relacional.

Uma forma de diminuir o acoplamento das classes de aplicação em relação ao FPOR é realizar estas definições em uma única classe, utilizando o padrão Objeto Unitário (LARMAN, 2004). Este padrão prevê a criação de uma classe pública “X” que contém um método estático denominado *obterInstância* que fornece uma única instância de “X”. Como a visibilidade para as classes públicas tem escopo global, em qualquer ponto do código das classes de aplicação pode-se invocar o método *obterInstância* para acessar a instância única do objeto “X” e, então, enviar uma mensagem para obter os objetos correspondentes ao mecanismo persistente e mapeamento objeto relacional definidos para a aplicação.

Definido o mecanismo persistente e o mapeamento objeto relacional, a classe da aplicação pode instanciar o objeto e solicitar os serviços de persistência: inserir, pesquisar, atualizar ou excluir objeto. No serviço de pesquisa, o FPOR disponibiliza três opções: pesquisar um objeto e pesquisar vários objetos com critério opcional. Uma vez feita a pesquisa de mais de um objeto, a navegação por estes objetos é possível através do serviço denominado próximo.

Após a realização de todos os serviços persistentes nos objetos persistentes, a classe de aplicação deve efetivar ou desfazer a transação.

A seguir abordaremos o funcionamento de cada um destes serviços mencionados, através de diagramas de atividade e comentários.

3.2.1.1 Definir Mecanismo Persistente

O processo de definição do mecanismo persistente (Figura 31) consiste em estabelecer um elo de comunicação entre a aplicação e o meio persistente responsável pela guarda e manipulação dos dados dos objetos persistentes. Uma classe de aplicação cria um objeto da classe *MecanismoPersistente*, o qual será utilizado como parâmetro na invocação dos serviços pertinentes disponíveis na camada de negócio.

Uma vez que o mecanismo persistente é definido pela aplicação e utilizado como parâmetro nas operações solicitadas ao FPOR, é possível a utilização de mecanismos de persistência de diferentes fornecedores em uma mesma aplicação, ao mesmo tempo. Com este panorama possibilitamos que objetos sejam acessados e mantidos em diferentes mecanismos de persistência simultaneamente em um cenário onde objetos persistentes de negócio estejam armazenados em diferentes meios de persistência. Neste processo não há a intervenção da classe persistente.

Migrações de fornecedores de mecanismos de persistência podem ser facilitadas com esta possibilidade de funcionamento. Migrar um objeto entre banco de dados de diferentes fornecedores seria simplesmente ler suas instâncias através do mecanismo de persistência do fornecedor origem e gravá-lo em outro mecanismo de persistência do fornecedor destino. Contudo o mapeamento objeto relacional utilizado deve ser o mesmo na manipulação do objeto nos dois mecanismos de persistência.

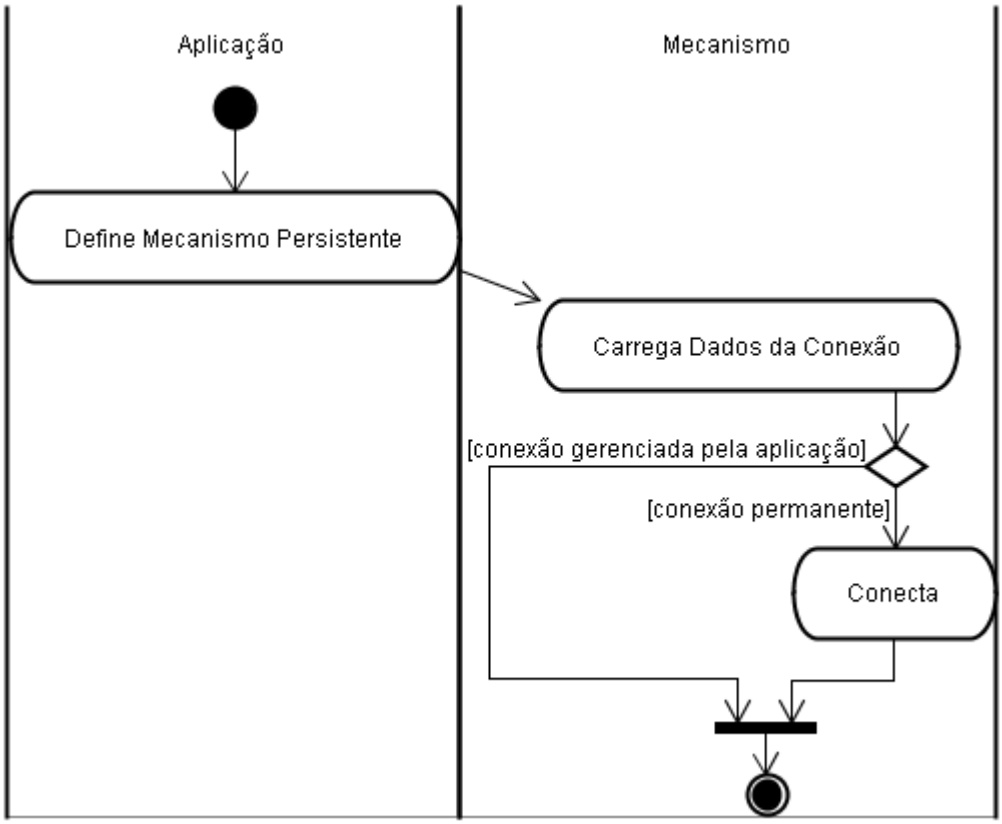


Figura 31 – Definir Mecanismo Persistente

3.2.1.2 Definir Mapeamento Objeto Relacional

Similar à definição do mecanismo persistente, a definição do mapeamento objeto relacional consiste em estabelecer um elo de comunicação entre a aplicação e o repositório do mapeamento objeto relacional localizada em um meio persistente (Figura 32). A classe aplicação define através de métodos estáticos do FPOR qual mapeamento objeto relacional a ser usado, informando o mecanismo persistente onde este se encontra. É possível realizar a alternância de mapeamentos objetos relacionais, bastando a camada de aplicação informar em tempo de execução qual o mapeamento objeto relacional que está em vigência. Porém, cabe ressaltar que este recurso deve ser usado apropriadamente, pois o objeto será mapeado no momento em que for instanciado.

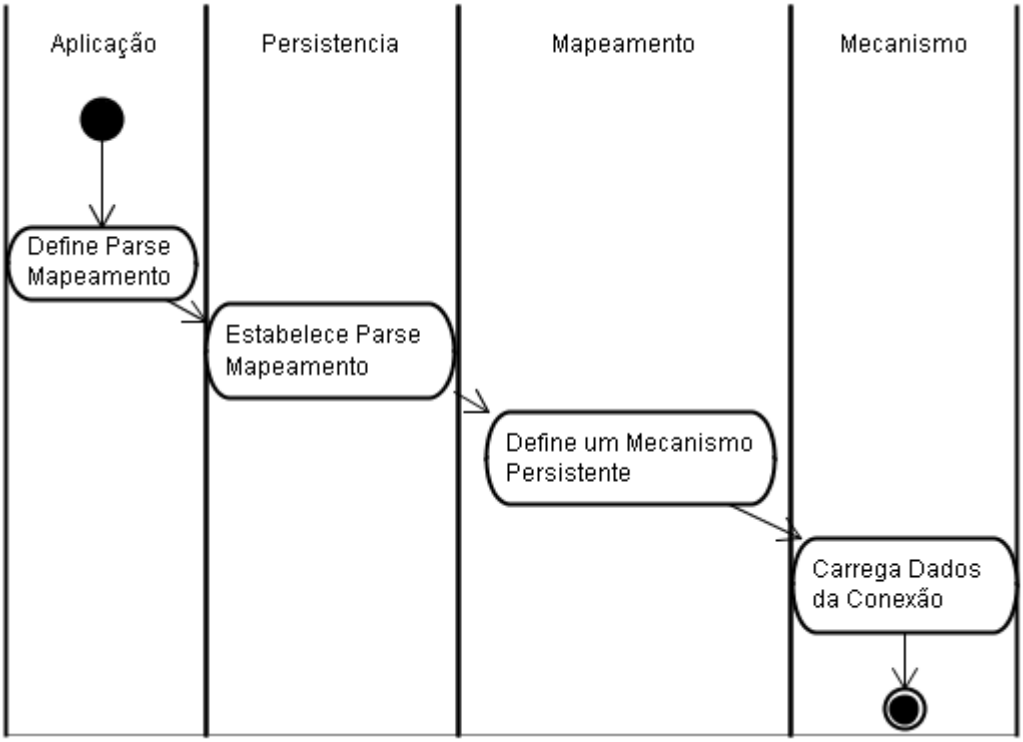


Figura 32 – Definir Mapeamento Objeto Relacional

3.2.1.3 Instanciar Objeto Persistente

Instanciar um objeto persistente é requisito obrigatório para a utilização de suas operações de persistência. No processo de manipulação do bytecode da classe persistente, descrito no item 3.1.1.2, é colocado um atributo denominado *persistencia*. Este atributo representa um objeto da classe Persistencia, sendo inicializado com uma nova instância. A instanciação desta classe prevê a coleta de informações do repositório do mapeamento objeto relacional a respeito da classe persistente que serão utilizadas durante o funcionamento do FPOR.

Inicialmente, a atividade de inicialização do atributo *persistencia* (Figura 33) verifica se a aplicação definiu um mapeamento objeto relacional, pois este elemento é fundamental para o mapeamento da classe e do objeto persistente. A classe do objeto persistente é obtida através do pacote java.lang.reflect para que seja possível a recuperação do seu mapeamento para o FPOR.

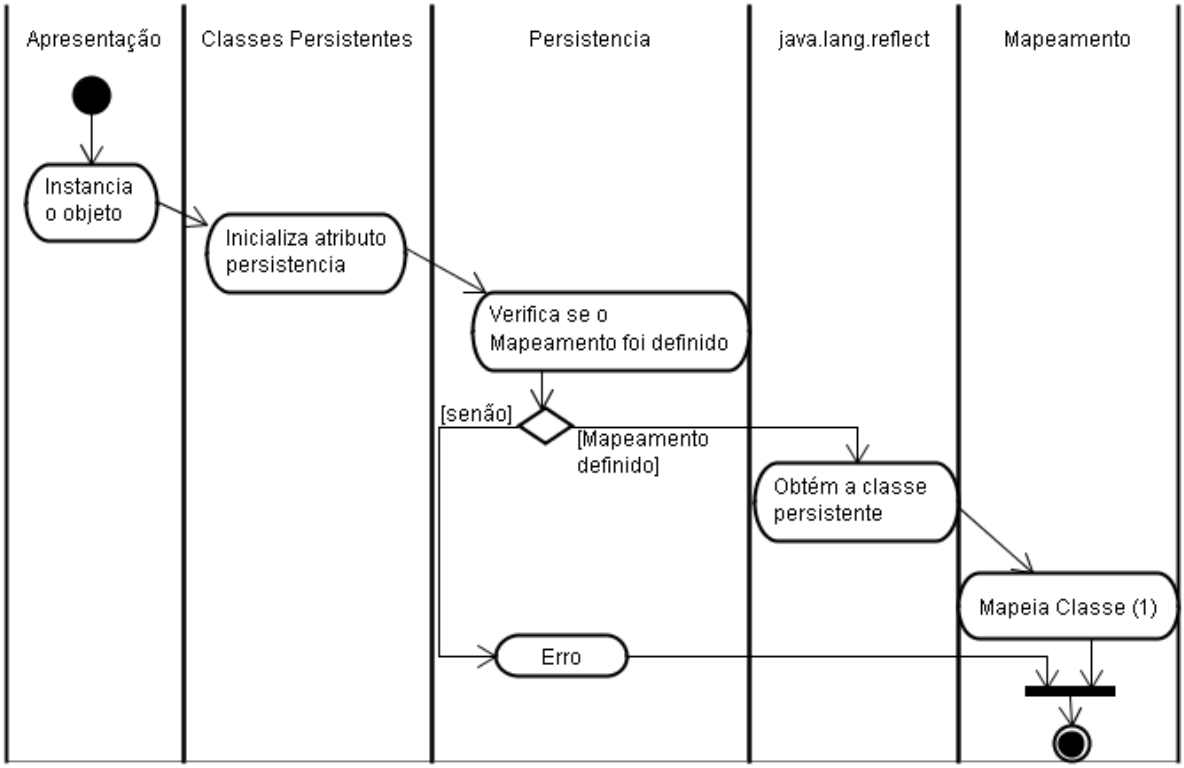


Figura 33 – Instanciar Objeto Persistente

A atividade Mapeia Classe tem por finalidade recuperar o mapeamento objeto relacional previsto para a classe persistente. Seu detalhamento está na Figura 34. Caso a classe persistente ainda não conste no mapeamento objeto relacional, o FPOR realiza o seu cadastramento automático no repositório colhendo informações através da reflexão e emite um erro informando que o mapeamento desta classe está incompleto.

A carga do mapeamento de uma classe persistente para dentro do FPOR já pode ter sido feita em função de outro objeto da mesma classe persistente já ter sido instanciado durante a aplicação. Neste caso, para fins de otimização, a atividade se restringe a somente mapear os valores do novo objeto.

As atividades “mapeia hierarquia” e “mapeia atributos” serão explicadas posteriormente nesta seção.

A atividade “acumula objeto” consiste em carregar o objeto para o modelo do FPOR com a finalidade de controlar o valor de seus atributos durante as operações

persistentes. O termo “acumula” é empregado nesta atividade em função da possibilidade de se ter mais de um objeto instanciado para a mesma classe persistente.

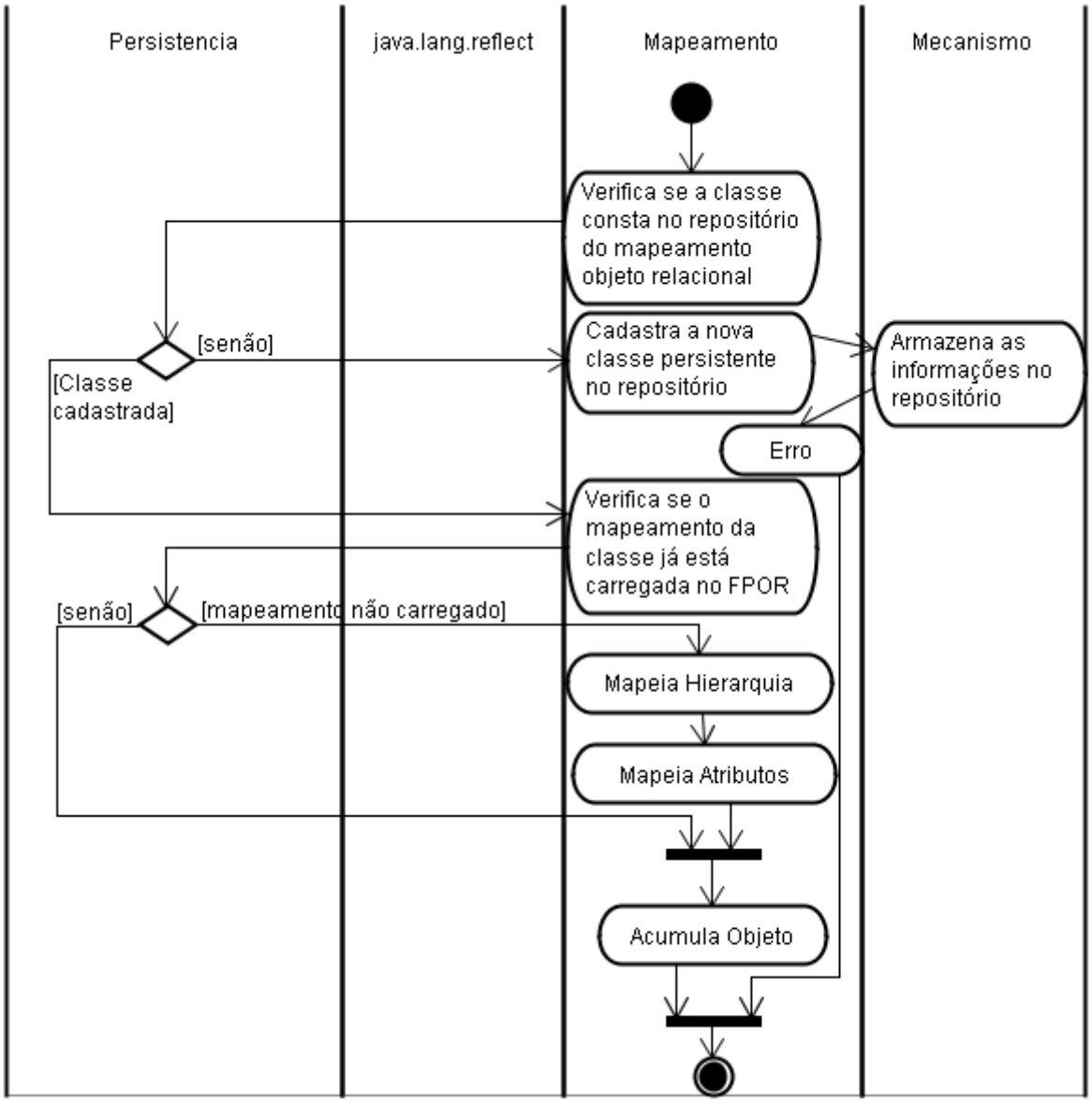


Figura 34 – Atividade Mapeia Classe

No mapeamento de uma classe, quando temos uma estrutura hierárquica de generalização ou especialização, a sua superclasse também deve ser mapeada. Por esta razão a atividade “mapeia hierarquia” tem como objetivo pesquisar as superclasses da classe persistente e efetuar o procedimento de mapeamento para cada super classe encontrada, conforme Figura 35. Podemos notar que a atividade

(1) da Figura 35 corresponde a uma chamada recursiva à atividade mapeia classe, descrito na Figura 34, que ocorre até que se trate uma classe que não tenha super classe, isto é, a classe topo da hierarquia de generalização ou especialização.

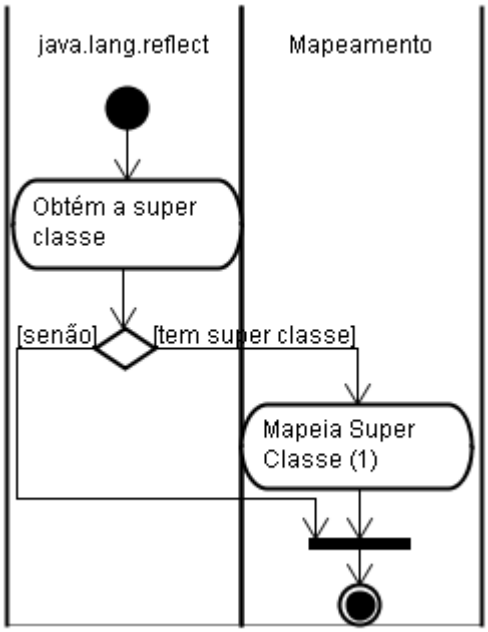


Figura 35 – Atividade Mapeia Hierarquia

A “mapeia atributos”, detalhada na Figura 36, tem como finalidade carregar o mapeamento de todos os atributos da classe persistente no FPOR. Os atributos que compõem a classe persistente são descobertos a partir do pacote java.lang.reflect e o seu mapeamento consta no repositório da ferramenta de mapeamento objeto relacional.

Para cada tipo de atributo, temos diferentes atividades de mapeamento. Para atributos primitivos simples temos a correspondência direta, bem como, para os atributos primitivos multivalorados temos a correspondência referencial. Atributos complexos requerem o mapeamento de outra classe, a qual este atributo se refere. Neste caso, novamente temos uma chamada reflexiva à atividade “mapeia classe”, detalhada na Figura 34, que irá mapear a classe a qual o atributo complexo se refere. Ao final do mapeamento, o FPOR terá em seu modelo todas as classes envolvidas na persistência de uma classe: oriundas de estruturas hierárquicas de generalização ou oriundas de relacionamentos. Diante desta característica,

atentamos para a importância de que todo o modelo de classes esteja documentado no mapeamento objeto relacional.

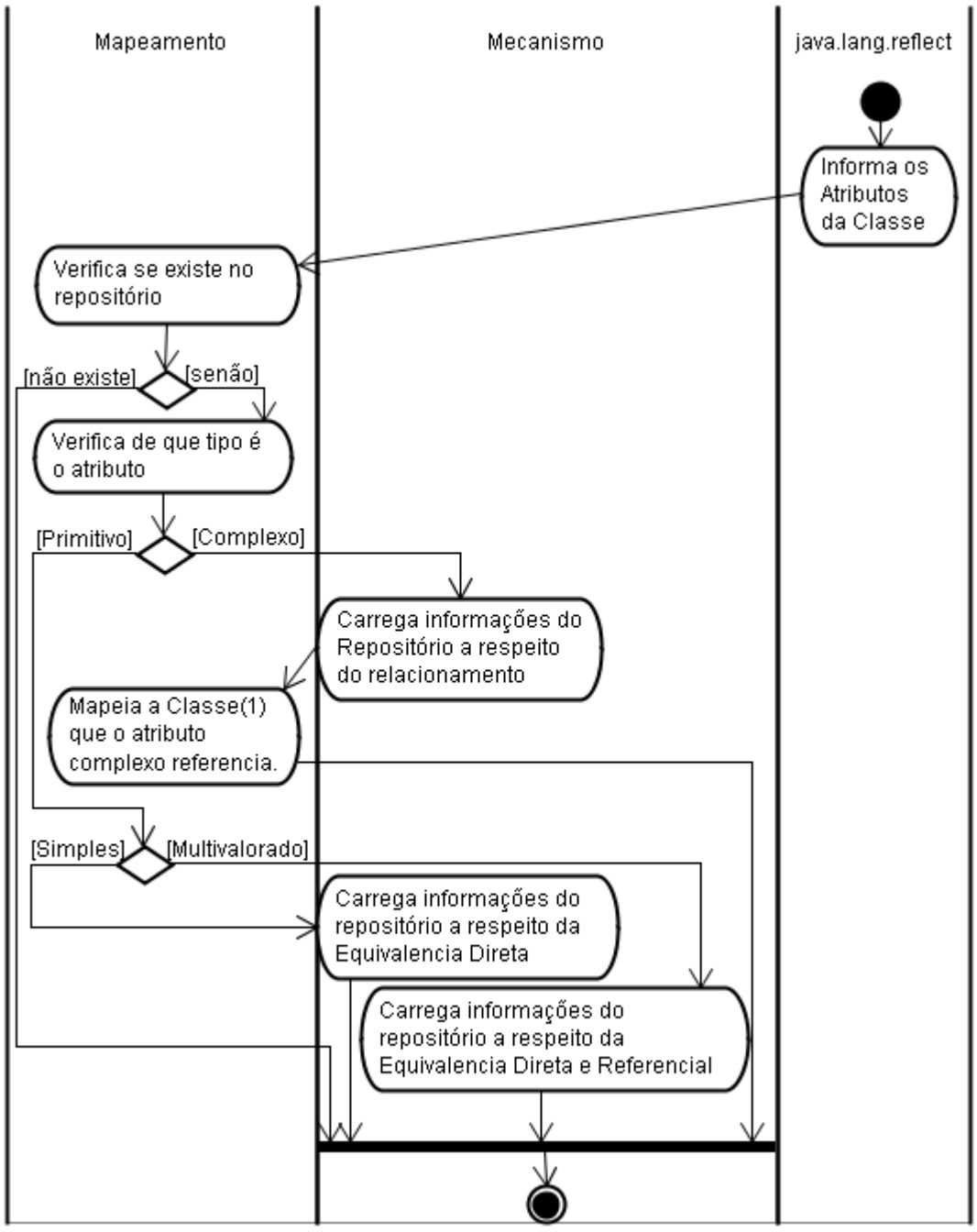


Figura 36 – Mapeia Atributos

Evoluindo o pensamento de que um atributo complexo deve ter as classes a que se refere mapeadas, sugerimos que classes mais utilizadas tenham um padrão de mapeamento, como imagens, sons, gráficos, e outros utilitários. Estes padrões

poderiam ser reutilizados por várias implementações nos diversos sistemas existentes. Estes padrões seriam tanto no mapeamento objeto relacional, bem como, na sua implementação no meio persistente de armazenamento.

3.2.1.4 Inserir Objeto

A primeira ação do FPOR neste processo é verificar se este objeto já está sendo inserido e, caso afirmativo, não realizar qualquer ação. Este controle de estado é necessário em função de relacionamentos mútuos entre objetos, no qual um objeto pode estar referenciando outro objeto, através de seu atributo complexo e vice versa. É fundamental esta precaução, já que, a inserção de um objeto, os objetos adjacentes também podem estar sendo inseridos, dependendo da regra de operação em cascata definida no mapeamento objeto relacional para a classe persistente.

A atividade “inserir objeto”, similar aos outros serviços persistentes, é baseados no objeto e na classe. Um mesmo objeto pode ser inserido sob várias classes, no caso de uma hierarquia de generalização. O procedimento de inclusão primeiro verifica se existe uma super classe e, caso afirmativo, invoca recursivamente o processo “inserir objeto”, informando o objeto e a super classe, sucedendo estas invocações por toda a sua estrutura hierárquica de generalização superior.

Toda invocação recursiva à atividade “inserir objeto” será assinalada com a marcação (1) ao final do nome do processo da Figura 37.

Continuando o procedimento de inclusão, o FPOR pesquisa as características dos atributos primitivos simples, atributos primitivos multivalorados e atributos complexos do objeto persistente. Atributos simples, por sua característica de correspondência direta, já discutida anteriormente, originam simples comandos de inclusão, podendo ser até em tabelas distintas. Atributos primitivos multivalorados dão origem a comandos de inclusão em tabela especificamente destinada ao armazenamento de seus valores. Já os atributos complexos, envolvem três grupos

distintos de objetos: objetos que devem ser incluídos antes, pois são referenciados pelo objeto; objetos inexistentes que devem ser incluídos após a inclusão do objeto, pois referenciam o objeto; e objetos existentes que devem ser atualizados, pois devem referenciar o objeto após a sua inclusão. Além destes objetos, os atributos complexos geram comandos de inclusão de registros em tabelas associativas, que resolvem relacionamentos n para n e tabelas que implementam atributos multivalorados.

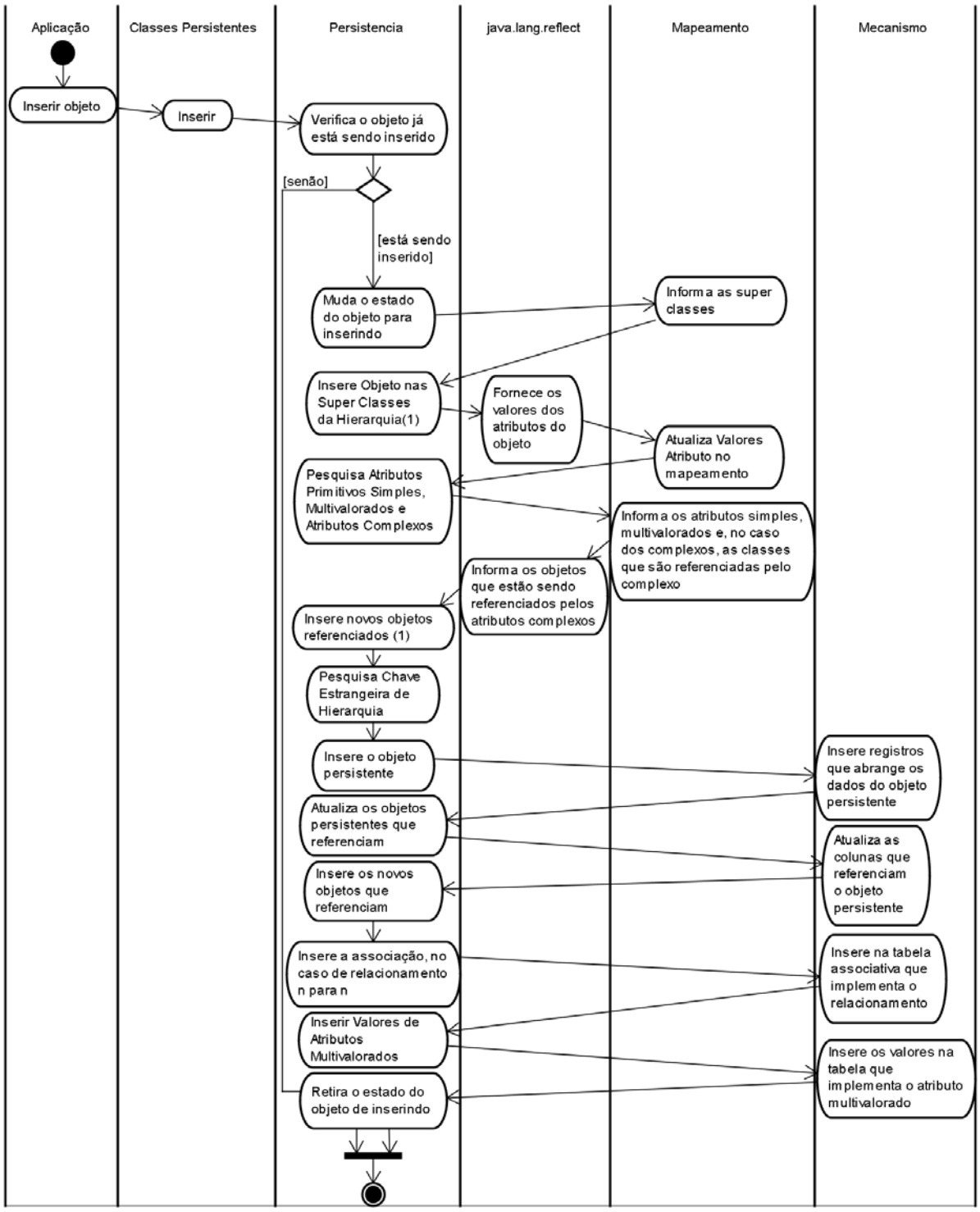


Figura 37 – Inserir Objeto

Quando mencionamos objetos que “referenciam o objeto” e “objetos referenciados” estamos tratando um desencontro entre a tecnologia de orientação a objetos e relacional. Conforme já discutido neste trabalho, a tecnologia orientada a objetos prevê que classes tenham relacionamentos bidirecionais, isto é, as classes

envolvidas têm atributos que referenciam a classe oposta no relacionamento. Estes atributos denominam atributos complexos, conforme já enunciado. A implementação de relacionamentos na tecnologia de banco de dados é feita através de chaves estrangeiras, que são relacionamentos unidirecionais, isto é, somente uma das entidades contém uma coluna que referencia a entidade oposta ao relacionamento. Objetos que “referenciam” são aqueles cuja representação no banco de dados contém a coluna que referencia a entidade oposta ao relacionamento. Objetos “referenciados” pelo objeto são quando a sua representação no banco de dados contém a coluna que referencia a entidade oposta ao relacionamento. Esta classificação é importante para ordenar as ações a serem realizadas nos serviços de persistência.

No exemplo da Figura 38, a classe *PessoaFisica* contém um relacionamento com a classe *CNH*. Este relacionamento é implementado através de um atributo complexo implementado na classe *PessoaFisica* que referencia um objeto da classe *CNH* e, inversamente, a classe *CNH* contém um atributo complexo que referencia um objeto da classe *PessoaFisica*. Dizemos que a classe *CNH* referencia a classe *Pessoa Física* em virtude da chave estrangeira, equivalente ao relacionamento entre estas classes, ser implementada na tabela que implementa a classe *CNH*. A tabela CNH contém a coluna PESSOA_FISICA_ID que referencia a chave primária da tabela PESSOA_FISICA, que implementa a classe *PessoaFisica*. Portanto denominamos que a coluna CNH.PESSOA_FISICA_ID é a coluna da chave estrangeira e a coluna PESSOA_FISICA.ID é a coluna estrangeira.

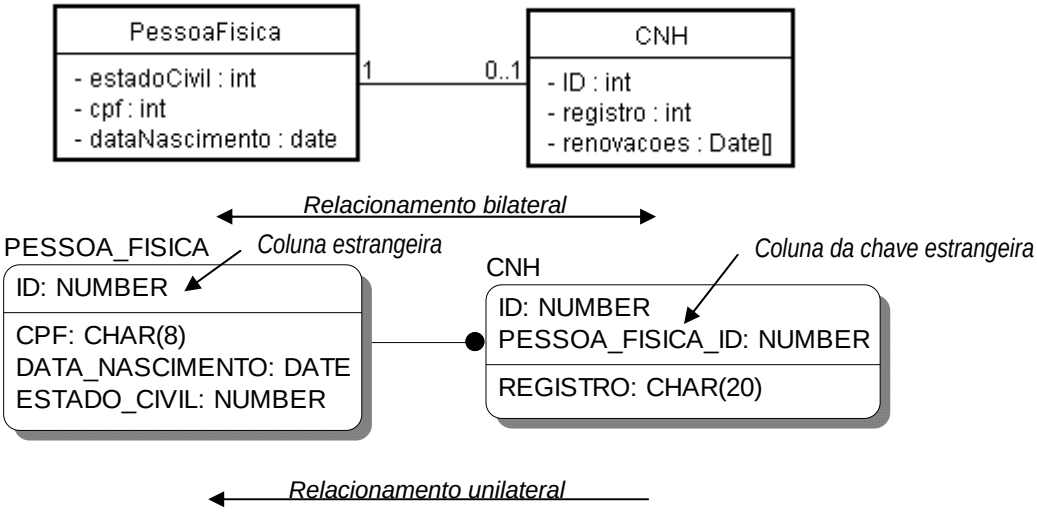


Figura 38 – Relacionamentos Bilaterais

A quantidade de tabelas e colunas que devem ser atualizadas em uma operação persistente varia em função do tipo e da cardinalidade do relacionamento entre objetos. Lembramos que para cada relacionamento é associado uma ou várias chaves estrangeiras, nas quais constam as colunas que são o elo de ligação entre as tabelas envolvidas no relacionamento. Estas colunas “elo” são utilizadas na formulação dos comandos SQL de atualização e recuperação junto ao banco de dados.

No contexto acima, a coluna estrangeira pode não corresponder a um atributo primitivo da classe que está na outra extremidade do relacionamento. Neste caso, esta coluna estrangeira é a chave primária da tabela correspondente à classe persistente e, ao mesmo tempo, a chave estrangeira que implementa o relacionamento entre a classe persistente e uma classe superior em uma estrutura hierárquica de generalização. O FPOR, quando ocorrer tal fato, verifica se a classe persistente tem uma super classe e, caso afirmativo, percorre todas as suas classes superiores recursivamente, até encontrar o atributo desejado. A Figura 39 exemplifica esta situação.

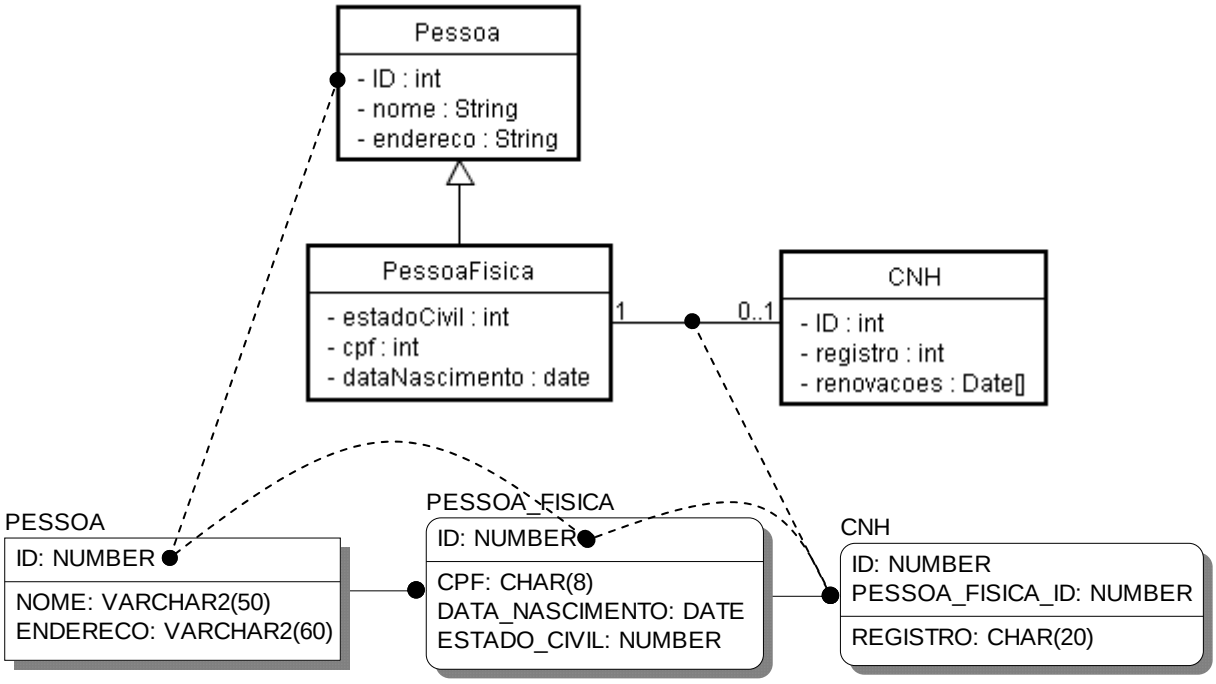


Figura 39 – Chave Estrangeira

3.2.1.5 Atualizar Objeto

O principal conceito no processo de atualização, similar aos demais processos de persistência, é tratar, além do objeto a ser atualizado, entidades relacionadas ao objeto.

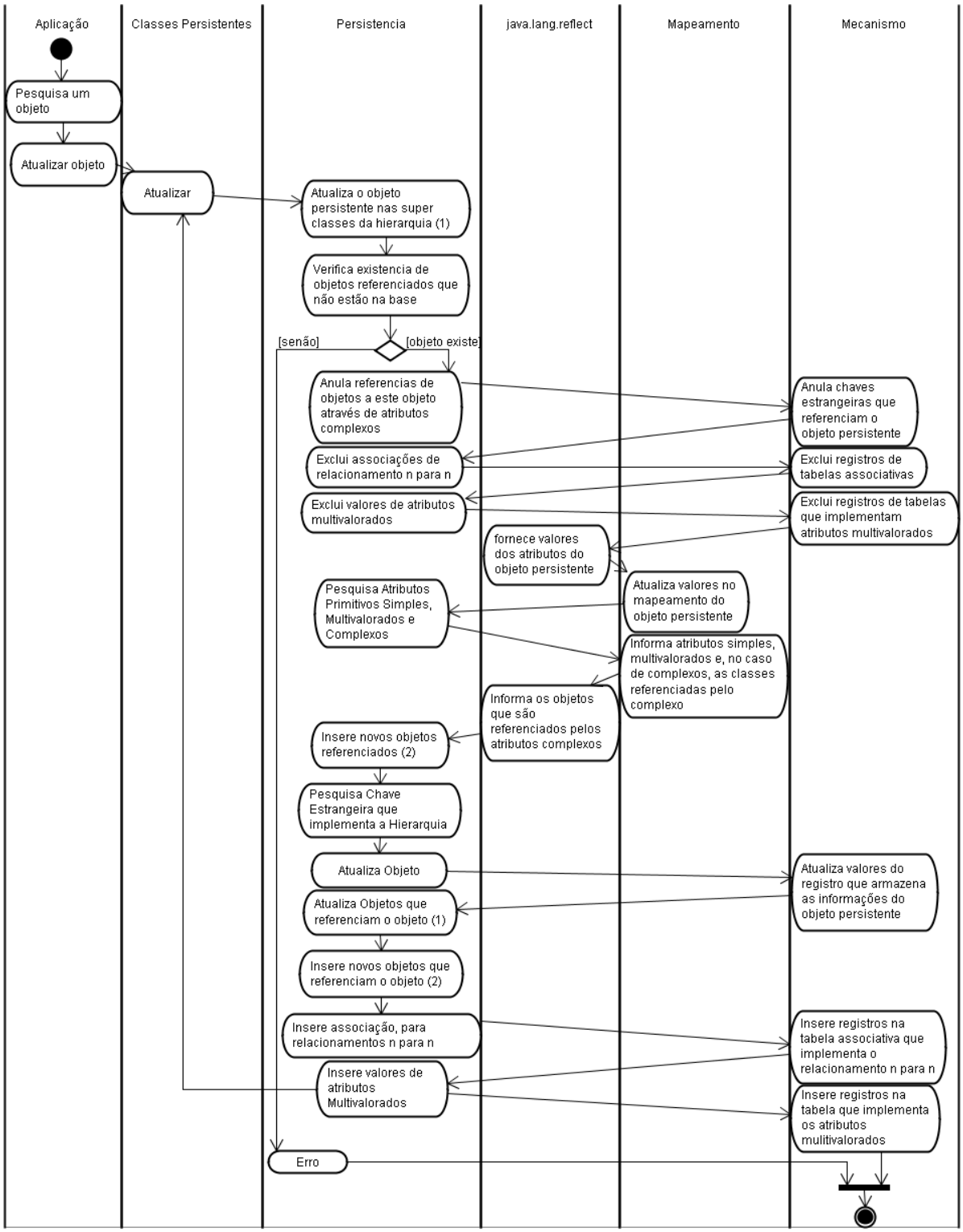


Figura 40 – Atualizar Objeto

O primeiro passo é atualizar recursivamente todas as classes superiores, caso a classe do objeto esteja em uma hierarquia de generalização.

As ações subseqüentes são dividas em duas fases: o tratamento de entidades periféricas antes da atualização do objeto e tratamento de entidades periféricas após a atualização do objeto.

Na primeira fase o FPOR mantém inalterados os dados recuperados, referentes ao objeto persistente, para realizar pesquisas a respeito de atributos multivalorados e atributos complexos. No caso de objetos referenciados por atributos complexos, o FPOR verifica a inexistência de algum destes objetos no meio persistente. Caso se confirme alguma inexistência, fica caracterizado que o meio persistente foi alterado por outro processo durante a edição do objeto, provocando erro de colisão, cujo conceito está descrito no capítulo 2. Os objetos que referenciam o objeto têm sua referência anulada temporariamente. Registros referentes a atributos multivalorados e registros referentes a relacionamentos n para n do objeto são temporariamente excluídos.

Na segunda fase o FPOR é atualizado com os valores correntes do objeto persistente. Realizamos novamente a pesquisa a respeito de atributos multivalorados e atributos complexos. Seguindo uma ordem coesa, o FPOR insere novos objetos referenciados, atualiza o objeto, insere objetos que referenciam o objeto, insere nas tabelas de relacionamento n para n e finalmente insere nas tabelas de atributos multivalorados.

Na Figura 40, processos com a marcação (1) invocam recursivamente o processo “atualiza objeto”. Já os processos com a marcação (2) utilizam o processo “inclui objeto”, já descrito nesta seção.

Na Figura 40 detalhamos a atividade “atualiza objeto”. Esta figura demonstra o controle de concorrência otimista (AMBLER,2003a) implementada no FPOR. Conforme já mencionado neste trabalho, o controle de concorrência otimista simplesmente acusa eventuais inconsistências que podem ocorrer durante a atualização de registros na base. A atividade realiza os seguintes passos para cada registro a ser atualizado: (i) bloqueia o registro; (ii) verifica se existe no meio persistente; (iii) verifica se está igual, isto é, verifica se algum outro processo alterou o registro; e (iv) efetiva a atualização do registro. O registro permanece bloqueado até que o programador efetive ou desfaça a transação.

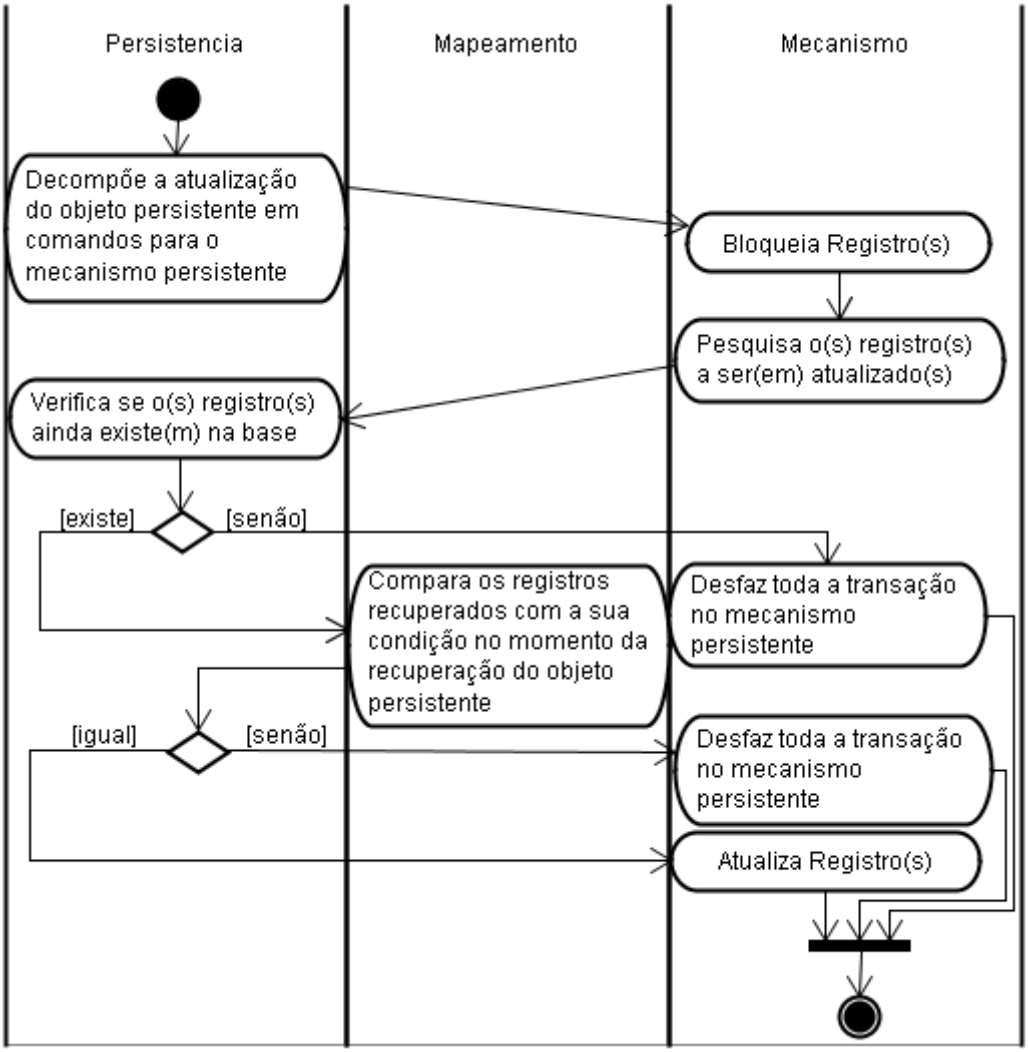


Figura 41 – Atividade Atualiza Objeto

Mesmo que relacionados, objetos persistentes devem ser tratados isoladamente. Um objeto A está sendo editado e está relacionado a um objeto B. Se o objeto B for modificado, isto é, o valor de suas propriedades for alterado, o programador deve solicitar um serviço de atualização para o objeto B. O fato de se atualizar o objeto A implica que somente a referência do objeto B seja atualizada pelo FPOR.

3.2.1.6 Excluir Objeto

Similar ao 3.2.1.4, a primeira ação do FPOR neste processo é verificar se este objeto já está sendo excluído para evitar, nos casos de relacionamentos bilaterais, a exclusão de um objeto que já está sendo excluído.

O passo seguinte é verificar se algum objeto referenciado não existe mais na base, o que caracteriza que a base foi alterada durante a edição do objeto e, portanto deve ser indicado erro de colisão. As demais ações se referem a entidades periféricas ao objeto que devem ser excluídas e anuladas, de acordo com a regra definida no mapeamento objeto relacional.

Esta atividade também trata os atributos multivalorados, excluindo os registros da tabela que armazena seus valores.

O passo “Excluir Objeto” do FPOR mostrado na Figura 42 é similar à atividade atualiza objeto da Figura 41, pois também implementa o controle de concorrência otimista (AMBLER,2003a), verificando se o objeto foi alterado na base de dados por outro processo durante a sua edição.

A marcação (1) indica que o processo utiliza uma chamada recursiva ao processo “Excluir Objeto” para realizar a sua tarefa como, por exemplo, o processo “Excluir Hierarquia”, no qual se exclui o objeto nas classes superiores em uma estrutura hierárquica de generalização.

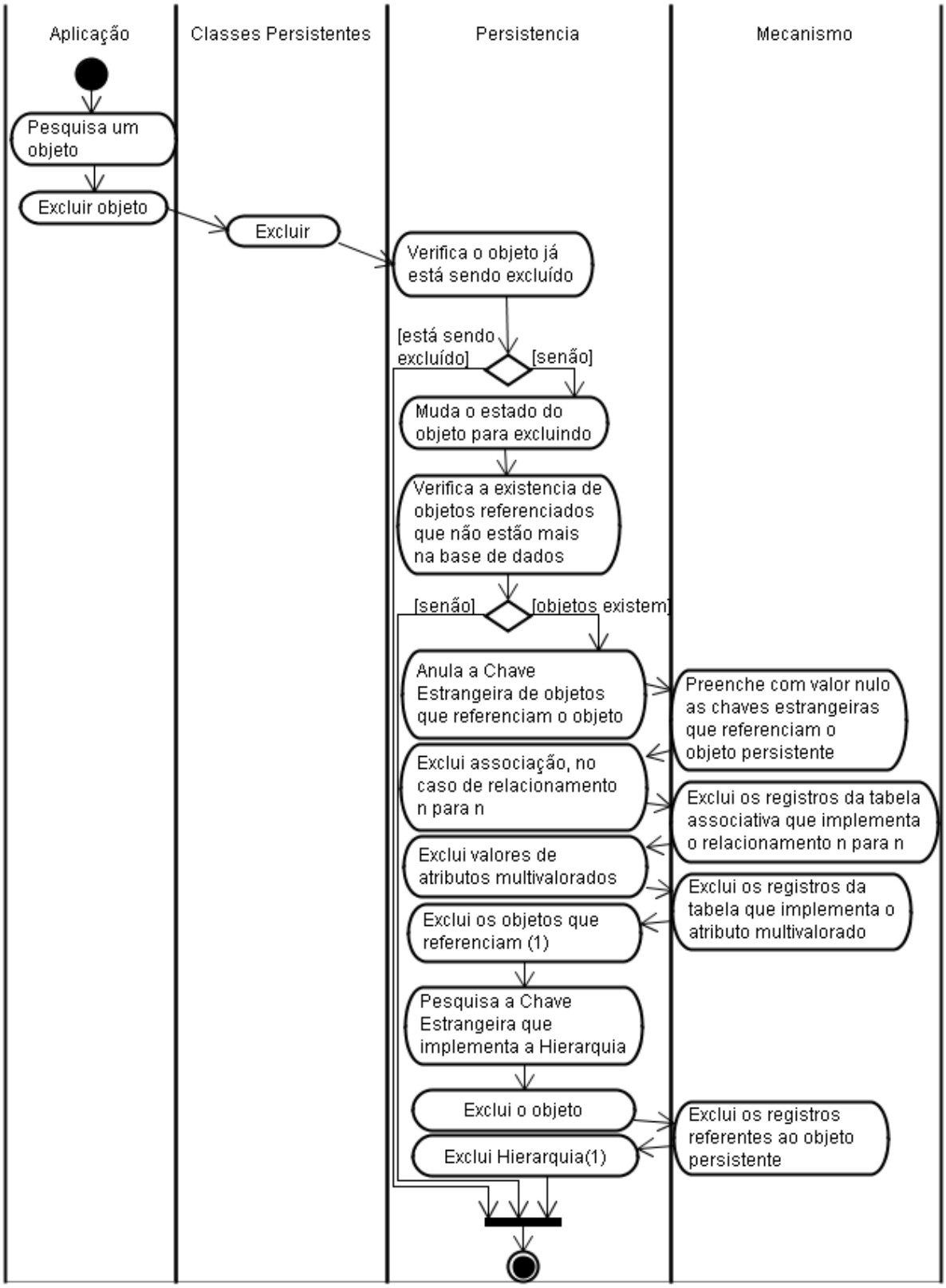


Figura 42 – Excluir Objeto

3.2.1.7 Pesquisar um Objeto

A classe de aplicação que utiliza um objeto persistente deve criar um critério para efetuar a pesquisa de um objeto. Um critério é composto por uma ou mais cláusulas que serão traduzidas e submetidas pelo framework no momento em que acontecer a busca de um objeto junto ao meio persistente. O critério é parâmetro obrigatório na invocação do serviço “pesquisa um”.

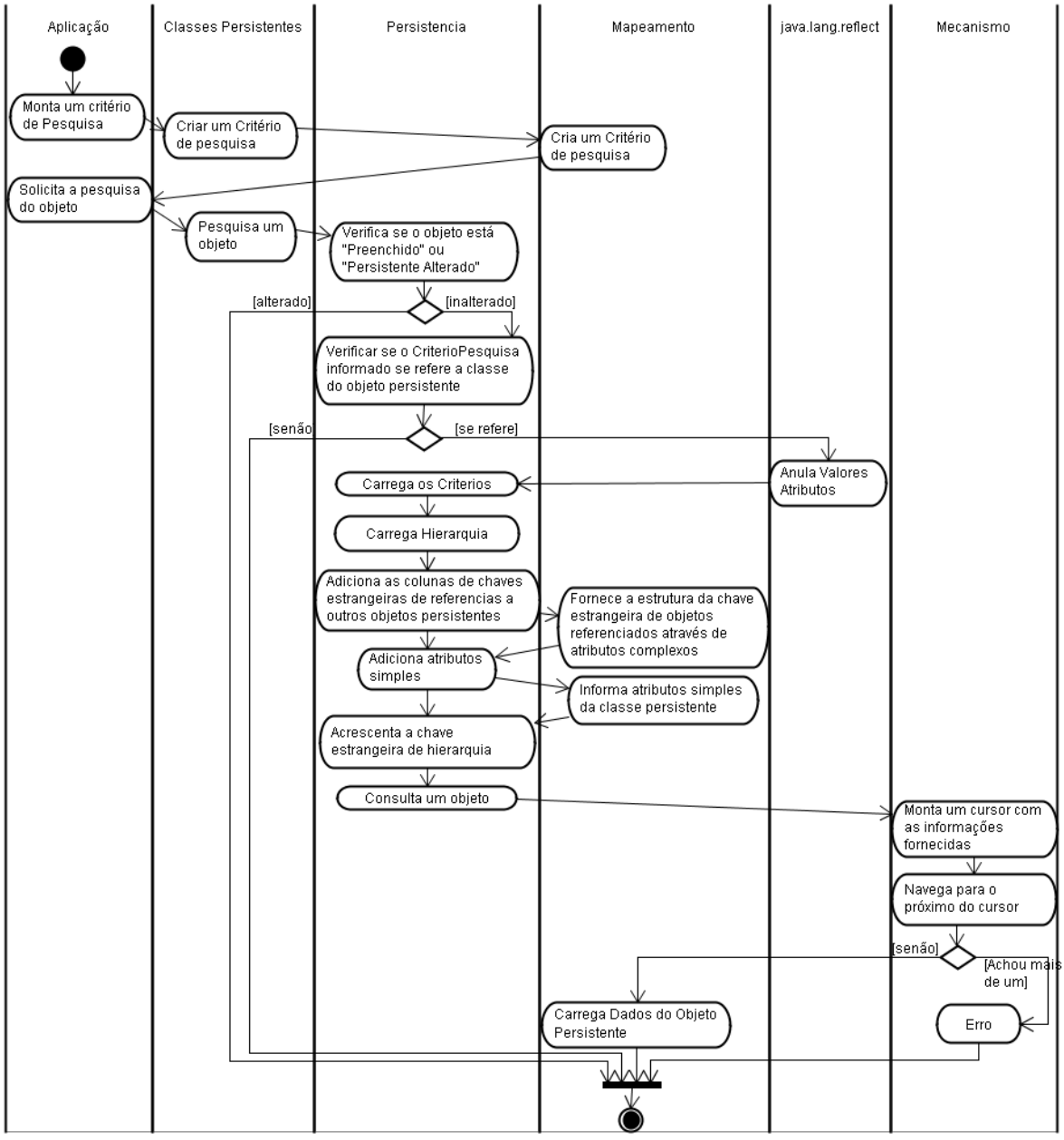


Figura 43 – Pesquisar um Objeto

Inicialmente o FPOR verifica se o objeto não está sujo ou foi alterado e, caso afirmativo, emite um erro informando que não é possível realizar uma pesquisa enquanto houver alterações pendentes de efetivação junto à base de dados.

Em seguida, os valores do objeto são anulados para que o objeto esteja vazio caso não exista nenhum objeto que satisfaça o critério definido.

A atividade “Carrega Hierarquia”, detalhada na Figura 44, tem por objetivo carregar as informações a respeito das classes que estejam em níveis superiores da estrutura hierárquica de generalização da classe persistente. Estas informações serão utilizadas na montagem dos comandos de recuperação do objeto persistente. O processo “Carrega hierarquia da super classe” da Figura 44 é uma chamada recursiva ao procedimento “Carrega Hierarquia”, pois este processo deve contemplar todas as classes superiores à classe persistente.

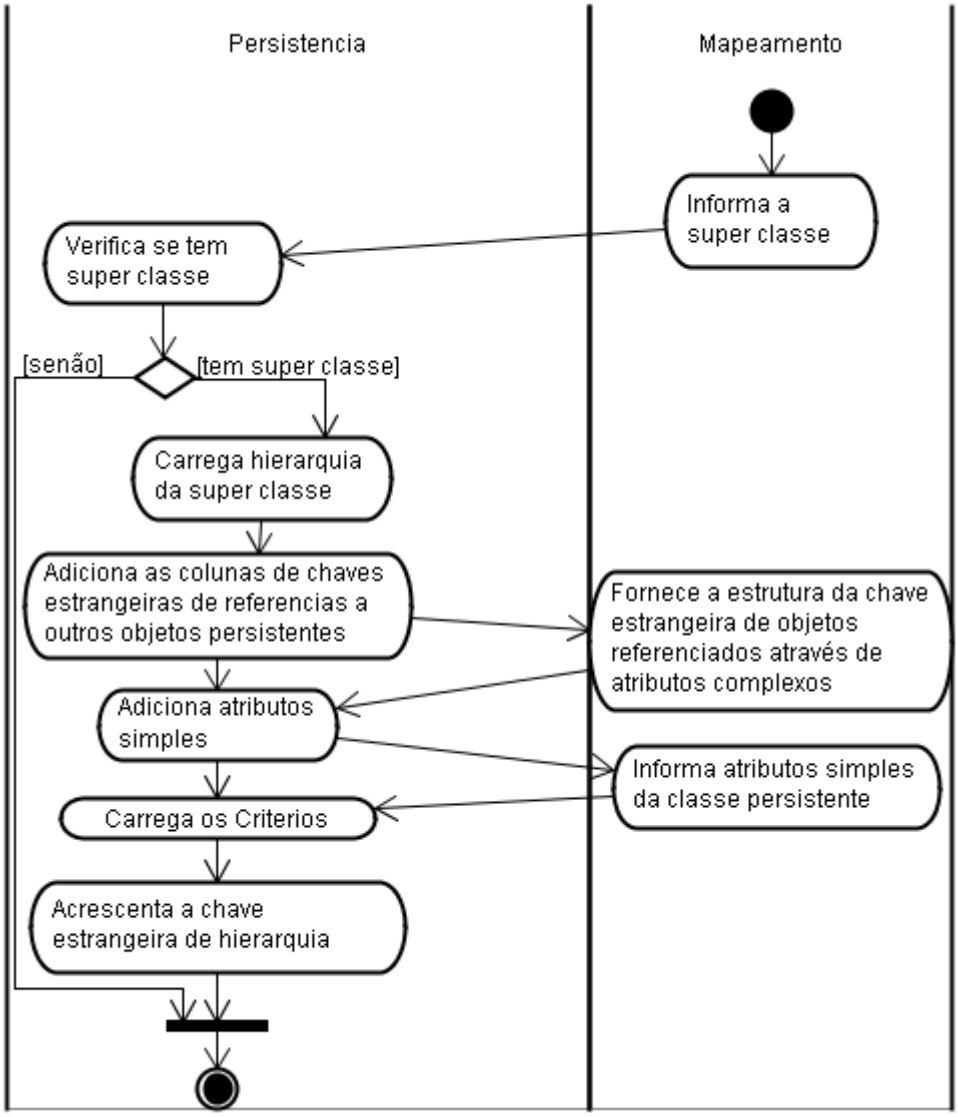


Figura 44 – Procedimento Carrega Hierarquia

Seguindo o fluxo da Figura 43, a instrução de busca dos dados desejados é montada baseada nos critérios definidos, informações da hierarquia de generalização ou especialização, atributos da classe persistente e referências a outros objetos persistentes. Quando a consulta é submetida, o mecanismo persistente monta um cursor com os dados fornecidos e navega para o próximo registro deste cursor. Caso tenha sucesso na recuperação do registro, o pacote mecanismo verifica se foi recuperado mais de um registro.

Sucedida a pesquisa, o FPOR atualiza os atributos primitivos simples no mapeamento objeto relacional com os valores recuperados pela pesquisa. Estes

valores são mantidos dentro do FPOR para a realização de serviços e controles de estado do objeto persistente.

Objetos referenciados por atributos complexos do objeto recuperado em uma primeira fase não são recuperados. O framework em um primeiro momento se restringe a somente guardar o cursor de pesquisa. O valor do atributo complexo permanece nulo até que o método “get” pertinente a este atributo seja acionado pela camada de apresentação ou negócio. Esta implementação incorpora ao FPOR a técnica de recuperação sob demanda, já discutida neste trabalho (AMBLER,2000b).

O objeto persistente finalmente é atualizado através da reflexão com os valores da pesquisa, porém somente seus atributos primitivos. A atualização dos atributos complexos e multivalorados através da técnica de recuperação sob demanda será detalhada na seção 3.2.2.

A identificação dos dados de um objeto no meio persistente é baseada nas chaves primárias do banco de dados. Esta identificação pode ser direta, através de um atributo primitivo simples, ou por referência, por intermédio de um atributo multivalorado ou complexo. A montagem de pesquisa por chave primária, no último caso mencionado, é realizada pelo FPOR através da pesquisa, na classe *ChaveEstrangeira* associada a *Referencia* do atributo complexo ou multivalorado, do *MapeamentoColuna* que seja chave primária do *MapeamentoObjetoBD* desta classe *ChaveEstrangeira*.

Ao invés de realizar instruções de joins, o FPOR soluciona a pesquisa de várias tabelas através de instruções isoladas, mas com critérios que direcionam o resultado para o contexto do objeto principal. A ordem de recuperação adotada segue a seguinte seqüência: o objeto principal e, em seguida, objetos periféricos, isto é, objetos referentes a atributos complexos, sob demanda. Os critérios de pesquisa utilizados na recuperação de objetos periféricos são os atributos primitivos que identificam o objeto principal e, ao mesmo tempo, fazem parte da chave estrangeira do objeto periférico, dispensando joins entre tabelas.

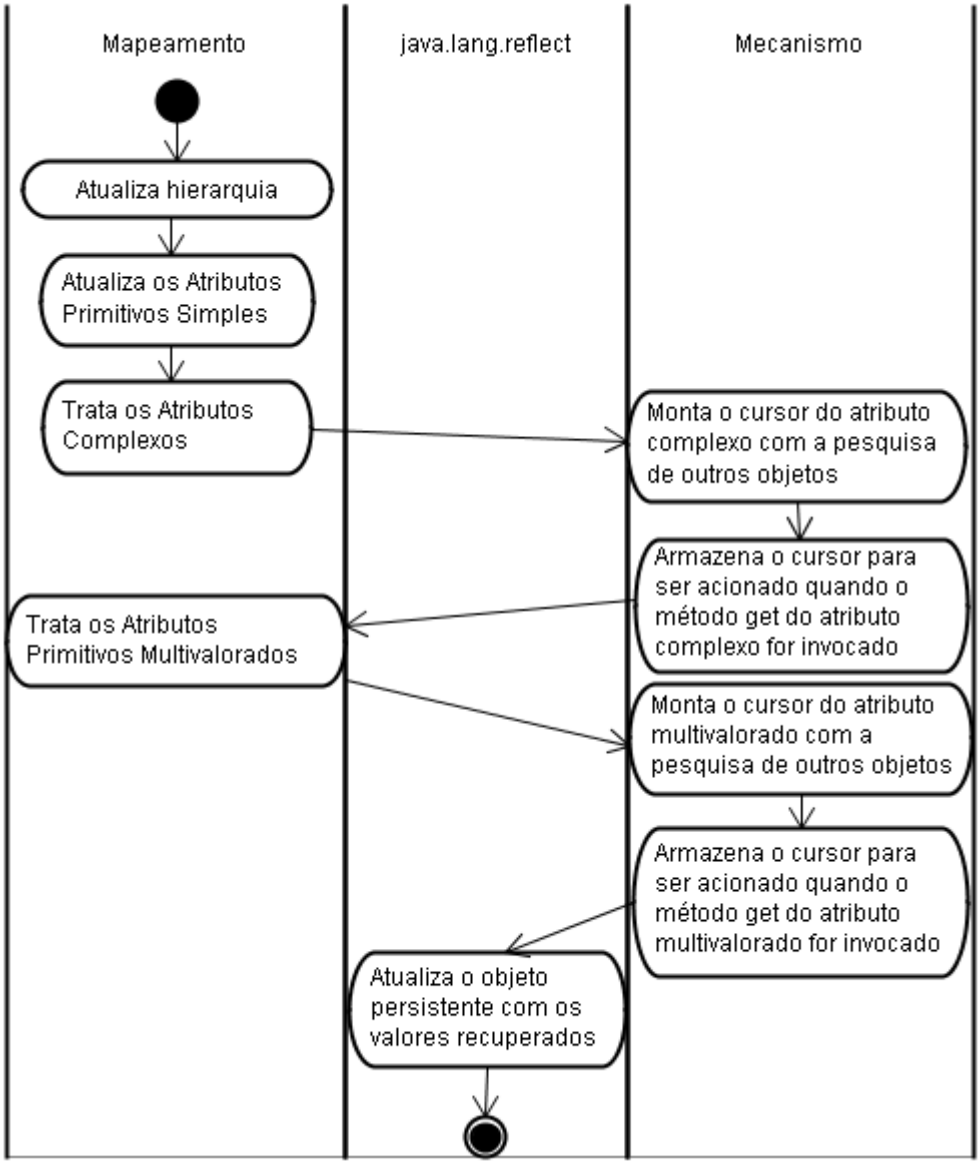


Figura 45 – Procedimento Carrega Dados do Objeto Persistente

Similar aos demais serviços persistentes, o procedimento de pesquisa é baseado no objeto e na classe. Portanto, quando se atualiza atributo de superclasse do objeto persistente, o FPOR invoca recursivamente o procedimento da Figura 45, passando o objeto e a superclasse, conforme Figura 46.

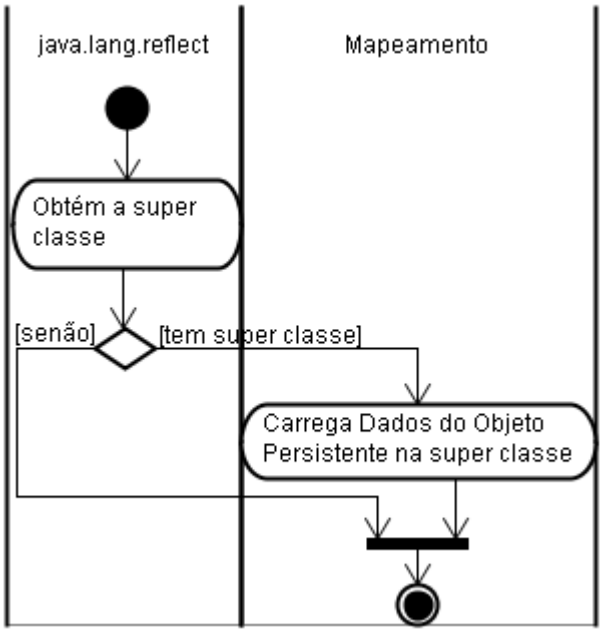


Figura 46 – Procedimento Atualiza Hierarquia

3.2.1.8 Pesquisar Objeto(s) com Critério ou sem Critério

Pesquisa de mais de um objeto se constitui de um subconjunto de ações do serviço de pesquisa de um objeto, como podemos observar na Figura 47, as principais diferenças em relação à pesquisa de um objeto são:

i. Possui navegação, pois se trata de uma pesquisa cujo resultado é uma coleção de objetos.

ii. o critério de pesquisa é opcional. Uma pesquisa sem critério de pesquisa recupera todos os objetos persistentes de uma determinada classe

iii. o objeto persistente, a princípio, fica vazio quando a consulta é feita, permanecendo neste estado até que haja uma navegação nesta pesquisa. A pesquisa se restringe a montar o cursor e mantê-lo no FPOR para futuras navegações.

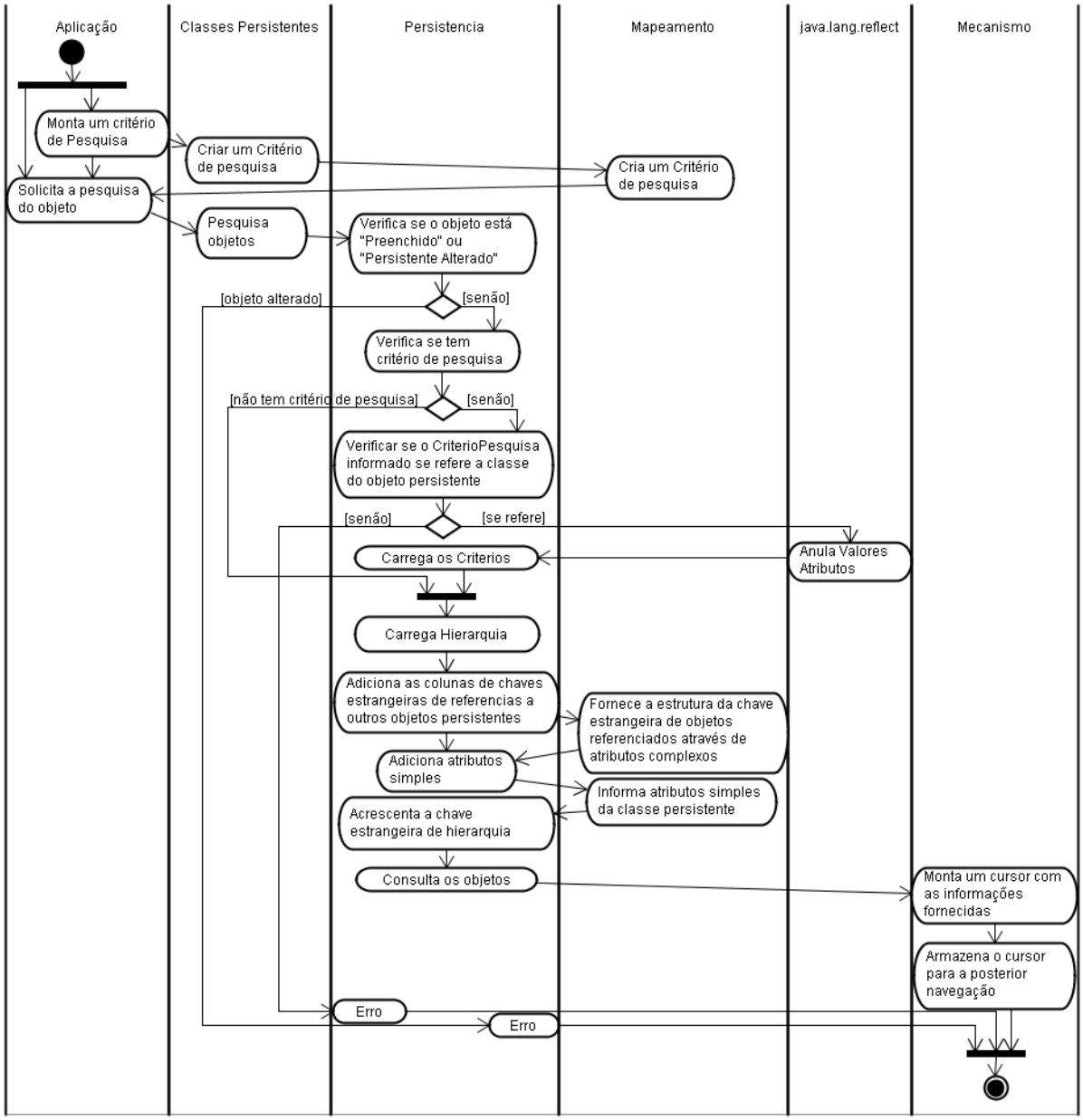


Figura 47 – Pesquisar Objetos

3.2.1.9 Navegar para o próximo

Esta operação só é valida para pesquisas de vários objetos. Cada invocação deste serviço representa uma navegação no cursor montado especificamente para a pesquisa. Seu funcionamento é similar aos procedimentos finais do serviço de pesquisa de um objeto, conforme Figura 48. O procedimento “Carrega Dados do Objeto Persistente” já foi detalhado na Figura 45.

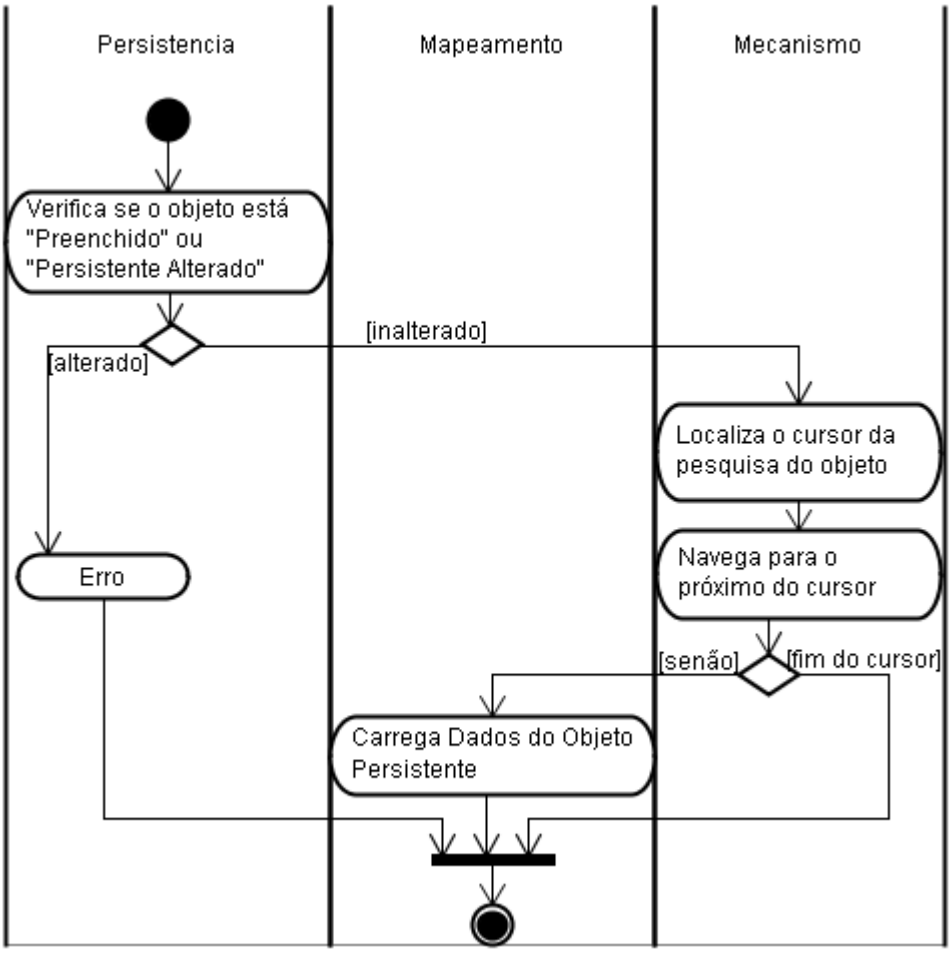


Figura 48 – Navegar para o próximo

O FPOR finaliza esta operação atualizando o objeto com os valores recuperados através da reflexão, conforme já descrito no item Figura 45.

3.2.1.10 Efetiva ou Desfaz

Os serviços de efetivar ou desfazer correspondem aos comandos de commit ou rollback no mecanismo persistente. No ambiente de banco de dados, uma transação só é finalizada quando o comando *commit* é acionado para efetivar ou o comando *rollback* é acionado para desfazer todas as ações da transação. Estes comandos são também importantes na operação de atualização de um objeto. Conforme já visto, a operação de atualização bloqueia os registros envolvidos no

seu processamento a fim de evitar colisões (AMBLER,2003a). A liberação destes registros só é feita através dos comandos de efetivação ou anulação.

3.2.2 Recuperação sob Demanda

Leituras sob demanda otimizam o desempenho na recuperação de objetos persistentes. Todos os acessos diretos a atributos complexos, sem a utilização do método “get” da respectiva propriedade, retornam *null*, pois estes não são carregados na recuperação do objeto persistente. Somente após a primeira referência ao atributo através de seu método “get”, o atributo complexo é preenchido.

No processo de transformação de classes persistentes, a ser descrito na seção 3.1.1.2, o *CompiladorJavassist* implementa uma chamada a um método de auditoria da classe *Persistencia* no início de cada método “get” da classe persistente. Quando o método de auditoria é acionado, o FPOR descobre o objeto e o atributo do método “get” que o acionou através da reflexão. Se o atributo for complexo ou multivalorado, localiza o correspondente cursor, montado no momento em que o objeto foi recuperado da base. Localizado o cursor e dependendo do tipo de atributo, efetua a recuperação de todos os objetos relacionados ao atributo complexo, ou todos os valores contidos no atributo multivalorado e, ao mesmo tempo, atualiza o objeto com estes novos elementos utilizando a reflexão (DEVEGILI,2000). No caso de coleção, todos os objetos da coleção que estejam associados ao objeto são recuperados. Com esses procedimentos, o atributo estará pronto para manipulação no ambiente de negócio. Caso o método “get” seja acionado novamente, o FPOR não efetivará outra vez este procedimento se o atributo já tiver sido carregado.

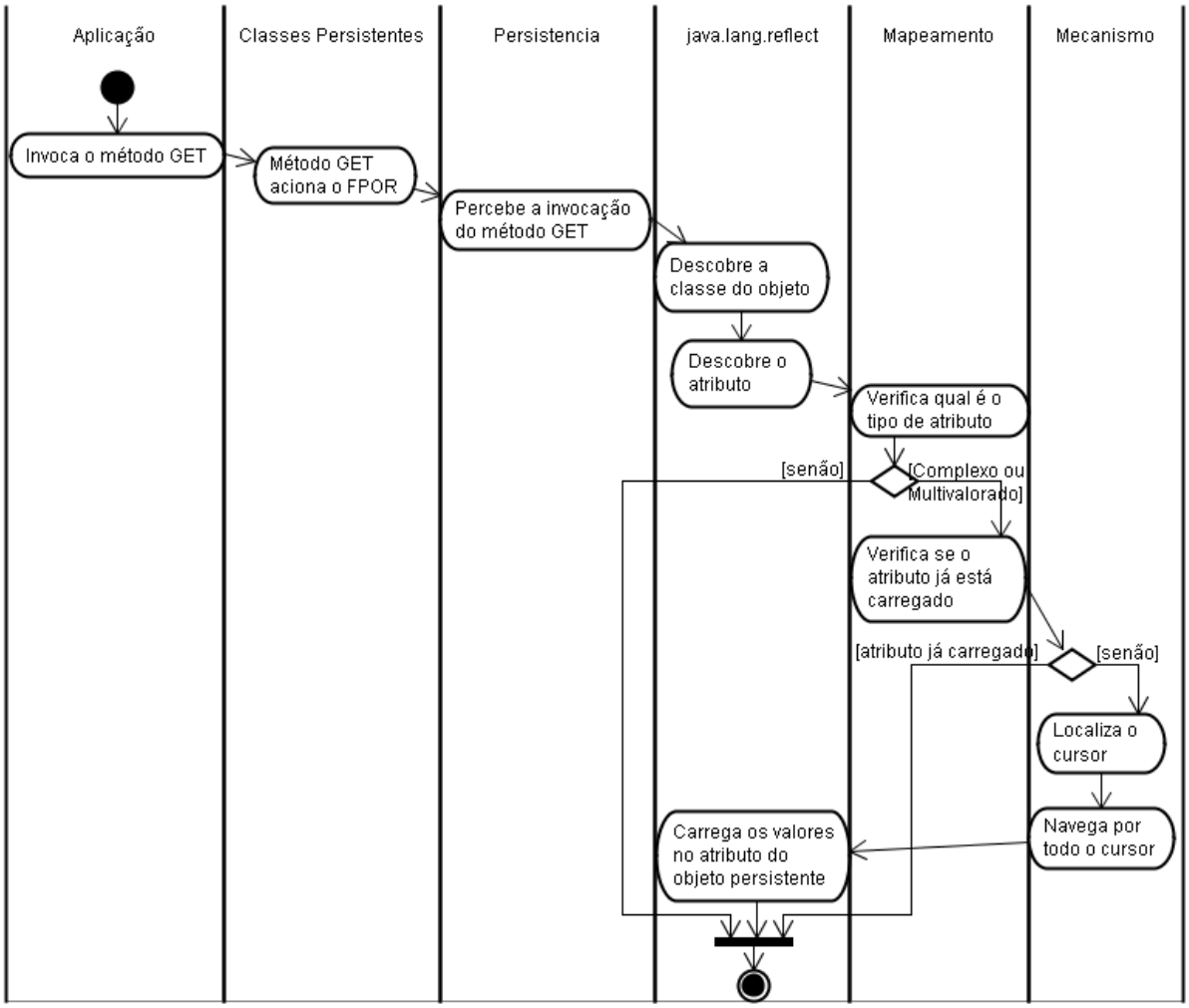


Figura 49 – Recuperação sob Demanda

No caso de exclusão, é feita a recuperação automática de todos os objetos relacionados, pois é imprescindível a atualização ou exclusão destes dependendo da opção em cascata configurada no mapeamento objeto relacional para o relacionamento, conforme Figura 49. Em alguns casos o banco de dados ou o próprio FPOR pode acusar a falta de um valor definido para uma determinada coluna durante uma operação no ambiente de banco de dados.

3.3 PROCESSO DE DESENVOLVIMENTO

O processo de desenvolvimento descrito a seguir se restringe a um subgrupo de etapas metodológicas de desenvolvimento de sistemas pertinentes ao framework

de persistência. Demais etapas e elementos previstos em uma metodologia de desenvolvimento devem ser preservados e não serão mencionados com o objetivo de abreviar e simplificar a explanação deste trabalho.

As tarefas do processo de desenvolvimento estão distribuídas sob dois aspectos distintos: em relação à fase de desenvolvimento e em relação aos papéis dos profissionais envolvidos.

Na perspectiva de fases de desenvolvimento, concebemos duas fases de desenvolvimento: lógica e física em dois ambientes de desenvolvimento: aplicação e banco de dados conforme Figura 50.

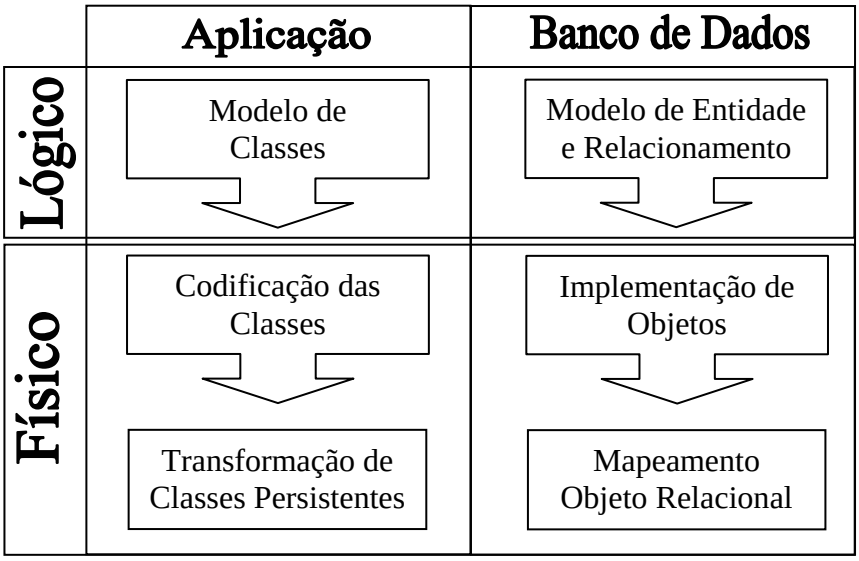


Figura 50 – Fases de Desenvolvimento

A etapa lógica engloba a elaboração dos diagramas de classe e de entidade relacionamento. Estes diagramas formam o embasamento para a concepção do mapeamento objeto relacional.

A etapa física começa pela codificação das classes desenhadas nos diagramas para uma linguagem orientada a objetos. A tarefa subseqüente é a transformação das classes persistentes, na qual se implementa de maneira transparente os métodos necessários à persistência no código binário das classes, descrita na seção 3.1.1.2. No ambiente de Banco de Dados, de forma similar a decodificação das classes, temos a codificação das entidades e relacionamentos em

objetos de banco de dados usando a linguagem SQL. O mapeamento objeto relacional consiste em realizar a “tradução” dos elementos do diagrama de classes para elementos do diagrama de entidade e relacionamentos, processo descrito no seção 3.1.5, que servirá de base para o funcionamento do framework de persistência.

No aspecto dos papéis dos profissionais envolvidos temos três perfis: Desenvolvedor, Administrador de Banco de Dados e o Administrador de Dados e Classes. A Figura 51 ilustra a distribuição das tarefas entre os perfis profissionais mencionados.

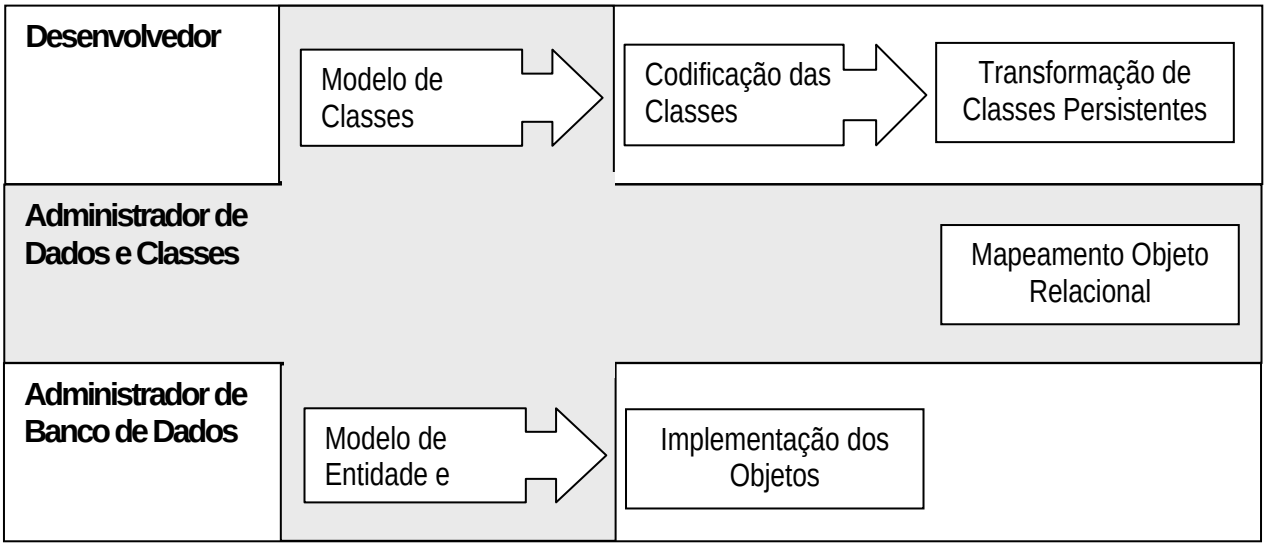


Figura 51 – Papéis Profissionais

Para que os desenvolvedores tenham transparência em relação à persistência do objeto e, ao mesmo tempo, o administrador de banco de dados não necessite de perícia na tecnologia de orientação a objetos, propomos neste trabalho o perfil de Administrador de Dados e Classes.

A separação do mapeamento no processo de desenvolvimento de sistemas sugere uma distinção entre duas áreas na organização do ambiente de Tecnologia da Informação: a dos desenvolvedores preocupados com as regras de negócio e outra preocupada com o melhor mapeamento objeto relacional.

Uma administração centralizada de dados já é fato em ambientes corporativos. A administração de dados pode ser definida como uma função da empresa responsável por desenvolver e administrar estratégias, procedimentos, práticas e planos capazes de disponibilizar os dados corporativos revestidos de integridade, privacidade, documentação e compartilhamento (BARBIERI,1994). As mesmas preocupações e cuidados devem acontecer com as classes em organizações corporativas. A implementação da administração centralizada de classes persistentes evita características desastrosas nos sistemas corporativos orientados a objetos como, por exemplo, a redundância de classes.

Profissionais especializados em organizar dados e classes corporativas teriam como atribuição a elaboração e manutenção dos modelos de classes e de diagramas de entidade relacionamento, bem como elaborar e manter o mapeamento objeto relacional. Este indivíduo influenciaria de forma decisiva tanto a modelagem de dados bem como a modelagem orientada a objetos elaborados pelos analistas, DBA’s e desenvolvedores, buscando a melhor proposta que atenda ao mapeamento objeto relacional de forma simples e prática, seguindo padrões difundidos em diversas publicações (AMBLER,2000a; AMBLER,1998; KELLER,1997).

O individuo que exercer este perfil profissional tem pleno domínio das tecnologias de orientação a objetos e de banco de dados sendo responsável pela compatibilização destas tecnologias, baseando-se no diagrama de classe e de entidade e relacionamento. A principal vantagem desta organização de perfis está nas manutenções realizadas no ambiente de banco de dados e no ambiente orientado a objetos. Quando o Administrador de Banco de Dados realizar manutenções no ambiente de persistência, por motivos de desempenho ou migração de versões ou de ferramentas, o desenvolvedor não será demandado por impactos na sua aplicação, bastando comunicar ao administrador de classes e dados. Este, por sua vez, realizará as alterações necessárias no mapeamento objeto relacional. Inversamente, quando o desenvolvedor realizar qualquer manutenção em seu ambiente orientado a objetos, o administrador de classes e dados se encarregará de realizar as atualizações necessárias no mapeamento objeto relacional, sem afetar o ambiente de persistência.

O perfil de administrador de classes e dados é estratégico visto que concentra alterações no ambiente orientado a objetos e no ambiente de banco de dados e desempenha um papel centralizador e corporativo, impedindo que existam classes e dados em duplicidade, e garantindo a coerência entre estes elementos. Conseqüentemente, o processo de desenvolvimento torna-se bem controlado e estruturado, o que leva a produzir softwares corporativos de qualidade superior.

4. AVALIAÇÃO DO FPOR

Para avaliação da proposta de framework para persistência de objetos – FPOR – apresentada no capítulo 3, foi realizada uma experimentação controlada, utilizando-se este framework na implementação de uma aplicação. A seleção da aplicação para o estudo de caso obedeceu aos seguintes pré-requisitos: (i) pequeno porte para que fosse possível efetuar esta implementação dentro do escopo de uma tese de mestrado e (ii) complexidade suficiente para avaliar as diversas características relevantes do FPOR.

É importante ressaltar que durante o processo de avaliação, foi possível identificar falhas tanto de modelo quanto de implementação do framework. Este trabalho possibilitou o amadurecimento do FPOR, evidenciando situações não previstas e provocando ajustes no seu funcionamento.

Este capítulo relata o estudo de caso, compara os resultados obtidos com os requisitos inicialmente estabelecidos (vide seção 1.3) neste trabalho e, finalmente, compara as características do FPOR com ferramentas para mapeamento objeto relacional disponíveis no mercado.

4.1 ESTUDO DE CASO

A aplicação utilizada para o estudo de caso do FPOR se refere a um Sistema de Gerência de Infrações de Trânsito – referenciado daqui por diante como SGIT.

O SGIT tem como finalidade principal a automação de um departamento de trânsito para registrar motoristas(Pessoa), carteiras de habilitação (CNH), veículos, multas e recursos de contestação a autos de infração de trânsito em primeira instância. Apesar de se tratar de um sistema simples, engloba diversos aspectos da modelagem orientada a objetos a serem resolvidos no processo de persistência de objetos. Este exemplo, oriundo de uma matéria da revista SQL Magazine (ZIMBRÃO,2003), foi utilizado para a demonstração de um processo de mapeamento de um modelo de classes para um banco de dados relacional. O modelo conceitual do SGIT está exposto na Figura 52.

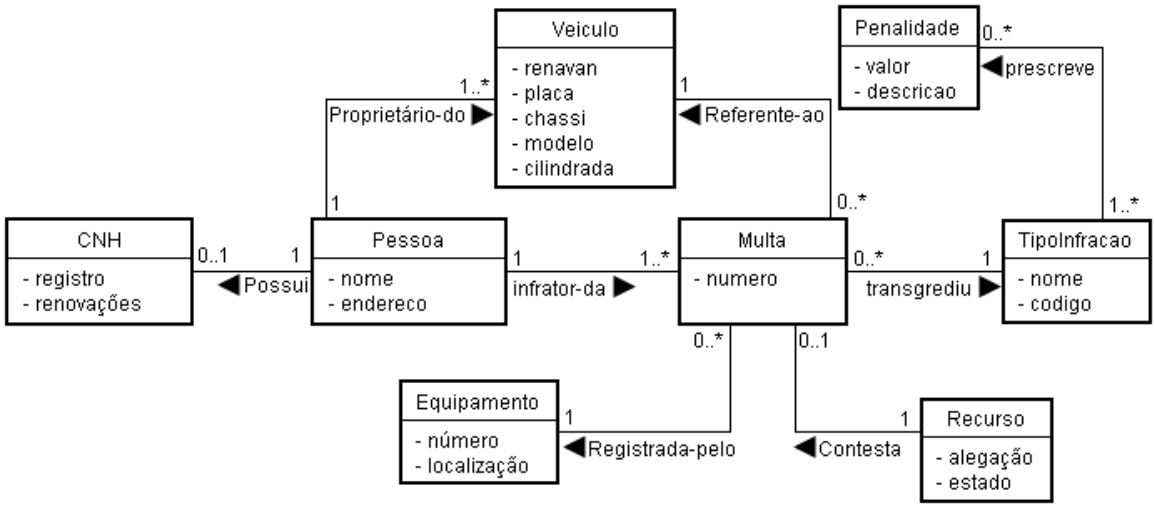


Figura 52 - Modelo de Domínio do SGIT

4.1.1 Modelo Orientado a Objetos

O estudo de caso segue a arquitetura em camadas, conforme Figura 52 que demonstra as camadas definidas neste trabalho:

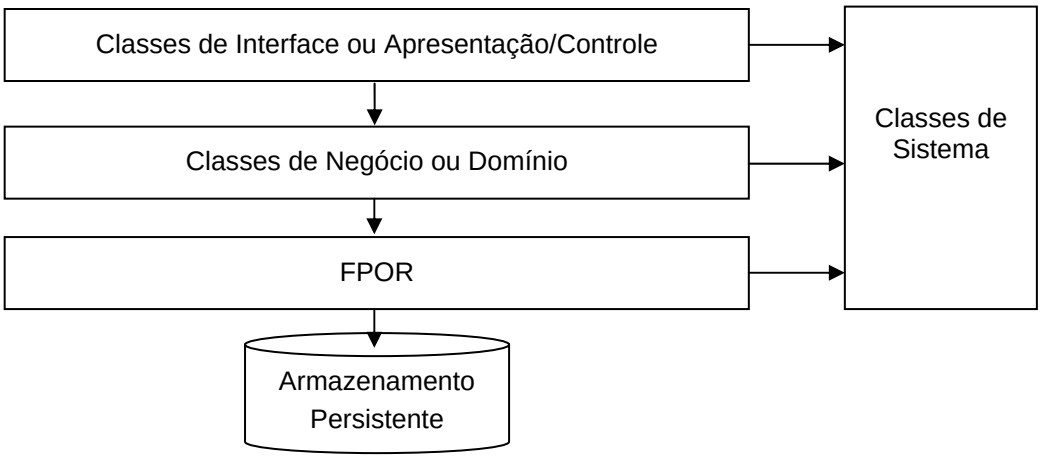


Figura 53 - Arquitetura em Camadas do SGIT

As classes de interface ou apresentação fornecem aos usuários a capacidade de interagir com o sistema, através de uma interface gráfica (GUI) ou outros tipos de interface, como por exemplo comando de voz ou entrada escrita. As classes de controle tratam da dinâmica da aplicação. As classes de Negócio ou Domínio modelam o domínio do negócio, no caso do estudo de caso um departamento de trânsito. Classes de Sistema fornecem funcionalidade específica do sistema operacional ou de terceiros, tornando a aplicação portátil em função do acoplamento fraco (LARMAN,2000) do ambiente operacional proporcionado (AMBLER,1998).

O diagrama das classes do estudo de caso é apresentado na Figura 54. Podemos observar que as alterações feitas a partir do modelo de domínio relativas à persistência se restringem à inclusão do atributo ID em cada classe persistente que se destina a identificar univocamente o objeto.

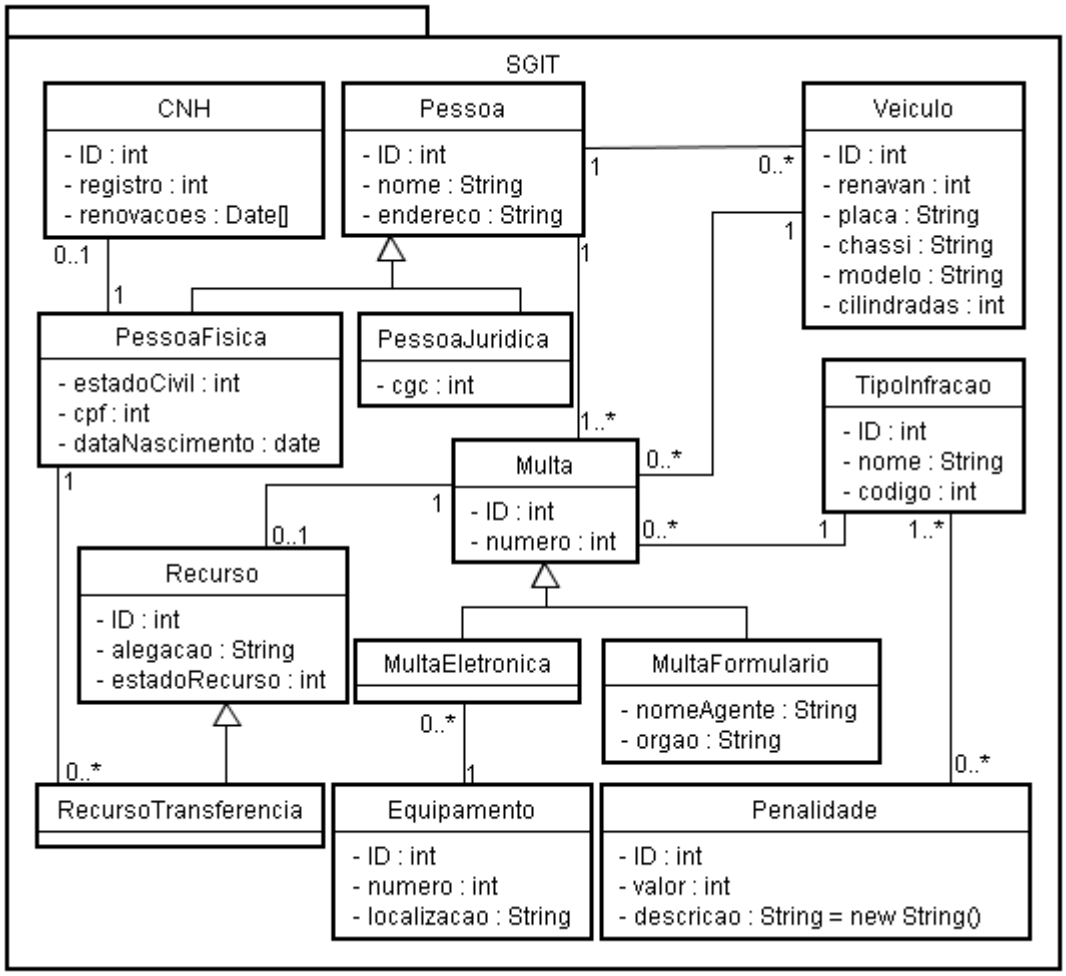


Figura 54 – Modelo de Classes SGIT

No contexto deste trabalho, as camadas de controle e apresentação foram reunidas em uma única camada, apresentada no modelo UML da Figura 55. Estas classes têm por objetivo apenas facilitar os testes da persistência dos objetos com os quais interagem. Representam basicamente as classes da camada de apresentação com pequenas porções de controle. Estas classes instanciam os objetos persistentes e, através dos métodos incorporados do FPOR, realizam operações de recuperação e atualização sobre estes objetos.

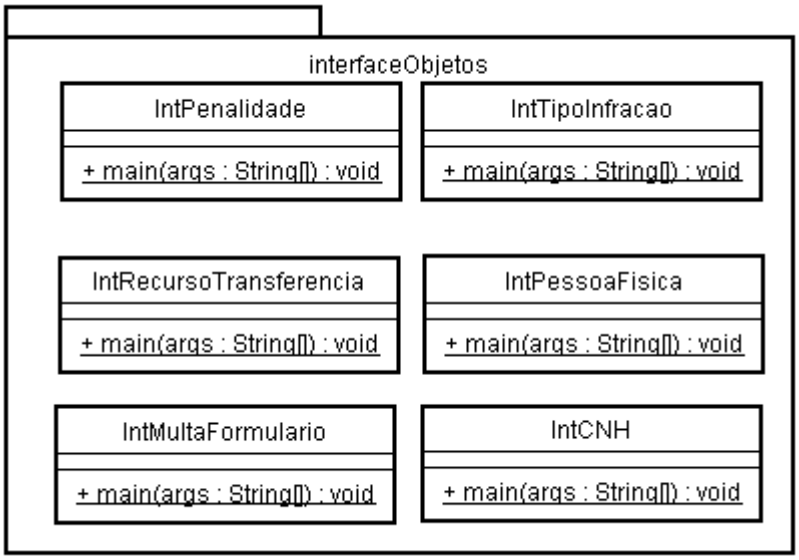


Figura 55 – Modelo de Classes Interface do SGIT

4.1.2 Modelo de Entidade Relacionamento

A Figura 56 apresenta o modelo de entidade relacionamento do SGIT. Este modelo foi obtido partindo-se do modelo de classes e aplicando os critérios de mapeamento descritos na seção 4.1.3.

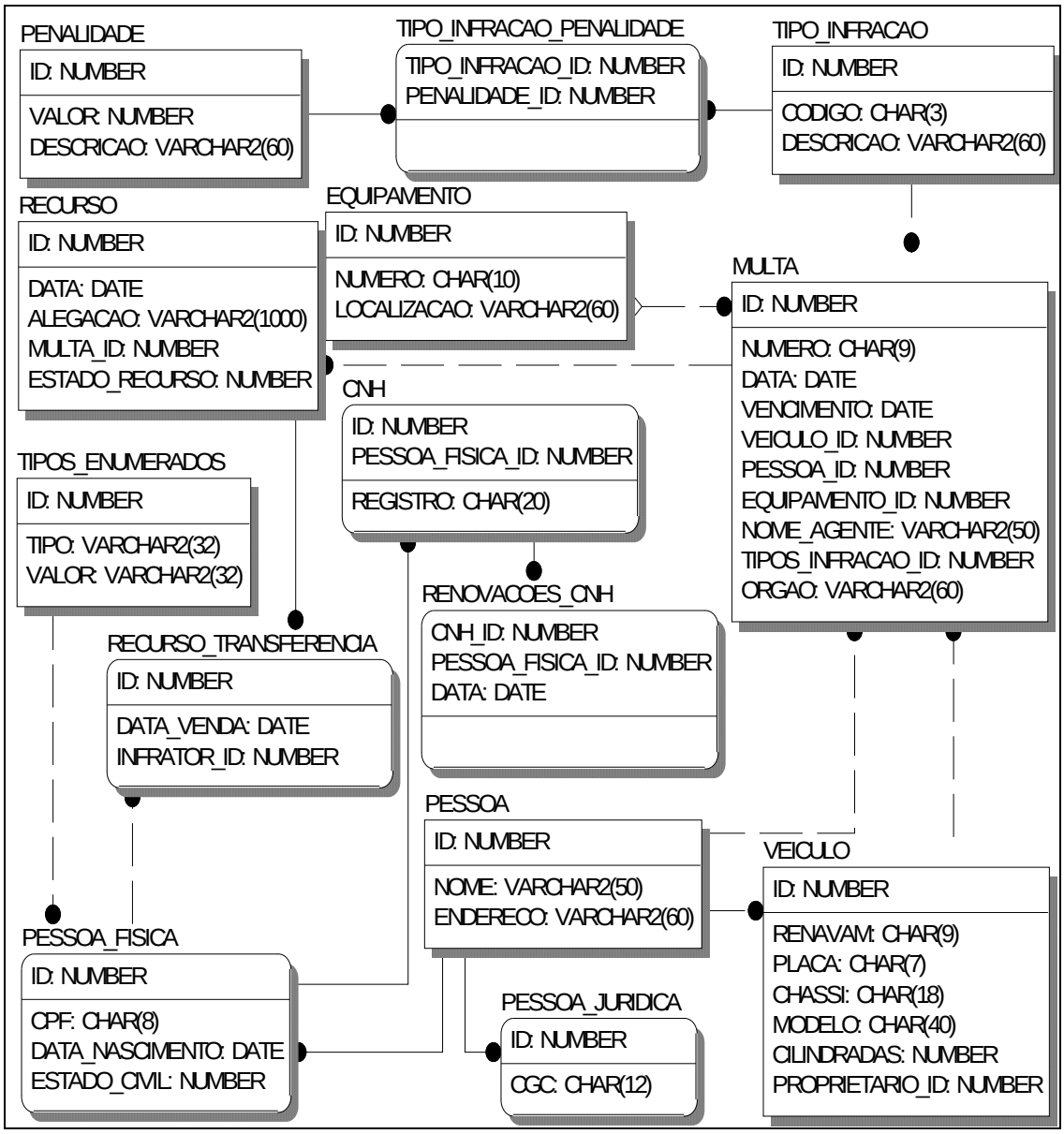


Figura 56 – Modelo de Entidade Relacionamento do SGIT

Através destes diagramas podemos observar as equivalências entre classes e entidades e como cada relacionamento foi implementado.

4.1.3 Critérios empregados no Mapeamento Objeto Relacional

Os principais conceitos de orientação a objetos abordados e utilizados no mapeamento para o modelo relacional são: identidade, classes, atributos, relacionamentos, herança e restrições. Os mapeamentos se basearam no item

3.1.3.7 e os tópicos a seguir descrevem a estratégia adotada para cada um destes conceitos.

4.1.3.1 Identidade de Objetos

Todo objeto deve possuir uma identificação única. Cada objeto deve corresponder a pelo menos um registro em uma tabela do Banco de Dados e sua identificação é implementada através de chaves artificiais denominadas *surrogate.* Este conceito é bem aceito junto à comunidade de desenvolvedores em modelagem de banco de dados relacional e consenso na comunidade de desenvolvedores de sistemas Orientados a Objetos.

A chave surrogate é um número inteiro gerado seqüencialmente e que não tem significado no negócio. Neste modelo de objetos e de dados o atributo identificador dos objetos e registros será denominado ID e será alimentado por um dispositivo gerador de números seqüenciais.

Esta abordagem desvincula a representação física da informação de características relevantes do negócio. Desta forma, alterações na aplicação que influenciam a identificação de um objeto não se refletem em mudanças na chave primária do registro correspondente. É importante lembrar que alterações em chaves primárias são normalmente trabalhosas, tendo em vista que outras tabelas e aplicações podem estar referenciando esta chave primária.

4.1.3.2 Classes

Será adotado um mapeamento entre classe e tabela na relação de um para um onde cada classe terá uma tabela correspondente de mesmo nome. Relacionamentos de cardinalidade muitos para muitos e atributos multivalorados podem dar origem a outras tabelas.

Para classes em hierarquias de generalização/especialização adotou-se uma tabela por classe. Por exemplo, no SGIT a classe Pessoa representa um exemplo de generalização das classes *PessoaJuridica* ou *PessoaFisica*. Estas classes foram mapeadas nas tabelas PESSOA, PESSOA_JURIDICA e PESSOA_FISICA respectivamente, cada uma contendo somente as colunas correspondentes aos atributos previstos para cada classe. As chaves primárias das tabelas que implementam as subclasses também são chaves estrangeiras da tabela que representa a superclasse nesta hierarquia.

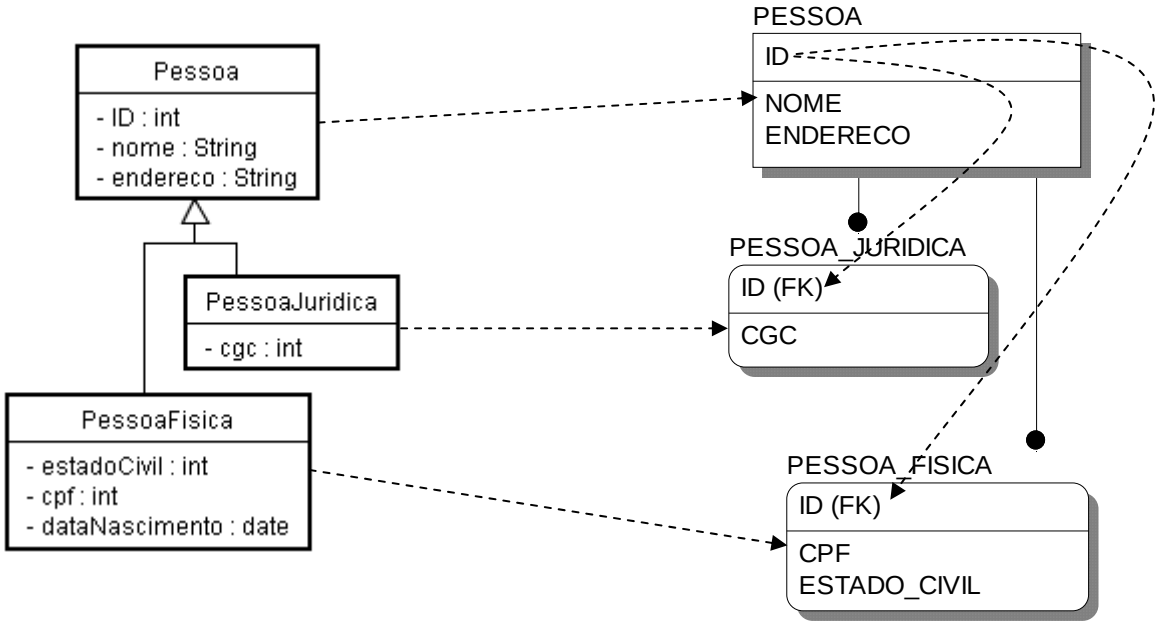


Figura 57 – Diagrama de Hierarquia

4.1.3.3 Atributos

Para o mapeamento é importante observar a implementação das características de tipo e cardinalidade de um atributo

A tecnologia de orientação a objetos permite que atributos de objetos assumam, além de tipos primitivos de dados, referências a outros objetos que denominamos atributos complexos.

- Atributos de Tipo Primitivo

Tipos primitivos serão diretamente mapeados nas colunas correspondentes (na relação de um para um) com equivalente tipo no banco de dados.

- Enumerados

Tipos enumerados são tipos com número finito de valores, tais como estado civil. Neste mapeamento, foi criada a tabela TIPO_ENUMERADOS para abrigar a codificação dos valores deste tipo de atributo. Apesar desta implementação, estes atributos, semelhantes aos tipos primitivos, são diretamente mapeados nas correspondentes colunas de banco de dados. O FPOR, por simplicidade, não fez qualquer tratamento em especial em relação a este atributo, sendo considerado um simples atributo numérico. Caberia à classe persistente, na implementação, a criação de um atributo descritivo que seja associada a uma função de banco de dados responsável por traduzir o código do atributo enumerado. Esta função receberia como parâmetro o valor do enumerado, consultaria a tabela TIPO_ENUMERADOS e retornaria o seu significado. Esta função seria associada ao atributo descritivo através do pacote *alternativo*, descrito na seção 3.1.3.3.

- Cardinalidade de Atributos

A cardinalidade de um atributo pode ser zero, um ou uma coleção. A questão de ser zero ou um reflete a obrigatoriedade do atributo, resolvida através de restrições implementadas no banco de dados. Coleções são implementadas em tabelas separadas onde uma chave estrangeira referencia a chave primária da tabela original para o gerenciador de banco de dados estabelecer restrições de integridade referencial. Um exemplo desta situação é a classe *CNH* que contém um atributo denominado *renovacoes*. Este atributo é implementado com a criação da tabela RENOVACOES_CNH que se relaciona com a tabela CNH e armazena as datas das renovações da carteira nacional de habilitação, representado pelo atributo *renovacoes* da classe *CNH*.

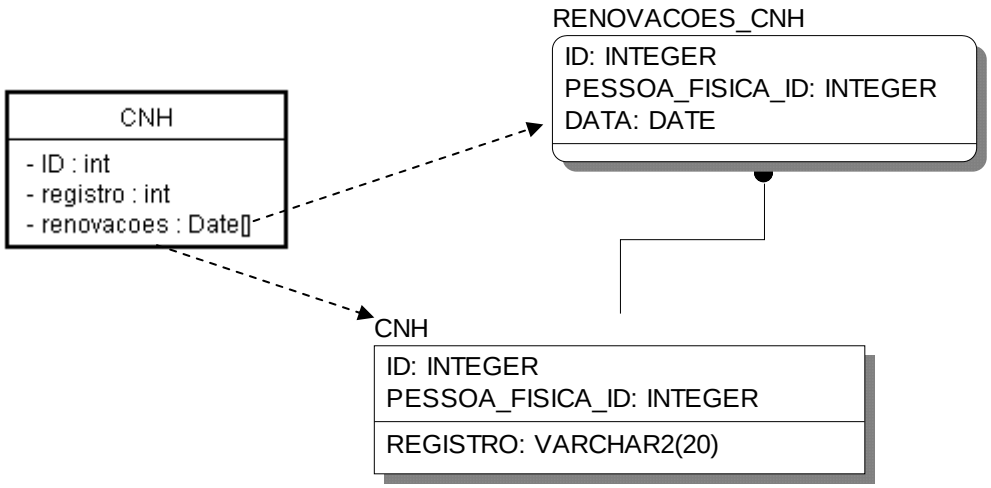


Figura 58 – Implementação de Coleções

4.1.3.4 Associações

Os atributos de cada classe no modelo conceitual são apenas de tipo primitivo. Na implementação surgem atributos cujo tipo são outras classes, que denominamos atributos complexos, sendo expressos em relacionamentos no modelo conceitual.

As associações entre classes se traduzem no relacionamento entre as tabelas que representam estas classes. Desta forma, chaves estrangeiras são introduzidas nestas tabelas no modelo relacional de banco de dados. Exploramos no FPOR a resolução destes relacionamentos através de *joins* elaborados em tempo de execução, baseados no mapeamento objeto relacional definido pelo administrador de dados e classes, para a recuperação das informações referenciadas. Os processos de resolução e mapeamento objeto relacional foram descritos no capítulo 3.

Os relacionamentos de cardinalidade muitos para muitos geram tabelas intermediárias para resolver a associação entre os objetos. Um exemplo deste tipo de relacionamento é a tabela TIPO_INFRACAO_PENALIDADE que implementa a associação entre as classes *TipoInfracao* e *Penalidade* representadas pelas tabelas TIPO_INFRACAO e PENALIDADE respectivamente.

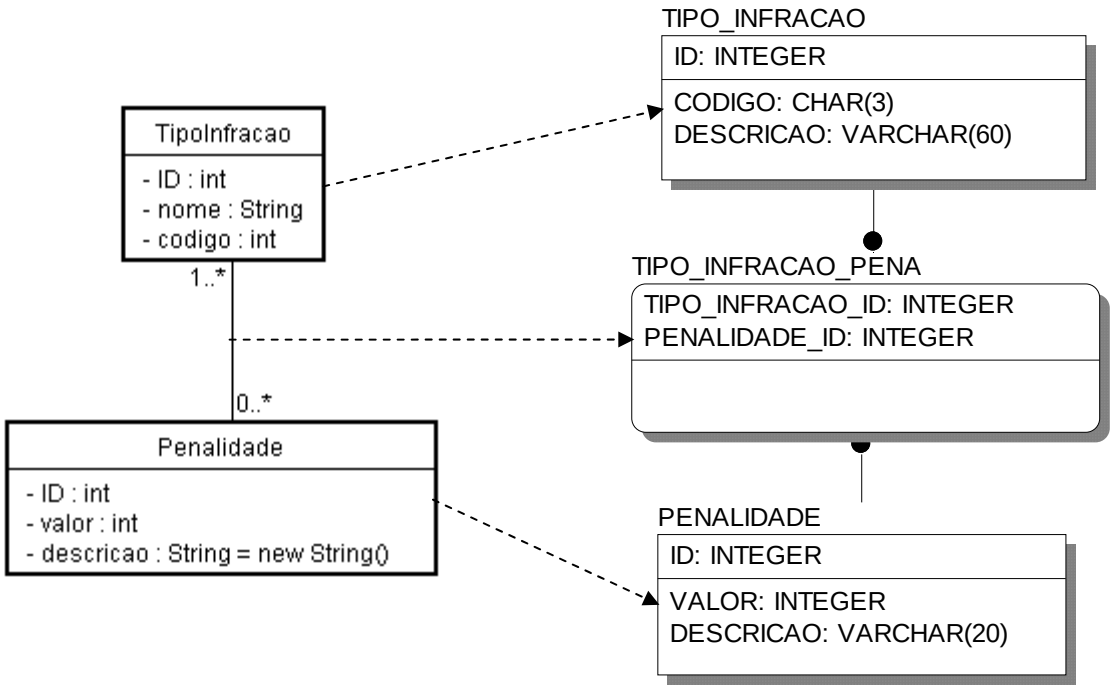


Figura 59 – Implementação de Relacionamento N para N

4.1.3.5 Restrições

Restrições do modelo UML podem ser mapeadas diretamente em *constraints* do modelo Entidade Relacionamento. *Constraints* são implementadas nos gerenciadores de banco de dados para garantir o cumprimento da regra de negócio ou integridade referencial.

As seguintes restrições definidas para atributos de objetos são tratadas através de mecanismos já existentes no banco de dados, não havendo intervenção do FPOR:

a) *constraints*, o que facilita a implementação de valores enumerados e obrigatoriedade. Um exemplo é o atributo *estadoCivil* da classe *PessoaFisica* que pode assumir somente os valores "solteiro", "casado", "separado", "divorciado", "viúvo". Esta coluna é implementada no modelo relacional através de uma *check constraint* que restringe os valores possíveis para a

coluna ESTADO_CIVIL da tabela PESSOA_FISICA em "SOLTEIRO", "CASADO", "SEPARADO", "DIVORCIADO” e ”VIUVO".

b) *triggers,* utilizados nas restrições de negócio, como por exemplo: “não permitir criar um recurso para uma multa que já venceu”. Esta regra é garantida através de uma *trigger* no evento de inclusão de um novo registro na tabela RECURSO que verifica se a MULTA associada ao registro está vencida. A opção de se utilizar este recurso para restrições pode levar a uma dependência da aplicação em relação ao fornecedor de banco de dados. Portanto recomenda-se utilizar classes de controle na realização desta restrição.

4.2 IMPLEMENTAÇÃO DO SGIT USANDO FPOR

Nesta seção demonstraremos a utilização do FPOR por classes de interface na persistência dos objetos de negócio do SGIT.

4.2.1 Classes Persistentes

Para a utilização do FPOR, o desenvolvedor apenas precisa explicitar a classe persistente como um subtipo da interface *InterfacePersistencia*, conforme demonstrado na Figura 60.

```
public class Penalidade implements InterfacePersistencia {
    private int ID;
    private int valor;
    private String descricao = new String();
    public  Penalidade() throws Exception {
      super();
    } // end Penalidade
```

```
    public String toString() {
      try
      {
         System.out.println("------------------------------");
         System.out.println("Classe Penalidade");
         System.out.println("*****************");
         System.out.println("ID        = "+this.ID);
         System.out.println("Valor     = "+this.valor);
         System.out.println("Descricao = "+this.descricao);
         System.out.println("------------------------------");
         //int espera =  System.in.read ();
         return "";
      }
       catch(Exception ex)
      {
         System.out.println("Problemas..."+ex);
         ex.printStackTrace();
         return "";
      }
    } // end toString
    public int getID() {
        return ID;
    } // end getID
    public void setID(int _ID) {
        ID = _ID;
    } // end setID
    public int getValor() {
        return valor;
    } // end getValor
    public void setValor(int _valor) {
        valor = _valor;
    } // end setValor

    public String getDescricao() {
        return descricao;
    } // end getDescricao
    public void setDescricao(String _descricao) {
        descricao = _descricao;
    } // end setDescricao
} // end Penalidade
```

Figura 60 - Código fonte da Classe Penalidade

Após ser compilada, esta classe é tratada pela classe *preCompiladorJavaAssist* que manipulará o seu código *bytecode* para disponibilizar os métodos da InterfacePersistencia (vide seção 3.1.1.2).

4.2.2 Classes de Interface

Como as classes de interface tinham apenas o objetivo de testar as funcionalidades das classes persistentes normalmente necessárias em uma aplicação, seus códigos consistem em uma única *main* que segue o roteiro de execução abaixo, conforme Figura 61, utilizando-se dos métodos disponibilizados pelo FPOR na classe persistente:

a) Definir um Mecanismo Persistente (Figura 61.a);

b) Definir o Mapeamento Objeto Relacional a ser utilizado (Figura 61.b);

c) Instanciar objetos de negócio (Figura 61.c);

d) Preencher seus atributos (Figura 61.d);

e) Incluir objetos na base de dados (Figura 61.e);

f) Pesquisar um objeto, definindo um critério de busca (Figura 61.f);

g) Pesquisar objetos, com um critério de busca (Figura 61.g);

h) Pesquisar todos os objetos (Figura 61.h);

i) Atualizar um objeto (Figura 61.i);

j) Pesquisar o objeto atualizado no banco de dados para verificar se realmente foi atualizado no SGBD (Figura 61.j);

k) Excluir todos os objetos da base (Figura 61.k);

l) Pesquisar se restou algum objeto na base (Figura 61.l);

```
public class IntPenalidade {
    public static void main(String[] args) {
    try
    {
61.a  // Criando um Mecanismo Persistente
      MecanismoPersistente mp =
                new MecanismoPersistente(     "usuario",
                                              "manager",
                                              "jdbc:oracle:oci8:@ora");
61.b  // Define o mapeamento...
      Persistencia.defineMapeamento(
                new ParseOracle(  "mapeamento",
                                  "manager",
                                  "jdbc:oracle:oci8:@ora"));
61.c  //Instanciando o objeto...
      Penalidade pen = new Penalidade();
61.d  // Carrega seus atributos
      pen.setID(1);
      pen.setValor(200);
      pen.setDescricao("Duzentas UFIR's");
61.m  // Conecta o Mecanismo Persistente
      mp.conecta();
61.e  // Realiza a operação de incluir
      pen.insere(mp);
      mp.efetiva();
      System.out.println(pen);
61.f  // Consulta um objeto com criterio...
      CriterioPesquisa cp1 = pen.criaCriterioPesquisa();
      cp1.adicionaCriterio("ID","=","1");
      if (pen.pesquisaUm(mp,cp1))
      {
         System.out.println(pen);
      }
61.g  // Consulta varios objetos, sem criterio...
      pen.pesquisa(mp);
      while (pen.proximo(mp))
            System.out.println(pen);
```

```
61.h  // Consulta varios objetos, com criterio (ID>1)...
      CriterioPesquisa cp2 = pen.criaCriterioPesquisa();
      cp2.adicionaCriterio("ID",">","1");
      pen.pesquisa(mp,cp2);
      while (pen.proximo(mp))
      {
         System.out.println(pen);
      }

61.i  // Realiza a operação de atualização
      if (pen.pesquisaUm(mp,cp1))
      {
         pen.setDescricao("Atualizado.");
         pen.atualiza(mp);
         mp.efetiva();
      }

61.j  // Pesquisa o objeto para verificar se foi realmente atualizado...
      if (pen.pesquisaUm(mp,cp1))
          System.out.println(pen);

61.k  // Deletando os objetos...
      pen.pesquisa(mp);
      while (pen.proximo(mp))
         pen.exclui(mp);
      mp.efetiva();

61.l  // Pesquisando para saber se tem algum objeto...
      pen.pesquisa(mp);
      boolean aindaExiste = false;
      while (pen.proximo(mp))
         aindaExiste = true;
      if (!(aindaExiste))
         System.out.println("Nenhum registro encontrado.");

61.n  mp.desconecta();
   }
   catch(Exception ex)
   {
      System.out.println("Problemas..."+ex);
      ex.printStackTrace();
   }
   } // end main
} // end IntPenalidade
```

Figura 61 - Código fonte da Classe IntPenalidade

O Mecanismo Persistente é um elemento essencial à persistência de objetos, pois através deste as ações de persistência são efetivadas.

Sua conexão é feita através da declaração de um objeto da classe *MecanismoPersistente* informando os parâmetros necessários, de acordo com a especialização de banco de dados que estiver sendo utilizada. Exemplificamos sua declaração no trecho de código demonstrado na Figura 61.a.

Neste caso, a especialização utilizada foi o ORACLE, no qual deve ser informado o usuário, senha e a *URL* de conexão com o nome do banco de dados.

Este objeto é utilizado como parâmetro para a realização de operações de persistência de objetos. Como demonstra a Figura 61.e, no qual se insere o objeto persistente *pen* no mecanismo persistente, representado pelo objeto *mp*. Porém, como pré-condição para a realização destas operações, o mecanismo persistente deve estar conectado (Figura 61.m). Conseqü entemente, ao final de todas as operações persistentes, deve-se liberar a conexão conforme Figura 61.n.

O conceito de transação está na classe mecanismo persistente, pois o FPOR se utiliza dos controles de transação do SGBD. A classe *MecanismoPersistente* disponibiliza os métodos *efetiva* e *desfaz* que correspondem à primitivas *commit_transaction* e *rollback_transaction* descritas no capítulo 2. No caso do trecho da Figura 61.e, exemplificamos a efetivação desta transação:

```
pen.efetiva();
```

A definição do mapeamento objeto relacional é através de um método estático denominado *defineMapeamento* da classe *Persistencia*. O trecho da Figura 61.b demonstra a definição de um Mapeamento Objeto Relacional.

O Mapeamento Objeto Relacional deve estar definido pela classe de interface antes da criação do objeto persistente. Considerando que o repositório do mapeamento objeto relacional se encontra em um mecanismo persistente, os parâmetros informados na sua definição são semelhantes à conexão de um Mecanismo Persistente. Este dispositivo só é utilizado na instanciação do objeto persistente, quando ocorre a pesquisa das características da classe do objeto junto ao repositório do mapeamento objeto relacional, tais como, atributos,

relacionamentos, tipos de atributos, etc. Portanto este processo só é efetuado uma vez para cada classe em toda a execução da aplicação, evitando freqüentes acessos ao repositório do mapeamento objeto relacional.

4.2.3 Recuperação de Objetos

A criação de um critério de pesquisa é imprescindível na recuperação refinada de um ou mais objetos. A criação de um objeto da classe *CriterioPesquisa* é feita por uma classe interface através do método *criaCriterioPesquisa* da classe persistente.

O objeto da classe *CriterioPesquisa* prevê a criação de sentenças de filtro, através do método *adicionaCriterio*, possibilitando a definição do refinamento da pesquisa.

A criação do objeto *CriterioPesquisa* e a utilização do método *adicionaCriterio* estão demonstradas nas Figura 61.h e Figura 61.f. Além do método *adicionaCriterio*, apesar de não validados, a classe CriterioPesquisa prevê os métodos *adicionaE*, *adicionaOU*, *inicioParenteses*, *fimParenteses*, *eAdicionaCriterio* e *ouAdicionaCriterio* para a elaboração de critérios de pesquisa mais complexos.

Uma vez definido o(s) critério(s), podemos realizar a pesquisa de um objeto ou de vários objetos.

A pesquisa de um objeto é feita através de um método específico para este fim, que provocará erro, caso tenha recuperado mais de um objeto da base com o critério definido. O método retorna um valor lógico, indicando se a pesquisa foi feita com sucesso. Demonstramos a realização deste tipo de pesquisa através da Figura 61.f.

A pesquisa coletiva de objetos pode ser opcionalmente baseada em um critério de pesquisa. Seu funcionamento é similar a um cursor do banco de dados. Em um primeiro momento, se realiza a pesquisa e depois percorre seu resultado, sendo cada objeto desta coleção acessível a cada navegação. As Figura 61.g e

Figura 61.h apresentam a pesquisa de uma coleção de objetos persistentes, sem e com critérios respectivamente, bem como, navegar no seu resultado.

A Figura 62 mostra um pedaço de código que foi inserido na classe *IntPenalidade* com a finalidade de demonstrar o controle de estados implementado no FPOR. Neste trecho é feita uma pesquisa de todos os objetos persistentes da classe *Penalidade*. Em seguida, é feita a navegação no resultado desta pesquisa e para cada objeto recuperado, o atributo *descricao* é atualizado para “Erro”.

Na navegação para o segundo objeto da classe *Penalidade* da coleção, o método *proximo* gera uma exceção, informando que não é possível realizar a navegação pois o objeto foi alterado.

```
...
       // Teste da exceção em função de navegação com objeto atualizado
       // Realiza uma nova pesquisa...
       pen.pesquisa(mp);
       while (pen.proximo(mp))
       {
          // Atualiza um atributo do objeto...
          pen.setDescricao("Erro");
          // Na segunda interação o método próximo
          // provoca uma exceção informando que não é possível
          // a navegação, pois o objeto foi atualizado.
       }
...
```

Figura 62 - Trecho de Demonstração do Controle de Estados

4.2.4 Hierarquia

A classe *PessoaFisica* é uma especialização da classe *Pessoa* e sua implementação foi feita em tabelas distintas no banco de dados. A Figura 63 demonstra este mapeamento.

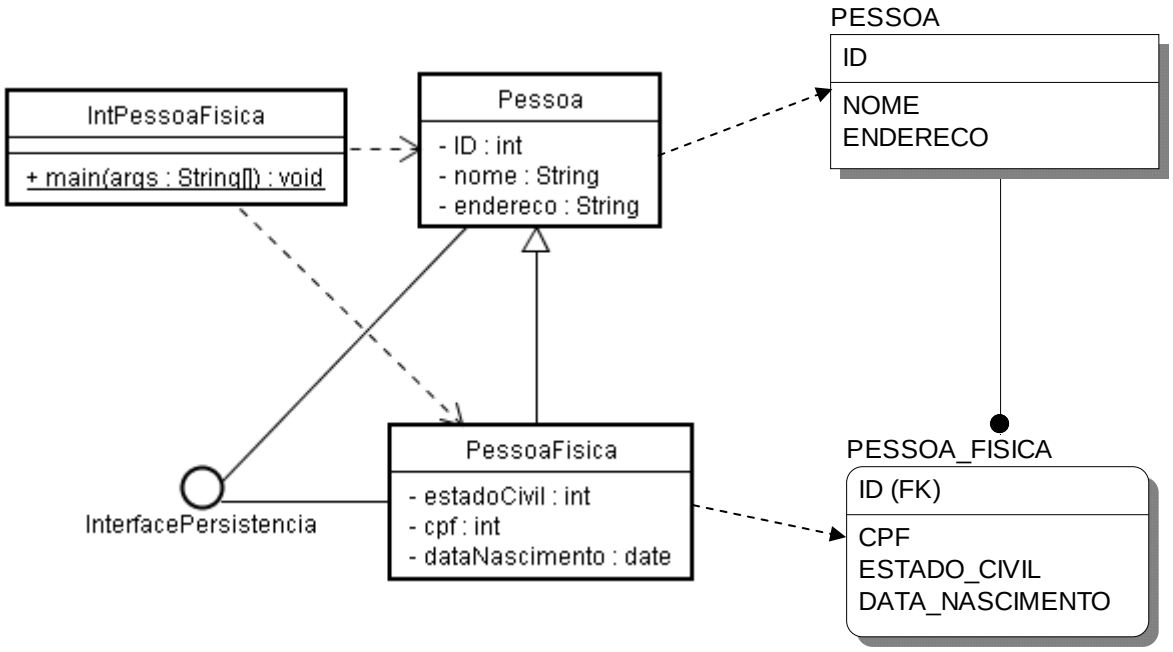


Figura 63 - Hierarquia Pessoa e PessoaFisica

A realização de serviços de persistência abrange além dos próprios atributos, sejam primitivos ou complexos, os atributos da superclasse. Na Figura 64.a, um objeto da classe *PessoaFisica*, ao ter suas propriedades preenchidas, verifica-se que algumas destas são de sua superclasse *Pessoa*. O FPOR ao realizar a inclusão deste objeto, deve também considerar os atributos da superclasse *Pessoa*.

A pesquisa de um objeto deve incluir a recuperação de classes superiores em sua estrutura hierárquica. A consulta pode ter como critério valores de um atributo de uma superclasse da hierarquia. Esta situação acontece na Figura 64.c, no qual o critério de pesquisa é o atributo ID da superclasse *Pessoa*.

As operações persistentes devem ser propagadas sobre classes de níveis hierárquicos superiores à classe persistente. Esta propagação é resolvida através de chamadas recursivas. Por exemplo, na Figura 64.a, quando o objeto da classe

persistente *PessoaFisica* é inserido, esta invoca o mesmo serviço de persistência, tendo como parâmetro o mesmo objeto persistente e a classe hierárquica imediatamente superior, no caso *Pessoa*, e assim sucessivamente até a classe "topo".

```
public class IntPessoaFisica {

public static void main(String[] args) {
 try
 {
    // Instancialização de Penalidade
    // Criando uma conexao...
    MecanismoPersistente mp =
                 new MecanismoPersistente(     "usuario",
                                               "manager",
                                               "jdbc:oracle:oci8:@ora");
    // Define o mapeamento...
    Persistencia.Persistencia.defineMapeamento(
                 new ParseOracle(  "mapeamento",
                                   "manager",
                                   "jdbc:oracle:oci8:@ora"));
    // Instanciando o objeto...
    PessoaFisica pf = new PessoaFisica();
    // Carrega seus atributos
    pf.setID(1);
    pf.setNome("Fulado de Tal");
    pf.setEndereco("Rua Fim do Mundo, 999 casa 1");
64.a java.util.GregorianCalendar gc =
                       new java.util.GregorianCalendar(1970,9,27);
    pf.setDataNascimento(gc.getTime());
    pf.setEstadoCivil(10);
    pf.setCpf(11111111);
    // Realiza a operação de incluir
64.b // Incluindo um objeto...
    pf.insere(mp);
    System.out.println(pf);
```

```
    // Carrega seus atributos
    pf.setID(2);
    pf.setNome("Fulado de Tal 2");
    pf.setEndereco("Rua Fim do Mundo, 999 casa 2");
    pf.setEstadoCivil(11);
    pf.setCpf(22222222);

    // Realiza a operação de incluir
    pf.insere(mp);
    System.out.println(pf);

    // Carrega seus atributos
    pf.setID(3);
    pf.setNome("Fulado de Tal 3");
    pf.setEndereco("Rua Fim do Mundo, 999 casa 3");
    pf.setEstadoCivil(12);
    pf.setCpf(33333333);

    // Realiza a operação de incluir
    pf.insere(mp);
    System.out.println(pf);

    // Consulta um objeto com criterio...
    CriterioPesquisa cp1 = pf.criaCriterioPesquisa();
    cp1.adicionaCriterio("ID","=","1");
    if (pf.pesquisaUm(mp,cp1))
    {
       System.out.println(pf);
    }
    // Consulta varios objetos, sem criterio...
    pf.pesquisa(mp);
    while (pf.proximo(mp))
          System.out.println(pf);
    // Consulta varios objetos, com criterio (ID>1)...
    CriterioPesquisa cp2 = pf.criaCriterioPesquisa();
    cp2.adicionaCriterio("ID",">","1");
    pf.pesquisa(mp,cp2);
    while (pf.proximo(mp))
    {
       System.out.println(pf);
    }
```

64.c

```
    // Realiza a operação de atualização
    if (pf.pesquisaUm(mp,cp1))
    {
       System.out.println(pf);
       pf.setEndereco(pf.getEndereco()+" Atualizado");
       pf.setEstadoCivil(pf.getEstadoCivil()+1);
       pf.atualiza(mp);
       System.out.println("Depois da atualizacao...");
       System.out.println(pf);
    }
    // Pesquisa o objeto para verificar se foi realmente atualizado...
    if (pf.pesquisaUm(mp,cp1))
        System.out.println(pf);

    // Deletando os objetos...
    pf.pesquisa(mp);
    while (pf.proximo(mp))
    {
       System.out.println(pf);
       pf.exclui(mp);
       System.out.println("Objeto excluído.");
    }
    // Pesquisando para saber se tem algum objeto...
    pf.pesquisa(mp);
    boolean aindaExiste = false;
    while (pf.proximo(mp))
    {
         aindaExiste = true;
    }
    if (!(aindaExiste))
       System.out.println("Nenhum registro encontrado.");
}
catch(Exception ex)
{
   System.out.println("Problemas..."+ex);
   ex.printStackTrace();
}
} // end main
} // end IntPessoaFisica
```

Figura 64 – Código Fonte IntPessoaFisica

4.2.5 Relacionamentos

A capacidade do FPOR em tratar relacionamentos entre objetos foi constatada na *IntCNH*, através da associação da classe *CNH* e *PessoaFisica*. A classe *CNH* tem um atributo complexo que referencia um objeto da classe *PessoaFisica* e, inversamente, a classe *PessoaFisica* contém um atributo complexo que referencia um objeto da classe *CNH*. Porém um relacionamento no ambiente orientado a objetos só pode se referir a uma chave estrangeira, no ambiente de banco de dados.

No banco de dados, o relacionamento entre a classe *PessoaFisica* e *CNH* tem a particularidade de ter a entidade *CNH* fraca, isto é, a tabela que implementa os dados da classe *CNH* tem o identificador da classe *PessoaFisica* fazendo parte de sua chave primária. Este fato demanda maior robustez nas operações com objetos da classe *CNH*, pois no momento da montagem das instruções de pesquisa, deve-se considerar o valor de atributos de outros objetos para compor a instrução. A situação descrita está ilustrada na Figura 65.

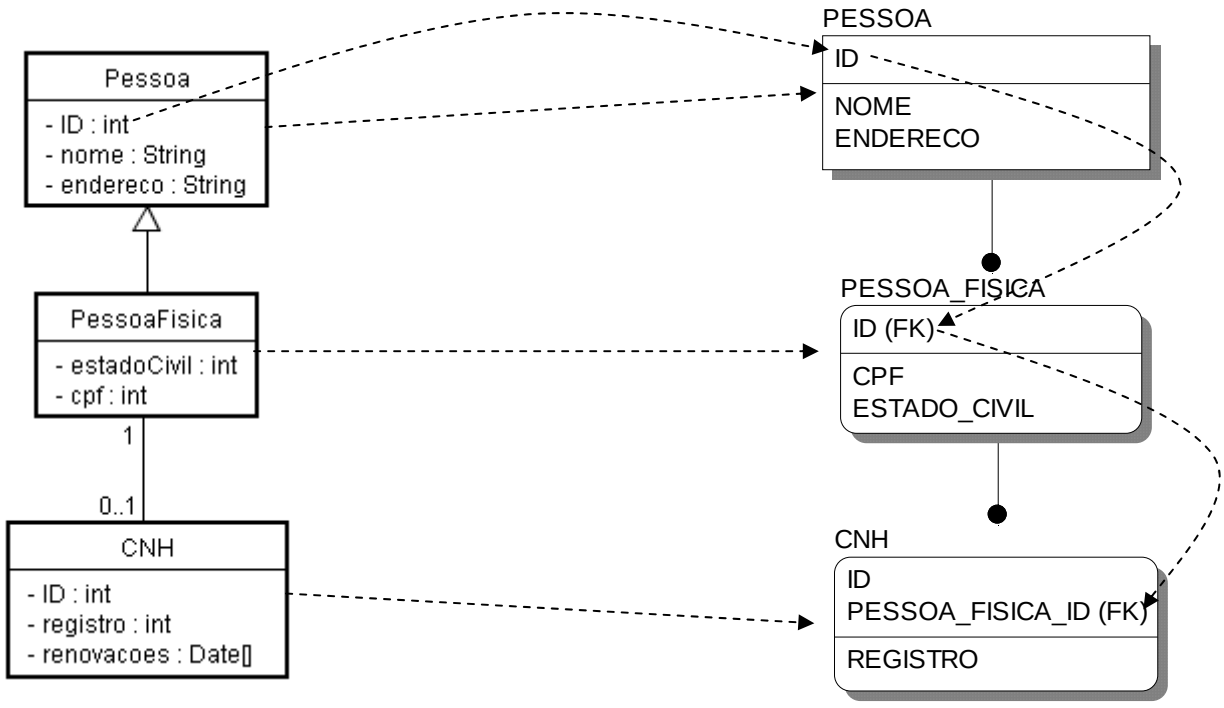


Figura 65 – Relacionamento CNH e PessoaFísica

4.2.6 Relacionamentos de Cardinalidade Muitos para Muitos

Para demonstrar o tratamento deste tipo de relacionamento utilizaremos a classe *Penalidade* e a classe *TipoInfracao*. O relacionamento de cardinalidade muitos para muitos entre estas classes é representado no ambiente de banco de dados através de uma tabela associativa denominada TIPO_INFRACAO_PENALIDADE. Esta tabela é composta pela chave primária da tabela TIPO_INFRACAO, que representa o objeto *TipoInfracao* possuidor do atributo complexo *penalidade*, combinada com a chave primária da tabela PENALIDADE que representa o objeto *Penalidade* ao qual o atributo complexo *penalidade* se refere.

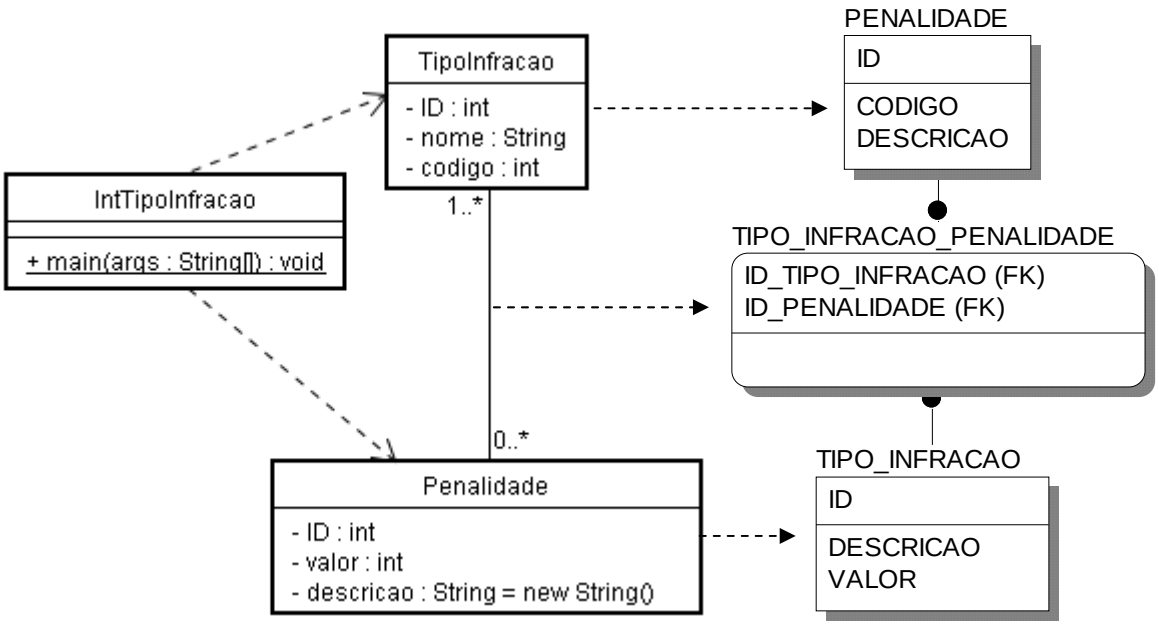


Figura 66 – Mapeamento Cardinalidade Muitos para Muitos

Como podemos observar nas Figura 67.a, Figura 67.b, Figura 67.c, vários objetos *penalidades* foram persistidos e relacionados a um objeto *TipoInfracao*, através de seu atributo complexo. No Figura 67.d, o objeto *TipoInfracao* é persistido com os relacionamentos aos objetos *penalidade*. Na base de dados, a tabela TIPO_INFRACAO_PENALIDADE foi preenchida corretamente, conforme definido no mapeamento objeto reacional que consta na Figura 66.

No Figura 67.e é demonstrada a pesquisa de um objeto *penalidade* pelo ID e, em seguida, este objeto é relacionado a um novo objeto *TipoInfracao*, para avaliar a capacidade de pesquisa de objetos do FPOR. A recuperação de objetos da classe *TipoInfracao* com os objetos da classe *penalidade* relacionados está demonstrado na Figura 67.f. A recuperação de todos os objetos da classe *penalidade* é feita com sucesso, porém não na mesma ordem de inclusão. Cabe ressaltar que os objetos da classe *penalidade* só são exibidos através do método *getPenalidade* da classe *TipoPenalidade*. A atualização do objeto *TipoPenalidade* pode ser feita em, além de seus atributos, nos objetos *penalidades* relacionados a esta classe. Na Figura 67.g é demonstrado este tipo de atualização. Observe que neste trecho só é solicitada a atualização do objeto da classe *TipoPenalidade*, porém o FPOR estende esta operação persistente às *penalidades* relacionadas, tendo em vista que o relacionamento entre as classes prevê que esta operação seja em cascata. A mesma situação acontece na operação de exclusão, demonstrada no Figura 67.h, no qual só é necessária a exclusão do objeto da classe TipoPenalidade, sem necessitar a exclusão individual dos objetos referenciados ao objeto da classe *penalidade*, pois esta operação está definida no mapeamento objeto relacional como em cascata.

```
public class IntTipoInfracao {
    public static void main(String[] args) {
     try
     {
        // Cria Conexao
        MecanismoPersistente mp = new
              MecanismoPersistente( "usuario",
                                    "manager",
                                    "jdbc:oracle:oci8:@ora");
        // Define o mapeamento objeto relacional
        Persistencia.Persistencia.defineMapeamento(new
              ParseOracle(    "mapeamento",
                              "manager",
                              "jdbc:oracle:oci8:@ora"));
```

```
// Instancialização TipoInfracao
TipoInfracao ti = new TipoInfracao();
// Instancialização de Penalidade
Penalidade pen = new Penalidade();

// Carrega seus atributos
pen.setID(1);
pen.setValor(200);
pen.setDescricao("Duzentas UFIR's");

// Realiza a operação de incluir
pen.insere(mp);
ti.getPenalidade().add(pen);

// Cria Nova Penalidade
pen = new Penalidade();

// Carrega seus atributos
pen.setID(2);
pen.setValor(500);
pen.setDescricao("Quinhentas UFIR's");

// Realiza a operação de incluir
pen.insere(mp);
ti.getPenalidade().add(pen);

// Cria Nova Penalidade
pen = new Penalidade();

// Carrega seus atributos
pen.setID(3);
pen.setValor(1000);
pen.setDescricao("Mil UFIR's");

// Realiza a operação de incluir
pen.insere(mp);
ti.getPenalidade().add(pen);

// Carrega seus atributos
ti.setID(1);
ti.setCodigo(11);
ti.setNome("Excesso do Limite de Velocidade");
```

67.a

67.b

67.c

67.d

```
// Realiza a operação de incluir
ti.insere(mp);

// Consulta um objeto com criterio...
CriterioPesquisa cp1 = ti.criaCriterioPesquisa();
cp1.adicionaCriterio("ID","=","1");
if (ti.pesquisaUm(mp,cp1))
   System.out.println(ti);

// Criando outro objeto Tipo Infração...
ti = new TipoInfracao();
ti.setID(2);
ti.setCodigo(22);
ti.setNome("Estacionar em local proibido");
ti.getPenalidade().clear();

// Cria Nova Penalidade
pen = new Penalidade();
pen.setID(4);
pen.setValor(0);
pen.setDescricao("Carro rebocado");
ti.getPenalidade().add(pen);
ti.insere(mp);

// Criando outro objeto Tipo Infração...
ti = new TipoInfracao();
ti.setID(3);
ti.setCodigo(33);
ti.setNome("Avançar o sinal");
ti.getPenalidade().clear();

// Cria Nova Penalidade
pen = new Penalidade();
pen.setID(5);
pen.setValor(0);
pen.setDescricao("Carteira suspensa");
ti.getPenalidade().add(pen);

// Cria Nova Penalidade
pen = new Penalidade();
```

```
CriterioPesquisa cpen = pen.criaCriterioPesquisa();
cpen.adicionaCriterio("ID","=","1");
if (pen.pesquisaUm(mp,cpen))
   ti.getPenalidade().add(pen);
ti.insere(mp);

// Consulta varios objetos, sem criterio...
ti.pesquisa(mp);
while (ti.proximo(mp))
   System.out.println(ti);

// Consulta varios objetos, com criterio (ID>1)...
CriterioPesquisa cp2 = ti.criaCriterioPesquisa();
cp2.adicionaCriterio("ID",">","1");
ti.pesquisa(mp,cp2);
while (ti.proximo(mp))
   System.out.println(ti);

// Realiza a operação de atualização
pen.setDescricao("Atualizado");
if (ti.pesquisaUm(mp,cp1))
{
   System.out.println(ti);
   ti.setNome("Atualizado.");
   Iterator iti = ti.getPenalidade().iterator();
   while (iti.hasNext())
   {
      Penalidade p = (Penalidade) iti.next();
      p.setDescricao("Alterado");
   }
   ti.atualiza(mp);
   // Depois da atualizacao...
   System.out.println(ti);
}
// Pesquisa o objeto para verificar se foi realmente atualizado...
if (ti.pesquisaUm(mp,cp1))
   System.out.println(ti);
```

67.e

67.f

67.g

```
67.h
        // Deletando os objetos...
        ti.pesquisa(mp);
        while (ti.proximo(mp))
        {
           System.out.println(ti);
           ti.exclui(mp);
           // Objeto excluído.
        }
        // Pesquisando para saber se tem algum objeto...
        ti.pesquisa(mp);
        boolean aindaExiste = false;
        while (ti.proximo(mp))
           aindaExiste = true;
        if (!(aindaExiste))
           System.out.println("Nenhum registro encontrado.");
     }
     catch(Exception ex)
     {
        System.out.println("Problemas..."+ex);
        ex.printStackTrace();
     }
} // end main

} // end IntTipoInfracao
```

Figura 67 – Código Fonte IntPessoaFisica

4.2.7 Manuseio simultâneo de várias Classes e Objetos

Durante a execução de uma aplicação vários objetos de diferentes classes podem estar sendo editados direta ou indiretamente de forma simultânea. O FPOR deve ser capaz de administrar as diferentes classes de dados e suas diferentes instâncias em sincronia.

Apesar da classe *RecursoTransferencia* possuir somente um relacionamento com a classe *PessoaFísica*, a classe de apresentação *IntRecursoTransferencia* envolve várias classes na sua manipulação. Este fato tem origem na superclasse *Recurso* que tem relacionamento com a classe *Multa*. A classe *Multa*, por sua vez, tem relacionamentos com outras duas classes: *Veículo* e *TipoInfracao*. Para

enriquecer o experimento, o relacionamento do *Recurso* foi feito com uma especialização de *Multa*, denominada *MultaEletrônica*. Este enriquecimento demandou a incorporação da classe *Equipamento* em função de seu relacionamento com *MultaEletrônica*.

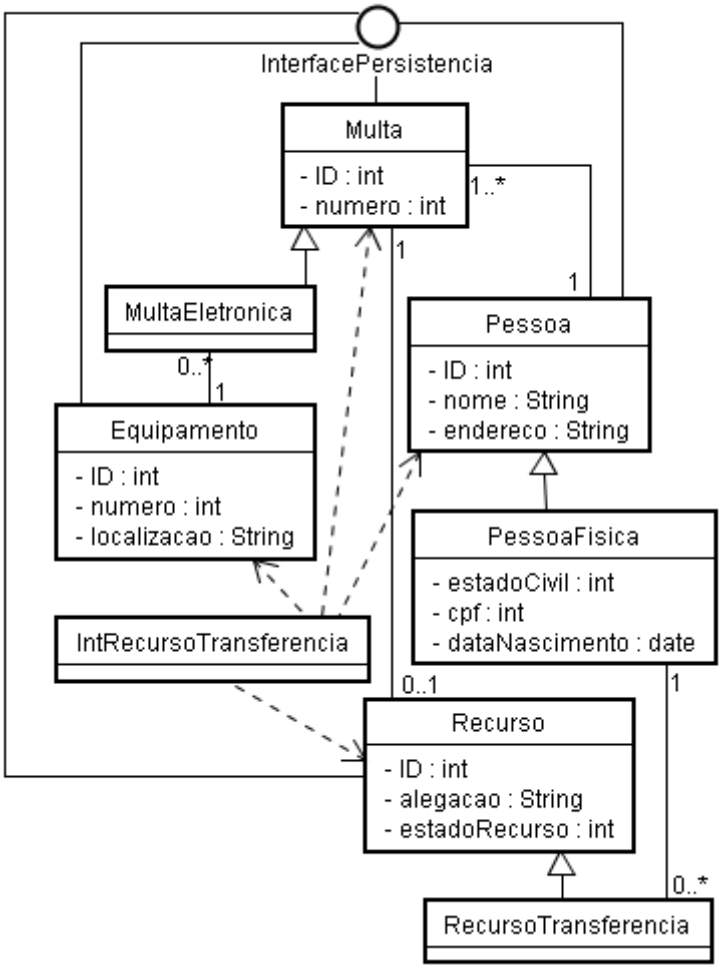


Figura 68 – Diagrama de Classes IntRecursoTransferencia

Considerando todas as classes envolvidas, contabilizamos o total de 7 classes com relacionamentos de diferentes cardinalidades manipuladas simultaneamente pelo FPOR.

A cada nova instância de uma classe persistente, mesmo sendo oriunda de um atributo complexo ou hierarquia, o FPOR verifica se a classe já consta em seu mapeamento e, em caso afirmativo, mapeia as características desta nova instância

da classe. Cada instância de classe tem seus valores individualizados no FPOR, permitindo o controle simultâneo de várias instâncias da mesma classe.

4.3 AVALIAÇÃO

Embora o estudo de caso tenha englobado seis classes de apresentação, 13 classes de negócio, 14 relacionamentos, três hierarquias e as mais diversas situações de desencontros objeto relacionais, não temos a pretensão de afirmar que esgotamos todas as possibilidades de um ambiente de desenvolvimento. Constatamos que para aplicações simples o FPOR resolveu de forma consistente as diferenças tecnológicas da orientação a objetos e do ambiente de banco de dados.

Nesta seção será feita uma avaliação qualitativa da adequação do FPOR aos requisitos especificados na seção 1.3.

4.3.1 Mapeamento Objeto Relacional

O mapeamento objeto relacional tem a qualidade de ser flexível. Apesar de não terem sido validadas todas as possibilidades de mapeamento objeto relacional possíveis neste modelo, destacamos as principais impedâncias resolvidas: (i) hierarquia, (ii) atributos multivalorados, (iii) relacionamentos, inclusive de cardinalidade muitos para muitos, (iv) operações em cascata e (v) atributos da mesma classe se referir a colunas de diferentes tabelas. Além destas questões, o FPOR propõe alternativa àqueles atributos de classes cujo mapeamento não é possível de ser feito diretamente ao ambiente de banco de dados.

O objetivo do mapeamento objeto relacional é harmonizar as características do modelo orientado a objetos com o modelo relacional. A correlação entre relacionamento de objetos e chaves estrangeiras, por exemplo, além de garantir a integridade referencial através do banco de dados, possibilita a liberdade de montagem e manutenção da estrutura de dados no ambiente de banco de dados, sem afetar o mapeamento da classe. O FPOR, em tempo de execução, se baseia

nos dados oriundos dos metadados do banco de dados para a montagem dos comandos SQL. Esta característica proporciona ao administrador de banco de dados a liberdade em alterar a chave estrangeira no modelo do ambiente de banco de dados, sem afetar sua correspondência com o relacionamento entre classes. Neste caso só é necessário atualizar o repositório do mapeamento objeto relacional a partir dos metadados do banco de dados.

No caso do framework HIBERNATE (HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1,2003), o mapeamento objeto relacional é feito a partir do modelo de classes persistentes, sem registrar as estruturas das tabelas no banco de dados. A visualização do modelo relacional no mapeamento fica prejudicada, tendo em vista que somente as classes persistentes têm o seu mapeamento realizado e simplesmente complementado com sua correspondência no modelo relacional. Os objetos de banco de dados não tem suas estruturas registradas no mapeamento do HIBERNATE, o que proporciona um maior grau de complexidade no processo de mapeamento. O FPOR entende que no mapeamento objeto relacional deve constar tanto o modelo de classe quanto o modelo relacional e a correspondência de seus elementos. Por exemplo, nos relacionamentos entre classes, o HIBERNATE sempre se baseia em alguma propriedade da classe oposta a associação para a resolução do relacionamento. Em contrapartida, o FPOR se utiliza da chave estrangeira implementada no banco de dados para resolver este relacionamento, pois neste ambiente é que deve ser resolvida a integridade referencial do relacionamento. Esta particularidade do FPOR proporciona ao relacionamento não estar somente vinculado a uma propriedade da classe, mas à uma ou mais colunas envolvidas nesta chave estrangeira no banco de dados, mecanismo pelo qual se implementa efetivamente o controle da integridade referencial.

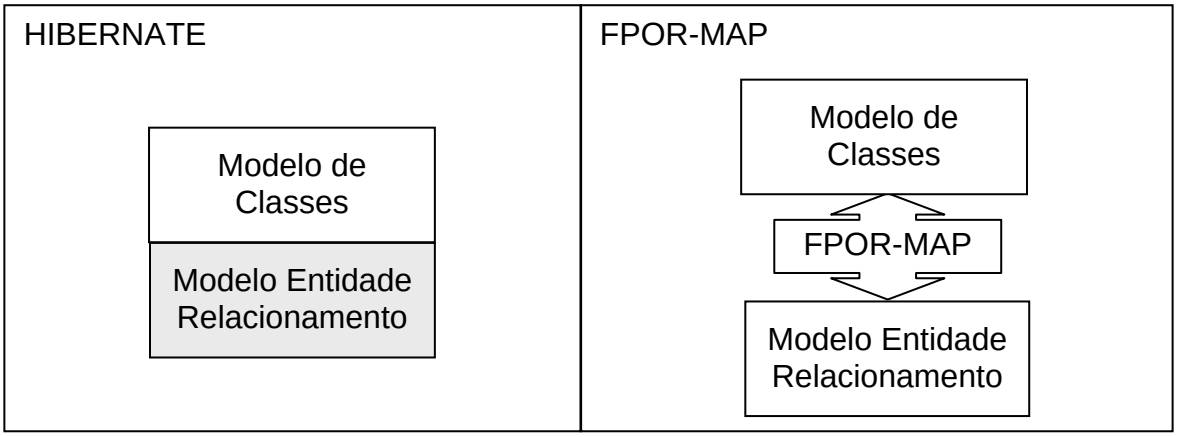


Figura 69 –Mapeamento Objeto Relacional HIBERNATE x FPOR-MAP

No framework CocoBase(2001) os relacionamentos são definidos no mapeamento objeto relacional através da criação de *links* (COCOBASE O/R MAPPING – PROGRAMMERS GUIDE v.1.,2001) pelo seu administrador, sem qualquer referência à chave estrangeira do banco de dados que garante a integridade do relacionamento. Neste framework, além do mapeamento das tabelas e colunas do banco de dados, devem ser definidas as cláusulas *where*, para as operações de consulta, atualização e exclusão nestas tabelas. O FPOR deduz a partir do seu mapeamento objeto relacional as cláusulas *where* necessárias a qualquer operação persistente solicitada a partir de um objeto.

A simplicidade no mapeamento objeto relacional alcançada pelo FPOR-MAP é uma propriedade a salientar. O HIBERNATE(HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1,2003) se utiliza de arquivos XML na definição do seu mapeamento objeto relacional, prevendo uma série de *tags* a serem definidas pelo seu usuário. Existe uma série de ferramentas para edição de documentos XML como o XDoclet, Middlegen e AndroMDA. Porém este fato não dispensa o conhecimento das *tags* pré-definidas pelo HIBERNATE, e o trabalho de incluir todos os elementos e características de todas as classes persistentes manualmente e correlacioná-las às estruturas do banco de dados através de tags complementares. Este modo de elaboração de um mapeamento objeto relacional, além de ser confuso, propicia erros, tendo em vista que se trata de um processo praticamente manual e exaustivo.

A facilidade da coleta automatizada das informações a respeito de estruturas de classes e dos objetos de banco de dados, através da reflexão e da consulta a metadados respectivamente, é um benefício proporcionado pelo FPOR-MAP aos administradores do repositório do mapeamento objeto relacional. Este melhoramento não consta em nenhuma das soluções de mercado que demandam o uso de um mapeamento objeto relacional.

Outra característica a ser ressaltada no mapeamento objeto relacional do FPOR é o acoplamento baixo em relação ao framework, tornando-o acessível a outras tecnologias de framework. O fato do mapeamento objeto relacional ser implementado em um pacote a parte torna possível o seu reaproveitamento por outros frameworks. Seu repositório pode ser facilmente migrado por diversas tecnologias de banco de dados, através de sua conversão para arquivos XML, apesar de não ter desenvolvido neste trabalho uma solução completa para esta facilidade.

Toda informação a ser recuperada do banco de dados deve estar retratada na classe persistente, tais como identificadores ou atributos que definam tipos de especialização de uma classe, quando todas as classes de uma hierarquia estiverem em uma única tabela. Esta condição pode ser considerada um ponto negativo do mapeamento objeto relacional do FPOR, pois a classe persistente é obrigada a implementar todos os atributos que forem necessários a sua correta recuperação do meio persistente. Ao invés, o mapeamento objeto relacional do FPOR poderia prever, quando houver o mapeamento de todas as classes de uma hierarquia em uma única tabela, que nesta tabela exista uma coluna que indentifique o tipo de especialização, dispensando assim o desenvolvedor de implementar este atributo na classe da hierarquia. Outro melhoramento que pode ser contemplado no FPOR é a geração automática de identificadores únicos para os objetos persistentes.

4.3.2 Transparência

O código fonte da classe persistente não é poluído com trechos destinados a serviços de persistência. Na Figura 60 pudemos demonstrar que em todo código não

se tem trechos destinados a implementação de persistência. A única prerrogativa exigida pelo FPOR é explicitar a implementação da interface *interfacePersistencia*.

O FPOR dispensa o desenvolvedor das classes de negócio do aprendizado de interfaces, padrões ou componentes de interação com o banco de dados previstos no uso do JDBC e EJB (PANDA,2003). Estes profissionais podem se concentrar na modelagem e implementação das classes de negócio sem tomar conhecimento de sua implementação no ambiente de banco de dados.

A Figura 71 apresenta um exemplo de manipulação de um objeto persistente, o método main da classe de interface IntCNH invocando métodos da *InterfacePersistência* sobre um objeto de negócio da classe persistente *CNH*. O modelo da classe CNH está demonstrado na Figura 70.

Estabelecer que serviços disponibilizados pelo FPOR devem estar nos próprios objetos persistentes é outra particularidade a ser ressaltada neste trabalho. Os frameworks citados anteriormente disponibilizam as mesmas operações de persistência de objeto através de classes específicas, cujos objetos são instanciados nas classes de interface para a recuperação e manipulação de objetos persistentes. A abordagem de implementação adotada pelo FPOR é disponibilizar os serviços de persistencia através das próprias classes persistentes, não havendo um terceiro componente que assuma a execução destas tarefas. Esta concepção é mais coerente e transparente já que o código da aplicação não incorpora objetos cuja incubência não esteja relacionada a atender o negócio que o aplicativo implementa.

As classes podem solicitar seus serviços de persistência de forma natural. Os métodos de persistência estarão disponíveis na própria classe, como se esta estivesse implementando os métodos previstos na interface *InterfacePersistencia* em seu código fonte, apesar destes métodos só existirem em seu *bytecode*. O programador não percebe que por trás destas manipulações existe uma arquitetura provendo as funcionalidades.

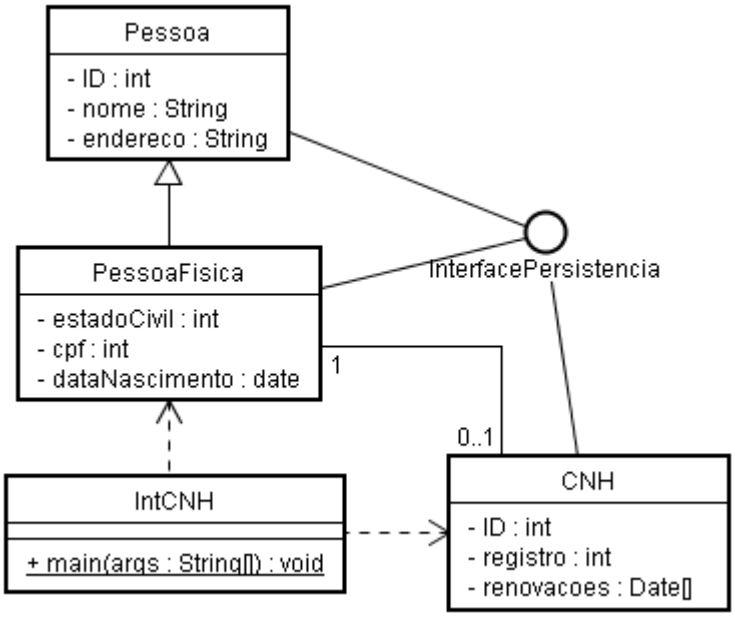


Figura 70 - Diagrama de Classes IntCNH

```
public class IntCNH {
public static void main(String[] args) {
try
{
    // Criando uma conexão...
    MecanismoPersistente mp =
            new MecanismoPersistente("usuario",
                                     "manager",
                                     "jdbc:oracle:oci8:@ora");
    // Define o mapeamento...
    Persistencia.Persistencia.defineMapeamento(
                  new ParseOracle(  "mapeamento",
                                    "manager",
                                    "jdbc:oracle:oci8:@ora"));
    // Instanciando o objeto persistente CNH...
    CNH cnh = new CNH();
    // Carrega seus atributos
    cnh.setID(1);
    cnh.setRegistro(100);
    Date[] d = new Date[10];
    java.util.GregorianCalendar gc =
                        new java.util.GregorianCalendar(1970,9,27);
    for (int i=0;i<10;i++)
    {
       gc.add(GregorianCalendar.DAY_OF_MONTH,1);
       d[i] = gc.getTime();
    }
    cnh.setRenovacoes(d);
    // Instancia outro objeto persistente, denominado PessoaFisica...
    PessoaFisica pf = new PessoaFisica();
    // Carrega seus atributos...
    System.out.println("Carregando os atributos...");
    pf.setID(1);
    pf.setNome("Fulado de Tal");
    pf.setEndereco("Rua Fim do Mundo, 999 casa 1");
    gc = new java.util.GregorianCalendar(1970,9,27);
    pf.setDataNascimento(gc.getTime());
    pf.setEstadoCivil(10);
    pf.setCpf(11111111);
```

71.e

```
71.a  // Relaciona PessoaFisica a CNH
      cnh.setPessoaFisica(pf);

71.b  // Realiza a operação de incluir...
      cnh.insere(mp);
      mp.efetiva();

      // Consulta um objeto persistente com criterio...
      MapeamentoClasse.CriterioPesquisa cp1 = cnh.criaCriterioPesquisa();
71.c  cp1.adicionaCriterio("ID","=","1");
      if (cnh.pesquisaUm(mp,cp1))
      {
         System.out.println(cnh);
      }

      // Consulta varios objetos, sem criterio...
      cnh.pesquisa(mp);
      while (cnh.proximo(mp))
            System.out.println(cnh);

      // Consulta varios objetos, com criterio (ID>1)...
      MapeamentoClasse.CriterioPesquisa cp2 = cnh.criaCriterioPesquisa();
      cp2.adicionaCriterio("ID",">","1");
      cnh.pesquisa(mp,cp2);
      while (cnh.proximo(mp))
71.d  {
         System.out.println(cnh);
      }
      // Realiza a operação de atualização
      cnh.setRegistro(cnh.getRegistro()+1);
      System.out.println("Atualizando o objeto...");
      if (cnh.pesquisaUm(mp,cp1))
      {
         System.out.println(cnh);
         cnh.setRegistro(cnh.getRegistro()+1);
         gc = new java.util.GregorianCalendar();
         cnh.getRenovacoes()[0] = gc.getTime();
```

```
        // Invoca o objeto pessoa física, pois é fundamental quando
        // CNH é atualizado no banco de dados (recuperação sob demanda).
        cnh.getPessoaFisica();
        cnh.atualiza(mp);
        mp.efetiva();
        // Exibe o objeto depois da atualizacao...
        System.out.println(cnh);
    }
    // Pesquisa o objeto para verificar se foi realmente atualizado...
    if (cnh.pesquisaUm(mp,cp1))
        System.out.println(cnh);
    // Deletando os objetos...
    // Refaz a pesquisa 1
    cnh.pesquisa(mp);
    while (cnh.proximo(mp))
    {
        System.out.println(cnh);
        // Invoca o objeto pessoa física, pois é fundamental quando
        // no processo de exclusão no banco de dados
        // (recuperação sob demanda).
        cnh.getPessoaFisica();
        cnh.exclui(mp);
        mp.efetiva();
        // Objeto excluído
    }
    // Pesquisando para saber se tem algum objeto...
    cnh.pesquisa(mp);
    boolean aindaExiste = false;
    while (cnh.proximo(mp))
         aindaExiste = true;
    if (!(aindaExiste))
        System.out.println("Nenhum registro encontrado.");
}
catch(Exception ex)
{
   System.out.println("Problemas..."+ex);
   ex.printStackTrace();
}
}
}
```

Figura 71 – Código fonte da Classe IntCNH

A Figura 71 apresenta um exemplo do emprego de objetos persistentes com o FPOR. Na Figura 71.b pode-se observar como um objeto depois de instanciado é de fato persistido no meio de armazenamento. A classe PessoaFisica relacionada à classe CNH, conforme Figura 71.a, não necessita ser incluída na base de dados, pois o relacionamento entre estas classes prevê a inclusão em cascata. Na Figura 71.c é exemplificada a recuperação de um objeto no meio persistente. Essa busca se dá para um único objeto (recuperação pela identificação do objeto, que é única) ou para uma coleção de objetos.

A geração dos comandos SQL consiste na combinação dos elementos do banco de dados a serem manipulados com os valores correntes das propriedades do objeto. Esta afirmação pode ser visualizada na Figura 72.

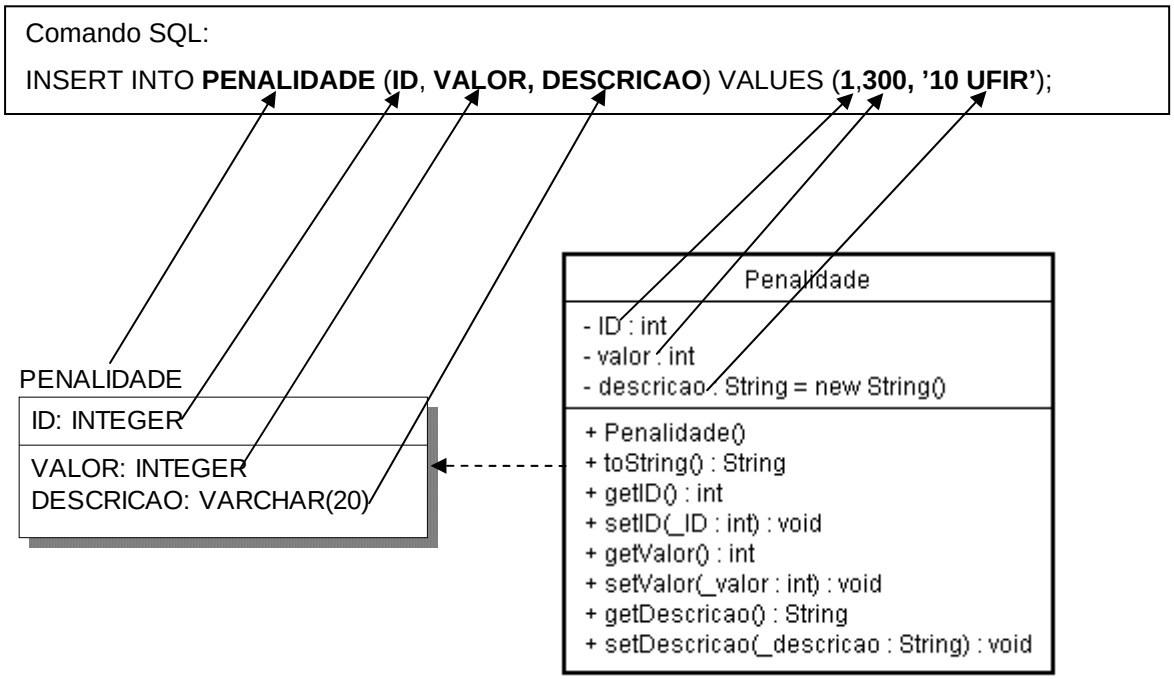


Figura 72 – Geração de comandos SQL

A utilização da reflexão dispensou o objeto persistente de informar os valores correntes de suas propriedades durante um serviço de persistência. O FPOR recolhe estas informações sem qualquer interação com o objeto persistente. Isto proporciona uma transparência do FPOR em relação ao objeto persistente na geração de comandos SQL.

Na Figura 72 temos a conversão dos tipos *int* e *Java.lang.String* em Java para os tipos *INTEGER* e *VARCHAR* no ambiente de banco de dados. A conversão de tipos de dados da linguagem Java para o ambiente SQL é feita de forma transparente pelo FPOR.

Outro aspecto abordado na compatibilidade de tipos de dados é o tratamento de coleções de objetos. Por exemplo, o atributo complexo *penalidade* da classe *TipoInfracao* é uma coleção de objetos da classe *Penalidade*. Este atributo pode ser declarado através de uma variedade de tipos de coleção prevista na linguagem JAVA, como por exemplo, *array*, *Collection*, *Map*, etc. A manutenção transparente deste tipo de atributo sempre é feita através de sua conversão para a classe java.util.list (THE COLLECTIONS FRAMEWORK TUTORIAL,2002).

Na Figura 73, a classe interface realiza a persistência de objetos persistentes da classe *MultaFormulario*. Quando realiza serviços de persistência sobre o objeto da classe *MultaFormulario*, o FPOR pode instanciar objetos das classes *TipoInfracao*, *Veiculo* e *PessoaJuridica* que estejam sendo referenciados. Estes objetos são denominados periféricos. Conforme a ação em cascata prevista para o serviço persistente, o FPOR realiza os serviços de persistência sobre os objetos periféricos de forma transparente.

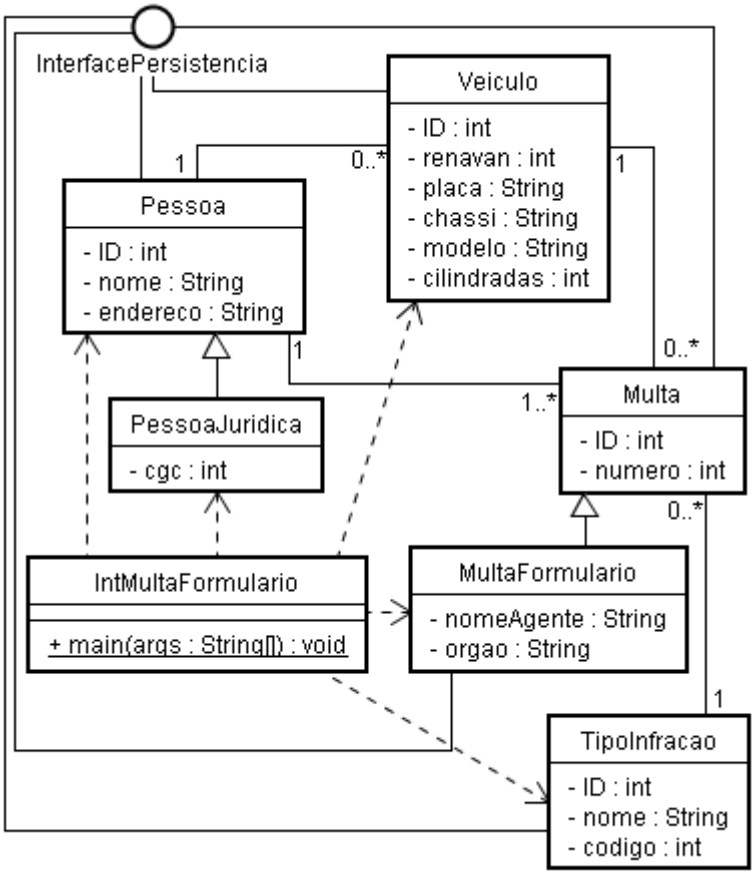


Figura 73 – Diagrama de Classes IntMultaFormulario

O atributo *renovacoes* da classe *CNH* é multivalorado e, portanto, tem um tratamento especializado, pois é armazenado no banco de dados em uma tabela a parte, denominada RENOVACOES_CNH, especificamente criada para atender este atributo. O FPOR realiza a persistência deste atributo de forma transparente, sem a necessidade de o programador atentar nesta peculiaridade.

4.3.3 Transparência no Tratamento de Coleções

Optamos por utilizar o conceito de cursor na recuperação de objetos pois avaliamos o seu manuseio como simples e de fácil implementação e entendimento. Tratamos a recuperação coletiva de objetos como uma seqüência de recuperações individuais, tornando-a semelhante à recuperação individual de objetos. Ao mesmo tempo, alcançamos uma considerável maleabilidade na montagem de critérios de consulta de objetos. Conceituamos que consultas *ad-hoc* que exigem um requinte

maior de pesquisa devem ser realizadas em ambientes próprios de consulta, denominados Data Warehouse (AMBLER,1998).

Não atentamos a questões como limite do tamanho de resultado de recuperação de objetos, gerenciamento de cursores abertos ou o desempenho de recuperação, pois concentramos nossas preocupações na forma de disponibilizar as informações desejadas. Estas questões podem ser evoluídas e melhor tratadas no FPOR.

4.3.4 Controle de estados

O controle de estados é baseado nos valores correntes do objeto, obtido pela reflexão, comparados aos valores recuperados do banco de dados. Apesar de não disponibilizar métodos que informem o estado atual do objeto em relação a sua persistência, o FPOR oferece o apoio ao desenvolvedor no tratamento adequado ao objeto, impedindo a perda de alterações efetuadas (vide seção 4.2.3).

4.3.5 Concorrência

O FPOR adota o controle de concorrência otimista. Realiza de forma eficiente, evitando colisões entre transações de banco de dados. Em todas as atualizações de objetos persistentes, o FPOR certifica se os dados do objeto permanecem inalterados desde sua recuperação da base e realiza o bloqueio dos registros necessários à realização do serviço. Dispensa assim a implementação de controles de *timestamp*.

4.3.6 Independência Arquitetônica

O FPOR obteve um baixo nível de acoplamento em relação ao mecanismo persistente e ao mapeamento objeto relacional. Conseqüentemente, o mecanismo persistente pode ser especializado em diferentes tecnologias como por exemplo GOP, ORDBMS e OODBMS. O mapeamento objeto relacional, de forma

semelhante, também pode ser especializado, sem comprometer o funcionamento do FPOR.

4.3.7 Acoplamento

Aplicações também apresentam um acoplamento baixo em relação ao FPOR. Este fato se deve à preservação do código fonte de classes persistentes intacto ao se utilizar o FPOR. Caso o framework seja descontinuado, não haverá impactos em relação à codificação das classes de negócio persistentes, tão somente às classes de interface, que deverão ser adaptadas a realizar serviços de persistência de outra forma.

5. CONCLUSÕES

Esta dissertação abordou o problema do descasamento de impedância (*impedance mismatch*) relacionado à persistência de objetos em bancos de dados relacionais.

Este tema é cada vez mais discutido tendo em vista que o processo de desenvolvimento de sistemas orientados a objeto atualmente tem recebido freqüentes investimentos em inovações tecnológicas e, em contrapartida, a modelagem relacional em SGBD é comprovadamente o método mais maduro e utilizado no armazenamento de dados de sistemas informatizados.

A ligação destas metodologias tem demandado dos programadores de linguagens orientadas a objetos o domínio da tecnologia de banco de dados para remediar suas impedâncias, elevando o custo e o prazo do desenvolvimento de aplicações. Agravando este quadro, as soluções adotadas proporcionam à aplicação orientada a objetos um acoplamento alto em relação ao banco de dados.

O problema neste trabalho foi abordado através da proposta de um framework para a realização da persistência de objetos em um banco de dados relacional.

As principais contribuições obtidas nesta dissertação estão descritas nos tópicos a seguir:

- uma pesquisa na literatura técnica a respeito deste assunto propiciou a coleta dos principais conceitos a serem considerados na implementação da persistência de objetos;

- foram apresentadas algumas soluções de persistência de objetos existentes, demonstrando seus pontos positivos e negativos;

- foi desenvolvido e apresentado um modelo de framework, denominado FPOR, para a implementação de persistência, enfocando três aspectos: mecanismo de persistência, mapeamento objeto relacional e serviços de persistência.

- o modelo foi implementado em linguagem Java e avaliado através de um estudo de caso;

- os serviços de persistência do FPOR são de fácil compreensão e manuseio. O programador é desobrigado do conhecimento da linguagem de manipulação de dados, proporcionando ganhos de produtividade na elaboração softwares orientados a objeto que necessitem de persistência.

- os serviços do FPOR são solicitados através das próprias classes persistentes. Porém, suas implementações não são afetadas por esta peculiaridade, dispensando o programador do aprendizado de interface, padrões e componentes de interação. Obtivemos esta transparência através da tecnologia de manipulação de *byte codes*.

- foi obtido um acoplamento baixo das aplicações em relação ao FPOR, pois as classes persistentes não são afetadas pela implementação dos serviços persistentes.

- foi desenvolvido um modelo para o mapeamento objeto relacional flexível, resolvendo as principais impedâncias tecnológicas. A maneira como foi elaborada a correspondência do modelo de classes com o modelo relacional possibilita manutenções transparentes, sem que as alterações de um modelo afetem o outro.

- foi elaborada uma ferramenta para o mapeamento objeto relacional, denominada FPOR-MAP, cujo manejo é simples e prevê a coleta automatizada de informações de estruturas de classes, através da reflexão, e de objetos de banco de dados, através de seu metadado.

- foi apresentado um processo de desenvolvimento de sistemas orientados a objetos utilizando-se o FPOR. Neste processo foi sugerida a criação de um novo perfil profissional, denominado administrador de classes e dados, responsável pela elaboração e controle dos modelos de classes e de diagramas de entidade e relacionamento, bem como a manutenção do mapeamento objeto relacional. Este novo perfil é estratégico, pois impede a duplicidade de classes e dados e garante a coerência entre estes elementos, tornando o processo de desenvolvimento bem controlado e estruturado.

- o FPOR proporciona uma independência arquitetônica em relação ao mecanismo persistente e ao mapeamento objeto relacional. Estes componentes podem ser adequados a diferentes tecnologias sem comprometer o FPOR, permitindo a portabilidade do serviço de persistência entre tecnologias.

6. TRABALHOS FUTUROS

Acreditamos que algumas evoluções e pesquisas, sugeridas nos tópicos a seguir, podem ser acrescentadas a este trabalho, evoluindo e ampliando o FPOR:

- submeter o FPOR a situações não previstas no estudo de caso como por exemplo: acessos simultâneos, grande volume de transações e recuperação de considerável quantidade de dados;
- introduzir no FPOR-MAP a percepção de inconsistências no mapeamento objeto relacional durante o seu uso;
- adotar uma notação gráfica para a equivalência objeto relacional e implementá-la na interface do FPOR-MAP (seção 3.1.3.7);
- implementar o tratamento de persistência de propriedades estáticas;
- introduzir comandos SQL semiprontos nos métodos implementados no processo de transformação de classes persistentes (seção 3.1.1.2). Esta nova técnica alivia o processamento do FPOR na montagem repetitiva de comandos SQL através da pesquisa do mapeamento objeto relacional em tempo de execução. Conseqüentemente, esta abordagem poderá proporcionar um aumento de desempenho;
- quando o FPOR atualiza os dados de um objeto persistente no banco de dados, não efetiva alterações realizadas em seus objetos relacionados, restringindo o escopo da operação somente ao objeto em questão e suas referências. Implementar um novo serviço, denominado *atualizaTudo*, que contemple, além da atualização do próprio objeto persistente, as alterações realizadas nos objetos associados;
- implementar o controle de recursos tais como o limite de tamanho do resultado e o gerenciamento de cursores abertos;

- aperfeiçoar o desempenho na recuperação de informações de objetos;
- implementar a geração automática de identificadores de objeto;
- avaliar a possibilidade de incorporar o FPOR em um container EJB do padrão J2EE (item 2.5.1.2);
- implementar as possibilidades de navegação existentes na álgebra relacional na montagem de critérios de pesquisa do FPOR;
- implementar a geração automática dos objetos de banco de dados a partir de um modelo de classes.
- evoluir o controle de transação para que quando um comando de *rollback* for acionado, todos as instâncias dos objetos associados às tabelas afetadas sejam atualizados com os valores mais recentes, antes do instrução.

BIBLIOGRAFIA

AMBLER, Scott.W. Análise e Projeto Orientados a Objeto: Seu Guia para Desenvolver Sistemas Robustos com Tecnologia de Objetos. v.2. Rio de Janeiro: IBPI Press, 1998, p. 88-351.

_____. *Concurrency Control*, AmbySoft Inc, Fev. 2003a. Disponível em: <http://www.agiledata.org/essays/concurrencyControl.html>. Acesso em: 30 nov. 2004.

_____. *Mapping Objects To Relational Databases*, AmbySoft Inc, Out. 2000a. Disponível em: <http://www.ambysoft.com/mappingObjects.html>. Acesso em: 30 nov. 2004

_____. *Implementing Referencial Integrity and Shared Business Logic*, AmbySoft Inc, Jan. 2003b. Disponível em: <www.agiledata.org/essays/referencialIntegrity.html>. Acesso em: 05 abr. 2004.

_____. *The Design of a Robust Persistence Layer For Relational Databases*, AmbySoft Inc, Nov. 2000b. Disponível em: <http://www.ambysoft.com/persistenceLayer.pdf>. Acesso em: 30 nov. 2004.

_____. *The Fundamentals of Mapping Objects to Relational Databases*, AmbySoft Inc, Jan. 2003c. Disponível em: <http://www.agiledata.org/essays/mappingObjects.html>. Acesso em: 31 mar. 2004

BALDWIN, Richard. T. Views, Objects, and Persistence for Accessing a High Volume Global Data Set. In: 20TH IEEE/11TH NASA GODDARD CONFERENCE ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSS'03), 2003, Estados Unidos da América. IEEE, 2003.

BARBIERI, Carlos. *Modelagem de Dados*. Rio de Janeiro:Infobook, 1994.

CHIBA, Shigeru., NISHIZAWA Muga. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators . In: 2ND INT'L CONF. ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING (GPCE'03) – LNCS 2830, pp.364-376, Springer-Verlag, 2003. Japão: Dept. of Mathematical and Computing Sciences – Tokyo Institute of Techology, 2003.

_____. *Getting Started with Javassist* - Disponível em: <http://www.csg.is.itech.ac.jp/~chiba/javassist/tutorial/tutorial.html> Acesso em: 30 ago. 2004

COCOBASE O/R MAPPING – BASIC CONCEPTS AND QUICK START GUIDE. Thought Incorporated, 2001. Disponível em: <http://www.thoughtinc.com> Acesso em: 04 dez 2004.

COCOBASE O/R MAPPING – PROGRAMMERS GUIDE v.1. Thought Incorporated, 2001. Disponível em: <http://www.thoughtinc.com/cber/ProgrammersGuideVol1.book.pdf> Acesso em: 09 dez 2004.

DEVEGILI, Augusto Jun. *Tutorial sobre Reflexão em Orientação a Objetos*. 2000 Florianópolis: Universidade Federal de Santa Catarina, 2000, Disponível em: <http://devegili.org/docs/reflexaooo.html>. Acesso em 30 nov. 2004.

GERBER, Samuel. Bytecode Inspection & Transformation. In: SEMINAR ENTERPRISE COMPUTING, 2003. Suíça: University of Applied Science Aargau, 2003. Disponível em <www.cs.fh-aargau.ch/~gruntz/ courses/sem/ws03/ByteCode.pdf> Acesso em: 30 nov. 2004.

HIBERNATE2 REFERENCE DOCUMENTATION VERSION 2.1.1, JBoss Inc., 2003. Disponível em: <http://www.hibernate.org/hib_docs/reference/en/pdf/hibernate_reference.pdf > Acesso em: 30 nov. 2004.

JDBC API DOCUMENTATION. Sun Microsystems Inc., 2002. Disponível em: <http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/index.html> Acesso em 30 nov. 2004.

KELLER, Wolfgang. Mapping Objects to Tables: A Pattern Language. In: PROCEEDINGS OF THE 2ND EUROPEAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMMING (EUROPLOP '97). SIEMENS TECHNICAL REPORT 120/SW1/FB. Munich, Germany: Siemens, 1997. Disponível em: <http://www.riehle.org/community-service/hillside-group/europlop-1997/p13final.pdf>. Acesso em: 30 nov. 2004.

KRUSZELNICKI, J. Persist Data with Java Data Object Part 2. *Java World*, 2002. Disponível em <http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-jdo.html> Acesso em: 30 nov. 2004.

LARMAN,Craig. Utilizando UML e padrões: uma introdução à análise e ao projeto orientado a objetos – Trad. Luiz A. Meirelles Salgado. Porto Alegre : Bookman, 2000

_____. _____. 2.ed. Porto Alegre:Bookman, 2004.

LERMEN, Alessandra de Lucena. Um Framework para Mapeamento de Aplicações Orientadas a Objetos a Bancos de Dados Relacionais/Objeto-Relacionais. In: SEMANA ACADÊMICA DO CPGCC, 3., 1998. Porto Alegre: Curso de Pós-

Graduação em Ciência da Computação - Instituto de Informática - Universidade Federal do Rio Grande do Sul, 1998. Disponível em: <http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana98/lermen.html>. Acesso em 30 nov. 2004.

MACHADO, Felipe Neri R., ABREU Maurício Pereira. *Projeto de Banco de Dados: Uma Visão Prática.* São Paulo: Editora Érica, 1995.

MACIEL, José Maurício Carré*. Mapeamento de Propriedades Dinâmicas de Objetos Descritas por Diagramas de Estado UML em Banco de Dados Relacionais*, Porto Alegre, 2001. 68 f. Dissertação(Mestrado em Informática) - Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre. 2001.

MAES, Pattie, Issues in Computational Reflection apud Maes, Pattie e Nardi, Daniele (eds.), Meta-Level Architectures and Reflection. 1988 North-Holland: *Elsevier Science Publishers*, 1988.

ORACLE8I SERVER ONLINE DOCUMENTATION. ORACLE Corporation, 2000. CD-ROM.

PANDA, Debu. J2EE for the DBA. *Oracle Technology Network*. 2003 Disponível em: <http://otn.oracle.com/oramag/oracle/03-may/o33j2ee.html> Acesso em 14 jan. 2004.

POLAK, Farup. ; ALHAJJ, Reda. Reengineering Relational Databases to Object-Oriented: Constructing the Class Hierarchy and Migrating the Data. In: EIGHTH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE'01) , out. 2001, Alemanha. IEEE, 2001.

PORTO, Fábio A.M. et al. *Persistent Object Synchronization with Active Relational Databases.* 1998. Rio de Janeiro: Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1998.

_____. *Object Life-Cycles in Active Relational Databases*. 1999. Rio de Janeiro: Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1999.

RAMAKRISHNAN Sudhakar. Don’t Be Stubborn When Persistence Is Concerned. *Oracle Technology Network*. 2003 Disponível em: <http://otn.oracle.com/oramag/webcolumns/2003/opinion/ramakrishnan_toplink.html> Acesso em: 30 nov. 2004.

RODRIGUES, Elisette Maria Figueiredo de Almeida. *Aplicando o Modelo MOF e o XMI para a Integração de Bancos de Dados Heterogêneos*. 2002. 82 f. Dissertação (Mestrado em Informática) – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2002.

RUMBAUGH, James. et al. *Modelagem e Projetos Baseados em Objetos.* Trad. Dalton Conde de Alencar. Rio de Janeiro: Editora Campus, 1994.

RUSSEL, C. *JAVA DATA OBJECTS Version 1.0.* Java Data Objects Expert Group – Sun Microsystems Inc, 2001.

SILVA, V. O papel dos frameworks J2EE. *B.NEWS*, 2003 Disponível em <http://www.b4unews.com.br/conteudo.php?ct=noticias&categoria=1&idnoticia=209&idcomunidade=1> Acesso em 09 jan. 2004.

SRINIVASAN, V.,CHANG, D.T. Object Persistence in Object-Oriented Applications, *IBM Systems Journal*, v.36, no.1, p.66 - 87, Estados Unidos da América: IBM Corp, 1997.

TANNEMBAUM, Andrew S. *Pratice Distributed Operating Systems*. New Jersey:Pratice-Hall, Inc. 1995.

THE COLLECTIONS FRAMEWORK TUTORIAL. Sun Microsystems Inc., 2002. Disponível em: <http://java.sun.com/docs/books/tutorial/collections/index.html> Acesso em 30 nov. 2004.

WHITE, James. Enterprise JavaBean Archiecture and Design Issues. In: 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE'01) , 2001, Canada: IEEE, 2001 Disponível em: <http://csdl.computer.org/comp/proceedings/icse/2001/1050/00/10500731.pdf> Acesso em: 30 nov. 2004.

WHITENACK, Bruce G., BROWN Kyle. Crossing Chasms: A Pattern Language for Object-RDBMS Integration “The Static Pattern”. KSC Knowledge Systems Corporation, 2002. Disponível em: <http://www.ksc.com/article5.htm> Acesso em: 30 nov. 2004.

ZIMBRÃO, Geraldo. Mapeamento Objeto-Relacional – Transforme um Modelo de Classes em um Modelo Relacional. *Revista SQL Magazine*, Rio de Janeiro: Neofício Editora v.5, ano 1, p. 28-33, Edição 5, 2003.