UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

EMULADOR DE SGBD ORIENTADO A OBJETOS

FERNANDO COLOMBO

BLUMENAU
2003

2003/2-14

FERNANDO COLOMBO

EMULADOR DE SGBD ORIENTADO A OBJETOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Alexander R. Valdameri – Orientador

BLUMENAU
2003

2003/2-14

EMULADOR DE SGBD ORIENTADO A OBJETOS

Por

FERNANDO COLOMBO

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Alexander R. Valdameri – Orientador, FURB
Membro:	Prof. Nome do professor, FURB
Membro:	Prof. Nome do professor, FURB

Blumenau, 25 de novembro de 2003

Dedico este trabalho aos meus dois filhinhos, Bernardo e Guilherme, embora a idéia que eles tenham sobre isto seja "aquela coisa que faz o pai não querer brincar com a gente".

AGRADECIMENTOS

À minha amada esposa Marília, por ter provido todo material e apoio necessários à conclusão deste trabalho (amor, carinho, compreensão, e é claro, comida e café).

À minha querida e energética sogra, Dona Tereza, por ter mantido meus filhos afastados de mim enquanto eu trabalhava.

À minha irmã Daniela e à minha cunhada Caroline, que também colaboraram com meus filhos.

Aos meus pais, Euclides e Cenira, por todo apoio e incentivo.

Ao meu orientador, Alexander Valdameri, que sempre esteve à disposição e compreendeu os atrasos.

Ao Sr. Nésio, meu chefe na Senior Sistemas, pelas oportunidades profissionais e científicas que sempre recebi naquela empresa.

RESUMO

Este trabalho descreve a implementação de um emulador de Sistema Gerenciador de Banco de Dados (SGBD) Orientado a Objetos. A implementação se baseia na norma ODMG 3.0, embora não seja compatível com ela. O software expõe as funcionalidades de um SGBD Orientado a Objetos e as implementa com auxílio de um SGBD relacional, usado de forma transparente. O texto mostra detalhes sobre mapeamento de modelos orientados a objetos para tabelas de um SGBD relacional, a linguagem OQL, a especificação do software e a integração do software com a plataforma Java. O software tem qualidade suficiente para uso em SIs pequenos, mas são necessárias extensões (explicadas no texto) para uso em aplicações de missão críticas.

Palavras chaves: banco de dados, orientação a objetos, SGBD, ODMG, mapeamento objeto-relacional, OQL.

ABSTRACT

This work describes the implementation of an object oriented Database Management System (DBMS). The implementation is based on the ODMG 3.0 specification, although it's not compatible with it. The software exposes the features of an object oriented DBMS and implements them by transparently using a relational DBMS. The text shows details about mapping object oriented models to relational DBMS tables, OQL language, software specification and software integration with Java platform. The software has enough quality to be used in small information systems, but extensions are required in order to support critical mission applications.

Keywords: database, object orientation, DBMS, ODMG, object-relational mapping, OQL.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

BNF – Backus-Naur Form

CASE – *Computer Aided Software Engineering* (Engenharia de Software Auxiliada por Computador)

JDBC – Java Database Connectivity

OQL – *Object Query Language* (Linguagem de Consulta de Objetos)

SGBD – Sistema Gerenciador de Banco de Dados

SGBDOO – SGBD Orientado a Objetos

SGBDOR – SGBD Objeto-Relacional

SGBDR – SGBD Relacional

SGDO – Sistema Gerenciador de Dados em Objetos

SQL – *Structured Query Language* (Linguagem de Consulta Estruturada)

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de software é em geral uma tarefa complexa, e entre os softwares mais difíceis de se desenvolver estão os Sistemas de Informação (SI). Evidentemente há SIs pequenos que podem ser desenvolvidos com ferramentas CASE tradicionais com esforço relativamente baixo. Mas as crescentes necessidades dos usuários dos SIs, a necessidade de manter os SIs atualizados tecnologicamente, e a grande concorrência que existe nesta área tornam o desenvolvimento da maioria dos SIs um desafio.

O conjunto de ferramentas e técnicas empregados na tarefa de desenvolver um SI influenciam diretamente a produtividade dos programadores e a qualidade final do SI. Conforme Stroustrup (1994, p. 362), "*The systems we construct tend to be at the limit of the complexity that we and our tools can handle*". E um dos principais aspectos destas ferramentas e técnicas é a forma com que os desenvolvedores mapeiam um modelo de realidade a uma representação formal que a máquina entenda.

Um SI costuma depender bastante de um SGBD, que se torna uma das peças que mais influenciam no resultado do desenvolvimento. Quanto melhores forem os recursos do SGBD para abstração da máquina e representação da realidade, melhor será o SI e mais produtivo será o desenvolvimento.

Embora o modelo orientado a objetos esteja consagrado como excelente método de abstração e de representação da realidade, ele ainda não é usado em sua essência, sobretudo no que diz respeito aos Sistemas Gerenciadores de Banco de Dados (SGBD).

Tradicionalmente, os SIs utilizam SGBDs relacionais (SGBDR). Embora existam SGBDs orientados a objetos em elevado grau de maturidade, eles ainda não estão acessíveis ao público em geral, na forma de produtos economicamente viáveis ou de softwares livres. Em paralelo a isto, as universidades e instituições de ensino em geral ainda dão mais enfoque aos SGBDRs. Sem dúvidas, a migração para SGBDs orientados a objetos será gradual, e haverão muitas soluções mistas, nas quais um SGBDR é usado em conjunto.

Neste sentido, foi feita a implementação de um emulador de SGBD orientado a objetos. O software agrega funcionalidades da orientação a objetos a um SGBDR, que é usado de forma transparente para programadores de SIs. Deste modo, os programadores de SIs se beneficiam da produtividade e qualidade propiciada pelo uso de um SGBD orientado a objetos, sem arcar com os custos de tal software.

1.1 OBJETIVOS

Este trabalho apresenta a implementação de um emulador de SGBD orientado a objetos, que usa um SGBDR de forma transparente para desenvolvedores de SIs. O software expõe todas as funcionalidades de um SGBD orientado a objetos conforme foi definido por Atkinson et al (1989).

O software se destina ao desenvolvimento de SIs que tenham características de aplicações *On-line Transaction Processing* (OLTP).

1.2 ESTRUTURA

O trabalho está dividido em 6 capítulos, sendo que esta introdução é o primeiro deles.

No capítulo 2, são mostradas noções de SGBD orientado a objetos, onde se detalham aspectos conceituais e teóricos de um software como este.

O capítulo 3 mostra técnicas de mapeamento entre modelos orientados a objetos e modelos relacionais. Estas técnicas são muito importantes porque o SGBDR é usado de forma massiva no trabalho.

Os capítulos 4 e 5 mostram o software propriamente dito. O capítulo 4 mostra a especificação do software, e o 5 mostra os principais aspectos da implementação.

Finalmente, o capítulo 6 mostra a conclusão e sugestões para trabalhos futuros.

1.2.1 Notações usadas

Há trechos deste texto que discorrem bastante sobre partes de programas ou comandos de determinada linguagem. Estas partes aparecem em uma fonte de tamanho fixo, com o fundo destacado. Exemplos:

a) A variável `x` contém o valor `'Olá Mundo'`;

b) O comando SQL `select * from Funcionarios` possui equivalente em OQL;

c) A sintaxe da união é *`operando`* `union` *`operando`*.

Quando o texto aparece em itálico, como no exemplo c, trata-se de um termo genérico, o qual assume-se que será substituído por outro na prática.

Há figuras que mostram diagramas feitos na linguagem *Unified Modeling Language* (UML) (OMG, 2000). Estas figuras referem-se sempre a softwares ou pedaços de software orientados a objetos.

Eventualmente são mostrados diagramas de tabelas de SGBDR que usam a notação mostrada na figura 1.

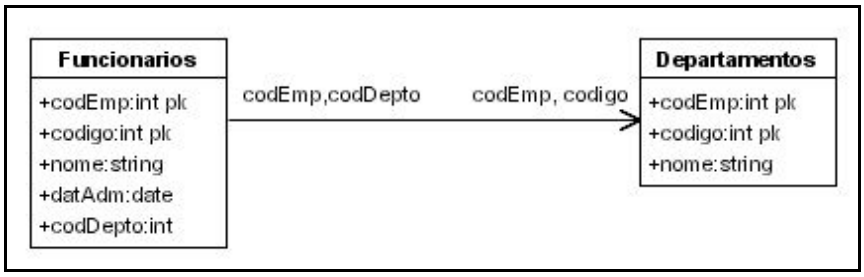

Figura 1 – Exemplo de diagrama de tabelas de SGBDR.

Este diagrama deve ser lido da seguinte forma:

a) existe uma tabela com nome **`Funcionarios`** que possui os campos **`codEmp`**, **`codigo`**, **`nome`**, **`datAdm`**, e **`codDepto`**; o tipo destes campos é respectivamente, **`int`**, **`int`**, **`string`**, **`date`** e **`int`**; a chave primária é formada pelos campos **`codEmp`** e **`codigo`**;
b) existe uma tabela com nome **`Departamentos`** que possui os campos **`codEmp`**, **`codigo`**, e **`nome`**; o tipo destes campos é respectivamente, **`int`**, **`int`**, e **`string`**; a chave primária é formada pelos campos **`codEmp`** e **`codigo`**;
c) existe uma chave estrangeira na tabela **`Funcionarios`** composta pelos campos **`codEmp`** e **`codDepto`**; esta chave referencia a tabela **`Departamentos`** nos campos **`codEmp`** e **`codigo`**.

2 NOÇÕES DE SGBD ORIENTADO A OBJETOS

Neste capítulo são mostradas noções de SGBD Orientado a Objetos (SGBDOO), com maior enfoque nos aspectos conceituais e teóricos. Também é mostrada uma introdução bastante sucinta a uma especificação de SGBDOO bastante usada.

2.1 CONCEITO

Um dos conceitos mais antigos de SGBD orientado a objetos é o *The object-oriented database system manifesto* (ATKINSON et al, 1989):

> An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an *ad hoc* query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

Segue uma breve descrição de cada uma das características. Primeiro as de SGBD:

a) persistência: certos dados (especificados pelo usuário) não devem ser perdidos quando a execução do sistema chega ao fim;
b) gerência de armazenamento secundário: a quantidade de dados não deve ficar limitada à memória do computador; deve haver um armazenamento secundário, que tenha capacidade para um grande volume de dados;
c) concorrência: mais de um usuário deve ser capaz de acessar o sistema de forma concorrente, sem que para isso os usuários tenham que controlar seus acessos manualmente;
d) recuperação: caso o sistema pare de executar de forma anormal (falha de hardware ou de software básico), resolvida a falha ele deve poder ser reiniciado com sucesso, e as operações realizadas até o momento da parada devem ter surtido o efeito desejado, observando-se a atomicidade das mesmas;
e) consultas *ad hoc*: usuários podem realizar consultas de forma simples, e quando tiverem necessidade, ou seja, sem programar as consultas previamente.

Agora, as características da orientação a objetos:

a) objetos complexos: são objetos compostos por outros objetos, formando uma árvore; inevitavelmente as folhas da árvore serão objetos simples, como números, textos, lógicos, datas, etc; exemplos de objetos complexos são conjuntos, listas, e tuplas;

b) identidade de objeto: a existência do objeto deve ser independente de seu valor, o que possibilita duas noções de equivalência: dois objetos podem ser idênticos (eles são o mesmo objeto) ou podem ser iguais (eles têm o mesmo valor);
c) encapsulamento: deve haver uma distinção clara entre a especificação e a implementação de uma operação, e deve ser possível proteger a implementação contra uso indevido;
d) tipos ou classes: tipos permitem que a definição dos dados fique separada da definição das operações, e forçam a corretude do programa porque permitem consistências em tempo de execução; classes são como tipos, com a diferença de que a própria classe é um objeto e pode ser manipulada (alterada ou passada como parâmetro);
e) herança: se dois ou mais tipos tiverem as mesmas características, ao invés de definí-las em cada tipo, basta definí-las num tipo compartilhado pelos demais, que os generaliza; isto evita duplicação de definições;
f) sobrecarga com ligação postergada: um mesmo nome pode ser usado para operações diferentes, dependendo da classe do objeto onde a operação é aplicada;
g) extensibilidade: o sistema vem com um conjunto de tipos pré-definidos. deve ser possível ao usuário definir novos tipos, e usá-los de modo que não haja distinção entre usar um tipo definido pelo sistema e um definido pelo usuário;
h) completude computacional: deve ser possível realizar qualquer função computável utilizando e transformando os dados; isto geralmente é obtido com o auxílio de uma linguagem externa.

2.1.1 Vertentes da indústria

É bastante seguro dizer que SGBDRs modernos representam o estado da arte em termos de confiabilidade, suporte a grandes volumes de dados, e atendimento de muitos usuários simultaneamente. Qualquer sistema de informação que desempenha missão crítica em empresas médias e grandes usam SGBDRs. Estes fatos fazem com que a indústria e o mercado tenham bastante respeito por SGBDRs, não só pelos softwares mas também pelo conceito em si, e isto representa uma força inercial muito grande que impede este conceito de ser substituído por outro de forma repentina e massiva.

Por outro lado, a comunidade de analistas de sistemas e programadores de aplicação apresentam o mesmo respeito pela OO, pelas conquistas que ela viabilizou por ser um

excelente método de abstração e de representação da realidade. Como a OO entra em conflito com o modelo relacional, e como os SGBDRs (que se baseiam no modelo relacional) estão fortemente consolidados, existem duas correntes que buscam trazer a OO para o mundo dos SGBDs:

a) a dos puristas, que buscam conceituar um SGBD que suporte a orientação a objetos de maneira independente do conceito de SGBDR;
b) a dos conservadores, que buscam trazer a orientação a objetos aos SGBDRs de maneira gradual e experimental, reduzindo ao mínimo possível eventuais conflitos com o modelo relacional.

A maioria dos puristas se baseiam na especificação *The object data standard: ODMG 3.0* (CATELL et al, 2000), desenvolvida pela organização Object Data Management Group (ODMG). Esta especificação é vista como a definição mais precisa da atualidade para um SGBDOO. É mostrada uma introdução desta especificação no tópico seguinte.

Os conservadores se baseiam na norma conhecida como SQL3 (ISO, 1999). A SQL3 é uma evolução da SQL/92 (ANSI, 1998), e introduz tipos estruturados, referências, herança de tipos e herança de tabelas. Ainda assim ela adere completamente à norma SQL/92, e mantém de forma bastante enraizada o modelo relacional. A SQL3 é considerada a definição mais precisa da realidade para um SGBDOR.

2.2 INTRODUÇÃO À NORMA ODMG 3.0

Neste tópico é apresentada uma breve introdução à especificação *The object data standard: ODMG 3.0* (CATELL et al, 2000). Ressalta-se que esta introdução não é um resumo da especificação, pois ela cobre somente aspectos relevantes para este trabalho, e alguns aspectos fundamentais não são mostrados.

A ODMG 3.0 é vista como a definição mais precisa de um SGBDOO por alguns autores, como Ramakrishnan (1998, p. 642) . Nota-se que a especificação não usa o termo SGBDOO, e sim o termo SGDO (Sistema Gerenciador de Dados em Objetos). Apesar de haver dois acrônimos, os autores e a comunidade em geral assumem que eles nomeiam a mesma coisa.

A especificação não define nem normatiza as características de SGBD, pois se assume que elas são amplamente conhecidas pelo público. Certos aspectos como transações, armazenamento secundário e recuperação são apenas citados ou definidos superficialmente.

2.2.1 Visão geral

O principal objetivo da especificação é prover portabilidade para o desenvolvimento de aplicações que necessitam de SGDOs. A especificação é suportada por representantes de quase toda indústria de SGDOs, de modo que ela se tornou um padrão verdadeiro. O trabalho feito na especificação também influenciou e sofreu influência de outras especificações, nas quais trabalham membros desta.

Um SGDO é um artefato que integra de forma transparente as capacidades de um banco de dados a uma linguagem de programação. Ele extende a linguagem de programação dando a ela persistência de dados, controle de concorrência, recuperação de dados, consultas associativas, e outras capacidades de banco de dados.

2.2.2 Modelo de objetos

Um SGDO deve suportar um conjunto mínimo de primitivas. As primitivas básicas são o objeto e o literal. O objeto é uma entidade que pode assumir vários estados, e o literal é um estado específico. Cada objeto tem uma identidade que o distingue dos demais. A distinção entre dois objetos existe mesmo que eles tenham o mesmo estado. Os literais são os estados em si, e portanto não têm identidade. O que distingue um literal de outro é o valor de cada um. Por questão de simplicidade, esta introdução usa o termo "objeto" tanto para objeto como para literal, exceto se o termo "literal" aparece no mesmo parágrafo.

Objetos são organizados em tipos. Todo objeto respeita um tipo, que determina os estados e o comportamento que o objeto pode assumir.

O estado de um objeto é definido por suas propriedades. Há dois tipos de propriedades: atributos e relacionamentos. Atributos dizem respeito somente ao próprio objeto, enquanto relacionamentos ligam o objeto com outros.

O comportamento de um objeto é definido por um conjunto de operações que podem ser executadas nele ou com ele.

2.2.2.1 Tipos e herança

Cada tipo tem dois aspectos: uma especificação e um conjunto de implementações. A especificação de um tipo determina características externas do tipo, que são independentes da linguagem de programação usada. Cada implementação do tipo define características internas, que devem implementar e respeitar a especificação do tipo. A implementação é feita numa

linguagem de programação escolhida, que deve ser suportada pela norma.

Os tipos se categorizam em classe, interface, e estrutura. A classe define o estado e o comportamento abstratos de seus objetos. Já a interface define apenas o comportamento abstrato, e a estrutura apenas o estado abstrato.

A herança de tipo é permitida. A herança é o relacionamento generalização-especialização que existe entre dois tipos, na qual um dos tipos é o mais genérico, e o outro é o mais específico. O tipo específico herda as definições de comportamento abstrato e estado abstrato do tipo genérico. Assim, um objeto do tipo específico é capaz de assumir todo estado e todo comportamento de um objeto do tipo genérico. Ressalta-se que um objeto do tipo específico pode ser usado em todo lugar que requer um objeto do tipo genérico, ou seja, a herança implica num relacionamento "é um" entre objetos.

É suportada herança múltipla de comportamento abstrato, mas não de estado abstrato. Ou seja, uma classe pode derivar de no máximo uma classe, e de várias interfaces. Uma interface não pode derivar de classes, mas pode derivar de várias interfaces. Este conceito é usado pelas linguagens Java e Smalltalk, para evitar ambigüidades e facilitar a gerência de memória. Ressalta-se que linguagens como C++ e Eiffel suportam herança múltipla de estado abstrato, e possuem mecanismos para resolução de ambigüidades.

2.2.2.2 Tipos primitivos

Os tipos primitivos servem para guardar e manipular valores básicos. São definidos os seguintes tipos primitivos:

a) long (inteiro de 32 bits com sinal);
b) long long (inteiro de 64 bits com sinal);
c) short (inteiro de 16 bits com sinal);
d) unsigned long (inteiro de 32 bits sem sinal);
e) unsigned short (inteiro de 16 bits sem sinal);
f) float (ponto flutuante com 32 bits);
g) double (ponto flutuante com 64 bits);
h) boolean (lógico);
i) octet (inteiro de 8 bits sem sinal);
j) caractere;
k) string (seqüência de caracteres);
l) enumeração (gerador de tipo que define valores discretos).

Não é permitida herança envolvendo tipos primitivos, e deve-se assumir a pré-existência de todos os objetos de cada um destes tipos (ou seja, não é possível instanciar objetos destes tipos).

2.2.2.3 Coleções

A norma deixa pré-definidos tipos especiais para coleções de objetos, que são muito usadas como principal meio de acesso aos objetos, e recebem tratamento especial na linguagem de consultas que será apresentada em seguida.

São definidos os seguintes tipos para coleções de objetos:

a) conjunto, que mantém uma coleção não-ordenada de elementos, e que não permite que mais de um elemento tenha determinado valor;
b) sacola, que mantém uma coleção não-ordenada de elementos, permitindo que mais de um elemento tenha determinado valor;
c) lista, que mantém uma coleção ordenada de elementos, cujo tamanho varia conforme elementos são adicionados ou removidos;
d) matriz, que mantém uma coleção ordenada de elementos, cujo tamanho não varia nas operações de adicionar ou remover elementos;
e) dicionário, que mantém uma coleção não-ordenada de pares chave-valor, e que não permite chaves duplicadas.

2.2.2.4 Tempo de vida dos objetos

Quanto ao tempo de vida, um objeto pode ser transiente ou persistente. Objetos transientes são armazenados no espaço de objetos definido pela linguagem de programação usada, que geralmente se constitui em memória volátil, como a memória RAM do computador. Objetos persistentes são armazenados no espaço de objetos definido pelo SGDO, que geralmente se constitui em memória não-volátil, como um arquivo que fica no disco rígido.

Quando um objeto é apenas criado, ele por definição é transiente. A forma como um objeto se torna persistente depende da linguagem usada. O SGDO suporta uma operação que tem a finalidade de persistir um objeto, e geralmente esta operação é disponibilizada na forma de uma API.

O tempo de vida de um objeto é independente de seu tipo. O tipo pode conter algumas instâncias que são transientes e outras que são persistentes. O comportamento e o estado

abstratos definidos no tipo valem tanto para os objetos transientes como para os persistentes, sendo possível usar objetos transientes e persistentes ao mesmo tempo, numa mesma operação. Por exemplo, é possível fazer a concatenação de uma lista contendo objetos transientes, com outra contendo objetos persistentes. O resultado será uma lista contendo tanto objetos transientes como persistentes.

Quando um objeto não é mais necessário, ele deve ser destruído de forma explícita.

2.2.2.5 Controle de concorrência e travamentos

A norma permite que seja feito acesso concorrente aos objetos persistidos no espaço de armazenamento do SGDO, e define o uso de um método convencional para isto, baseado em travamentos. São suportados os seguintes tipos de travamentos:

a) travamento para leitura, que pode ser obtido por mais de um processo ao mesmo tempo, desde que não esteja em vigência um travamento de gravação;
b) travamento para gravação, que pode ser obtido por no máximo um processo, desde que não esteja em vigência um travamento de outro processo;
c) travamento para atualização, que evita um tipo de deadlock comum na qual dois processos obtém inicialmente um travamento para leitura, e depois competem por um travamento para gravação.

A norma define que os travamentos podem ser obtidos de forma implícita (toda vez que uma operação de leitura ou gravação se faz necessária) ou explícita (através de ativações de operações do SGDO que têm a finalidade de efetuar um travamento).

Os travamentos são mantidos pelo tempo em que a transação estiver em vigência, e são removidos assim que a transação acaba.

2.2.2.6 Transações

Uma transação é um bloco de operações executado sobre os requisitos ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Atomicidade implica que ou todas as operações do bloco são realizadas, ou nenhuma é. Consistência implica que o bloco deve iniciar sua execução com a base de dados consistente, e após o término, a base de dados deve continuar consistente. Isolamento implica que as alterações feitas pelas operações só são visíveis a outros processos após o bloco terminar sua execução com sucesso. Durabilidade implica que as operações feitas no bloco têm efeitos duradouros.

O nível de isolamento definido pela norma é o da serialização, que diz que o resultado de transações concorrentes deve ser igual ao resultado que seria obtido se cada transação rodasse sozinha, uma após a outra.

A norma define uma interface chamada Transaction, que possui as operações `commit()` e `abort()` entre outras. A operação `commit()` confirma as alterações feitas durante a transação, enquanto `abort()` descarta tais alterações e deixa a base no estado em que se encontrava quando a transação se iniciou. Estas duas operações causam a finalização da transação e a remoção de todos os travamentos que foram criados pelo processo na vigência da transação.

2.2.3 Linguagem de definição de objetos

A *Object Definition Language* (ODL) serve para especificar tipos. Ressalta-se que a especificação de um tipo é a parte externa dele, que é independente da linguagem de programação adotada.

A ODL não possui meios para definir a implementação dos tipos. Isto deve ser feito com uma linguagem de programação suportada pela norma (as linguagens suportadas serão vistas mais adiante). Tipicamente, um fonte em ODL é submetido a uma ferramenta que gera fontes na linguagem escolhida. Os fontes gerados já são capazes de manter o estado dos objetos, sendo necessário apenas escrever o corpo dos métodos.

O quadro 1 mostra um exemplo de fonte em ODL.

```
module exemploODL {

  class Pessoa (extent Pessoas) {
    attribute string nome;
    attribute date datNas;
  };

  class Funcionario extends Pessoa (extent Funcionarios) {
    attribute float salario;
    relationship set<Dependente> deps inverse resp;
  };

  class Dependente extends Pessoa {
    relationship Funcionario resp inverse deps;
  };

};
```

Quadro 1 – exemplo de um fonte ODL.

2.2.4 Linguagem de consulta de objetos

A linguagem de consulta de objetos se chama *Object Query Language* (OQL). Trata-se de uma linguagem simples, mas poderosa, sendo capaz de expressar:

a) cálculos matemáticos;
b) operações com texto;
c) operações com datas;
d) operações em listas (concatenação, inversão e seleção de elemento);
e) operações em coleções (união, intersecção, excetuação, e outras);
f) filtragem de objetos;
g) ordenação e hierarquização de objetos;
h) operações relacionais entre objetos.

A OQL foi concebida para ser semelhante à SQL, porém sem determinadas restrições encontradas na última, e com adições voltadas para a OO. Apesar disso, ressalta-se que a OQL não é compatível com a SQL em termos de sintaxe dos comandos. Ou seja, o porte de fontes na linguagem SQL para OQL exige que a maioria dos comandos seja reescrito.

Por ser uma linguagem voltada para consultas, um comando OQL pode ser qualquer expressão que resulte num valor. Por exemplo, `1 + 5` é um comando OQL válido, e resulta em `6`. Além disso, como não existe o conceito de tabela, o comando `select` atua em coleções. Assim, o comando `select nome from Pessoas` itera por cada objeto da coleção `Pessoas`, e coloca o valor do atributo `nome` na lista resultante.

O comando `select` é desnecessário em muitos casos. Por exemplo, se for necessário obter a coleção de pessoas, uma forma é fazer `select p from Pessoas p`, e outra é fazer simplesmente `Pessoas`. Estes dois comandos têm o mesmo resultado.

Algumas construções ambíguas no SQL foram corrigidas na OQL. Por exemplo, em SQL, o comando `select count(*) from Pessoas` retorna sempre uma linha, enquanto o comando `select nome from Pessoas` retorna várias linhas, uma para cada registro da tabela `Pessoas`, sendo que a única coisa que mudou foi a expressão que precede a cláusula `from`. Em OQL, o primeiro comando corresponde a `count(Pessoas)`, e o segundo corresponde a `select nome from Pessoas`. A pessoa que lê cada comando tem uma noção bem mais clara do resultado de cada um.

Uma grande vantagem da OQL, vinda da orientação a objetos, é a capacidade de navegar através dos relacionamentos entre objetos. Para exemplificar, seja o seguinte

comando: obter a relação dos funcionários que mostre em qual departamento cada funcionários trabalha. Em SQL, o comando é este:

```
select
  f.nome, d.nome
from
  Funcionarios f, Deptos d
where
  f.codDep = d.codigo
```

Em OQL, o comando fica assim:

```
select nome, depto.nome from Funcionarios
```

Observa-se que `depto` é uma referência para um objeto que representa o departamento do funcionário. Esta referência existe porque as classes `Funcionario` e `Departamento` estão relacionadas. O caractere `.` que liga o termo `depto` ao termo nome tem semântica idêntica ao `.` nas linguagens C++, Java e Pascal: ele indica o acesso a um membro do objeto (no caso, o atributo `nome`). Esta notação elimina boa parte das junções em comandos de consulta.

A OQL tem várias operações para coleções de objetos. A seleção de colunas é apenas uma delas. É possível realizar a concatenação, união, intersecção, e excetuação de coleções, e o interessante é que estas operações podem ser feitas sem a cláusula `select`. Por exemplo, seja o seguinte comando: obter a contagem dos itens de serviço somados aos itens de mercadoria. Em SQL o comando fica assim:

```
select count(*) from (
  select 1 from ItensServico
  union all
  select 1 from ItensMercadoria)
```

Em OQL, o comando é este:

```
count(ItensServico || ItensMercadoria)
```

Uma operação interessante é a escolha de um elemento da coleção. Por exemplo, seja o seguinte comando: obter o nome do funcionário mais velho. Em SQL, o comando fica assim:

```
select nome, datNas from Funcionarios order by 2 desc
```

Este comando tem o inconveniente de resultar numa relação contendo todos os funcionários, e além disso ele traz uma informação que não foi solicitada: a data de nascimento do funcionário. Em OQL, o comando é este:

```
last(select nome from Funcionarios order by datNas)
```

Este comando traz somente aquilo que foi solicitado. Observa-se que o banco tem melhores condições de otimizar uma consulta se souber exatamente aquilo que se deseja. Um

SGDO bem implementado não criará cursores, nem realizará classificação (*sorting*) para executar este comando, diferente do que acontece com o comando submetido ao SGBDR, que obriga o banco a fazer estas operações.

2.2.5 Integração a linguagens de programação

A norma define as seguintes funcionalidades que devem ser fornecidas por uma linguagem de programação e pelo ambiente de execução desta linguagem:

a) a implementação dos métodos;
b) a execução dos métodos quando invocados a partir de consultas;
c) a criação (instanciação) de objetos;
d) o armazenamento de objetos transientes;
e) a ativação de operações para persistir objetos;
f) a ativação de operações relacionadas à transações.

A norma formaliza a integração com as linguagens C++, Java e Smalltalk. Outras linguagens simplesmente não são suportadas pela especificação, e as funcionalidades acima citadas precisam ser implementadas por uma destas linguagens, para que se confirme a aderência à norma.

Esta dependência explícita que o SGDO tem de uma linguagem de programação traz o benefício de que, ao processar uma consulta, o SGDO pode chamar métodos escritos na linguagem escolhida.

2.3 FUNCIONALIDADES ADICIONAIS

Além das funcionalidades de SGBDOO descritas até o momento, propõe-se certas funcionalidades adicionais, descritas neste tópico. Estas funcionalidades foram trazidas de linguagens de programação orientadas a objetos modernas como Java e C#.

2.3.1 Destruição automática de objetos

O SGBDOO deve destruir um objeto automaticamente quando ele detectar que tal objeto não é mais necessário. Um objeto deixa de ser necessário quando torna-se impossível consultá-lo, e isto acontece quando não é possível navegar até o objeto através de referências.

Objetos podem ser referenciados por outros objetos, ou por coleções globais, definidas em detalhes no próximo tópico. Coleções globais contém referências para objetos e a coleção em si não pode ser destruída. Desta forma, usuários expressam o desejo de manter um objeto

(ou seja, de que ele não seja destruído automaticamente) ao fazerem o objeto ser referenciado por uma coleção global.

Também há o caso de objetos que são fortemente associados a outros objetos, tais como os itens de um pedido. Estes objetos não precisam estar em coleções globais para que sejam mantidos, pois há um outro objeto que os referencia e este está numa coleção global. Em geral, não precisa sequer haver uma coleção global definida para tais objetos, pois geralmente eles não são manipulados separadamente do objeto referenciador.

2.3.2 Coleção global

Uma coleção global é semelhante a uma entidade que possui referências diretas para objetos de determinada classe. A coleção lembra fracamente uma tabela de SGBDR, pois coleções globais são os principais meios de se consultar objetos. Por exemplo, o comando s`select nome from Funcionarios` está atuando na coleção global `Funcionarios`.

As coleções globais para objetos de uma classe são definidas junto com a classe, pelo analista que a modelou. Pode haver mais de uma coleção global para determinada classe. A classe `Funcionario`, por exemplo, pode ter uma coleção chamada `Funcionarios`, que é o cadastro geral dos funcionários da empresa, e outra chamada `Demitidos`, que mantém os funcionários que foram demitidos. O mesmo objeto `Funcionario` pode estar presente em uma destas coleções, em ambas, ou em nenhuma delas, conforme a necessidade do usuário.

O usuário pode, a qualquer momento, adicionar objetos na coleção ou remover dela. A remoção de um objeto de uma coleção não implica na destruição do mesmo, pois ele pode estar presente em outra coleção. O fato de um objeto não estar presente em qualquer coleção também não implica em sua destruição, pois ele pode estar sendo referenciado por outro objeto.

As definições apresentadas até agora se aplicam a referências fortes. Existem mais dois tipos de referência: suave e fraca. Detalhes sobre os tipos de referência são mostrados a seguir.

2.3.3 Referência forte, suave, e fraca

Há três tipos de referência:

a) forte, que sempre impede que o objeto seja destruído;
b) suave, que impede que o objeto seja destruído automaticamente, mas não que ele seja destruído explicitamente;

c) fraca, que não impede que o objeto seja destruído.

Se o objeto for atingível por uma cadeia de referências fortes que se inicia em uma coleção global, existe a garantia de que ele não será destruído, nem que isto ocorra de forma explícita. Deste modo, referências fortes impõem integridade referencial.

Se o objeto não for atingível por uma cadeia de referências fortes, e houver uma ou mais referências suaves para ele, existe a garantia de que ele não será destruído automaticamente, mas ele pode ser destruído explicitamente. Geralmente tais objetos precisam passar por algum processamento antes que sejam destruídos.

Se o objeto não for atingível por uma cadeia de referências fortes, nem houverem referências suaves para ele, e houver uma ou mais referências fracas para ele, o objeto pode ser destruído automaticamente. Referências fracas servem para otimizar processos, no sentido de que o processo poderia ser feito sem o objeto, mas é feito com melhor performance com o objeto. Referências fracas também oferecem um meio de testar a utilidade de um objeto.

Quando um objeto é destruído, todas as referências suaves e fracas que haviam para ele ficam nulas automaticamente. Observa-se que uma referência nula não é inválida nem mal-formada.

O tipo da referência é determinado na associação e na coleção global. Por exemplo, uma coleção global pode usar referências fracas, que não impedem que os objetos por ela referenciados sejam destruídos.

3 TÉCNICAS DE MAPEAMENTO OBJETO-RELACIONAL

Como este trabalho implementa um SGBDOO através de um SGBDR, ele faz uso de técnicas de mapeamento de objetos para o modelo relacional, composto basicamente por variáveis de relação (tabelas), tuplas (registros) e relacionamentos (chaves estrangeiras). Este capítulo apresenta as principais técnicas de mapeamento existentes.

Este texto se baseia no trabalho de Ambler (2003), na obra *The fundamentals of mapping objects to relational databases*.

3.1 PREMISSAS DO MAPEAMENTO OBJETO-RELACIONAL

3.1.1 Características do armazenamento de objetos

Como a orientação a objetos foi implementada com sucesso inicialmente nas linguagens de programação, elas podem ser usadas como base para se estudar o armazenamento de objetos.

Os ambientes de execução de linguagens de programação orientadas a objetos armazenam todos os seus objetos na memória RAM. Cada objeto ocupa um intervalo contínuo de bytes, que armazena todo estado do objeto. Este intervalo é chamado de bloco de memória, e não se admite a intersecção dos intervalos de dois blocos.

Se o tipo de um objeto é uma classe que deriva de outras, o tamanho do bloco é suficiente para armazenar os estados de todas as classes pertencentes à hierarquia. Nestes casos o bloco é dividido internamente em sub-blocos, cada um guardando o estado da classe correspondente. Isto implica que podem surgir dois objetos que pertencem a uma mesma classe base (ou seja, compartilham certos estado e comportamento abstratos) e apesar disso têm formatos internos diferentes.

Referências para objetos são implementadas através de ponteiros, ou seja, a referência para um outro objeto é um número que corresponde ao endereço da memória do bloco do objeto referenciado. Como dois objetos não podem ocupar o mesmo bloco, o endereço também serve para implementar a identidade dos objetos, nas linguagens que suportam este conceito.

3.1.2 Erros de mapeamento

A semelhança entre as definições de classe e tabela provoca a intuição de que uma classe do modelo orientado a objetos pode ser mapeada para uma tabela do modelo relacional,

e em conseqüência, um objeto pode ser mapeado para um registro, e a identidade dos objetos pode ser mapeada para a chave primária da tabela. Mas isto se constitui num erro pelos seguintes motivos:

a) classes suportam herança, enquanto tabelas não suportam;
b) classes suportam o relacionamento n-para-m, enquanto tabelas não suportam;
c) objetos suportam referências entre si, enquanto registros não suportam;
d) objetos têm identidade independente de seu estado, enquanto registros não têm;
e) objetos que pertencem à mesma classe podem ter estados abstratos diferentes, enquanto registros que pertencem à mesma tabela sempre têm o mesmo estado abstrato.

O resultado é que o mapeamento puro e simples de classes para tabelas gera bases de dados muito propensas ao aparecimento de inconsistências.

3.1.3 Identidade dos objetos

A identidade de um objeto é um valor que identifica aquele objeto entre os demais objetos existentes. Para ser bem sucedida, a implementação da identidade deve observar as seguintes premissas ou recomendações:

a) é fundamental que a identidade de um objeto independa de seu estado, ou seja, ela não pode ser formada por valores de atributos definidos na classe; em outras palavras, a identidade não pode ter qualquer sentido para o negócio do SI;
b) é desejável que um objeto não tenha mais de uma identidade, pois se isto fosse permitido seria difícil determinar se duas identidades quaisquer se referem ou não ao mesmo objeto;
c) é desejável que todas as identidades (ou seja, as identidades de todos objetos), sejam do mesmo tipo e compartilhem um único espaço de valores, para facilitar a comparação de duas identidades, e também a implementação de caches e de controles de concorrência;
d) é desejável que um objeto nunca mude de identidade, ou que mude somente em ocasiões muito raras e controladas (como numa mudança no esquema ou restauração de *backup*).

A única forma de mapear as identidades para atender os requisitos acima é criar um mecanismo de geração de identidades por demanda, que é chamado toda vez que um objeto é instanciado. Para que isto não se torne um gargalo de performance, usa-se geradores de

seqüência ou campos auto-incrementados, comuns em SGBDRs. Outra solução é fazer os processos solicitarem faixas de identidades ao invés de identidades isoladas, de modo que uma nova solicitação só é feita quando a faixa corrente for exaurida.

Para garantir uniformidade e simplicidade na implementação de caches e de controles, as identidades devem ser números do mesmo tamanho, que pode ser 64, 96, ou 128 bits. O tamanho do número depende da quantidade de objetos envolvidos e do mecanismo de geração do mesmo. Este número costuma ser chamado de OID em trabalhos relacionados. OID é uma sigla para *Object Identity*.

Cada OID é armazenado no SGBDR junto com o objeto por ele identificado. Por um lado isto representa maior consumo de disco, memória, banda de comunicação e processamento em geral. Mas esta abordagem traz muitos benefícios implícitos, pois o OID é uma entidade discreta e bem controlada. Ele é menor do que a maioria das chaves (especialmente as compostas), cada OID novo tem um valor maior do que o anterior, de modo que freqüentemente encontram-se faixas de OIDs consecutivos, e o valor dos OIDs nunca muda. Estas características se refletem nos seguintes benefícios:

a) a implementação de caches não precisa conhecer a definição das chaves primárias, que pode ser diferente de tabela para tabela;
b) muitas operações de leitura podem ser feitas apenas pelo OID ao invés de se usar uma chave composta, o que diminui o tamanho do comando de busca e a quantidade de parâmetros que trafegam do cliente para o banco – o mesmo vale para a cláusula `WHERE` de comandos `UPDATE`;
c) os índices baseados em OIDs são bastante otimizados, especialmente em SGBDRs modernos, que são projetados para aproveitar características de OIDs;
d) chaves estrangeiras para tabelas que têm OID sempre referenciam o OID ao invés de uma chave formada por atributos da classe, o que diminui o tamanho dos registros que têm muitas chaves estrangeiras;
e) a grande maioria das junções entre tabelas é feita através de OIDs;
f) se for necessário fazer algum reparo na base de dados, OIDs são úteis como um meio uniforme de se identificar registros defeituosos.

Por estas razões, muitas pessoas inicialmente condenam o uso de OIDs, mas depois de alguma prática se tornam defensoras deste mecanismo. Ainda assim, ressalta-se que os OIDs prejudicam a performance em alguns cenários. O correto é afirmar que o uso de OIDs em SGBDRs pode ser prejudicial ou benéfico para a performance, conforme as características de

cada situação.

3.2 MAPEAMENTO DE HIERARQUIAS DE CLASSES

Este tópico mostra quatro técnicas de mapeamento de hierarquias de classes, juntamente com as desvantagens e vantagens de cada uma. É usado como exemplo a hierarquia do diagrama mostrado na figura 2.

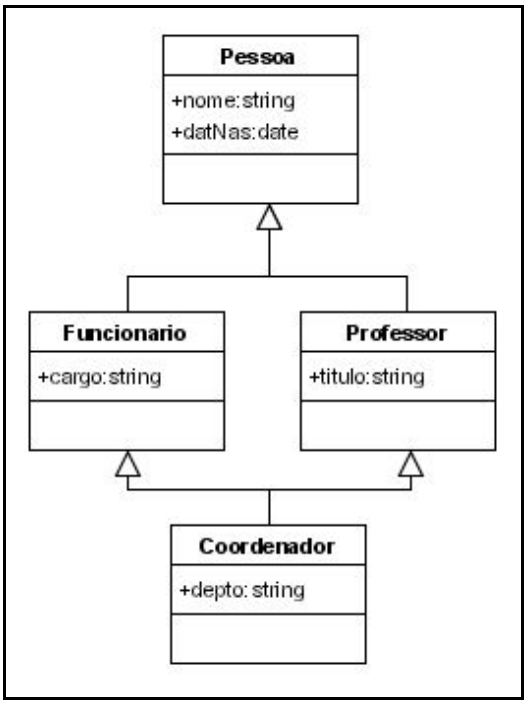

Figura 2 – Diagrama para exemplificar mapeamento de hierarquias.

3.2.1 Técnica "uma tabela por classe"

Consiste em definir uma tabela por classe. Cada tabela contém somente campos para atributos declarados na classe correspondente, mais o OID do objeto. A chave primária destas tabelas sempre será o OID, e caso seja necessária uma chave primária para o negócio, usa-se uma cláusula de unicidade em conjunto com cláusulas de obrigatoriedade, para os campos que fariam parte da chave primária.

Se uma classe qualquer `B` deriva de outra `A`, então o OID da tabela de `B` deve ser uma chave estrangeira para a tabela de `A`. Se a classe `B` deriva de `A1` e `A2` (herança múltipla), devem haver duas chaves estrangeiras em `B`, sendo uma para `A1` e outra para `A2`. Como na herança simples, cada chave estrangeira deve ser composta somente pelo campo OID.

Ao aplicar esta técnica na hierarquia de exemplo, o resultado obtido é o mostrado na figura 3.

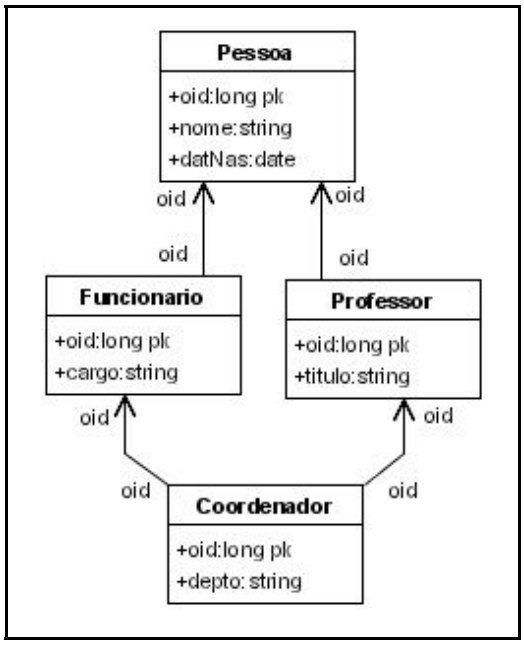

Figura 3 – Exemplo de aplicação da técnica "uma tabela por classe".

Observa-se que esta abordagem é a mais natural de todas, pois:

a) não há redundância – só são inseridos registros quando necessário, e todos os campos dos registros inseridos são usados pelo objeto;
b) a forma como são usadas cláusulas de consistência (*constraints*) no SGBDR protegem a base contra quase todas as inconsistências;
c) não são necessários campos adicionais para detectar o tipo de um objeto – basta ver em quais tabelas ele está presente;
d) a presença de um registro numa tabela garante que o objeto pertence à classe da tabela;
e) caso um objeto seja promovido de tipo, basta fazer uma inserção de registro; de forma análoga, caso seja rebaixado, basta fazer uma exclusão de registro;
f) relacionamentos entre objetos mapeiam para chaves estrangeiras de uma forma simples e direta.

Esta abordagem só permite uma inconsistência. Sejam as classes `A` e `B`, sendo que `A` não deriva de `B`, nem `B` deriva de `A`. Em geral, o tipo de um objeto não pode ser formado pela classe `A` e pela classe `B` ao mesmo tempo. Portanto, se existe um registro na tabela de `A` com

`OID = x`, não pode haver um registro na tabela de `B` com `OID = x`, mas o SGBDR não proibirá esta condição.

3.2.2 Técnica "uma tabela por tipo"

A técnica "uma tabela por classe" implica que um objeto que pertence a uma classe derivada terá suas partes armazenadas em tabelas diferentes, que são ligadas entre si com chaves estrangeiras. Numa tentativa de eliminar esta característica, que provoca o consumo excessivo de recursos, foi criada a técnica "uma tabela por tipo".

A diferença entre "classe" e "tipo" é que uma "classe" não contém todos os atributos (pois alguns atributos estarão em classes base e derivadas), enquanto um "tipo" contém. Deste modo, criar uma tabela por tipo significa que cada tabela conterá todos os campos do tipo correspondente, inclusive os campos para atributos definidos em classes base.

Por exemplo, se `B` e `C` são classes que derivam de `A`, a tabela de `B` conterá campos para atributos definidos em `A` e `B`, e a tabela de `C` conterá campos definidos para atributos em `A` e `C`. Se `A` for abstrata, não haverá tabela para `A`. Se `A` for não-abstrata, então a tabela de `A` conterá os atributos definidos em `A`. Independente do caso, qualquer tabela usará o OID como chave primária.

Quando um objeto `B` é salvo, é inserido apenas um registro na tabela de `B`. As tabelas de `A` e de `C` ficam intactas.

Ao aplicar esta técnica na hierarquia de exemplo, o resultado obtido é o mostrado na figura 4.

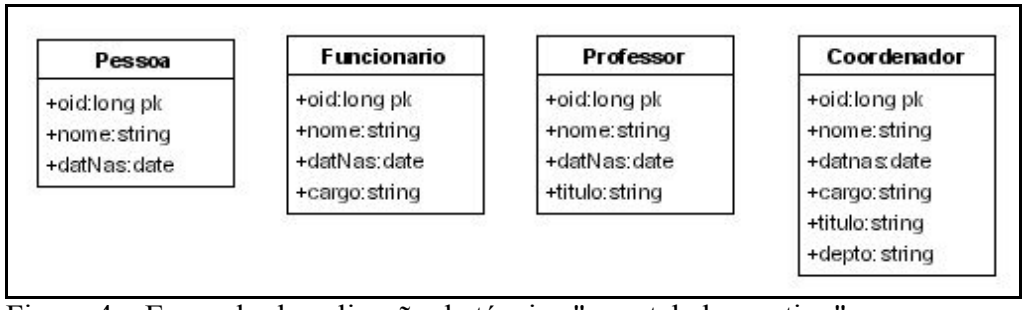

Figura 4 – Exemplo de aplicação da técnica "uma tabela por tipo".

Esta técnica tem os seguintes inconvenientes em relação à "uma tabela por classe":

a) há redundância de definição – um atributo em `A` causa o aparecimento de um campo correspondente na tabela de A e em cada tabela que deriva de `A`;

b) a consulta de objetos por classe é dificultada – se for necessário processar todos os objetos `A`, será necessário ler várias tabelas através da claúsula `UNION`;
c) não é possível ter chaves estrangeiras para relacionamentos entre objetos – se um objeto precisa referenciar outro em `A`, não basta criar uma chave estrangeira para a tabela de `A`, pois o objeto referenciado pode ser do tipo `B` e estar na tabela de `B`.

Estes inconvenientes, em especial o citado no item c, são motivos suficientes para que raramente se utilize esta técnica.

3.2.3 Técnica "uma tabela por hierarquia"

Numa tentativa de resolver os problemas das técnicas "uma tabela por classe" e "uma tabela por tipo", surgiu a técnica "uma tabela por hierarquia".

Primeiro identificam-se todas as hierarquias do sistema (uma hierarquia é um grafo conexo onde cada nó é uma classe e cada aresta é uma herança). Observa-se que, por ser um grafo conexo, cada classe pode pertencer a no máximo uma hierarquia.

Uma vez identificadas as hierarquias, cria-se uma tabela por hierarquia. Esta tabela conterá um campo para cada atributo definido em cada classe da hierarquia, mais o OID, que será a chave primária, e mais um campo que costuma se chamar CLSID (abreviação do Inglês *class id*). O campo CLSID é obrigatório e contém o código da classe do objeto (sem ele, seria impossível determinar a qual classe o objeto pertence).

Uma alternativa para o CLSID é definir um campo lógico para cada classe da hierarquia, sendo que se o objeto pertence àquela classe, o campo possui o valor `true`, senão possui o valor `false`.

Qualquer objeto que pertença a uma classe da hierarquia será armazenado na tabela da hierarquia, mesmo que não pertença a todas as classes da hierarquia.

Ao aplicar esta técnica na hierarquia de exemplo, o resultado obtido é o mostrado na figura 5.

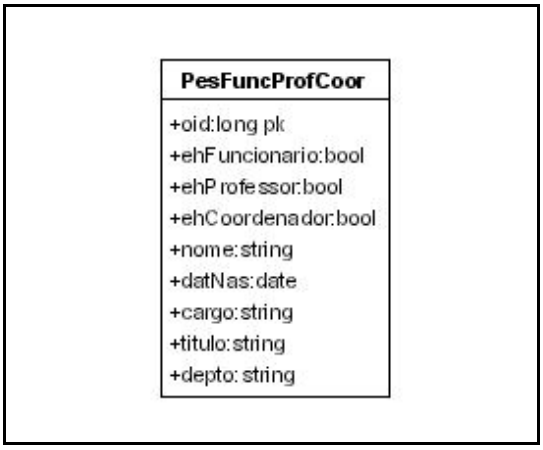

Figura 5 – Exemplo de aplicação da técnica "uma tabela por hierarquia".

Todos os inconvenientes da "uma tabela por tipo" são removidos, porém surgem os seguintes inconvenientes:

a) a maioria dos objetos não vai ocupar todo espaço reservado para o registro – haverá bastante redundância de espaço;
b) é necessário que haja, em cada registro, um ou mais campos que armazenam informações sobre o tipo do objeto;
c) caso seja necessário consultar apenas objetos de determinada classe, é necessário fazer uma filtragem, pois a tabela poderá conter registros que não pertencem à classe desejada.

Apesar destes inconvenientes, esta técnica é considerada muito superior à técnica "uma tabela por tipo". Assim ela é bastante usada em sistemas menos complexos, que têm hierarquias pequenas. Em sistemas complexos, é usada uma variante deste técnica, vista a seguir.

3.2.4 Técnica "uma tabela por grupo de classes"

Esta técnica consiste em fazer um balanço entre as técnicas "uma tabela por classe" e "uma tabela por hierarquia".

Primeiro, separam-se as classes em grupos, de modo que nenhuma classe pertença a mais de um grupo. Em geral, as classes de cada grupo pertencem à mesma hierarquia, mas não é obrigatório que seja deste modo.

Depois, cria-se uma tabela para cada grupo, sendo que a tabela deverá conter um campo para cada atributo de cada classe do grupo, mais o OID (que é a chave primária), mais um atributo lógico para cada classe do grupo.

Se `B` deriva de `A` e as duas classes foram definidas na mesma tabela, então uma chave estrangeira é desnecessária. Se `B` está na tabela `TB`, e `A` na tabela `TA`, então é necessária uma chave estrangeira em `TB`, que referencia `TA`. Se `TA` contém somente a classe `A`, então esta chave estrangeira pode conter apenas o OID. Se `TA` contém mais classes além de `A`, então haverá um campo lógico em `TA` chamado `ehA`, que determina se o registro pertence à classe `A`, e também haverá um campo lógico em `TB` chamado `ehB`, que determina se o registro pertence à classe `B`. Assim, a chave estrangeira será a chave composta `OID,ehB` em `TB`, que referencia `OID,ehA` em `TA`.

Ao aplicar esta técnica na hierarquia de exemplo, o resultado obtido é o mostrado na figura 6, assumindo-se que foi criado um grupo para `Pessoa` e `Professor`, e outro para `Funcionario` e `Coordenador`.

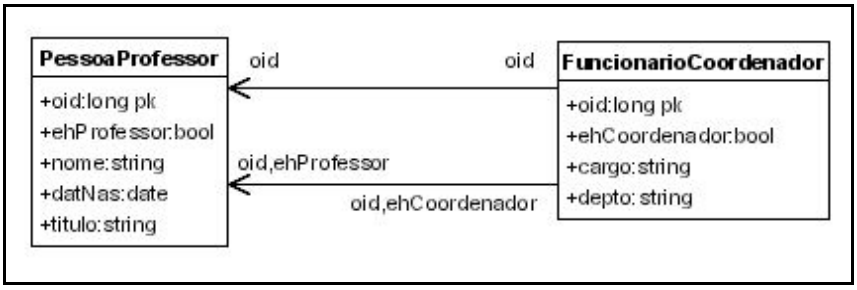

Figura 6 – Exemplo de aplicação da técnica "uma tabela por grupo de classes".

Esta técnica não elimina a necessidade de chaves estrangeiras para heranças, porém ela reduz bastante o aparecimento destas chaves. Também não elimina a necessidade dos campos que determinam a qual classe pertence o objeto, nem a filtragem para trazer somente objetos de determinada classe, porém reduz drasticamente o volume de espaço necessário, principalmente se há muitos objetos para classes daquela hierarquia.

3.3 MAPEAMENTO DE ASSOCIAÇÕES

O mapeamento de associações é bem mais simples do que o de hierarquia. Ele consiste basicamente na criação de chaves estrangeiras que usam o OID. Dependendo da cardinalidade da associação, são criadas chaves estrangeiras em lugares diferentes, como se vê na relação de

casos abaixo:

a) cardinalidade 1-para-n: apenas é criada uma chave estrangeira na tabela do lado n;
b) cardinalidade 1-para-1: são criadas duas chaves estrangeiras, uma em cada lado;
c) cardinalidade m-para-n: é criada uma tabela intermediária que possui duas chaves estrangeiras.

Seja o diagrama de classes da figura 7:

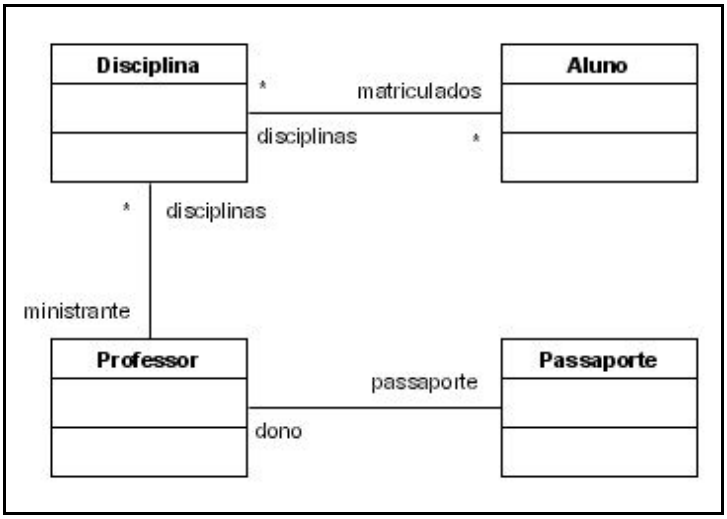

Figura 7 – Exemplo de diagrama de classes para mapeamento de relacionamentos.

O resultado da aplicação das regras de mapeamento baseadas na cardinalidade é o visto na figura 8.

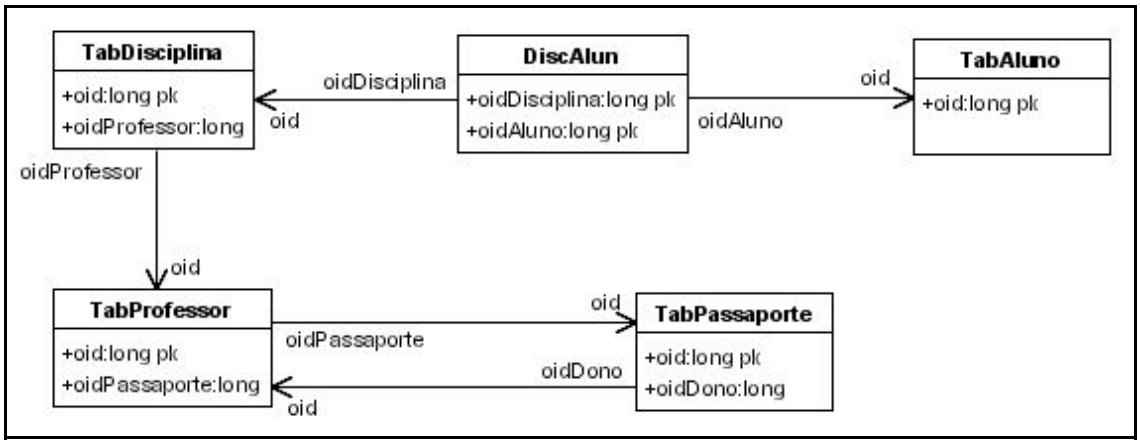

Figura 8 – Exemplo de aplicação das técnicas de mapeamento de relacionamento.

Observa-se o seguinte:

a) que embora pareça redundante, o professor e o passaporte têm OIDs distintos. É fundamental que seja deste modo, pois tratam-se de objetos diferentes;
b) o mapeamento é independente do tipo da referência definido na associação entre as classes; ou seja, sempre são criadas chaves estrangeiras, mesmo que as referências sejam suaves ou fortes.

O controle de utilidade dos objetos é feito pelo SGBDOO e não pelo SGBDR. As chaves estrangeiras são criadas por questões de otimização, de estabilidade do SGBDOO, e de garantia contra mudanças nos dados efetuadas por terceiros.

3.4 MAPEAMENTO DE COLEÇÕES GLOBAIS

O mapeamento de coleções globais é bastante simples. Cada coleção global se transforma num atributo lógico com o nome **pertA*coleção***, que é a abreviação de "pertence à coleção".

Por exemplo, seja a classe **Funcionario**, que define as coleções globais **Funcionarios** e **EmFerias**. A tabela que armazena objetos desta classe pode ser definida assim conforme a figura 9.

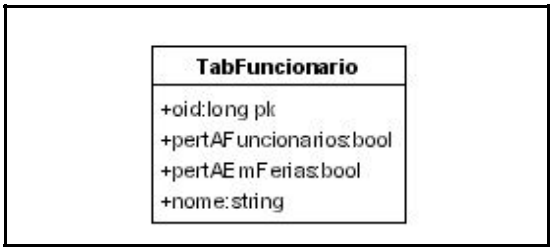

Figura 9 – Exemplo de tabela para classe com coleções globais.

4 ESPECIFICAÇÃO

Neste capítulo é apresentada a especificação do software.

4.1 PREMISSAS

A norma ODMG 3.0 foi usada como base para as funcionalidades que o software deve ter, mas não houve a intenção de suportar esta norma, pois isto está fora da abrangência deste trabalho.

Uma das premissas é maximizar a produtividade de programadores de SIs, e deixar o software com qualidade comercial. Deste modo, foram inclusas as seguintes funcionalidades:

a) herança múltipla no estilo do C++;
b) destruição automática de objetos;
c) coleções globais;
d) referências suaves e fracas.

O software não suporta as seguintes partes da norma ODMG 3.0:

a) objetos com métodos;
b) a linguagem ODL;
c) travamento explícito de objetos;
d) ligação com C++;
e) ligação com Smalltalk.

O software suporta as seguintes funcionalidades da norma ODMG 3.0, porém a sintaxe e a semântica destas funcionalidades pode diferir da norma:

a) objetos com interfaces;
b) transações;
c) meta-dados;
d) a linguagem OQL;
e) ligações com Java.

Detalhes sobre cada funcionalidade suportada serão fornecidos ao longo deste capítulo, bem como a razão de não suportar certas funcionalidades.

4.2 PAPEL DOS USUÁRIOS

Qualquer usuário direto do software tem um ou mais dos seguintes papéis:

a) analista: modela os dados a serem persistidos;

b) consulente: cria e submete consultas *ad hoc* ao SGBDOO, com o objetivo de obter dados para tomada de decisão ou para confecção de relatórios;
c) programador: cria, lê ou mantém arquivos-fonte escritos em Java contendo acessos ao SGBDOO;
d) implantador: instala e configura o software para entrar em produção;
e) consultor: especialista no SGBDR escolhido, que interfere no processo de criação das tabelas para que as mesmas sejam otimizadas para o SGBDR.

Cada função do software é útil para um ou mais destes papéis. Estas funções estão descritas no diagrama de caso de uso da figura 10.

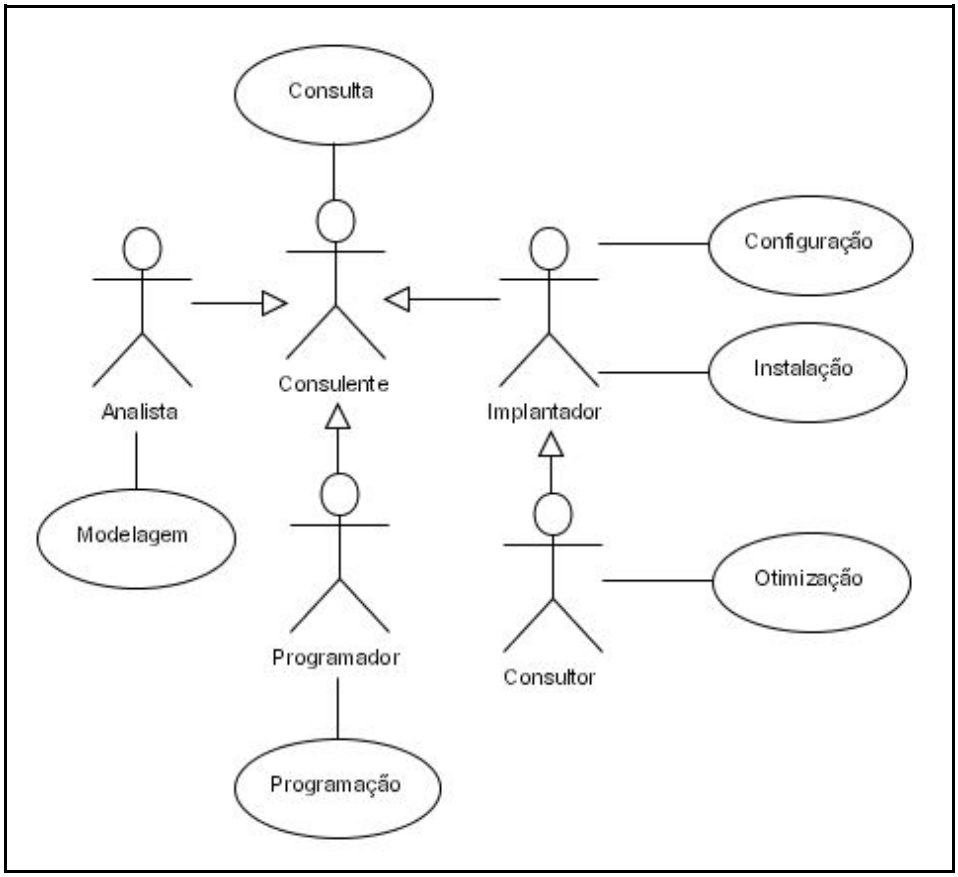

Figura 10 – Diagrama de caso de uso com os principais papéis dos usuários e os principais processos.

4.3 MODELAGEM DE DADOS

O analista modela seus dados através de diagramas de classe expressados em UML. Estes diagramas podem conter uma boa parte dos recursos da UML, porém alguns recursos não são suportados, ao passo que são suportadas algumas extensões que não são padronizadas

pela OMG.

Os recursos mais interessantes da UML são suportados. A lista destes recursos inclui mas não se limita a:

a) pacotes;
b) classes do tipo simples ou abstrata;
c) atributos do tipo lógico, numérico, data, texto, ou enumeração.
d) associações do tipo simples, agregação, ou composição;
e) associações com cardinalidade 1-para-1, 1-para-n, ou m-para-n;
f) herança simples ou múltipla, no estilo do C++.

Entre os recursos não suportados estão os seguintes, pelas razões descritas:

a) métodos: as técnicas de programação modernas recomendam forte separação entre dados e processamento, e esta prática tem se mostrado bem sucedida porque cria modelos mais limpos e concisos; para forçar esta prática, o suporte a métodos foi abolido;
b) interfaces: há duas motivações para não suportar interfaces:
   - interfaces só podem conter métodos, e métodos não são suportados,
   - há polêmica sobre a real utilidade das interfaces quando há suporte a classes abstratas; aderiu-se à corrente que diz que interfaces na verdade não passam de classes abstratas;
c) associações n-árias (onde há mais que dois papéis): haveria complexidade excessiva e desnecessária para contemplar os objetivos do software; raros são os casos em que associações n-árias não implicam na adição de uma classe com associações binárias comuns;
d) consistências em OCL: geraria complexidade e confusão para os usuários, pois já existe uma linguagem usada abundantemente que é a OQL; assim, o software permite que as consistências sejam expressas em OQL ao invés de OCL, o que prejudica apenas o intercâmbio de modelos;
e) nomes com espaços e outros símbolos gráficos: embora a UML permita nomes assim, os nomes de atributos, classes e papéis em associações devem aderir à regra dos identificadores da linguagem Java;
f) atributos de qualquer tipo de dados: o tipo dos atributos deve ser numérico, lógico, data, texto, ou enumeração; tipos estruturados como matrizes e classes não são suportados em atributos; se forem necessários, obtém-se o mesmo efeito com o uso

de associações.

São fornecidos os seguintes recursos que não são padronizados na UML:

a) consistências em OQL: em função de não haver suporte à OCL, optou-se por suportar a OQL para definição de consistências, visto que OQL é usada em diversos lugares; observou-se que a expressividade da OQL para definição de consistências é satisfatória, e inclusive em alguns casos observa-se diminuição no tamanho do texto em relação à OCL;
b) consistências de unicidade: em função de não haver suporte à OCL, permite-se a especificação de consistências de unicidade para coleções de objetos, na qual se informa um conjunto de atributos, de maneira semelhante a um SGBDR;
c) coleções globais: permitem a implementação do coletor de lixo no SGBDOO, que será visto em detalhes mais adiante.

Estes recursos adicionais, não padronizados pela UML, são encontrados com freqüência em ferramentas CASE, e de fato se mostram úteis no desenvolvimento de sistemas de informação. Para este projeto, optou-se por trocar a portabilidade dos diagramas UML por maior produtividade na produção de sistemas novos.

Ainda assim, observa-se que a notação da UML é completamente respeitada, e assim obtém-se o benefício da comunicação formal entre as pessoas, proporcionado pelo fato de a UML ser amplamente difundida nas empresas e em instituições de ensino. Em outras palavras, uma pessoa que entende UML é capaz de ler os diagramas aceitos por este software, também possui boa parte do conhecimento necessário para criar e modificar tais diagramas.

4.4 VALIDAÇÃO DO MODELO DE DADOS

A modelagem dos dados é feita através de uma ferramenta que tenha este fim (como uma ferramenta CASE). Esta ferramenta não é parte do software que é objeto deste trabalho. Uma vez que a modelagem foi feita, o modelo deve ser submetido ao software para que seja validado e incorporado ao ambiente de trabalho dos usuários.

A ferramenta de modelagem gera, a partir dos diagramas de classe, arquivos com extensão `xcls`. É gerado um arquivo por classe. Cada arquivo contém todas as informações sobre determinada classe, incluindo mas não se limitando a:

a) pacote onde a classe se encontra;
b) nome da classe;
c) nome das classes-base;

d) definição completa de cada atributo da classe;
e) definição completa de cada associação que a classe tem com outras.

O conteúdo do arquivo `xcls` está no formato XML, sendo editável por qualquer ferramenta que suporte esta linguagem, ou até mesmo por editores de texto comuns. Assim, na ausência da ferramenta de modelagem, é possível criar estes arquivos de outras formas.

O software possui uma ferramenta que lê todos os arquivos `xcls`, remonta todo modelo de dados na memória, faz a validação deste, e emite um pequeno relatório textual, com o resultado deste processo. O relatório contém algumas informações estatísticas (quantidade de classes, atributos, associações, etc.), seguido da relação dos erros e avisos. Há 50 tipos de erro, e 10 tipos de aviso, mas possivelmente existem erros e avisos que ainda não foram previstos.

Se o modelo está válido, é feita a incorporação do mesmo ao ambiente de trabalho dos usuários. A incorporação faz com que as classes, coleções, objetos pré-definidos, e outros elementos do modelo, fiquem disponíveis para serem usadas pelos usuários. Em outras palavras, a incorporação se compara a rodar um script DDL num SGBDR. Detalhes sobre a incorporação serão mostrados mais adiante.

4.5 CONSULTAS

O consulente realiza consultas através da linguagem OQL (*Object Query Language*). Por ser uma linguagem voltada para consultas, ela não possui os seguintes recursos:

a) declaração de variáveis;
b) bloco de comandos;
c) comandos de controle de fluxo;
d) declaração de procedimentos, funções, classes, módulos, etc.

Como foi dito, o software não se propõe a ficar em conformidade com a especificação ODMG 3.0. Assim foi desenvolvida uma variação da OQL, sem recursos considerados supérfluos e com alguns recursos adicionais. Ainda assim, muitos comandos expressos na OQL original são suportados.

A seguir seguem os recursos da OQL que são suportados. A sintaxe completa, descrita em BNF, pode ser vista no Anexo A.

4.5.1 Tipos de dados, valores, e sintaxe dos literais

São suportados os seguintes tipos de dados em comandos da OQL:

a) lógico;
b) numérico;
c) data;
d) texto;
e) classe (tipo abstrato);
f) coleção.

4.5.1.1 Lógico

É um tipo de dados que suporta somente os valores discretos verdadeiro (`true`) e falso (`false`).

4.5.1.2 Numérico

Suporta os números reais com precisão limitada somente pela memória. Não há suporte para números complexos e nem para dízimas. O ponto decimal é usado para separar a parte inteira da fracionária. A notação exponencial também é suportada. Exemplos de números:

a) `137`;
b) `3.1415926539`;
c) `000.010` (equivalente a `0.01`);
d) `3e5` (equivalente a `300000`);
e) `3e+5` (equivalente `3e5`);
f) `3e-5` (equivalente a `0.00003`).

4.5.1.3 Data

Suporta datas do calendário gregoriano. Usa-se a sintaxe `date(`*`ano`*`,`*`mês`*`,`*`dia`*`)` para expressar datas. A ordem dos componentes (ano seguido pelo mês, seguidos pelo dia) foi escolhida para concordar com a ordenação das datas. As datas são ordenadas pelo ano. As que tiverem o mesmo ano são ordenadas entre si pelo mês, e as que tiverem os mesmos ano e mês são ordenadas entre si pelo dia.

Os componentes da data devem ser números inteiros, ou expressões constantes que resultam em números inteiros. Variáveis não podem ser informadas. Caso seja necessário construir uma data por variáveis, usa-se a função `getDate(`*`ano`*`,`*`mês`*`,`*`dia`*`)`, que será mostrada mais adiante.

Exemplos:

a) `date(2003,5,1)` – 1 de Maio de 2003;

b) `date(1976+1,12,3)` – 3 de Dezembro de 1977;

c) `date(1995,3-7,20)` – 20 de Agosto de 1994;

d) `date(0,10,31)` – 31 de Outubro do ano zero;

e) `date(-10, 7, 16)` – 16 de Julho do ano 10 a.C.;

Pelo exemplo c, nota-se que se um componente é informado fora de sua faixa natural, a data é ajustada de modo que fique consistente. Deste modo, `date(1995,3-7,20)` se transforma em `date(1995,-4,20)`, que equivale a `date(1994,8,20)`.

4.5.1.4 Texto

Suporta seqüências de caracteres Unicode. Usa-se aspas simples para delimitar textos. Exemplos:

a) `'Isto é um texto'` – Isto é um texto;

b) `'Caixa d''água'` – Caixa d'água;

c) `'Primeira linha\nSegunda linha'`.

Observa-se que o caractere `'` deve ser colocado em duplicidade para não ser interpretado como fim do texto. De restante, uma expressão textual deve respeitar a codificação que a ferramenta usa para converter os caracteres em Unicode (UNICODE, 2003). Os exemplos acima, e a ferramenta de consultas interativas (apresentada mais adiante), usam a codificação UTF-8 (por isso o trecho `\n` é convertido para uma quebra de linha). Caso o texto esteja num fonte Java, pode estar em UTF-8 ou UTF-16 (ISO, 2000), pois estas são as codificações suportadas pelo compilador Java.

O texto deve conter no mínimo um caractere. A expressão `''` provoca um erro de compilação. Isto é necessário para distinguir corretamente um texto do valor especial `null`, a ser explicado mais adiante.

O tamanho máximo do texto é limitado pelas condições do ambiente. Muitos ambientes possuem um limite virtual de cerca de 2^{32} caracteres.

Não há um tipo de dado especial para o caractere. Funções que manipulam texto estabelecem que o tipo de cada caractere é um texto ou um número inteiro, dependendo da função.

4.5.1.5 Classe (tipo abstrato)

Uma classe é um tipo de dados abstrato. Diferente dos demais tipos, o ambiente de execução só poderá trabalhar com valores da classe depois que ela for definida formalmente perante o ambiente. Isto é feito na fase de incorporação do modelo de dados, que será mostrada mais adiante.

Valores de uma classe são chamados de instâncias ou de objetos. O formato destes valores, assim como os dados neles contidos, são inteiramente definidos pela classe. Muitas linguagens de programação, inclusive a UML, permitem especificar as operações que podem ser feitas com estes valores, mas não é o caso deste software. O emulador permite apenas criar, destruir, alterar, e consultar estes valores, sendo que a consulta pode envolver um objeto em cálculos.

A linguagem OQL implementada por este software não suporta constantes para objetos. Em geral, objetos são criados através de um programa escrito em Java. Detalhes sobre a criação de objetos serão apresentados mais adiante.

O uso de objetos em comandos OQL se dá por referências. Por exemplo, se `p` é uma variável do tipo `PessoaFisica`, então o valor de `p` é uma referência para uma instância da classe `PessoaFisica`.

Referências para objetos permitem acessar membros daquele objeto. Se a classe `PessoaFisica` define o atributo `nome` do tipo texto, o valor de `p.nome` é valor deste atributo na instância referenciada por `p`. O ponto que separa a referência do atributo é o operador de navegação, explicado mais adiante.

Associações entre classes se transformam em referências num comando OQL. Se `v` é uma variável do tipo `Veiculo`, e esta classe possui uma associação com `PessoaFisica` na qual o papel da última é `proprietario`, então o valor de `v.proprietario` é uma referência para uma instância de `PessoaFisica`, e é válido usar `v.proprietario.nome`.

Pode-se verificar a igualdade e a desigualdade entre referências. Se `x` e `y` são referências, sempre que `x=y` é verdadeiro, `x` e `y` estão referenciando o mesmo objeto, e sempre que `x<>y` é verdadeiro, `x` e `y` estão referenciando objetos distintos. Isto funciona devido ao fato de cada objeto ter uma identidade que o distingue dos demais.

A identidade é análoga à posição de um objeto na memória, e a referência é análoga a um ponteiro para aquele objeto. Porém, o verdadeiro formato interno das identidades e das

referências é escondido do usuário, para induzir a simplicidade, corretude, e segurança nos programas. A OQL não possui meios para se obter o valor interno das referências.

4.5.1.6 Coleção de valores

Coleções são entidades que contém um ou mais elementos. O tipo dos elementos pode ser qualquer, inclusive coleção.

Qualquer coleção se enquadra em uma das definições abaixo:

a) sacola (do Inglês *bag*): não mantém a posição de cada elemento, nem proibe a presença de mais de um elemento com o mesmo valor;
b) lista: mantém cada elemento numa posição fixa, mas não proibe a presença de mais de um elemento com o mesmo valor;
c) conjunto: não mantém a posição de cada elemento, mas proibe a presença de mais de um elemento com o mesmo valor;
d) lista e conjunto ao mesmo tempo: mantém cada elemento numa posição fixa e proibe a presença de mais de um elemento com o mesmo valor.

Coleções suportam as operações filtragem, ordenação, seleção de elemento, e outras que serão apresentadas mais adiante.

4.5.2 O estado nulo

O estado nulo serve para especificar a ausência de valor. Quando o estado de uma variável ou atributo é nulo, ou quando um comando OQL possui `null` em alguma parte, o usuário quer dizer que ali não foi informado um valor.

Já que o estado nulo indica ausência de valor, não é possível conceituá-lo como sendo um valor. Portanto, o conjunto de estados que uma variável pode assumir inclui cada estado associado a um valor possível, mais o estado adicional nulo.

Esta semântica foi introduzida pela IBM quando foi criado o SQL, e é usada até hoje por qualquer SGBDR que suporte SQL. Inclusive foi adotada pelas normas SQL92 (ANSI, 1998) e SQL3 (ISO, 1999).

Assim, a frase "o valor de `x` é `null`" está errada, pois `null` não é um valor. As frases corretas seriam "o estado de `x` é `null`" ou "`x` não possui valor". Porém, devido ao modo como se aprende informática, existe a tendência de assumir que `null` é um valor, e esta forma é muito mais usada no cotidiano do que assumir que `null` não é um valor. Assim, por questão

de inteligibilidade, certas partes deste trabalho deixam a entender que `null` é um valor. Mas ressalta-se que isto só ocorre onde a distinção entre valor e estado é inútil para a compreensão do texto.

O fato de `null` indicar ausência de valor traz uma série de implicações. A primeira delas é que geralmente, quando `null` é usado numa expressão, o resultado da expressão toda é `null`. Alguns exemplos:

a) `5 + null` resulta em `null` e não em `5`, pois `null` não indica o valor `0`; ocorre que uma das parcelas da soma está ausente, e em conseqüência, a soma não pode ser efetuada e o resultado da mesma também fica ausente;

b) `5 = null` resulta em `null`, e não em `false`, pois `null` não indica o valor `0`; ocorre que uma das partes da igualdade está ausente, sendo impossível realizar a operação, ficando o resultado da mesma também ausente;

c) `null = null` resulta em `null` e não em `true`, pois as duas partes da igualdade são ausentes, sendo impossível realizar a operação;

d) `null <> null` resulta em `null` e não em `false`, pois as duas partes da desigualdade são ausentes, sendo impossível realizar a operação;

e) `true and null` resulta em `null` e não em `false`, pois `null` não indica o valor `false`;

f) `false or null` resulta em `null` e não em `false`, pois `null` não indica o valor `false`;

g) `not null` resulta em `null` e não `true`, pois `null` não indica o valor `false`;

h) `1 + 5 / null` resulta em `null`, pois se reduz para `1 + null`, que resulta em `null`;

i) `x = null` resulta em `null`, não importando o valor de `x` ou que `x` seja nulo;

j) `x <> null`, `x + null` e `x and null` também resultam em `null` não importando o valor de `x` ou que `x` seja nulo.

Mas há também operações que usam `null` e não retornam `null`, pois o resultado pode ser deduzido de forma independente do valor que está ausente. Tratam-se de algumas expressões lógicas. Exemplos:

a) `false and null` resulta em `false`, pois o operando da esquerda (`false`) garante este resultado de modo independente do valor que está ausente; o mesmo ocorre para `null and false` e `x and false and null` – nestas duas o resultado é `false`;

b) `true or null` resulta em `true`, pois o operando da esquerda (`true`) garante resultado de modo independente do valor que está ausente; o mesmo ocorre para `null or true` e `x or true or null` – nestas duas o resultado é `true`.

A ausência ou presença de valor pode ser determinada pelos predicados `is null` e `is not null`, que serão apresentados mais adiante.

4.5.3 Sintaxe e semântica das expressões

A linguagem OQL é bem simples e de certo modo, bastante parecida com a SQL. Ainda assim ela é mais poderosa, e certas construções não ortogonais, como a do `group by`, foram modificadas. A concisão foi uma das principais motivações durante a concepção da OQL.

Todo comando OQL resulta num valor, cujo tipo é um dos tipos apresentados no tópico. Na ferramenta de consultas interativas, o valor resultante é enviado em sua forma textual para a saída padrão. Deste modo, é possível usar a ferramenta de consultas interativas como uma simples calculadora. O quadro 2 abaixo mostra uma seqüência de comandos OQL.

```
oql> 1 + 1
2

oql> 1 + 2 * 3 / 5
2.2

oql> today()
date(2003,10,25)

oql> first(PessoasFisicas)
nome = 'Smith'
titulo.nome = 'Agent'
datNas = date(1968, 10, 2)

oql> |
```

Quadro 2 - Seqüência de comandos em OQL

Observe que uma simples expressão matemática é um comando OQL válido. Nos subtópicos que seguem, são apresentados as construções de linguagens suportadas pela OQL. Deve-se assumir que quando `null` aparece no lugar de um operando, o resultado da expressão é `null`, exceto se for definido diferente, onde o `null` é citado de forma explícita.

4.5.3.1 Expressões binárias

4.5.3.1.1 e-lógico e ou-lógico

Sintaxe do e-lógico: ***`operando` `and` `operando`***

Sintaxe do ou-lógico: ***`operando` `or` `operando`***

O e-lógico resulta em **`false`** se um dos operandos é **`false`**, em **`true`** se ambos são **`true`**, e em **`null`** em qualquer outra situação. O ou-lógico resulta em **`true`** se um dos operandos é **`true`**, em **`false`** se ambos são **`false`**, e em **`null`** em qualquer outra situação. Os operandos devem ser lógicos. Exemplos:

a) **`true and false`** resulta em **`false`**;

b) **`true and x`** resulta em **`x`**;

c) **`null and true`** resulta em **`null`**;

d) **`true or false`** resulta em **`true`**;

e) **`false or x`** resulta em **`x`**;

f) **`null or false`** resulta em **`null`**.

4.5.3.1.2 Igualdade e desigualdade

Sintaxe da igualdade: ***`operando = operando`***

Sintaxe da desigualdade: ***`operando <> operando`***

A igualdade resulta em **`true`** se os valores dos operandos são iguais, e em **`false`** se são diferentes. A desigualdade resulta em **`true`** se os valores dos operandos são diferentes, e em **`false`** se forem iguais. Se um dos operandos é nulo, as operações resultam em **`null`**. Os operandos devem ser de tipos compatíveis, e não podem ser coleções. Exemplos:

a) **`1 = 2`** resulta em **`false`**;

b) **`false = false`** resulta em **`true`**;

c) **`a = null`** resulta em **`null`**;

a) **`1 <> 2`** resulta em **`true`**;

b) **`false <> false`** resulta em **`false`**;

c) **`a <> null`** resulta em **`null`**.

Observa-se que:

a) **`(x=y) <> (x<>y)`** nunca resulta em **`false`**;

b) **`x=y`** resulta em **`not (x<>y)`**.

4.5.3.1.3 menor-que, maior-que, menor-ou-igual-a, maior-ou-igual-a

Sintaxe do menor-que: ***operando*** **<** ***operando***

Sintaxe do maior-que: ***operando*** **>** ***operando***

Sintaxe do menor-ou-igual-a: ***operando*** **<=** ***operando***

Sintaxe do maior-ou-igual-a: ***operando*** **>=** ***operando***

O menor-que resulta em **true** se o valor do operando da esquerda é menor do que o do da direita, senão em **false**. O maior-que resulta em **true** se o valor do operando da esquerda é maior do que o do da direita, senão em **false**. O menor-ou-igual-a resulta em **true** se o valor do operando da esquerda é menor ou igual ao do da direita, senão em **false**. O maior-ou-igual-a resulta em **true** se o valor do operando da esquerda é maior ou igual ao do da direita, senão em **false**. Se um dos operandos é nulo, as operações resultam em **null**. Os operandos devem ser de tipos compatíveis, e não podem ser coleções. Exemplos:

a) **10 > 5** resulta em **true**;
b) **10 > 10** resulta em **false**, **10 >= 10** resulta em **true**;
c) **'A' < 'X'** resulta em **true**;
d) **'A' < 'AA'** resulta em **true**;
e) **'15' > '6'** resulta em **false** (a comparação é de textos);
f) **null > null** resulta em **null**;
g) **null < 5** resulta em **null**.

Observar que:

a) **(x > y) = (y < x)** nunca resulta em **false**, assim como **(x >= y) = (y <= x)**;
b) **x > y** resulta em **not (x <= y)**, assim como **x < y** resulta em **not (x >= y)**;
c) se **x = y** não resulta em **false**, então **x = y**, **x >= y** e **x <= y** têm o mesmo resultado;
d) se **x <> y** não resulta em **false**, então **x > y** e **x >= y** têm o mesmo resultado, assim como **x < y** e **x <= y**.

4.5.3.1.4 pertence- à-classe, não-pertence-à-classe

Sintaxe do pertence-à-classe: ***operando*** **is** ***operando***

Sintaxe do não-pertence-à-classe: ***operando*** **is not** ***operando***

O pertence-à-classe resulta em `true` se o valor do operando da esquerda é uma referência para um objeto que pertence à classe informada no operando da direita. O não-pertence-à-classe resulta em `true` se o valor do operando da esquerda é uma referência para um objeto que não pertence à classe informada no operando da direita. O operando da esquerda deve ser uma referência qualquer, e o da direita deve ser uma referência para um objeto `Class`.

Exemplos:

a) `p is PessoaFisica` resulta em `true` se `x` pertence à classe `PessoaFisica`, `false` se não pertence, e `null` se `x` é nulo;

b) `v is not Veiculo` resulta em `true` se `v` pertence à classe `Veiculo`, `false` se não pertence, e `null` se `v` é nulo.

Observa-se o seguinte:

a) `null is x` sempre resulta em `null`, assim como `null is not x`;

b) cada classe é um objeto em si, e quando o nome de uma classe é usado como um operando, o tipo dele é uma referência para um objeto `Class`;

c) as construções `x is null` e `x is not null`, ou seja, quando o `null` é explicitamente especificado no comando, não são operações pertence-à-classe e não-pertence-à-classe. São operações ausência-de-valor e presença-de-valor, respectivamente.

4.5.3.1.5 Soma numérica

Sintaxe: ***`operando + operando`***

É a simples soma dos dois operandos, que devem ser números. Exemplos:

a) `1 + 1` resulta em `2`;

b) `-2 + -3` resulta em `-5`;

c) `5e3 + 5e-3` resulta em `5000.005`;

d) `10 + null`, `null + 10` e `null + null` resultam em `null`.

4.5.3.1.6 Subtração numérica

Sintaxe: ***`operando - operando`***

É a subtração do operando da esquerda pelo da direita, os quais devem ser números. Exemplos:

a) `1 - 1` resulta em `0`;

b) `5.5 - 10` resulta em `-9.5`;

c) `4e4 - 4e3` resulta em `36000`;

d) `null - 5`, `5 - null` e `null - null` resultam em `null`.

4.5.3.1.7 Multiplicação numérica

Sintaxe: ***`operando * operando`***

É a multiplicação de um operando pelo outro, os quais devem ser números. Exemplos:

a) `3 * 7` resulta em `21`;

b) `5.5 * -4` resulta em `-22`;

c) `9e3 * 1e4` resulta em `9e7`;

d) `10 * null`, `null * 10` e `null * null` resultam em `null`.

4.5.3.1.8 Divisão numérica

Sintaxe: ***`operando / operando`***

É a divisão do operando da esquerda pelo da direita, os quais devem ser números. A divisão por zero causa uma interrupção em todo cálculo. Exemplos:

a) `1 / 4` resulta em `0.25`;

b) `4 / 0.25` resulta em `16`;

c) `1 / 3` resulta em `0.333333333333333`;

d) `5 / null`, `null / 5` e `null / null` resultam em `null`;

e) `5 / 0` e `null / 0` são operações inválidas.

Caso ocorra uma divisão por zero, o cálculo inteiro é interrompido e uma exceção é lançada.

4.5.3.1.9 Resto da divisão numérica

Sintaxe: ***`operando`*** **`mod`** ***`operando`***

É o resto da divisão do operando da esquerda pelo da direita, os quais devem ser números. Exemplos:

a) `1 mod 4` resulta em `1`;

b) `4 mod 0.25` resulta em `0`;

c) `45 mod -7` resulta em `3`;

d) `-45 mod -7` resulta em `-3`;

e) `10 mod null` e `null mod 5` resultam em `null`;

f) `10 mod 0` e `null mod 0` são operações inválidas.

Caso o operando da direita seja zero, o cálculo inteiro é interrompido e uma exceção é lançada.

4.5.3.1.10 Soma de data com número

Sintaxe: ***`operando + operando`***

Quando um dos operandos é uma data e outro é um número, o resultado é a data incrementada em tantos dias quanto for o número. Se o número não for inteiro a parte fracionária será ignorada. Se o número for negativo, a data resultante será menor do que a data sendo somada. Se for zero, será igual. Exemplos:

a) `date(2002, 10, 5) + 365` resulta em `date(2003, 10, 5)`;

b) `date(2003, 10, 1) + -30` resulta em `date(2003, 9, 1)`;

c) `15 + date(1977,11, 18)` resulta em `date(1977, 12, 3)`;

d) `date(2002, 10, 5) + null`, `null + date(2002, 10, 5)` e `null + null` resultam em `null`.

4.5.3.1.11 Subtração de data por número

Sintaxe: ***`operando - operando`***

Quando o operando da esquerda é uma data e o da direita é um número, o resultado é a data decrementada em tantos dias quanto for o número. Se o número não for inteiro a parte fracionária será ignorada. Se o número for negativo, a data resultante será maior do que a data sendo somada. Se for zero, será igual. Exemplos:

a) `date(2003, 10, 5) - 365` resulta em `date(2002, 10, 5)`;

b) `date(2003, 9, 1) - -30` resulta em `date(2003, 10, 1)`;

c) `date(1977, 12, 3) - 15` resulta em `date(1977, 11, 18)`;

d) `date(2002, 10, 5) - null`, `null - 15` e `null - null` resultam em `null`.

4.5.3.1.12 Diferença entre datas

Sintaxe: ***`operando - operando`***

Quando os dois operandos da subtração forem datas, o resultado é a diferença entre estas datas, expressa na quantidade de dias decorridos do operando da direita até o operando da esquerda. Se a data da esquerda estiver após a data da direita, o resultado será positivo. Se estiver antes, será negativo. Se as datas forem iguais, será zero. Exemplos:

a) `date(2003, 10, 5) - date(2002, 10, 5)` resulta em `365`;

b) `date(2003, 10, 1) - date(2003, 9, 1)` resulta em `30`;

c) `date(2003, 9, 1) - date(2003, 10, 1)` resulta em `-30`;

d) `date(2003, 9, 1) - null`, `null - date(2003, 10, 1)` e `null - null` resultam em `null`.

4.5.3.1.13 Replicação de texto

Sintaxe: ***`operando * operando`***

Se um dos operandos da multiplicação for um texto e o outro for um número, o resultado será o texto replicado tantas vezes quanto for o número. Se o número não for inteiro a parte fracionária será ignorada. Se o número for menor que `1`, o resultado será `null`. Exemplos:

a) `'aba ' * 3` resulta em `'aba aba aba '`;

b) `5.29 * '1'` resulta em `'11111'`;

c) `'x' * 1` resulta em `'x'`;

d) `0 * 'x'` resulta em `null`;

e) `'x' * -4` resulta em `null`;

f) `'A' * null` e `null * 'X'` resultam em `null`.

4.5.3.1.14 Concatenação de texto

Sintaxe: ***`operando || operando`***

É um texto formado pelo operando da esquerda seguido pela da direita, os quais devem ser textos. Se um dos operandos for `null`, o resultado não será `null`, e sim o valor do outro operando. Exemplos:

a) `'Olá' || 'Mundo'` resulta em `'OláMundo'`;

b) `'10' || '15'` resulta em `'1015'`;

c) `'x' || null` resulta em `'x'`;

d) `null || 'y'` resulta em `'y'`;

e) `null || null` resulta em `null`.

4.5.3.1.15 Concatenação de coleções

Sintaxe: `operando || operando`

É uma lista onde os primeiros elementos são os do operando da esquerda e os últimos são os do operando da direita, os quais devem ser coleções. Se um dos operandos for `null`, o resultado não será `null`, e sim uma lista com os elementos do outro operando. Exemplos:

a) `list(1, 3, 2) || list(2, 3, 5)` resulta em `list(1, 3, 2, 2, 3, 5)`;

b) `set(1, 2, 3, 3, 2, 4, 1) || set(1, 2, 1, 3, 1, 4)` resulta em `list(1, 2, 3, 4, 1, 2, 3, 4)`;

c) `list('a', 'b', 'c', null) || list(null, 'null', 'x')` resulta em `list ('a', 'b', 'c', null, null, 'null', 'x')`;

d) `null || list(2, 1)` resulta em `list(2, 1)`;

e) `set(-1, 0, null, 1) || null` resulta em `list(-1, 0, null, 1)`;

f) `null || null` resulta em `null`.

4.5.3.1.16 União simples de coleções

Sintaxe: `operando union operando`

É uma sacola contendo os elementos do operando da esquerda e os do operando da direita, os quais devem ser coleções cujo tipo dos elementos deve ser compatível. Se um dos operandos for `null`, o resultado não será `null`, e sim um saco com os elementos do outro operando. Exemplos:

a) `list(1, 3, 2) union list(2, 3, 5)` resulta em `bag(1, 2, 2, 3, 3, 5)`;

b) `set(1, 2, 3, 3, 2, 4, 1) union set(1, 2, 1, 3, 1, 4)` resulta em `bag(1, 1, 2, 2, 3, 3, 4, 4)`;

c) `list('a', 'b', 'c', null) union list(null, 'null', 'x')` resulta em `bag (null, null, 'a', 'b', 'c', 'null', 'x')`;

d) `null union list(2, 1)` resulta em `bag(1, 2)`;

e) `set(-1, 0, null, 1) union null` resulta em `bag(-1, 0, 1)`;

f) `null union null` resulta em `null`.

4.5.3.1.17 União de coleções gerando conjunto

Sintaxe: `operando union distinct operando`

É um conjunto contendo os elementos do operando da esquerda e os do operando da direita, os quais devem ser coleções cujo tipo dos elementos deve ser compatível. Se um dos operandos for `null`, o resultado não será `null`, e sim um conjunto com os elementos do outro operando. Exemplos:

a) `list(1, 3, 2) union distinct list(2, 3, 5)` resulta em `set(1, 2, 3, 5)`;

b) `set(1, 2, 3, 3, 2, 4, 1) union distinct set(1, 2, 1, 3, 1, 4)` resulta em `set(1, 2, 3, 4)`;

c) `list('a', 'b', 'c', null) union distinct list(null, 'null', 'x')` resulta em `set(null, 'a', 'b', 'c', 'null', 'x')`;

d) `null union distinct list(2, 1)` resulta em `set(1, 2)`;

e) `set(-1, 0, null, 1) union distinct null` resulta em `set(-1, 0, 1)`;

f) `null union distinct null` resulta em `null`.

4.5.3.1.18 Intersecção de coleções

Sintaxe: ***`operando`*** **`intersect`** ***`operando`***

É um conjunto contendo os elementos presentes tanto no operando da esquerda como no da direita, os quais devem ser coleções cujo tipo dos elementos deve ser compatível. Se não houver elementos presentes nas duas coleções, o resultado será `null`. Exemplos:

a) `list(1, 3, 2) intersect list(2, 3, 5)` resulta em `set(2, 3)`;

b) `set(1, 2, 3, 3, 2, 4, 1) intersect set(1, 2, 1, 3, 1, 4)` resulta em `set(1, 2, 3, 4)`;

c) `list('a', 'b', 'c', null) intersect list(null, 'null', 'x')` resulta em `null`;

d) `null intersect list(2, 1)` resulta em `null`;

e) `set(-1, 0, null, 1) intersect null` resulta em `null`;

f) `null intersect null` resulta em `null`.

4.5.3.1.19 Excetuação de coleções

Sintaxe: ***`operando`*** **`except`** ***`operando`***

É uma lista contendo os elementos presentes no operando da esquerda, exceto os que estão presentes no operando da direita, os quais devem ser coleções cujo tipo dos elementos deve ser compatível. Se o operando da esquerda for `null`, ou se a lista resultante ficar vazia, o resultado será `null`. Se o operando da direita for `null`, o resultado será uma lista contendo os

elementos do operando da esquerda. Exemplos:

a) `list(1, 3, 2) except list(2, 3, 5)` resulta em `list(1)`;

b) `set(1, 2, 3, 3, 2, 4, 1) except set(1, 2, 1, 3, 1, 4)` resulta em `null`;

c) `list('a', 'b', 'c', null) except list('null', 'x')` resulta em `list ('a', 'b', 'c', null)`;

d) `null except list(2, 1)` resulta em `null`;

e) `set(-1, 0, null, 1) except null` resulta em `list(-1, 0, null, 1)`;

f) `null except null` resulta em `null`.

4.5.3.1.20 pertence-a-coleção e não-pertence-a-coleção

Sintaxe do pertence-a-coleção: ***operando* `in` *operando***

Sintaxe do não-pertence-a-coleção: ***operando* `not in` *operando***

O pertence-a-coleção resulta em `true` se um dos elementos do operando da direita, que deve ser uma coleção, é igual ao valor do operando da esquerda. Senão resulta em `false`. O não-pertence-a-coleção resulta em `false` se nenhum dos elementos do operando da direita, que deve ser uma coleção, é igual ao valor do operando da esquerda.

Exemplos:

a) `4 in list(1, 2, 3, 4)` resulta em `true`;

b) `3 not in bag(null, 1, 3, 5)` resulta em `false`;

c) `f in Funcionarios` resulta em `true` se a coleção `Funcionarios` possui uma referência para o objeto referenciado por `f`, em `false` se não possui.

d) `null in list(1, 2, 3)` resulta em `false`;

e) `null in list(1, 2, null, 3)` resulta em `true`;

f) `4 in null resulta em false`.

Observa-se que `x not in y` sempre resulta em `not (x in y)`.

4.5.3.2 Expressões unárias

4.5.3.2.1 Negação lógica

Sintaxe: `not` ***operando***

É a negação do operando, que deve ser lógico. Se o valor do operando for `false`, resulta em `true`. Se for `true`, resulta em `false`. Se for `null`, resulta em `null`. Exemplos:

a) `not false` resulta em `true`;

b) `not true` resulta em `false`;

c) `not (a = b)` resulta em `a <> b`;

d) `not null` resulta em `null`.

4.5.3.2.2 Oposição numérica

Sintaxe: `-operando`

É o oposto numérico do operando, que deve ser um número. Exemplos:

a) `-5` resulta em `-5`;

b) `--5.7` resulta em `5.7`;

c) `-x` resulta em `0 - x`;

d) `-0` resulta em `0`;

e) `-null` resulta em `null`.

4.5.3.2.3 Reforço de sinal numérico

Sintaxe: `+operando`

Resulta no valor exato do operando (que deve ser um número), sem realizar qualquer operação sobre ele. É suportado para fins de legibilidade. Exemplos:

a) `+5` resulta em `5`;

b) `+-5` resulta em `-5`;

c) `+x` resulta em `x`;

d) `+null` resulta em `null`.

4.5.3.3 Expressão condicional

Sintaxe:

```
case [seletor]
[ when [condição-n] then [resultado-n] ]...
[ else resultado-padrão ]
end
```

Se `seletor` for informado, então o valor dele é comparado com o valor de cada `condição-n`. No primeiro caso em que um valor igual for encontrado, a expressão resulta no `resultado-n` correspondente. O operando `seletor` deve ser numérico, lógico, textual, uma data, ou uma referência para objeto, e o tipo de todas as condições em condição-n deve ser igual ao tipo de `seletor`.

Se ***seletor*** não for informado, então cada ***condição-n*** é avaliada. No primeiro caso em que a condição resulta em **true**, a expressão resulta no ***resultado-n*** correspondente. Todas as condições em ***condição-n*** devem ser lógicas.

Se não for encontrado qualquer caso para algum resultado-n, a expressão resulta em ***resultado-padrão*** se houver, ou em **null** se não houver.

Exemplos:

a) **case 2 when 1 then 'A' when 2 then 'B' when 3 then 'C' end** resulta em **'B'**;

b) **case when f.datNas > date(1975) then 'Senhor' else 'Cara' end** resulta em **'Senhor'** ou **'Cara'**, dependendo da data de nascimento do funcionário referenciado por **f**.

Observa-se o seguinte:

a) **case a when x1 then y1 when x2 then y2 end** sempre resulta em **case when a=x1 then y1 when a=x2 then y2 end**;

b) **case a when x then y else null end** é uma redundância, pois **else null** é desnecessário;

c) **case null when x then y else z end** sempre resulta em **null** e não em **z**, pois é impossível determinar se **null<>x**.

4.5.3.4 ausência-de-valor e presença-de-valor

Sintaxe da ausência-de-valor: ***operando*** **is null**

Sintaxe da presença-de-valor: ***operando*** **is not null**

A ausência-de-valor resulta em **true** se o valor do operando for **null**, e **false** se não for **null**. A presença-do-valor resulta em **true** se o operando não for **null**, **false** se for **null**. O operando pode ser de qualquer tipo, inclusive referência ou coleção. Exemplos:

a) **1 is null** resulta em **false**;

b) **1 is not null** resulta em **true**;

c) **date(1977,12,3) is null** resulta em **false**;

d) **date(1977,12,3) is not null** resulta em **true**;

e) **null is null** resulta em **true**;

f) **null is not null** resulta em **false**;

g) `x is null` resulta em `true` se `x` é nulo (não contém valor), e em `false` se `x` é não nulo (contém valor);

h) `x is not null` resulta em `false` se `x` é nulo (não contém valor), e em `true` se `x` é não nulo (contém valor).

Observa-se o seguinte:

a) `x is null` nunca retorna `null`, assim como `x is not null`;

b) `x is not null` e `not (x is null)` sempre retornam o mesmo valor;

c) O trecho `is null` não é o operador `is` aplicado ao operando `null` – trata-se de um predicado, assim como o trecho `is not null`.

4.5.3.5 Acesso a membro de objeto

Sintaxe: ***`operando.nome`***

O operando deve ser uma referência para objeto, e ***`nome`*** deve ser o nome de um membro do objeto referenciado. Resulta no valor deste membro. Exemplos:

a) `x.a` resulta no valor do membro `a`, para o objeto referenciado por `x`; se `x` for `nulo`, resulta em `null`;

b) `x.a.b` resulta em `(x.a).b`.

Observa-se que:

a) se a referência for nula, o resultado é `null`, sem que seja gerado um erro;

b) `null.x` é um erro de sintaxe, pois `x` não é membro do tipo nulo.

4.5.3.6 Filtragem de coleção

Sintaxe: ***`operando`*** `where` ***`filtro`***

O operando deve ser uma coleção, e ***`filtro`*** deve ser uma expressão que resulte num valor lógico. Cada elemento da coleção é usado como escopo da expressão ***`filtro`***. A filtragem resulta numa lista contendo somente os elementos para os quais ***`filtro`*** resultou em `true`. Exemplos:

a) `list(1, 2, 3, 4) where $value > 2` resulta em `list(3, 4)`;

b) `Funcionarios where datNas < day(1970, 1, 1)` resulta na lista dos funcionários que nasceram antes de 1 de Janeiro de 1970;

c) `Veiculos where fabricante.nome = 'BMW'` resulta nos veículos cadastrados que foram fabricados pela BMW;

d) `bag(1, 1, 2, 3, 3, 4, 4, 5) where $value mod 2 = 1` resulta em `list(1, 1, 3, 3, 5)`;

e) `list(set(1, 2), set(2, 3, 4), set(1, 3, 4)) where 3 in $value` resulta em `list(set(2, 3, 4), set(1, 3, 4))`;

f) `set('banana', 'laranja', 'limão') where $value = 'abacaxi'` resulta em `null`.

Observa-se o seguinte:

a) caso o tipo dos elementos da coleção não seja referência, usa-se a palavra `$value` para denotar valor do elemento;

b) `x where y = 1` e `x where $value.y = 1` resultam no mesmo valor;

c) `x where true` resulta numa lista contendo todos os elementos de `x`;

d) `x where false` resulta em `null`, assim como `x where null`;

e) `null where x` resulta em `null`.

4.5.3.7 Ordenação de coleção

Sintaxe: ***operando*** `order by` ***ordenador-1*** `[asc|desc][,` ***ordenador-n*** `[asc|desc]...]`

O operando deve ser uma coleção. Os ordenadores devem ser expressões que retornam um valor lógico, ou um número, ou uma data, ou um texto. O escopo dos ordenadores é cada elemento da coleção.

O resultado é uma lista contendo todos os elementos do operando, ordenados primeiro pelo ordenador 1, depois pelo 2 (se houver), e assim por diante até o ordenador n (enquanto houver). Se um ordenador for seguido por `desc` no comando, a ordenação do mesmo será ascendente. Caso contrário será ascendente. O `asc` é suportado para fins de legibilidade.

Se um dos ordenadores retorna `null` para determinado elemento, aquele elemento não pode ser ordenado e fica fora da coleção resultante.

Exemplos:

a) `bag(1, 2, 3, 4, 5, 6) order by $value mod 3` resulta em `list(3, 6, 1, 4, 2, 5)`;

b) `list(1, 2, 3) order by $value desc` resulta em `list(3, 2, 1)`;

c) `set(date(1999, 10, 5), date(2000, 9, 5), date(2001, 12, 3)) order by $value.day, $value.month` resulta em `list(date(2001, 12, 3), date`

`(2000, 9, 5), date(1999, 10, 5))`;

d) `Funcionarios order by salario desc` resulta na lista dos funcionários ordenada em ordem descendente de salário (os que ganham mais aparecem antes dos que ganham menos);

e) `Veiculos order by proprietario.nome` resulta na lista de veículos ordenada pelo nome de seus proprietários.

Observa-se o seguinte:

a) caso o tipo dos elementos da coleção não seja referência, usa-se a palavra `$value` para denotar valor do elemento;

b) `x order by y` e `x order by $value.y` resultam no mesmo valor;

c) `x order by c`, onde `c` é uma expressão constante, resulta numa lista contendo os elementos de `x`;

d) `x order by null` resulta em `null`;

e) `null order by x` resulta em `null`.

4.5.3.8 Mapeamento de coleção

Sintaxe: `select item-1 [nome-1] [, item-n [nome-n] ]...`
`from operando`

Se houver mais de um item precedendo a palavra `from`, mapeia o valor do operando, que deve ser uma coleção, em uma coleção de listas. Nesta coleção, haverá uma lista para cada elemento do operando. Na lista, o valor de cada elemento é o valor de um dos itens que precedem a palavra from, calculado a partir do elemento do operando. O primeiro elemento corresponde ao primeiro item da lista, e assim por diante.

Se houver apenas um item precedendo a palavra `from`, mapeia o valor do operando, que deve ser uma coleção, em outra coleção. Nesta coleção, haverá um elemento para cada elemento do operando. O valor do elemento na coleção resultante corresponde ao valor do item que precede a palavra from, calculado a partir do elemento do operando.

Exemplos:

a) `select -$value from list(1, -1, 5, 3)` resulta em `list(-1, 1, -5, -3)`;

b) `select -$value, $value from list(1, 3, -5)` resulta em `list(list(-1, 1), list(-3, 3), list(5, -5))`;

c) `select titulo.nome, nome from Funcionarios` resulta em `bag(list ('Agent', 'Smith'), list('Programador', 'Silva'))`;

d) `select nome || ' tem ' || tostr(idade) || ' anos.' from Funcionarios` resulta em `bag('Smith tem 36 anos.', 'Silva tem 18 anos.')`.

É possível nomear cada item, de modo que os elementos resultantes do mapeamento podem ser usados em outras operações de OQL como se fossem membros de um objeto. Exemplo:

a) `select nome || ' tem ' || tostr(idade) || ' anos.' frase from Funcionarios order by frase`;

b) `select nome from (select nome, (today() - datNas)/365.25 idade from Funcionarios) where idade > 20`.

Observa-se, pelo exemplo b acima, que caso o nome de um item seja omitido e sua expressão seja apenas o membro de um objeto, o nome do item é o próprio nome do membro. Se houver ambigüidade (um item nomeado tem o mesmo nome de um item que é apenas o membro de um objeto), qualquer tentativa de usar aquele nome implica em erro de semântica.

4.5.3.9 Produto cartesiano

Sintaxe: ***item-1 nome-1*** `[,` ***item-n nome-n***`]...`

Resulta em uma coleção formada pelo produto cartesiano dos itens, os quais devem todos ser coleções. Cada elemento da coleção resultante é uma lista, e cada lista contém um valor vindo dos itens, sendo que todas as combinações possíveis desses valores estarão presentes. Exemplos:

a) `list(1, 2, 3) a, list('a', 'b') b` resulta em `bag(list(1, 'a'), list(1, 'b'), list(2, 'a'), list(2, 'b'), list(3, 'a'), list(3, 'b'))`;

b) `set(0, 1) x, set(0, 1) y, set(0, 1) z order by x, y, z` resulta em `bag (list(0, 0, 0), list(0, 0, 1), list(0, 1, 0), list(0, 1, 1), list (1, 0, 0), list(1, 0, 1), list(1, 1, 0), list(1, 1, 1))`;

c) `Clientes a, Representantes b where a.cidade = b.cidade` resulta numa coleção ligando clientes a representantes pela cidade.

Observa-se que no exemplo c existe uma junção. Junções são suportadas na OQL, mas seu uso é muito menos freqüente do que na SQL, pois na primeira é possível navegar pelos objetos. Por exemplo, em SQL, uma relação de funcionários que mostre o departamento de

cado um é obtida com a seguinte junção:

```
select a.nome, b.nome from Funcionarios a, Deptos b
where a.codDepto = b.codigo
```

Em OQL, o mesmo efeito é obtido pelo seguinte comando, que não possui junção:

```
select nome, depto.nome from Funcionarios
```

A OQL até suporta a junção, porém ela fica evidentemente redundante, como neste caso:

```
select a.nome, b.nome from Funcionarios a, Deptos b where a.depto = b
```

4.5.3.10 Seleção de elemento de coleção

Sintaxe para o primeiro elemento: `first(operando)`

Sintaxe para o último elemento: `last(operando)`

Resulta no primeiro ou no último elemento do operando, que deve ser uma coleção. Exemplo:

a) `first(list(1, 2, 3))` resulta em `1`, `last(list(1, 2, 3))` resulta em `3`.

b) `first(Funcionarios order by nome)` resulta no funcionário que aparece em primeiro em ordem alfabética.

c) `last(Funcionarios order by datNas)` resulta no funcionário mais novo;

d) `first(Pedido where datEmi >= date(2003, 10, 1) order by datEmi)` resulta no primeiro pedido emitido a partir de Outubro de 2003.

Observa-se que é comum usar o operador `order by` em conjunto com a operação de seleção. Caso a coleção seja uma sacola, é escolhido um elemento de forma arbitrária. Por exemplo, `first(bag(1, 1, 2, 7, 10, 10))` pode resultar em `1`, `2`, `7` ou `10`.

4.5.3.11 Contagem, soma, mínimo, máximo, e média

Sintaxe da contagem: `count(operando)`

Resulta num número inteiro que representa quantidade de elementos não nulos que estão no operando, que deve ser uma coleção. Caso o operando seja nulo ou uma coleção composta somente por nulos, a contagem resulta em zero. Exemplos:

a) `count(bag(1, 2, 3, 3, null))` resulta em `4`;

b) `count(Funcionarios)` resulta na quantidade de funcionários da coleção `Funcionarios`.

Sintaxe da soma: `sum(operando)`

Resulta num número que representa a soma dos elementos não nulos do operando, que deve ser uma coleção de números. Caso o operando seja nulo ou uma coleção composta somente por nulos, resulta em `null`. Exemplos:

a) `sum(list(1, 2, 5, null, -3))` resulta em `1+2+5+-3`, que equivale a `5`;

b) `sum(select valor from NotasFiscais where datEmi=date(2003, 10, 1))` resulta no volume financeiro de vendas no dia 1 de Outubro de 2003.

Sintaxe da média: `avg(operando)`

Resulta num número que representa a média dos elementos não nulos do operando, que deve ser uma coleção de números. Caso o operando seja nulo ou uma coleção composta somente por nulos, resulta em `null`. Exemplos:

a) `avg(list(1, 2, 5, null, -3))` resulta em `(1+2+5+-3)/4`, que equivale a `1.25`;

b) `sum(select salario from Funcionarios)` resulta na média salarial da empresa.

Observa-se que se `count(x) = 0` então `avg(x) = null`, e se `count(x) > 0`, então a`vg(x) = sum(x) / count(x)`.

Sintaxe do mínimo: `min(operando)`

Resulta no menor valor encontrado em elementos não nulos do operando, que deve ser uma coleção de lógicos, números, textos ou datas. Caso o operando seja nulo ou uma coleção composta somente por nulos, o mínimo resulta em `null`. Exemplos:

a) `min(list(1, 2, 5, null, -3))` resulta em `-3`;

b) `min(list('C', 'D', 'E', null, 'B'))` resulta em `'B'`;

c) `min(select salario from Funcionarios)` resulta no menor salário pago a um funcionário.

Sintaxe do máximo: `max(operando)`

Resulta no maior valor encontrado em elementos não nulos do operando, que deve ser uma coleção de lógicos, números, textos ou datas. Caso o operando seja nulo ou uma coleção composta somente por nulos, o máximo resulta em `null`. Exemplos:

a) `max(list(1, 2, 5, null, -3))` resulta em `5`;

b) `max(list('C', 'D', 'E', null, 'B'))` resulta em `'D'`;

c) `max(select salario from Funcionarios)` resulta no maior salário pago a um funcionário.

Observa-se que a sintaxe destas operações difere bastante da sintaxe das mesmas na SQL. Houve preocupação em melhorar a concisão da linguagem, pois na SQL o operando que elas recebem retorna um único valor, havendo necessidade de dividir as expressões em agregadas e não-agregadas. Por exemplo, o comando SQL `select max(salario) from Funcionarios` retorna uma única linha, enquanto o comando `select trunc(salario) from Funcionarios` retorna várias linhas. Isto implica que o comando `select max(salario) + trunc(salario) from Funcionarios` não é possível, e precisa de consistência especial.

A superior concisão da OQL implica que certas operações são realizadas com mais facilidade, assim como ocorreu com as junções. Por exemplo, seja a obtenção da relação de funcionários com a quantidade de dependentes de cada um. Em SQL, isto é feito assim:

```
select codFun, count(*) from Dependentes group by codFun
```

Em OQL, o comando é este:

```
select codFun, count(deps) from Funcionarios
```

O termo `deps` é um membro dos objetos `Funcionario`, que mantém a coleção de dependentes do mesmo. Observa-se que o `group by` é desnecessário na OQL.

4.5.3.12 Agrupamento

Sintaxe do agrupamento:

```
operando group by item-1 [nome-1]
[, item-n [nome-n]]...
```

O agrupamento resulta numa coleção de listas, como no mapeamento e no produto cartesiano. Cada lista contém um elemento para cada item especificado no comando, e um elemento adicional no final, que é uma coleção dos objetos que pertencem ao grupo. Por exemplo, seja um comando que agrupa um conjunto de textos pelo tamanho dos textos. O comando que faz isto é o seguinte:

```
list('abc', 'xy', 'hei', 'lev', 'km') group by length($value)
```

E ele resulta em:

```
bag(list(2, bag('xy', 'km')), list(3, bag('abc', 'hei', 'lev')))
```

Ao todo, foram encontrados dois grupos. O primeiro grupo é o dos textos cujo tamanho é 2, e o segundo é o dos textos cujo tamanho é 3.

Diferente da SQL, os objetos usados para montar o agrupamento não são perdidos, e podem inclusive ser usados em outras operações. Por exemplo, seja um comando que agrupa

as notas fiscais pelo mês da emissão, e para cada mês apresenta a quantidade de notas, a soma dos valores das notas, e o valor da mais nota. O comando que faz isto é este:

```
select
  yearOf(mes),
  monthOf(mes),
  count(partition),
  sum(select valor from partition),
  max(select valor from partition)
from
  NotasFiscais group by firstDateOfMonth(datEmi) mes
```

Para compreender o comando, divide-se o mesmo em duas partes. A parte que será executada primeiro é a seguinte:

```
NotasFiscais group by firstDateOfMonth(datEmi) mes
```

Esta parte é que faz o agrupamento propriamente dito. Ela resulta num valor que tem o seguinte formato:

```
bag(
  list(mês-1, notas-mês-1),
  list(mês-2, notas-mês-2),
  ...
  list(mês-n, notas-mês-n)
)
```

Cada lista tem como primeiro elemento o dia do mês (vindo da expressão `firstDateOfMonth(datEmi)`) e como segundo elemento uma coleção das notas fiscais encontradas naquele mês. A exemplo do mapeamento, estes elementos podem ser usados como se fossem membros de um objeto. O primeiro elemento recebe o nome `mes`, especificado no comando. O segundo recebe o nome `partition`, que é reservado pela linguagem, e usado por definição nos agrupamentos. Como `partition` é o nome de uma coleção, é possível utilizá-lo em operações de contagem, soma, etc. Deste modo, o resultado gerado pela primeira parte do comando pode ser usado pela segunda parte, que é a seguinte:

```
select
  yearOf(mes),
  monthOf(mes),
  count(partition),
  sum(select valor from partition),
  max(select valor from partition)
from
  primeira-parte
```

Esta parte do comando é entendida como um simples mapeamento que assume a existência dos membros `mes` e `partition` em cada objeto do operando.

4.5.4 Precedência das operações

A tabela 1 mostra a precedência das operações. As operações que aparecem primeiro na relação são sempre executadas antes das que aparecem em depois.

Precedência das operações
`is classe`, `is null`, `is not classe`, `is not null`
`-` unário, `+` unário, `not` unário
`in`, `not in`
`*`, `/`, `mod`, `intersect`
`+` binário, `-` binário, `||`, `union`, `union distinct`, `except`
`>`, `<`, `>=`, `<=`
`=`, `<>`
`and`
`or`
produto cartesiano
`where`
`group by`
`order by`
`select`

Tabela 1 – Precedência entre as operações da OQL.

Quando uma parte da expressão possui duas operações seguidas e com a mesma precedência, a execução ocorre da esquerda para direita.

A precedência pode ser explicitada pelo uso dos parênteses, como é mostrado em vários exemplos deste texto.

4.5.5 Funções escalares da OQL

A tabela 2 mostra a relação das principais funções escalares da OQL.

Função	Resultado
`getDate(ano, mes, dia)`	Data montada a partir dos operandos.
`yearOf(data)`	Ano gregoriano de `data` (número).
`monthOf(data)`	Mês gregoriano de `data` (número de 1 a 12).
`dayOf(data)`	Dia gregoriano de `data` (número de 1 a 31).
`firstDateOfYear(data)`	Primeira data do ano de `data`.
`firstDateOfMonth(data)`	Primeira data do mês de `data`.
`incYear(data, anos)`	`data` com o ano incrementado em `anos` (ou decrementado, se `anos` for negativo).
`incMonth(data, meses)`	`data` com o mês incrementado em `meses` (ou decrementado, se `meses` for negativo).
`toStr(valor)`	`valor` convertido para um texto.
`toNumber(texto)`	`texto` convertido para um número usando a base 10.
`trunc(numero)`	Parte inteira de `numero`.
`substr(texto, inicio, tam)`	Trecho de `texto`, iniciando em `inicio` e com tamanho `tam`. A posição do primeiro caractere é 1.

Tabela 2 – Funções escalares da OQL.

4.6 CRIAÇÃO, MODIFICAÇÃO, E DESTRUIÇÃO DE OBJETOS

Uma vez que o modelo de dados feito pelo analista foi incorporado ao ambiente, o SGBDOO entra em produção, e pode-se criar objetos nele. Os objetos criados têm uma vida na qual podem ser consultados e modificados. Uma vez que um objeto se torna inútil, ele pode ser destruído.

As operações de criar, modificar, e destruir objetos, só podem ser feitas por programas rodando no ambiente Java. Estes programas precisam ser desenvolvidos e colocados em execução para que estas operações se realizem. E com isso surge o papel do programador.

4.6.1 Representação dos objetos na linguagem Java

No programa, objetos persistentes são representados por objetos Java. Por exemplo, se o analista modelou uma classe chamada `Funcionario`, haverá uma classe Java também chamada de `Funcionario`. Assim, é possível ter uma referência da linguagem Java para um objeto `Funcionario`, e através dela é possível acessar todos membros do objeto, e inclusive navegar por objetos referenciados. O SGBDOO está fortemente acoplado ao ambiente Java, e quase se tem a impressão de que ele é parte da própria plataforma, embora não passe de uma

biblioteca de classes.

Para cada classe modelada, existe uma interface Java que deriva de `IPersistent`. Por ser uma interface, não é possível utilizar o operador `new` da linguagem Java para instanciar um objeto. Deve-se usar o método `newInstance(Class)`, da classe `IObjectContext`. O objeto não é persistido simplesmente por ser instanciado. Ele só será persistido quando for chamado o método `save()` no objeto. O quadro 3 mostra a criação de um objeto.

```
void criaFuncionario(IObjectContext db) throws Exception {

    // Cria uma nova instância.
    Funcionario f =
      (Funcionario) db.newInstance(Funcionario.class);

    // Seta o valor dos atributos.
    f.setNome("Smith");
    f.setTitulo("Agent");
    f.setDatNas(GregorianDates.newDate(1968, 10, 2));

    // Persiste a instância.
    f.save();

}
```

Quadro 3 – Exemplo de criação de objeto no SGBDOO.

Objetos persistidos podem ser consultados através do método `getObject(Class, String)`, da classe `IObjectContext`. O primeiro parâmetro que o método recebe é a classe do objeto desejado, e o segundo é uma expressão OQL que serve para identificar o objeto desejado. Um exemplo disso pode ser visto no quadro 4.

```
void alteraSmith(IObjectContext db) {

  Funcionario f = (Funcionario) db.getObject(
    Funcionario.class, "nome='Smith'");
  f.setTitulo("Virus");
  f.save();

}
```

Quadro 4 – Exemplo de consulta de objetos no ambiente Java.

Quando um objeto é persistido no SGBDOO, ele não é perdido até que seja destruído (a menos, é claro, que ocorra um dano no hardware ou nos arquivos que mantém os dados). Em outras palavras, a existência de um objeto persistido é preservada mesmo que ele tenha sido persistido por uma execução do programa, e consultado numa execução subseqüente,

sendo ambas separadas por um espaço de tempo indefinido. Isto também ocorre se a primeira execução terminou de forma abortiva.

Pode haver mais de um processo (execução do programa Java) rodando ao mesmo tempo. Neste caso, se um objeto é persistido por um processo, o mesmo objeto poderá ser consultado por qualquer processo ativo que esteja usando a mesma instância do SGBDOO. Isto é observado inclusive quando os processos estão rodando em computadores diferentes.

Conclui-se que os objetos Java não são os objetos persistentes em si, mas sim um meio conveniente de acessá-los.

4.6.2 Ciclo de vida dos objetos

Um objeto é criado com o método `newInstance(Class)`, da classe `IObjectContext`, mas isto por si só não o persiste. O método `save()` do objeto deve ser invocado para que ele seja persistido.

Uma vez que foi persistido, o objeto fica disponível para consultas e modificações enquanto for necessário. Caso seja necessário modificar um objeto, basta obter uma referência para ele (o que será mostrado em seguida), modificar os atributos desejados, e salvá-lo novamente com o método `save()`. A operação que o método `save()` realiza é análoga à operação de salvar um arquivo. Ou seja, o objeto só será persistido na primeira vez que o `save()` for invocado. Nas demais vezes, o objeto será apenas atualizado.

Se algum atributo do objeto é modificado sem que seja chamado o método `save()` após esta modificação, ela não surtirá efeitos para outros processos. Observa-se que o método `save()` sempre deve ser chamado de forma explícita. O software nunca salva objetos implicitamente.

Quando um objeto se torna inútil (porque não é referenciado por qualquer outro objeto), o software o destrói automaticamente. Portanto, quando um objeto é persistido pela primeira vez, o SGBDOO exige que ele seja referenciado ou por uma coleção global, ou por outro objeto (caso contrário o objeto pode ser destruído imediatamente após ser persistido).

Um objeto também pode ser destruído explicitamente, chamando-se o método `destroy()` dele. Sempre que isto ocorre, o software verifica se o objeto está sendo referenciado por outro, ou por uma coleção global. Se for este o caso, o método `destroy()` lançará uma exceção.

Não é comum destruir um objeto explicitamente, especialmente se o SGBDOO consegue fazer isto de forma automática. O mais comum é simplesmente remover o objeto de uma coleção onde ele é armazenado. Isto é suficiente para que o objeto não apareça mais em consultas feitas naquela coleção. Observe que a integridade referencial é respeitada, pois quando um objeto é removido de uma coleção, ele não é necessariamente destruído. Assim, ele continuará disponível através de outra coleção ou de uma referência vinda de outro objeto.

4.6.3 Contexto de objetos

Para acessar o SGBDOO, é necessário ter um contexto de objetos, representado por uma instância da interface `IObjectContext`. Sem ele, é impossível fazer qualquer operação no SGBDOO. As operações mais usadas no `IObjectContext` são as seguintes:

a) criação de objetos, feita através do método `newInstance(Class)`;
b) consultas, feitas através dos métodos g`etObject(Class, String)`, `findObject(Class, String)`, `query`*`Sufixo`*`(`*`args`*`)`, e `newQuery(`*`args`*`)`;
c) criação de novos contextos, feita através do método `newContext()`;
d) criação de contextos de transação, feita através do método `newTransaction()`.

A criação de objetos é bastante simples e foi exemplificada no tópico anterior. Os aspectos mais importantes das demais operações são apresentados em seguida.

4.6.4 Consultas simples

A consulta (ou obtenção) de um único objeto pode ser feita através dos métodos `getObject(Class, String)` e `findObject(Class, String)`, ambos de `IObjectContext`. O primeiro parâmetro indica a classe do objeto desejado, e o segundo deve conter uma expressão em OQL usada para identificar o objeto entre os demais daquela classe. A diferença é que, se o objeto não for encontrado, o `getObject` lança uma exceção, enquanto o `findObject` retorna `null`.

Outros tipos de consultas podem ser feitas pelos métodos `query`*`Sufixo`*`(`*`args`*`)`, e `newQuery(`*`args`*`)`. A versão mais simples do `query` é `query(String)`. Este método retorna uma coleção do Java (objeto `Collection`). O quadro 5 exemplifica o método.

```
void imprimeAgentes(IObjectContext db) {

  Collection agents = db.query(
    "Funcionarios where titulo='Agent'");

  for (Iterator i = agents.iterator(); i.hasNext();) {

    Funcionario agente = (Funcionario) i.next();
    System.out.println(agente.getNome());

  }

}
```

Quadro 5 – Exemplo do método **`query(String)`**.

Caso o resultado não seja uma coleção, pode-se usar o método **`queryO(String)`**, que retorna apenas um **`Object`**, conforme mostrado no quadro 6.

```
void imprimeQtdAgentes(IObjectContext db) {

  Number qtd = (Number) db.queryO(
    "count(Funcionarios where titulo='Agent')");

  System.out.println("Existem " + qtd + " agentes.");

}
```

Quadro 6 – Exemplo do método **`queryO(String)`**.

Observa-se que quando o tipo do resultado é conhecido (como no exemplo do quadro 6), pode-se fazer um *typecast* para o tipo conhecido. É necessário prever o caso de uma expressão resultar em **`null`**, como ocorre no quadro 7.

```
void imprimeMediaSal(IObjectContext db) {

  Object obj = db.queryO(
    "sum(select salario from Funcionarios) / " +
    "count(select salario from Funcionarios)");

  if (obj != Null.SINGLETON) {
    System.out.println("Média salarial: " + obj);
  } else {
    System.out.println("Não existem salários para " +
                       "computar a média.");
  }

}
```

Quadro 7 – Exemplo do **`queryO(String)`** que pode resultar em **`null`**.

Observa-se que quando uma expressão OQL resulta em `null`, não é retornado `null` do Java, e sim `Null.SINGLETON` (`Null` é uma classe Java, e `SINGLETON` é um atributo estático que guarda a única instância daquela classe). Isto é necessário porque o `null` do Java tem um comportamento diferente do `null` da OQL.

4.6.5 Consultas otimizadas

Num SI, há dois tipos de consultas que exigem um modelo de execução mais otimizado:

a) consultas que utilizam dados externos (parâmetros);

b) consultas que precisam ser feitas várias vezes, com parâmetros diferentes.

Para estas consultas, usa-se uma instância da interface `IQuery`, que é obtida através do método `newQuery()` de `IObjectContext`. Um objeto `IQuery` possui meios para especificar parâmetros numa consulta, e para executar uma consulta várias vezes sem processar o comando OQL a cada execução. O quadro 8 mostra um exemplo disto.

```
void mostraFuncionarios(IObjectContext db, int[] empresas) {

  IQuery q = db.newQuery();
  q.setStmt("Funcionarios where empregador.codigo=:0");

  for (int i = 0; i < empresas.length; ++i) {

    System.out.println("Empresa " + empresas[i] + ":");
    q.setParam(0, new Integer(empresas[i]));

    for (Iterator j = q.run().iterator(); j.hasNext(); ++j) {
      Funcionario f = (Funcionario) j.next();
      System.out.println("  " + j.getNome());
    }

  }

}
```

Quadro 8 – Exemplo de uso da `IQuery` com parâmetros e múltiplas execuções.

4.6.6 Criação de novos contextos

Existem diversas razões para se fazer uso de *multithreading* num programa. Por exemplo, o programa pode rodar no servidor e atender a vários usuários simultaneamente, ou executar certas tarefas em *background* de modo a permitir que o usuário trabalhe enquanto algum processo mais demorado é realizado. O software permite isto através da criação de novos contextos de objetos.

Uma instância de `IObjectContext` não é *thread safe*, ou seja, se duas ou mais *threads* concorrentes usarem a mesma instância de `IObjectContext`, é quase certo que haverá funcionamento errático e corrompimento das estruturas de controle internas. Portanto o programador deve obter uma instância diferente para cada *thread* concorrente. Foi concebido assim por questão de simplicidade na implementação e performance.

O programador usa o método `newContext()`, de `IObjectContext`, para obter um novo contexto. Um exemplo disso é mostrado no quadro 9.

```
void iniciaCalculo(IObjectContext db) {

  // Cria uma instância da thread.
  Thread calculo = new ThreadCalculo();

  // Cria um novo contexto, e atribui à thread.
  IObjectContext dbCalculo = db.newContext();
  calculo.setContext(dbCalculo);

  // Inicia a thread.
  calculo.start();

  // Neste ponto, a thread está executando e usando um
  // contexto criado só para ela.

}
```

Quadro 9 – Exemplo de criação de novo contexto.

4.6.7 Transações

Transações fornecem atomicidade a um conjunto de operações feitas no SGBD, no sentido de que ou todas as operações são confirmadas, ou nenhuma é. O SGBDOO suporta transações com os requisitos ACID (Atomicidade, Consistência, Isolamento, e Durabilidade). Para criar transações, usa-se o método `newTransaction()` de `IObjectContext`.

O método `newTransaction()` retorna uma instância da interface `ITransaction`. Os aspectos mais relevantes de `ITransaction` são os seguintes:

a) deriva de `IObjectContext`, ou seja, uma transação é em primeiro lugar um contexto de objetos;
b) possui o método `commit()`, que serve para confirmar as operações e finalizar a transação;
c) possui o método `rollback()`, que serve para descartar as operações e finalizar a transação.

O fato de derivar de **`IObjectContext`** traz uma implicação importante: quando uma transação é criada, existem dois contextos ativos: o que recebeu a chamada a **`newTransaction()`**, e o que foi retornado por esta chamada. Estes dois contextos são independentes um do outro, e em geral, alterações feitas em objetos obtidos por um contexto não são percebidas no outro. Isto implica que todo objeto modificado no contexto da transação precisa ser lido em tal contexto. O quadro 10 traz um exemplo disto.

```
void fazSmithAgente(IObjectContext db) {

  // Lê a primeira vez fora da transação.
  Funcionario smith = (Funcionario) db.getObject(
    Funcionario.class, "nome='Smith'");

  if (smith.getTitulo() != "Agent") {

    ITransaction trans = db.newTransaction();

    // Lê novamente, dentro da transação.
    smith = (Funcionario) db.getObject(
      Funcionario.class, "nome='Smith'");
    smith.setTitulo("Agent");
    smith.save();

    ...


  }

}
```

Quadro 10 – Exemplo leitura de objetos numa transação.

Esta concepção força a seguinte prática de programação: a leitura dos objetos a serem modificados é feita no contexto da transação, e não fora dele. Isto deixa os programas melhor preparados para ambientes de alta concorrência.

A transação se inicia imediatamente antes da primeira operação feita no contexto da transação. Em outras palavras, uma chamada ao **`newTransaction()`** não inicia a transação (apenas cria um contexto para ela). A transação se inicia quando é feita a primeira leitura ou salvamento de objeto naquele contexto, e existe até que ocorra uma chamada a **`commit()`** ou **`rollback()`**.

Alterações feitas no contexto da transação não são confirmadas nem visíveis em outros contextos até que seja chamado o método **`commit()`**. Somente quando a chamada ao **`commit()`** retorna, existe a certeza de que as alterações foram confirmadas, e estão visíveis para os demais contextos. Se for chamado o método **`rollback()`**, todas as alterações são descartadas,

como se nunca tivessem sido realizadas.

Imediatamente após o retorno do método **`commit()`** ou **`rollback()`**, não há transação ativa. O contexto da transação fica como se tivesse apenas sido criado, e portanto, a primeira operação feita com ele a partir deste momento implicará no início da transação.

O SGBDOO suporta transações aninhadas. Para isto basta chamar o método **`newTransaction()`** em um contexto de transação. Quando uma transação aninhada está em vigência, a transação mais externa fica congelada, e qualquer operação feita no contexto dela causa o lançamento de uma exceção, pois se constitui num erro de programação. É necessário que seja deste modo porque transações aninhadas obedecem a regra dos programas estruturados, na qual o bloco de comandos externo só continua sua execução quando o bloco interno terminar.

4.6.8 Operações em coleções globais

Quando um objeto é persistido pela primeira vez, ele é adicionado automaticamente em determinadas coleções globais, definidas na modelagem. Após isto, ele pode ser explicitamente removido de coleções onde esteja, ou adicionado em coleções onde não esteja. Para isto, cada classe Java associada a objetos persistentes possui os métodos **`addTo`*`Coleção`*`()`** e **`removeFrom`*`Coleção`*`()`**, para cada coleção onde aquele objeto possa estar.

Por exemplo, seja a classe **`Funcionario`**. Ela define as coleções **`Funcionarios`**, **`Aposentados`**, e **`EmFerias`**. Isto implica que os métodos mostrados no quadro 11 farão parte da classe Java para **`Funcionario`**.

```
boolean addToFuncionarios();
boolean removeFromFuncionarios();

boolean addToAposentados();
boolean removeFromAposentados();

boolean addToEmFerias();
boolean removeFromEmFerias();
```

Quadro 11 – Métodos que operam nas coleções de uma classe de exemplo.

A coleção **`Funcionarios`** é a coleção padrão (objetos persistidos pela primeira vez são automaticamente adicionados nela). Já as coleções **`Aposentados`** e **`EmFerias`** só conterão

objetos adicionados explicitamente nelas. O exemplo do quadro 12 mostra operações feitas na coleção **`EmFerias`**.

```
void feriasParaSmith(IObjectContext db) {

  Funcionario smith = (Funcionario) db.getObject(
    Funcionario.class, "nome='Smith'");
  smith.addToEmFerias();

}

void smithSaiDasFerias(IObjectContext db) {

  Funcionario smith = (Funcionario) db.getObject(
    Funcionario.class, "nome='Smith'");
  smith.removeFromEmFerias();

}
```

Quadro 12 – Exemplo de uso de métodos que operam em coleções globais.

Os métodos retornam um valor lógico (**`boolean`**). O valor retornado é **`true`** se a operação teve sucesso, e **`false`** se ela era redundante (ou seja, se houve tentativa de adicionar um objeto que já estava na coleção, ou de remover um objeto que não estava).

A chamada do método que opera numa coleção implica numa chamada ao método **`save()`**. Em essência, os dois métodos mostrados no quadro 13 fazem a mesma coisa.

```
void m1(Funcionario f) {
  f.removeFromAposentados();
}

void m2(Funcionario f) {
  f.save();
  f.removeFromAposentados();
}
```

Quadro 13 – Exemplo de redundância na chamada ao método **`save()`**.

A única diferença é que **`m2`** é um pouco mais lento.

Observa-se que um objeto pode estar ao mesmo tempo em mais de uma coleção, e que o fato dele não estar presente em qualquer coleção não implica que é inútil ou inacessível (conforme explicado no capítulo anterior).

4.7 INSTALAÇÃO

A instalação é um processo relativamente complexo, que seria mais simples se houvesse um instalador. Como implementar tal instalador foge do escopo deste trabalho, este tópico especifica como deve ser feita a instalação.

4.7.1 Componentes

Os seguintes componentes são necessários para que o software funcione:

a) arquivos com extensão `xcls`, que representam o modelo lógico da base de dados;
b) utilitário para geração de mapeamentos;
c) utilitário para geração de script DDL para o SGBDR;
d) utilitário para geração de classes Java a partir do modelo lógico;
e) biblioteca de execução;
f) driver JDBC do SGBDR;
g) ferramenta de consultas interativas.

Assume-se que o SGBDR esteja instalado e rodando.

4.7.2 Criação de tabelas no SGBDR

O primeiro passo é criar as tabelas no SGBDR. Isto é feito em três etapas:

a) geração de mapeamento;
b) geração de script DDL;
c) execução do script DDL.

Na geração de mapeamento, é gerado o arquivo de mapeamento. Trata-se de um arquivo no formato XML, que contém a definição abstrata das tabelas no SGBDR. Para cada tabela, é especificado o nome físico dela, as classes que fazem parte dela, e o nome físico dos campos que se originam dos atributos. O utilitário de geração de mapeamento gera um arquivo padrão, que mapeia uma classe por tabela. É possível modificar o arquivo de mapeamento manualmente, de maneira a se obter tabelas diferentes.

O arquivo de mapeamento gerado na etapa anterior é usado como entrada na ferramenta de geração do script DDL. Esta ferramenta gera um arquivo que contém comandos DDL (como CREATE TABLE), capazes de criar todas as tabelas do banco, com os OIDs, as consistências de unicidade, as chaves estrangeiras, as tabelas intermediárias, etc. Também é gerado um script padrão, que pode ser modificado manualmente.

Feito isto, basta executar o script DDL no banco. Se não houverem tabelas com nome conflitante, todas as tabelas da base de dados serão criadas, e o SGBDR estará pronto para persistir os dados.

Ressalta-se que os arquivos de mapeamento e o script DDL gerados são completos e funcionais, não sendo obrigatório alterá-los. A única razão deles serem gerados é fornecer uma abertura para otimizações.

4.7.3 Incorporação

A incorporação consiste em criar classes Java para serem ligadas ao SI. Estas classes são criadas a partir do modelo lógico (arquivos `xcls`) e do arquivo de mapeamento usados. O arquivo de mapeamento é usado durante a criação das classes pois o método `save()` de cada classe é gerado conforme as tabelas definidas no SGBDR.

São gerados dois conjuntos de classes: interfaces (que dependem exclusivamente do modelo lógico), e classes que implementam estas interfaces (que dependem do modelo lógico e do mapeamento). Os programadores do SI usam somente as interfaces, de modo que jamais sofrem influência de uma instalação em particular do SI. As classes que implementam as interfaces são usadas internamente, para realizar o acesso explícito ao SGBDR.

4.7.4 Ativação do SGBDOO

O SGBDOO é ativado automaticamente, junto com o SI, não sendo necessário deixá-lo rodando na máquina como um serviço. Quando um contexto é criado, o SGBDOO solicita uma conexão ao SGBDR do *pool* de conexões. A partir deste ponto, os acessos feitos ao SGBDOO podem implicar em acessos feitos ao SGBDR, de modo transparente para o usuário.

4.7.5 Ferramenta de consultas interativa

Existe uma ferramenta pronta para realização de consultas interativas, que usa a entrada e a saída padrões (console). Basta rodar esta ferramenta que o SGBDOO é ativado automaticamente, como se ela fosse um SI.

Esta ferramenta permite que o usuário digite comandos OQL para serem executados imediatamente. O quadro 14 mostra a saída gerada pela ferramenta após a execução de alguns comandos.

```
oql> select nome from Funcionarios
Smith
Silva

oql> count(Funcionarios)
2

oql> |
```

Quadro 14 – Exemplo de saída da ferramenta de consultas interativas.

4.8 OTIMIZAÇÃO

A otimização consiste em interferir no processo de criação das tabelas no SGBDR, de maneira a preparar o SGBDR para os volumes de dados e consultas que surgirão enquanto o SGBDOO for utilizado.

Há duas oportunidades de otimização: interferênca no mapeamento e interferência no script DDL.

4.8.1 Interferência no mapeamento

O mapeamento padrão usa a técnica "uma tabela por classe". Sabe-se que a melhor técnica é a "uma tabela por grupo de classes", porém a ferramenta de geração do mapeamento não tem como identificar os grupos de classes candidatos a se tornarem tabelas. Assim, é fornecida a oportunidade de utilizar esta técnica manualmente, através da edição do arquivo de mapeamento padrão. Mas geralmente a maior parte do arquivo fica intacta, pois raras são as hierarquias grandes o suficiente para apresentarem problemas de performance.

Uma vez alterado o arquivo de mapeamento, é necessário tomar cuidado para não gerá-lo novamente. Se houverem mudanças no modelo lógico, é provável que o arquivo de mapeamento precise de alterações para refletir as mudanças.

4.8.2 Interferência no script DDL

Consiste em alterar o script DDL de modo que a base de dados esteja melhor preparada para o volume de dados envolvido e para que os campos mais usados como filtro de consultas sejam indexados. Em outras palavras, admite-se alterar os parâmetros de criação das tabelas e a criação de índices não-exclusivos (para quer não surjam restrições que não foram planejadas pelo analista).

As seguintes alterações não podem ser feitas no script DDL:

a) remoção de atributos das tabelas, nem de tabelas inteiras;
b) remoção de *constraints* de qualquer tipo;
c) inclusão de novas *constraints*;
d) alteração do tipo de dados dos atributos, exceto se o novo tipo for compatível com o anterior;
e) renomeação de qualquer elemento.

Estas alterações estão proibidas porque em geral implicam na perda de interoperabilidade entre o software e o SGBDR.

5 IMPLEMENTAÇÃO

Este capítulo apresenta de forma detalhada os aspectos mais relevantes da implementação do software. A implementação se baseia na especificação já apresentada, de maneira que este capítulo apresenta somente aspectos internos da implementação de forma bastante objetiva.

5.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

Foram utilizadas as seguintes técnicas e linguagens para implementar o software:

a) a metodologia orientada a objetos;
b) a metodologia de desenvolvimento em espiral, na qual a especificação e a implementação são construídas em conjunto;
c) técnicas de compilação e análise de programas compilados, usadas para implementar o processador de OQL e a montagem de plano de execução;
d) a linguagem UML, usada na maioria dos diagramas deste texto;
e) a linguagem Java (SUN MICROSYSTEMS, 2003), usada para implementar todos os programas e algoritmos;
f) a linguagem SQL, usada para geração de scripts e comandos que interagem com o SGBDR.

As seguintes ferramentas foram fundamentais para este trabalho:

a) Poseidon for UML, versão 1.6 CE (GENTLEWARE, 2003) – ferramenta para criação de diagramas UML;
b) Eclipse, versão 3.0 (ECLIPSE.ORG, 2003) – ferramenta para edição, compilação e depuração de programas Java;
c) JavaCC, versão 2.1 (JAVA.NET, 2003) – ferramenta para geração de *parsers* LLx, usada para gerar o *parser* da OQL;
d) Velocity, versão 1.3 (ASF, 2003) – ferramenta para geração de texto a partir de modelos, usada para gerar os fontes Java;
e) MySQL, versão 4.0 (MYSQL AB, 2003) – SGBDR usado na implementação de referência.

Nenhuma delas apresentou problemas que prejudicassem os progressos do trabalho.

5.2 PRINCIPAIS PARTES DA IMPLEMENTAÇÃO

O software é formado pelos pacotes mostrados na figura 11.

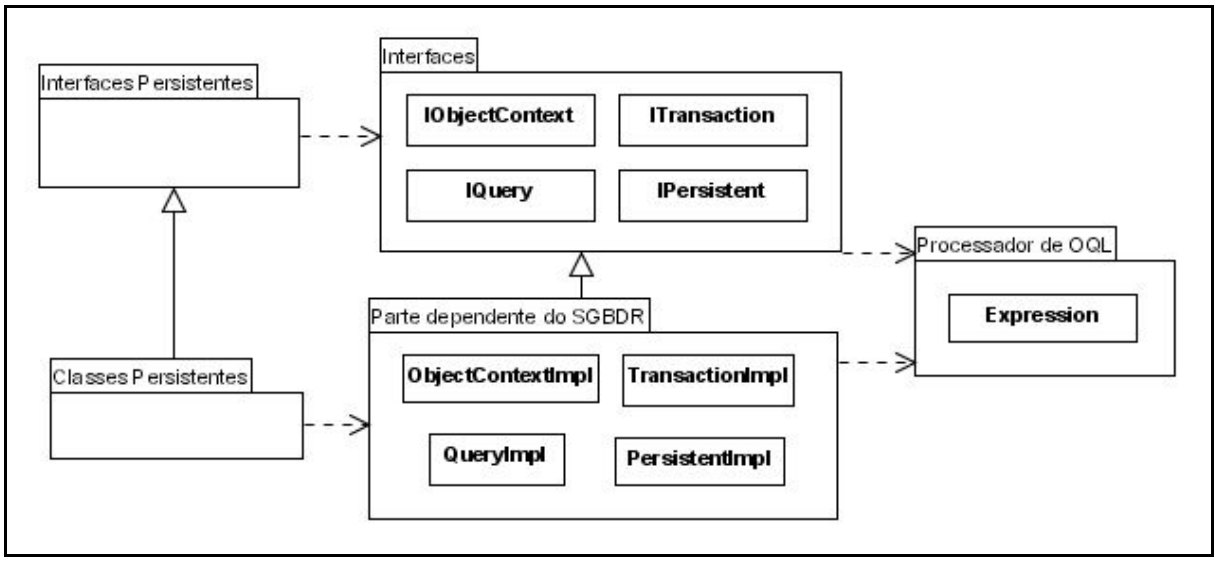

Figura 11 – Pacotes que formam a implementação do software.

Além destes pacotes, existem os seguintes aplicativos utilitários:

a) gerador do mapeamento padrão;
b) gerador de fontes Java a partir do modelo de dados;
c) gerador do script DDL para criação de entidades no SGBDR;
d) ferramenta de consultas interativa.

Estes aplicativos não fazem parte do núcleo do software e portanto a implementação dos mesmos é descrita apenas brevemente.

5.3 PROCESSADOR DE OQL

O processador de OQL é responsável pelo *parsing*, análise semântica, e execução de consultas *ad hoc* expressas em OQL. Ele é uma peça completamente independente, podendo inclusive ser utilizado por outro software como um interpretador de expressões genérico.

O processador de OQL é independente da origem dos dados sendo consultados, e muito menos de como estes dados são armazenados. Usuários do processador de OQL devem especificar objetos que forneçam os dados a ele. Estes objetos devem implementar certas interfaces, para que o processador de OQL possa usá-los. Fora isso, os objetos podem obter os dados de qualquer forma. Porém, especificamente neste trabalho, o mais comum é que os dados sejam obtidos através de uma conexão JDBC, para depois serem repassados ao processador OQL, que os utiliza na montagem do resultado da consulta.

O processador de OQL implementa todas as funcionalidades da OQL de maneira independente da origem dos dados. Isto inclui filtragem, ordenação, e consultas hierárquicas. Mesmo que o banco não tenha estes recursos, eles são suportados pelo software, pois o processador de OQL os implementa. É claro que a performance não é comparável à de um banco que possui estes recursos, pois tal banco realiza otimizações com base na disposição dos dados em meio físico. Por exemplo, se for feita uma filtragem, o processador de OQL vai percorrer todos os objetos, enquanto um banco com este recurso vai usar um índice para percorrer um intervalo mínimo de objetos.

Quando o processador de OQL é utilizado com um banco que possui determinado recurso, a execução da consulta é desviada em todo ou em parte para o banco, de modo a aproveitar a performance do banco. Porém o processador de OQL é independente deste processo, que é inteiramente gerenciado e realizado pelo driver do banco. Por esta razão, neste tópico será apresentado o processador de OQL de uma forma mais genérica, independente da utilização de um SGBDR, mas muito importante para a emulação de um SGBDOO.

5.3.1 Noção geral da implementação

A grande maioria do código do processador de OQL é formado pelas classes que representam construções de linguagem. Existe uma classe base abstrata, chamada **`Expression`**, que define métodos virtuais (alguns abstratos) que devem ser implementados corretamente pelas classes derivadas (ou seja, conforme a especificação do método em **`Expression`**). A figura 12 mostra um diagrama com as principais classes.

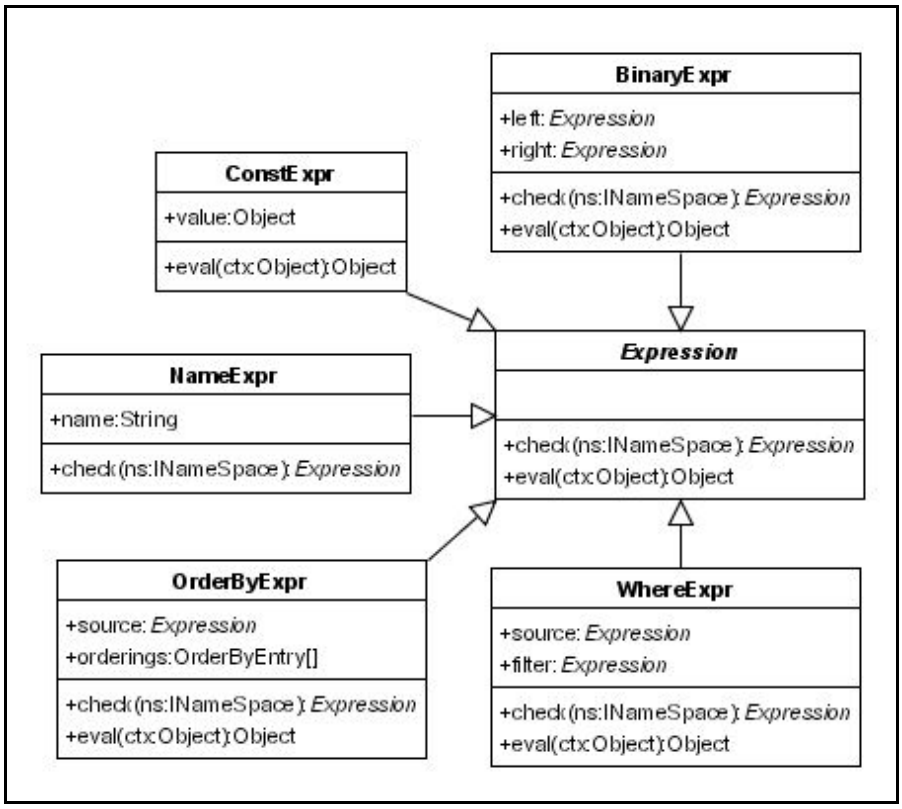

Figura 12 – Diagrama com as principais classes do Processador de OQL.

Cada uma destas classes é capaz de realizar a análise semântica e a execução a parte da expressão que lhe diz respeito. Por exemplo, a classe `BinaryExpr` é responsável pela análise semântica e execução das expressões binárias (dois operandos separados por um operador). A análise semântica é feita no método `check()`, que é virtual. No caso de `BinaryExpr`, o método `check()` verifica se é permitido aplicar o operador aos dois operandos. A execução é feita pelo método `eval()`, também virtual, que retorna o resultado da expressão. No caso de `BinaryExpr`, o método `eval()` chama recursivamente o `eval()` de `left` e `right`, aplica aos resultados o operador, e retorna o resultado.

Como se vê, a execução das consultas se dá diretamente por estas classes, sem que seja gerado código em qualquer conjunto de instruções para depois ser executado. Ou seja, o processador OQL possui uma implementação semelhante à das linguagens de *script*. Esta modelagem foi escolhida pelos seguintes motivos:

a) simplicidade da implementação, com os inerentes ganhos na produtividade do desenvolvimento;
b) grande quantidade de recursos para inspecionar e transformar expressões, o que facilita a implementação do gerador do plano de execução, que será visto mais adiante;

c) maior performance na compilação.

Esta abordagem é bastante usada em implementações de bancos de dados SQL, pois em geral o banco desconhece as consultas que serão executadas até receber uma solicitação para executar uma consulta imediatamente. Apesar disso, deve ficar claro que nas situações onde o banco conhece as consultas antes de receber ordem de executá-las, é mais eficiente compilar as consultas (transformá-las em código executável). Mas isto fica fora do escopo deste trabalho.

Caso seja criada uma nova construção de linguagem, basta criar uma nova classe que deriva de `Expression`, e implementar ali a análise semântica e a própria execução do elemento da linguagem, além de modificar o *parser*. Feito isto, a construção já pode ser usada em comandos OQL. Não é necessário alterar a implementação das construções de linguagem que já existem, nem modificar compiladores e muito menos criar novas instruções.

5.3.2 Parsing

O *parser* recebe um comando OQL em uma `String` do Java. Ele faz as análises léxica e semântica em um passo apenas, e gera uma árvore de objetos em memória, que representa o comando OQL. Os objetos desta árvore pertencem às classes que representam expressões, apresentadas acima.

O *parser* é do tipo LL1, e foi gerado pela ferramenta JavaCC, que gera tanto o analisador léxico como semântico, a partir de uma definição de linguagem semelhante a um arquivo BNF, porém específica para o JavaCC. O *parser* não possui ambigüidades.

A precedência das operações e os parênteses são tratados pelo *parser*, de maneira que não são necessárias consistências nem transformações adicionais após a fase de *parsing*.

Uma considerável limitação do *parser* é que ele não possui qualquer mecanismo de recuperação em caso de erro no texto de origem. Em outras palavras, ao encontrar um erro, o *parsing* é interrompido, e apenas aquele erro é mostrado ao usuário. Erros que eventualmente existam na frente não são mostrados. Isto não chega a ser tão prejudicial, visto que os comandos OQL costumam ser bem pequenos.

5.3.3 Análise Semântica

A análise semântica costuma ser feita logo após o *parsing*. Quem faz a análise semântica são as classes que representam expressões, conforme foi explicado acima.

O ponto de entrada da análise semântica é o método virtual `check()`, da classe `Expression`. Cada derivação de `Expression` deve sobrecarregar este método e efetuar a análise semântica da construção de linguagem que a classe implementa. Este método deve retornar um objeto da classe `Expression` que esteja consistente (isto é, que não necessite uma chamada a `check()`). Na maioria das classes, a implementação do `check()` faz as verificações e retorna o próprio objeto que recebeu a chamada (o objeto `this`). Estas classes costumam modificar o estado de seus objetos, adicionando informações sobre o tipo da expressão, ou informações úteis para execução do elemento de linguagem. Mas há algumas classes cuja análise semântica gera informações num formato bastante diferente do objeto que recebeu a chamada a `check()`, que elas precisam instanciar um objeto `Expression` totalmente diferente, e retorná-lo ao invés de retornar `this`. Seja como for, a única exigência do método `check()` é que o objeto retornado, seja o próprio objeto que recebeu a chamada, seja um outro totalmente diferente, não deve necessitar de uma nova chamada ao método `check()` para poder executar.

Caso o método `check()` encontre um erro de semântica, deve lançar a exceção `InvalidExpression` com a mensagem correspondente ao erro. Esta expressão não deve ser capturada durante a análise semântica. Em outras palavras, quando um erro de semântica é encontrado, a análise é abortada, e o tratamento do erro deve ser feito fora do código que realiza análise semântica. Assim como o *parser*, a análise semântica também está limitada a identificar no máximo um erro por comando.

Caso uma expressão `A` contenha operandos (que também são instâncias de `Expression`), o método `check()` de `A` primeiro chama recursivamente o `check()` de cada operando, para depois fazer a análise semântica propriamente dita. É naturalmente óbvio que os operandos devem ser analisados antes da expressão mais externa, visto que antes da análise, os operandos não possuem informações sobre tipo de dado, e nem mesmo se sabe se eles estão corretos semanticamente. Não há problema na chamada recursiva ao `check()` dos operandos, pois o grafo do comando OQL é uma árvore e portanto não possui ciclos.

Uma vez que um objeto `Expression` foi analisado, é possível chamar o método `getType()` dele. Este método retorna um objeto que representa o tipo de dado da expressão. Este objeto diz se a expressão é numérica, lógica, textual, ou se é uma data ou uma coleção de objetos. O método `getType()` é chamado diversas vezes para objetos diferentes, durante a análise semântica. Seja por exemplo, a expressão binária que testa a igualdade de dois operandos. Para estar semanticamente correta, os tipos dos operandos devem ser aplicáveis ao operador de igualdade ao mesmo tempo. Para conhecer o tipo dos operandos, o método `check`

`()` chama o método `getType()` para cada operando.

5.3.3.1 Resolução de nomes

O comando OQL pode conter nomes (também chamados de identificadores), que representam um elemento pré-definido externo à linguagem OQL, tal como uma variável, uma constante, uma função, ou um atributo ou método de uma classe. Para cada nome que aparece no comando, o analisador semântico deve determinar se ele existe e se pode ser usado no lugar onde apareceu.

Quando o *parser* encontra um nome, ele coloca um objeto da classe `NameExpr` na árvore do comando. O objeto `NameExpr` simplesmente guarda o nome encontrado (uma `String`). A classe `NameExpr` não tem meios para determinar se existe um elemento com aquele nome, muito menos para determinar o tipo do elemento, se ele existir. Por isso ela delega esta tarefa a um objeto que implemente a interface `INameSpace`.

A interface `INameSpace` define um método chamado `getCheckedExpr()`. Este método recebe uma `String` contendo o nome do elemento e retorna um objeto `Expression` que representa o elemento. Tudo que o método `check()` da classe `NameExpr` faz é retornar o valor resultante da chamada ao `getCheckedExpr()`. Neste processo, o objeto `NameExpr` se torna inútil e é perdido. Ou seja, antes da análise semântica, um comando OQL pode conter vários objetos `NameExpr` na árvore. Após a análise, a árvore fica sem objetos `NameExpr`.

A implementação do método `getCheckedExpr()` deve retornar um objeto `Expression` que esteja consistido e em condições de executar. Em outras palavras, o objeto `Expression` retornado deve permitir a chamada dos métodos `getType()` e `eval()`, sendo este último visto mais abaixo.

O `INameSpace` usado para resolver nomes vem como um argumento do método `check()`. Toda chamada ao `check()` de `Expression` recebe um `INameSpace` como argumento, e caso hajam operandos, o `NameSpace` recebido é repassado aos operandos, de forma recursivamente. A chamada ao `check()` da expressão mais externa, feita por código usuário do processador de OQL, deve especificar este `INameSpace`. Ou seja, o processador de OQL não possui qualquer implementação de `INameSpace`. No caso do emulador, a implementação do `INameSpace` fica no driver do banco, e ela fornece acesso às coleções e classes de objetos persistentes, de modo que possam ser usadas em comandos OQL.

5.3.4 Execução do comando

Uma vez feita a análise semântica, o comando pode ser executado. Executar um comando OQL significa calcular o valor resultante do comando. O comando pode conter várias partes que precisam ser calculadas conforme as construções de linguagem usadas. Por isso cada classe que representa uma construção de linguagem (e portanto, deriva de `Expression`) é também responsável pela execução do cálculo.

O ponto de entrada para a execução do comando é o método virtual `eval()`, da classe `Expression`. Cada classe deve sobrecarregar este método e efetuar a execução conforme a construção de linguagem representada pela classe. O método `eval()` retorna o valor da expressão.

Caso a expressão contenha operandos, estes devem obviamente ser calculados primeiro. Neste caso a expressão mais externa primeiro chama o `eval()` dos operandos, para depois realizar o cálculo sobre os valores retornados. Neste ponto algumas otimizações são feitas, no sentido de economizar cálculos desnecessários. Por exemplo, seja a expressão `case A then B else C end`. Esta expressão possui os operandos `A`, `B` e `C`. O único operando que precisa ser calculado sempre é o operando `A`. Dependendo de seu valor, será calculado o operando `B` ou `C`, mas nunca ambos.

Fora isto não há otimizações adicionais no processador de OQL. Expressões constantes não são convertidas em literais antes da execução. Por exemplo, o *parser* converte a expressão `1+2` em uma árvore contendo 3 objetos, mostrada na figura 13.

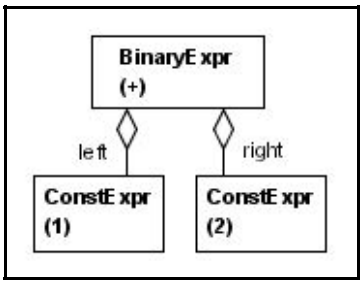

Figura 13 – Exemplo de árvore gerada pelo *parser*.

O analisador semântico apenas consiste a expressão e mantém esta árvore intacta. E toda vez que for chamado `eval()` na expressão principal, será realizada a soma. O processador de OQL nunca tenta deduzir que o resultado desta expressão sempre será `3`. A

principal razão disto é favorecer a performance de compilação da maioria dos comandos, que não possuem necessidade deste tipo de otimização. Ainda assim, é bastante simples implementar esta otimização, caso venha a se tornar necessária.

5.3.4.1 Acesso a dados externos

Eventualmente, durante a execução de um comando, o processador de OQL precisa acessar dados externos que ele não sabe onde muito menos como estão armazenados. Exemplos destes dados seriam variáveis, constantes (não literais), coleções e classes de objetos persistentes, estes últimos muito relevantes para o emulador.

Estes dados são obtidos através de um contexto, que é passado como argumento do método `eval()`. O processador de OQL abstrai completamente o formato deste contexto, que para ele não passa de um `Object` do Java (a classe a qual todos os objetos pertencem). Cada derivação de `Expression` que implemente o `eval()` e necessite de um contexto, deve assumir que o contexto é representado ou no mínimo é acessível pelo objeto passado como argumento ao `eval()`.

As classes do processador de OQL, tais como `BinaryExpr`, `UnaryExpr`, e `WhereExpr`, não utilizam diretamente o argumento recebido em `eval()`, limitando-se a simplesmente repassar este argumento ao `eval()` dos operandos. Para estas classes, o tipo do objeto que representa o contexto pode ser qualquer um.

Quem realmente precisa do contexto são os objetos `Expression` gerados durante o processo de resolução de nomes. Assim, a implementação do `INameSpace` é quem dita o formato do contexto, e o código que escolhe um `INameSpace` para ser passado ao método `check()` da expressão mais externa, deve estar ligado ao código que escolhe um contexto para ser passado ao método `eval()` desta mesma expressão.

5.3.5 Resumo das classes

A tabela 3 mostra as principais classes e quais funcionalidades da linguagem elas implementam.

Classe	Funcionalidade
`Parser`	Classe que faz o *parsing*, gerada pelo JavaCC.
`Expression`	Classe abstrata da qual devem derivar todas as classes que representam expressões OQL.
`ConstExpr`	Classe gerada pelo *parser* quando ele encontra um literal.
`NameExpr`	Classe gerada pelo *parser* quando o nome de uma variável ou membro é encontrado.
`NameSpace`	Interface que deve ser implementada por usuários do processador de OQL para a resolução de nomes.
`BinaryExpr`	Classe que implementa todas as expressões binárias.
`UnaryExpr`	Classe que implementa todas as expressões unárias.
`SelectExpr`	Classe que implementa o mapeamento.
`WhereExpr`	Classe que implementa a filtragem.
`OrderByExpr`	Classe que implementa a ordenação.
`GroupByExpr`	Classe que implementa o agrupamento.

Tabela 3 – Principais classes do processador de OQL.

5.4 CONTEXTO DE OBJETOS

A implementação do contexto de objetos fica numa classe que implementa a interface `IObjectContext`, e faz parte do driver do SGBDR. Em geral, cada contexto possui uma conexão JDBC com o SGBDR em questão. Esta conexão é usada para realizar consultas e alterações nos objetos persistidos. Detalhes sobre estes processos serão mostrados mais adiante.

5.4.1 Criação do primeiro contexto

O primeiro contexto é criado através de uma chamada ao seu método *constructor*. Esta é a única ocasião em que é necessário conhecer o driver do SGBDR. Mas isto pode ser eliminado com a utilização de *factoring*, na qual a classe do driver é descoberto em tempo de execução (geralmente lida de um arquivo de configuração), e o contexto instanciado por meios de reflexão do Java.

O *constructor* do contexto recebe dois argumentos: a localização do arquivo de mapeamento, e uma `String` contendo os parâmetros de conexão com o SGBDR. Esta `String` também costuma ser lida de um arquivo de configuração.

Ao ser instanciado desta forma, a primeira coisa que o contexto faz é estabelecer uma conexão JDBC com o SGBDR. Esta conexão é mantida enquanto o contexto não for fechado explicitamente. Ela é usada por diversas operações envolvendo o contexto.

5.4.2 Cache de objetos Java

Como foi visto na especificação, os objetos persistentes são representados no ambiente Java como simples objetos Java. Estes objetos são instanciados a medida que são consultados ou criados, de maneira que pode haver uma enorme quantidade de objetos persistidos, maior até do que a suportada pelo ambiente Java.

Porém, para dar a impressão de que os objetos manipulados no ambiente Java são de fato os objetos persistidos, é necessário garantir que todas as consultas feitas para o mesmo objeto persistido no mesmo contexto retornem o mesmo objeto Java. Ou seja, a assertiva mostrada no quadro 15 jamais poderá falhar.

```
void testaObjetosJava(IObjectContext db) {

  Funcionario f1 = db.getObject(
    Funcionario.class, "nome='Smith'");

  algumProcesso();

  Funcionario f2 = db.getObject(
    Funcionario.class, "nome='Smith'");

  assert f1 == f2; // f1 e f2 devem referenciar o mesmo objeto.

}
```

Quadro 15 – Demonstração da invariante na qual todas as consultas feitas a um objeto no mesmo contexto retornam o mesmo objeto Java.

É claro que isto não vale para o caso de `algumProcesso()` destruir o objeto ou mudar o nome do funcionário. Mas para todos os outros casos, a assertiva deve funcionar.

Isto é implementado através de um cache dos objetos consultados. A chave do cache é o OID do objeto. Além de garantir a especificação, o cache traz benefícios para performance, pois todo objeto lido por OID é primeiramente procurado no cache (e a leitura por OID é muito comum em operações de navegação).

O cache suporta situações como a encontrada no programa do quadro 16.

```
void leOsItens(IObjectContext db) {

  Iterator i = db.query("ItemsPedido").iterator();
  while (i.hasNext()) {
    ItemPedido item = (ItemPedido) i.next();
    System.out.println(item.getValor());
  }
}
```

Quadro 16 – Exemplo de programa que lê uma enorme quantidade de objetos.

Este programa lê todos os itens de pedido persistidos. A quantidade de itens pode ser enorme, muito maior do que cabe no cache. Porém, quando o cache fica cheio, os itens do cache que não estão sendo referenciados por variáveis no ambiente Java são removidos do cache. Isto é feito pelo uso da classe `SoftReference`, que é embutida no ambiente Java. Esta classe possui meios para detectar que um objeto Java não é mais utilizado por parte alguma do ambiente. Quando o contexto detecta isto, ele remove aquele objeto do cache.

5.4.3 Criação de objetos

Conforme a especificação, o método `newInstance(Class)`, de `IObjectContext`, é responsável por instanciar objetos Java. Este método recebe como parâmetro o objeto `Class` de uma interface Java que é independente de qualquer SGBDR. A implementação do método procura a classe que implementa a interface recebida, e através de reflexão, instancia o objeto Java.

O objeto instanciado é imediatamente registrado junto ao contexto. Isto é necessário porque a instância vai precisar de funcionalidades do contexto, como o cache de objetos e a conexão JDBC com o SGBDR. Cada objeto Java que representa um objeto persistido sempre fica registrado a um (e no máximo um) contexto.

Feitos estes passos, o objeto Java instanciado é retornado ao chamador de `newInstance(Class)`, e está pronto para uso.

5.5 CLASSES DE OBJETOS PERSISTENTES

Os objetos persistentes precisam de duas classes Java para funcionar: uma interface que é independente do SGBDR usado (em geral, o ambiente Java considera que uma interface é uma classe), e uma classe específica para o SGBDR usado, que implementa a interface.

A interface independente de SGBDR define os seguintes métodos:

a) `save()`;

b) `destroy()`;
c) métodos que operam em coleções globais;
d) métodos que acessam os atributos e relacionamentos da classe (*getters* e *setters*).

A classe dependente do SGBDR implementa estes métodos, e adiciona o seguinte:
a) variáveis para guardar o valor dos atributos e dos relacionamentos;
b) uma variável para guardar o OID (do tipo `long`);
c) uma variável para guardar o contexto (do tipo referência para `IObjectContext`);
d) variáveis de controle, que guardam informações a respeito do estado dos atributos.

5.5.1 Estados dos atributos

Para beneficiar a performance, o valor dos atributos e relacionamentos não são carregados imediatamente quando o objeto é lido, pois não há garantia de que eles serão usados. Da mesma forma, quando o objeto é salvo, somente os valores alterados são colocados no comando `UPDATE` que vai ao SGBDR. Isto implica que um atributo qualquer de um objeto persistente tem os seguintes estados:

a) não-lido, na qual o valor do atributo é desconhecido e depende de um acesso ao SGBDR;
b) lido, na qual o valor do atributo é conhecido e não foi alterado;
c) sujo, na qual o valor foi alterado pelo programador e fará parte do comando `UPDATE` quando o objeto for salvo.

Estes estados não são usados antes de o objeto ser persistido pela primeira vez, pois esta operação implica num comando `INSERT` envolvendo todos os atributos. Após a inserção, todos os atributos possuem o estado "lido".

Quando é feita uma consulta, a maioria dos atributos ficam no estado "não-lido", sendo que alguns ficam no estado "lido", pois seu valor pode ser descoberto durante a consulta. A mudança de "não-lido" para "lido" ocorre quando o *getter* do atributo é chamado. Nesta ocasião, o valor do atributo é lido do SGBDR, e isto pode implicar num impacto à performance. Trata-se de uma limitação bem conhecida do software.

A mudança para "sujo" ocorre quando o *setter* do atributo é chamado. Nesta ocasião, o valor do atributo e o estado anteriores do atributo são ignorados. A mudança de "sujo" para "lido" ocorre quando o objeto é salvo.

Quando o `save()` é chamado num objeto que não possui atributos sujos, não ocorre qualquer acesso ao SGBDR, e a chamada é simplesmente ignorada.

5.5.2 Primeira chamada ao `save()`

Quando um objeto é instanciado através do método `newInstance(Class)`, de `IObjectContext`, o valor da variável que mantém seu OID é zero. Quando o método `save()` é chamado para um objeto cujo OID é zero, trata-se da primeira chamada ao `save()`.

A primeira coisa que ela faz é solicitar ao contexto um OID (a operação de computar o OID será descrita mais adiante). Depois é aberta uma transação local, e é inserido um registro em cada tabela definida no mapeamento para guardar uma parte do estado do objeto. Cada registro recebe o OID do objeto e o valor dos atributos correspondentes. Também são setados os campos das coleções globais. Se tudo transcorreu sem problemas, a transação é confirmada e o método retorna. Nestas circunstâncias, o estado de todos os atributos será "lido", e o valor da variável que guarda o OID será o valor do OID.

Se houve algum problema, são tomadas as medidas para que o método `save()` não apresente efeitos colaterais. A transação é abortada, e o valor da variável OID permanecerá zero. A exceção que informa a causa o problema será propagada para o chamador. A maioria dos problemas que ocorrem no método `save()` são causados por tentativas de violar invariantes (como tentar inserir mais de um objeto com a mesma chave, ou deixar nulo um atributo que é obrigatório). Assim sendo, o chamador poderá corrigir o problema na mesma instância, e depois invocar o método `save()` novamente.

5.5.3 Chamadas ao `save()` para um objeto já persistido

Quando o método `save()` é chamado para um objeto cujo OID é diferente de zero, trata-se de uma chamada para um objeto já persistido.

Inicialmente o método monta comandos `UPDATE`, com base nos atributos que estão no estado "sujo". É montado um comando `UPDATE` para cada tabela definida no mapeamento para guardar uma parte do estado do objeto. Estes comandos `UPDATE` sempre contém a cláusula `where OID=`*`valor-do-OID`*, e sem qualquer filtro adicional. Depois é aberta uma transação e cada comando `UPDATE` é emitido ao SGBDR. Se tudo transcorreu sem problemas, a transação é confirmada e o método retorna. Nestas circunstâncias, todos os atributos que tinham o estado "sujo" passarão a ter o estado "lido".

Se houve algum problema, são tomadas as medidas para que o método **`save()`** não apresente efeitos colaterais. A transação é abortada. A exceção que informa a causa o problema será propagada para o chamador. A maioria dos problemas que ocorrem no método **`save()`** são causados por tentativas de violar invariantes (como tentar inserir mais de um objeto com a mesma chave, ou deixar nulo um atributo que é obrigatório). Assim sendo, o chamador poderá corrigir o problema na mesma instância, e depois invocar o método **`save()`** novamente.

5.5.4 O método `destroy()`

O método **`destroy()`** faz com que um objeto seja destruído na base persistente. A primeira coisa que ele faz é montar um comando **`DELETE`** para cada tabela definida no mapeamento para guardar uma parte do estado do objeto. Estes comandos **`DELETE`** sempre contém a cláusula **`where OID=valor-do-OID`**, e sem qualquer filtro adicional. Depois são montados comandos **`UPDATE`** para limpar referências fracas e suaves. Finalmente é aberta uma transação e os comandos são submetidos ao SGBDR. Se tudo transcorreu sem problemas, a transação é confirmada, atribui-se zero para a variável que guarda o OID, e o método retorna.

Se houve algum problema, em geral é porque existem referências fortes para o objeto sendo destruído. A transação é abortada, e o método retorna sem qualquer mudança em suas variáveis.

5.5.5 Métodos que operam em coleções globais

Cada coleção global onde pode estar um objeto implica na existência de dois métodos na interface e portanto, na implementação:

a) o método **`addTo`*`Coleção`*`()`**, que adiciona o objeto na coleção correspondente;

b) o método **`removeFrom`*`Coleção`*`()`**, que remove o objeto de tal coleção.

Como a coleção global mapeia para um simples campo lógico no SGBDR, o método **`addTo`*`Coleção`*`()`** apenas realiza um **`UPDATE`** setando aquele campo para **`true`**. O método **`removeFrom`*`Coleção`*`()`** faz a mesma coisa, mas seta o campo para **`false`**. Chamadas a estes métodos sempre implicam no salvamento do objeto.

5.6 IMPLEMENTAÇÃO DA IQUERY

Todas as consultas são feitas por intermédio de objetos **`IQuery`**. A classe concreta de tais objetos é definida pela implementação do contexto. A implementação da **`IQuery`** contém

o seguinte:

a) uma variável que referencia o contexto;
b) uma variável `String` que guarda o comando OQL;
c) variáveis que guardam os parâmetros do comando;
d) variáveis de controle;
e) variáveis para cache;
f) o método `setStmt(String)`;
g) os métodos `setParam(int, Object)` e `setParam(String, Object)`;
h) o método `run()`.

Quando um objeto `IQuery` é instanciado, apenas o contexto dele é iniciado. O método `setStmt(String)` deve ser chamado para especificar o comando OQL a ser executado. O comando passado pode conter parâmetros precedidos por `:`. Por exemplo, o comando `Funcionarios where codigo = :codFun` está preparado para receber o parâmetro `codFun`.

O método `setStmt(String)` faz imediatamente o *parsing* do comando recebido. Se houver algum problema, é lançada uma exceção e o objeto `IQuery` fica sem mudanças em seu estado interno.

Uma vez setado o comando, se houver algum parâmetro, deve-se chamar o método `setParam(int, Object)` ou `setParam(String, Object)`. A primeira versão recebe o índice do parâmetro, que começa de zero. A segunda recebe o nome do parâmetro, que deve corresponder ao nome usado no comando. O segundo argumento destes métodos é o valor do parâmetro. Um exemplo pode ser visto no quadro 17.

```
...
IQuery qry = db.newQuery();
qry.setStmt("Funcionarios where codigo = :codFun");
qry.setParam("codFun", new Integer(codigo));
...
```

Quadro 17 – Exemplo de chamada ao método `setParam`.

Depois de se atribuir um valor aos parâmetros, o comando pode ser executado através do método `run()`. Na primeira execução, o método `run()` faz análise semântica do comando e monta o plano de execução do mesmo. A execução do comando se dá por uma simples chamada ao `eval()` do objeto `Expression` que representa o comando. Mais detalhes sobre o que ocorre nesta chamada ao `eval()` serão fornecidos no tópico seguinte.

5.7 MONTAGEM DO PLANO DE EXECUÇÃO DAS CONSULTAS

A maioria dos comandos OQL são implementados no processador de OQL por algoritmos que usam força bruta. O processador de OQL é uma peça independente, e por isso desconhece da presença de índices e da possibilidade de se usar recursos do SGBDR para otimizar comandos. Assim, o comando OQL `Funcionarios where codigo = :codFun`, se fosse executado diretamente pelo processador de OQL, faria uma pesquisa seqüencial sobre uma seleção que traz todos os funcionários do SGBDR.

Como este comportamento é inadmissível para a maioria das aplicações, existe um processo entre a análise semântica de um comando e a execução do mesmo, que se chama montagem do plano de execução da consulta. Este processo transforma o objeto `Expression` que representa o comando OQL em outro objeto `Expression`, que embora tenha o mesmo efeito do original, executa de forma muito mais rápida.

Esta transformação é feita com base em uma análise que ocorre na árvore de objetos do comando. Por exemplo, o comando `select nome from Funcionarios where datNas < :datNas order by nome` gera a árvore de objetos mostrada na figura 14.

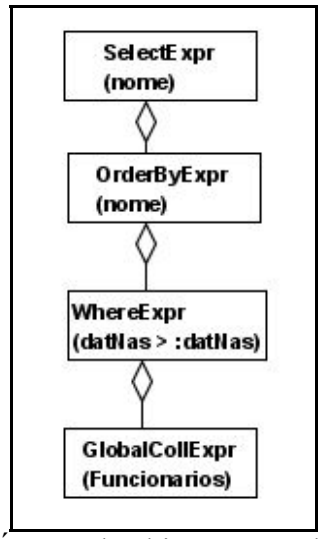

Figura 14 – Árvore de objetos antes da otimização.

Esta árvore é analisada das folhas em direção à raiz. No primeiro passo, ela é transformada na árvore mostrada na figura 15.

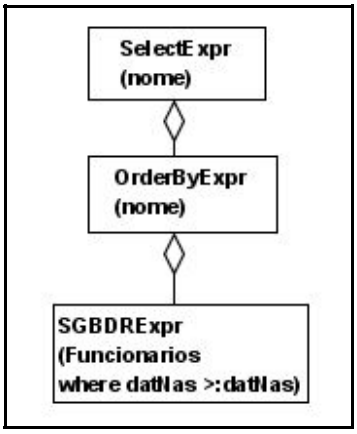

Figura 15 – Árvore de objetos após 1 passo de otimização.

Após o segundo passo ela ficará conforme a figura 16.

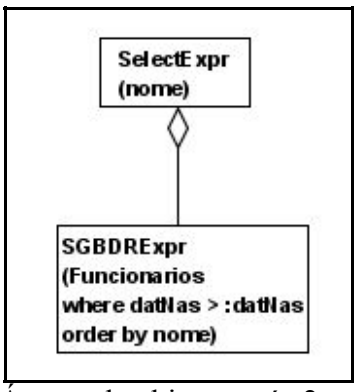

Figura 16 – Árvore de objetos após 2 passos de otimização.

E após o terceiro passo, o resultado será o da figura 17.

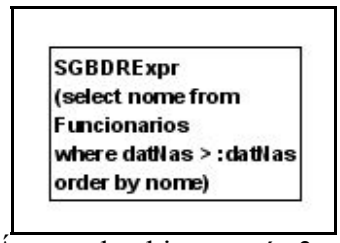

Figura 17 – Árvore de objetos após 3 passos de otimização.

Observa-se que agora todo comando OQL foi condensado em um único nó. Quando o `eval()` for chamado, o comando que será enviado ao SGBDR será este:

```
select nome from TabFuncionario
where emFuncionarios=1 and datNas < :datNas order by nome
```

Observa-se que sem a transformação, o comando que seria enviado ao SGBDR para a árvore original seria este:

```
select OID from TabFuncionario where emFuncionarios=1
```

Este comando traria o OID de todos os objetos que estão na coleção `Funcionarios`. A filtragem e a ordenação seriam feitas em memória, pelo processador de OQL, ocasionando em grande perda de performance.

A montagem de plano de execução também resolve a navegação entre objetos. Por exemplo, seja o seguinte comando:

```
select nome, proprietario.nome from Veiculos
```

O comando a ser enviado ao SGBDR será o seguinte:

```
select
  a.nome, b.nome
from
  TabVeiculo a, TabPessoa b
where
  a.emVeiculos=1 and a.OIDproprietario = b.OID
```

Podem ocorrer casos em que o SGBDR não tem meios para produzir o mesmo efeito do comando OQL. Exemplo:

```
first(Funcionarios order by datNas)
```

Como em geral o SGBDR não tem como realizar a operação `first`, a árvore resultante da transformação será a mostrada na figura 18.

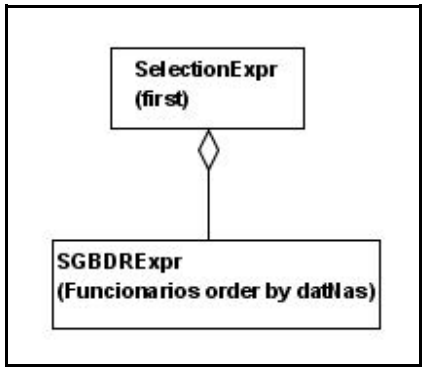

Figura 18 – Otimização máxima obtida com a operação `first`.

Esta árvore provoca o envio do seguinte comando ao SGBDR:

```
select OID from TabFuncionario where emFuncionarios=1 order by datNas
```

Observa-se que o SGBDR está realizando uma ordenação que seria desnecessária se ele "soubesse" que o usuário deseja somente o primeiro elemento, pois neste caso bastaria uma simples pesquisa. Infelizmente isto é uma limitação do SGBDR, e não há como eliminar este tipo de problema. Ressalta-se que pode haver uma implementação diferente para cada SGBDR, de maneira que recursos específicos de um SGBDR podem ser usados. Sabe-se, por exemplo, que o Oracle suporta o seguinte comando:

```
select /*+FIRST_ROWS*/ OID from TabFuncionario
where emFuncionarios=1 order by datNas
```

Este comando instrui o Oracle a trazer as primeiras linhas o mais rapidamente possível, o que implica que ele fará várias pesquisas ao invés de uma ordenação. Já o SQL Server da Microsoft suporta o seguinte comando:

```
select OID from TabFuncionario where emFuncionarios=1 and
datNas <=
  any (select datNas from TabFuncionario where emFuncionarios=1)
```

Este comando instrui o SQL Server a trazer somente os funcionários cuja data de nascimento é menor ou igual à data de nascimento de todos os demais funcionários. O Oracle também suporta este comando, mas a abordagem do `/*+FIRST_ROWS*/` apresenta performance superior.

Conclui-se que o uso de implementações específicas para cada banco implica que serão usados todos os recursos que o banco oferece, não se limitando ao SQL básico.

5.8 TRANSAÇÕES

Transações se iniciam por uma chamada ao `newTransaction()` de `IObjectContext`, que retorna um objeto `ITransaction`. A classe deste objeto é determinada pela implementação do contexto. Como `ITransaction` é um contexto, em geral a classe que implementa `ITransacation` deriva da classe que implementa `IObjectContext`, ou ambas derivam de uma base comum.

O contexto que recebe uma chamada ao `newTransaction()` se limita a instanciar o objeto da transação e retorná-lo ao chamador. O objeto instanciado recebe como parâmetro o contexto que o instanciou (chamado de contexto-pai daqui para frente).

A transação retornada é também um contexto de objetos. Ou seja, ela também possui cache de objetos, capacidade de instanciar objetos, realizar consultas, etc.

Quando a primeira operação é feita na transação, ela solicita ao contexto-pai sua conexão JDBC com o SGBDR. O contexto-pai então transfere sua conexão para a transação, e fica a partir daí impedido de realizar operações para o usuário. A transação então chama métodos da conexão que iniciam a transação. E só então a primeira operação é realizada.

As demais operações são realizadas normalmente, como se a transação fosse um contexto normal. Quando é chamado o método `commit()` ou `rollback()` na transação, ele é simplesmente propagado para a conexão, que é então devolvida ao contexto-pai. Este por sua vez fica livre para realizar operações até que se inicie outro processo na transação.

Os métodos `commit()` e `rollback()` também limpam de forma forçada o cache de objetos da transação, de modo que se for iniciada uma nova transação, será necessário ler os objetos novamente.

Como a transação é um contexto, é possível chamar `newTransaction()` na própria transação, e assim é feito o suporte a transações aninhadas.

5.9 GERAÇÃO DE ARQUIVOS FONTE

5.9.1 Arquivo de mapeamento O-R

O utilitário que gera arquivo de mapeamento O-R padrão usa um algoritmo do tipo "força-bruta". O arquivo de mapeamento padrão usa a técnica "uma tabela por classe". Tudo que o algoritmo faz é iterar por todas as classes do modelo lógico, gerando uma tabela por classe.

Também é feito um mapeamento de nomes. Por exemplo, pode ser que o analista tenha criado um atributo com o nome OID. Este atributo implica na geração de um campo que iria entrar em conflito com o campo OID existente em cada tabela. O algoritmo detecta isto e gera um nome diferente para o campo que guarda o valor deste atributo.

Embora o arquivo de mapeamento esteja no formato XML, o algoritmo usa a classe `PrintStream` do Java, que grava apenas texto. O algoritmo coloca tags XML de forma explícita no texto gerado.

5.9.2 Fontes Java para objetos persistentes

Para cada classe do modelo lógico existem dois fontes: um para a interface, que é independente do SGBDR, e outro para a classe que implementa a interface, dependente do SGDBR. Deste modo, o algoritmo que gera os fontes possui uma parte que é independente do

SGBDR, e outra que é dependente.

A geração destes fontes é uma tarefa relativamente complexa, devido à quantidade de situações que podem ser encontradas neles. Por isso, optou-se por usar a biblioteca Velocity. Ela permite que se crie um modelo do fonte contendo diretivas que depois são substituídas por texto que corresponde a um trecho específico do fonte.

O algoritmo de geração das interfaces apenas itera por cada classe do modelo gerando o fonte de cada uma. Já a geração dos fontes que implementam as interfaces é mais complexa, porque a implementação depende do mapeamento. Assim, o algoritmo primeiro carrega o mapeamento, e depois itera por cada classe gerando o fonte de cada uma. A parte mais complexa é a geração dos métodos `save()` e `destroy()`, que envolve todas as tabelas onde o objeto é persistido, e no caso do `destroy()`, também todas as referências fracas e suaves para o objeto.

5.9.3 Script DDL para criação de entidades no SGBDR

O algoritmo que gera o script DDL também é do tipo "força-bruta". Inicialmente ele carrega para memória o modelo lógico e o mapeamento. Depois ele itera por todas as tabelas especificadas no mapeamento, emitindo um comando `CREATE TABLE` para cada uma. No final, são emitidos vários comandos `ALTER TABLE` *`tabela`* `ADD FOREIGN KEY ...`, que garantem a integridade referencial da base.

5.10 IMPLEMENTAÇÃO DE REFERÊNCIA

A implementação de referência, usada para validar o software nos âmbitos conceitual e prático, suporta o SGBDR MySQL versão 4.0. O MySQL foi escolhido pelos seguintes motivos:

a) é um SGBDR relativamente pequeno, que possui poucos recursos – se houver sucesso com MySQL, é quase certo que haverá sucesso também com SGBDRs mais robustos;
b) embora seja pequeno, suporta integridade referencial e transações aninhadas, recursos considerados fundamentais para que um SGBDR seja suportado por este software;
c) a instalação do MySQL é muito rápida e pouco intrusiva – após instalado, o MySQL ocupa apenas um diretório e não mexe em qualquer configuração do sistema operacional;

d) a performance do MySQL é comparável à de um SGBDR mais robusto;
e) trata-se de um software livre, dispensando qualquer burocracia que poderia surgir com o uso de software comercial.

Todos os recursos descritos neste texto são suportados pela implementação de referência.

A implementação de referência usa o mínimo de recursos específicos do MySQL, podendo ser usada como base para uma implementação para outro SGBDR. Estima-se que sejam necessárias apenas 16 horas de trabalho para se obter suporte a outro SGBDR a partir da implementação de referência.

6 CONSIDERAÇÕES FINAIS

6.1 CONCLUSÃO

O software desenvolvido atende aos requisitos de SGBDOO apresentados no capítulo 2. Em outras palavras, o software emula um SGBDOO completamente, permitindo que seus principais usuários – os analistas e programadores – utilizem o método orientado a objetos para modelar as entidades persistentes de sistemas de informação. Ressalta-se que isto é alcançado levando-se em consideração o uso da linguagem Java para codificação de processos e o uso transparente de um SGBDR para a persistência dos dados.

O software possui qualidade suficiente para ser usado em ambientes de produção que não sejam de missão crítica. Em outras palavras ele pode ser usado no desenvolvimento de sistemas de informação complementares. O suporte a aplicações de missão crítica não era objetivo deste trabalho, e não foi implementado por questão de tempo. Ainda assim foi incluso neste capítulo ações sugeridas para esta implementação.

A base de código do software ficou organizada, com pacotes bem definidos, sem referências circulares nem implementações espalhadas. Também há poucas variáveis e métodos globais. Ressalta-se que a parte do software que lida com o MySQL tem característica de componente, ou seja, pode ser substituída por outra sem alterar a que já existe nem as dependentes. Assim é possível implementar o suporte a um SGBDR diferente do MySQL ou mesmo a um SGBD não-relacional. Inclusive é possível implementar diretamente uma camada de persistência e recuperação, tornando o software um SGBDOO completo!

6.2 LIMITAÇÕES

Segue o sumário das principais limitações:

a) não tem performance suficiente para aplicações de missão crítica;
b) a instalação e configuração do software são tarefas trabalhosas;
c) não tem suporte para conversão dos dados no caso de ocorrerem mudanças no modelo lógico.

6.3 TÉCNICAS E FERRAMENTAS USADAS

A metodologia orientada a objetos, unida à linguagem UML, mostraram sua excelente expressividade em representar modelos de funcionamento e aspectos da implementação do software. A metodologia de desenvolvimento em espiral se mostrou adequada para sanar

programaticamente problemas de complexidade e funcionalidade ao mesmo tempo que a especificação é mantida atualizada.

A linguagem Java mostrou que é capaz de lidar com situações de grande complexidade e que exigem performance. Os seguintes recursos da linguagem se mostraram cruciais para o sucesso da implementação:

a) liberação automática de memória;
b) reflexão;
c) classes anônimas;
d) a classe `SoftReference`.

A ferramenta de diagramação Poseidon se mostrou relativamente instável, sendo necessárias algumas reinicializações. Ainda assim, cumpriu seu papel de auxiliar na especificação e documentação. Ressalta-se que foi usada a versão 1.6, e no momento em que este trabalho foi concluído, a versão 2.0 já havia sido lançada.

O ambiente de desenvolvimento Eclipse se apresentou bastante confiável. Recursos como *refactoring*, identação de fontes automática, e dedução de código tornaram o desenvolvimento bastante produtivo.

O gerador de *parser* JavaCC se mostrou bastante produtivo. O *parser* gerado tem excelente performance. Observa-se que o fonte gerado pelo JavaCC não é voltado para entendimento humano. Portanto, houve algum trabalho para depurar o *parser*, mas nada que comprometesse o progresso deste projeto.

A biblioteca de geração de textos a partir de modelos Velocity satisfez completamente as necessidades do projeto.

6.4 SUGESTÕES PARA TRABALHOS FUTUROS

As sugestões para trabalhos futuros são baseadas nas limitações do software. Acredita-se que cada uma das limitações implica num trabalho tão complexo que seria necessário outro projeto deste tamanho para resolvê-las.

6.4.1 Performance para suporte a aplicações de missão crítica

O principal aspecto que limita a performance é a leitura dos atributos em um comando separado do comando de consulta dos objetos. A sugestão para resolver isto consiste em tentar detectar, por meio de computação de estatísticas, quais atributos serão lidos nos objetos sendo

consultados. Deste modo, na próxima vez que aquela consulta ocorrer, haverá conhecimento dos atributos que serão lidos, e os mesmos poderão ser facilmente adicionados no comando que vai ao SGDBR.

A questão principal deste processo é como computar estas estatísticas, e isto implica num mecanismo que identifica com precisão a consulta que está sendo realizada. O comando OQL pode ser usado para identificar a consulta, mas ele não é suficiente, pois o mesmo comando pode ser usado em processos diferentes. Assim, deve haver uma maneira de identificar o processo, e isto pode ser feito exigindo-se do programador que ele marque o início e o fim de cada processo.

Embora esta solução pareça intrusiva (pois ela exige participação do programador), existe um benefício inerente: ela possibilita a implementação de otimizações agressivas. Por exemplo, quando o SGBDOO detecta que um processo específico se iniciou, pode preparar, em *background*, todos os comandos que este processo vai executar, de maneira que no momento em que eles são solicitados, as fases de *parsing*, análise semântica e montagem do plano de execução não precisam ser feitas.

Existe uma sugestão adicional para beneficiar a performance, cuja implementação é mais simples: consiste em gerar *stored procedures* no SGBDR para comandos OQL que não tenham suporte direto no SQL. Deste modo, o processador de OQL realizaria somente operações que pudessem ocorrer completamente na memória.

6.4.2 Facilidade na instalação e na configuração

Consegue-se mais facilidade na instalação e na configuração do software com a criação de programas que façam o papel de GUI. Embora pareça simples, existe uma considerável quantidade de programas a serem feitos, incluindo mas não se limitando a um instalador, um mapeador, um gerador de fontes, e um gerador de script DDLs. A idéia é não obrigar o usuário a manipular arquivos de configuração nem scripts.

Estes programas precisam interagir com o SGBDR escolhido, no sentido de testar a conexão com ele, e criar usuários e bases de dados. Uma implementação importante neste sentido seria "esconder" o SGBDR completamente, tanto dos usuários como dos programadores. Quando o software é instalado, ele iria instalar e configurar um SGBDR de forma transparente. Deste modo, este software teria mais a conotação de um SGBDOO completo do que um emulador de SGBDOO.

6.4.3 Conversão automática dos dados em caso de mudanças no modelo lógico

O processo de conversão automática dos dados é ao mesmo tempo importante e complexo. É importante porque a conversão manual costuma ser lenta e muito propensa ao aparecimento de erros. É complexo porque o algoritmo que faz isto precisa ser sofisticado e capaz lidar com uma quantidade enorme de situações.

Há várias abordagens para conversão. A mais simples dela consiste exportar todos os objetos para um arquivo, destruir a base antiga, criar a base nova a partir do modelo novo, e importar todos os objetos de volta. Porém esta abordagem possui as seguintes limitações:

a) ela só funciona se o modelo novo é compatível com o antigo; isto é, se o analista dividiu uma classe em duas, ou se incluiu invariantes, esta conversão simples provavelmente vai falhar;
b) ela é extremamente lenta se comparada à conversão *in-place* (conversão realizada com comandos DDL do SGBDR, como ALTER TABLE).

Portanto, uma solução completa envolve uma GUI onde o analista mapeia o modelo velho no modelo novo (para resolver casos de incompatibilidade entre modelos), e um algoritmo que é capaz de transformar um esquema de entidades do SGBDR em outro.

6.5 RELEVÂNCIA PESSOAL

Este projeto trouxe os seguintes benefícios para seu autor:

a) bastante conhecimento antes inexistente a respeito dos SGBDOOs e SGBDRs.
b) experiência nas técnicas e ferramentas usadas;
c) experiência em projetos desta dimensão;
d) conhecimento e experiência na metodologia científica;
e) profunda realização pessoal.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. **The fundamentals of mapping objects to relational databases.** [S.l]: AgileAlliance, 2003. Disponível em: <http://www.agiledata.org/essays/mappingObjects.html>. Acesso em: 12 set. 2003.

AMERICAN NATIONAL STANDARDS INSTITUTE – ANSI. **ANSI INCITS 135-1992:** Information systems – Database language – SQL. Washington: ANSI, 1998.

APACHE SOFTWARE FOUNDATION – ASF. **Velocity.** Version 1.3.1. Delaware: ASF, 2003. Disponível em: <http://jakarta.apache.org/velocity/index.html>. Acesso em: 21 set. 2003.

ATKINSON, Malcolm et al. **The object-oriented database system manifesto.** [S.l]: CiteSeer, 1989. Disponível em: <http://citeseer.ist.psu.edu/atkinson89objectoriented.html>. Acesso em: 1 set. 2003.

CATELL, R. G. G. et al. **The object data standard: ODMG 3.0.** San Francisco: Morgan Kaufmann, 2000.

ECLIPSE.ORG**. Eclipse**. Version 2.1. [S.l.]: Eclipse.org, 2003. Disponível em: <http://www.eclipse.org>. Acesso em: 20 ago. 2003.

GENTLEWARE. **Poseidon.** comun. ed. Version 1.6. Hamburgo: Gentleware, 2003. Disponível em: <http://www.gentleware.com/>. Acesso em: 20 ago. 2003.

ISO. **ISO/IEC 9075-1:1999:** Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). Genebra, 1999.

_____. **ISO/IEC 10646-1:2000:** Information technology – Universal multiple-octet coded character set (UCS) - Part 1: Architecture and basic multilingual plane. Genebra, 2000.

JAVA.NET. **Java compiler compiler:** The Java parser generator. [S.l.]: Java.net, 2003. Disponível em: <https://javacc.dev.java.net/>. Acesso em: 21 set. 2003.

MYSQL AB. **MySQL database server.** Version 4.0. Sweden: MySQL AB, 2003. Disponível em: <http://www.mysql.com/products/mysql/index.html>. Acesso em: 21 set. 2003.

OBJECT MANAGEMENT GROUP – OMG. **Unified Modeling Language specification.** Version 1.3. [S.l.]: OMG, 2000. Disponível em: <http://www.omg.org>. Acesso em: 20 ago. 2003.

RAMAKRISHNAN, Raghu. **Database management systems.** Boston: McGraw-Hill, 1998.

STROUSTRUP, Bjarne. **The C++ programming language.** 2nd ed. Massachusetts: Addison-Wesley, 1994.

SUN MICROSYSTEMS. **Java 2 platform standard edition.** Version 1.4.2. Palo Alto: Sun Microsystems, 2003. Disponível em: <http://java.sun.com/>. Acesso em: 21 set. 2003.

UNICODE. **The unicode standard.** Version 4.0. [S.l.]: Unicode, 2003. Disponível em: <http://www.unicode.org/>. Acesso em: 21 set. 2003.

ANEXO A – BNF da linguagem OQL

Abaixo segue o BNF da linguagem OQL, conforme ela é suportada por este software.

```
query ::= ( selectExpr | orderByExpr )

orderByEntry ::= ( ( expr ) ( <ASC> | <DESC> )? )

selectExpr ::= ( <SELECT> ( <DISTINCT> )? projectionAttributes
               <FROM> orderByExpr )

orderByExpr ::= groupByExpr ( <ORDER> <BY> orderByEntry ( ","
                orderByEntry )* )?

groupByExpr ::= whereExpr ( <GROUP> <BY> groupColumn ( ","
                groupColumn )* ( <HAVING> expr )? )?

whereExpr ::= iteratorExpr ( <WHERE> expr )?

iteratorExpr ::= expr ( ( <AS> )? labelIdentifier ( "," expr
                 ( <AS> )? labelIdentifier )* )?

projectionAttributes ::= projection ( "," projection )*

groupColumn ::= expr <AS> labelIdentifier

projection ::= expr ( ( <AS> )? labelIdentifier )?

exprList ::= ( expr ( "," expr )* )

expr ::= ( andExpr ( <OR> andExpr )* )

andExpr ::= ( equalityExpr ( <AND> equalityExpr )* )

equalityExpr ::= ( relationalExpr ( eqOp relationalExpr )* )

eqOp ::= "=" | "<>"

relationalExpr ::= ( additiveExpr ( relOp additiveExpr )* )

relOp ::= ">" | ">=" | "<" | "<="

additiveExpr ::= ( multiplicativeExpr ( addOp
                 multiplicativeExpr )* )

addOp ::= "+" | "-" | "||" | <UNION> | <UNION> <DISTINCT> |
          <EXCEPT>

multiplicativeExpr ::= ( inExpr ( multOp inExpr )* )

multOp ::= "*" | "/" | <MOD> | <INTERSECT>

inExpr ::= ( unaryExpr ( <IN> unaryExpr )? )

unaryExpr ::= "+" unaryExpr | unOp unaryExpr |
              postfixExpr ( <IS> ( <NOT> )? ( <NULL> |
              postfixInner ) )?
```

Continuação do BNF da linguagem OQL.

```
unOp ::= "-" | <NOT>

postfixExpr ::= ( primaryExpr ( ( "." | "->" ) postfixInner )? )

postfixInner ::= ( labelIdentifier ( argList )? | "(" query ")" )
                 ( ( "." | "->" ) postfixInner )?

primaryExpr ::= collectionExpr | aggregateExpr |
                collectionConstruction | caseExpr | dateExpr |
                labelIdentifier ( argList )? | literal |
                ":" ( <Identif> | <IntegerConst> ) |
                "(" query ")"

collectionExpr ::= ( <FIRST> | <LAST> | <EXISTS> ) "(" query ")"

aggregateExpr ::= ( <SUM> | <MIN> | <MAX> | <AVG> | <COUNT> )
                  "(" query ")"

collectionConstruction ::= ( <SET> | <LIST> | <BAG> ) "(" expr
                           ( "," expr )* ")"

caseExpr ::= <CASE> ( expr )? caseEntryList ( <ELSE> expr )?
             <END>

caseEntryList ::= ( caseEntry ( caseEntry )* )

caseEntry ::= <WHEN> expr <THEN> expr

dateExpr ::= <DATE> "(" expr "," expr "," expr ")"

argList ::= "(" ( expr ( "," expr )* )? ")"

labelIdentifier ::= <Identif>

literal ::= booleanLiteral | numericLiteral |
            stringLiteral | <NULL> )

booleanLiteral ::= <FALSE> | <TRUE>

numericLiteral ::= <NumericConst> | <IntegerConst>

stringLiteral ::= <StringConst>
```

Fim.