Prof. Everton Castelão Tetila / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 2, Issue 5, September- October 2012, pp.540-549

Projeto de Banco de Dados Orientado a Objetos com Db4objects e Java

Prof. Everton Castelão Tetila
Universidade Federal da Grande Dourados
Mestre em Engenharia de Produção pela Universidade Paulista
Faculdade de Ciências Exatas e Tecnologia
Rodovia Dourados/Itahum, Km 12 - Unidade II. Caixa Postal 533, CEP: 79.804-970
Dourados – MS, Brasil.

Resumo
Os bancos de dados tradicionais têm suprido as necessidades da maioria das aplicações comerciais convencionais, mas possuem algumas limitações quando existe a necessidade de tratar objetos complexos e efetuar operações não convencionais. O sistema de banco de dados orientado a objetos oferece a flexibilidade para lidar com requisitos sem estar limitado ao tipo de dados e linguagens de consulta dos sistemas tradicionais. Nele, o projetista tem o poder de especificar tanto a estrutura de objetos complexos quanto as operações que podem ser aplicadas a eles. Com o intuito de mostrar os conceitos e relatar a experiência dos bancos de dados orientado a objetos, este trabalho apresenta o projeto Sistema de Matrículas do curso de Bacharelado em Sistemas de Informação da UFGD. Para a realização deste projeto, foram utilizados os aplicativos NetBeans IDE 7.1 e Db4o 8.0. Palavras-Chave bancos de dados orientados a objetos, Java, Db4o, Sistema de Matrículas.

Palavras-Chave: bancos de dados orientados a objetos, Java, Db4o, Sistema de Matrículas.

1. Introdução

Sistemas e modelos de dados tradicionais, como relacionais, de rede e hierárquicos, têm tido muito sucesso no desenvolvimento das tecnologias de banco de dados exigidas para muitas aplicações de banco de dados de negócios tradicionais. Porém, eles têm certas deficiências quando aplicações de bancos de dados mais complexas precisam ser projetadas e implementadas – por exemplo, bancos de dados para projeto de engenharia e manufatura (CAD/CAM e CIM[i]), experimentos científicos, telecomunicações, sistemas de informações geográficas e multimídia. Essas aplicações mais recentes possuem requisitos e características que diferem daqueles das aplicações de negócios tradicionais, como estruturas mais complexas para objetos armazenados; a necessidade de novos tipos de dados para armazenar imagens, vídeos, ou itens de texto grandes; transações de maior duração; e a necessidade de definir operações fora do padrão específicas da aplicação (ELMASRI, R.; NAVATHE, S. B., 2010).

O modelo de dados orientado a objetos (OO) apresenta a flexibilidade necessária para tratar os requisitos sem estar limitado ao tipo de dados e a linguagem de consulta de banco de dados tradicionais, como a *Structured Query Language* (SQL). Isso é possível graças ao poder dado ao projetista para especificar tanto a estrutura de objetos complexos (classes) quanto as operações que podem ser aplicadas a esses objetos (métodos).

Segundo Silberschatz *et. al.* (2005), à medida que as bases de dados foram sendo utilizadas por um número maior de aplicações, as limitações impostas pelo modelo relacional tornaram-se um obstáculo. Como resultado, pesquisadores da área de bancos de dados inventaram novos modelos de dados que superassem as restrições do modelo relacional.
Uma outra razão para o uso crescente de bancos de dados OO, decorre-se do sucesso atual das linguagens de programação OO, como JAVA, C++ e PHP no desenvolvimento de aplicações de software. Para Chaudhri & Zicari (2000) e Dan Lo *et. al.* (2002), a oferta de bancos de dados OO tem aumentado, principalmente, por causa do crescimento de uso da plataforma Java.

O modelo de dados objeto-relacional (OR), por sua vez, é composto por tabelas e objetos. As tabelas representam as classes e cada registro de uma tabela representa uma instância da classe correspondente, ou seja, um objeto. Para armazenar os objetos no banco de dados é preciso utilizar um *framework*[ii] que faça o mapeamento OR (*Hibernate*, por exemplo). Esse mecanismo causa o que se costuma chamar de diferença de impedância OR. "Diferença de impedância é uma definição procedente da engenharia elétrica e é usada na análise de sistemas para identificar a inadequada ou excessiva capacidade de um sistema para se ajustar a outro" (AMBLER, 2006 *apud* LEONE, A.; CHEN D., 2007).
A Figura 1 ilustra a diferença entre os modelos de dados OR e OO.

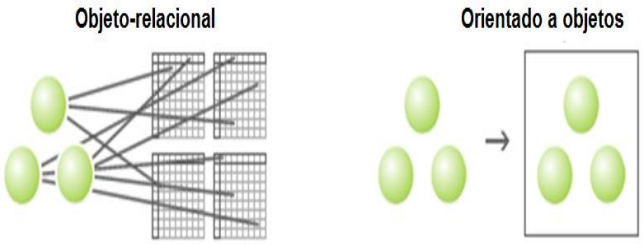

FIGURA 1. Demonstração de armazenamento de dados: Banco de Dados OR *versus* OO. Fonte: DB4O (2009).

A diferença de impedância OR é causada pelo fato de que a lógica OO é baseada em princípios da engenharia de software que modelam os objetos conforme o domínio do problema, enquanto o modelo relacional é baseado em princípios matemáticos que organizam dados para tornar o armazenamento e a recuperação eficientes (PATERSON, 2004).

Nesse sentido, o presente trabalho apresenta o projeto de banco de dados OO – Sistema de Matrículas – do curso de Bacharelado em Sistemas de Informação da Universidade Federal da Grande Dourados (UFGD). Isto permite conceituar os tipos de dados da Linguagem de Definição de Objetos, *Object Definition Language* (ODL); apresentar a Linguagem de Consulta de Objetos, *Object Query Language* (OQL); e descrever o acoplamento (*binding*) com a linguagem Java.

2. Metodologia

Segundo GIL (2010), uma pesquisa pode ser classificada em: exploratória, descritiva ou explicativa.

A pesquisa exploratória, aplicada neste trabalho, envolve o levantamento bibliográfico, o que proporciona maior familiaridade com o problema, a fim de torná-lo mais explícito.

Com base em procedimentos técnicos, GIL (2010) classifica uma pesquisa de várias formas. Neste trabalho são utilizadas as seguintes pesquisas:

✓ Pesquisa bibliográfica – É desenvolvida com base em material já elaborado, constituído principalmente de livros e artigos científicos. Esta pesquisa é a base para a obtenção dos conhecimentos científicos e técnicos para desenvolver o projeto de banco de dados OO, proposto por este trabalho.

✓ Pesquisa-ação – Consiste em um tipo de pesquisa com base empírica que é concebida e realizada em estreita associação com uma ação ou com a resolução de um problema coletivo e no qual os pesquisadores e participantes representativos da situação ou do problema estão envolvidos de modo cooperativo ou participativo.

Marconi (2007) informa que uma pesquisa quanto ao objeto pode ser classificada em: bibliográfica, de laboratório e de campo. Uma pesquisa de campo foi realizada com a Universidade Federal da Grande Dourados (UFGD), para mostrar os conceitos e relatar a experiência dos bancos de dados OO. Os dados coletados foram baseados nos modelos da *Unified Modeling Language* (UML) utilizados para criar o projeto de banco de dados OO; e os resultados foram analisados com base na experiência pessoal dos participantes do estudo.

3. Conceitos de bancos de dados orientados a objetos

Um banco de dados orientado a objetos (BDOO) é uma coleção de dados na forma de objetos, os quais só podem ser manipulados por métodos definido para a classe a qual eles pertencem. Os conceitos de BDOOs são os mesmos das linguagens de programação OO – a diferença está na persistência dos objetos. A persistência dos objetos consiste no armazenamento de dados em memória secundária, de modo a garantir a permanência das informações.

BDOOs são projetados para serem integrados diretamente com aplicações que utilizam a linguagem de programação OO. Com isso, o armazenamento de objetos ocorre de forma transparente, não existindo a necessidade de mapear código como no modelo OR. Além disso, BDOOs são adequados para trabalhar com objetos complexos, como imagens e textos longos.

3.1 Identidade de objeto, estrutura de objeto e construtores de tipos

O sistema de gerenciamento de banco de dados orientado a objetos (SGBDOO) garante que cada objeto tenha uma identidade única gerada por ele, o Identificador de Objeto, ou *Object Identifier* (OID). Para garantir que este OID seja único, ele não depende dos dados armazenados no objeto. O OID não é visível aos usuários externos, mas é usado pelo sistema para identificar o objeto, criar e gerar referência entre eles.

Um modo formal de representar os objetos em bancos de dados OO é visualizar cada objeto como uma tripla (i, c, v), na qual "i" é o identificador único do objeto (OID), "c" é o construtor de tipo (ele indica como o estado do objeto é construído) e "v" é o estado do objeto (ou valor corrente) (ELMASRI, R.; NAVATHE, S. B., 2005).

O modelo de dados normalmente inclui vários construtores de tipos. Os construtores de tipos mais comuns são *atom*, *tuple*, *set*, *bag*, *list* e *array*. O tipo *atom* é formado por um valor atômico, como inteiros, números reais, cadeias de caracteres, boleanos e outros tipos básicos que o sistema tenha suporte. Objetos do tipo *tuple* são construídos através da união de atributos, como tuplas (tipos de registro) no modelo relacional básico. Por exemplo, um tipo estruturado que pode ser criado é: struct Nome<PrimeiroNome:string, InicialMeio: char, Sobrenome: string>.

Os construtores *Set*, *Bag*, *List* e *Array* são conhecidos como Tipos Coleção ou Tipos

Empilhados. *Set* e *Bag* são representados pelo conjunto de objetos, e formado pelo conjunto dos OIDs dos objetos referenciados. *Set* só aceita valores distintos e, em *Bag*, os valores podem se repetir. Os tipos *List* e *Array* também são formados por um conjunto de OIDs, mas são ordenados. A principal diferença é que o tipo *List* comporta um número indeterminado de valores e, em *Array*, esta quantidade deve ser pré-estabelecida.

3.2 Encapsulamento de operações, métodos e persistência

O conceito de encapsulamento é uma das principais características das linguagens e sistemas OO. Nos bancos de dados tradicionais, porém, ele não é aplicado, pois toda a estrutura do banco está visível e existem operações padronizadas que são aplicáveis a todos os tipos de objetos. Em BDOO este conceito é aplicado separando os atributos em visíveis e ocultos. Os Atributos Visíveis podem ser acessados diretamente, já os Atributos Ocultos só são acessados através de métodos específicos para cada objeto, os quais são predefinidos pelo programador. Atributos visíveis e ocultos são semelhantes à denominação de atributos *public* e *private* na programação OO.

Os métodos são utilizados para manipular os dados ocultos do banco de dados, eles podem ser utilizados para criar, destruir, recuperar, fazer cálculos e atualizar os dados.

Nas aplicações desenvolvidas em linguagens de programações OO os objetos são criados pelos construtores da classe que ele pertence, mas nem todos os objetos são armazenados no banco de dados. Os objetos que, quando a aplicação é encerrada, deixam de existir, são chamados de Objetos Transientes. Porém, objetos transientes podem ser armazenados no banco de dados e, a partir disso, tornam-se Objetos Persistentes.

Há duas maneiras de tornar um objeto transiente em persistente, por nomeação e por alcançabilidade.

Persistência por Nomeação consiste em dar um nome persistente ao objeto; todos os nomes devem ser únicos para cada objeto em um determinado banco de dados; o nome persistente será utilizado para iniciar o acesso ao banco de dados. Como não é interessante dar nomes únicos para todos os objetos em um grande banco de dados, devido à grande quantidade de nomes que seriam necessários, a maioria dos objetos são persistidos através do mecanismo de Alcançabilidade. Por meio deste mecanismo todos os objetos que forem alcançados através de um objeto persistente também se tornam persistentes. Por exemplo, se existir um objeto A que é construído, através de um construtor ***Set*** que contenha todos os objetos de uma classe B e, este objeto A for nomeado tornando-o persistente, todos os objetos de B que fazem parte, ou forem adicionados a este conjunto, também se tornarão persistentes, pois estarão alcançáveis através de A.

3.3 Hierarquias de tipo e herança

Um banco de dados OO pode identificar os objetos que pertencem ao mesmo tipo (ou classe) e também pode definir novos tipos com base nos tipos conhecidos.

Quando um tipo é definido, através de um tipo predefinido, pode-se chamá-lo de Subtipo; e o tipo que deu origem a ele de Supertipo. Um subtipo herda todos os métodos e atributos que estejam definidos para o seu supertipo, assim não há a necessidade de definí-lo novamente – apenas acrescenta-se os métodos e atributos necessários para o subtipo. Dessa forma, é possível criar uma hierarquia de tipos para mostrar os relacionamentos de supertipo e subtipo entre todos os tipos declarados no sistema.

3.4 Objetos complexos

Uma das grandes vantagens dos bancos de dados OO é a facilidade para lidar com Objetos Complexos. Os objetos complexos podem ser divididos em estruturados e não estruturados.

Os objetos complexos Não-estruturados são basicamente objetos que exigem grande espaço de armazenamento, como imagens e textos longos (documentos). São chamados de não-estruturados, pois o SGBDOO não conhece a sua estrutura.

Os softwares do SGBDOO não possuem capacidade para processar diretamente condições de seleção e outras operações baseadas em valores desses objetos, a não ser que a aplicação forneça o código para realizar as operações de comparação necessárias para a seleção. Em um SGBDOO, isso pode ser obtido definindo-se o novo tipo de dado abstrato para os objetos não interpretados e fornecendo os métodos para seleção, comparação e apresentação desses objetos. Por exemplo, considere objetos que são imagens *bitmap* bidimensionais. Suponha que a aplicação precise selecionar a partir de uma coleção de tais objetos somente aqueles que incluam certo padrão. Nesse caso, o usuário deve fornecer o programa de reconhecimento do padrão, como um método em objetos do tipo *bitmap*. O SGBDOO recupera, então, um objeto do banco de dados e aplica nele método para reconhecimento do padrão para determinar se o objeto adere ao padrão desejado (ELMASRI, R.; NAVATHE, S. B., 2005).

Já os objetos complexos Estruturados têm suas estruturas conhecidas, pois são objetos derivados do uso recursivo dos construtores existentes, fornecidos pelo SGBDOO, formando assim, vários níveis de recursão.

Existem dois tipos de semântica para referenciar os objetos complexos e seus componentes de cada nível.

A Semântica de Propriedade, ou "é-parte-de" e "é-componente-de", os subobjetos são tratados como parte do objeto complexo. Dessa maneira, estes

subobjetos não possuem OIDs e tem que ser acessados por métodos no objeto proprietário. Como são parte do objeto proprietário, se este for excluído, os subobjetos também serão.

Na Semântica de Referência, ou "é-associado-com", os componentes dos objetos complexos são tratados como objetos independentes que podem ser referenciados pelo objeto complexo. Sendo assim, os objetos que compõem o objeto complexo possuem os seus próprios OIDs e métodos. Além disso, não são excluídos se o objeto complexo for excluído e, quando o objeto complexo precisar acessar seus dados, deve ser feito através de seus métodos predefinidos.

4. Padrões de Banco de Dados de Objetos

Esta seção apresenta o padrão *Object Data Management Group* (ODMG). O padrão é composto pelos componentes: modelo de objetos, *Object Definition Language* (ODL) e *Object Query Language* (OQL), descritos a seguir.

4.1 O Modelo de Objetos ODMG

O modelo de objetos ODMG fornece os tipos de dados, os tipos construtores e outros conceitos que podem ser utilizados na ODL para especificar esquemas de bancos de dados de objetos. Assim, ele deve fornecer um modelo de dados padrão para banco de dados orientados a objetos, da mesma forma que a SQL descreve um modelo de dados padrão para bancos de dados relacionais.

No modelo de objetos ODMG, objetos e literais são os blocos de construção básicos do modelo de objetos. A principal diferença entre os dois é que um objeto possui um OID e um estado (ou valor atual), enquanto um literal possui somente um valor, mas não um OID.

Um Objeto é composto por identificador, nome, tempo de vida e estrutura. O Identificador é um componente que possui uma identificação única para o objeto em todo o banco de dados (OID). Todo objeto deve possuir um identificador. Opcionalmente, pode ser dado um Nome único em todo o banco de dados para um objeto, pelo qual o sistema é capaz de localizá-lo – os nome são utilizados como ponto de partida em um banco de dados OO. Localizando um objeto pelo nome, o usuário pode localizar outros objetos que são referenciados por ele. O Tempo de Vida, por sua vez, define se os objetos são persistentes ou transientes. E a Estrutura especifica como ele é construído pelos construtores de tipos. A estrutura define se um objeto é atômico (inteiro, real, caracteres, boleanos, etc) ou coleção (*set*, *bag*, *list e array*).

Um Literal é um valor que não possui um OID. Os literais podem ser atômicos, estruturados ou coleção. Atômicos são aqueles valores de dados básicos e são predefinidos. Estruturados são aqueles valores construídos a partir de tuplas, estruturas embutidas, como *Date*, *Interval*, *Time* e *Timestamp*, e estruturas definidas pelo usuário. Literais Coleção especificam um valor que corresponde a uma coleção de objetos ou de valores, mas a coleção propriamente dita não possui um OID.

São considerados objetos atômicos todos os objetos definidos pelo usuário que não sejam objetos de coleção, desse modo também inclui objetos estruturados criados com o uso do construtor *struct*. A maioria desses objetos serão estruturados, possuirão uma estrutura complexa com vários atributos, relacionamentos e operações, mas ainda assim, por não serem objetos coleção, serão considerados objetos atômicos. Eles são definidos com o uso da palavra-chave *class* na ODL e possuem três componentes básicos: atributos, relacionamentos e operações.

Um Atributo é uma propriedade que descreve algum aspecto de um objeto. Atributos possuem valores (os quais normalmente são literais com uma estrutura simples ou complexa) que são armazenados dentro do objeto. Porém, os valores de atributos também podem ser OIDs de outros objetos. Um Relacionamento é uma propriedade que especifica que dois objetos no banco de dados estão relacionados. No modelo de objeto do ODMG, somente relacionamentos binários são representados explicitamente, e cada relacionamento binário é representado por um *par de referências inversas* especificadas por meio da palavra-chave *relationship*. Além dos atributos e relacionamentos, o projetista pode incluir Operações nas especificações de tipo de objeto (classe). Cada tipo objeto pode possuir uma série de assinaturas de operação, que especificam o nome da operação, seus tipos de argumento e seu valor retornado, se for o caso. Os nomes de operação são exclusivos dentro de cada tipo de objeto, mas eles podem ser sobrecarregados, fazendo que o mesmo nome de operação apareça em tipos de objetos distintos. (ELMASRI, R.; NAVATHE, S. B., 2010).

4.2 Object Definition Language (ODL) - Linguagem de Definição de Objetos

A ODL é projetada para dar suporte às construções semânticas do modelo de objeto ODMG e é independente de qualquer linguagem de programação em particular. Seu uso principal é para criar especificações de objetos — ou seja, classes e interfaces. Logo, a ODL não é uma linguagem de programação completa. Um usuário pode especificar um esquema de banco de dados na ODL independentemente de qualquer linguagem de programação, e depois utilizar os *bindings* da linguagem específica para indicar como as construções ODL podem ser mapeadas em construções nas linguagens de programação específicas, como C++, SMALLTALK e JAVA (ELMASRI, R.; NAVATHE, S. B., 2010).

4.3 Object Query Language (OQL) - Linguagem de Consulta de Objetos

A OQL foi projetada para trabalhar de perto com as linguagens de programação para as quais um *binding* ODMG é definido, como C++, SMALLTALK e JAVA. Logo, uma consulta OQL embutida em uma dessas linguagens de programação pode retornar objetos que combinam com o sistema de tipos dessa linguagem. Além disso, as implementações de operações de classe em um esquema ODMG podem ter seu código escrito nessas linguagens de programação. A sintaxe OQL para consultas é semelhante à sintaxe da linguagem de consulta do padrão de consulta relacional, SQL, com recursos adicionais para os conceitos ODMG, como identidade de objeto, objetos complexos, operações, herança, polimorfismo e relacionamentos (ELMASRI, R.; NAVATHE, S. B., 2010).

5. Db4objects (Db4o)

O *Db4objects* (Db4o) é um banco de dados orientado a objetos, desenvolvido como projeto software livre de código aberto (*open-source*), projetado para aplicações do tipo embarcada, cliente-servidor e desktop. O banco de dados também é distribuído em uma licença comercial. "Um serviço baseado em assinatura custa 1.200,00 dólares por ano" (MITCHELL R. L., 2005).

O Db4o permite armazenar os objetos diretamente no banco de dados, sem precisar utilizar comandos SQL ou qualquer tipo de *framework* que faça o mapeamento objeto-relacional. De acordo com o site do fabricande "avaliações de banco de dados mostram que o Db4o pode ser até 44 vezes mais rápido que o *Hibernate* e *MySQL*, uma combinação popular de mapeador objeto-relacional e banco de dados" (DB4OBJECTS, 2009).

Empresas como a Bosch, Hertz, BMW, Intel, Seagate entre outras, utilizam o Db4o. São inúmeras as vantagens em relação ao banco de dados relacional: o banco de dados é nativo em *Java* ou *.Net*, oferece rapidez de inserção e consulta (processamento de 200 mil objetos por segundo), utiliza pouco recurso computacional, tem fácil aprendizado, não possui nenhuma linha de código SQL para *Create, Read, Update e Delete* (CRUD), e disponibiliza acesso direto ao banco de dados sem utilizar mapeamento objeto-relacional (GUERRA, 2007).

5.1 Operações CRUD

A seguir serão demonstradas as operações *Create*, *Read*, *Update* e *Delete* para a manipulação de objetos no Db4O. Maiores detalhes podem ser observados em (DB4OBJECTS, 2011).

Para configurar o banco de dados *Db4O 8.0* crie um novo projeto no *NetBeans IDE 7.1* e adicione a biblioteca *db4o-8.0.184.15484-all-java5* à sua aplicação (exemplo: *Bibliotecas/Adiocionar JAR/pasta.../db4o-8.0.184.15484-all-java5*). Esse é o único arquivo necessário para a aplicação comunicar-se com o banco de dados.

Para demonstrar a manipulação dos objetos no banco de dados será utilizada a classe Aluno a seguir.

```
1   public class Aluno {
2       private int codigo;
3       private String nome;
4       public Aluno(int codigo, String nome) {
5           this.codigo = codigo;
6           this.nome = nome;
7       }
8       public int getCodigo(){
9           return codigo;
10      }
11      public void setCodigo(int codigo){
12          this.codigo=codigo;
13      }
14      public String getNome(){
15          return nome;
16      }
17      public void setNome(String nome){
18          this.nome=nome;
19      }
20      public String toString(){
21          return "Código:" + codigo + " Nome:" + nome;
22      }
23  }
```

Para abrir o banco de dados, a classe *Db4oEmbedded* possui o método estático *openFile*(). No código abaixo, o método *openFile*() é o método responsável por abrir o banco de dados *banco.yap* (linha 5). Caso o banco de dados não exista, ele irá criá-lo no caminho especificado. A interface *ObjectContainer*, por sua vez, fornece os métodos para armazenar, consultar, alterar e excluir objetos. Dentro do bloco de código *try* (linhas 6 a 8) serão realizadas as operações CRUD.

```
1   import com.db4o.Db4oEmbedded;
2   import com.db4o.ObjectContainer;
3   public class Main {
4       public static void main(String[] args) {
5           ObjectContainer db = Db4oEmbedded.openFile("c:/banco.yap");
6           try {
7               // Códigos para as operações CRUD.
8           } finally {
9               db.close();
10          }
11      }
12  }
```

No código abaixo é realizada a operação de inserção. Para isso, é instanciado um objeto da classe Aluno, definido através do construtor. Para armazenar o aluno no banco de dados é necessário passar o objeto como parâmetro para o método

store() (linha 8).

```
1   import com.db4o.Db4oEmbedded;
2   import com.db4o.ObjectContainer;
3   public class Main {
4       public static void main(String[] args) {
5           ObjectContainer db = Db4oEmbedded.openFile("c:/banco.yap");
6           try {
7               Aluno novo = new Aluno(1, "Danilo Pedroso");
8               db.store(novo);
9               System.out.println("Aluno armazenado: " + novo);
10          } finally {
11              db.close();
12          }
13      }
14  }
```

Como resultado do código acima, têm-se como saída: Aluno armazenado: Código: 1 Nome: Danilo Pedroso.

Para recuperar objetos no Db4o, existem três formas possíveis: através da *Query-By-Example* (QBE*), Native Queries* (NQ) *e Simple Object Data Access* (SODA) (DB4OBJECTS, 2011). A seguir, será demonstrado como recuperar objetos através da QBE.

Conforme Parsaye *el al.* (1989) "a QBE foi projetada para que consultas fossem construídas interativamente". Ela permite realizar uma consulta em que o usuário especifica os valores dos atributos que ele deseja que sejam recuperados, construindo um exemplo da operação desejada.

Por exemplo, o código abaixo seleciona todos os alunos com o nome *Danilo Pedroso*, independente do número de seu código (atributo). Neste caso, deve-se utilizar o valor zero para o atributo código, de forma que este não precise ser especificado – zero é o valor padrão para números inteiros.

```
1   import com.db4o.Db4oEmbedded;
2   import com.db4o.ObjectContainer;
3   import com.db4o.ObjectSet;
4   public class Main {
5       public static void main(String[] args) {
6           ObjectContainer db = Db4oEmbedded.openFile("c:/banco.yap");
7           try {
8               Aluno novo = new Aluno(0,"Danilo Pedroso");
9               ObjectSet<Aluno> lista = db.queryByExample (novo);
10              while (lista.hasNext()){
11                  System.out.println(lista.next());
12              }
13          } finally {
14              db.close();
15          }
16      }
17  }
```

No código abaixo é realizada a operação de alteração. Para isso, é necessário utilizar o método *store*() para alterar o objeto no banco de dados. O Db4o é capaz de identificar se o objeto existe no banco de dados para ser alterado, caso contrário, ele irá armazenar um novo objeto no banco de dados.

```
1   import com.db4o.Db4oEmbedded;
2   import com.db4o.ObjectContainer;
3   import com.db4o.ObjectSet;
4   public class Main {
5       public static void main(String[] args) {
6           ObjectContainer db= Db4oEmbedded.openFile("c:/banco.yap");
7           try {
8               ObjectSet<Aluno> lista = db.queryByExample (new Aluno (1,"Danilo
9               Pedroso"));
10              Aluno aluno = lista.next();
11              aluno.setNome("Danilo Pedroso Vargas");
12              db.store(aluno);
13              lista = db.queryByExample (Aluno.class);
14              while (lista.hasNext()){
15                  System.out.println(lista.next());
16              }
17          } finally {
18              db.close();
19          }
20      }
21  }
```

No código acima, primeiro é selecionado o aluno *Danilo Pedroso* de código 1 (linhas 8 e 9). Logo após, é possível recuperar o aluno para ser alterado (linha 10). Depois disso, é alterado o nome do objeto aluno para *Danilo Pedroso Vargas* (linha 11). Na linha 12 o banco de dados é atualizado. Finalmente, é mostrado o resultado da alteração nas linhas 13 a 16.

O processo de exclusão é semelhante ao de atualização. Primeiro retira-se o objeto do banco, depois aplica-se o método *delete*():

```
1   import com.db4o.Db4oEmbedded;
2   import com.db4o.ObjectContainer;
3   import com.db4o.ObjectSet;
4   public class Main {
5       public static void main(String[] args) {
6           ObjectContainer db= Db4oEmbedded.openFile("c:/banco.yap");
7           try {
```

```
8               ObjectSet<Aluno> lista = db.queryByExample (new Aluno (1,"Danilo
9               Pedroso Vargas"));
10                      Aluno aluno = lista.next();
11                      db.delete(aluno);
12                      lista = db.queryByExample (Aluno.class);
13                      while (lista.hasNext()){
14                              System.out.println(lista.next());
15                      }
16              } finally {
17                      db.close();
18              }
19      }
20  }
```

No código acima, primeiro é selecionado o aluno *Danilo Pedroso* de código 1 (linha 8 e 9). Logo após, é possível recuperar o aluno para ser excluído (linha 10). Depois disso, é excluído o objeto aluno do banco de dados (linha 11). Finalmente, é mostrado o resultado nas linhas 12 a 15.

Para armazenar objetos relacionados, utiliza-se o método *store*() no objeto que determina a relação entre objetos. Por exemplo, para armazenar o curso de um determinado aluno é preciso armazenar apenas o objeto aluno, pois curso é uma variável de aluno. Para isso, deve-se modificar a classe aluno, adicionando um relacionamento entre as classes Aluno e Curso. A seguir, segue a nova classe Curso e a classe Aluno alterada.

```
1   public class Curso {
2       private String nome;
3       private String periodo;
4       public Curso(String nome, String periodo) {
5           this.nome = nome;
6           this.periodo = periodo;
7       }
8       public String getNome(){
9           return nome;
10      }
11      public void setNome(String nome){
12          this.nome=nome;
13      }
14      public String getPeriodo(){
15          return periodo;
16      }
17      public void setPeriodo(String periodo){
18          this.periodo=periodo;
19      }
20      public String toString(){
21          return "Nome:" + nome + " Periodo:" + periodo;
22      }
23  }
```

```
1   public class Aluno {
2       private int codigo;
3       private String nome;
4       private Curso curso;
5       public Aluno(int codigo, String nome) {
6           this.codigo = codigo;
7           this.nome = nome;
8       }
9       public int getCodigo(){
10          return codigo;
11      }
12      public void setCodigo(int codigo){
13          this.codigo=codigo;
14      }
15      public String getNome(){
16          return nome;
17      }
18      public void setNome(String nome){
19          this.nome=nome;
20      }
21      public Curso getCurso() {
22          return curso;
23      }
24      public void setCurso(Curso curso) {
25          this.curso = curso;
26      }
27      public String toString(){
28          return "Código:" + codigo + " Nome:" + nome + "\n" + "Curso:"+
29          curso.getNome();
30      }
31  }
```

O código abaixo armazena os objetos relacionados aluno e curso utilizando, para isso, o método *store*(). Primeiro são criados o objeto aluno *Danilo Pedroso Vargas* de código 1 (linha 8) e o objeto curso *Sistema de Informação*, período *Noturno* (linha 9). Logo após, o objeto curso é relacionado ao objeto aluno (linha 10). Depois disso, é atualizado o banco de dados (linha 11).

```
1   import com.db4o.Db4oEmbedded;
2   import com.db4o.ObjectContainer;
3   import com.db4o.ObjectSet;
4   public class Main {
5       public static void main(String[] args) {
6           ObjectContainer db= Db4oEmbedded.openFile("c:/banco.yap");
7           try {
8               Aluno aluno = new Aluno(1,"Danilo Pedroso Vargas");
9               Curso curso = new Curso("Sistemas de Informação","Noturno");
10              aluno.setCurso(curso);
11              db.store(aluno);
12          } finally {
13              db.close();
```

```
14                  }
15              }
16      }
```

Para recuperar objetos relacionados, utiliza-se o método *query*(). O código abaixo recupera o curso ao qual o aluno *Danilo Pedroso Vargas* pertence.

```
1       import com.db4o.Db4oEmbedded;
2       import com.db4o.ObjectContainer;
3       import com.db4o.ObjectSet;
4       import com.db4o.query.Query;
5       public class Main {
6               public static void main(String[] args) {
7                       ObjectContainer db= Db4oEmbedded.openFile("c:/banco.yap");
8                       try {
9                               Query query = db.query();
10
        query.constrain(Aluno.class);
11
        query.descend("nome").constrain("Danilo Pedroso Vargas");
12                              Aluno aluno = (Aluno) query.execute().next();
13                              System.out.println (aluno.getCurso());
14                      } finally {
15                              db.close();
16                      }
17              }
18      }
```

6. Projeto de banco de dados OO com Java e Db4o

Esta seção apresenta o projeto de banco de dados OO para o Sistema de Matrículas do curso de Bacharelado em Sistemas de Informação (BSI) da UFGD. Para a realização deste projeto, foram utilizados os seguintes aplicativos:

✓ *NetBeans IDE 7.1*, disponível em (NETBEANS, 2012).
✓ Banco de dados OO *Db4o 8.0*, disponível em (DB4OBJECTS, 2012).
✓ *Java SE Development Kit (JDK) 7u2*, disponível em (ORACLE, 2012).

6.1 Descrição de requisitos

O curso de BSI da UFGD deseja informatizar o seu sistema de matrículas para uma abordagem de banco de dados orientado a objetos. Com isso, será possível especificar tanto a estrutura de objetos complexos, quanto às operações que podem ser aplicadas à esses objetos, sem estar limitado ao tipo de dados e linguagens de consulta disponíveis em sistemas de banco de dados tradicionais.

O banco de dados UFGD registra o coordenador, os docentes, os discentes, as disciplinas e as turmas do curso de BSI. Depois da fase de levantamento e análise de requisitos, foi definida a seguinte descrição do *minimundo* – a parte do sistema que será representada no banco de dados:

✓ O **Coordenador** do curso de BSI define as disciplinas que serão ofertadas no curso em um determinado semestre letivo.
✓ Diversas disciplinas podem ser ofertadas no curso de BSI.
✓ Mais de uma turma pode ser aberta para uma mesma disciplina.
✓ **Discentes** selecionam as disciplinas e se matriculam nas turmas, conforme os horários disponíveis.
✓ **Docentes** usam o sistema para obter a lista de alunos matriculados nas disciplinas que ministram aulas.
✓ Todos os usuários do sistema devem ser validados com login e senha.

6.2 Principais Funcionalidades

✓ **Requisito funcional 1:** Matricular o discente: o sistema deve permitir que o discente matricule-se nas disciplinas ofertadas pelo curso e deve impedir que haja colisão de horários entre as disciplinas ofertadas.
✓ **Requisito funcional 2:** Aprovar matrícula: o sistema deve permitir que o coordenador do curso aprove a matrícula realizada pelo discente. Somente após a aprovação do coordenador, o aluno é adicionado nas turmas ofertadas.
✓ **Requisito funcional 3:** Retirar o discente de uma turma: o sistema deve permitir ao coordenador do curso retirar o discente de uma determinada turma.
✓ **Requisito funcional 4:** Imprimir os discentes matriculados em uma turma: o sistema deve exibir ao docente a listagem dos discentes matriculados em uma determinada turma.
✓ **Requisito funcional 5:** Imprimir os horários do discente: o sistema deve exibir ao discente, os horários das disciplinas que ele possui aula.

6.3 Diagrama de Caso de uso

O diagrama de caso de uso representa as principais funcionalidades do sistema (OMG, 2011). A Figura 2 apresenta o diagrama de caso de uso do Sistema de Matrículas.

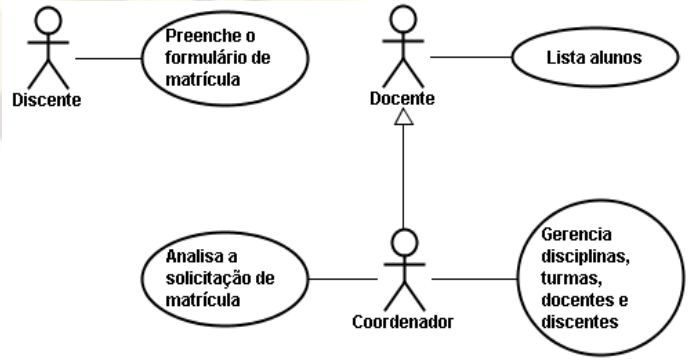

FIGURA 2 – Diagrama de Caso de Uso do Sistema de Matrículas.

6.4 Diagrama de Classe

O diagrama de classe representa a estrutura e as relações das classes (OMG, 2011). A Figura 3 apresenta o diagrama de classe do Sistema de Matrículas.

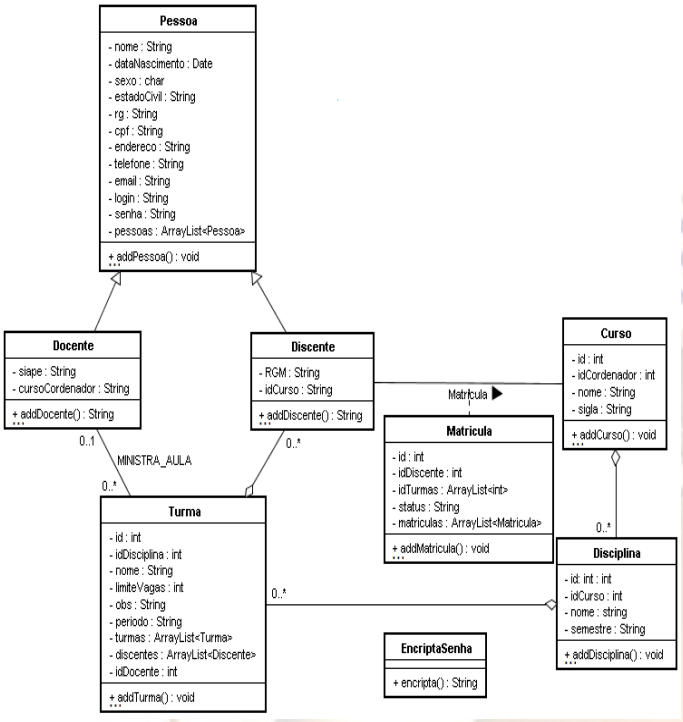

FIGURA 3 – Diagrama de Classes do Sistema de Matrículas.

6.5 Implementação

O código-fonte do Sistema de Matrículas está disponível para *download* em (UFGD, 2012).

7. Conclusão

Este trabalho apresentou o projeto de banco de dados OO para o Sistema de Matrículas do curso de Bacharelado em Sistemas de Informação da UFGD.

O projeto de banco de dados OO foi definido com base nos aplicativos *Db4o 8.0* e *Netbeans IDE 7.1*. A partir da combinação dos aplicativos, verificou-se que ambos mantêm uma correspondência entre os conceitos usados no programa e sua representação no banco de dados – os modelos de dados adotados no Db4o e no Java são idênticos.

Segundo Elmasri e Navathe (2010), os bancos de dados OO são projetados de modo que possam ser integrados diretamente – ou *transparentemente* – ao software desenvolvido usando linguagens de programação OO.

Uma pesquisa-ação foi proposta para mostrar os conceitos dos bancos de dados OO, experimentar as ações do projeto de banco de dados OO e relatar a experiência. Os resultados demonstraram que a principal vantagem encontrada no projeto de banco de dados OO é a facilidade de especificar mudanças à medida que elas ocorram; ou criar novas classes, sem ter que se preocupar com o armazenamento no banco de dados. No Db4o as modificações são transparentes não existindo a necessidade de reestruturação, o que diminui consideravelmente o tempo de implementação e manutenção no banco de dados.

Espera-se, com isso, que o Projeto de banco de dados OO apresentado, venha contribuir com o desenvolvimento de novos projetos de software que utilizem a abordagem de banco de dados OO.

8. Referências

1. AMBLER, S. W. *The Object-Relational Impedance Mismatch*. Ambysoft Inc, 2006. Disponível em: http://www.agiledata.org/essays/impedance Mismatch.html. Acesso em: 18 jan. 2012.
2. CHAUDHRI, A. B; ZICARI, R. *Succeeding with Object Databases:* A Practical Look at Today's Implementations with Java and XML. Wiley, 2000. 464p.
3. DAN LO, C. T; CHANG, M.; FRIEDER, O.; GROSSMAN, D. The Object Behavior of Java Object-Oriented Database Management Systems. Information Technology: Coding and Computing, 2002. *Proceedings*. 247-253p.
4. DB4OBJECTS. *Db4o 8.0 for Java*. 2012. Disponível em: http://www.db4o.com/Download-Now.aspx. Acesso em: 06 jan. 2012.
5. DB4OBJECTS. *Db40 - Banco de objetos de código aberto*. 2009. Disponível em: http://www.db4o.com/portugues/db4o%20Product%20Information%20V5.0(Portuguese).pdf Acesso em: 06 out. 2011.
6. DB4OBJECTS. *Db4o tutorial*. 2011. Disponível em: http://community.versant.com/Documentation/Reference/db4o-8.0/java/tutorial. Acesso em: 27 de outubro de 2011.
7. ELMASRI, R.; NAVATHE, S. B. *Sistemas de Banco de Dados*. 4. ed., São Paulo: Pearson Addison Wesley, 2005. 724p.
8. ELMASRI, R.; NAVATHE. *Sistemas de Banco de Dados*. 6. ed., São Paulo: Pearson Addison Wesley, 2010. 788p.
9. FAYAD, M. E.; SCHMIDT, D. C. Object-oriented Application frameworks. *Communications of the ACM*, vol. 40, 10 p., 1997.
10. GIL, A. C. *Como elaborar projetos de pesquisa?* 5. ed., São Paulo: Atlas, 2010. 200 p.
11. GUERRA, Glaucio. *DB4Objects na terra de gigantes do BD relacional com Java - Parte I*. SQL Magazine, 2007. Disponível em: http://www.devmedia.com.br/articles/view-comp.asp?comp=4121. Acesso em: 05 de novembro de 2011.

12. LEONE, A.; CHEN D. Implementation of an object oriented data model in an information system for water catchment management: Java JDO and Db4o Object Database. *Environmental Modelling & Software*. vol. 22, p.1805-1810, 2007.
13. MARCONI, M. A. *Metodologia do Trabalho Científico*. 7. ed., São Paulo: Atlas, 2007. 225 p.
14. MITCHELL R. L. *Db4o Database Upgrade Debuts*. Computerworld, 30 p., 2005.
15. NETBEANS. *NetBeans IDE 7.1*. 2012. Disponível em: http://netbeans.org/downloads. Acesso em: Acesso em: 06 jan. 2012.
16. OMG. OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language™ (OMG UML), Superstructure.* 2011. versão 2.4.1. Disponível em: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/. Acesso em: 08 out. 2011.
17. ORACLE. *Java SE Development Kit (JDK) 7u2*. Disponível em: http://www.oracle.com/technetwork/java/javase/downloads/index.html. Acesso em: 06 out. 2011.
18. PARSAYE, K.; CHIGNELL M.; KHOSHAFIAN S.; WONG H. *Intelligent databases: object-oriented, deductive hypermedia technologies*. New York: John Wiley & Sons, 1989.
19. PATERSON, Jim. *Simple Object Persistence with the db4o Object Database*. O'Reilly, Sebastopol, California, 2004. Disponível em: http://onjava.com/pub/a/onjava/2004/12/01/db4o.html. Acesso em: 17 jan. 2012.
20. SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. *Database System Concepts*. 5. ed., McGraw-Hill, 2005. 1168p.
21. UFGD. *Sistema de Matrículas*. Disponível em: http://www.do.ufgd.edu.br/evertontetila-/sistemaMatriculas.zip. Acesso em: 07 jan. 2012.

[i] **Computer-Aided Design/Computer-Aided Manufacturing** (projeto auxiliado por computador/fabricação auxiliada por computador) e *Integrated Manufacturing* (fabricação integrada ao computador).

[ii] **Frameworks** representam uma estrutura formada por blocos pré-fabricados de software que os programadores podem usar, estender ou adaptar para uma solução específica e linguagens de padrões (FAYAD, M. E.; SCHMIDT, D. C., 1997).