

**UNIVERSIDADE PRESBITERIANA MACKENZIE
FACULDADE DE COMPUTAÇÃO E INFORMÁTICA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO**

**PERSISTÊNCIA DE OBJETOS
VIA MAPEAMENTO OBJETO-RELACIONAL**

**CAMILA ARNELLAS COELHO
REINALDO COELHO SARTORELLI**

**São Paulo
2004**

**Camila Arnellas Coelho
Reinaldo Coelho Sartorelli**

**PERSISTÊNCIA DE OBJETOS
VIA MAPEAMENTO OBJETO-RELACIONAL**

**Trabalho de Graduação Interdisciplinar
apresentado à Faculdade de Computação e
Informática da Universidade Mackenzie como
exigência parcial para a conclusão do curso de
Sistemas de Informação.**

Orientador: Prof. Dr. Luciano Silva

**São Paulo
2004**

RESUMO

Este estudo tem como objetivo apresentar as principais técnicas de persistência de objetos, com relevância na técnica de Mapeamento Objeto-Relacional (MOR).

Atualmente, é comum o uso de bancos de dados relacionais no meio corporativo, e da programação orientada a objetos nas aplicações de interface. Os desenvolvedores frequentemente deparam-se com estes dois paradigmas diferentes: o modelo relacional e o modelo de objetos.

Uma maneira de se trabalhar com um paradigma orientado a objetos junto a SGBD relacionais é justamente o propósito da técnica de MOR.

O estudo de caso proposto apresentará um exemplo prático de aplicação destes conceitos, através da utilização de um *framework* para a persistência de objetos, o *Object Relational Bridge* (OBJ).

Palavras-chave: Mapeamento Objeto-Relacional, persistência, impedância, padrões de projeto, OBJ, JDBC.

ABSTRACT

The purpose of this work is to introduce the main current techniques of object persistence, with a special approach on the topic of Object-Relational Mapping (ORM).

Nowadays, the Relational Databases still represent the majority of data storage facilities found on corporations. Meanwhile, object-oriented programming is becoming the technology of choice for most applications. However, programmers often face two different paradigms: the Relational Model and the Object Model.

An elegant and productive way to circumvent this issue is the purpose of ORM. It provides ways to keep RDBMS and still use an object-oriented approach to the data.

A case study is presented as a practical approach of the implementation of ORM techniques, using an O/R framework known as *OJB (Object Relational Bridge)*.

Keywords: Object-Relational Mapping, persistence, impedance, design patterns, OJB, JDBC.

SUMÁRIO

INTRODUÇÃO.....	9
CAPÍTULO 1 - O problema da persistência de objetos.....	11
1.1. Persistência de Dados.....	11
1.2. Persistência em um contexto orientado a objetos.....	14
1.3. O descompasso objeto-relacional.....	18
CAPÍTULO 2 - Mecanismos de persistência de objetos.....	21
2.1. Serialização de objetos.....	21
2.2. Persistência em XML.....	23
2.3. Bancos de Dados Estendidos (OR).....	27
2.4. Bancos de Dados Orientados a Objetos.....	30
2.5. Outros mecanismos de persistência de objetos.....	35
2.6. Conectividade entre o SGBD e aplicações O.O.	36
CAPÍTULO 3 - Mapeamento Objeto-Relacional.....	39
3.1. Persistência em SGBDR.....	39
3.2. Conceitos básicos de MOR.....	42
3.3. Tipos comuns de MOR.....	44
3.4. Implementações de Camadas de Persistência.....	51
3.5. Padrões de Mapeamento Objeto-Relacional.....	56
3.6. Considerações Finais.....	61
CAPÍTULO 4 - Estudo de caso: Object Relational Bridge.....	63
4.1. Introdução.....	63
4.2. Requisitos funcionais.....	63
4.3. Estrutura da aplicação.....	64
4.4. Implementação.....	66

4.5. Comparativos.....	70
CONCLUSÕES.....	76
REFERÊNCIAS BIBLIOGRÁFICAS.....	78
ANEXO A: Script de criação do banco de dados.....	81
ANEXO B: XML de conexão e mapeamento.....	82
ANEXO C: Código do estudo de caso.....	86

LISTA DE ABREVIATURAS

ADT/TAD	Abstract Data Type/Tipo de Dados Abstrato
API	Application Program Interface
BLOB/CLOB	Binary/Character Large Objects
CRUD	Create, Read, Update, Delete
FIFO	First In, First Out
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
MOR/ORM	Mapeamento Objeto-Relacional/Object Relational Mapping
OCI	Oracle Call Interface
ODBC	Open Database Connectivity
ODMG	Object Data Management Group
OID	Object Identifier
OO	Orientação a Objetos
PBR	Persistence by Reachability
SGDB	Sistema Gerenciador de Bancos de Dados
SGDBOO	Sistema Gerenciador de Banco de Dados Orientado a Objetos
SGDBOR	Sistema Gerenciador de Banco de Dados Objeto-Relacional (Estendido)
SGDBR	Sistema Gerenciador de Banco de Dados Relacionais
SQL	Structured Query Language
UML	Unified Modeling Language
XML	Extensible Markup Language
XND	XML Native Database

LISTA DE FIGURAS

Figura 1	Persistência de objetos [extraído de WOL03].	16
Figura 2	Exemplo de Classe “Pessoa”.	21
Figura 3	Representação dos dados em tabela.	24
Figura 4	Representação em tabela de uma classe.	28
Figura 5	Arquitetura de 3 camadas [extraído de IBM00].	40
Figura 6	Exemplo de MOR (adaptação de [RAJ02; Fig. 5.2]).	45
Figura 7	Relacionamento 1:1.	47
Figura 8	Relacionamento 1:n.	48
Figura 9	Relacionamento n:n.	48
Figura 10	Direcionalidade.	49
Figura 11	Gateway Pattern [extraído de FOW02].	57
Figura 12	Active Record Pattern [extraído de FOW02].	57
Figura 13	Data Mapper Pattern [extraído de FOW02].	58
Figura 14	Herança de Tabela Simples [extraído de FOW02].	59
Figura 15	Herança de Tabelas Concretas [extraído de FOW02].	60
Figura 16	Herança de Classe e Tabela [extraído de FOW02].	60
Figura 17	Diagrama de Classes do estudo de caso.	65
Figura 18	Modelo ER do estudo de caso.	65

INTRODUÇÃO

Atualmente, as linguagens de programação orientadas a objetos (ou simplesmente O.O.) possuem um papel importante enquanto paradigma de programação, tendo sua adoção em crescimento constante, tanto na indústria, quanto em meios acadêmicos.

Devido à frequente necessidade das aplicações de persistirem dados, para que possam recuperar a informação gerada por elas em novos processos, e de uma forma que seja possível compartilhá-la entre tipos de sistemas distintos, o uso de técnicas de persistência de objetos em SGBDR (Sistemas Gerenciadores de Bancos de Dados Relacionais), como o mapeamento objeto-relacional (MOR), tem se tornado um tema comum e relevante para aplicações desenvolvidas utilizando a orientação a objetos.

Aplicações de médio e grande porte, em geral, utilizam os SGBDR como principal meio de armazenamento de dados, devido principalmente aos sistemas legados, em detrimento da utilização de Sistemas de Bancos de Dados Orientados a Objetos (SGBDOO), que ainda não atingiram um grau relevante de aceitação.

No Capítulo 1, conceitualiza-se persistência de dados, sua importância, os principais mecanismos e os fatores envolvidos na persistência de objetos. No Capítulo 2, abordam-se os principais métodos de persistência de objetos, e finalmente, no Capítulo 3, apresenta-se o método de MOR, uma técnica que tem como principal propósito contornar o problema de impedância, mais conhecido pelo termo em inglês *impedance mismatch*, que ocorre na utilização da programação orientada a objetos para persistência de dados em bancos relacionais.

Em linhas gerais, pode-se dizer que, através das técnicas de MOR, é possível escrever aplicações de negócio em linguagens orientadas a objetos que persistam objetos em SGBDR, sem a necessidade da codificação de uma linha sequer de instrução SQL. As ferramentas de MOR se encarregam de gerar e executar estas instruções, provendo transparência entre a

aplicação (interface) e o mecanismos de persistência (SGBD).

Estamos utilizando o termo MOR (ou ORM - *Object-Relational Mapping*) para qualquer camada de persistência onde os comandos SQL são gerados automaticamente, a partir de uma descrição baseada em metadados (*metadata*) de mapeamento de objetos em relação às estruturas relacionais (tabelas).

Além da conceituação teórica dos métodos de persistência de objetos - o tema central deste trabalho – com relevância nas tecnologias de MOR para a utilização dos SGBD legados, apresenta-se também uma aplicação prática destes conceitos através da utilização de um arcabouço (*framework*) de persistência de objetos utilizando MOR em Java (Capítulo 4).

O *framework* em questão é o *OJB (Object Relational Bridge)*, que utiliza o mapeamento objeto-relacional para atingir este objetivo: apresentar um mecanismo de persistência de objetos transparente para a aplicação orientada a objeto utilizando SGBDR como meio de armazenamento de dados. A escolha por este *framework* deve-se ao fato de ser um projeto de software livre, robusto, de fácil utilização e de alto grau de aceitação pela comunidade Java em geral.

CAPÍTULO 1 - O problema da persistência de objetos

A persistência dos dados é um fator de importância fundamental para qualquer sistema que trabalhe e dependa de informações e que, portanto, necessite de algum mecanismo de armazenamento estável para estas informações.

A persistência de dados no paradigma da orientação a objetos envolve a persistência dos objetos. As linguagens orientadas a objeto têm como base o conceito de objeto enquanto elemento fundamental da linguagem.

Neste capítulo, aborda-se a necessidade da persistência de dados nos sistemas, e os conceitos envolvidos na persistência de objetos. Reserva-se uma atenção especial para o descompasso existente entre a linguagem relacional, utilizada nos SGBD, e as linguagens orientadas a objetos, isto é, o problema da impedância entre o modelo de dados e o modelo de objetos do paradigma da orientação a objetos.

1.1. Persistência de Dados

Para explicar a persistência de dados, será primeiramente apresentada uma situação real, na qual seria de extrema importância a existência de um mecanismo de persistência. Suponha um sistema que mostre estatísticas em tempo real, onde seus valores passem por constantes atualizações na memória, e com estes dados, gera-se um gráfico vital para a sobrevivência de uma determinada empresa. Caso, porventura, a energia elétrica falhar, os dados contidos no sistema (ou seja, na memória da máquina) se perderão. O prejuízo causado à empresa será imensurável.

Logo, a necessidade de persistência de dados é um requisito fundamental da maioria dos sistemas, seja através de bancos de dados, arquivos de texto ou XML, de forma que o retorno destes dados à memória do computador, quando o estado normal do sistema estiver restaurado, possa ocorrer, sem que haja perda de informação.

A persistência de dados é, portanto, uma forma de assegurar que os dados utilizados e alterados em um determinado sistema sejam duráveis, através da manutenção dos dados em um meio de armazenamento estável, que possibilite a restauração posterior.

A persistência dos dados é uma necessidade que existe desde os sistemas primordiais da computação. Na evolução dos depósitos de dados, as primeiras gravações eram feitas em discos e fitas magnéticas, utilizando-se de arquivos de texto puro. Quando havia a necessidade da gravação de um grupo de dados, no qual se davam na maioria dos casos, separavam-se os campos através de algum caracter especial (como vírgula(,) ou ponto-e-vírgula(;), por exemplo), e demarcava-se um novo registro com o caracter especial de quebra de linha. A gravação e recuperação era feita em ordem FIFO, ou seja, os dados eram recuperados na mesma ordem em que foram gravados.

A necessidade crescente de gravar-se cada vez mais informações em disco, para que pudessem ser recuperadas na medida da necessidade, seguida da necessidade da consulta de informações específicas de forma eficiente, sem a necessidade de rotinas complexas de conversões de formatos e inúmeras operações de I/O, trouxe-nos o que hoje é conhecido como sistemas de bancos de dados.

Bancos de dados podem ser definidos como depósitos ou repositórios, que contêm dados, objetivando o armazenamento, a disponibilização de informações (consultas) e a manutenção. Basicamente, um banco de dados é composto por quatro componentes [DAT00]:

- **Hardware:** meio físico, composto por dispositivos de armazenamento primário (memória) e secundário (discos);
- **Software:** meio lógico, composto pelos programas que permitem a manipulação dos dados (as operações CRUD: criação, leitura, atualização e deleção);

- **Dados:** o elemento essencial do banco de dados, que caracteriza-o enquanto um sistema de informação;
- **Usuários:** agentes que interagem com o banco de dados, como programadores, administradores de dados e usuários finais.

Os SGBD (Sistemas Gerenciadores de Bancos de Dados) são sistemas que, em conjunto com os arquivos de dados, fornecem um ambiente conveniente para armazenar e recuperar informações [SKS01]. Alguns representantes desta categoria são o SGBD da Oracle, o DB/2 da IBM, entre outros. Eles passam não somente a se preocupar em gravar os dados em disco, mas também com o que está sendo gravado em disco, e isso trouxe mais uma necessidade a necessidade de se validar os dados gravados no banco de dados.

Além disso, um SGDB deve ser capaz de lidar com acessos simultâneos, através de transações. Os requisitos de um SGBD são descritos pelas propriedades *ACID* [SPE03]:

- **Atomicidade:** As transações devem ser atômicas. Caso uma parte da transação falhe, seja por erro de software ou hardware, toda a transação deve ser cancelada.
- **Consistência:** O banco de dados deve manter-se em um estado consistente, respeitando a integridade referencial.
- **Isolamento:** As transações não devem afetar a execução de outras transações simultâneas, ou seja, as transações devem operar de forma isolada, sem interferirem entre si.
- **Durabilidade:** Refere-se ao próprio conceito de persistência. A garantia da durabilidade dos dados é feita através de *logs* de transação e *backups* do banco de dados.

Ao contrário dos sistemas primários de persistência de dados, um banco de dados que possui estas propriedades pode assegurar não somente que os dados estarão persistidos em disco, mas que serão gravados com valores válidos e coerentes.

Os sistemas de bancos de dados evoluíram de sistemas de rede e hierárquicos aos chamados bancos de dados relacionais.

Bancos de dados relacionais (SGBDR) são baseados no modelo relacional, cujo fundamento encontra-se na álgebra relacional. Este tipo de abordagem permite o desenvolvimento de um modelo lógico dos dados armazenados no banco. Os dados são armazenados em *tuplas*, um conjunto de atributos, constituídos por domínio (tipo de dados) e valor (o dado em si). A definição de um domínio ou combinação de domínios que identifica individualmente cada elemento da relação é chamada de chave primária, enquanto a identificação da relação entre os itens únicos da tabela é dada por uma chave estrangeira.

Dados os conjuntos S_1, S_2, \dots, S_n (conjuntos não necessariamente distintos), R é uma relação destes n conjuntos, se for um conjunto de n -tuplas, onde o primeiro elemento é um elemento do primeiro conjunto (S_1), o segundo elemento do segundo conjunto (S_2), e assim por diante [COD70]. Relações podem ser unárias, binárias, ternárias ou de ordem n .

Estes são os conceitos básicos por trás do modelo relacional, que caracteriza a tecnologia de persistência de dados mais utilizada na atualidade, de maior maturidade, e que apresenta-se enquanto a justificativa da tecnologia de mapeamento objeto-relacional para a persistência de objetos neste tipo de modelo. Estes conceitos, por sua vez, serão abordados a seguir.

1.2. Persistência em um contexto orientado a objetos

Antes da entrada no âmbito da persistência de objetos, apresentam-se alguns conceitos básicos que caracterizam o paradigma da orientação a objetos.

Alguns autores definem a programação orientada a objetos como uma programação de

tipos de dados abstratos (TADs) e seus relacionamentos [MUE97]. Um tipo de dado abstrato é constituído por “Dados”, isto é, a descrição de estruturas dos dados, e “Operações”, que são a sua interface e os métodos de instância. Há dois tipos especiais de operações:

- o “Construtor”, que descreve as ações que serão tomadas a partir da criação da entidade TAD
- o “Destruidor”, que são as ações executadas quando a entidade é removida da memória.

Na orientação a objetos, o conceito de TAD é chamado de “Classe”.

Objetos são abstrações além de simples tipos de dados abstratos. Um objeto é uma instância de uma classe, sendo constituído por atributos (cuja implementação caracteriza uma variável de instância) com um determinado estado (conjunto de valores dos atributos em um determinado momento) e capaz de interagir com mensagens enviadas por outros objetos através da implementação dos métodos de sua classe.

De forma geral, um objeto deve apresentar as seguintes características [FAS97]:

- **Identidade:** este é o fator que distingue um objeto de outro objeto que possua o mesmo valor, ou o mesmo estado. Portanto, ela deve ser independente dos valores que o objeto possui.
- **Estado:** os valores de seus atributos em um determinado momento (variáveis de instância). Objetos podem passar por transições de estados ou manter o mesmo estado em todo seu ciclo de vida.
- **Comportamento:** é uma abstração que permite a interação com o objeto: o conjunto de operações de um objeto (a sua interface), as respostas destas operações e as

mudanças que estas operações podem causar ao objeto e a outros objetos do sistema. Toda interação com um objeto deve ocorrer através de sua interface.

- **Encapsulamento:** a forma de ocultar-se os detalhes de implementação de um objeto de outros objetos. Os atributos dos objetos geralmente são privados, cujo acesso apenas ocorre através dos métodos públicos do mesmo.

O modelo de objetos da OMT (*Object Modelling Technique* [RUM90], precursora da metodologia UML), representa a estrutura estática dos objetos de um sistema, apresentando os fatores que os caracterizam (identidade, estado, e operações), além dos relacionamentos entre os objetos, descritos pelas associações (ligações potenciais), agregações (relacionamento parte-todo) e generalizações (herança) [RIC96].

Um objeto pode ser transiente ou persistente. Um objeto transiente existe apenas na memória volátil durante a execução do programa, sendo limitado ao tempo de vida do objeto que o instanciou. Quando o programa termina, o objeto deixa de existir.

Persistência, no contexto O.O., é a capacidade de um objeto de manter seu estado além do ciclo de vida da aplicação. Normalmente, isto significa que o objeto deve ser gravado em um meio de armazenamento estável, para que possa ser recriado em um momento futuro. A persistência possibilita que o objeto seja acessível mesmo após ter sido removido da memória (Fig. 1).

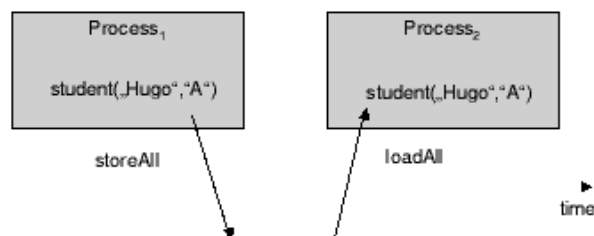


Fig. 1: Persistência de objetos [extraído de WOL03].

Dado o conceito de objeto enquanto elemento fundamental de dados da orientação a objetos, conclui-se que o problema da persistência não se trata da persistência dos dados diretamente, mas sim da necessidade de desenvolvimento de mecanismos que possibilitem a persistência dos objetos, ou de todo o grafo de objetos¹, em seus estados originais.

Algumas das tarefas envolvidas para a persistência de objetos incluem a necessidade da gravação dos valores das variáveis de instância de um objeto, da conversão dos ponteiros de objetos em memória (as referências aos objetos) em identificadores permanentes e únicos (*swizzling*), e da gravação dos métodos da classe do objeto (um recurso raramente implementado pelos mecanismos de persistência), de forma permanente.

Swizzling é o processo de transformação de identificadores permanentes do objeto (*Object ID*, ou simplesmente, *OID*) em endereços de memória [KRO99]. Na maioria das linguagens O.O., ponteiros são uma forma de endereçamento de memória, com validade limitada à execução do programa. Ao armazenar um objeto, os ponteiros de memória podem ser associados a um identificador permanente único, válido em todo o ciclo de vida do objeto, estando em memória ou não. Este é o método utilizado pelos SGBDOO para manterem referências únicas a respeito de cada objeto armazenado.

Na terminologia relacional, o identificador de objetos é chamado “chave”. A determinação de OIDs simplifica a estratégia de chaves na utilização de um SGBDR, e possibilita a replicação de objetos. Um OID não deve ser associado a nenhuma regra de negócio, e deve ser único na hierarquia de classes, e preferencialmente, entre todas as classes [AMB98]. O objeto deve manter sua identidade mesmo se houver alteração de todos seus métodos e variáveis de instância [SKS01].

A estratégia de geração do OID não deve ser proprietária (levando-se em conta a portabilidade, confiabilidade e manutenção da aplicação) e não deve ser tarefa do mecanismo de persistência (do SGBD, por exemplo).

¹ Ou hierarquia de composição [SKS01], refere-se a objetos que são compostos de outros objetos (complexos).

Há diversas opções para se efetuar a persistência de objetos, mas muitas delas ainda são tecnologias bastante recentes, sem histórico de sucesso e/ou de baixo índice de adoção, como os bancos de dados O.O. e os bancos de dados baseados em XML.

Atualmente, a tecnologia mais utilizada para este fim para aplicações corporativas ainda são os sistemas gerenciadores de bancos de dados relacionais (SGBDR). Uma forma eficiente de se utilizar SGBDR com orientação a objetos é o método de MOR, que será apresentado com mais detalhes nos capítulos seguintes.

1.3. O descompasso objeto-relacional

Grande parte do desenvolvimento de aplicações de negócio modernas utiliza a tecnologia O.O., mas mantém como principal meio de persistência os bancos de dados relacionais.

Enquanto o paradigma da orientação a objetos é baseado em princípios provados pela engenharia de software, o paradigma relacional é baseado em princípios matemáticos, especificamente em teoremas da álgebra relacional.

Por consequência das diferenças entre os dois modelos, há uma dificuldade do ponto de vista técnico para a implementação rápida e descomplicada de aplicações que envolvam programação em linguagem O.O. e o armazenamento de informações em um SGBD relacional. Este é um problema tão recorrente que já recebeu um nome: "*Impedance Mismatch*" [AMB02].

Como foi visto anteriormente, os dados em uma abordagem O.O. são representados por objetos, incluindo os valores dos atributos e as operações do objeto.

Há uma diferença de representação: os objetos são referenciados em memória, enquanto os dados são referenciados por chaves em tabelas, que existem fisicamente. Além disso, os objetos possuem campos multivalorados, ao contrário do modelo relacional, onde os dados devem ser atômicos.

O modelo relacional apresenta uma forma bem distinta de representação de dados. No modelo relacional, os dados são representados por *tuplas*, ou registros de uma tabela, e não apresentam comportamento associado a eles.

A linguagem SQL é o padrão adotado pelos SGBDR para operações de CRUD (recuperação e atualização de dados), permitindo que tabelas sejam combinadas para expressar o relacionamento entre os dados.

Pode-se colocar o problema da seguinte forma: com a diferença entre as abordagens, o programador deverá programar em duas linguagens diferentes. A lógica da aplicação é implementada utilizando uma linguagem orientada a objetos, enquanto utiliza-se a SQL para criar e manipular dados no banco de dados. Quando os dados são recuperados do banco de dados relacional eles devem ser traduzidos para a representação específica da linguagem O.O.

Os bancos relacionais apresentam algumas limitações em termos de modelagem. Em SGBDR, não há o conceito de instância, comportamento e herança. Também há questões de performance, quando muitas junções são necessárias para representar um objeto, além do próprio descompasso semântico em relação às linguagens O.O.: há um conjunto limitado de tipos de dados no banco relacional, e um número infinito de classes em potencial. Estes, entre outros fatores, tem como consequência o descompasso entre os modelos.

Se a arquitetura do sistema não está alinhada ao modelo de dados, a produtividade é afetada, em virtude da necessidade de dispô-la em camadas e de se criar uma interface entre elas, o que não ocorreria se o mesmo modelo de dados fosse adotado por ambas aplicações.

Alguns autores afirmam que a diferença cultural entre os programadores O.O. e os DBAs é mais relevante que a questão técnica, já que as preocupações de modelagem possuem a mesma essência [AMB03]. Um dos argumentos é de que qualquer modelo relacional pode ser modelado por objetos, e de que, com um projeto adequado e uma estruturação cuidadosa do sistema, tudo pode ser encapsulado [MUL99].

Entretanto, a realidade de muitos sistemas é diferente: o modelo de dados já se encontra em produção há um período de tempo considerável, e as empresas que dependem do mesmo dificilmente se disporiam a alterá-lo para atender aos novos paradigmas de desenvolvimento, devido a questões de investimento e risco, entre outras.

Uma proposta de integração entre os dois modelos é o objetivo da técnica de Mapeamento Objeto/Relacional (MOR). Os mecanismos de MOR trabalham para transformar uma representação de dados em outra, através de técnicas de tradução entre os diferentes esquemas de dados, para, enfim, lidar com este descompasso em situações onde a utilização de um banco de dados orientado a objetos não é uma opção.

CAPÍTULO 2 - Mecanismos de persistência de objetos

No capítulo anterior, foram apresentados os conceitos básicos da metodologia de orientação a objetos e a importância da persistência dos objetos. Procurou-se demonstrar o problema do descompasso de impedância entre a utilização de uma linguagem estruturada, como a SQL, do lado do banco de dados, e de uma linguagem orientada a objetos para acesso e processamento dos dados armazenados em um banco relacional. Enfim, a importância de se alinhar a arquitetura do sistema com a arquitetura dos dados.

Existem diversas tecnologias que permitem a persistência de objetos. Neste capítulo, serão apresentadas as principais delas, iniciando pelo método de serialização de objetos, em seguida pelo armazenamento em arquivos XML, passando pelas técnicas de persistência de objetos implementadas pelos SGBDOR e SGBDOO atuais e as principais bibliotecas de conectividade (JDBC/ODBC) entre a interface O.O. e o banco em questão.

2.1. Serialização de objetos

Quando trabalha-se com orientação a objetos e necessita-se da persistência de certos objetos de um determinado sistema, depara-se com o seguinte problema: como recuperar o objeto persistido de forma a reconstituí-lo em seu estado na memória?

Para exemplificar, considere um simples aplicativo de agenda telefônica. Um modelo possível de objeto para este sistema está representado abaixo (Fig. 2):

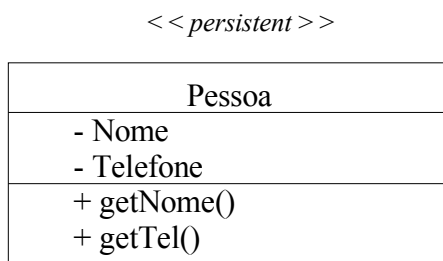


Fig. 2: Exemplo de Classe Pessoa.

A forma mais simples de se realizar a persistência destes dados seria através da gravação dos mesmos diretamente em disco, em um formato de arquivo sequencial (“*flat-file*”). Assim, é preciso levar em consideração o formato em que será armazenado cada atributo do objeto, para que seja possível recuperá-lo depois. Dependendo do grau de complexidade do objeto (como referências a outros objetos e tipos de dados complexos), muito esforço será necessário tanto na gravação quanto na reconstituição do objeto persistido desta forma.

Para evitar este esforço de desenvolvimento de funções específicas de persistência e restituição de objetos, as linguagens orientadas a objeto disponibilizam bibliotecas de *serialização de objetos* (como é o caso de Java, C++, etc).

O método de serialização, ou linearização, essencialmente transforma um objeto (ou o grafo do objeto que compõe seu estado) em um fluxo de dados (mais especificamente, um *stream* de *bytes*), e grava este fluxo em disco. Para recuperar este objeto, efetua-se a operação inversa. As classes de serialização se encarregam em reconstituir o objeto automaticamente no mesmo estado em que foi armazenado, incluindo o estado de objetos nele contidos ou por ele referenciados [QUA00].

Em Java, um objeto deve implementar a interface `java.io.Serializable` para poder ser armazenado desta forma [ECK02]. Esta interface não possui métodos ou campos a serem implementados, pois seu propósito é apenas identificar os objetos que podem ter seus estados “serializados” ou “desserializados”. A maioria das classes da biblioteca Java implementam esta interface diretamente ou por herança. Campos estáticos ou transientes não são serializados automaticamente, por não constituírem parte da instância.

Para cada definição de classe é calculado um “número de série”, ou número de versão, que identifica cada objeto para a restauração posterior [FLA97].

Esta estratégia de persistência é a forma mais simples e direta de persistência de objetos. Entretanto, a simples serialização não permite operações mais complexas como a alteração dos

objetos da aplicação. Se o objeto receber mais um atributo, como o número do celular por exemplo, a leitura do objeto serializado retornará erros, pois seu “número de série” será alterado.

Além disso, a gravação de objetos em arquivos sequenciais (*flat-file*) causam um detrimento na performance, devido à necessidade de acessos constantes ao disco. Eventuais falhas no processo podem resultar em um arquivo corrompido, e portanto, na perda dos dados. Por último, uma pesquisa nos objetos armazenados desta forma só é possível após a reconstituição em memória de todos os objetos serializados, o que pode ser altamente ineficiente, considerando um volume grande de registros.

2.2. Persistência em XML

A persistência de dados em XML é um recurso cada vez mais utilizado no contexto de aplicações Web, devido ao fato do formato XML estar se tornando rapidamente um padrão de documentos para a troca de dados na Internet.

A linguagem XML (*Extended Markup Language*) é derivada da linguagem de marcação de textos SGML (*Standard Generalized Markup Language*), que também originou a linguagem de marcação HTML (*Hypertext Markup Language*) [W3C04].

XML nada mais é que uma abordagem padronizada para a descrição de dados. O padrão XML tem como propósito ser uma forma simples de identificação de dados, que pode ser interpretada tanto por pessoas, devido à flexibilidade de seus atributos, quanto por máquinas, devido à estruturação lógica de seus componentes.

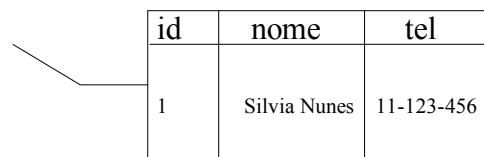
A linguagem XML pressupõe dois conceitos básicos: a utilização de estruturas de marcação (*tags*) e o aninhamento correto destas estruturas de forma hierárquica (*well-formedness*).

Considere a classe utilizada no exemplo anterior. Uma forma possível de representá-la

em um formato XML seria:

```
<agenda>
  < Pessoa id="1">
    < nome>Silvia Nunes</ nome>
    < tel>11-123-456</ tel>
  </ Pessoa>
</ agenda>
```

O mapeamento entre o XML e um objeto, em uma linguagem orientada a objetos, é um processo relativamente simples. O elemento raiz, no caso Agenda, representa o nome da classe principal, enquanto os elementos filhos representam os objetos Pessoa com seus respectivos atributos, Nome e Telefone. Seguindo basicamente o mesmo princípio, seria possível mapear o mesmo XML em um modelo relacional [WIL00], com está exemplificado na Fig. 3.



id	nome	tel
1	Silvia Nunes	11-123-456

Fig 3: Representação dos dados em tabela.

Além do documento XML, muitas vezes utiliza-se também um arquivo “descritor de tipos” DTD (*Document Type Definition*), que inclui a definição dos tipos de dados de cada elemento (texto, numérico, etc.), ou, em implementações mais recentes, um arquivo “*descritor de esquema*”, o chamado XML Schema, que utiliza o próprio formato XML para a definição dos tipos. Estes arquivos são necessários para o mapeamento dos atributos em bancos de dados tradicionais, como será visto adiante.

A persistência de objetos em formato XML segue o mesmo princípio da serialização: uma classe se encarrega em converter os dados para um arquivo com formatação XML (ao invés de um simples fluxo de dados) e estes dados são em seguida gravados em um arquivo no disco. Para a recuperação do objeto, a classe deve fornecer mecanismos de mapeamento entre os

atributos do objeto no estado em que foi armazenado para um objeto utilizável na linguagem de programação.

As vantagens do uso do formato XML na persistência dos objetos, além da simplicidade inerente desta tecnologia, está no fato deste formato basear-se em um padrão bem definido de arquivos de texto puro, com adoção cada vez maior da indústria de modo geral.

Outra vantagem deste tipo de persistência de dados está no fato dos atributos dos objetos não serem mais mantidos em um formato binário de arquivo, mas sim em um formato estruturado que pode ser lido por humanos, o que torna a aplicação muito mais flexível.

Este mecanismo também permite a recuperação de objetos com versões diferentes, possibilitando desta forma a recuperação de objetos, mesmo após terem sido alterados pelo aplicativo (como a adição de atributos, por exemplo), da mesma forma que uma versão anterior do aplicativo também é capaz de recuperar objetos armazenados em um padrão novo.

Os recursos do mecanismo de persistência em XML podem variar de acordo com a biblioteca utilizada para o processamento e persistência utilizando este padrão de documentos nas linguagens orientadas a objeto.

A tecnologia XML engloba aspectos de armazenamento (o próprio documento XML), metadados (esquemas e DTDs), linguagens de consulta, como a *Xquery*, e interfaces de programação, como SAX, DOM e XSLT [WIL00].

Alguns exemplos de bibliotecas relativamente simples de persistência de dados em XML na linguagem Java são: JDOM, cuja integração nas bibliotecas da linguagem está prevista para a versão 1.5; Castor, uma iniciativa do projeto Apache para disponibilizar este recurso às aplicações Java, além de *Xerces* e *Xalan*, também integrantes do projeto Apache [<http://www.apache.org>].

Por outro lado, as desvantagens de métodos de persistência baseados em XML são as mesmas de se utilizar simples arquivos de texto armazenados em disco: a integridade dos

arquivos pode ser afetada se houver falhas no processo de gravação em disco e os acessos ao disco afetam a performance do sistema. Tanto o método de serialização quanto a persistência em XML, não fornecem nenhuma forma de controle de transações e/ou integridade de dados.

Há algumas propostas de bancos de dados XML para minimizar estas desvantagens. Elas dividem-se basicamente entre o suporte dos SGBDR atuais ao formato, e os chamados bancos de dados XML nativos (ou XND).

Alguns sistemas SGBDR mais recentes já apresentam a funcionalidade de importação e exportação de dados no formato XML [WIL00]. Geralmente os dados de um documento XML são mapeados no banco de dados relacional através do relacionamento de um determinado elemento com uma coluna específica de uma tabela. Este método pode não garantir que o documento XML retornará exatamente da mesma forma como foi armazenado previamente. Alguns bancos simplesmente utilizam objetos *BLOB* [*Binary Large Objects*] para o armazenamento de XML. Ressalta-se também que a abordagem dos SGBD tradicionais para o padrão XML é altamente dependente das extensões proprietárias de cada fabricante.

O banco de dados DB/2 da IBM, por exemplo, utiliza um esquema XML chamado DAD (*Document Access Definition*) que define como cada atributo XML será mapeado no banco de dados, incluindo informações de tabela, coluna e, opcionalmente, a definição do relacionamento entre tabelas (por uma chave primária ou estrangeira) relativas a cada elemento. O conceito de mapeamento de dados em bancos relacionais será visto com mais detalhes nos capítulos seguintes.

Já a tecnologia de bancos de dados XML nativos, ou XND, por sua vez, tem como principal propósito trabalhar com o formato XML tanto no armazenamento quanto na recuperação dos dados, sem a necessidade de um mapeamento posterior para uma outra estrutura de dados.

Um XND define um modelo lógico para armazenamento e recuperação de documentos

XML. O documento XML consiste em sua unidade lógica fundamental de armazenamento, da mesma forma que uma *tupla* de uma tabela é a unidade fundamental de armazenamento em bancos de dados relacionais. Esse tipo de sistema não apresenta um modelo físico de armazenamento em particular, podendo utilizar um banco orientado a objetos, relacional, ou mesmo um banco de dados hierárquico. Em geral, utiliza-se um formato proprietário de armazenamento.

A aplicação deste tipo de solução é adequada para dados centrados em documentos, semi-estruturados, que possuem uma estrutura complexa com diversos aninhamentos como capítulos de um livro, informações médicas, entre outros – ao contrário de aplicações centradas em dados, como ordem de vendas, dados de consumidores, dentre outras.

Alguns exemplos de bancos XML nativos existentes: Tamino¹, Yggdrasill², Excelon³. Outro tipo de aplicação que se enquadra nesta categoria são os sistemas de portal de gerenciamento de conteúdo, que utilizam XML como meio de armazenamento de dados.

A tecnologia existente, entretanto, ainda não implementa recursos para garantir a performance e a disponibilidade semelhantes aos recursos existentes nos bancos de dados tradicionais, e as extensões XML que permitem as consultas, controle de concorrência e transações, por exemplo, ainda não se apresentam bem definidas.

2.3. Bancos de Dados Estendidos

Como foi visto anteriormente, novos recursos têm sido integrados aos SGBDR, como o suporte a arquivos XML e a linguagens O.O.. Os principais desenvolvedores de bancos de dados comerciais têm mantido esforços para adaptar seus sistemas às necessidades do paradigma O.O..

Os chamados bancos de dados objeto-relacionais (SGBDOR) nada mais são que bancos

1 Software AG: <http://www.softwareag.com/>

2 Media Fusion: <http://www.mediafusion-usa.com/>

3 eXcelon: <http://www.objectstore.net/>

de dados relacionais estendidos com estas novas funcionalidades.

Os SGBDOR trazem a vantagem da robustez e tradição dos SGBD tradicionais, como o suporte a processamento paralelo, replicação, alta-disponibilidade, segurança e todas as propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade).

Algumas das características dos bancos estendidos incluem a extensão dos tipos de dados básicos (“CREATE TYPE” no Oracle, por exemplo), encapsulamento e herança de tipos, suporte a objetos complexos e coleções, suporte a métodos definidos pelo usuário (*procedures*), regras e *triggers*, além do suporte a novos tipo de dados, incluindo a pesquisa aos mesmos. Enfim, o suporte à linguagem SQL3 (atualmente conhecida como SQL-99) [MUL99].

Desta forma, os SGDBOR permitem o armazenamento de objetos utilizando o modelo relacional, como apresentado na Fig. 4.

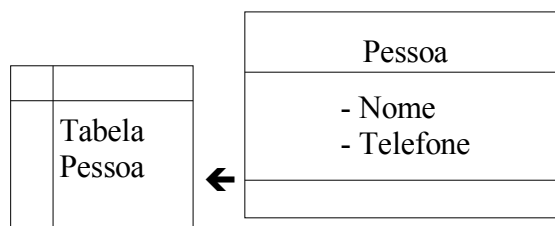


Fig 4: Representação em tabela de uma classe.

A linguagem SQL-99 é a proposta ANSI/ISO para substituir o atual SQL-92. A principal novidade da nova versão é o suporte a estas estruturas OR. A idéia básica é permitir a criação de tipos de dados estruturados pelo usuário (além de tipos abstratos, tipos distintos e *rowtypes*), o que seria a aproximação da orientação a objetos, e de funções definidas também pelo usuário.

Além disso, novos tipos de dados básicos foram incluídos na especificação da linguagem. Além dos tipos CHAR, FLOAT, DATE, etc., novos tipos de dados incluem o suporte a objetos BLOB, que representam dados multimídia, e CLOB (*Character Large*

Objects), entre outros.

Os tipos de objetos ou tipos de dados estruturados definidos pelo usuário permitem o mapeamento do modelo de dados de um objeto diretamente para um esquema de banco de dados objeto-relacional, o que evita a necessidade de reestruturação do modelo de dados em um formato “linha-coluna” de tabelas relacionais, e assim evita-se também a necessidade de inúmeros relacionamentos (“JOINS”) entre tabelas para possibilitar a representação do objeto.

Um tipo estruturado, definido pelo usuário, pode ter os seguintes aspectos [EIM99]:

- A comparação de valores pode ser feita apenas por funções definidas pelo usuário;
- Podem ter um ou mais atributos, que podem ser um dos tipos pré-definidos ou de outros tipos estruturados;
- Todos os aspectos de seu comportamento podem ser descritos por métodos, funções e procedimentos;
- Podem participar de hierarquias de tipo, onde tipos especializados (“sub-tipos”) possuem todos os atributos e rotinas associadas aos tipos mais generalizados (“super-tipos”), e ainda assim podem definir novos atributos e rotinas;
- Seus atributos são encapsulados.

Os objetos complexos englobam referências a outros objetos, coleções (*set*, *bag*, *list* e *arrays*), colunas com tipo definido pelo usuário e tabelas aninhadas. O acesso aos atributos dos tipos definidos pelo usuário pode ser feito tanto através da notação funcional (*objeto* (*atributo*)) quanto pela notação por “ponto” (*objeto.atributo*).

Outra característica do SQL-99 é a inclusão de novos predicados de pesquisa, entre eles o predicado *SIMILAR*, que permite a utilização de expressões regulares e o predicado

DISTINCT, que é similar ao *UNIQUE*, diferindo na forma em que lida com valores nulos.

A tendência atual é de que os sistemas de bancos de dados relacionais passem a adotar estas extensões para trabalharem com novos tipos de dados, como dados geográficos e imagens, além de adaptarem-se aos novos paradigmas de programação.

Um SGBDOR deve permitir a busca, acesso e manipulação de dados complexos utilizando o padrão SQL, mantendo as regras do modelo relacional.

Atualmente, os principais bancos de dados já adotam um modelo objeto-relacional, entre eles o DB/2, da IBM, que trabalham com objetos utilizando o conceito de “extensões relacionais” (*Relational Extenders*), o Informix, com o conceito de “lâminas de dados”, e o Oracle (8x e 9x), da Oracle, com o conceito de “cartuchos”.

Infelizmente, não há uma padronização definida para estes sistemas, apesar de muitas propostas [STO90; DAT95].

Enquanto os desenvolvedores não adotarem um padrão na abordagem objeto-relacional, haverá uma dependência do usuário nas soluções específicas e extensões proprietárias dos distribuidores da aplicação, o que pode prejudicar a portabilidade e escalabilidade da aplicação. Além disso, os SGBDOR atuais não implementam muitas das características descritas acima, ou implementam de forma ainda muito rudimentar.

2.4. Bancos de Dados Orientados a Objetos

Os bancos de dados orientados a objetos adicionam a persistência para linguagens orientadas a objetos de forma transparente, ao contrário dos outros métodos. Eles permitem o acesso aos dados diretamente através de objetos. Sendo assim, não há a necessidade de se reconstituir os objetos pela linguagem de programação. Da mesma forma, elimina-se a necessidade do código adicional para efetuar persistência de objetos ao se utilizar este tipo de armazenamento. Não há descompasso de impedância: o que é armazenado é exatamente o

mesmo que será recuperado posteriormente.

Alguns autores definem os bancos de dados orientados a objetos como bancos de dados de Terceira Geração. Os antigos bancos de dados Hierárquicos e de rede enquadrariam-se na primeira geração, e os bancos relacionais na segunda. Algumas propostas foram feitas para definir o que caracterizaria realmente um banco de dados O.O., ou de terceira geração, e abaixo apresentamos uma das propostas preliminares, mantida como referência sobre o assunto.

Um banco de dados O.O. deve satisfazer basicamente dois critérios [ATK89]:

1) O banco deve caracterizar-se enquanto um SGBD, implementando os seguintes recursos:

- **Persistência:** O objeto deve ser capaz de sobreviver fora dos limites da aplicação que o criou. Este recurso é equivalente à definição de durabilidade nas propriedades ACID.
- **Gerenciamento de memória secundária:** Este recurso inclui o gerenciamento de índices, *clusters* de dados, *buffers* e otimização de consultas (*queries*). Enfim, recursos de performance em geral. Estes fatores devem ser invisíveis ao usuário, isto é, deve haver uma independência entre o modelo físico e lógico do sistema.
- **Concorrência:** Deve permitir o acesso simultâneo aos dados e prover mecanismos (como *locks* de acesso) para que os dados sejam mantidos em um estado íntegro mesmo sendo acessados concorrentemente. Este recurso é o equivalente à definição de atomicidade das propriedades ACID.
- **Tolerância a falhas:** Em caso de falhas de software ou hardware, o sistema deve fornecer mecanismos de recuperação, isto é, de retorno a um estado coerente de dados.

- **Queries “ad-hoc”:** O sistema deve permitir alguma forma de consulta de alto nível à base de dados, efetiva em qualquer SGBDOO (independente de aplicação).

2) O banco deve caracterizar-se enquanto um sistema O.O., apresentando:

- **Suporte a objetos complexos:** Construtores aplicados a tipos simples de objetos representam objetos complexos, e estes devem ser ortogonais (aplicáveis a qualquer objeto). Entre eles estão os construtores de tupla, que representam as propriedades de uma entidade, *set*, *list* e *bag*.
- **Identidade:** Cada objeto deve apresentar um identificador de objeto (OID) único, que deve ser persistente. Dois objetos podem compartilhar componentes, sendo assim atualizações nos componentes devem refletir em ambos.
- **Encapsulamento:** Distinção entre especificação (interface) e implementação (estado). Em geral, o encapsulamento deve envolver os dados de forma a mantê-los apenas acessíveis pelas operações (ou métodos) do objeto.
- **Tipos e classes:** O suporte a mecanismos estruturados de dados, sejam tipos ou classes (dependendo da abordagem da linguagem), é fundamental. Este conceito substitui o esquema dos bancos de dados tradicionais pelo conceito de conjunto de classes ou tipos.
- **Hierarquias de classes ou tipos:** O conceito de herança (simples) é um requisito fundamental de um SGBDOO.
- **Sobrecarga e late binding:** O polimorfismo é implementado através de métodos de sobrecarga e sobrescrita. *Late binding* é necessário para que a linguagem seja capaz de associar, em tempo de execução, a mensagem enviada a um objeto a seu respectivo método.

- **Extensibilidade:** A linguagem deve apresentar mecanismos de definição de novos tipos pelo usuário, além de tipos pré-definidos.
- **Completeness computacional:** Refere-se à capacidade do sistema de executar qualquer função computável, utilizando-se da linguagem de manipulação de dados do próprio SGBD.

Opcionalmente, um banco O.O. pode implementar os seguintes recursos: herança múltipla, verificação de tipos, distribuição de dados, definição de transações pelo usuário e controle de versões [ATK89].

De uma forma geral, as propostas de definição de Bancos de Dados de Terceira Geração concordam com a necessidade da implementação dos aspectos de herança, funções, encapsulamento e tipos estendidos para que um banco de dados possa ser caracterizado como orientado a objetos.

Os SGBDOO armazenam os objetos incluindo seus atributos (dados) e opcionalmente, os métodos de acesso aos dados. O relacionamento nos Bancos de Dados Orientados a Objetos são dados por um conjunto de vínculos entre os objetos, incluindo as multiplicidades padrão um-para-um, um-para-muitos e muitos-para-muitos, por meio de OIDs internos. Os bancos de dados Orientados a Objetos em geral baseiam-se nas coleções e iteradores para operarem com relacionamentos entre objetos.

As operações de relacionamento de objetos, portanto, operam sobre coleções. “Set” é a representação típica de coleções de objetos: uma coleção não-ordenada de objetos ou literais, onde não se permite duplicações. “Bag” é uma coleção não-ordenada de objetos ou literais que pode conter duplicações. “List” é uma coleção ordenada de objetos ou literais, e “Array” representa uma coleção ordenada de objetos ou literais de tamanho dinâmico, acessíveis por

posição. Finalmente, há o tipo de coleção “Dictionary”, uma sequência não ordenada de pares de associações sem chaves duplicadas [MUL99]. Cada uma destas coleções possui operadores apropriados que atuam no objeto de forma transitiva (percorrendo todo o grafo).

Os bancos de dados orientados a objetos também possuem uma linguagem de consulta: a OQL (*Object Query Language*), proposta pela ODMG (*Object Database Management Group*). Entretanto, ela ainda é pouco utilizada, dando lugar muitas vezes à utilização de iteradores em coleções de objetos.

A ODMG é a responsável pela definição dos padrões dos bancos orientados a objetos, assim como pela definição do modelo de objeto. Entretanto, as implementações variam entre os distribuidores. Outras propostas de padrão da ODMG que ainda não atingiram um grau significativo de aceitação são a ODL (*Object Definition Language*) e a OML (*Object Manipulation Language*) [CAT00].

A performance dos SGBDOO tem recebido constantes melhorias. Em geral, a instanciação de um objeto em consultas no banco apenas gera uma referência ao objeto, que somente é inicializado se houver uma requisição de fato. Isto é possibilitado por técnicas de *lazy loading* e objetos *proxy* [FAS97]. Os bancos de dados O.O. recentes fazem uso de métodos de *caching* de objetos, isto é, objetos são mantidos em memória ao invés de serem lidos diretamente do disco. Desta forma, o SGBDOO otimiza seus recursos, evitando uma carga desnecessária na inicialização de todos os atributos referentes a um objeto em uma consulta à base de dados.

Além da transparência entre a linguagem da aplicação e o banco de dados, e dos diversos recursos disponíveis nos bancos de dados O.O., eles são especialmente adequados para representar dados cuja representação em tabelas é pouco eficiente (como dados geoespaciais e de CAD/CAM) [RAJ02].

A desvantagem deste tipo de sistema está principalmente no fato de ainda constituir-se

em uma tecnologia de adoção recente, com implementações ainda em desenvolvimento, como é o caso da OQL, e a falta de uma padronização entre os distribuidores. Além disso, os sistemas de bancos de dados relacionais são legados na maioria das organizações. Sua substituição por sistemas Orientados a Objetos ainda é uma perspectiva bastante remota.

2.5. Outros mecanismos de persistência de objetos

Como foi apresentado nas seções anteriores, quando se trata de aplicações mais robustas, o método de persistência mais utilizado ainda são os bancos de dados relacionais, enquanto aplicações de pequeno e médio porte, voltadas à *Web*, e aplicações específicas utilizam bancos de dados XML ou bancos de dados O.O.. Métodos primitivos de persistência de objetos, como a utilização da serialização de objetos, também são bastante utilizados em aplicações simples que não necessitam de recursos mais sofisticados como o controle de transações e concorrência.

Um mecanismo de persistência de objetos que tem ganhado uma certa notoriedade no mercado é o mecanismo de persistência em memória [BJW87]. Um exemplo de aplicação desta tecnologia é a arquitetura *Prevayler*¹, que tem como base o conceito de “prevalência”, que refere-se ao fato dos objetos “prevalerem” em memória em todo seu ciclo de vida. Todas as operações que ocorrem sobre estes objetos são gravadas em arquivos de *log*, e em períodos de pouco acesso aos dados, o sistema encarrega-se de gravar *snapshots* dos estados correntes dos objetos. Assim, se há alguma falha no sistema com perda de memória, os objetos têm um meio de retornarem ao seu último estado estável, através da recuperação do *snapshot*, que é feita em conjunto com a leitura e processamento dos comandos armazenados nos arquivos de *log*.

Outro exemplo de aplicação desta tecnologia é o projeto HSQLdb², baseado no projeto *Hypersonic*, desenvolvido para ser usado em plataformas embarcadas (dispositivos portáteis). Esse tipo de banco, apesar de relacional, apresenta a possibilidade de rodar em memória (por

1 Prevayler: <http://www.prevayler.org/>

2 HSQLDB: <http://hsqldb.sourceforge.net/>

exemplo, através da cláusula `CREATE MEMORY TABLE`) ou em disco, e o suporte nativo à linguagem Java, assim como o suporte à linguagem padrão SQL.

Há outras implementações de bancos de dados 100% Java, como o *Cloudscape*³, da IBM e o *JDataStore*⁴, da Borland.

Entretanto, estes sistemas ainda atuam em *nichos* muito específicos de aplicações, sem muita representatividade no contexto de aplicações empresariais de médio a grande porte.

2.6. Conectividade entre o SGBD e aplicações O.O.

A presença de mercado dos bancos de dados relacionais ainda é predominante, porém, o modelo de objetos tende a ser utilizado cada vez mais pelas aplicações de interface de negócios. Por este fato, os principais fornecedores de bancos de dados estão evoluindo suas arquiteturas para adaptarem-se aos novos paradigmas de programação.

Muitas soluções de conectividade entre SGBDR e aplicações foram desenvolvidas ao longo dos anos, entre elas as tecnologias ODBC e SQL CLI (*Call Level Interface*), para as linguagens que não trabalham com orientação a objetos.

Para linguagens O.O., existem alguns mecanismos de conectividade específicos, destacando-se o SQLJ e o JDBC. O primeiro é a implementação dos principais fornecedores de SGBD para a utilização de SQL diretamente nos programas Java. A vantagem de se utilizar está na portabilidade do código. Ao contrário da linguagem PL/SQL do banco de dados Oracle, por exemplo, a SQLJ permite a reutilização do código em plataformas diferentes de bancos de dados.

JDBC (*Java Database Connectivity*) é a interface de conectividade com banco de dados padrão que acompanha as implementações atuais de desenvolvimento da linguagem Java. Esta interface evoluiu ao longo dos anos deixando de ser apenas uma ponte de conectividade com a

3 Cloudscape: <http://www-306.ibm.com/software/data/cloudscape/>

4 JDataStore: <http://www.borland.com/jdatastore/>

tecnologia ODBC, para tornar-se uma tecnologia independente, com um rico conjunto de recursos de conectividade disponíveis, e atualmente é parte integral da tecnologia J2EE (*Java 2 Enterprise Edition*). Através das classes fornecidas pela API é possível efetuar qualquer operação no banco de dados de dentro de uma aplicação Java.

A utilização do JDBC é relativamente simples. Inicialmente, deve-se instanciar o *driver* específico do banco de dados, utilizando o método *forName*:

```
Class.forName("com.dummy.Driver")
```

Através da reflexão, o objeto referente ao *driver* é instanciado dinamicamente [FLA97]. Após criada a instância do *driver*, o gerenciador de *drivers* JDBC realizará a inicialização e registro para que ele passe a receber as requisições de conexão.

O próximo passo é estabelecer a conexão com o banco, possibilitada através do método *getConnection* da classe gerenciadora *DriverManager* do JDBC. A partir daí, basta utilizar os métodos disponibilizados pela API para executar consultas ou mesmo criar tabelas ou alterar registros no banco de dados relacional.

As novas implementações de JDBC (JDBC 2 e 3) já incluem a possibilidade de conexão através da interface JNDI, isto é, sem a necessidade dos métodos de reflexão e sem o acoplamento gerado pela classe de gerenciamento de *drivers* do JDBC 1. Outros recursos são o suporte a transações distribuídas e *pool* de conexões, utilização de pontos de controle, entre outros [SPE03].

O propósito original da JDBC é facilitar a utilização de bancos de dados na linguagem Java, a partir do encapsulamento do banco de forma que os desenvolvedores tivessem uma interface consistente de acesso ao mesmo, sem a necessidade do conhecimento da camada física entre a aplicação e o modelo de dados.

Outras linguagens orientadas a objeto, como C++, também possuem bibliotecas de

acesso ao banco de dados semelhantes ao SQLJ e JDBC.

Um exemplo de implementação, em C++, de um *middleware* para comunicação com o banco de dados, é a interface OCCI (*Oracle C++ Call Interface*). Esta interface foi desenvolvida para o banco de dados Oracle, e modelada de acordo com a especificação JDBC. Sendo assim, ela implementa construtores de linguagem semelhantes.

A OCCI é uma interface transparente de manipulação de objetos, com tipos definidos pelo usuário, como se fossem instâncias de classe C++. Os relacionamentos entre objetos são implementados por referências (REFs), e utilizados pelos mecanismos de navegação para acesso aos dados objeto-relacionais. Esta biblioteca é específica para bancos de dados Oracle.

Apesar da disponibilidade de bibliotecas orientadas a objeto para a linguagem C++, ainda é comum as aplicações C++ utilizarem ODBC puro para efetuar a conectividade ao SGBD.

CAPÍTULO 3 - Mapeamento Objeto-Relacional

No capítulo anterior, foram abordadas as formas mais comuns de persistência de objetos, através de arquivos serializados, documentos XML ou SGBD (O.O. e O/R), e os principais métodos de conectividade entre bancos de dados e aplicação.

Neste capítulo, será retomada a relevância da utilização de bancos de dados relacionais para a persistência de dados no contexto atual, e então, será apresentado o método de mapeamento objeto-relacional, uma solução elegante para o problema que aparece quando o sistema precisa armazenar de forma permanente os dados gerados por ele em um SGBDR: o descompasso de impedância.

O MOR nada mais é que o ato de conversão de objetos em memória para dados relacionais, e vice-versa. Em geral, pressupõe-se que o modelo de dados já existe, e que temos que adaptar nosso sistema orientado a objetos para trabalhar com este esquema pré-existente. Caso contrário, a utilização de bancos O.O. nativos pode ser mais adequada do que a geração de tabelas baseadas no modelo de objetos, em conjunto com mecanismos de MOR, que somente adicionariam uma camada de complexidade desnecessária à aplicação.

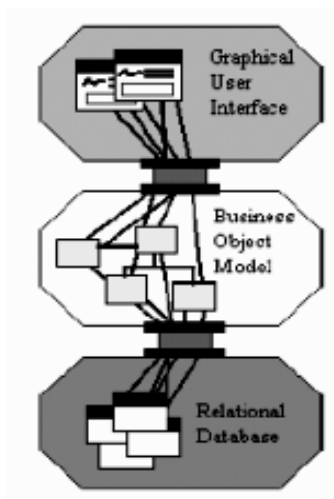
O objetivo deste capítulo é, em suma, apresentar os diversos mecanismos de MOR, a aplicação da camada de persistência e apresentar também as principais ferramentas existentes com este propósito.

3.1. Persistência em SGBDR

Como foi visto, a forma de persistência de dados predominante atualmente ainda baseia-se na utilização de sistemas legados de bancos de dados relacionais. É comum encontrarmos esta arquitetura já existente, e em produção, tanto em sistemas de médio porte quanto em sistemas com alto volume de dados. Bancos de dados relacionais provaram ao longo das décadas que são uma forma estável, segura e eficiente de armazenamento de dados.

Ressalta-se também o fato de que muito investimento foi e ainda é empregado pelas empresas na manutenção dos sistemas de bancos de dados relacionais, pois ele é tido como a base das operações que envolvem a tecnologia da informação. Isto sem contar os gastos com treinamento e equipe e de licenciamento deste tipo de sistema investidos ao longo do processo.

Os SGBDR são considerados sistemas críticos na maioria das organizações, enquanto o modelo de objetos é adotado no desenvolvimento das aplicações de negócio. Portanto, reforça-se a necessidade da utilização de metodologias que permitam que haja uma integração efetiva entre as interfaces, que tendem a ser orientadas a objetos, e os dados, armazenados no modelo relacional. Uma arquitetura de sistemas utilizando ambos os modelos é apresentada na Fig. 5:



*Fig. 5: Arquitetura de 3 camadas.
[extraído de IBM00]*

Muitas vezes, mecanismos simples de persistência, como a serialização de objetos, torna o sistema incapaz de lidar com uma grande quantidade de dados. Uma solução apropriada para este problema seria persistir as informações em um SGBDR, por ser um sistema robusto, geralmente apresentando suporte a alto volume de dados, controle de concorrência, transações e otimizações. Adotando-se esta solução, elimina-se muitos problemas e preocupações como

segurança, rapidez na manipulação das informações e distribuição de dados.

O modelo relacional envolve a existência de tabelas uni-dimensionais de armazenamento de dados, onde cada linha ou tupla representa um determinado registro no banco de dados. É uma abordagem simples e eficiente de persistência, entretanto, retorna-se ao problema inicial da impedância entre a linguagem relacional e a linguagem O.O..

Objetos incluem estruturas de dados como listas e mapas, e utilizam herança e ligações diretas entre os objetos para se relacionarem entre si. Além disso, os objetos são criados e modificados em memória, sendo necessário coordenar o que ocorre na memória com o que ocorre em disco [FOW02].

De uma forma geral, na persistência de objetos em SGBD, a identidade dos objetos, seus relacionamentos (herança, agregação e associação) e seu estado devem ser preservados nas tabelas relacionais.

O esforço gasto no processo de persistência manual de objetos no banco de dados acaba trazendo não apenas uma camada de complexidade a mais, mas também resulta em uma inconsistência com o modelo O.O.. A alternativa de se acoplar a linguagem SQL no código O.O. acaba por descaracterizar o segundo modelo, pois desta forma, o programa perde as características básicas do modelo O.O.: a possibilidade de reutilização e a facilidade de manutenção do código.

A criação de uma camada de persistência de objetos permite diminuir o acoplamento entre o banco de dados e a aplicação. Desta forma, uma mudança em um modelo pode ser traduzida no outro, sem que haja a necessidade de se reestruturar toda a aplicação [AMB03] Logo, através da camada de MOR, pequenas mudanças no esquema relacional deixam de afetar o código orientado a objeto como um todo, e assim, evita-se a necessidade do desenvolvedor da aplicação de atuar também como um administrador de dados, a partir do momento que o conhecimento do esquema do banco deixa de ser fundamental.

Através de MOR, é possível persistir os objetos em um SGBDR, obtendo-se todas as vantagens trazidas por ele, e evitando-se os problemas que aparecem ao realizar-se a persistência em O.O. utilizando um modelo relacional (impedância de modelos).

3.2. Conceitos básicos de MOR

O mapeamento pode ser visto como um processo semelhante ao processo de desenvolvimento de um compilador. A maioria dos compiladores de linguagens de programação convertem um determinado código-fonte em um programa real através de três operações básicas. Primeiro efetua-se a análise léxica do código-fonte, separando as palavras-chave e os *tokens* da linguagem de forma compreensível pelo compilador. A seguir, é feita a análise sintática, que identifica construtores válidos de linguagem nos grupos de *tokens*. Finalmente, o gerador de código interpreta estes construtores e gera o código executável.

O processo de mapeamento de um objeto, por exemplo, definido em uma estrutura de documento XML, para uma tabela do banco de dados relacional, segue o mesmo princípio. Inicialmente, analisa-se o código para reconhecer o que caracteriza um atributo e seus respectivos valores. Verifica-se então se o documento é válido, isto é, se o documento está bem formado. Verifica-se também se o documento segue a estrutura definida em um esquema ou DTD associado a ele. Após estas análises, é possível extrair os dados do documento e determinar a melhor forma de adaptá-los em uma estrutura relacional.

O mapeamento objeto-relacional (MOR) é uma técnica de tradução entre o esquema relacional e o modelo de objetos, via mapeamento dos estados de um objeto no modelo relacional de armazenamento (SGDBR). Desta forma, adiciona-se uma forma consistente de se trabalhar com SGDBR em linguagens O.O..

O mapeamento realizado entre os objetos de uma linguagem O.O. e as tabelas de um SGBDR torna possível a persistência de objetos em um SGBDR de modo transparente.

Persistência transparente refere-se à habilidade de se manipular os dados armazenados em um SGBDR diretamente pela linguagem O.O., e é o objetivo do MOR. Isso retoma aos seguintes aspectos desejáveis em um sistema de persistência de objetos [QUA00]:

- **Persistência ortogonal:** a persistência deve ser válida para todos os objetos do sistema, independente de seu tipo;
- **Persistência transitiva (PBR):** se um objeto é persistente, então todos os objetos referenciados por esse objeto devem ser promovidos a objetos persistentes.

As técnicas de mapeamento têm como fundamento padrões de projetos (*design patterns*), que procuram encontrar soluções para diferentes problemas, modelos de dados e de objetos. Também são utilizadas, no processo de mapeamento, técnicas de *cache* para a obtenção de melhor performance em relação aos métodos tradicionais de persistência em SGBDR (SQL com JDBC ou ODBC).

Apesar de muitas linguagens apresentarem bibliotecas para realizarem a tarefa da persistência dos objetos, como por exemplo, a API de serialização de objetos, este tipo de solução muitas vezes é inadequada para sistemas de maior complexidade, onde é esperado que o mecanismo de persistência seja consideravelmente mais robusto e poderoso.

Os *frameworks* de MOR trabalham independentemente da aplicação e do banco de dados. A camada de persistência deve intermediar a camada de dados e a camada de aplicação.

Esse tipo de solução tem como foco aplicações que necessitam acessar dados legados, bases heterogêneas, ou gerenciar objetos de negócio distribuídos e persistentes. Um *framework* de MOR deve apresentar recursos de consulta de dados, suporte a transações, concorrência (*locking* otimista ou pessimista [AMB03]), e enfim, possibilitar a persistência transparente, encapsulando o acesso ao banco de dados relacional.

3.3. Tipos comuns de MOR

Para permitir a persistência de objetos em um SGBDR, algum acordo deve ser feito quanto à forma como os dados serão armazenados. O mapeamento deve envolver cada elemento de um objeto: seus atributos, relacionamentos e sua herança. Logo, os conceitos da programação O.O. devem ser mapeados para estruturas de tabelas relacionais (agregação, associação, herança, polimorfismo) [KEL97].

Isso traz algumas questões, que devem ser consideradas no mapeamento entre os modelos e na escolha de um *framework* de MOR, entre elas [JOH03]:

- Como converter os valores das colunas em objetos e apresentá-los no resultado das consultas?
- Como atualizar os dados caso o estado de um objeto mapeado seja alterado?
- Como modelar os relacionamentos entre os objetos?
- Como modelar a herança dos objetos no modelo relacional?
- Qual estratégia de *caching* pode ser utilizada para minimizar os acessos ao SGBD?
- Como executar funções agregadas?

A principal tarefa do mapeamento O/R envolve a identificação das construções da orientação a objetos que se deseja extrair do esquema relacional, entre elas a identificação das classes, dos relacionamentos que podem ser representados no modelo de objetos e estabelecer as cardinalidades [LER99].

Como foi visto, o processo de mapeamento busca, basicamente, a tradução entre os modelos O.O. e relacional, partindo do princípio de que há uma arquitetura comum entre ambos. As técnicas de MOR, em geral, lidam com esquemas relacionais na 3FN (terceira forma normal) [QUA00]. Como as técnicas de MOR foram desenvolvidas para trabalhar com modelos entidade-relacionamento pré-existentes, a tarefa de tradução, ou mapeamento, consistirá na

adaptação do modelo de objetos ao modelo de dados.

As principais técnicas de mapeamento de objetos em SGBDR podem ser descritas como:

- **Mapeamento classe - tabela:** Mapeamento de uma classe em uma ou mais tabelas, ou de uma tabela para uma ou mais classes, e mapeamento de herança;
- **Mapeamento atributo - coluna:** Mapeamento de tipos em atributos;
- **Mapeamento relacionamento - chave estrangeira:** Mapeamento dos relacionamentos O.O. em relacionamentos entre tabelas.

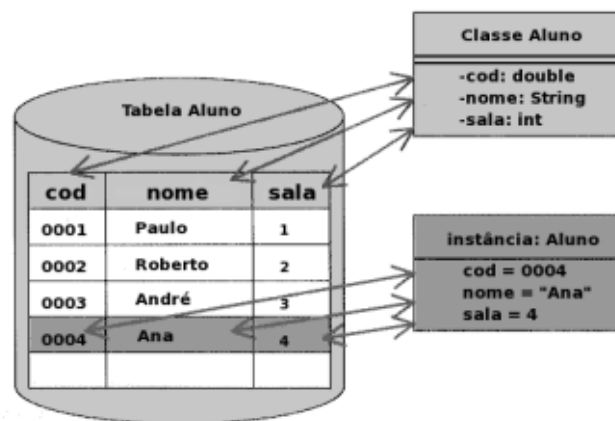


Fig. 6: Exemplo de MOR (adaptação de [RAJ02; fig. 5.2]).

Para modelos de dados muito simples, classes podem ser mapeadas diretamente em tabelas, em uma relação um-para-um (Fig. 6). Esta é a forma mais intuitiva de mapeamento classe-tabela, onde todos os atributos da classe persistente são representados por todas as colunas de uma tabela no modelo relacional. Assim, cada instância do objeto pode ser armazenada em uma tupla da tabela. Porém, este tipo de modelo pode conflitar com o modelo entidade-relacionamento existente, que pressupõe a normalização das tabelas e a otimização de consultas.

Uma estratégia mais realista de mapeamento classe-tabela divide-as em duas categorias

[OBJ03]:

- **Mapeamento de *subset***, onde os atributos da classe persistente representam algumas ou todas colunas de uma tabela. Esta estratégia convém para casos onde todos os atributos de uma classe persistente são mapeados a uma mesma tabela, e onde não há preocupação de incluir as colunas que não fazem parte do modelo de negócios. Pode referir-se também à herança de tabelas simples.
- **Mapeamento de *superset***, onde os atributos da classe persistente são derivados de colunas de múltiplas tabelas. Este tipo de mapeamento é usado para criar "classes de visão", que ocultam o modelo físico de dados, ou para mapear uma árvore de herança de classes utilizando o mapeamento vertical.

Os atributos de uma classe, por sua vez, podem ser mapeados para zero ou mais colunas de uma tabela de um SGBDR, pois os atributos de um objeto não são necessariamente persistentes. Caso um atributo seja um objeto por si só, o mapeamento pode ser feito para várias colunas da tabela. Os atributos podem ser caracterizados em [IBM00]:

- **Atributos Primitivos:** Este termo denota um atributo de uma classe que é mapeado a uma coluna de uma tabela, isto é, refere-se ao valor de um tipo de dados específico (*int, float, double*, dentre outros).
- **Atributos de Referência:** Atributos de referência representam relacionamentos com outras classes, isto é, referem-se a atributos cujo tipo é uma referência a outro objeto ou conjunto de objetos (composição).

Em um modelo orientado a objetos, uma classe pode se relacionar com outra através de

agregação ou associação. A cardinalidade (ou multiplicidade) de um relacionamento pode ser [1:1], [1:n], [n:1], [n:m]. Tabelas são relacionadas utilizando-se das mesmas cardinalidades.

Relacionamentos, no modelo de objetos, são implementados com combinações de referências a objetos e operações. Quando a multiplicidade é 1 (0..1 ou 1), o relacionamento é implementado por uma referência a um objeto ou operação de *get/set*. Quando a multiplicidade é N (N, 0..*, 1..*), o relacionamento é implementado através de um atributo de coleção (por exemplo, um *array*), e por operações de manipulação deste *array* [AMB03].

Sendo assim, as relações entre objetos são implementadas explicitamente através de atributos de referência, enquanto as relações entre tabelas são realizadas através de associações de chaves estrangeiras. Um *framework* de MOR pode mapear as relações entre objetos utilizando as chaves estrangeiras das tabelas correspondentes.

O mapeamento de associações preocupa-se, basicamente, com duas categorias de relacionamentos entre objetos [AMB03]. A primeira baseia-se na multiplicidade:

- **Mapeamento 1:1:** O relacionamento 1:1 é implementado através de restrições de chave estrangeira no modelo relacional e é sempre bi-direcional (Fig. 7).

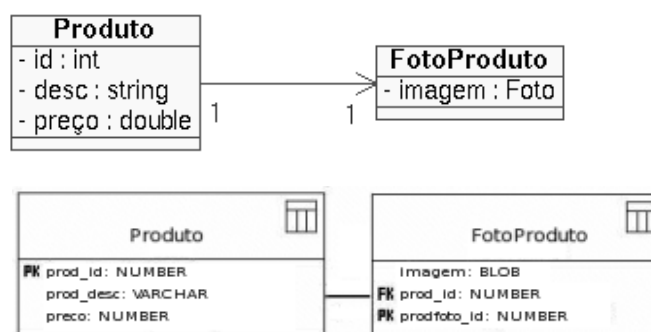


Fig. 7: Relacionamento 1:1 (adaptação de [IBM02]).

- **Mapeamento 1:n:** O relacionamento 1:n (Fig. 8) pode ser implementado de forma similar ao mapeamento 1:1, onde a chave estrangeira é adicionada à classe associativa. Os atributos dos objetos agregados podem ser mapeados a uma única tabela, ou através da criação de uma tabela associativa para o tipo agregado [KEL97].

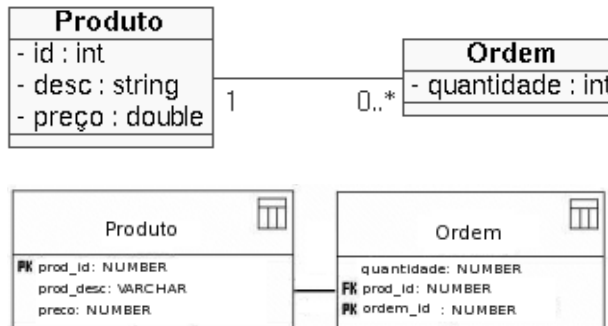


Fig. 8: Relacionamento 1:n (adaptação de [IBM02]).

- **Mapeamento n:m:** Para implementar relacionamentos n:m (muitos-para-muitos), que não existem fisicamente no modelo relacional, utiliza-se uma tabela associativa (Fig 9). O relacionamento passa a ser representado em uma tabela distinta no banco de dados, contendo os OIDs (ou chaves estrangeiras) dos objetos participantes da associação.

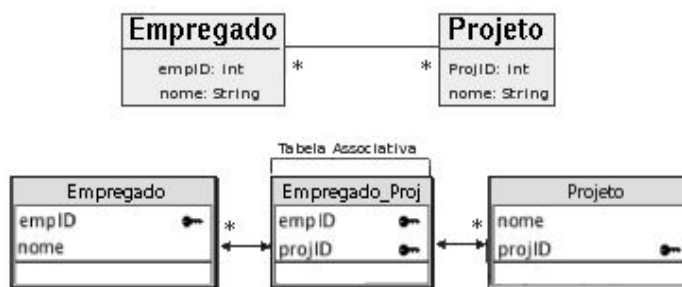


Fig. 9: Relacionamento N:N.

A segunda categoria de relacionamentos baseia-se em dois tipos de “direcionalidade”: relacionamentos unidirecionais e relacionamentos bidirecionais (Fig. 10). O relacionamento unidirecional ocorre quando um objeto relaciona-se a outro, mas este segundo desconhece as classes associadas a ele. O modelo relacional apenas trabalha com relacionamentos bidirecionais, inclusive, este é um fator de descompasso de impedância entre as tecnologias [AMB03].

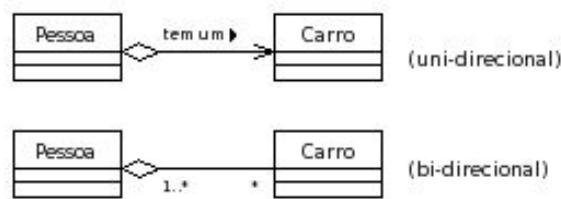


Fig. 10: Direcionalidade.

Outra forma de relacionamento entre objetos são os relacionamentos recursivos (por exemplo, um time pode ser integrante de outros times). O mapeamento em tabelas pode ser feito da mesma forma que um mapeamento n:m, isto é, através da criação de uma tabela associativa. A diferença é que, neste caso, ambas colunas serão chaves estrangeiras da mesma tabela.

Um aspecto essencial da tecnologia O.O. é a herança. A herança permite que dados e comportamentos de uma superclasse (classe base ou classe pai) sejam reaproveitados por subclasses (as classes derivadas da classe pai). Bancos de dados relacionais não possuem o conceito de herança: entidades não podem herdar atributos de outras entidades.

Entre as principais técnicas para representar hierarquias em um esquema relacional estão o mapeamento distribuído de herança (horizontal e vertical) e o mapeamento de filtro de herança, ou de tabela simples. No primeiro, cada subclasse é mapeada em uma tabela separada, e todos os atributos herdados são replicados na tabela. Essa abordagem é eficiente quando a superclasse tem menos atributos que a subclasse. O mapeamento distribuído de herança é difícil

de ser implementado quando coleções heterogêneas de objetos precisam ser recuperadas. [IBM00]. No segundo tipo, as classes são representadas em uma única tabela. Cada registro da tabela utiliza atributos pertinentes à sua subclasse, enquanto os outros atributos são mantidos nulos. Esta é a forma mais rápida de mapeamento, com o custo de manutenção e espaço. Os padrões de projeto referentes ao mapeamento de heranças serão vistos na próxima seção.

Outros fatores a se considerar no mapeamento são a existência de uma restrição a respeito da ordem dos dados - o que envolve o mapeamento de coleções - e o mapeamento dos metadados, geralmente definidos em um documento descritor XML, também frequentemente empregados pelos *frameworks* de MOR.

O mapeamento dos metadados refere-se à criação de um arquivo de detalhamento da forma como as colunas de um SGBDR serão mapeadas em atributos de objetos, incluindo informações a respeito da multiplicidade dos relacionamentos, junções, integridade referencial, etc. Assim, evita-se código repetitivo através da geração de código ou programação reflexiva do *framework* em questão.

No processo de mapeamento também é importante levar em consideração quais informações adicionais deverão ser mantidas pelo objeto, por exemplo, informação de chave primária, de contadores e números de versionamento. Outra informação relevante refere-se à existência do objeto no banco de dados, o que resultaria na decisão entre um comando UPDATE da linguagem SQL, ou de um comando INSERT. Uma técnica para isto seria implementar uma variável *booleana* a cada classe persistente. Essas informações não precisam ser implementadas nos objetos de negócio, mas devem ser tratadas de alguma forma pela aplicação de MOR.

Como foi visto, há mais de uma forma de efetuar-se o mapeamento entre o modelo de objetos e o modelo de dados relacional. A estratégia de encapsulamento do acesso ao banco de dados determinará como será implementado o mapeamento.

3.4. Implementações de Camadas de Persistência

A criação de uma camada de persistência deve permitir o desacoplamento entre o modelo de dados e a aplicação. Numa arquitetura típica de n-camadas, é comum a presença de uma camada para o acesso aos dados que separe os mecanismos de persistência da lógica de negócios.

Como foi visto, há diferenças técnicas entre o modelo persistente do SGBDR e o modelo transiente da programação O.O. que devem ser consideradas, entre elas: [BER02]

- Bancos de dados utilizam identificadores baseados em chaves, enquanto objetos possuem identidade intrínseca (OID);
- A quantidade de objetos em memória tipicamente representa apenas uma pequena parte dos objetos do banco de dados;
- Bancos de dados possuem linguagem de pesquisa que possibilitam consultas eficientes através do uso de índices, mas não conseguem acessar objetos residentes em memória;
- Mudanças em objetos residentes em memória devem ser persistidas de uma forma que não afete a integridade referencial do banco de dados e/ou a performance do mesmo;
- Bancos de dados apresentam o controle de transações (*commit* e *rollback*);
- Bancos de dados apresentam semânticas complexas de *locking* e isolamento de dados;
- Relacionamentos entre objetos são implícitos, e em um banco de dados são bi-direcionais;
- Pode ser desejável utilizar uma estrutura lógica diferente de tabelas relacionais para os objetos residentes em memória .

Para lidar com estas questões de forma eficiente, é preciso interceptar as interações entre a aplicação e os objetos de dados.

O desenvolvimento de uma camada de persistência somente é simples se há um relacionamento linear entre os objetos da aplicação O.O. e as tabelas correspondentes do banco de dados relacional. Mas é comum encontrar objetos ou esquemas de banco que possuam estruturas complexas, e não simplesmente apresentem um relacionamento “um-para-um”. Um DBA, por exemplo, pode decidir isolar a informação referente a cartões de crédito de uma tabela de clientes, e isto implicaria na codificação de rotinas SQL para duas interações separadas, resultando em duas requisições no banco para cada objeto.

As implementações de camadas de persistência, desenvolvidas na linguagem O.O., muitas vezes tratam da geração dos esquemas de dados (mapeamentos) automaticamente e podem até mesmo efetuar uma engenharia reversa criando hierarquia de classes a partir de um esquema de tabelas em banco de dados.

Estas ferramentas têm como propósito facilitar e automatizar o processo de mapeamento entre os modelos, através de diferentes abordagens [BER02]:

- **Ferramentas de mapeamento primitivo sem identificador:**

As ferramentas que se enquadram nesta categoria simplesmente automatizam a criação de pesquisas na base de dados ou atualizações via JDBC. É criada uma classe onde as variáveis de instância correspondem às colunas na tabela, e todas as interações são explícitas, isto é, os comandos DDL apenas são refletidos nas classes e não há preocupação com a definição de uma chave primária correspondente. Desta forma, se duas consultas retornarem a mesma coluna, isso refletirá na criação de dois objetos diferentes correspondentes à mesma tupla relacional.

- **Ferramentas de mapeamento direto com identificador:**

Da mesma forma do mapeamento primitivo, também há uma classe que basicamente reflete as colunas de uma tabela. Entretanto, estas ferramentas mantêm um índice (tabela *hash*) de todos os registros criados por conexão. Assim, as regras de negócio do esquema podem ser escritas independente das aplicações de interface com o usuário.

Esta forma é eficiente em processamento de dados "centrados em documento", com um padrão conhecido, mas não recomendados para transações "*fine-grained*" onde há objetos de quantidade desconhecida sendo constantemente lidos e escritos em resposta a ações dos usuários.

Destacamos as seguintes APIs da linguagem Java que utilizam mapeamento direto, entre outros recursos: *Hibernate*¹, *Castor JDO*² e *OJB*³. A primeira, *Hibernate*, utiliza reflexão (*reflection*) para recuperar informações sobre os objetos e seu respectivo mapeamento com o banco relacional em tempo de execução, o que elimina a necessidade de codificação do mapeamento na camada de persistência, gerando código SQL à medida que for necessário. Desta forma, ele generaliza a camada de persistência, permitindo que eventuais mudanças no esquema relacional afetem minimamente o código da camada de persistência. O *Castor JDO*, apesar do nome, não implementa o modelo Sun JDO (*Java Data Objects*). Sua proposta é fornecer o MOR para diversas estruturas de representação de objetos, como documentos XML, tabelas SQL e diretórios LDAP. Por fim, a ferramenta *OJB* (*Object Relational Bridge*), do projeto Apache-DB, também utiliza a abordagem de reflexão, permitindo que objetos sejam manipulados sem a necessidade da implementação de alguma interface em especial ou da herança de alguma classe específica, através de persistência transitiva.

1 Hibernate: <http://www.hibernate.org/>

2 Castor JDO: <http://www.castor.org/>

3 OJB: <http://db.apache.org/ojb/>

- **Ferramentas de mapeamento “generalizado”:**

Esta alternativa reconhece os objetos persistentes como objetos "especiais", com tratamento diferenciado. As regras de negócio podem ser implementadas através de eventos pré-definidos, e pouco código é necessário para criar a definição de um objeto. As constantes podem ser utilizadas para operações de recuperação e atribuição (“*get/set*”) de valores, como no exemplo:

```
String phone = employee.getString(Employee.PHONE_NR);
```

Ao invés de depender da implementação de métodos específicos, como por exemplo:

```
String phone = employee.getPhoneNr();
```

- **Geradores, *Proxies* e objetos de relacionamento**

Outra abordagem é a geração de código baseado em definições escritas em XML, através de objetos *proxy* representativos dos objetos da aplicação. Através deste tipo de objeto (subclasse), pode-se interceptar o acesso aos objetos de dados e elaborar métodos para lidar com os problemas envolvidos na persistência de acordo. Outro método é a representação de relacionamentos como objetos explícitos, mas assim, adicionam-se objetos desnecessários ao modelo, e apenas o acesso aos relacionamentos são interceptados. Finalmente, pode-se alterar a máquina virtual da linguagem, como a JVM da linguagem Java, para lidar com as questões envolvidas na persistência.

As ferramentas de MOR devem considerar também a utilização de técnicas implícitas ou explícitas de otimização de junções. O mapeamento de junções entre tabelas pode ser feito de diferentes maneiras. Uma forma seria mapear a junção como se fosse uma visão relacional em um único objeto. Outra forma seria preservar dois objetos distintos e separar os campos

referentes a cada um. Mas para isso a junção deve ser do tipo "*Outer Join*", caso contrário, os objetos que não se enquadrarem na junção serão eliminados da memória.

Outro fator a se considerar na implementação de uma ferramenta de MOR são as coleções de objetos. Quando recupera-se um conjunto de objetos com atributos em comum, isto caracterizaria uma coleção de registros em associação com estes respectivos atributos. As ferramentas de MOR devem ser capazes de lidar com este tipo de dados característico da orientação a objetos.

Além das abordagens citadas acima, destacam-se também dois outros tipos de abordagens, adotadas especificamente pelo padrão EJB-CMP e JDO, ambos da Sun Microsystems¹. Eles utilizam o conceito de mapeamento direto, mas lidam de forma diferenciada com as questões de persistência, como o controle de transação e concorrência.

CMP (*Container Managed Persistence*) é parte da especificação EJB (*Enterprise JavaBeans*), que definem componentes distribuídos de três tipos: seção (lógica de negócios), mensagem (reação às mensagens) e entidade (contendo os dados). CMP, portanto, é um tipo especial de EJBs de entidade [SPE03]. Esta tecnologia baseia-se na definição de classes abstratas de acesso para cada variável de instância. As chamadas ao banco de dados são geradas em tempo de instalação (*deploy*), para manter a independência do mecanismo de persistência e esquema, ao contrário da tecnologia precedente, EJB-BMP [THO02].

JDO é a proposta da linguagem Java de padronizar os métodos de MOR. Ela utiliza processamento posterior de "*bytecode*". Isto permite o acesso aos mecanismos internos (*engine*) da ferramenta para assim interceptar o acesso e endereçamento no que diz respeito aos dados das variáveis de instância do objeto.

Enfim, os *frameworks* de MOR tem como propósito reduzir o esforço necessário na codificação de rotinas específicas de mapeamento.

A persistência transparente é, portanto, o principal objetivo das implementações de

¹ Sun Microsystems: <http://www.sun.com/>

MOR. Esta transparência permite a manipulação e navegação entre os objetos persistentes diretamente pela linguagem, como se fossem objetos transientes.

3.5. Padrões de Mapeamento Objeto-Relacional

Padrões de projeto são documentos, elaborados por desenvolvedores, que contém instruções para resolver um problema dentro de um contexto específico. Estes documentos são como uma receita do caminho a seguir para resolver determinados problemas de *design*. Em outras palavras, são modelos conceituais que podem ser aplicados em determinadas áreas de uma aplicação de acordo com a necessidade [SPE03].

Em geral, as ferramentas de MOR são desenvolvidas para trabalhar com um “Modelo de Domínios” (*Domain Model*). Sua base é a criação de um modelo de objetos que incorpora ambos os dados e o comportamento, criando uma rede de objetos interligados. É similar a um modelo de banco de dados, mas envolve dados e processos, atributos multi-valorados e herança [FOW02]. Implementar manualmente uma camada de persistência utilizando este tipo de modelo é uma tarefa de alta complexidade, por isso geralmente opta-se por utilizar ferramentas e *frameworks* de MOR.

Martin Fowler [FOW02], autor de diversos livros a respeito de padrões de projeto, propôs alguns padrões específicos de MOR, utilizados na base de muitos *frameworks*, que serão descritos nesta seção.

Os padrões de MOR dividem-se entre padrões de arquitetura e padrões estruturais. A escolha básica para o padrão da arquitetura está dividida em quatro modelos: *Row Data Gateway*, *Table Data Gateway*, *Active Record* e *Data Mapper*.

Os primeiros são baseados em *gateways*, um objeto que encapsula o acesso a sistemas ou recursos externos (Fig. 11). Em ambos os casos, tem-se em memória as classes referentes às tabelas do seu banco de dados.

Os *gateways* e as tabelas do banco de dados possuem a mesma forma: para cada tabela há uma classe, e para cada campo há uma coluna correspondente no banco de dados. Outra característica do *gateway* é que ele contém todo o código de mapeamento do banco para uma aplicação. O *gateway* é aquele que faz os acessos ao banco de dados, nada mais.

O *Row Data Gateway* é um objeto que age como um *gateway* para um único registro no banco de dados, isto é, um objeto que reflete um registro de uma tabela. O *Table Data Gateway* age sobre uma tabela, mantendo todo o código SQL de acesso, encapsulando a lógica de acesso do banco de dados.

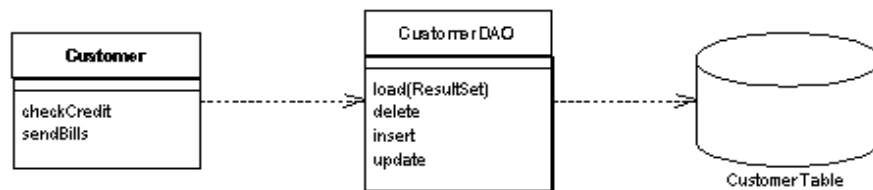


Fig. 11: Gateway Pattern [extraído de FOW02]

O *Active Record*, por sua vez, combina o *gateway* e o objeto de domínio em uma única classe, combinando a lógica de negócio e o acesso ao banco de dados (Fig. 12).

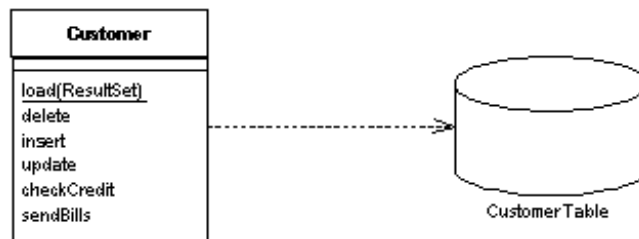


Fig. 12: Active Record Pattern [extraído de FOW02]

Finalmente, há o *Data Mapper* (Fig. 13). Este, por ser mais flexível, é o mais complexo. A grande diferença entre ele e o modelo de *gateway* está na inversão da dependência e do controle.

Com os *gateways* de dados, a lógica de domínio deve conhecer a estrutura do banco de dados, mesmo não lidando com SQL. No *Data Mapper*, o domínio de objetos pode ignorar completamente o *layout* de banco de dados, agindo como um mediador entre os objetos em memória e o banco de dados. Sua responsabilidade é transferir dados entre ambos, e geralmente é utilizado com um “Modelo de Domínios”.

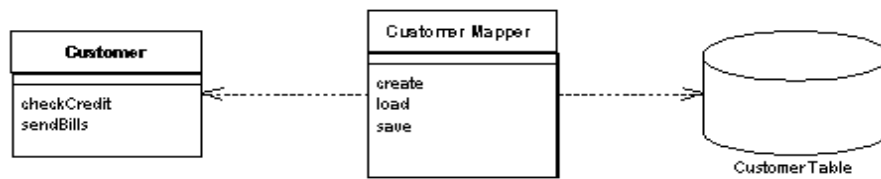


Fig. 13: O Data Mapper isola o objeto de domínio do banco de dados [extraído de FOW02].

Nos projetos de MOR, geralmente o foco está nos aspectos estruturais, abordados anteriormente (cap. 3.3). Os padrões estruturais descrevem como os dados em um banco relacional refletem (mapeiam) em dados em um modelo de objetos. Entretanto, a maior complexidade encontra-se nos aspectos de arquitetura e comportamento.

Há duas questões (impedâncias) envolvidas no processo:

- **diferença de representação:** os objetos são referenciados em tempo real e em memória, enquanto os dados relacionais são referenciados por chaves que estão alocadas em outras tabelas;
- **atributos multivalorados:** os objetos lidam com múltiplas referências em único campo, através de coleções, enquanto não há campos multivalorados no modelo relacional.

Uma forma de lidar com o primeiro problema é através da manutenção da identidade relacional de cada objeto com a adição de um campo identificador (armazenamento da chave

primária nos atributos do objeto).

Para o segundo caso, geralmente utiliza-se uma técnica chamada de *lazy loading* [WOL03], onde o objeto não contém todos os dados: os dados são atribuídos ao objeto apenas quando necessário.

Outra questão a se considerar refere-se ao conceito de herança. Há diversas estratégias de mapeamento de uma estrutura de herança de um modelo de objetos para um modelo relacional, cada uma elaborada para lidar com um problema específico, entre elas:

- **Herança de tabela simples:** Na estratégia de herança de tabela simples (*flat*), uma tabela representa todas as classes da hierarquia (Fig. 14). Essa é uma forma eficiente de mapear classes onde consultas são efetuadas no nível da classe base (superclasse). Entretanto, há uma perda de espaço considerável (valores nulos) e a necessidade de uma coluna (no caso, “type”) para possibilitar a identificação das instâncias correspondentes a cada classe.

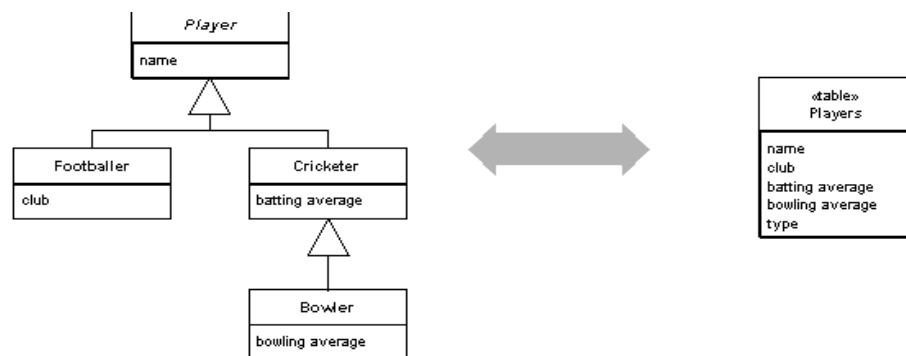


Fig. 14: Herança de Tabela Simples [extraído de FOW02].

- **Herança horizontal:** Na estratégia de herança horizontal, ou herança de tabelas concretas (Fig. 15), cada classe concreta (não-abstrata) é associada à sua respectiva

tabela, incluindo os atributos da classe herdada. A rapidez na persistência e alteração de instâncias é adquirida desta forma pelo custo da desnormalização. Qualquer alteração na classe abstrata deverá ser refletida nas tabelas das subclasses.

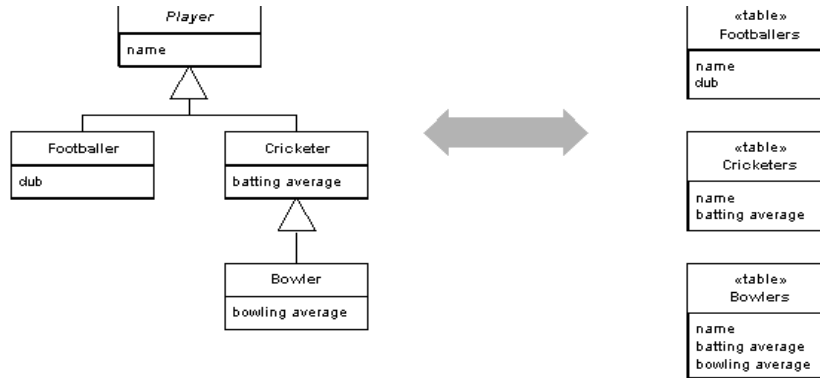


Fig. 15: Herança de Tabelas Concretas [extraído de FOW02]

- **Herança vertical:** Na estratégia de herança vertical, cada classe da hierarquia, inclusive classes abstratas, é associada a uma tabela separada no banco de dados (Fig 16). Através do mapeamento vertical, múltiplas tabelas são acessadas e todos os dados são extraídos para um objeto. Essa é a forma mais flexível de se lidar com dados legados complexos. Porém, maior esforço será necessário para a reconstituição dos objetos, em comparação com as estratégias anteriores.

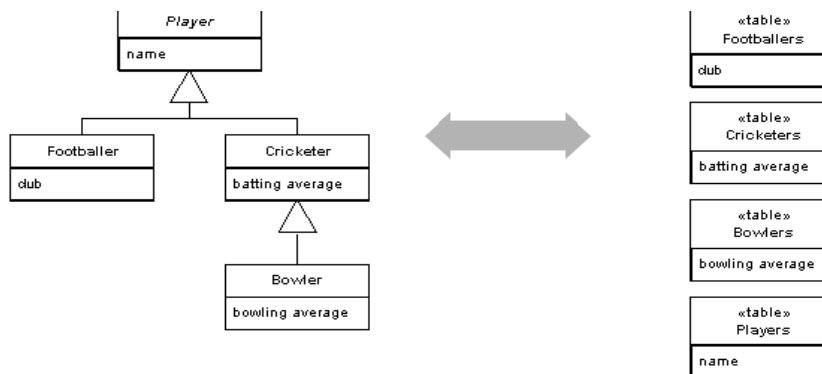


Fig. 16: Herança de Classe e Tabela [extraído de FOW02]

3.6. Considerações Finais

Neste capítulo abordou-se a aplicação de uma camada de persistência utilizando técnicas de MOR. Foi visto também que a utilização de formas simples de persistência, como a codificação de rotinas de bancos de dados relacionais em SQL diretamente nas classes de negócio O.O., resulta em código difícil de se manter e de se estender futuramente. Esta alternativa só é viável em aplicações muito pequenas e protótipos [AMB01]. Com este procedimento, há um acoplamento direto entre as classes de negócio da aplicação e o esquema do banco de dados relacional, e conseqüentemente, a cada mudança que ocorrer no modelo relacional, como por exemplo a renomeação de uma coluna, irá implicar na reescrita do código fonte da aplicação.

Uma abordagem um pouco melhor para este problema é o encapsulamento das rotinas SQL em "classes de dados", por exemplo, com a criação de *stored procedures* no banco de dados para representar os objetos. Mas ainda assim, se há mudanças no modelo, será preciso recompilar o código novamente.

Finalmente, apresentou-se a abordagem do mapeamento objeto-relacional (MOR), que permite que, através da criação da camada de persistência, seja feito o mapeamento entre objetos e o banco de dados relacional, e assim, mudanças no esquema deixam de implicar em mudanças na aplicação.

Através do mapeamento objeto-relacional, o código passa a ser muito mais consistente (eliminando a maioria dos problemas de impedância entre os modelos), simples de se manter e de ser estendido futuramente.

A principal idéia por trás dos *frameworks* de MOR é o mapeamento entre as tuplas de um banco de dados relacional e objetos em memória, de forma que possam ser manipulados por linguagens orientadas a objeto. A grande desvantagem desta técnica está no impacto da performance das aplicações, mas considerando as vantagens mencionadas acima, e com a

construção e aplicação adequada da camada de persistência, este fator pode ser bastante reduzido.

O mapeamento objeto-relacional pode ser utilizado também em diferentes mecanismos de persistência, entretanto, esta situação ser pouco adequada, especialmente quando o sistema enquadrar-se em um desses casos [JOH03]:

- No caso da modelagem centrada em objetos, o resultado é um esquema relacional artificial, onde há a necessidade de efetuar-se junções complexas para operações comuns de recuperação de dados e há falta de integridade referencial.
- No caso da modelagem relacional, o resultado é uma camada de objetos com relacionamento de um-para-um com as tabelas do SGBD, o que pode ser ineficiente.
- As consultas e atualizações no SGBD passam a ser ineficientes.
- Tarefas que podem ser efetuadas de maneira simples com operações relacionais podem necessitar de código significativo em aplicações O.O., ou resultar na criação de objetos desnecessários.

Em geral, o mapeamento objeto-relacional não é o método mais apropriado em sistemas onde há requisitos de OLAP (*Online Analytic Processing*) ou de *data warehousing*, que foram desenvolvidos para trabalhar especialmente com operações relacionais.

Um dos segredos do sucesso do mapeamento objeto-relacional é o entendimento de ambos paradigmas relacional e O.O.) e suas diferenças, e fazer “tradeoffs” baseados neste conhecimento [AMB98].

CAPÍTULO 4 - Estudo de caso: *Object Relational Bridge*

4.1. Introdução

Mapeamento objeto-relacional é um requisito comum de muitos projetos de software. As atividades envolvidas na persistência de dados são tediosas e passíveis de erro. Considerando as inevitáveis mudanças nos requisitos que ocorrem no ciclo de vida de um sistema, o mecanismo de persistência de dados deve ser mantido em sincronia com o código fonte, além das questões de portabilidade que podem estar envolvidas.

O *framework* OJB (*Object Relational Bridge*) é uma ferramenta que tem como propósito minimizar estes problemas, disponibilizando uma camada de persistência de objetos transparente para a aplicação O.O., onde a persistência é implementada sem que o objeto em questão necessite implementar uma interface ou herdar atributos persistentes.

Neste capítulo apresenta-se um estudo de caso onde a técnica de mapeamento objeto-relacional é aplicada utilizando o *framework* OJB.

4.2. Requisitos funcionais

A aplicação proposta neste estudo de caso não apresenta interface gráfica ou lógica de negócios, tendo sido desenvolvida com o objetivo de apresentar uma aplicação prática e didática da aplicação de MOR, através do *framework* OJB.

A escolha por este *framework* deve-se ao fato de ser uma ferramenta flexível (suporte a múltiplas APIs: o padrão ODMG 3.0, JDO, além do PB), integrante do projeto Apache (alto índice de adoção da comunidade Java e *open source* em geral), e principalmente, por apresentar-se enquanto uma ferramenta robusta de MOR, com recursos de *query*, *caching*, objetos *proxy*, enfim, de persistência transparente.

Esta aplicação foi desenvolvida na linguagem Java, utilizando a IDE Eclipse para a programação e *deployment*. Por ser baseada em Java, ela é compatível com os principais sistemas operacionais e arquiteturas atuais (Unix, Windows, MacOS, etc).

Abaixo, estão relacionadas as ferramentas utilizadas no processo, suas respectivas versões, e onde obtê-las:

- **J2SDK 1.4.2:** ferramentas e bibliotecas para programação Java;
- **Eclipse 3.0 M8:** Ambiente de desenvolvimento Java robusto e modularizado;
- **ObjectRelationalBridge (OBJ) 1.0-rc6:** *framework* de MOR do projeto Apache;
- **MySQL 4.0.18:** Banco de dados relacional *open-source*;
- **MySQL Connector/J 3.0:** *driver* do banco MySQL para JDBC.

O *script* de criação dos bancos encontra-se disponível em (ANEXO A). A instalação e configuração adequada destas ferramentas caracterizaram um ambiente funcional para a aplicação apresentada neste estudo.

4.3. Estrutura da aplicação

A aplicação proposta neste estudo de caso é um sistema de **controle financeiro**. O funcionamento do sistema foi bastante simplificado, com o objetivo de demonstrar a aplicação de MOR de forma clara e prática.

O sistema tem como base a criação de transações, associadas a diferentes “contas” (bancária, dinheiro, cartão de crédito, etc). Cada transação no sistema possui os seguintes dados: a data referente ao crédito ou débito (tipo de transação), o item (de compra, ou salário, etc), o valor e a instituição (banco, loja, etc).

Para tanto, foi desenvolvido o diagrama de classes, como visto na Fig. 17:

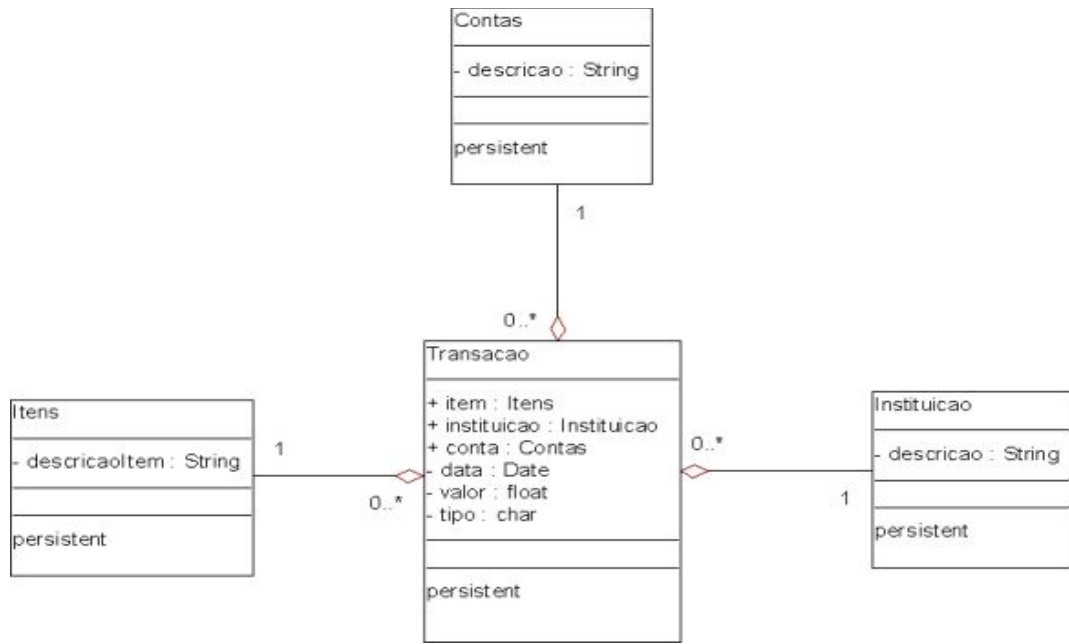


Fig. 17: Diagrama de Classes do estudo de caso.

Com base neste modelo, foi desenvolvido o modelo entidade-relacionamento (Fig. 18):

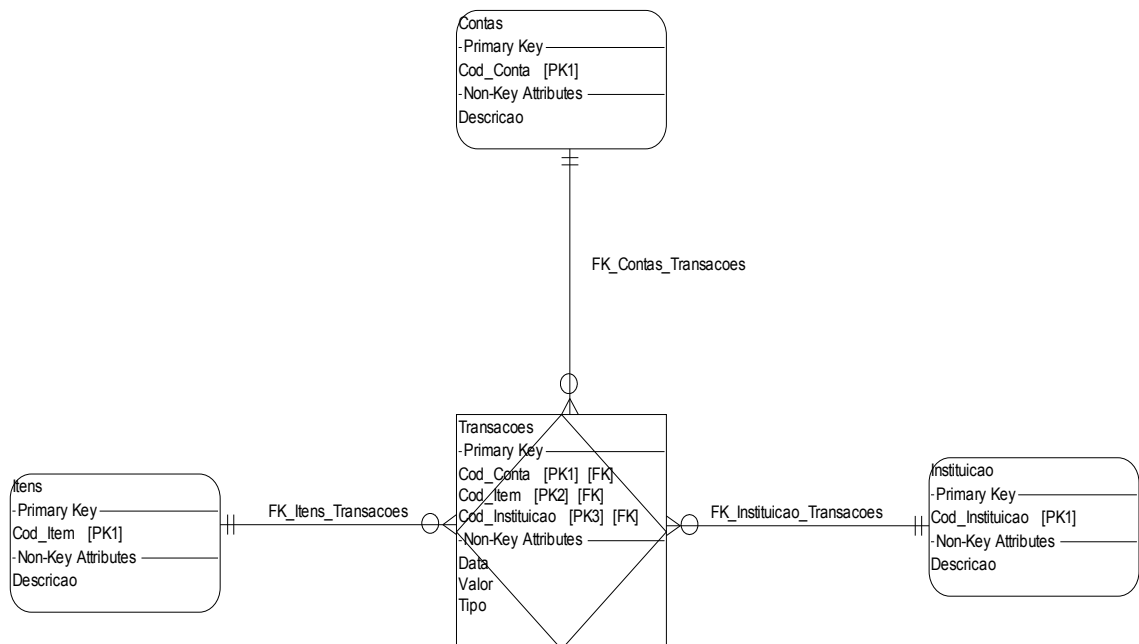


Fig. 18: Modelo ER do estudo de caso.

Desta forma, o processo de mapeamento incluirá os seguintes aspectos:

- mapeamento de classe-tabela simples, um-para-um;
- mapeamento de atributos primitivos (todos atributos possuem tipos equivalentes JDBC para a representação no banco);
- mapeamento de relacionamentos um-para-muitos.

O *framework* OJB implementa os respectivos mapeamentos através do arquivo XML **repository_user.xml**, que será detalhado na próxima seção.

4.4. Implementação

Inicialmente definem-se os parâmetros de conexão JDBC necessários para a comunicação entre o *framework* (OJB) e o banco de dados (MySQL).

O elemento `jdbc-connection-descriptor`, descrito no **repository.xml**, é o responsável por armazenar estas informações, entre elas, o driver, o protocolo, a localização do banco de dados e os dados de usuário, descritas abaixo (Listagem 4.1):

```
<jdbc-connection-descriptor
    jcd-alias="MYSQLCON"
    default-connection="true"
    plataforma="MySQL"
    jdbc-level="2.0"
    driver="com.mysql.jdbc.Driver"
    protocol="jdbc"
    subprotocol="mysql"
    dbalias="//localhost:3306/controle_financeiro"
    username="root"
    password=""
    batch-mode="false"
>
```

Listagem 4.1: Parâmetros de Conexão JDBC

Configura-se também o elemento `sequence-manager`, com o valor compatível com o banco em questão. Este atributo é o responsável pela geração de OIDs do OJB. No caso,

utilizamos o próprio recurso de geração de seqüências do banco de dados (Listagem 4.2).

```
<sequence-manager className="org.apache.obj.broker.util.sequence.
SequenceManagerNativeImpl">
</sequence-manager>
```

Listagem 4.2: *Sequence Manager*

Em seguida, descrevem-se os respectivos mapeamentos no documento XML **repository_user**, que contém as especificações da forma como as classes serão mapeadas no banco de dados. Este arquivo é referenciado pelo **repository.xml** através da notação XML “&”, que indica a chamada a outros arquivos na mesma estrutura.

Para a nossa classe Conta, relacionada à classe Transação, define-se o seguinte mapeamento de atributos primitivos (Listagem 4.3):

```
<field-descriptor
  name="codConta"
  column="COD_CONTA"
  jdbc-type="INTEGER"
  primaryKey="true"
  autoincrement="false"
  access="readonly"
/>
<field-descriptor
  name="descricao"
  column="DESCRICAO"
  jdbc-type="VARCHAR"
/>
```

Listagem 4.3: Mapeamento de atributos

O relacionamento “para-muitos”, por sua vez, define-se através do elemento de coleção **<collection-descriptor>** (Listagem 4.4):

```
<collection-descriptor
  name="transacoes"
  element-class-
ref="br.com.mackenzie.controlefinanceiro.Transacao"
  auto-retrieve="true"
  auto-update="true"
  auto-delete="true">
  <inverse-foreignkey field-ref="codConta" />
</collection-descriptor>
```

Listagem 4.4: Mapeamento de coleções

Para cada caso de relacionamento “para-muitos”, deve ser definida uma coleção, ao contrário de uma relação um-para-um, onde se definiria um <reference-descriptor> simples, como pode ser visto no mapeamento abaixo, referente à classe Transação (Listagem 4.5):

```
<reference-descriptor
  name="contas"
  class-ref="br.com.mackenzie.controlefinanceiro.Conta"
>
  <foreignkey field-ref="codConta" />
</reference-descriptor>
```

Listagem 4.5: Mapeamento 1:1

As classes Java da aplicação encontram-se divididas em classes de acesso aos dados (objetos DAO) e classes de negócio (*Value Objects*), com o objetivo de isolar os componentes responsáveis pela camada de apresentação (no caso, uma interface hipotética) das camadas de negócio e de acesso aos dados.

A comunicação entre as regras de negócio e a camada de dados ocorre através dos *JavaBeans*, que se responsabilizam por efetuar a conexão entre a interface e a classe DAO respectiva. Os *JavaBeans* utilizados são classes serializadas com atributos públicos, que mantêm os valores em um *bean*, removendo a necessidade de múltiplas chamadas para métodos que retornam um único valor, que resultariam em uma queda de performance significativa.

Seguindo esta componentização, a classe Contas apresenta os métodos de atribuição e recuperação de valores (*getter/setters*) (Listagem 4.6):

```
public class Conta implements Serializable{
    private int codConta;
    private String descricao;

    public String getDescricao() {return descricao;}
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public int getCodConta() {return codConta;}
}
```

Listagem 4.6: Classe Conta

Não há regra de negócio para a atribuição de código (codConta), pois como pode ser observado no esquema do banco de dados (ANEXO B), o campo CodConta é sequencial, atribuído automaticamente pelo SGBDR.

Por último, tem-se a definição das classes DAO referentes ao objeto Conta. Foram criadas duas classes, uma responsável pelas operações de INSERT, UPDATE e DELETE (OperacoesContas) e outra para a consulta dos valores do objeto (ConsultaContas).

O *framework* tem como base de funcionamento a classe *PersistenceBroker* (PB - Listagem 4.7), que deve ser instanciada no construtor de cada classe DAO:

```
public OperacaoContas(PersistenceBroker broker) {  
    this.broker = broker;  
}
```

Listagem 4.7: *Persistence Broker*

As operações são efetuadas com base no PB, a API de mais baixo nível do OJB, que faz a interface com o mecanismo de persistência. O PB responsabiliza-se pelas traduções necessárias entre as camadas e pelo gerenciamento das transações.

Um INSERT, por exemplo, consistiria nas operações a seguir (Listagem 4.8):

```
broker.beginTransaction();  
broker.store(conta);  
broker.commitTransaction();
```

Listagem 4.8: INSERT via *framework* de MOR

Desta forma, todas as operações seguem a mesma lógica orientada a objetos (objetos, métodos e mensagens). A operação de consulta necessita da declaração de um iterador para receber o valor de cada coluna. O critério de consulta, por sua vez, deve ser uma instância da classe *Criteria*. Esta classe tem métodos de consulta equivalentes aos encontrados na linguagem SQL (*addEqualTo* para o operador '=', *addGreaterThan*, para o operador '>', entre outros). A referência dos métodos da classe *Criteria* encontra-se na documentação da API do OJB.

Um exemplo de consulta na classe Conta, implementado pela classe DAO ConsultaConta, segue abaixo (Listagem 4.9):

```
Criteria criteria = new Criteria();

try{
    criteria.addEqualTo("codConta", new Integer(idConta));

    Query queryConta = new QueryByCriteria(Conta.class,
criteria);

    Iterator i = broker.getIteratorByQuery(queryConta);

    if(i.hasNext()){
        c = (Conta)i.next();
    } else {
        System.out.println("Registro não encontrado.");
    }
}
```

Listagem 4.9: Consulta via classe *Criteria*

Isto resume as principais operações e conceitos envolvidos na utilização do *framework* de MOR, OJB. O código completo e comentado da aplicação deste estudo de caso encontra-se em (ANEXO C).

4.5. Comparativos

Com a apresentação do estudo de caso, procurou-se demonstrar a utilização da orientação a objetos em toda a aplicação, inclusive nas questões de persistência de objetos em SGBDR, devido à utilização do *framework* de MOR como camada de persistência.

Sem o desenvolvimento de uma camada de persistência, o acesso ao SGBD ocorreria via programação SQL embutida no código, o que contraria os princípios do paradigma O.O., especialmente no que diz respeito aos fatores de encapsulamento e reutilização de código. A seguir, serão apresentados alguns exemplos para ilustrar esta situação.

O exemplo da página seguinte (Listagem 4.10) apresenta a implementação da classe DAO equivalente à classe Conta, descrita na seção anterior, porém utilizando chamadas JDBC

nativas:

```
import java.sql.*;

class Conta {
    String url = "jdbc:mysql://localhost/controle_financeiro";
    String username = "username";
    String passwd = "password";
    Connection con = null;
    Statement st = null;

    try {
        Class.forName("com.mysql.jdbc.Driver").newInstance();
        con = DriverManager.getConnection(url, username, passwd);
        con.setAutoCommit(false);
        st = con.createStatement();
        st.executeUpdate("INSERT INTO contas " +
            "VALUES (1, 'Cartão de crédito')");
        con.commit();
    } catch(Exception ex) {
        con.rollback();
    } finally {
        con.close();
    }
}
```

Listagem 4.10: Classe DAO JDBC

O conhecimento estrutural da tabela “Contas” é necessário neste tipo de solução, pois está se trabalhando com SQL diretamente.

A probabilidade de erros no código são maiores, pois há um novo fator de erro em potencial, o erro no código SQL.

Além disso, há uma nova preocupação: o estado da transação. É importante que se defina o *"rollback"* explicitamente, caso ocorra uma exceção, neste tipo de solução manual. Através de um *framework* como o OJB, estas questões são lidadas automaticamente.

Não há nenhum tipo de encapsulamento na utilização manual de JDBC: os valores são

passados diretamente aos atributos da classe DAO. A reutilização do código é prejudicada, por estar intimamente ligada com a estrutura do banco utilizado e a sintaxe SQL do mesmo.

O *framework* divide esta mesma operação em dois componentes:

- Primeiro, a atribuição de valores deve ser efetuada através de um *JavaBean* (Listagem 4.11):

```
Conta conta = new Conta();
OperacaoContas opeConta = new OperacaoContas(broker);

conta.setDescricao("Loja de Roupas");
opeConta.inserir(conta);
```

Listagem 4.11: Atribuição de valores ao objeto Conta

- Segundo, uma classe DAO responsabiliza-se pela manipulação dos dados (Listagem 4.12):

```
public void inserir(Conta contaVO) throws DataAccessException {
    PersistenceBroker broker = null;
    try {
        broker = ServiceLocator.getInstance().findBroker();

        broker.beginTransaction();
        broker.store(contaVO);
        broker.commitTransaction();

    } catch (PersistenceBrokerException pbe) {}
}
```

Listagem 4.12: Método INSERT da classe DAO

A programação JDBC nativa apresenta-se como uma solução de maior complexidade especialmente para consultas ao banco de dados. No caso da consulta de uma transação,

exemplificada no estudo de caso (ANEXO C), o código JDBC para tanto encontra-se a seguir (Listagem 4.13):

```
String tabelas = "itens, instituicoes, contas, transacoes";

StringBuffer sb = new StringBuffer();
sb.append("itens.COD_ITEM = 1");
sb.append("AND instituicoes.COD_INST = 1");
sb.append("AND contas.COD_CONTA = 1");
sb.append("AND transacoes.COD_CONTA = conta.COD_CONTA");
sb.append("AND transacoes.COD_ITEM = itens.COD_ITEM");
sb.append("AND transacoes.COD_INST = instituicoes.COD_INST");

String query = "select * from " + tabelas + "where " + sb;

try {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    Connection con = DriverManager.getConnection(url, username,
                                                password);

    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    ResultSetMetaData rsmd = rs.getMetaData();
    int numberOfColumns = rsmd.getColumnCount();

    while (rs.next()) {
        for (i = 1; i <= numberOfColumns; i++) {
            System.out.print(rs.getString(i));
        }
        System.out.println();
    }
    stmt.close();
    con.close();

} catch(SQLException ex) {}
```

Listagem 4.13: Consulta via JDBC

Como pode ser observado na listagem 4.13, para efetuar uma conexão JDBC, instancia-se o driver, indica-se o endereço do banco e os parâmetros de acesso diretamente na aplicação.

Através da camada de persistência, a mesma consulta poderia ser efetuada com muito menos esforço de programação, como pode ser observado abaixo (Listagem 4.14):

```
Criteria criteriaTransacao = new Criteria();

// "join" implícito
criteriaTransacao.addEqualTo("instituicoes.codInstituicao", "1");
criteriaTransacao.addEqualTo("contas.codConta", "1");
criteriaTransacao.addEqualTo("itens.codItem", "1");

Query query = QueryFactory.newQuery(Transacao.class,
                                   criteriaTransacao, true);

Iterator i = broker.getIteratorByQuery(query);
Transacao t = null;

while(i.hasNext()){
    t = (Transacao) i.next();
    System.out.println(t.getDescricao());
}
```

Listagem 4.14: Consulta via *framework* de MOR

Através de uma solução manual utilizando JDBC, a aplicação perde em flexibilidade, portabilidade, escalabilidade, eficiência e produtividade em geral. Para sistemas de maior nível de complexidade, comuns em ambientes corporativos que lidam com alto volume de dados e esquemas de bancos complexos, o desenvolvedor necessitaria conhecer a estrutura do banco em questão, a sintaxe SQL específica e efetuar junções complexas para manipular os dados em uma aplicação O.O..

A camada de persistência permite a abstração da camada de dados, e assim, o código passa a independe da configuração do banco. Os componentes da aplicação são isolados do

mecanismo de persistência. Se houver alterações na estrutura do banco, ou de banco para outro mecanismo de persistência, elas não refletirão na aplicação, somente nos descritores XML.

Um *framework* de MOR disponibiliza, em geral, diversas otimizações no que diz respeito ao acesso aos dados, entre elas, a manutenção de um *cache* de objetos, suporte a transações distribuídas, *pool* de conexões, compartilhamento de sessões, além da possibilidade de se efetuar operações de persistência de objetos. A implementação manual destas operações via código JDBC, por sua vez, é uma tarefa complexa e trabalhosa, comparando-se à utilização dos recursos de um *framework* de MOR, como o OJB, que efetua muitas destas operações automaticamente e via descrições XML (sem a necessidade da codificação).

CONCLUSÕES

Para qualquer sistema de informação, a persistência dos dados é um fator fundamental. A persistência de dados pode ser feita através da gravação dos dados em um dispositivo de armazenamento secundário. Entretanto, as limitações destes mecanismos em aplicações de médio a grande porte, que envolvem um volume relativamente grande de dados e necessidades de consulta e controle de concorrência, trouxeram a necessidade dos sistemas gerenciadores de bancos de dados (SGBD). Nesta categoria, destaca-se o modelo relacional, que provou ao longo das décadas desde sua concepção, ser um mecanismo eficiente e confiável de armazenamento de dados, possibilitando a manipulação, a manutenção da integridade e a segurança dos dados.

A persistência de objetos, por sua vez, envolve o paradigma da orientação a objetos, que difere em diversos aspectos do paradigma relacional. Objetos, além de dados, possuem comportamento. Objetos podem ser multi-valorados, englobando objetos complexos ou mesmo outros objetos, e terem atributos provenientes de relacionamentos (herança) com outros objetos. Estas, entre outras características, trazem uma camada de complexidade a mais no âmbito da persistência.

Alguns dos métodos de persistência de objetos incluem a serialização, o mapeamento dos objetos em documentos XML, e SGBD que suportam esta tecnologia, como os SGBDOR e os SGBDOO.

Porém, no contexto atual, os SGBDR representam o mecanismo de persistência de maior aceitação e maturidade. Por outro lado, a orientação a objetos nas camadas de negócio, e a metodologia UML de análise de sistemas já constituem o paradigma de desenvolvimento dominante no mercado.

Este fato leva a uma impedância entre o modelo da aplicação e o modelo de dados. A persistência de objetos por meio da técnica de mapeamento de objeto relacional permite que

contorne-se a impedância existente entre as metodologias O.O. e relacional, aproveitando assim os benefícios de ambas metodologias.

Para lidar com bancos relacionais em sistemas O.O., os desenvolvedores que não optam pelo desenvolvimento de uma camada de persistência, muitas vezes vão de encontro aos fundamentos da O.O., de reutilização e portabilidade do código. A partir do momento em que há o acoplamento da aplicação ao modelo de dados, qualquer mudança no esquema do banco afeta diretamente a aplicação, sendo necessário o retrabalho na adaptação de todas as classes de acesso ao banco. Dependendo do grau de complexidade da aplicação, o custo da reestruturação dos sistemas será altíssimo, o que prova que a falta de separação entre as camadas de dados e negócios resultam na ineficiência da aplicação como um todo.

O propósito do MOR é, basicamente, prover uma camada de persistência transparente entre aplicação orientada a objeto e o mecanismo de persistência relacional. O mecanismo de MOR atua na tradução transparente entre os modelos, exceto no processo de mapeamento, que envolve a definição de metadados em arquivo descritor, geralmente XML.

A utilização de técnicas de MOR, em especial através de *frameworks* de MOR, como o OJB, apresentado no estudo de caso, possibilita às empresas que tem como base SGBDR, direcionarem seus investimentos em inovação e tecnologia, com a utilização de soluções O.O., e, ao mesmo tempo, manterem a confiabilidade, segurança, e tradição dos bancos de dados relacionais. Através do MOR diminui-se o acoplamento entre o modelo de dados relacional e a aplicação O.O., permitindo a portabilidade e a escalabilidade futura das aplicações, e enfim, a diminuição do custo de manutenção do software.

Entretanto, o sucesso desta abordagem tem dependência intrínseca com a forma como o sistema foi estruturado, e podem não trazer o resultado esperado caso o projeto do banco relacional seja inadequado, assim como no caso de projetos que possuam requisitos de processamento analítico (OLAP).

REFERÊNCIAS BIBLIOGRÁFICAS

- [AMB98] AMBLER, S. W. **Building Object Applications that Work**. 1st ed. Cambridge: Cambridge University Press, 1998.
- [AMB01] AMBLER, S. W. Strategies for Storing Java Objects. **Java Developer's Journal**. New Jersey, v. 6, n. 8, p. 62-70, Aug 2001.
- [AMB03] AMBLER, S. W. **Agile Database Techniques**. 1st ed. New York: Wiley & Sons, 2003.
- [ATK89] ATKINSON, M.; *et al.* **The Object-Oriented Database System Manifesto**. Proceedings of the First International Conference on Deductive and Object-Oriented Databases, Japan, 1989. Disponível em: <<http://citeseer.ist.psu.edu/atkinson89objectoriented.html>>. Acesso em: 11 dez, 2003.
- [BER02] BERGLAS, Anthony. **Object Relational Mapping Tools**. SimpleORM Whitepaper, 2002. Disponível em: <<http://www.simpleorm.org/ORMTools.html>>. Acesso em: 10 fev. 2004.
- [BJW87] BIRRELL, J.; JONES; M. WOBBER, E. **A simple and efficient implementation for small databases**. Proceedings of the 11th ACM Symposium on Operating System Principles, S.I, 1987. Disponível em: <<http://birrell.org/andrew/papers/>>. Acesso em: 11 dez, 2003.
- [CAT00] CATTELL, R.; *et al.* **The Object Data Standard: ODMG 3.0**. San Francisco: Morgan Kaufmann, 2000.
- [COD70] CODD, E. F. **A Relational Model of Data for Large Shared Data Banks**. Communications of the ACM, v. 13, n. 6, 1970, pp. 377-387. Disponível em: <<http://www.acm.org/classics/nov95/>>. Acesso em: 10 jan. 2004.
- [DAT00] DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. 7 ed. Rio de Janeiro: Editora Campus, 2000.
- [DAT95] DATE, C. J.; DARWEN, H. **The Third Manifesto**. SIGMOD Record, v. 24, n. 1. S.I., 1995. Disponível em: <<http://citeseer.ist.psu.edu/darwen95third.html>>. Acesso em: 01 fev. 2004.
- [ECK02] ECKEL, B. **Thinking in Java**. 3rd ed. New Jersey: Prentice Hall, 2002.
- [EIM99] EISENBERG, A.; MELTON, Jim. **SQL:1999, formerly known as SQL3**. SIGMOD Record, v. 28, n. 1, pp. 131-138. S.I, 1999. Disponível em: <<http://dbs.uni-leipzig.de/en/lokal/standards.pdf>>. Acesso em: 01 fev 2004.
- [FAS97] FASSELL, M. **Foundations of Object Oriented Mapping**. S. l., 1997. Disponível em: <<http://www.chimu.com/publications/objectRelational/>>. Acesso em: 01 fev. 2004.

- [FLA97] FLANAGAN, D. **Java in a Nutshell**. 2nd ed. New York: O'Reilly, 1997.
- [FOW02] FOWLER, M. **Patterns of Enterprise Application Architecture**. 1st ed. New York: Addison-Wesley, 2002.
- [IBM00] IBM Rational Whitepapers. **Integrating Object and Relational Technologies**. Disponível em: <<http://www.rational.com/media/whitepapers/>>. Acesso em: 10 jan 2004.
- [IBM02] IBM Rational Whitepapers. **Mapping Object to Data Model with the UML**. Disponível em: <<http://www.rational.com/media/whitepapers/>>. Acesso em: 10 jan 2004.
- [JOH03] JOHNSON, R. **Expert One-on-One J2EE Design and Development**. Chicago: Wrox Press, 2003.
- [KEL97] KELLER, W. **Mapping Objects to Tables: A Pattern Language**. Proceedings of the European Pattern Languages Conference, Irsee, Germany: 1997. Siemens Technical Report 120/SW1/FB.
- [KRO99] KROENKE, D. M. **Database Processing: Fundamentals, Design and Implementation**. 7th ed. New Jersey: Prentice Hall, 1999.
- [LER99] LERMEN, A. **Um framework de mapeamento ODMG para SGBD O/R**. Cap. 4. Dissertação de Mestrado, UFRS, 1999. Disponível na biblioteca central da UNICAMP.
- [MUE97] MUELLER, P. **Introduction to Object-Oriented Programming in C++**. Berlin, 1997. Disponível em: <<http://www.zib.de/Visual/people/mueller/Courses/>>. Acesso em: 01 fev. 2004.
- [MUL99] MULLER, R. **Database Design for Smarties: Using UML for Data Modeling**. 1st ed. San Francisco: Morgan Kaufmann, 1999.
- [OBJ03] OBJECTMATTER. **Mapping Tool Guide**: Visual BSF, 1998-2003. Disponível em <<http://www.objectmatter.com/vbsf/docs/maptool/guide.html>>. Acesso em: 10 jan, 2004.
- [QUA99] QUADROS, N. **Um arcabouço reflexivo para persistência de objetos**. Tese de pós-graduação, Lab. de Sistemas Distribuídos do IC, UNICAMP, 1999.
- [RIC96] RICARTE, I. L. **Programação Orientada a Objetos com C++**. DCA/FEE UNICAMP, 1996. Disponível em: <<http://www.dca.fee.unicamp.br/courses/>>. Acesso em: 02 fev. 2004.
- [RAJ02] ROMAN, E.; AMBLER, S. JEWELL, T. **Mastering Enterprise Java Beans**. 2nd ed. New York: Wiley, 2002.

- [RUM90] RUMBAUGH, J.; *et al.* **Object-Oriented Modeling and Design**. 1st ed. New Jersey: Prentice-Hall, 1990.
- [SKS01] SILBERSCHATZ, A; KORTH, H.; SUDARSHAN, S. **Database Systems Concepts**. 4th ed. Boston: McGraw-Hill, 2001.
- [SPE03] SPERKO, R. **Java Persistence for Relational Databases**. 1st ed. APress, 2003.
- [STO90] STONEBRAKER, M.; *et al.* **Third-Generation Database System Manifesto**. SIGMOD Record, v.19 n.3, p.31-44. S.l, 1990. Disponível em: <<http://citeseer.ist.psu.edu/for90thirdgeneration.html>> Acesso em: 02 fev. 2004.
- [THO02] THOMAS, T. M. **Java Data Access: JDBC, JNDI, and JAXP**. 1st ed. New York: M&T: 2002.
- [W3C04] BRAY, T.; *et al.* **Extensible Markup Language (XML) 1.0**. 3rd ed. W3C Recommendation Paper. Disponível em: <<http://www.w3.org/TR/2004/REC-xml-20040204/>>. Acesso em: 10 jan. 2004.
- [WIL00] WILLIAMS, K.; *et al.* **Professional XML Databases**. 1st ed. Chicago: Wrox Press, 2000.
- [WOL03] WOLFGANG, K.; **Persistence Options for Object-Oriented Programming**. Alemanha, 2004. Disponível em: <<http://www.objectarchitects.de/>>. Acesso em: 10 abr. 2004.

ANEXO A: *Script de criação do banco de dados.*

```
# controle_financeiro.sql

CREATE DATABASE controle_financeiro;

USE controle_financeiro;

CREATE TABLE `contas` (
  `COD_CONTA` int(11) NOT NULL auto_increment,
  `DESCRICAO` varchar(50) NOT NULL default '',
  PRIMARY KEY (`COD_CONTA`)
) TYPE=INNODB;

CREATE TABLE `instituicoes` (
  `COD_INSTITUICAO` int(11) NOT NULL auto_increment,
  `DESCRICAO` varchar(50) NOT NULL default '',
  PRIMARY KEY (`COD_INSTITUICAO`)
) TYPE=INNODB;

CREATE TABLE `itens` (
  `COD_ITEM` int(11) NOT NULL auto_increment,
  `DESCRICAO` varchar(50) NOT NULL default '',
  PRIMARY KEY (`COD_ITEM`)
) TYPE=INNODB;

CREATE TABLE transacoes (
  COD_CONTA int(11) not null,
  COD_ITEM int(11) not null,
  COD_INSTITUICAO int(11) not null,
  valor double not null default '0',
  tipo char not null default '',
  data datetime not null,
  PRIMARY KEY (COD_ITEM, COD_CONTA, COD_INSTITUICAO),
  INDEX (COD_ITEM), INDEX (COD_CONTA), INDEX (COD_INSTITUICAO),
  CONSTRAINT FK_CONTA_TRAN FOREIGN KEY(COD_CONTA) REFERENCES contas
(COD_CONTA),
  CONSTRAINT FK_INST_TRAN FOREIGN KEY(COD_INSTITUICAO) REFERENCES
instituicoes(COD_INSTITUICAO),
  CONSTRAINT FK_ITEM_TRAN FOREIGN KEY(COD_ITEM) REFERENCES itens(COD_ITEM)
) TYPE=INNODB;
```

ANEXO B: XML de conexão e mapeamento.

repository.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Arquivo de Configuração do OJB: Configurações de Conexão -->
<!DOCTYPE descriptor-repository SYSTEM "repository.dtd"
[
<!ENTITY internal SYSTEM "repository_internal.xml">
<!ENTITY user SYSTEM "repository_user.xml">
]>

<descriptor-repository version="1.0" isolation-level="read-uncommitted">

  <!-- Configuração do Banco de Dados que iremos utilizar (MySQL) -->
  <jdbc-connection-descriptor
    jcd-alias="MYSQLCON"
    default-connection="true"
    plataforma="MySQL"
    jdbc-level="2.0"
    driver="com.mysql.jdbc.Driver"
    protocol="jdbc"
    subprotocol="mysql"
    dbalias="//localhost:3306/controle_financeiro"
    username="root"
    password=""
    batch-mode="false"
  >

  <!-- Configuração do Pool da Conexão -->
  <connection-pool
    maxActive="5"
    ValidationQuery=""
  />

  <sequence-manager
    className="org.apache.ojb.broker.util.sequence.SequenceManagerNativeImpl">
  </sequence-manager>

  </jdbc-connection-descriptor>

  <!-- Arquivo XML que contem os mapeamentos internos do OJB -->
  &internal;

  <!-- Arquivo XML que contem os mapeamentos definidos pelo desenvolvedor
  -->
  &user;

</descriptor-repository>
```

repository_user.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- Arquivo de Configuração do mapeamento O/R da aplicação -->

<!-- Inicio do mapeamento da tabela CONTAS -->
  <class-descriptor
    class="br.com.mackenzie.controlefinanceiro.Conta"
    table="CONTAS"
  >
    <field-descriptor
      name="codConta"
      column="COD_CONTA"
      jdbc-type="INTEGER"
      primarykey="true"
      autoincrement="false"
      access="readonly"
    />
    <field-descriptor
      name="descricao"
      column="DESCRICAO"
      jdbc-type="VARCHAR"
    />

    <collection-descriptor
      name="transacoes"
      element-class-ref="br.com.mackenzie.controlefinanceiro.Transacao"
      auto-retrieve="true"
      auto-update="true"
      auto-delete="true">
      <inverse-foreignkey field-ref="codConta" />
    </collection-descriptor>

  </class-descriptor>

<!-- Fim do Mapeamento da tabela CONTAS -->

<!-- Inicio do mapeamento da tabela INSTITUICOES -->
  <class-descriptor
    class="br.com.mackenzie.controlefinanceiro.Instituicao"
    table="INSTITUICOES"
  >
    <field-descriptor
      name="codInstituicao"
      column="COD_INSTITUICAO"
      jdbc-type="INTEGER"
      primarykey="true"
      autoincrement="false"
      access="readonly"
    />
    <field-descriptor
      name="descricao"
      column="DESCRICAO"
      jdbc-type="VARCHAR"
    />

    <collection-descriptor
      name="transacoes"
      element-class-ref="br.com.mackenzie.controlefinanceiro.Transacao"
```

```

        auto-retrieve="true"
        auto-update="true"
        auto-delete="true">
        <inverse-foreignkey field-ref="codInstituicao" />
    </collection-descriptor>

</class-descriptor>
<!-- Fim do Mapeamento da tabela INSTITUICOES -->

<!-- Inicio do mapeamento da tabela ITENS -->
<class-descriptor
    class="br.com.mackenzie.controlefinanceiro.Item"
    table="ITENS"
>
    <field-descriptor
        name="codItem"
        column="COD_ITEM"
        jdbc-type="INTEGER"
        primarykey="true"
        autoincrement="false"
        access="readonly"
    />
    <field-descriptor
        name="descricaoItem"
        column="DESCRICAO"
        jdbc-type="VARCHAR"
    />

    <collection-descriptor
        name="transacoes"
        element-class-ref="br.com.mackenzie.controlefinanceiro.Transacao"
        auto-retrieve="true"
        auto-update="true"
        auto-delete="true">
        <inverse-foreignkey field-ref="codItem" />
    </collection-descriptor>

</class-descriptor>
<!-- Fim do Mapeamento da tabela ITENS -->

<!-- Inicio do mapeamento da tabela TRANSACOES -->
<class-descriptor
    class="br.com.mackenzie.controlefinanceiro.Transacao"
    table="TRANSACOES"
>
    <field-descriptor
        name="codConta"
        column="COD_CONTA"
        jdbc-type="INTEGER"
        primarykey="true"
    />
    <field-descriptor
        name="codItem"
        column="COD_ITEM"
        jdbc-type="INTEGER"
        primarykey="true"
    />
    <field-descriptor
        name="codInstituicao"

```

```

        column="COD_INSTITUICAO"
        jdbc-type="INTEGER"
        primarykey="true"
    />
    <field-descriptor
        name="data"
        column="DATA"
        jdbc-type="DATE"
    />
    <field-descriptor
        name="valor"
        column="VALOR"
        jdbc-type="FLOAT"
    />
    <field-descriptor
        name="tipo"
        column="TIPO"
        jdbc-type="VARCHAR"
    />

    <!-- Referencia tabela Itens -->
    <reference-descriptor
        name="itens"
        class-ref="br.com.mackenzie.controlefinanceiro.Item"
    >
        <foreignkey field-ref="codItem" />
    </reference-descriptor>

    <!-- Referencia tabela Contas -->
    <reference-descriptor
        name="contas"
        class-ref="br.com.mackenzie.controlefinanceiro.Conta"
    >
        <foreignkey field-ref="codConta" />
    </reference-descriptor>

    <!-- Referencia tabela Instituicoes -->
    <reference-descriptor
        name="instituicoes"
        class-ref="br.com.mackenzie.controlefinanceiro.Instituicao"
    >
        <foreignkey field-ref="codInstituicao" />
    </reference-descriptor>

</class-descriptor>
<!-- Fim do Mapeamento da tabela TRANSACOES -->

```

ANEXO C: Código do estudo de caso.

```
/*
 * Classe ControleFinanceiro
 *
 */

import br.com.mackenzie.controlefinanceiro.Conta;
import br.com.mackenzie.controlefinanceiro.dao.OperacaoContas;

import br.com.mackenzie.controlefinanceiro.Item;
import br.com.mackenzie.controlefinanceiro.dao.OperacaoItens;

import br.com.mackenzie.controlefinanceiro.Instituicao;
import br.com.mackenzie.controlefinanceiro.dao.OperacaoInstituicoes;

import br.com.mackenzie.controlefinanceiro.Transacao;
import br.com.mackenzie.controlefinanceiro.dao.OperacaoTransacoes;
import br.com.mackenzie.controlefinanceiro.dao.ConsultaTransacoes;

import org.apache.ogb.broker.PersistenceBroker;
import org.apache.ogb.broker.PersistenceBrokerFactory;
import java.util.Date;

/**
 *
 * @author Reinaldo/Camila
 *
 * Esta classe tem como objetivo, gravar algumas informações
 * no banco de dados, e efetuar uma consulta utilizando uma
 * técnica de INNER JOIN.
 */
public class ControleFinanceiro {

    public static void main(String[] args) {
        //Cria e instancia classe de conexão com banco de dados (Via
arquivo XML).
        final PersistenceBroker broker;
        broker = PersistenceBrokerFactory.defaultPersistenceBroker();

/*INSERE REGISTROS NA TABELA CONTAS*/
        try{
            //Cria e instancia classe de contas e operações na conta.
            Conta conta = new Conta();
            OperacaoContas opeConta = new OperacaoContas(broker);

            //Insere registros na tabela contas.
            conta.setDescricao("Loja de Roupas");
            opeConta.inserir(conta);
            conta.setDescricao("Loja de Sapatos");
            opeConta.inserir(conta);
            conta.setDescricao("Loja de Brinquedos");
            opeConta.inserir(conta);
            conta.setDescricao("Loja de Automoveis");
```

```

        opeConta.inserir(conta);
    }
    catch (Exception e){
        System.out.println("Erro ao inserir dados das contas." +
e.getMessage());
        System.exit(0);
    }

/*INSERE REGISTROS NA TABELA INSTITUICOES*/
    try{
        //Cria e instancia classe de instituicoes e operações na
        instituicao.
        Instituicao instituicao = new Instituicao();
        OperacaoInstituicoes opeinstituicao = new
OperacaoInstituicoes(broker);

        //Insere registros na tabela instituicao.
        instituicao.setDescricao("DasRoupas SA");
        opeinstituicao.inserir(instituicao);
        instituicao.setDescricao("Sapataria São Miguel");
        opeinstituicao.inserir(instituicao);
        instituicao.setDescricao("Brinquedolandia SA");
        opeinstituicao.inserir(instituicao);
        instituicao.setDescricao("CarraoNaHora SA");
        opeinstituicao.inserir(instituicao);
    }
    catch (Exception e){
        System.out.println("Erro ao inserir dados das
instituicoes." + e.getMessage());
        System.exit(0);
    }

/*INSERE REGISTROS NA TABELA ITENS*/
    try{
        //Cria e instancia classe de itens e operações no item.
        Item item = new Item();
        OperacaoItens opeitem = new OperacaoItens(broker);

        //Insere registros na tabela item.
        item.setDescricaoItem("Camisas");
        opeitem.inserir(item);
        item.setDescricaoItem("Calcas");
        opeitem.inserir(item);
        item.setDescricaoItem("Meias");
        opeitem.inserir(item);
        item.setDescricaoItem("Sapatos");
        opeitem.inserir(item);
        item.setDescricaoItem("Tennis");
        opeitem.inserir(item);
        item.setDescricaoItem("Carrinhos");
        opeitem.inserir(item);
        item.setDescricaoItem("Bolas");
        opeitem.inserir(item);
        item.setDescricaoItem("Gol");
        opeitem.inserir(item);
        item.setDescricaoItem("Uno");
        opeitem.inserir(item);
        item.setDescricaoItem("BMW");
        opeitem.inserir(item);
    }

```



```

        catch (Exception e) {
            System.out.println("Erro ao inserir dados dos itens." +
e.getMessage());
            System.exit(0);
        }
    /*INSERE REGISTROS NA TABELA TRANSACOES*/
    try {
        //Cria e instancia classe de transacoes e operações na
transacao.
        Transacao transacao = new Transacao();
        OperacaoTransacoes opetransacao = new OperacaoTransacoes
(broker);

        //Insere registros na tabela transacao.
transacao.setCodigoConta(1);
transacao.setCodigoInstituicao(1);
transacao.setCodigoItem(1);
transacao.setData(new Date());
transacao.setValor(127);
transacao.setTipo("D");
opetransacao.inserir(transacao);

transacao.setCodigoConta(3);
transacao.setCodigoInstituicao(3);
transacao.setCodigoItem(7);
transacao.setData(new Date());
transacao.setValor(50);
transacao.setTipo("D");
opetransacao.inserir(transacao);
    }
    catch (Exception e) {
        System.out.println("Erro ao inserir dados das transacoes."
+ e.getMessage());
        System.exit(0);
    }
    /*CONSULTA REGISTROS A PARTIR DE UMA QUERY SQL(NAO ACONSELHAVEL)*/
    try {
        //Cria e instancia classe de consulta de transacoes.
ConsultaTransacoes consultaTransacaoSQL = new
ConsultaTransacoes(broker);
        //Cria um array para receber o resultado da consulta.
Transacao transacoes[] = null;
        //Recebe o array de transações com os dados selecionados.
transacoes = consultaTransacaoSQL.findTransacaoByChave
(1,1,1);

        //Mostra o resultado da consulta SQL em tela.
if(transacoes.length>0){
            for(int i=0; i < transacoes.length; i++){
                //Exibe os dados encontrados.
                Conta contas = transacoes[i].getConta();
                System.out.println("Conta = " +
contas.getDescricao() );
                Instituicao instituicoes = transacoes[i].
getInstituicoes();
                System.out.println("Instituicao = " +
instituicoes.getDescricao());
                Item itens = transacoes[i].getItens();

```



```

    public int getCodConta() {
        return codConta;
    }
    /**
     * @return Returns the transacoes.
     */
    public Transacao[] getTransacoes() {
        return transacoes;
    }

    /**
     * @param transacoes The transacoes to set.
     */
    public void setTransacoes(Transacao[] transacoes) {
        this.transacoes = transacoes;
    }
}

/*
 * Classe Instituicao
 *
 */
package br.com.mackenzie.controlefinanceiro;

import br.com.mackenzie.controlefinanceiro.Transacao;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe representa a tabela INSTITUICOES no banco de dados.
 */
public class Instituicao {
    /*Propriedades*/
    private int codInstituicao;
    private String descricao;

    /*Referenciado pela FK*/
    private Transacao[] transacoes;

    /**
     * @return Returns the codInstituicao.
     */
    public int getCodInstituicao() {
        return codInstituicao;
    }

    /**
     * @param codInstituicao The codInstituicao to set.
     */
    public void setCodInstituicao(int codInstituicao) {
        this.codInstituicao = codInstituicao;
    }

    /**
     * @return Returns the descricao.

```

```

    */
    public String getDescricao() {
        return descricao;
    }

    /**
     * @param descricao The descricao to set.
     */
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    /**
     * @return Returns the transacoes.
     */
    public Transacao[] getTransacoes() {
        return transacoes;
    }

    /**
     * @param transacoes The transacoes to set.
     */
    public void setTransacoes(Transacao[] transacoes) {
        this.transacoes = transacoes;
    }
}

/*
 * Classe Item
 *
 */
package br.com.mackenzie.controlefinanceiro;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe representa a tabela ITENS no banco de dados.
 */
public class Item {
    /*Propriedades*/
    private int codItem;
    private String descricaoItem;

    /*Referenciado pela FK*/
    private Transacao[] transacoes;

    /**
     * @return Returns the codItem.
     */
    public int getCodItem() {
        return codItem;
    }

    /**
     * @param codItem The codItem to set.
     */
    public void setCodItem(int codItem) {
        this.codItem = codItem;
    }
}

```

```

    /**
     * @return Returns the descricaoItem.
     */
    public String getDescricaoItem() {
        return descricaoItem;
    }

    /**
     * @param descricaoItem The descricaoItem to set.
     */
    public void setDescricaoItem(String descricaoItem) {
        this.descricaoItem = descricaoItem;
    }

    /**
     * @return Returns the transacoes.
     */
    public Transacao[] getTransacoes() {
        return transacoes;
    }

    /**
     * @param transacoes The transacoes to set.
     */
    public void setTransacoes(Transacao[] transacoes) {
        this.transacoes = transacoes;
    }
}

/*
 * Classe Transacao
 */
package br.com.mackenzie.controlefinanceiro;
import java.util.Date;
import br.com.mackenzie.controlefinanceiro.Item;
import br.com.mackenzie.controlefinanceiro.Conta;
import br.com.mackenzie.controlefinanceiro.Instituicao;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe representa a tabela TRANSACOES no banco de dados.
 */
public class Transacao {
    //Propriedades.
    private int codConta;
    private int codItem;
    private int codInstituicao;
    private Date data;
    private double valor;
    private String tipo;

    //Classes para as FKs.
    private Item itens;
    private Conta contas;
    private Instituicao instituicoes;
}

```

```

/**
 * @return Returns the codigoConta.
 */
public int getCodigoConta() {
    return codConta;
}

/**
 * @param codigoConta The codigoConta to set.
 */
public void setCodigoConta(int codigoConta) {
    this.codConta = codigoConta;
}

/**
 * @return Returns the codigoInstituicao.
 */
public int getCodigoInstituicao() {
    return codInstituicao;
}

/**
 * @param codigoInstituicao The codigoInstituicao to set.
 */
public void setCodigoInstituicao(int codigoInstituicao) {
    this.codInstituicao = codigoInstituicao;
}

/**
 * @return Returns the codigoItem.
 */
public int getCodigoItem() {
    return codItem;
}

/**
 * @param codigoItem The codigoItem to set.
 */
public void setCodigoItem(int codigoItem) {
    this.codItem = codigoItem;
}

/**
 * @return Returns the data.
 */
public Date getData() {
    return data;
}

/**
 * @param data The data to set.
 */
public void setData(Date data) {
    this.data = data;
}

/**
 * @return Returns the tipo.
 */

```

```

public String getTipo() {
    return tipo;
}

/**
 * @param tipo The tipo to set.
 */
public void setTipo(String tipo) {
    this.tipo = tipo;
}

/**
 * @return Returns the valor.
 */
public double getValor() {
    return valor;
}

/**
 * @param valor The valor to set.
 */
public void setValor(float valor) {
    this.valor = valor;
}

/**
 * @return Returns the contas.
 */
public Conta getConta() {
    return this.contas;
}

/**
 * @param contas The contas to set.
 */
public void setConta(Conta conta) {
    this.contas = conta;
}

/**
 * @return Returns the instituicoes.
 */
public Instituicao getInstituicoes() {
    return instituicoes;
}

/**
 * @param instituicoes The instituicoes to set.
 */
public void setInstituicoes(Instituicao instituicoes) {
    this.instituicoes = instituicoes;
}

/**
 * @return Returns the itens.
 */
public Item getItens() {
    return itens;
}

```

```

    /**
     * @param itens The itens to set.
     */
    public void setItens(Item itens) {
        this.itens = itens;
    }

    /**
     * @param valor The valor to set.
     */
    public void setValor(double valor) {
        this.valor = valor;
    }
}

/*
 * Classe ConsultaContas
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import java.util.Iterator;

import org.apache.ajb.broker.PersistenceBroker;
import org.apache.ajb.broker.PersistenceBrokerException;
import org.apache.ajb.broker.query.Criteria;
import org.apache.ajb.broker.query.Query;
import org.apache.ajb.broker.query.QueryByCriteria;

import br.com.mackenzie.controlefinanceiro.Conta;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe consulta de dados referentes a
 * tabela CONTAS, de acordo com a chave primaria.
 */
public class ConsultaContas {
    //Instancia a classe do Framework.
    private PersistenceBroker broker;

    //Contrutor.
    public ConsultaContas(PersistenceBroker broker){
        this.broker = broker;
    }

    public Conta findContaByID(int idConta){
        //Gera instância de Conta.
        Conta c = new Conta();
        //Cria objeto criterio.
        Criteria criteria = new Criteria();

        try{
            //Indica o critério de seleção.
            criteria.addEqualTo("codConta", new Integer(idConta));

            //Gera query utilizando o criterio criado acima.

```



```

        Query queryConta = new QueryByCriteria(Conta.class,
criteria);

        //Recebe retorno da consulta.
        Iterator i = broker.getIteratorByQuery(queryConta);

        //Caso encontre resultado, retorna valor.
        if(i.hasNext()){
            c = (Conta)i.next();
        }
        else {
            System.out.println("Registro não encontrado.");
        }

        //Limpa variaveis.
        i = null;
        queryConta = null;
        criteria = null;
    }
    catch(PersistenceBrokerException e){
        System.out.println("Erro: " + e.getClass().getName() + " -
" + e.getMessage());
    }
    catch(Throwable t){
        System.out.println("Erro: " + t.getClass().getName() + " -
" + t.getMessage());
    }
    finally{
        return c;
    }
}

}

/*
 * Classe ConsultaInstituicoes
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import java.util.Iterator;

import org.apache.ajb.broker.PersistenceBroker;
import org.apache.ajb.broker.PersistenceBrokerException;
import org.apache.ajb.broker.query.Criteria;
import org.apache.ajb.broker.query.Query;
import org.apache.ajb.broker.query.QueryByCriteria;

import br.com.mackenzie.controlefinanceiro.Instituicao;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe consulta de dados referentes a
 * tabela INSTITUICOES, de acordo com a chave primaria.
 */
public class ConsultaInstituicoes {
    //Instancia a classe do Framework.

```

```

private PersistenceBroker broker;

//Contrutor.
public ConsultaInstituicoes(PersistenceBroker broker){
    this.broker = broker;
}

public Instituicao findInstituicoesByID(int idInstituicao){
    //Gera instância de Conta.
    Instituicao ins = new Instituicao();
    //Cria objeto criteria.
    Criteria criteria = new Criteria();

    try{
        //Indica o critério de seleção.
        criteria.addEqualTo("codInstituicao",new Integer
(idInstituicao));

        //Gera query utilizando o criterio criado acima.
        Query queryInstituicao = new QueryByCriteria
(Instituicao.class, criteria);
        //Recebe retorno da consulta.
        Iterator i = broker.getIteratorByQuery(queryInstituicao);

        //Caso encontre resultado, retorna valor.
        if(i.hasNext()){
            ins = (Instituicao)i.next();
        }
        else {
            System.out.println("Registro não encontrado.");
        }

        //Limpa variaveis.
        i = null;
        queryInstituicao = null;
        criteria = null;
    }
    catch(PersistenceBrokerException e){
        System.out.println("Erro: " + e.getClass().getName() + " -
" + e.getMessage());
    }
    catch(Throwable t){
        System.out.println("Erro: " + t.getClass().getName() + " -
" + t.getMessage());
    }
    finally{
        return ins;
    }
}

}

/*
 * Classe ConsultaItens
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

```

```

import java.util.Iterator;

import org.apache.ogb.broker.PersistenceBroker;
import org.apache.ogb.broker.PersistenceBrokerException;
import org.apache.ogb.broker.query.Criteria;
import org.apache.ogb.broker.query.Query;
import org.apache.ogb.broker.query.QueryByCriteria;

import br.com.mackenzie.controlefinanceiro.Item;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe consulta de dados referentes a
 * tabela ITENS, de acordo com a chave primaria.
 */
public class ConsultaItens {
    //Instancia a classe do Framework.
    private PersistenceBroker broker;

    //Contrutor.
    public ConsultaItens(PersistenceBroker broker){
        this.broker = broker;
    }

    public Item findItensByID(int idItem){
        //Gera instância de Conta.
        Item ite = new Item();
        //Cria objeto criteria.
        Criteria criteria = new Criteria();

        try{
            //Indica o critério de seleção.
            criteria.addEqualTo("codItem",new Integer(idItem));

            //Gera query utilizando o criterio criado acima.
            Query queryItem = new QueryByCriteria(Item.class,
criteria);

            //Recebe retorno da consulta.
            Iterator i = broker.getIteratorByQuery(queryItem);

            //Caso encontre resultado, retorna valor.
            if(i.hasNext()){
                ite = (Item)i.next();
            }
            else {
                System.out.println("Registro não encontrado.");
            }

            //Limpa variaveis.
            i = null;
            queryItem = null;
            criteria = null;
        }
        catch(PersistenceBrokerException e){
            System.out.println("Erro: " + e.getClass().getName() + " -
" + e.getMessage());
        }
        catch(Throwable t){

```

```

        System.out.println("Erro: " + t.getClass().getName() + " -
" + t.getMessage());
    }
    finally{
        return ite;
    }
}

}

/*
 * Classe ConsultaTransacoes
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import java.util.Iterator;
import java.util.ArrayList;

import org.apache.ojb.broker.PersistenceBroker;
import org.apache.ojb.broker.PersistenceBrokerException;
import org.apache.ojb.broker.query.Criteria;
import org.apache.ojb.broker.query.Query;
import org.apache.ojb.broker.query.QueryFactory;

import br.com.mackenzie.controlefinanceiro.Transacao;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe consulta de dados referentes a
 * tabela TRANSACOES, de acordo com a chave primaria.
 */
public class ConsultaTransacoes {
    //Instancia a classe do Framework.
    private PersistenceBroker broker;

    //Contrutor.
    public ConsultaTransacoes(PersistenceBroker broker){
        this.broker = broker;
    }

    public Transacao[] findTransacaoByChave(int idItem, int idInstituicao,
int idConta){
        //Variavel retorno.
        Transacao consulta[] = null;

        //Cria objeto de criterios de seleção.
        Criteria criteriaTransacao = new Criteria();

        try{
            //Indica o critério de seleção.
            // ** Para que este tipo de JOIN de tabelas existam,
            // é necessário que o arquivo repository_user.xml
            // tenha as chaves das tabelas devidamente ligadas,
            // em ambas as tabelas, para que as referencias sejam
            // cruzadas, caso contrario gerará um erro.

```

```

        criteriaTransacao.addEqualTo("instituicoes.codInstituicao",
"1");
        criteriaTransacao.addEqualTo("contas.codConta", "1");
        criteriaTransacao.addEqualTo("itens.descricao", new String
("Camisas"));
        criteriaTransacao.addEqualTo("tipo", new String("D"));
        //Gera a Query da consulta.
        Query queryResult = QueryFactory.newQuery(Transacao.class,
criteriaTransacao, true);

        //Recebe resultado da busca.
        Iterator i = broker.getIteratorByQuery(queryResult);

        //Caso encontre resultado, gera array de transacoes.
        ArrayList array = new ArrayList();
        while(i.hasNext()){
            Transacao conTrans = (Transacao) i.next();
            array.add(conTrans);
            conTrans = null;
        }

        //Limpa variaveis.
        i = null;
        queryResult = null;
        criteriaTransacao = null;

        //Insere os dados do array, num array de Transacoes.
        if(array.size() > 0){
            consulta = new Transacao[array.size()];
            for(int j=0; j<array.size(); j++){
                consulta[j] = (Transacao) array.get(j);
            }
        }
        catch(PersistenceBrokerException e){
            System.out.println("Erro: " + e.getClass().getName() + " -
" + e.toString());
        }
        catch(Throwable t){
            System.out.println("Erro: " + t.getClass().getName() + " -
" + t.toString());
        }
        finally{
            //Retorna array de transacoes.
            return consulta;
        }
    }
}

/*
 * Classe OperacaoContas
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import org.apache.objb.broker.PersistenceBroker;
import org.apache.objb.broker.PersistenceBrokerException;

```

```

import br.com.mackenzie.controlefinanceiro.Conta;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe operações que poder ser realizadas na
 * tabela CONTAS, de acordo com a instancia informada.
 */
public class OperacaoContas {
    private PersistenceBroker broker;

    public OperacaoContas(PersistenceBroker broker) {
        this.broker = broker;
    }

    //Metodo para Inserir registros.
    public boolean inserir(Conta conta){
        boolean retorno = false;

        try{
            broker.beginTransaction();
            broker.store(conta);
            broker.commitTransaction();

            retorno = true;
        }
        catch (PersistenceBrokerException e){
            try{
                broker.abortTransaction();
                System.out.println("Erro");
            }
            catch (Exception ex){
                System.out.println("Erro");
            }
            System.out.println("Erro");
        }
        catch (Throwable t){
            try{
                broker.abortTransaction();
                System.out.println("Erro");
            }
            catch (Exception ex){
                System.out.println("Erro");
            }
            System.out.println("Erro");
        }
        finally{
            return retorno;
        }
    }

    //Metodo para Alterar registros
    public boolean alterar(Conta conta) {
        boolean retorno = false;

        try{
            broker.beginTransaction();
            broker.store(conta);
            broker.commitTransaction();

```

```

        retorno = true;
    }
    catch (PersistenceBrokerException e) {
        try {
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    catch (Throwable t) {
        try {
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    finally {
        return retorno;
    }
}

//Metodo para excluir registros
public boolean excluir(Conta conta) {
    boolean retorno = false;

    try {
        broker.beginTransaction();
        broker.delete(conta);
        broker.commitTransaction();

        retorno = true;
    }
    catch (PersistenceBrokerException e) {
        try {
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    catch (Throwable t) {
        try {
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
}

```

```

        finally {
            return retorno;
        }
    }
}

/*
 * OperacaoInstituicoes
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import org.apache.ogb.broker.PersistenceBroker;
import org.apache.ogb.broker.PersistenceBrokerException;

import br.com.mackenzie.controlefinanceiro.Instituicao;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe operações que poder ser realizadas na
 * tabela INSTITUICOES, de acordo com a instancia informada.
 */
public class OperacaoInstituicoes {
    private PersistenceBroker broker;

    public OperacaoInstituicoes(PersistenceBroker broker) {
        this.broker = broker;
    }

    //Metodo para Inserir registros.
    public boolean inserir(Instituicao instituicao){
        boolean retorno = false;

        try{
            broker.beginTransaction();
            broker.store(instituicao);
            broker.commitTransaction();

            retorno = true;
        }
        catch (PersistenceBrokerException e){
            try{
                System.out.println("Erro");
                broker.abortTransaction();
            }
            catch (Exception ex){
                System.out.println("Erro");
            }
            System.out.println("Erro");
        }
        catch (Throwable t){
            try{
                System.out.println("Erro");
                broker.abortTransaction();
            }
            catch (Exception ex){

```



```

        System.out.println("Erro");
    }
    System.out.println("Erro");
}
finally{
    return retorno;
}
}

//Metodo para Alterar registros
public boolean alterar(Instituicao instituicao) {
    boolean retorno = false;

    try{
        broker.beginTransaction();
        broker.store(instituicao);
        broker.commitTransaction();

        retorno = true;
    }
    catch (PersistenceBrokerException e){
        try{
            broker.abortTransaction();
        }
        catch (Exception ex){
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    catch (Throwable t){
        try{
            broker.abortTransaction();
        }
        catch (Exception ex){
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    finally{
        return retorno;
    }
}

//Metodo para excluir registros
public boolean excluir(Instituicao instituicao){
    boolean retorno = false;

    try{
        broker.beginTransaction();
        broker.delete(instituicao);
        broker.commitTransaction();

        retorno = true;
    }
    catch (PersistenceBrokerException e){
        try{
            broker.abortTransaction();
        }
        catch (Exception ex){

```

```

        System.out.println("Erro");
    }
    System.out.println("Erro");
}
catch(Throwable t){
    try{
        System.out.println("Erro");
        broker.abortTransaction();
    }
    catch(Exception ex){
        System.out.println("Erro");
    }
    System.out.println("Erro");
}
finally {
    return retorno;
}
}

}

/*
 * OperacaoItens
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import org.apache.ajb.broker.PersistenceBroker;
import org.apache.ajb.broker.PersistenceBrokerException;

import br.com.mackenzie.controlefinanceiro.Item;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe operações que poder ser realizadas na
 * tabela ITENS, de acordo com a instancia informada.
 */
public class OperacaoItens {
    private PersistenceBroker broker;

    public OperacaoItens(PersistenceBroker broker) {
        this.broker = broker;
    }

    //Metodo para Inserir registros.
    public boolean inserir(Item item){
        boolean retorno = false;

        try{
            broker.beginTransaction();
            broker.store(item);
            broker.commitTransaction();

            retorno = true;
        }
        catch(PersistenceBrokerException e){
            try{
                System.out.println("Erro");
            }

```

```

        broker.abortTransaction();
    }
    catch (Exception ex) {
        System.out.println("Erro");
    }
    System.out.println("Erro");
}
catch (Throwable t) {
    try {
        System.out.println("Erro");
        broker.abortTransaction();
    }
    catch (Exception ex) {
        System.out.println("Erro");
    }
    System.out.println("Erro");
}
finally {
    return retorno;
}
}

//Metodo para Alterar registros
public boolean alterar(Item item) {
    boolean retorno = false;

    try {
        broker.beginTransaction();
        broker.store(item);
        broker.commitTransaction();

        retorno = true;
    }
    catch (PersistenceBrokerException e) {
        try {
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    catch (Throwable t) {
        try {
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    finally {
        return retorno;
    }
}

//Metodo para excluir registros
public boolean excluir(Item item) {
    boolean retorno = false;

```

```

        try{
            broker.beginTransaction();
            broker.delete(item);
            broker.commitTransaction();

            retorno = true;
        }
        catch (PersistenceBrokerException e){
            try{
                broker.abortTransaction();
            }
            catch (Exception ex){
                System.out.println("Erro");
            }
            System.out.println("Erro");
        }
        catch (Throwable t){
            try{
                System.out.println("Erro");
                broker.abortTransaction();
            }
            catch (Exception ex){
                System.out.println("Erro");
            }
            System.out.println("Erro");
        }
        finally {
            return retorno;
        }
    }
}

/*
 * OperacaoTransacoes
 *
 */
package br.com.mackenzie.controlefinanceiro.dao;

import org.apache.ajb.broker.PersistenceBroker;
import org.apache.ajb.broker.PersistenceBrokerException;

import br.com.mackenzie.controlefinanceiro.Transacao;

/**
 *
 * @author Reinaldo/Camila
 * Esta classe exibe operações que poder ser realizadas na
 * tabela TRANSACOES, de acordo com a instancia informada.
 */
public class OperacaoTransacoes {
    private PersistenceBroker broker;

    public OperacaoTransacoes(PersistenceBroker broker) {
        this.broker = broker;
    }

    //Metodo para Inserir registros.

```

```

public boolean inserir(Transacao transacao){
    boolean retorno = false;

    try{
        broker.beginTransaction();
        broker.store(transacao);
        broker.commitTransaction();

        retorno = true;
    }
    catch(PersistenceBrokerException e){
        try{
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch(Exception ex){
            System.out.println("Erro" + ex.getMessage());
        }
        System.out.println("Erro" + e.getMessage());
    }
    catch(Throwable t){
        try{
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch(Exception ex){
            System.out.println("Erro" + ex.getMessage());
        }
        System.out.println("Erro" + t.getMessage());
    }
    finally{
        return retorno;
    }
}

//Metodo para Alterar registros
public boolean alterar(Transacao transacao) {
    boolean retorno = false;

    try{
        broker.beginTransaction();
        broker.store(transacao);
        broker.commitTransaction();

        retorno = true;
    }
    catch(PersistenceBrokerException e){
        try{
            broker.abortTransaction();
        }
        catch(Exception ex){
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    catch(Throwable t){
        try{
            broker.abortTransaction();
        }
    }
}

```

```

        catch (Exception ex) {
            System.out.println("Erro");
        }
        System.out.println("Erro");
    }
    finally {
        return retorno;
    }
}

//Metodo para excluir registros
public boolean excluir(Transacao transacao) {
    boolean retorno = false;

    try {
        broker.beginTransaction();
        broker.delete(transacao);
        broker.commitTransaction();

        retorno = true;
    }
    catch (PersistenceBrokerException e) {
        try {
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro" + ex.getMessage());
        }
        System.out.println("Erro" + e.getMessage());
    }
    catch (Throwable t) {
        try {
            System.out.println("Erro");
            broker.abortTransaction();
        }
        catch (Exception ex) {
            System.out.println("Erro" + ex.getMessage());
        }
        System.out.println("Erro" + t.getMessage());
    }
    finally {
        return retorno;
    }
}
}

```