UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um *framework* para construção de aplicações OO sobre SGBD relacional

por

KURT WERNER MOLZ

Dissertação submetida à avaliação, como
requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação.

Prof. Dr. Carlos Alberto Heuser
Orientador

Porto Alegre, fevereiro de 1999.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Molz, Kurt Werner

Um *framework* para construção de aplicações OO sobre SGBD relacional / Kurt Werner Molz. - Porto Alegre : CPGCC da UFRGS, 1999.

82 f.:il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR - RS,1999. Orientador: Heuser, Carlos Alberto.

1. Orientação a Objetos 2. Banco de Dados Relacional. 3. Persistência de Objetos. I. Heuser, Carlos A. II. Título

Agradecimentos

Ao Prof. Dr. Carlos Alberto Heuser, pela sua valiosa orientação e aos incentivos fornecidos para a realização deste curso e na elaboração da presente dissertação.

Agradeço também aos professores membros da banca, por terem aceitado a fazerem parte desta, e assim fazem-me sentir honrado por tais presenças.

A todos os colegas do Instituto que com o passar do tempo se tornaram meus grandes amigos, que através do espírito de companheirismo tornaram mais alegres os momentos dentro do Instituto de Informática. Aos colegas e amigos da Unisc, que deram sempre que foi necessário, todo o apoio. A todas as demais pessoas que conheci durante este período.

A todos os funcionários do Instituto de Informática e da Biblioteca pelo esmero e disposição que sempre colocaram em seus serviços.

A minha mãe Dulce A. Molz e em memória ao meu pai Arlindo Molz, pelo seu profundo amor e apoio, e por tudo que me ensinaram no transcorrer da minha vida. Ao meu irmão Rolf F. Molz, com o qual dividi vários dos momentos vividos no transcorrer deste curso. Agradeço também a minha esposa Claudia pelo amor e confiança depositada em mim e pela paciência em escutar as tantas palavras dificeis que eu tinha que desabafar.

Agradeço a Deus, sem o qual nada seria possível, pela vida, pela esperança e pelas bênçãos.

Por fim, agradeço a todas as pessoas que ajudaram, direta ou indiretamente, durante a realização deste curso de mestrado.

A todos vocês o meu muito obrigado!

Sumário

Lista de Abreviaturas

ADT	Abstract Data Type
DP	Domínio do Problema
GOP	Gatway Object Persistence
GD	Gerenciador de Dados
IH	Interface Homem-Máquina
ODMG	Object Database Management Group
SGBDOO	Sistema Gerenciador de Banco de Dados Orientado a Objetos
SGBDR	Sistema Gerenciador de Banco de Dados Relacional

Lista de Figuras

Lista de Tabelas

Resumo

O paradigma da orientação a objetos está se tornando a abordagem preferida para construção de sistemas em ambiente de banco de dados. Por outro lado, a tecnologia relacional é amplamente adotada para gerenciar dados corporativos. Os bancos de dados relacionais tornaram-se o padrão no armazenamento de dados para aplicações de processamento de transações on-line (OLTP). Estas tendências estão motivando a necessidade de construção de aplicações orientadas a objetos que acessem banco de dados relacionais.

O uso de conceitos orientado a objetos, como herança, permitem uma modelagem mais adequada e uma melhor implementação da aplicação baseada em sistema de banco de dados orientado a objetos. Entretanto, os resultados do projeto orientado a objetos, podem também ser aplicados em sistemas clássicos de banco de dados.

O trabalho apresenta o uso de padrões de projeto na construção de uma arquitetura de um *framework* que auxilie o mapeamento de uma aplicação OO a um SGBD relacional. Esta arquitetura segue a abordagem de persistência de objetos baseada em *gateways*, que é uma camada de software inserida entre o sistema gerenciador de banco de dados e a aplicação orientada a objetos, cujo o objetivo é dar suporte a um modelo de programação de aplicações OO.

A característica principal desta arquitetura é a separação clara das classes que tratam da base de dados em relação às classes que tratam do domínio do problema da aplicação. Esta divisão de responsabilidades permite a substituição das classes referentes a base de dados por outras, permitindo a migração da aplicação entre bases de dados diferentes.

São apresentados neste trabalho, formas de mapeamentos de esquemas orientados a objetos para esquemas relacionais. Estes mapeamentos acontecem do modelo OO para o modelo relacional. É importante salientar, que a arquitetura que está sendo proposta, não vai impedir que aplicações estruturadas deixem ter acesso a base de dados relacional mapeada, pois esta abordagem foi escolhida para permitir que novas aplicações OO tenham acesso a base de dados relacionais já existentes.

Como a implementação deste trabalho segue a abordagem de *gateway*, são apresentados os conceitos de orientação objetos, e como estes serão suportados na arquitetura, ou seja, o que o *gateway* deverá implementar.

Palavras-chave: Orientação a Objetos, Banco de Dados Relacional, Persistência de Objetos.

Title: "Object-oriented application design in a relational database."

Abstract

The paradigm of the object-oriented is becoming the approach preferred for construction of systems in database environment. On the other hand, the technology relational is adopted thoroughly for management corporate data. The relational databases they became the pattern in the storage of data for applications of processing of transactions on-line (OLTP). These tendencies are motivating the need of construction of applications object-oriented that acessem relational databases.

The way of using object-oriented conception, how inheritance, to make possible the better modeling and implementation based in object-oriented database systems. Therefore, the objetc-oriented design results, also is possible to application in classics database systems.

The work presents the use of project patterns in the construction of an architecture of a framework that aids the mapeamento of an application OO to a SGBD relacional. This architecture follows the approach of set persistence of objects in gateways, that is a software layer inserted among the system database manager and the object-oriented application, whose the objective is to give support to a model of programming of applications OO.

The main characteristic of this architecture is the clear separation of the classes that are about the database in relation to the classes that are about the domain of the problem of the application. This division of responsibilities allows the substitution of the referring classes the database for other, allowing the migration of the application among different databases.

They are presented in this work, forms of mapping the object-oriented model for relational model. These mappings happens of the model OO for the model relational. It is important to point out, that the architecture that it is being proposed, won't impede that structured applications let to have access to the relational database, because this approach was chosen to allow that new applications OO has access the relational database already existent.

As the implementation of this work follows the gateway approach, the concepts of object-oriented are presented, and as these they will be supported in the architecture, that is to say, which the gateway should implement.

Keywords: Object-Oriented, Relational Databases, Object Persistence.

1 Introdução

1.1 O Problema

O paradigma orientado a objetos está se tornando a abordagem preferida para a construção de sistemas de informação escaláveis e flexíveis. Sistemas de informação normalmente manipulam informações persistentes, isto é, informações que sobrevivem à execução dos programas que as criaram. A implementação de persistência em sistemas de informação pode ser provida de várias formas[CHA97].

Do ponto de vista do desenvolvimento de software, a solução mais simples para prover persistência é a utilização de um sistema de gerência de banco de dados OO (SGBDOO). Este implementa a persistência de forma transparente ao programador. As operações de carga e descarga de informações do banco de dados para a memória são executadas implicitamente. Entretanto, apesar de já existirem há vários anos, os SGBDOO tem tido apenas uma aceitação marginal na prática da indústria de software [STO96]. O predomínio continua sendo dos sistemas de gerência de banco de dados relacionais (SGBDR).

Este trabalho concentra-se em uma abordagem alternativa, a abordagem conhecida por *gateway*. Nesta abordagem o mapeamento entre o modelo OO e o modelo relacional é realizado por uma camada intermediária de software, o *gateway*.

Ao construir uma aplicação OO, é recomendável separar o tratamento da persistência das classes que representam o domínio de problema da aplicação [LAR98,MAR95]. Se o tratamento da persistência estiver codificado diretamente dentro das classes de domínio de problema, mudanças no projeto da base de dados relacional terão repercussão direta sobre as classes de domínio de problema. Por este motivo, aplicações OO construídas sobre SGBD relacional contém uma camada gerenciadora de dados, que faz o mapeamento OO-relacional e isola as classes de domínio de problema dos detalhes da base de dados.

Ao projetar uma base de dados voltada para a construção de aplicações OO, apresentam-se duas alternativas. A primeira é projetar a base de dados exclusivamente para o uso através das aplicações OO [REI96,RUM91]. Neste caso, a base de dados inclui construções artificiais, como OIDs (*object identifiers*), destinadas a simplificar o mapeamento OO-relacional. A outra alternativa é a de usar as regras clássicas de projeto de banco de dados [BAT92], com os conceitos usuais da abordagem relacional, como chaves primárias e estrangeiras. Esta alternativa tem a vantagem de permitir que a base de dados seja utilizada da forma usual por aplicações não OO. Além disso, é aplicável a bases de dados legadas, não projetadas para o uso com aplicações OO.

Também, o controle da carga/descarga de objetos para a base de dados pode ser implementado com variados graus de transparência para o usuário. A maior transparência é oferecida por interfaces do tipo ODMG [CAT97]. Neste tipo de

interface, todas operações de carga/descarga de objetos são transparentes ao usuário. No outro extremo, encontra-se uma interface na qual o usuário tem total controle das operações de carga/descarga, o que, se, por um lado, torna o desenvolvimento de aplicações mais complexo, simplifica a camada de gerência de dados.

1.2 Alternativas

Existem três abordagens principais para a persistência de objetos: a abordagem de persistência baseada em *gateway*, a abordagem com banco de dados objeto-relacional, e a abordagem com banco de dados orientado a objetos (também conhecida como abordagem de linguagem de programação persistente). Cada uma dessas três abordagens de persistência de objetos suportam certas classes de aplicações orientadas a objetos, e cada uma delas tem sido afetada pelos requisitos destas classes de aplicações por elas suportadas. A seguir, uma descrição sucinta destas abordagens é fornecida.

1.2.1 Persistência de Objetos baseada em *Gateways*

O *gateway* de persistência de objetos é uma camada de software inserida entre o sistema gerenciador de banco de dados e a aplicação orientada a objetos. São usados para dar suporte a um modelo de programação de aplicações OO, enquanto usam depósitos de dados não orientados a objetos para armazenar os dados de seus objetos. *Gateways* são comumente usados em casos onde os usuários querem desenvolver aplicações sobre dados armazenados em banco de dados não orientados a objetos usando modelos de programação orientados a objetos. Os objetos da aplicação tem um modelo de dados diferente do esquema de dados no banco de dados. Com isso, os *gateways* executam um mapeamento entre o esquema orientado a objetos da aplicação e o esquema não orientado a objetos utilizado no armazenamento dos dados. É função do *gateway* fazer, em tempo de execução, o processo de tradução entre modelos de forma transparente para o desenvolvedor.

A persistência de objetos através da abordagem de *gateway* é utilizada por muitos sistemas, onde se incluem o Persistence [PER96], o SMRC [REI96] , o VisualAge C++ Data Access Builder, UniSQL/M, Subtleware/SQL, entre outros. Como característica básica, esta abordagem mantém uma independência entre a aplicação e os dados.

1.2.2 Sistemas Gerenciadores de Banco de Dados Objeto-Relacional

Os SGBD's Objeto-Relacional são construídos sobre a premissa de que estender o modelo relacional é a melhor maneira para atender as mudanças nas novas aplicações orientadas a objetos. Estes bancos adicionam suporte para modelagem de dados orientada a objetos pela extensão do modelo relacional de dados e da linguagem de consulta, mantendo as tecnologias já consagradas dos bancos relacionais relativamente intactas. Existem duas classes de banco de dados objeto-relacional no mercado; aqueles que foram construídos a partir do zero, como o Illustra e o UniSQL, e aqueles que estão ou serão construídos pela extensão de bancos de dados relacionais do mercado, como o

DB2, Informix, Oracle e Sybase. Esta abordagem é essencialmente uma abordagem de baixo para cima, sendo centrada nos dados.

As atividades de padronização nesta área são baseadas na extensão do padrão SQL. O grupo X3H2, que é o comitê americano responsável pela especificação dos padrões do SQL, tem trabalhado nas extensões de objetos sobre o SQL desde 1991. Estas extensões fazem parte do novo projeto de padronização do SQL chamado de SQL3. O padrão SQL3 [ISO94] é uma tentativa de padronizar as extensões do modelo relacional e da linguagem de consulta.

1.2.3 Sistemas Gerenciadores de Banco de Dados Orientados a Objetos

Os bancos de dados orientados a objetos são basicamente construídos no princípio de que a melhor forma de adicionar persistência a um objeto é tornar persistentes os objetos que são usados nas linguagens de programação orientadas a objetos (OOLP) como o C++ e o Smalltalk. Como os bancos de dados orientados a objetos tem suas raízes nas linguagens de programação OO, eles são freqüentemente referenciados como sistemas de linguagem de programação persistente. Entretanto, estes bancos de dados vão muito além de simplesmente adicionar persistência a objetos de uma linguagem de programação. Isto porque, muitos SGBDOO foram construídos para servir ao mercado de projeto e manufatura ajudados pelo computador, as aplicações de CAD e CAM, aplicações estas onde características como acesso navegacional rápido, controle de versões e transações longas são extremamente importantes. Além disto, os SGBDOO suportam aplicações avançadas de banco de dados, proporcionando características como suporte a persistência de objetos de mais de uma linguagem de programação, distribuição de dados, modelos de transações avançados, versões, evolução de esquemas e geração dinâmica de novos tipos de dados. Embora muitas destas características tem pouco a ver com orientação a objetos, os bancos de dados orientados a objetos enfatizam-nas em seus sistemas e aplicações. Existem muitos sistemas de banco de dados orientado a objetos no mercado. Alguns exemplos são: Gemstone, Objectivity/DB, ObjectStore, Ontos, O2 entre outros. Esta abordagem é essencialmente uma abordagem de cima para baixo, sendo centralizada na aplicação ou na linguagem de programação.

Um padrão para os SGBDOO tem sido especificado pelo Object Database Management Group (ODMG). O ODMG é um consórcio que consiste principalmente de fabricantes de banco de dados. Este grupo especificou o padrão ODMG-93, publicado em forma de livro [ODM96]. O ODMG-93 define uma Linguagem de Definição de Objetos (ODL), uma Linguagem de Consulta de Objetos (OQL), e uma linguagem de mapeamento entre o C++ ou Smalltalk para a ODL e OQL. O ODMG está correntemente trabalhando na linguagem de mapeamento do Java para a ODL e OQL.

1.3 O Trabalho

Este trabalho apresenta a arquitetura de um *framework*, que segue a abordagem de *gateway*, para o desenvolvimento de aplicações OO sobre SGBD relacional. A arquitetura foi desenvolvida tendo em vista os seguintes princípios:

- As classes referentes a gerência de dados formam uma camada separada das classes que modelam o domínio de problema da aplicação, procurando tornar a camada domínio de problema independente do tipo de SGBD usado.
- Considera-se que a base de dados foi projetada usando as técnicas convencionais de projeto de base de dados relacional. Isso permite que a base de dados seja usada por outras aplicações não OO e que o *framework* seja usado para bases de dados legadas.
- O objetivo do *framework* não é o de substituir um SGBDOO, nem mesmo de substituir um *gateway* OO/relacional de grande porte. Um exemplo de um software deste tipo é OpenDM [ODM96] que faz o mapeamento ODMG para Adabas. O objetivo do *framework* é prover uma arquitetura simples que possa ser utilizada sem auxílio de outros softwares.
- Caches de objetos materializados serão mantidos pelo *framework*. Sua funcionalidade e forma de aplicação serão abordadas no trabalho.
- A arquitetura utiliza o conceito de materialização sob demanda, conhecido na literatura como *lazy-evaluation*, que indica que nem todos os objetos são materializados de uma vez, ou seja, uma determinada instância é materializada somente quando for necessário.
- Utilização do padrão de projeto *smart reference* [LAR98], com o objetivo de implementar a materialização sob demanda.

No trabalho foram usados padrões de projeto OO para a construção da arquitetura do *framework*, que serão abordados durante a descrição da arquitetura. O objetivo dos padrões dentro da comunidade de software é criar uma forma de literatura para auxiliar os desenvolvedores de software na resolução de problemas [APP97]. Os padrões ajudam a criar um linguagem compartilhada para a comunicação da experiência sobre estes problemas e suas soluções. Os padrões utilizados neste trabalho foram obtidos em [LAR98] e [GHJ95].

O trabalho está organizado como segue. O capítulo 2 discute alternativas de mapeamento OO-Relacional, descrevendo alguns problemas gerados pelo mapeamento. São tratados neste capítulo algumas regras de mapeamento que podem ser aplicadas na solução desta tarefa. O capítulo 3 apresenta as características da abordagem de persistência baseada em *gateways*, mostrando os conceitos de orientação a objetos e onde serão implementados, ou seja, o que o *gateway* deverá implementar. O capítulo 4 apresenta a estrutura geral de um *gateway* projetado segundo a abordagem aqui proposta. Especificamente, mostra que serviços o *gateway* provê a seus usuários. E finalmente, o capítulo 5 apresenta a estrutura interna de um *gateway* construído segundo a abordagem proposta, mostrando a comunicação entre seus componentes. São descritos também, alguns dos padrões de projeto utilizados no desenvolvimento da arquitetura.

2 Mapeamento OO-Relacional

Este capítulo apresenta mapeamentos de esquemas orientados a objetos para esquemas relacionais. Neste trabalho, este mapeamento acontece do modelo OO para o modelo relacional. Cabe ressaltar, que a arquitetura que está sendo proposta, não vai impedir que aplicações estruturadas, ou seja, do paradigma relacional, deixem de ter acesso a base de dados relacional mapeada. Esta abordagem foi escolhida para permitir que novas aplicações OO tenham acesso a bases de dados relacionais já existentes.

A integração destes dois ambientes, linguagem de programação OO e SGBD relacional, gera a necessidade do mapeamento. O problema origina-se na lógica que os desenvolvedores devem usar, para que aplicações com tabelas relacionais possam conter objetos [LOO95].

Alguns tipos de dados podem ser armazenadas em um banco de dados relacional. Se uma aplicação precisa armazenar outros tipos de dados, como por exemplo, os tipos de dados definidos pelo usuário, estes dados podem ser armazenados em um campo "BLOB", ou escrito em um arquivo, com uma referência ao arquivo armazenado em uma tabela apropriada no banco de dados. Os dados escritos no arquivo, neste caso, não estão sujeitos as regras do banco de dados relacional, como os controles de concorrência, gerenciamento de transação, recuperabilidade, controle de acessos e assim por diante. A aplicação tem responsabilidade pela lógica embutida no objeto que está em um campo "BLOB".

Pelo fato dos SGBD relacionais não gerenciarem modelos de objetos, o programador deve escrever uma quantia grande de linhas de código, necessárias para compatibilizar o SGBDR com estruturas semânticas como heranças e relacionamentos. Considere um exemplo simples de modelo de objeto, mostrado na figura 2.1 abaixo, na qual encontram-se dois tipos de empregados: o horista e o mensalista. Cada um tem seus atributos específicos, e alguns atributos são comuns para ambos. Para simplificar, ignore as operações neste momento. As duas tarefas básicas de mapeamento são:

1. Projetar as tabelas para representar os dados.
2. Escrever linhas de código para transformar o objeto "Empregado" em linhas de tabelas e vice-versa.

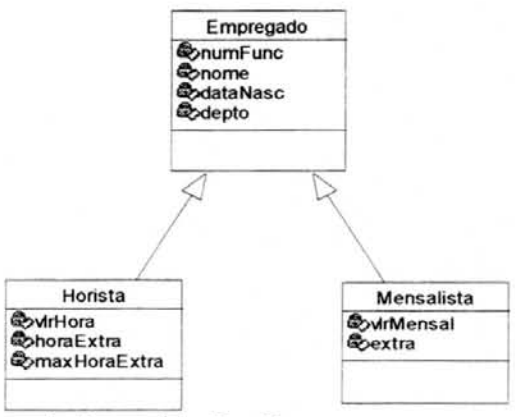

FIGURA 2.1 - Um exemplo de modelo de objetos

Existem muitas abordagens alternativas. A lógica e o código para transformar objetos em tabelas e vice-versa, são determinadas por qual das alternativas será usada para nivelar e particionar os objetos em tabelas.

2.1 Alternativas de mapeamento

Conforme mencionado anteriormente, um banco de dados relacional pode ser visto como uma coleção de tabelas. Ao se mapear um esquema OO para um esquema relacional, os objetos de uma classe correspondem às linhas de uma tabela, e os atributos definidos para estes objetos correspondem às colunas de uma tabela.

A fim de mapear uma aplicação baseada no modelo OO para tabelas relacionais, deve-se escolher entre algumas alternativas de mapeamento. É necessário fornecer detalhes que não pertencem ao esquema OO, como chaves primárias para cada tabela, e atribuir um domínio para cada um dos atributos dessas tabelas. As alternativas consideradas são:

1. **Mapeamento de classes de objetos para tabelas** - cada classe pode ser mapeada para uma ou mais tabelas. Da mesma forma, uma tabela pode corresponder a mais de uma classe.

A forma mais utilizada é mapear diretamente uma classe para uma tabela relacional. Um mapeamento mais complexo, aquele que representa uma junção entre muitas tabelas, pode dificultar sua instanciação, e impossibilitar sua alteração. Nas próximas seções estão listadas algumas regras que podem ajudar nesta decisão. Duas exceções a este conceito básico incluem a criação de *projection objects* para minimizar o tráfego na área de trabalho e *view objects* para suporte a decisões[LOO95].

Projection Objects : para tabelas com muitas colunas, pode ser eficiente criar-se uma projeção da tabela para uma classe projeção, e recuperar o total de colunas apenas quando necessário. Por exemplo, uma linha de um tabela de Clientes pode conter 50 colunas, mas apenas o nome e o número do telefone são necessárias em um determinado momento.

View Objects : para aplicações que suportam decisões, se tornará muitas vezes necessário criar uma tabela visão, que representa uma junção do banco de dados, e mapear esta visão para um objeto da classe. Isto permite segurar total vantagem da álgebra relacional e ocultar o modelo de dados físicos da aplicação OO.

2. **Mapeamento de relacionamentos para tabelas** - um relacionamento entre classes pode ou não ser mapeado para uma única tabela, dependendo da sua cardinalidade, bem como das preferências do projetista em relação à extensibilidade, número de tabelas e desempenho.

Todo relacionamento "muitos-para-muitos" entre classes deve sempre ser mapeado para uma tabela separada, a fim de satisfazer à 1FN. A concatenação das chaves primárias de ambas as classes relacionadas se torna a chave primária dessa nova tabela.

Um relacionamento "um-para-muitos" entre classes pode ser mapeado para um tabela separada, ou pode ser "embutido" como chave estrangeira na tabela, para a classe do lado "muitos" do relacionamento.

Um relacionamento "um-para-um" entre classes pode ser mapeado para uma tabela separada, ou pode ser "embutido" como chave estrangeira em uma das classes que participam do relacionamento.

Com isso, o mapeamento dos relacionamentos pode ser das seguintes maneiras:

Tabelas Distintas
Ao utilizar esta forma de mapeamento, o relacionamento entre classes deve ser representado como uma tabela distinta no banco de dados. Esta abordagem proporciona maior flexibilidade de modelagem, uma vez que, a inclusão e a remoção de novos relacionamentos se tornem transparentes para as outras tabelas.

Chave Estrangeira Embutida
Esta é a maneira mais comum de tratar relacionamentos 1-1 e 1-N. Neste caso, para uma dada classe, a chave primária da classe referenciada, é embutida na própria classe.

Tabelas Embutidas
Neste caso, duas classes são incorporadas em uma única tabela. Cada incorporação melhora performance da aplicação, e em contrapartida, aumenta o custo de extensibilidade, além de violar a normalização.

Conforme a tabela 2.1, dependendo da cardinalidade do relacionamento e da necessidade de rapidez versus flexibilidade, diferentes técnica de mapeamento de relacionamento são apropriados. Na tabela abaixo, eficiência significa que a aplicação apresenta melhor performance de execução, a flexibilidade indica que o desenvolvedor apresenta facilidades na modelagem, e a opção N/A não se aplica na situação em questão.

TABELA 2.1 – Técnicas de mapeamento de relacionamentos.

Tipo de Relacionamento	***Tabela Embutida***	***Chave Estrangeira Embutida***	***Tabelas Distintas***
1-1	Eficiente	Flexível	Ineficiente
1-N	Ineficiente	Eficiente	Flexível
N-N	N/A	N/A	Eficiente e Flexível

3. **Mapeamento de generalizações para tabelas** - Assim como nos relacionamentos, a maneira como realizamos os mapeamentos de herança de objetos para o banco de dados pode ter um impacto na performance.

Uma super classe pode ser abstrata ou concreta. Uma super classe abstrata não precisará ser instanciada e consequentemente não tem nenhum dado físico associado a ela, ou seja, não há nenhuma tabela correspondente. Uma super classe concreta é uma que pode existir em si própria, e terá com isso sua própria tabela associada.

Existe várias formas de mapear herança para uma tabela relacional:

Partição Vertical
Neste tipo de mapeamento cada tipo de classe do modelo de objeto é mapeado para uma tabela correspondente. A desvantagem disto é que irá criar uma hierarquia de herança a qual será exigido muitos caminhos para recuperação de informações básicas de objetos.

Partição Horizontal
Neste caso, apenas classes folhas são mapeadas para tabelas incluindo todos os seus atributos herdados. Esta maneira pode dar uma melhor performance desde que apenas uma tabela necessite ser acessada para instâncias de uma dada classe folha. Os dados da super classe estarão presentes em ambas as tabelas.

Partição Tipada
Outro forma de manipular herança é mapear todas as classes de uma árvore de herança para uma tabela simples usando uma coluna para distinguir as subclasses. Isto possibilita transformar objetos de múltiplas classes em uma simples classe, mas causa violação de normalização.

Consultas a nível da super classe devem ser implementadas muito diferentemente, dependendo do mapeamento de herança. Se um super classe é concreta, uma consulta a super classe pode ser eficientemente manipulada pela consulta a tabela correspondente a super classe. Se uma super classe é abstrata, a consulta deve ser executada repetidamente em cada tabela correspondente a subclasse.

Por outro lado, recuperar uma instância de uma subclasse é muito rápido se todas as super classes forem abstratas, pois todas as colunas necessárias são alocadas na tabela correspondente a subclasse. Subclasses com super classes concretas podem necessitar de pesadas junções para a recuperação de instâncias.

Como mostra a tabela 2.2 a seguir, dependendo se a herança é a partir de uma super classe abstrata ou concreta, diferentes técnicas de mapeamentos de herança são apropriadas. Na tabela abaixo, eficiência significa que a aplicação apresenta melhor performance de execução, e flexibilidade indica que o desenvolvedor apresenta facilidades na modelagem e manutenção do modelo.

TABELA 2.2 – Técnicas de mapeamento de heranças.

Tipo de Herança	*Partição Horizontal*	*Partição Vertical*	*Partição Tipada*
Super classe Abstrata	Busca eficiente na subclasse	Busca ineficiente na subclasse	Eficiente mas inflexível
Super Classe Concreta	Ineficiente consulta na super classe	Eficiente consulta na super classe	Eficiente mas inflexível

2.2 Problemas gerados pelo mapeamento

Como pode ser visto, existem diversas formas de se realizar o mapeamento entre os modelos. De qualquer forma, esta é uma tarefa que provocará algumas dificuldades no momento da codificação da aplicação, como a perda de informações semânticas do modelo, a necessidade de se escrever muitas linhas de códigos de transformação entre modelos, e a complexidade na leitura de dados a partir de tabelas [LOO95]. Estes problemas devem ser resolvidos pelo *gateway*, tirando esta responsabilidade do desenvolvedor da aplicação. A seguir, estes problemas serão abordados com mais detalhe.

2.2.1 Perda de informações semânticas.

Utilizando-se a abordagem mais natural, ou seja, mapear objetos para tabelas com uma tabela por classe, o objeto da figura 2.1 mostrada anteriormente pode ser representada por três tabelas:

Empregado (numFunc, nome, dataNasc, depto)
Horista (numFunc, vlrHora, horaExtra, maxHoraExtra)
Mensalista (numFunc, vlrMensal, extra)

Os campos participantes da chave primária são sublinhados em cada definição de tabela. Um pré-requisito para este mapeamento é que os tipos de dados nas classes devam ser suportados pelos tipos de dados do SGBD relacional.

O primeiro problema com esta representação é que a informação semântica está sendo perdida no mapeamento. A estrutura de herança hierárquica desapareceu. A partir das três tabelas não fica claro se existe alguma que representa um supertipo e quais seriam as subtipos. Não é razoável esperar que o SGBD deduza este sentido a partir dos nomes das tabelas. Há uma possibilidade de que as tabelas pudessem ser nomeadas como Emp_A, Emp_B, Emp_C, e não ficaria claro a existência de atributos herdados de outras tabelas.

Utilizando-se uma segunda abordagem, que mapeia cada classe folha para uma tabela, o problema ainda permanece. Os atributos da superclasse são então explicitamente movidos para as subclasses. Com isso, duas tabelas são necessárias:

Horista (numFunc, nome, dataNasc, depto, vlrHora, horaExtra, maxHoraExtra)
Mensalista (numFunc, nome, dataNasc, depto, vlrMensal, extra)

Este mapeamento também oculta a estrutura da herança hierárquica, porque não é deixado claro que os tipos empregado horista e o empregado mensalista são subtipos de um tipo genérico. O programador deve codificar a lógica na aplicação manuseando consistentemente as partes do supertipo nas duas tabelas.

Sob outro enfoque, utilizando-se uma outra abordagem, que seria a quebra da estrutura de herança em uma tabela simples, o mapeamento do exemplo ficaria:

Empregado (numFunc, nome, dataNasc, depto, vlrHora,
horaExtra, maxHoraExtra, vlrMensal, extra)

Aqui, o programador novamente arca com a responsabilidade de corretamente designar valores aos atributos apropriados. Dependendo do tipo do objeto Empregado a ser persistente, alguns atributos deverão ser nulos e outros serão atribuídos com valores não nulos. Por exemplo, um empregado mensalista tem seu valor de hora nulo, mas seu valor mensal não é nulo.

2.2.2 Necessidade de escrever muitas linhas de código de transformação.

De acordo com o modelo de objetos, o que se observa é que Empregado é uma superclasse, e que empregado horista e empregado mensalista são subclasses que herdam atributos de empregados. Devido ao SGBD relacional não suportar herança, esta lógica terá que ser programada dentro do código da aplicação. O esforço vinculado a esta implementação está demonstrado no exemplo a seguir, que foi codificado em C++, para criar um objeto persistente de Empregado em um banco de dados relacional. Foram usadas no exemplo três tabelas, uma para cada classe do modelo de objetos. Com isso, a aplicação precisa diferenciar os tipos de empregados e inserir as linhas nas tabelas apropriadas.

```
/* Os valores dos atributos de empregados já estão no objeto C++ */

EXEC SQL INSERT INTO Empregado
      (numFunc, nome, dataNasc, depto)
      VALUES (:emp->numFunc, :emp->nome, :emp->dataNasc, :emp->depto);
if (emp->type==1)
  EXEC SQL INSERT INTO Horista
      (numFunc, vlrHora, horaExtra, maxHoraExtra)
      VALUES (:emp->numFunc, :emp->vlrHora, :emp-> horaExtra, :emp->
      max_HoraExtra);
```

```
else if (emp->type==2)
    EXEC SQL INSERT INTO Mensalista
          (numFunc, vlrMensal, extra)
           VALUES (:emp->numFunc, :emp->vlrMensal, emp->extra);
EXEC SQL COMMIT WORK RELEASE;
```

Lendo o código acima, nota-se que existem muito mais linhas de código do que a necessária para armazenar o mesmo objeto empregado em C++ em um banco de dados orientado a objetos.

```
Transaction addEmp;
Ref<empregado> emp = new (database) Empregado;
/*obtém os valores do objeto emp  usando C++*/
addEmp.commit( );
```

Com o armazenamento em banco de dados relacional, o programador deve codificar a lógica dentro da aplicação corretamente, e assegurar que os campos estejam adequadamente preenchidos, o que não ocorre com os SGBDs orientados a objetos, pois estes compartilham com o C++ do mesmo modelo de hierarquia.

2.2.3 Complexidade na instanciação de objetos a partir de tabelas.

Uma outra parte do problema é escrever a lógica para trazer dados do banco de dados relacional ao ambiente de programação orientada a objetos. Continuando com o mesmo exemplo, porém usando o mapeamento de hierarquia quebrada, com todas as subclasses representadas em uma única tabela, o código para carregar os dados dos empregados do departamento D1 a partir de um banco de dados relacional inclui o seguinte fragmento:

```
Departamento *d=new Departamento;
d->nome= "D1"
Empregado  *temp_emp=new Empregado;
EXEC SQL DECLARE c1 CURSOR
      SELECT *
      FROM Empregado
      WHERE depto_nome= "D1";
EXEC SQL WHENEVER NOT FOUND GOTO end_of_table;
for(;;)                  /* busca linha até o not_found ser feito */
{
     EXEC SQL FETCH c1
     INTO : temp_emp;
     if temp_emp->emp_type = = 1
     {
       Horista *h=new Horista;
       h->numFunc = temp_emp->numFunc;
       h->nome = temp_emp->nome;
```

```
        h->dataNasc = temp_emp->dataNasc;
        h->dept_nome = d;
        h->vlrHora = temp_emp->vlrHora;
        h->horaExtra = temp_emp->horaExtra;
        h->maxHoraExtra = temp_emp->maxHoraExtra;
    }
    else if temp_emp->emp_type  = =2
    {
        Mensalista *s = new Mensalista;
        s->numFunc = temp_emp->numFunc;
        s->nome = temp_emp->nome;
        s->dataNasc = temp_emp->dataNasc;
        s->dept_nome = d;
        s->vlrMensal = temp_emp->vlrMensal;
        s->extra  = temp_emp->extra ;
    }
    /* o código fará  alguma coisa com o objeto */
}
```

Um bom número de linhas são necessárias para tratar corretamente o tipo de informação que está armazenada nos atributos da tabela do banco de dados relacional. Cada linha da tabela de empregado é lida para uma instância temporária da classe empregado, cuja estrutura corresponde a sua linha. O tipo de um empregado é usado para determinar como instanciar a hierarquia de classes. Cada objeto empregado contém um ponteiro para o objeto departamento D1, bem como um valor para cada atributo específico.

Se as informações do empregado fossem instanciadas a partir de um objeto persistente em um banco de dados orientado a objetos, a lógica seria reduzida para:

```
Set <Ref<Empregado>> D1_Dept_Empregado;
oq1(D1_Dept_Empregado,  "select ee from ee in  Empregado \
                              where ee.dept.nome = "D1"  ");
/* o código faz alguma coisa com o objeto, interagindo sobre o conjunto */
```

Cada um dos objetos Empregado do banco de dados é automaticamente tipado corretamente para o ambiente C++, porque o C++ e o SGBD orientado a objetos compartilham dos mesmos tipos. Empregado e seus subtipos são reconhecidos em ambos os ambientes, C++ e SGBD. Cada objeto lido a partir do banco de dados mantém sua informação de tipo, como um objeto do C++.

2.3 Regras de Mapeamento

Esta seção tem como objetivo identificar todos os casos que podem ocorrer durante o mapeamento entre os modelos, e juntamente descrever quais as regras de

mapeamentos a serem adotadas em cada caso. Sejam **A** e **B** duas classes definidas no modelo de dados OO, os seguintes casos podem ocorrer:

Caso 1: A é uma classe onde todo atributo possui tipo atômico, conforme é mostrado na figura 2.2.

Regra 1: "A classe **A** é mapeada para uma esquema **A** com todos os seus atributos. A determinação da chave primária do esquema **A** ocorrerá conforme sua regra específica."

Regra da Chave Primária: "O atributo da classe **A** que tiver sido especificado pelo projetista como sendo o identificador, constituirá a chave primária para esse esquema. Se a classe **A** estiver envolvida numa hierarquia de herança, sendo subclasse de uma outra classe **S**, então **A** herda os atributos e métodos de sua superclasse. Isso equivale ao conceito de generalização do modelo ER [LAE93] e, portanto, será mapeado para o modelo relacional de maneira equivalente. Nesse caso, a chave primária do esquema **A** será a mesma chave primária definida para o esquema **S,** mantendo-se sua integridade referencial. A opção de remoção para esse atributo, que constitui uma chave estrangeira para a superclasse, é a propagação."

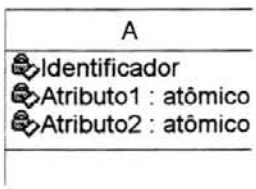

FIGURA 2.2 - Regras de Mapeamento - Caso 1

Esquema da Relação A :
A(Identificador, Atributo1, Atributo2)

Caso 2: A é uma classe que possui algum atributo de tipo estruturado, que constitui uma referência a uma instância de uma outra classe **B,** como na figura 2.3.

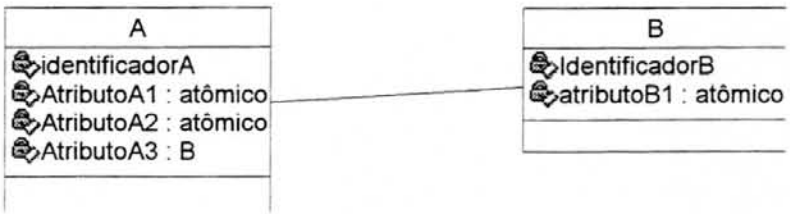

FIGURA 2.3 - Regras de Mapeamento - Caso 2

Esquema da Relação A :
A(identificadorA, atributoA1, atributoA2,atributoA3)

Regra 2: "A classe **A** é mapeada para um esquema **A** que terá esse atributo embutido, sendo garantida a sua restrição de integridade referencial, tendo como opção de remoção a substituição por nulos. A determinação da chave primária do esquema **A** ocorrerá conforme sua regra específica."

Caso 3: A é uma classe que possui algum atributo de tipo estruturado, que constitui uma referência a um conjunto de instâncias de uma outra classe **B,** como na figura 2.4.

Regra 3: "Uma tabela separada deverá ser criada para capturar o atributo estruturado da classe **A**, tendo como chave primária a composição das chaves primárias dos esquemas **A** e **B,** sendo garantidas as suas restrições de integridade referencial, com opções de remoção de propagação e bloqueio, respectivamente. A determinação da chave primária de **A** ocorrerá conforme sua regra específica."

FIGURA 2.4 - Regras de Mapeamento - Caso 3

Esquema da Relação A :
A(identificadorA, atributoA1, atributoA2)
Nova tabela criada :
A_B(identificadorA, identificadorB)

Caso 4: **A** é uma classe que possui algum atributo de tipo estruturado, que constitui uma referência a uma instância de uma classe **B,** e esse atributo foi especificado como invertido.

Caso 4.1: A classe B possui também um atributo de tipo estruturado, que constitui uma referência inversa a uma instância da classe **A**, como na figura 2.5.

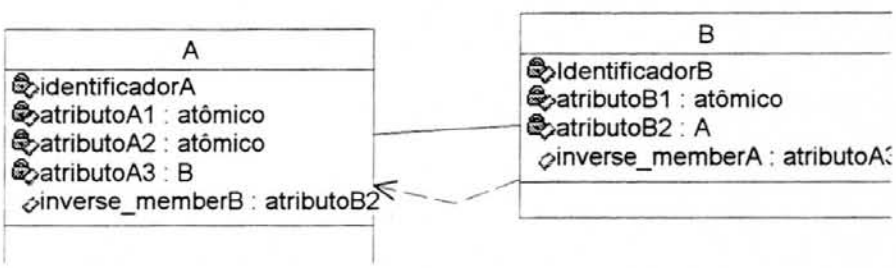

FIGURA 2.5 - Regras de Mapeamento - Caso 4.1

Esquema das Relações :
A(identificadorA, atributoA1, atributoA2,atributoA3)
B(identificadorB, atributoB1)

Regra 4.1: "Nesse caso, as classes **A** e **B** estão envolvidas num relacionamento "um-para-um" que será representado por uma única tabela. Uma das classes participantes do relacionamento, **A** por exemplo, deverá conter após o mapeamento para o esquema **A,** além de seus próprios atributos, um atributo embutido que constitui uma chave estrangeira para o outro esquema, **B** no caso. Será garantida a restrição de integridade referencial para esse atributo, tendo como opção de remoção a substituição por nulos, o qual constituirá uma chave alternativa para o esquema **A**. A determinação da chave primária para essas tabelas ocorrerá conforme sua regra específica."

Caso 4.2: A classe **B** possui também um atributo de tipo estruturado, que constitui uma referência inversa a um conjunto de instâncias da classe **A**, como na figura 2.6.

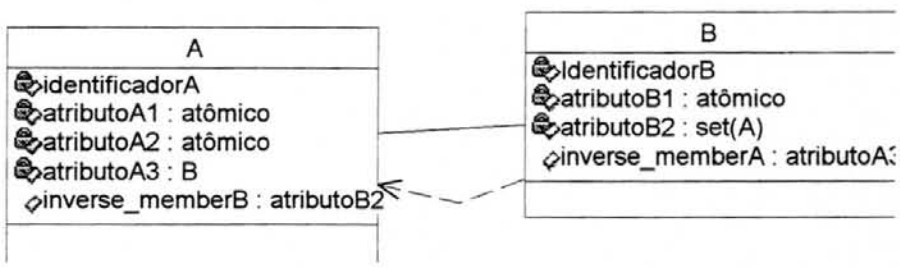

FIGURA 2.6 - Regras de Mapeamento - Caso 4.2

Esquema das Relações :
A(identificadorA, atributoA1, atributoA2,atributoA3)
B(identificadorB, atributoB1)

Regra 4.2: "Nesse caso, as classes **A** e **B** estão envolvidas num relacionamento "um-para-muitos" que será representado por uma tabela única. A classe **A** deverá conter após o mapeamento para um esquema **A,** além de seus próprios atributos, um atributo embutido que constitui uma chave estrangeira para o esquema **B,** sendo garantida a sua restrição de integridade referencial, tendo como opção de remoção a substituição por nulos. A determinação da chave primária para essas tabelas ocorrerá conforme sua regra específica."

Caso 5: A é um classe que possui algum atributo estruturado, que constitui uma referência a um conjunto de instâncias de uma classe **B**, e **B** possui também um atributo

estruturado, que é uma referência inversa a um conjunto de instâncias de **A**, como na figura 2.7.

Regra 5: "As classes **A** e **B** estão envolvidas num relacionamento "muitos-para-muitos" que será representado por uma tabela separada, tendo como chave primária a composição das chaves primárias dos esquemas A e B. Restrições de integridade referencial deverão ser garantidas para os atributos dessa nova tabela, tendo como opções de remoção a propagação. A determinação da chave primária para essas tabelas ocorrerá conforme sua regra específica."

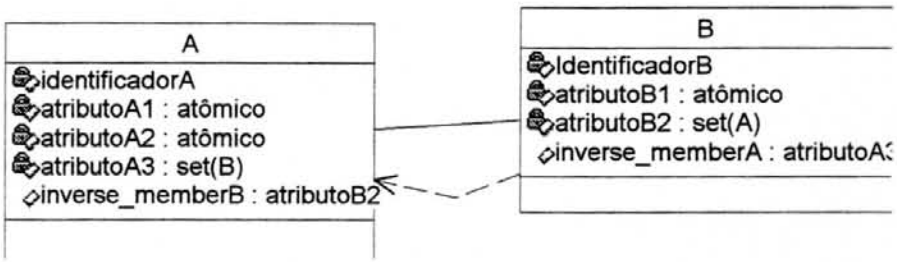

FIGURA 2.7 - Regras de Mapeamento - Caso 5

Esquema das Relações :
A(identificadorA, atributoA1, atributoA2)
B(identificadorB, atributoB1)
Nova tabela criada :
A_B(identificadorA, identificadorB)

O mapeamento entre o modelo de objetos e o modelo relacional que está sendo considerado na arquitetura do *framework* proposto, segue a abordagem do padrão de Representação de Objetos como Tabelas [BRO96], que propõem a definição de uma tabela para cada classe de objeto persistente, e com os atributos dos objetos mapeados para colunas. Se um objeto possui somente atributos com tipos primitivos de dados, o mapeamento fica altamente facilitado. Sendo assim, a forma de mapeamento será:

- Cada tabela corresponde a uma classe homônima.
- Cada atributo não chave estrangeira corresponde a um atributo da classe correspondente a tabela. Para cada atributo da classe existem métodos "get" e "set" que servem para atribuir e recuperar o valor do atributo.
- Cada chave estrangeira corresponde a referências bi-direcionais entre os objetos relacionados.

3 Formas de Suporte aos Conceitos de OO.

Como a implementação deste trabalho segue a abordagem de *gateway*, o objetivo deste capítulo é situar o leitor quanto as diferenças que existem na implementação, quando se utiliza a abordagem de persistência através de *gateway* em relação as demais abordagens. Para isto, serão apresentados os conceitos de orientação a objetos, e como estes serão suportados na arquitetura, ou seja, o que o gateway deverá implementar.

As aplicações OO que são programadas nas linguagens de programação OO existentes, como C++ e Smalltalk, usam características de modelagem como encapsulamento, herança e *binding* dinâmico. Estas características são assumidas como familiares em todas aplicações que utilizam essas linguagens de programação OO. Mesmo que os SGBDOO não suportem todas as características nativas disponíveis em uma linguagem de programação OO, os modelos por eles suportados são muito mais complexos que os modelos de dados suportados pelos banco de dados relacionais tradicionais. Entre estes dois modelos, os bancos de dados objeto-relacional começam a suportar muitas destas características.

Um dos principais problemas que os SGBDOO resolvem por suportar o modelo de dados de uma linguagem de programação OO é o problema de casamento de impedância[BDK92], problema que existe nos SGBD relacionais, onde o modelo de dados usado na aplicação é diferente do modelo de dados usado pelo banco de dados. Esta diferença de modelos de dados causam dois problemas para as aplicações:

1. Um programador de aplicações deve programar em duas linguagens diferentes com sintaxes, semânticas e tipos distintos. Usa uma linguagem de programação como C++ ou Pascal OO, e uma linguagem de manipulação de dados do SGBD, ou seja, o SQL. A lógica da aplicação é implementada usando uma linguagem de programação, enquanto o SQL é usado para criar e manipular dados no banco de dados.

2. Quando qualquer dado é recuperado da base relacional, ele deve ser traduzido de sua representação de banco de dados para uma representação específica da linguagem de programação da aplicação na memória. Da mesma forma, qualquer atualização que necessite ser feita nos dados deve ser explicitamente comunicada ao banco de dados usando outro comando SQL, fazendo com que os dados devam ser traduzidos da representação da memória de volta para a representação de banco de dados. Toda esta comunicação de ida e volta entre a base de dados e a aplicação gera um processamento desnecessário que poderia ser eliminado se o mesmo modelo de dados fosse utilizado na aplicação e na base de dados.

Os bancos de dados orientados a objetos reduzem os problemas acima descritos, fornecendo um suporte completo as características de modelagem de dados de qualquer linguagem de programação orientada a objetos. Nestes bancos, o modelo de dados que é usado pela aplicação é idêntico ao modelo usado pelo banco no armazenamento dos dados. Com isso, eles tem obtido bons resultados na solução do segundo problema mencionado acima. Porém, eles tem tido pouco sucesso na solução do primeiro problema, especialmente quando consultas a objetos estão envolvidas.

Os bancos de dados objeto-relacionais incorporaram características dos modelos de dados existentes nas principais linguagens de programação, porém, o modelo de dados usado na aplicação ainda não é idêntico ao modelo usado pelo banco de dados. Com isso, não solucionaram totalmente o segundo problema. Em relação ao primeiro problema, facilidades foram incorporadas, especialmente quando consultas a objetos estão envolvidas.

A persistência de objetos através de *gateway* pretende reduzir estes problemas através de ferramentas mapeadoras de esquema e geração automática de código. A intenção é dar ao usuário a impressão de estar trabalhando com apenas um modelo de dados, o modelo de dados usado na aplicação. Existe apenas um usuário para o qual isto não é válido, que é aquele usuário que define o mapeamento entre os modelos de dados da aplicação e do banco de dados. Com isso, para o usuário, é como se um banco de dados orientado a objetos estivesse sendo usado.

Como podemos ver, muitas questões complexas aparecem no fornecimento de suporte a modelos de dados orientados a objetos. Por isso, serão analisadas como e onde estas características serão suportadas. Veja a tabela 3.1 a seguir:

TABELA 3.1 – Forma de suporte as características OO.

Característica	Aplicação Orientada a Objetos	Gateway de Persistência de Objetos	Sistema Gerenciador de Banco de Dados Relacional
Identificador de Objetos (OID)	Gerado pela aplicação quando se instância um objeto na memória	Não está envolvido	Deve ser implementado através de atributo
Objetos Complexos	Suportado pela linguagem OO	Deve executar o mapeamento	Tabelas normalizadas no banco de dados
Objetos Compostos	Implementado pela linguagem sob forma de heranças	Deve executar o mapeamento	Tabelas normalizadas no banco de dados
Relacionamento	Implementado pela linguagem sob forma de associação	Deve executar o mapeamento	Tabelas normalizadas no banco de dados
Encapsulamento	Suportado pela linguagem OO	Não está envolvido	Não suporta
Herança	Suportado pela linguagem OO	Deve executar o mapeamento	Não suporta
Sobrescrita e Sobrecarga de métodos	Suportado pela linguagem OO	Não está envolvido	Não suporta

3.1 Identificador de objetos (OID)

A identidade do objeto é uma questão importante que necessita ser gerenciada na persistência de um objeto. Em um programa em tempo de execução, objetos podem ser criados, copiados, removidos e acessados. Uma vez que nenhum destes objetos transientes persistem além da vida de um processo, o endereço de memória de um objeto em um processo pode ser usado como uma identificação (ID) de um objeto (OID). Em banco de dados, OIDs, como dados, devem ser persistentes. Um OID por definição, refere-se a exatamente um objeto na base de dados. A referência a um objeto com o mesmo OID em aplicações diferentes, refere-se no banco de dados ao mesmo objeto.

Bancos de dados não orientados a objetos também tratam o problema do identificador de objeto (ou identificador de registro), porém usualmente eles o obtém através de um identificador baseado em valor (chave primária). Bancos de dados relacionais suportam tipicamente acessos a dados persistentes baseados em valores, isto é, se uma aplicação necessita acessar a uma linha particular em um base de dados, ela deve consultar a base usando o nome da relação onde a linha se encontra e o valor de uma chave primária para a linha na tabela. Esta forma de acesso a dados persistentes é inadequada em aplicações orientadas a objetos, uma vez que objetos podem ter chaves primárias idênticas mas serem objetos diferentes. Exemplificando, dois empregados podem ter seus próprios carros, com o mesmo modelo e ano, mas cada objeto carro respectivamente não pode ser compartilhado entre os dois empregados, resultando que existe dois objetos carros com valores idênticos, sendo que cada um deles está associado a cada objeto empregado.

Em um *gateway*, a identificação de objetos será sempre limitada pelo banco de dados ou sistema de arquivos que armazenará os dados da aplicação. O identificador do objeto deverá ser mapeado para um atributo da tabela, normalmente a chave primária.

Alguns bancos de dados objeto-relacionais tratam o acesso através de OID, além do tradicional acesso baseado em valor de chave. Um método para o suporte de OID em SGBDOR é a criação de um identificador único para toda linha no banco de dados, independente do valor contido nesta linha. Qualquer linha em qualquer tabela dentro destes bancos podem ser acessados diretamente usando-se o identificador único da linha.

3.2 Objetos Complexos

Um mecanismo de objetos complexos permite que um objeto contenha atributos que sejam também objetos. Em outras palavras, o esquema de um objeto não está na primeira forma normal, diferente das tuplas do modelo relacional. Os componentes de uma tupla são somente de tipos simples, como inteiro, caracter ou "BLOB". Um exemplo de uma definição em C++ de um objeto complexo é:

```
class Pessoa {
        char *nome;
        int idade;
```

```
    Carro carro;
    Set<Pessoa> filhos;
    List<string> telefones;
    Set<Pessoa> mesma_idade;
}
```

No exemplo acima, uma instância da classe Pessoa é uma objeto complexo, que contém como atributos, dois atributos básicos (nome e idade), um objeto da classe Carro embutido, um conjunto de objetos da classe Pessoa como filhos, uma lista de caracteres strings como telefones, e outro conjunto da classe Pessoa com objetos pessoas da mesma idade, onde todos estes objetos estão embutidos.

Os objetos que são definidos dentro de outros objetos, por exemplo, o atributo carro dentro da classe Pessoa, fazem parte inteiramente dos objetos que os contém, e não podem possuir seus próprios identificadores únicos. Componentes de objetos complexos são automaticamente criados (recursivamente) quando o objeto de nível mais alto é criado, e são automaticamente removidos quando o objeto que os contém é removido. A conexão entre os objetos complexos e seus componentes é sempre válida e não pode ser removida. Com isso, objetos complexos diferem de objetos compostos, que serão descritos a seguir.

Nos *gateways,* objetos complexos do modelo de dados de uma aplicação necessitam ser mapeados para os dados armazenados no banco de dados. Qualquer mapeamento é normalmente facilitado por algum gerador específico de código de aplicação, que pode fazer uma tradução entre o modelo de dados da aplicação e o modelo de dados do banco de dados. Por exemplo, considere as duas soluções a seguir, para o armazenamento de um conjunto de atributos de tamanho fixo de um objeto complexo sobre um banco de dados relacional:

1. Armazenar os elementos do conjunto em uma tupla com uma coluna para cada elemento do conjunto (múltiplas colunas poderiam ser necessárias para um elemento, caso este fosse também um objeto complexo).
2. Armazenar o conjunto de elementos em tabelas separadas, com cada tupla desta tabela armazenando um elemento simples, juntamente com seu índice neste conjunto.

Outra solução que poderia ser proposta para este problema seria armazenar o objeto complexo em um campo do tipo "BLOB" (binary large object), abordagem que foi utilizada na proposta SMRC da IBM[REI96].

Bancos de dados objeto-relacional fornecem extensões ao modelo relacional para suportar dados que não estejam na primeira forma normal, tais como as listas, conjuntos, etc. Estas extensões no modelo de dados podem ser usadas para definir objetos complexos dentro de uma aplicação.

Bancos de dados orientados a objetos são extremamente fortes no suporte a objetos complexos uma vez que são baseados nas linguagens de programação OO, que por sua vez, possuem extensas características para definição de objetos complexos no modelo de dados.

3.3 Objetos Compostos

Objetos compostos são objetos individuais que são relacionados e formam parte de um grupo. Aplicações OO utilizam este conceito, como um grupo de objetos que fazem parte de um objeto pai, que é normalmente uma coleção. Um objeto composto deve ser tratado como sendo um objeto particular, e que pode ser apontado por outros objetos simples. Um ponteiro, de um objeto apontador para um objeto apontado, é uma forma de relacionamento especial. Um exemplo de uma aplicação onde o suporte a objetos compostos é necessário é mostrado a seguir.

Um objeto Carro deve consistir de um conjunto de objetos Motor, Rodas, Chassis, etc. É muito provável que o objeto Motor permaneça inalterado para muitos modelos de Carros, enquanto que o desenho do Chassis apresenta diferenças em cada modelo. Neste caso, é necessário tratar algumas partes do carro e seus componentes como uma entidade, enquanto se compartilha outros componentes com outros carros. Um exemplo de código é mostrado a seguir, junto com a figura 3.1 que ilustra o problema.

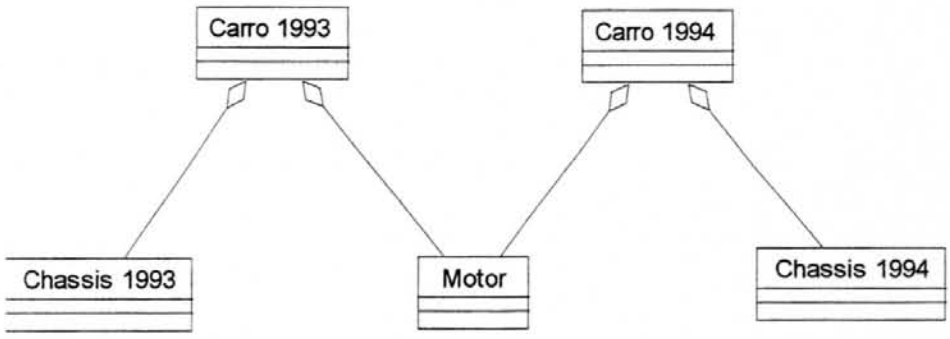

FIGURA 3.1 - Objetos Compostos

```
class Car {
        char *nome;
        char *modelo;
        Ref<motor> motor;
        CompositeRef<chassis> chassis;
}
```

Nos *gateways*, o suporte a objetos compostos é fornecido usando uma combinação de mapeamento de esquemas e geração de código de aplicação. Como o *gateway* depende dos sistemas gerenciadores de banco de dados para armazenar seus dados, e estes apresentam variações, podem haver limitações ao suporte fornecido[CHA97].

Os bancos de dados objeto-relacional fornecem o suporte a objetos compostos, usando uma combinação de gatilhos, usados na propagação de remoção, tipos abstratos de dados, e tipos de coleção, como listas e conjuntos[CHA97].

O suporte a objetos compostos em bancos de dados OO variam, sendo que estes podem ou não ser suportados. No ObjectStore, por exemplo, o suporte aos objetos compostos não está explicitamente presente[SRI97].

3.4 Relacionamentos

Relacionamentos são uma generalização das restrições de integridade referencial nos bancos de dados relacionais, onde uma chave estrangeira aponta para uma chave primária, e esta referência é automaticamente mantida pelo banco de dados. Relacionamentos são referências entre objetos que possuem as seguintes características: propagação automática de remoção, a atribuição de um lado no relacionamento bidirecional automaticamente atribui o outro lado, e a remoção de uma entrada em um lado também automaticamente remove a entrada inversa no outro lado, mantendo a integridade referencial.

Na figura 3.2 é mostrado um exemplo de três tipos de relacionamentos que ocorrem normalmente nas aplicações. Primeiro, empregado (representado pela classe empregado) pode estar relacionado a outro empregado pelo relacionamento cônjuge, que é um exemplo de relacionamento um para um. Continuando, empregados são relacionados aos departamentos (representado pela classe Departamento) , através de um relacionamento um para muitos, ou seja, cada empregado pode trabalhar em apenas um departamento enquanto um departamento pode possuir vários empregados. Finalmente, empregados trabalham em projetos (representado pela classe projeto), um exemplo de relacionamento muitos para muitos, ou seja, um empregado pode trabalhar em muitos projetos e um projeto pode ter muitos empregados.

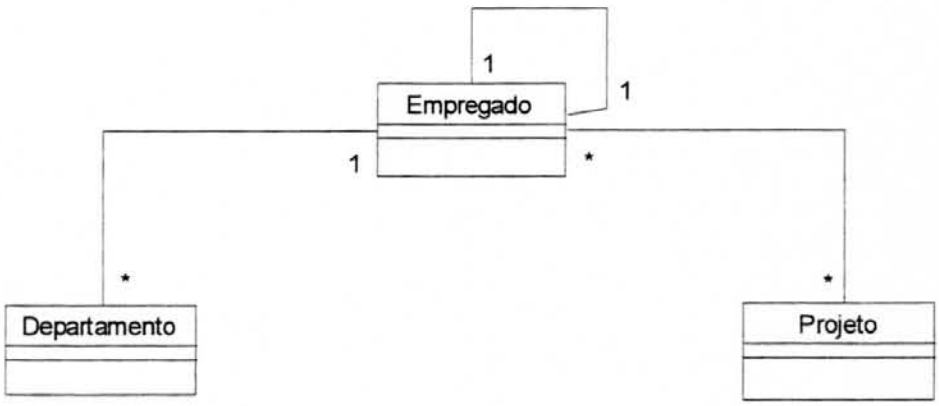

FIGURA 3.2 - Tipos de Relacionamentos

Nos *gateways*, relacionamentos podem ser suportados no nível da aplicação por uma combinação de mapeamento de esquema e geração de consultas apropriadas em tempo de execução, que automaticamente recupera objetos relacionados. Relacionamentos a nível de aplicação necessitam ser mapeados para o banco de dados, através de chaves primárias, chaves estrangeiras e identificadores de linha. Isto pode implicar em algumas limitações, especialmente em termos de performance, uma vez que

a recuperação de objetos relacionados pode requerer múltiplas consultas, com junções sobre banco de dados relacionais.

Sistemas objeto-relacionais fornecem mais facilidades para o suporte a relacionamentos. Identificadores de linhas e coleções nos SGBDOR, juntamente com o suporte a integridade referencial, podem ser usados para dar o suporte completo a relacionamentos.

Nos SGBDOO, uma vez que são permitidos listas e conjuntos como atributos de objetos, os relacionamentos entre objetos podem ser implementados com um conjunto de ponteiros, com o objetivo de armazenar a associação, não sendo necessário novos objetos para implementar um relacionamento.

3.5 Encapsulamento

Em uma linguagem de programação orientada a objetos, o conceito de ADT (Tipo Abstrato de Dados) é normalmente referenciado a uma classe, e a interface está referenciada a um conjunto de métodos públicos. Uma interface está disponível aos usuários de uma ADT, e esta interface não se altera, mesmo se alterações são necessárias em sua estrutura interna. Sendo assim, a interface encapsula a estrutura interna de um ADT.

No armazenamento de objetos persistentes, um nível a mais de implementação necessita ser considerado, em adição a interface e a estrutura interna presentes em um ADT de linguagens de programação. Este nível é conhecido como implementação fisica do banco de dados. Está implementação fisica determina se índices primários e secundários estão disponíveis para o acesso aos dados[CHA97].

Para exemplificar, na figura 3.3 a seguir, a classe Empregado tem uma interface com dois métodos, Contratação( ), e Aumento_Salário( ). A estrutura de dados de cada objeto Empregado é um registro de três campos, chamados código, nome e salário. O modelo fisico do banco de dados consiste de objetos Empregados indexados por dois índices, um código (índice primário) e outro o nome (índice secundário).

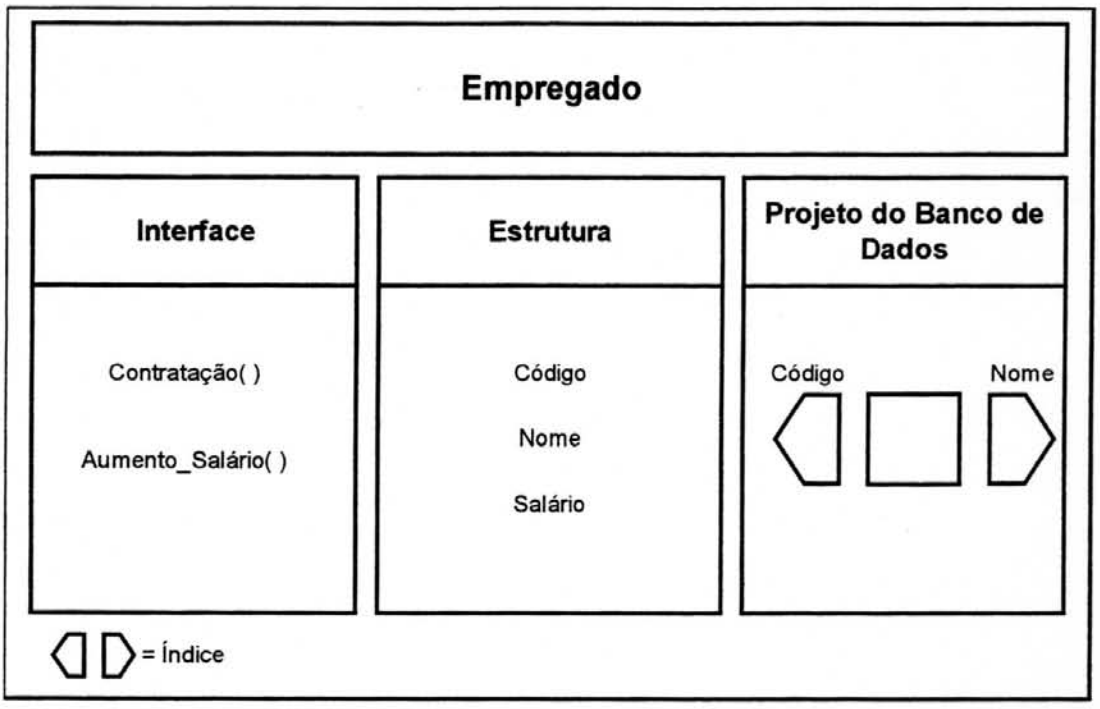

FIGURA 3.3 - Encapsulamento

O encapsulamento nos *gateways* é suportado na aplicação, mas não no modelo de dados usado para o armazenamento no banco de dados. Objetos que são encapsulados na aplicação necessitam ser construídos a partir de dados do banco de dados, usando um mecanismo de tradução. Este esquema é extremamente flexível, possibilitando que diferentes aplicações usem os mesmos dados, utilizando esquemas de mapeamentos diferentes, e portanto, com regras de encapsulamento diferentes.

O padrão SQL/3 suporta o encapsulamento usando ADT's, que podem ser colunas nas tabelas[CHA97]. As linhas de uma tabela não são encapsuladas. Isto porque sistemas objeto-relacionais são compatíveis com a forma normal do modelo relacional.

Os SGDBOO partem do encapsulamento procedural de ADT's, reforçados pelas linguagens de programação. Permitem acesso a estrutura de uma classe através de seus métodos. A quebra das regras de encapsulamento acontece, em alguns SGBDOO, através da execução de consultas ad hoc.

3.6 Herança

Aplicações OO utilizam a herança para realizar implementações de situações reais. Por exemplo, em um banco de dados de uma universidade, a hierarquia mostrada na figura 3.4 deve ser implementada. A herança, que é uma das ferramentas de modelagem, em conjunto com o suporte ao encapsulamento descrito anteriormente, habilita o compartilhamento de implementações entre classes que fazem parte da mesma

hierarquia. Existem muitos tipos de herança em uso nas várias linguagens de programação, e os SGBDOO são afetados por estas variações.

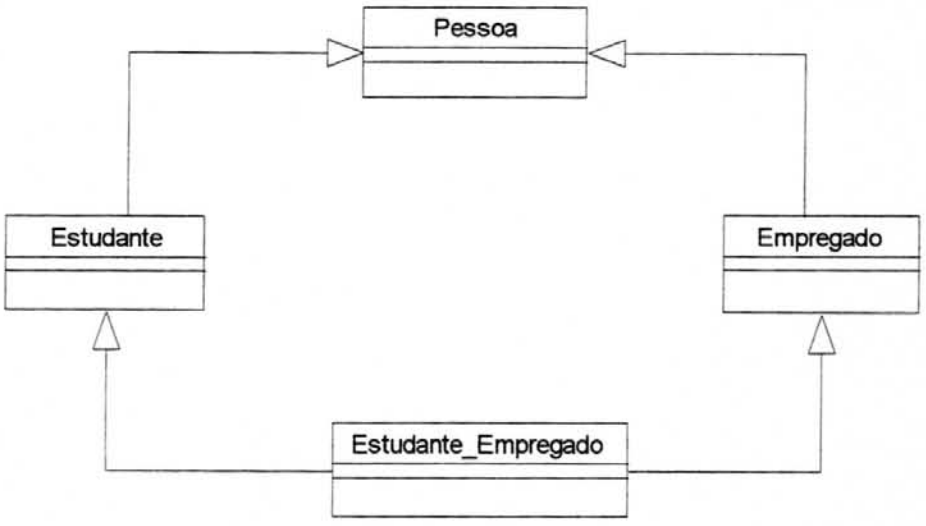

FIGURA 3.4 - Herança

Herança múltipla é o termo usado quando uma classe simples pode herdar de múltiplos pais. Na figura 3.4 acima, a classe Estudante_Empregado herda de ambas as classes, Estudante e Empregado, e é um exemplo de herança múltipla. Sua utilização pode causar problemas de portabilidade, pois os diferentes sistemas que implementam esta característica usam regras incompatíveis para resolver conflitos[SRI97].

Nos *gateways*, a herança é suportada pela linguagem OO, e o gateway tem a função de realizar o mapeamento de esquemas. No capítulo anterior, foram apresentadas algumas alternativas de mapeamentos de esquemas.

Os SGBDOR suportam herança, mas são usadas duas hierarquias separadas, uma para tabelas e uma para os ADT's. A funcionalidade e a semântica desses dois tipos de herança fazem parte dos trabalhos de padronização do SQL/3. Por exemplo, não está claro o que significa herança de tabela, uma vez que tabelas não são encapsuladas[CHA97].

Os SGBDOO suportam as características de herança que já existem nas linguagens de programação orientada a objetos, e, diferentemente do SQL/3, não criaram nenhuma nova semântica. Entretanto, com a finalidade de suportar o cruzamento de linguagens no suporte ao armazenamento de objetos no banco de dados, ou seja, acessar um objeto Smalltalk a partir de um programa C++, eles impõem algumas limitações nas características das linguagens de programação que podem ser usadas no desenvolvimento de tais aplicações.

3.7 Sobrescrita e Sobrecarga de Métodos

A herança permite aos usuários de uma classe estender a interface desta classe, através da redefinição de métodos que já foram definidos em níveis mais altos na hierarquia de classes. Tal sobrescrita de um método, permite que um método que está declarado em uma superclasse, possa ser reimplementado em uma subclasse. Complementando, linguagens de programação OO, permitem a sobrecarga de métodos, onde um novo método pode ser definido com o mesmo nome de um outro método já existente, mas com um conjunto de argumentos diferentes. Ambos, sobrescrita e sobrecarga, são utilizados para escrever códigos de fácil entendimento.

Nos *gateways*, como os objetos de uma aplicação são os objetos da linguagem de programação, e como seus métodos não são armazenados no banco de dados, estas características não são afetadas. Somente os dados dos objetos são armazenados, e estes são traduzidos de uma esquema para outro no momento de instanciação de um objeto persistente.

Já os SGBDOR armazenam ambos, os dados e os métodos, dentro do banco de dados. Estes métodos podem ser usados em consultas, *stored procedures*, e em funções definidas pelo usuário, além dos programas de aplicações.

Nos SGBDOO, os métodos normalmente são executados no cliente, uma vez que estes métodos são escritos nas linguagens de programação, cujo ambiente está disponível somente no cliente. A maioria dos sistemas não armazena métodos na base de dados.

4 Abordagem de persistência adotada no *framework*

Este capítulo descreve a abordagem de persistência que está sendo adotada no *framework*, ou seja, quais serviços de persistência estarão disponíveis aos objetos do domínio do problema. Para isto, inicialmente é descrito como a aplicação foi dividida em componentes, suas interfaces e funcionalidades. A seguir é fornecida uma visão geral da arquitetura, para em seguida ser detalhado o componente domínio do problema. Os padrões de projeto OO utilizados na arquitetura do *framework*, serão abordados durante a descrição da arquitetura.

4. 1 Divisão da Aplicação em Componentes

A materialização e a desmaterialização dos objetos a partir de um dispositivo de armazenamento persistente não deve ser responsabilidade das classes de objetos persistentes do domínio do problema. O Padrão *Expert* [LAR98] é contrário a esta abordagem, e sugere que devam ser destas classes, ou seja, que não fosse implementado uma camada adicional para o gerenciamento de dados. Contudo, esta abordagem apresenta alguns inconvenientes, como:

- Aumenta o acoplamento das classes de objetos persistentes ao conhecimento de como armazenar no respectivo dispositivo, violando a regra de projeto de baixo acoplamento;
- Aumenta a complexidade do objeto com uma responsabilidade nova e não relacionada ao domínio do problema;
- Diminui a coesão entre o conjunto de classes que colaboram para fornecer serviços;
- Dificulta a portabilidade da aplicação entre SGBDs.

O padrão *Database Broker* [BRO96] propõem a criação de uma classe que é responsável pela materialização, desmaterialização e pelo cache de objetos. O padrão também sugere a utilização de uma classe para cada classe de objeto persistente. Uma outra alternativa, seria a utilização de uma classe apenas, com um meta dicionário e instâncias parametrizadas, porém sua implementação aumenta a complexidade do *framework*, e por isto não será utilizada.

A figura 4.1 a seguir ilustra que cada objeto persistente deve possuir sua própria classe gerenciadora, e que podem existir diferentes tipos de classes gerenciadoras para diferentes mecanismos de armazenamento persistente, ou diferentes sistemas gerenciadores de banco de dados.

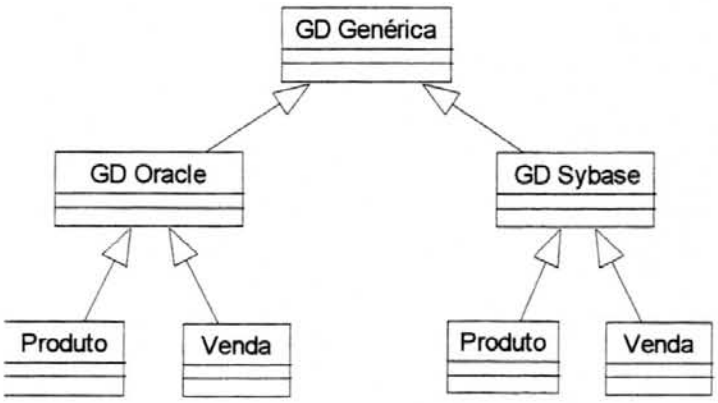

FIGURA 4.1 - Hierarquia do padrão *Database Broker*

As classes Produto e Venda são classes concretas responsáveis pela materialização de seus respectivos objetos, a partir de um banco de dados definido. Esta definição é estabelecida através das superclasses abstratas GDOracle e GDSybase.

Baseado neste padrão, esta seção do trabalho continua com um estudo mais detalhado de como projetar um modelo de objetos dividido em componente , sendo que o componente gerenciador de dados será a abordagem central do próximo capítulo, pois sua utilização é indispensável para a implementação da arquitetura proposta.

A figura 4.2 mostra a aplicação divida em camadas, que se comunicam trocando informações.

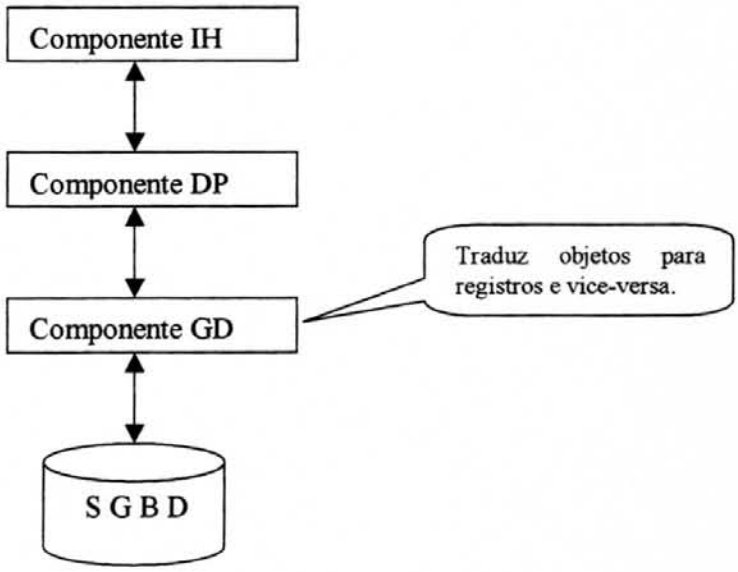

FIGURA 4.2 - Divisão em camadas da Aplicação

A arquitetura da aplicação em três camadas representada acima, será descrita a seguir, onde serão mostradas as vantagens de sua utilização, bem como as responsabilidades de cada uma destas camadas.

Normalmente, aplicações OO são divididas em camadas. A arquitetura utilizada neste trabalho, segue a proposta definida em [COA95], que define os componentes que fazem parte do modelo de objetos, cujas funções são de guiar e organizar o trabalho de modelagem em uma análise. Foi um tipo de particionamento do modelo de objetos, que ficou dividido nos seguintes componentes: domínio do problema (DP), interação humana (IH), gerenciador de dados (GD), interação de sistemas (IS) e o último componente, fora de escopo do sistema (FE), para os objetos que poderiam ser desconsiderados pelo menos na fase em que se encontra a modelagem.

O componente domínio do problema, contém os objetos que correspondem diretamente ao problema que está sendo modelado. Eles tem pouco, ou nenhum, conhecimento sobre os objetos dos outros componentes. O componente de interação humana, contém os objetos que fornecem uma interface entre os objetos do domínio do problema e as pessoas. Dentro de um modelo de objetos, tais objetos freqüentemente correspondem a janelas ou relatórios específicos. O componente de interação de sistema, contém os objetos que fornecem uma interface entre os objetos do domínio do problema e os outros sistemas, encapsulando protocolos de comunicação e detalhes específicos de implementação, deixando os objetos do domínio de problema livres destes tipos de detalhes.

Os objetos do componente gerenciador de dados são uma importante parte do modelo de objetos. Estes objetos fornecem uma interface entre os objetos do domínio do problema e o mecanismo de persistência de objetos, que normalmente é um sistema gerenciador de banco de dados, sendo estes objetos normalmente correspondentes a objetos do domínio do problema que necessitam de persistência e pesquisa.

4.1.1 Relação entre os Componentes DP e GD

Quando existe a necessidade de persistência aos objetos DP, ou quando existem objetos DP que exigem algum tipo de pesquisa sobre algum número de objetos persistentes, então há a necessidade de adicionar um objeto GD correspondente.

O objeto DP interage com seu objeto GD correspondente, durante a criação de um novo objeto, a recuperação de um objeto já existente na base dados, operação esta conhecida como materialização, ou a atualização deste objeto na base de dados.

Um objeto DP pode solicitar a outro objeto GD que este suporte uma ação de pesquisa sobre uma grande quantidade de objetos que este objeto DP se relacione, ou seja, um objeto DP não se comunica só com seu objeto GD correspondente, mas com todos os objetos com que ele se relacione.

4.1.2 Relação entre os Componentes IH e DP

Na maioria das vezes, um objeto de Interface Humana (IH) consegue todo o suporte que necessita diretamente do objeto DP. Eles se interagem da seguinte maneira:

- para criar um novo objeto, um objeto IH envia uma mensagem de 'Criar' para sua classe DP correspondente. O novo objeto DP registra-se a si próprio com seu objeto GD.
- para pesquisar um conjunto de objetos, um objeto IH envia uma mensagem 'Pesquisar' para o objeto DP que mantém o conjunto. Por sua vez, o objeto DP pede ao correspondente objeto GD para fazer a pesquisa solicitada.

Eventualmente, objetos IH precisam trabalhar diretamente com os objetos GD. Para isto, eles se interagem da seguinte maneira:

- para pesquisar um conjunto de objetos, quando o conjunto DP não está disponível, um objeto IH envia uma mensagem 'Pesquisar' para o objeto GD responsável pelo conjunto de objetos cujo objeto IH necessita.
- envia uma mensagem de *Start*, *Rollback* ou *Commit* de uma transação de banco de dados para o objeto GD responsável pelo conjunto de objetos que o objeto IH necessita trabalhar.

4.2 Visão Geral da Arquitetura

A arquitetura de uma aplicação que usa o *framework* foi concebida com o objetivo de separar o tratamento da persistência do domínio do problema, ou seja, seguir o padrão *Database Broker*. Esta separação objetiva tornar as classes de domínio de problema menos dependentes de mudanças na forma de armazenamento[JOH96,ROG97].

Uma aplicação está organizada em quatro componentes (conjuntos de classes) interrelacionados, conforme mostrado na Figura 4.3.

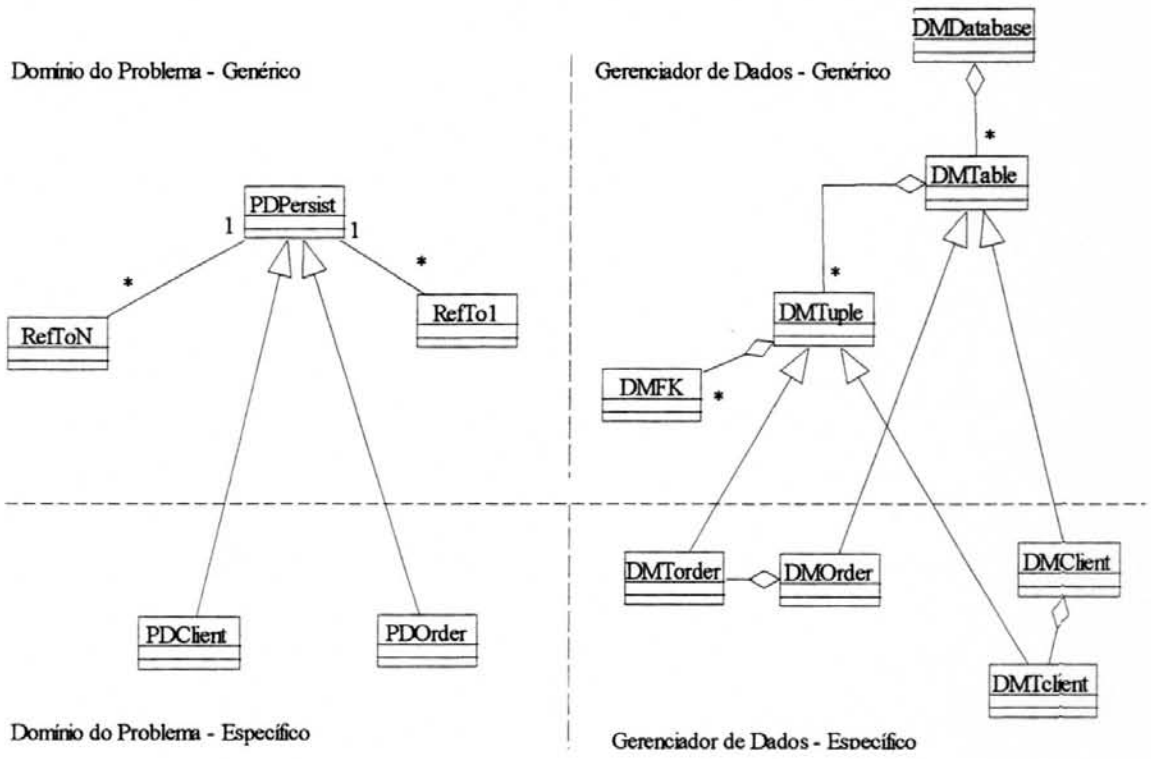

FIGURA 4.3 - Arquitetura de uma aplicação que usa o *Framework*

Na parte superior da figura, estão representados os componentes que compõe o *framework* propriamente dito, isto é, as classes genéricas que serão especializadas para aplicações específicas. Na parte inferior, encontram-se os componentes específicos de uma aplicação.

Na parte da esquerda da figura, estão representados os componentes que contém as classes que representam o domínio de problema (PD – *Problem Domain*) da aplicação. Estas serão as classes construídas e usadas pelo programador das aplicações. Já as classes que aparecem na direita da figura, formam o componente gerenciador de dados (DM – *Database Manager*), isto é, as classes que implementam o mapeamento OO-relacional. Estas classes não são utilizadas nem referenciadas pelo programador da aplicação, e sim por um administrador de dados, que conhece os padrões de mapeamentos que serão utilizados na aplicação.

As classes dos componentes DP tem associações com classes dos componentes GD. Para simplificar o diagrama, estas associações não estão representadas na figura acima. O componente DP genérico contém classes abstratas de onde se derivam todas as classes persistentes do domínio do problema. Através da interface fornecida por estas classes, os objetos persistentes podem ser armazenados, recuperados, inseridos e excluídos do banco de dados, bem como são tratados seus relacionamentos. Os métodos destas classes são métodos *template*, que contém o código necessário para a comunicação com as classes associadas no componente GD.

O componente DP-específico é formado pelas classes que modelam o domínio de problema da aplicação. No exemplo da figura acima, são classes que modelam pedidos e clientes. Estas classes são especializações das classes do componente DP genérico.

O componente GD-genérico é formado por classes abstratas genéricas. Estas classes contém métodos abstratos, servindo apenas para definir a interface das classes GD específicas da aplicação

Já no componente GD-específico cada classe corresponde a uma construção (tabela, tupla, chave estrangeira) da base de dados relacional. Os métodos destas classes realizam o mapeamento entre os modelos e contém código SQL específico para cada operação a ser realizada sobre a base de dados. Nesta versão da arquitetura, este componente é de responsabilidade de um administrador de dados, que deve desenvolver os métodos de forma manual. Entretanto, estas classes e seu métodos poderiam ser gerados automaticamente a partir do esquema da base de dados a ser usada.

O mapeamento entre o modelo de objetos e o modelo relacional de dados adotado, já foi explicado anteriormente no capítulo 2.

A figura 4.4 mostra uma visão geral da arquitetura, já com classes de domínio de problema que serão usadas durante a explicação da arquitetura.

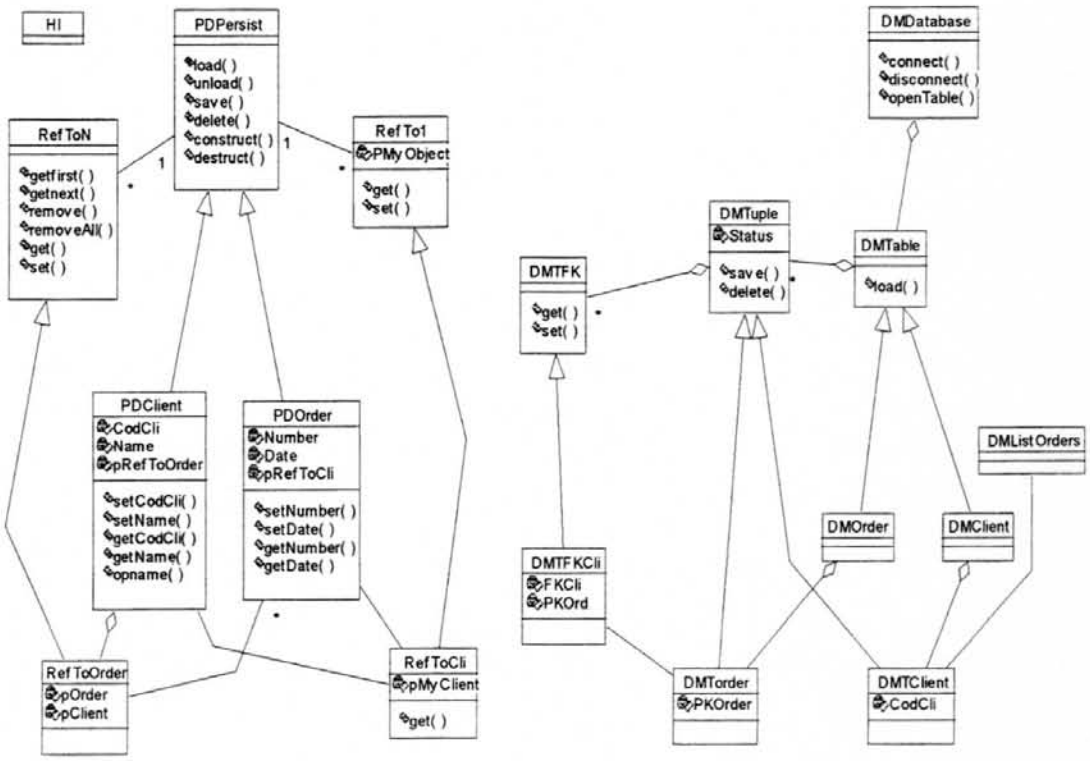

FIGURA 4.4 - Modelo de objetos completo da arquitetura do *framework*

Os métodos e atributos, especificados no modelo, são demonstrados mais adiante, conforme forem explicados o fluxo de informações dentro do modelo de objetos.

4.3 O Componente Domínio do Problema

Nesta seção será abordada a metodologia de persistência que foi adotada no *framework*, ou seja, serão descritas as classes responsáveis por prover os serviços de persistência necessários a qualquer classe do domínio do problema, bem como uma descrição do comportamento destas classes. A figura 4.5 mostra os estados pelos quais um objeto persistente pode passar, bem como os métodos que levam de um estado para outro.

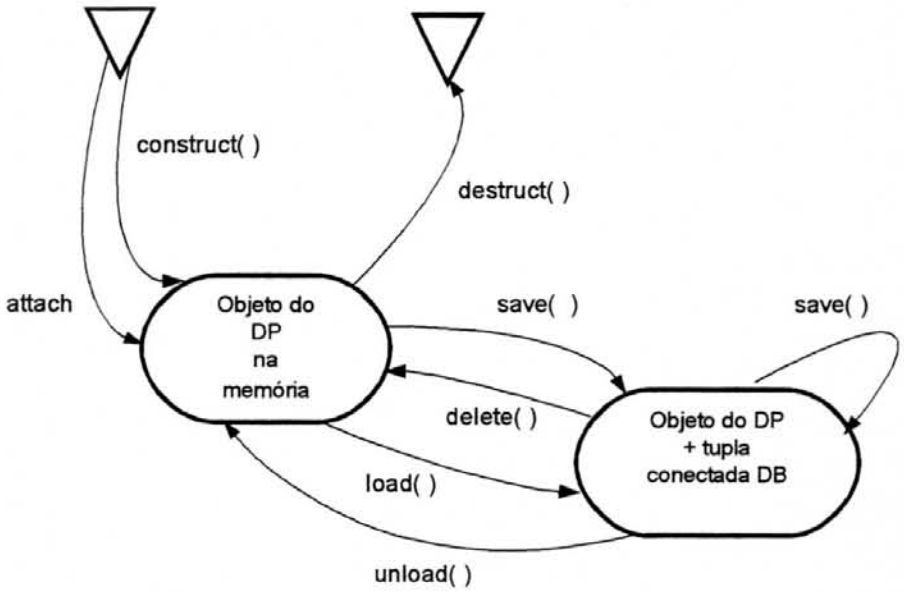

FIGURA 4.5 - Estados de um objeto persistente

Conforme a figura mostra, um objeto possui basicamente dois estados:

- O estado inicial do objeto, quando ele é construído, é o de existir na memória, mas estar desconectado de uma tupla na BD. Neste estado o objeto pode ser destruído, sem efeito sobre a BD.

- Caso sejam realizadas operações de BD (*load()* ou *save()*), o objeto passará ao estado de *conectado com uma tupla*. Para que o objeto possa ser removido da memória, é necessário que a operação de exclusão seja utilizada (*delete()*) ou que o objeto seja desconectado da tupla na BD (*unload()*).

4.3.1 Classes responsáveis pela Persistência

As classes que compõe o componente DP-genérico são as classes abstratas PDPersist, RefTo1 e RefToN, ilustradas na figura 4.6, das quais são derivadas todas as classes persistentes do domínio do problema. A classe PDPersist é a classe da qual são

derivadas as classes persistentes que representam entidades. Já as classes RefTo1 e RefToN servem para representar relacionamentos. A classe abstrata RefToN implementa a associação a um conjunto de objetos relacionados, enquanto que a classe RefTo1 implementa a associação a exatamente um objeto.

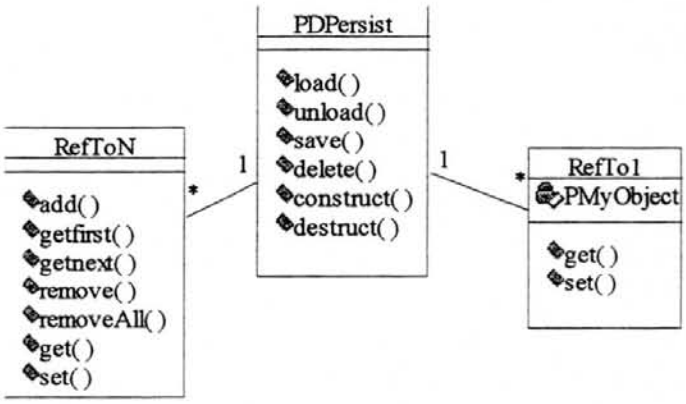

FIGURA 4.6 - Classe do componente DP genérico

Os métodos destas classes são do tipo *template*, isto é, estão implementados no *framework*, mas fazem referência a métodos abstratos dentro do *framework* (no componente GD) que serão implementados na aplicação que instância o *framework*. Estes métodos, bem como o modo de utilização, serão descritos com mais detalhe a seguir.

4.3.2 Classes do Domínio do Problema

Para que possa ser melhor entendida a funcionalidade e os objetivos do *framework*, foram definidas duas classes para o componente Domínio do Problema (DP) específico da aplicação. Estas classes mostram a herança e o fluxo de dados na arquitetura.

A figura 4.7 mostra a estrutura das duas classes exemplo, que modelam pedidos e clientes (PDClient e PDOrder). São classes especializadas das classes do componente DP genérico.

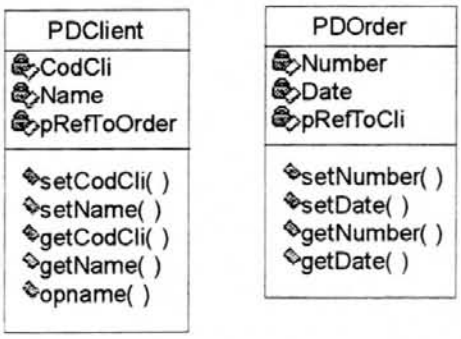

FIGURA 4.7 - Classes do componente domínio do problema específico

Qualquer classe do componente domínio do problema apresenta a mesma definição no *framework*. A seguir está exemplificada a classe PDClient:

```
class PDClient:PDPersist {
private:
        TListClient DataCli          ; //lista gerenciadora dos dados dos clientes


        RefToOrd * pOrder            ;
public:
        DMClient   DMC               ;
        PDClient()                   ;
        Variant GetValue(String)     ;
        void   SetValue(String,String);
        bool    Load(int)            ;
        bool    Save(void)           ;
        bool    Destruct(void) ;
        bool    Construct(void)      ;
        bool    Delete(void)         ;
} ;
```

4.3.3 Seqüências de utilização das classe DP

Nesta seção são apresentados exemplos típicos de chamadas aos métodos das classes DP.

4.3.3.1 Tratamento de Entidades

No tratamento de classes (entidades) simples, não relacionadas, é utilizada apenas a classe PDPersist. Esta é uma classe abstrata que contém as operações de persistência dos objetos do domínio do problema, removendo este conhecimento específico de persistência destas classes, fazendo com que o analista não se preocupe com estas operações. Desta classe são derivadas as classes persistentes que representam entidades. PDPersist possui métodos abstratos, os quais dão suporte para que suas subclasses possam ser codificadas de forma específica, conforme o domínio da aplicação. A classe PDPersist, por ser parte do *framework*, modela a aplicação como um todo, de forma genérica, não podendo, portanto, possuir informações relacionadas com o domínio da aplicação específico. Os exemplos utilizados para explicar a arquitetura serão baseados no seguinte modelo do domínio do problema mostrado na figura 4.8.

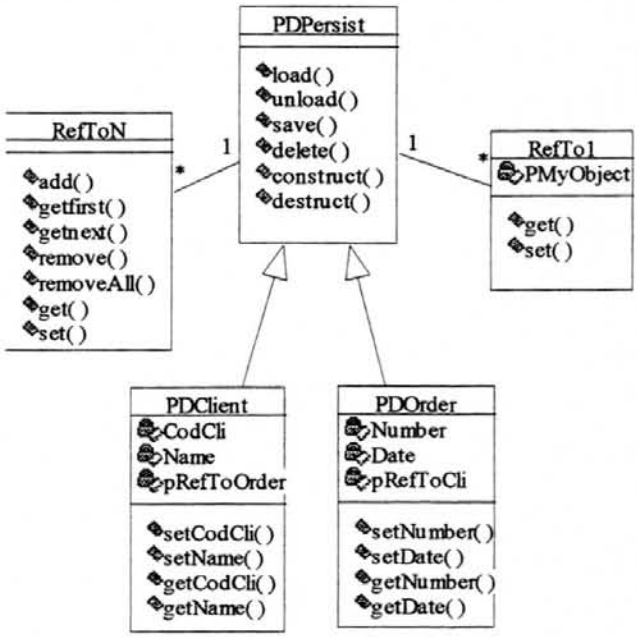

FIGURA 4.8 - Classes que implementam entidades

No *framework* desenvolvido, a classe PDPersist aparece codificada da seguinte forma:

```
class PDPersist {
        void Load()         ;
        void UnLoad()       ;
        void Save()         ;
        void Delete()       ;
        void Construct()    ;
        void Destruct()     ;
} ;
```

Os métodos da classe PDPersist são herdados pelas classes DP específico, de forma que, quando um método for invocado, o mesmo será implementado nas classes específicas da aplicação, estando a classe PDPersist isenta de quaisquer manipulações de dados, servindo apenas para dar suporte as suas subclasses, conforme o domínio do problema da aplicação. Os métodos da classe DPPersist são descritos a seguir, bem como um exemplo de utilização codificado em C++ para melhor entendimento.

Método Construct( )
Responsável pela instanciação de um objeto novo na memória. Este procedimento deve ser executado antes mesmo de associar o objeto a uma chave primária no banco de dados. Exemplo de definição do método Construct() no *framework*:

```
bool DMClient::Construct(void) {

    DMTC.PKClient.Append() ;
    DMTC.PKClient.CurrentData->Status = NEW ;
    DMTC.PKClient.CurrentData->PK = -1 ;
```

```
    DataCli->Append() ;
    DataCli->CurrentData->CodCli = -1 ;

    return true ;
}
```

Conforme o exemplo apresentado, é instanciado um novo objeto, instanciando também novos objetos na classe PKClient, e adicionando uma nova entrada na lista de clientes que se mantém na memória.

Método Destruct( )
Método responsável pela remoção de um objeto da memória, procedimento que deve ser executado após seu uso pela aplicação. A utilização do método *Destruct()* é importante, pois todos os objetos que não estão sendo mais utilizados na aplicação são destruídos, visto que os objetos são criados e destruídos dinamicamente, conforme a necessidade dos mesmos nas aplicações.

Método Load( )
O método Load( ) busca no banco de dados uma tupla, através da chave primária contida no objeto do DP, transferindo os dados ao objeto. Aqui está um exemplo de como é usado:

```
1. Client cli;
2. cli.setCodCli( );
3. cli.Load( );
4. Cstring nome = cli.getname( );
5. cli.setAddress("Rua 28 de setembro, 36");
```

Se uma operação de Load( ) é chamada sobre um objeto já carregado, esta função não fará nada. Isto ocorre porque quando um objeto é carregado para memória, uma classe DMTuple é instanciada no componente gerenciador de dados específico.

Exemplificando, se a classe PDClient disparar o método *Load()*, tal método dispara o *Load()* da classe DMClient no componente GD específico. A classe DMClient contém instruções SQL que fazem o acesso à base de dados, transferindo os dados para o objeto PDClient que disparou o *Load()*. Quando o acesso aos dados for efetuado através da classe DMClient, o objeto DMTClient é instanciado, que manterá o *status* (NotModified) e a chave primária da tupla carregada para memória.

O diagrama da figura 4.9 ilustra, de forma mais clara, como é executado o método *Load()* do componente DP.

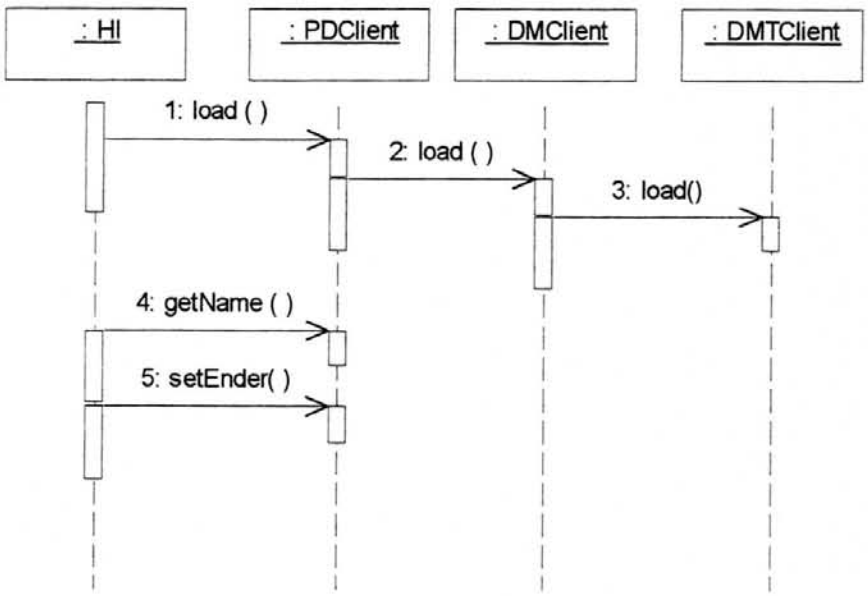

FIGURA 4.9 - Diagrama de eventos para o método Load( )

Método Unload( )
Método que possibilita que um objeto que está associado a uma tupla do banco de dados seja desconectado. O objeto não pode ser removido da memória se estiver conectado ao banco de dados.

Método Save( )
Apresenta dentro do modelo a função de atualizar o banco de dados. Sua funcionalidade está dividida em duas possibilidades:

- Executar a alteração de atributos (SQL "update").

Neste caso o método registra na base de dados as mudanças efetuadas sobre um objeto que teve seus dados carregados para a memória a partir do banco de dados. Quando um objeto carregado do banco de dados é modificado, todas as modificações realizadas afetam somente o objeto, e não a tupla associada do banco de dados. Para tornar estas modificações persistentes, deve ser executada a função Save( ). Esta operação utiliza as informações armazenadas no objeto, transferindo-as a tupla correspondente. O exemplo anterior pode ser concluído da seguinte maneira:

```
1. Client cli;
2. cli.setCodCli( );
3. cli.Load( );
4. Cstring nome = cli.getname( );
5. cli.setAddress("Rua 28 de setembro, 36");
6. cli.Save( );
```

A chamada da operação Save( ) atualiza a tupla com a chave primária que foi armazenada em cli.CodCli, alterando o endereço do cliente.

Normalmente existirá uma janela ou tela de diálogo para editar um objeto persistente. Quando se quer editar um objeto, cria-se uma instância da classe, atualiza-o

com dados da tupla e transfere-se estes dados para a interface. Quando o usuário pressionar o botão de OK, a função Save( ) do objeto deve ser executada para atualizar as modificações no banco de dados. Se o usuário pressionar o botão Cancela, simplesmente não se chama a função Save( ), e exclui-se a instância do objeto. A figura 4.10 mostra o diagrama de eventos para este método.

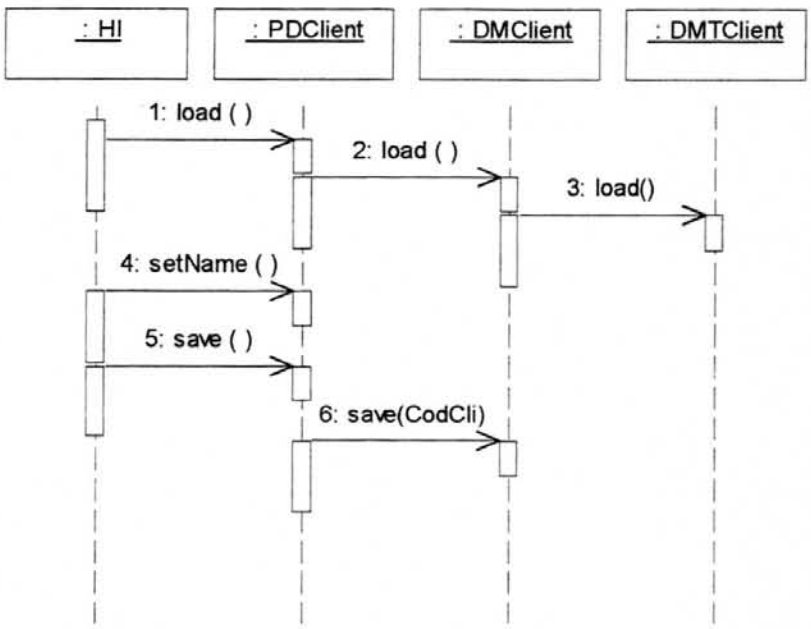

FIGURA 4.10- Diagrama de eventos para o método Save( )

- Executar a inserção de novas tuplas (SQL "insert").

A outra função do método save() é a de inserir novas tuplas na BD. Para criar um objeto persistente deve-se inicialmente criar um objeto transiente, inicializando-o com suas informações, e tornando-o persistente através da chamada a operação Save( ). Esta operação insere uma nova tupla no banco de dados, transferindo as informações contidas no objeto, e ligando-o a tupla inserida. Por exemplo:

```
1. Client cli;
2. cli.setName("Kurt Molz");
3. cli.setAddress("28 de setembro, 36");
4. cli.Save( );
5. ...
6. Cli.id = cli.getCodCli( );
7. cli.setAddress("Rua Marechal Floriano, 100");
8. cli.Save( );
```

Inicialmente, o objeto cli é usado como um objeto temporário, onde são atribuídas as informações de nome e endereço. A seguir, o objeto é salvo, sendo inserida uma nova tupla no banco de dados, ficando agora o objeto ligado a esta tupla. Com isso, pode ser executada a função membro que obtém o código do cliente criado, e eventualmente alterarmos novamente o endereço cadastrado para este objeto. A transação termina com uma nova chamada a operação Save( ).

Quanto a chave primária, pode ser de duas formas:

- A chave primária é um atributo do objeto a ser informado antes da inserção. A chave primária que é usada para ligar o objeto depende do tipo de chave existente na tabela relacional. Por exemplo, a chave da classe Client poderia ser seu CPF. Neste caso, deveríamos atribuir o seu conteúdo através de um método de set da classe, antes de chamar o método Save( ).

- A chave primária é um identificador gerado internamente pelo banco, este campo não necessita ser informado no objeto. Na implementação da classe Client foi utilizada esta abordagem, através da utilização do atributo CodCli. Neste tipo de chave, o valor do atributo não precisa ser inicializado, pois o banco de dados ficará encarregado por esta tarefa.

Método Delete( )

Para destruir objetos, deve ser chamada a função Delete( ), que remove do banco de dados a tupla associada ao objeto. Este método não remove o objeto instanciado na memória.

```
1. Client cli;
2. cli.setCodCli( );
3. cli.Load( );
4. cli.Delete( );
```

Note que o objeto deve estar associado a uma tupla no banco de dados, o que é obtido com a operação Load( ). Esta operação remove do banco de dados a tupla associada ao objeto. Entretanto, o objeto que está na memória não é excluído, sendo necessário a execução do método destruct( ). A figura 4.11 mostra o diagrama de eventos para este método.

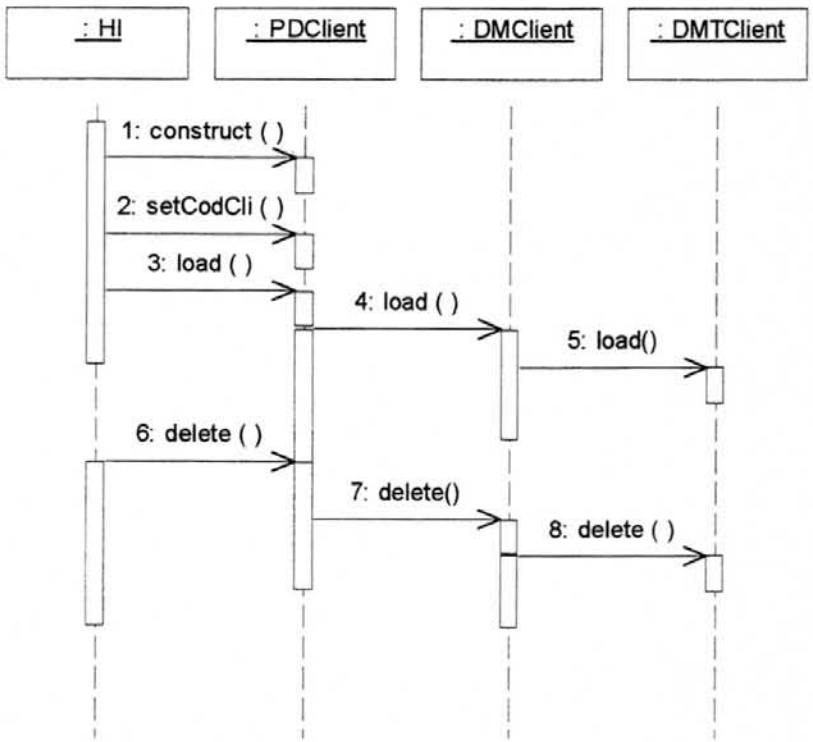

FIGURA 4.11- Diagrama de eventos para o método Delete( )

4.3.3.2.Tratamento de Associações

Nesta seção serão abordadas as associações entre as classes do modelo de objetos. Estas associações representam os relacionamentos do modelo relacional.

Uma associação é implementada através de duas classes, uma para cada extremo da associação. Estas classes são especializações da classe RefTo1 ou da classe RefToN, conforme cardinalidade do relacionamento. Estas duas classes genéricas serão descritas a seguir.

4.3.3.2.1 Referências "para 1"

A classe RefTo1 tem a função de dar o suporte para referências "para um". A figura 4.12 abaixo apresenta o modelo de objetos com a utilização da classe de relacionamento RefTo1.

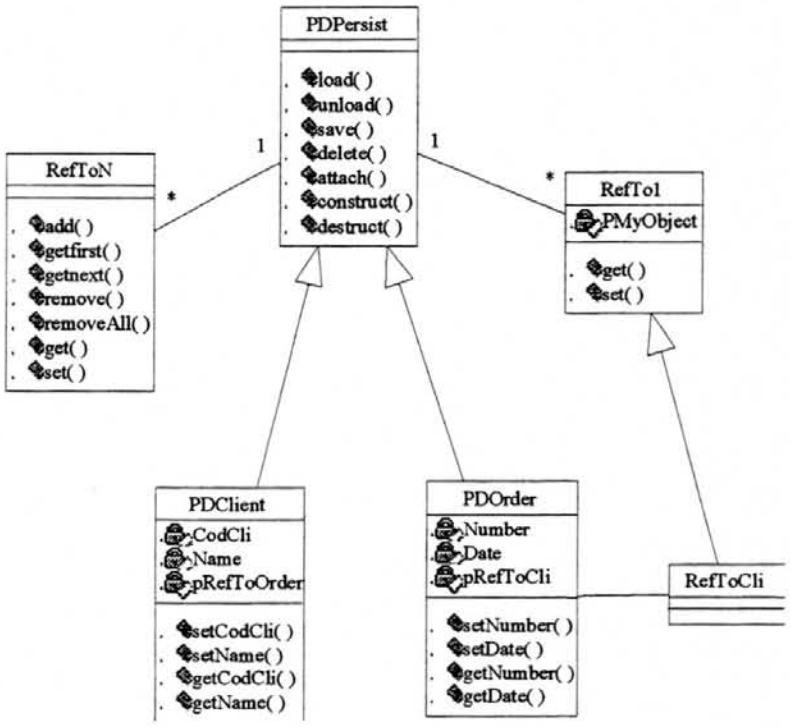

FIGURA 4.12- Referência para "1"

Os métodos que aparecem a nível de DP, nas classes referência, que serão herdados da classe RefTo1, são descritos a seguir.

Métodos Get( ) e Set( )
Métodos que permitem que o desenvolvedor execute atribuições a referência, através do método set( ), ou execute a obtenção do mesmo, através do método get( ). Suas funcionalidades são as mesmas dos métodos que trabalham com dados membros das classes, porém estes métodos trabalham com as referências entre as classes.

Para exemplificar a utilização desta classe da arquitetura, é demonstrado a seguir um exemplo que trabalha com as classe Cliente e Pedido que constam do modelo da figura 4.12. As tuplas da tabela Pedido possuem uma chave estrangeira chamada CodCli, que especifica o cliente deste pedido. No diagrama de classe ele é representado como uma referência, através do ponteiro *pMyObject* na classe de referência *RefToCli*.

Em geral, chaves estrangeiras são traduzidas como agregação de objetos persistentes. Estes objetos serão chamados de objetos estrangeiros, e os objetos que contém estes objetos estrangeiros de objetos contenedores. A classe do objeto estrangeiro depende da tabela ao qual se refere a chave estrangeira. No exemplo, CodCli é uma chave estrangeira que se refere a tabela Cliente, então o objeto estrangeiro correspondente a chave estrangeira será a classe Cliente. Objetos estrangeiros sempre são objetos persistentes, e eles são ligados a suas correspondentes chaves estrangeiras.

As classes que possuem associações fornecem funções que permitem acesso a seus objetos estrangeiros. Por exemplo:

```
1. Order ord;
2. ord.setNumber( );
3. ord.Load( );
4. Client* pMyObject  = ord.pRefToCli.get( );
5. Cstring name = pMyClient ->getName( );
```

Objetos estrangeiros são *construídos sob demanda*, usando o conceito conhecido como *lazzy evaluation*. Ou seja, o objeto Cliente ligado ao objeto Pedido é construído somente quando o usuário executa a chamada a função get( ) da classe referência *RefToCli*. O ponteiro para o objeto Cliente fica armazenado na classe de referência. O exemplo acima é mostrado na figura 4.13 em um diagrama de eventos.

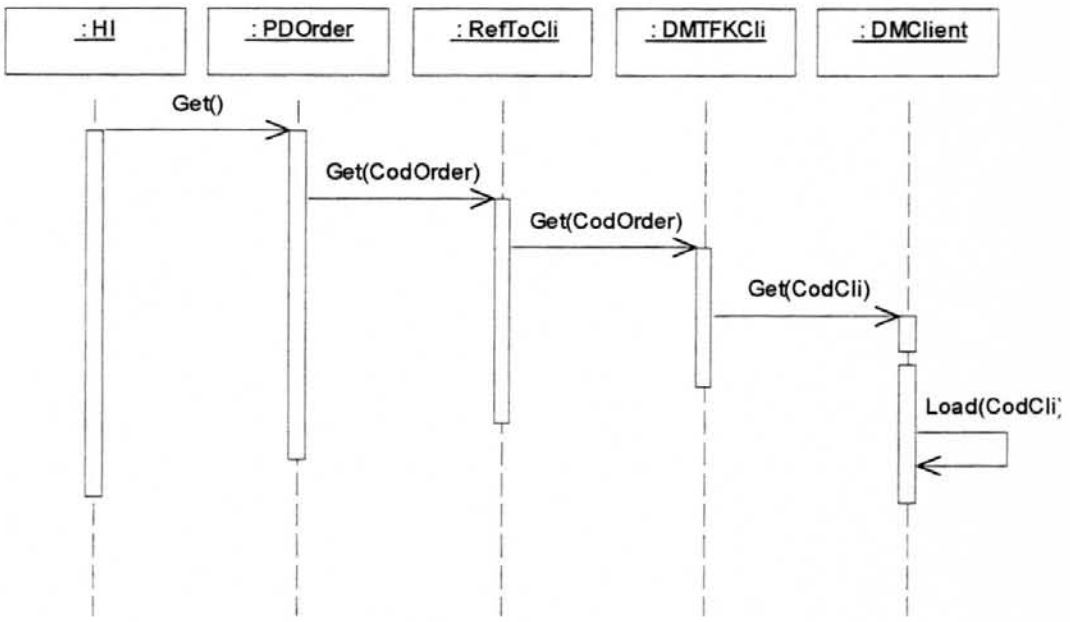

FIGURA 4.13- Diagrama de eventos para referência "para 1"

Uma vez que os objetos estrangeiros são construídos por seus objetos contenedores, é de responsabilidade do contenedor destruí-los também. Pode-se também atribuir um objeto estrangeiro a um objeto contenedor. Por exemplo:

```
1. Order ord;
2. ord.setNumber( );
3. ord.Load( );
4. ...
5. Client cli;
6. cli.setCodCli( );
7. cli.Load( );
8. ord.pRefToCli.set(&cli);
9. ord.save( );
```

Quando a função Save( ) é executada, a chave estrangeira é atualizada com o valor da chave que corresponde ao objeto estrangeiro que está relacionado.

4.3.3.2.2 Referências para "N"

A classe abstrata RefToN dá o suporte para referências para muitos, suportando a interação de uma lista de objetos persistentes associados. Sempre quando um objeto necessitar implementar um relacionamento 1- n, esta classe será utilizada. A figura 4.14 apresenta o modelo de objetos com a utilização da classe de referência RefToN.

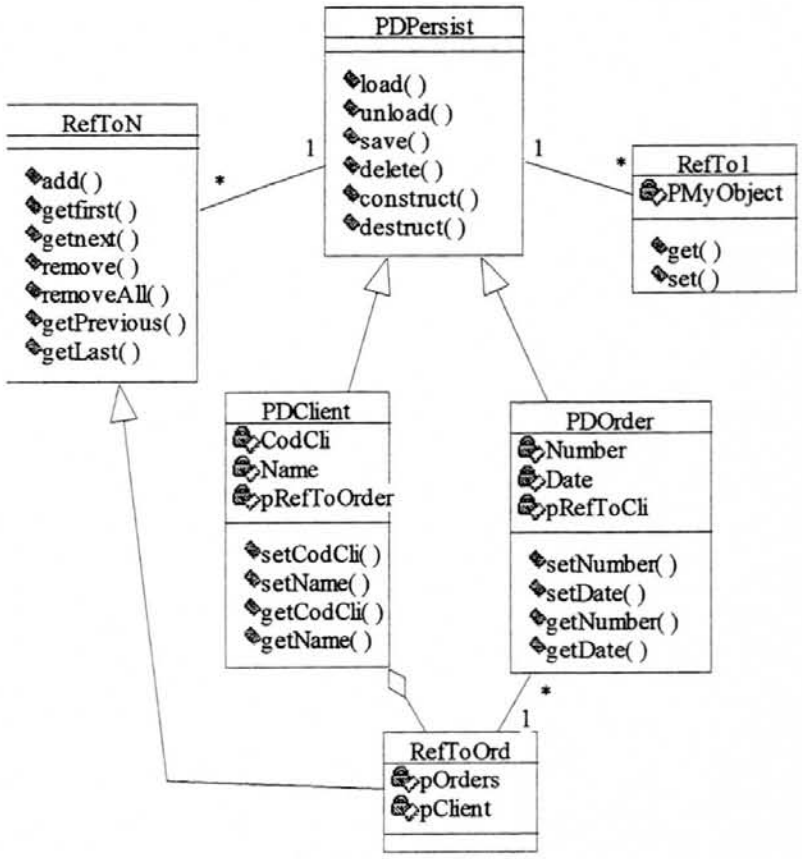

FIGURA 4.14- Referências "para N"

A definição da classe RefToOrder, que herda de RefToN será:

```
class RefToOrd : RefToN {
      public:
      TListClient * DataCli   ;
      DMListOrders DMLO       ;

        bool GetFirst   (void) ;
        bool GetNext    (void)      ;
        bool GetPrevious(void)      ;
        bool GetLast    (void) ;
```

```
        bool Remove    (void) ;
        bool RemoveAll (void) ;
};
```

O diagrama de eventos da figura 4.15, a seguir, mostra as classes envolvidas na manipulação da DMListOrder, no momento da obtenção de um pedido:

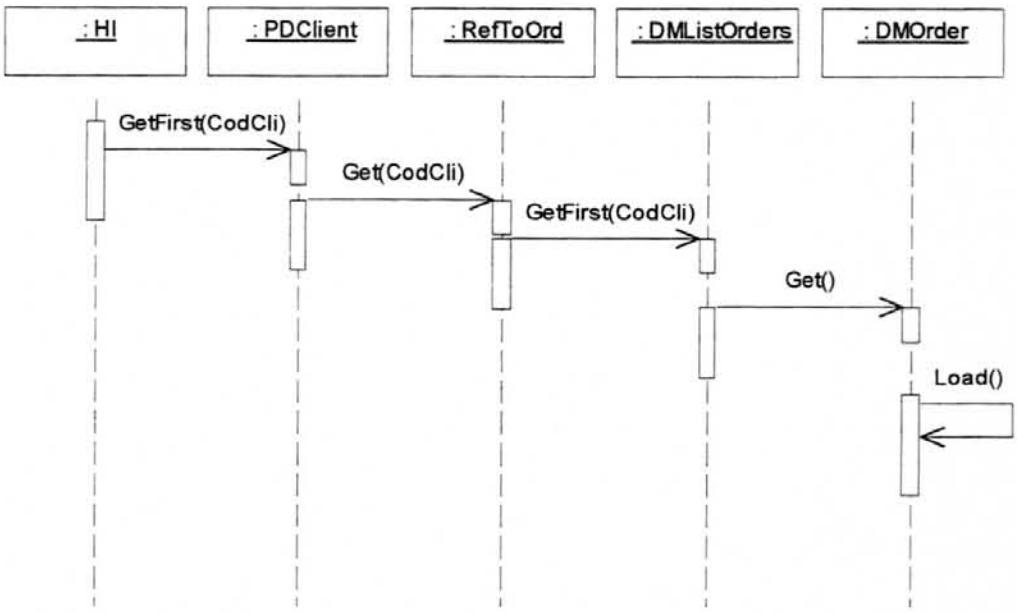

FIGURA 4.15- Exemplo de uso de classes referência

O diagrama acima representado mostra como é feita a recuperação de um pedido, tomando como parâmetro o código de um cliente. No exemplo, é necessária a recuperação de um pedido. Para isto, é disparado o método GetFirst() – neste caso recupera o primeiro pedido, na classe PDClient, parametrizando o código do cliente e passando para a classe RefToOrd. Tal classe dispara o método Get() em DMListOrders, a qual faz o acesso ao banco de dados, de forma a recuperar todos os pedidos de um determinado cliente.

Da mesma forma com explicado no diagrama 4.15, podem ser executadas as demais operações sobre a lista de pedidos, sendo possível a navegação entre todos os pedidos de um determinado cliente, bem como a remoção dos mesmos.

A seguir, são descritos os métodos que aparecem a nível de DP que são herdados da classe RefToN, com a finalidade de manipular objetos persistentes:

Método GetFirst( )

Retorna o primeiro elemento do conjunto. No caso do exemplo que foi implementado no *framework*, o método GetFirst() retorna o primeiro pedido de um determinado cliente. A nível de domínio de problema, o programador não toma conhecimento de como o método vai ser instanciado, porém o método dispara um outro

método GetFirst() no componente gerenciador de dados, que faz a busca e instancia o novo objeto na memória. O método GetFirst() aparece definido da seguinte forma, no componente DP:

```
bool RefToOrd::GetFirst(void) {
        return DMLO.GetFirst( DataCli->CurrentData->CodCli ) ;
}
```

Este método dispara o método GetFirst() do objeto DMListOrders, instanciado aqui no exemplo como DMLO, no componente GD. Tal método possui o seguinte código em DMLO:

```
bool DMListOrders::GetFirst(int codcli) {
      if ( LastCli != codcli ) {
      LastCli = codcli ;
     RemoveAll() ;
   }
      Set <TLocateOption, 0, 1> Sensitive ;
      Banco->Order->Open() ;
      bool Sucesso = Banco->Order->Locate("CodCli", codcli, Sensitive) ;

      int Pedido = int(Banco->Order->FieldValues["CodOrder"]) ;
      int aux = DMLOPK.LocateCod( Pedido, NULL ) ;
      if ( aux == NULL ) {
        DMLOPK.Append() ;
        DMLOPK.CurrentData->PK = Pedido ;
      }
      else DMLOPK.GoTo( aux ) ;

      pDMO->Load( Pedido ) ;
      Banco->Order->Close() ;
       return Sucesso ;
}
```

Toda a codificação a nível de acesso a banco de dados é feita na classe DMLO, para que o programador não necessite obter o conhecimento de tais procedimentos.

Método GetNext()

Retorna o próximo elemento do conjunto. Assim como em GetFirst(), possui o método definido no componente DP, porém a sua execução é feita em DMLO. A definição do método no componente DP:

```
bool RefToOrd::GetNext(void) {
        return DMLO.GetNext(DataCli->CurrentData->CodCli)  ;
}
```

Método GetPrevious()

O método GetPrevious() carrega na memória o elemento anterior ao atual. A definição deste método no componente DP:

```
bool RefToOrd::GetPrevious(void) {
      return DMLO.GetPrevious(DataCli->CurrentData->CodCli) ;
}
```

Método GetLast()

O método GetLast() tem como função obter o último elemento da lista. Este método obtém o último pedido da lista, e instancia o objeto correspondente na memória. A definição do método no componente DP específico:

```
bool RefToOrd::GetLast(void) {
      return DMLO.GetLast(DataCli->CurrentData->CodCli) ;
}
```

Método Remove()

Remove o elemento da posição atual. Este método permite que seja removido o objeto da lista, na posição corrente. A definição do método no componente DP:

```
bool RefToOrd::Remove(void) {
      return DMLO.Remove() ;
}
```

A definição deste método na classe DMLO do componente GD é:

```
bool DMListOrders::Remove(void) {
      bool Sucesso = (DMLOPK.nObjects > 0) ? true : false ;
   pDMO->Destruct() ;
   DMLOPK.Delete() ;

      return Sucesso ;
}
```

Método RemoveAll()

Remove todos os elementos do conjunto. Como os demais método do componente DP, na aplicação é definido apenas a classe que vai executar a instrução propriamente dita:

```
bool RefToOrd::RemoveAll(void) {
      return DMLO.RemoveAll() ;
}
```

A definição do método no componente GD:

```
bool DMListOrders::RemoveAll(void) {
      bool Sucesso = (DMLOPK.nObjects > 0) ? true : false ;
```

```
    while ( DMLOPK.nObjects > 0 ) {
            pDMO->Destruct() ;
            DMLOPK.Delete()  ;
    }
        return Sucesso ;
}
```

Este método vai eliminando todos os componentes da lista de pedidos, até que o número de objetos da lista seja igual a zero.

Basicamente, uma lista persistente é uma coleção de objetos persistentes. Poderemos carregar, adicionar, modificar ou remover membros desta lista, chamando no final o método Save( ) para garantir todas as modificações.

Condições podem ser impostas aos objetos que pertencem a um conjunto. Tais condições identificam este conjunto. Por exemplo, uma lista de Pedidos em atrasos na entrega, poderia ser um lista associada a um determinado cliente, sendo que esta lista apresenta uma condição específica que é o atraso na entrega.

Normalmente será atribuído a todos os objetos o mesmo valor para uma determinada chave estrangeira. Deste modo, o que se obtém, é uma classe que representa um relacionamento 1 para muitos entre duas tabelas ou classes.

5 A Arquitetura que implementa a persistência

Este capítulo descreve o componente gerenciador de dados, componente responsável pela implementação dos serviços de persistência vistos no capítulo anterior. Para isto, serão estudados alguns conceitos básicos para o projeto de um *framework* orientado a objetos, e alguns padrões de projeto OO que são utilizados neste componente.

A intenção, nesta parte inicial, é usar o problema de persistência como um veículo para explicar as questões gerais para um projeto de *framework*, além de algumas questões críticas de serviços de persistência, antes de entrar no detalhamento da arquitetura de persistência.

Sendo assim, o problema principal a ser estudado no momento são os objetos persistentes. Objetos persistentes em uma aplicação OO são todos aqueles objetos que se necessita manter suas informações após o término da aplicação. Suponha que em uma aplicação de Faturamento, pode ser por exemplo, um ponto de vendas, se exija que um grande número de produtos existam em algum mecanismo de armazenamento persistente, como um banco de dados, e eles devam ser carregados para a memória local durante o uso da aplicação. Neste exemplo, os objetos persistentes são as instâncias dos produtos, que requerem armazenamento persistente.

O objetivo neste momento é identificar como projetar serviços de persistência a objetos, pois este é um problema muito comum, e para isto será necessário conhecer alguns padrões e princípios de projeto orientado a objetos.

A solução para o problema exposto está na utilização de um *framework* de persistência. Este *framework* é um conjunto de classes reusáveis e extensível, que fornecem serviços para os objetos persistentes. Normalmente são escritos para trabalhar com banco de dados relacionais. De uma forma simplificada, um *framework* de persistência deve traduzir objetos para registros (tuplas) e salvá-los em uma base de dados, e traduzir registros para objetos quando estes forem recuperados da base de dados.

Em relação ao dispositivo de armazenamento, se um SGBDOO é usado, nenhum *framework* de persistência é necessário. No entanto, se outro mecanismo for utilizado, este *framework* será altamente desejável.

5.1 Características do *Framework*

Frameworks são conjuntos de classes interrelacionadas, as quais têm abstrações comuns a todas as aplicações que pertencem a um mesmo domínio. É um subsistema que tem como objetivo principal ter suas classes reutilizadas e estendidas. Um dos tipos de *frameworks* existentes, os *frameworks* de utilidade, são genéricos para muitos domínios de aplicações, tais como *framework* de interface gráfica para o usuário, como por

exemplo o Microsoft Foundation Class ou o Smalltalk-80 Model-View-Controller. Neste grupo, insere-se os *frameworks* de persistência, ou seja, *frameworks* cujos objetivos são de prestar serviços a objetos persistentes.

O *framework* de persistência projetado neste trabalho, ilustrará claramente as características de um *framework*, que de uma forma geral, são as seguintes:

- Possui um conjunto de classes coesas, que colaboram para fornecer serviços;
- Possui classes concretas e especialmente classes abstratas, que definem interfaces aos objetos que participarão de interações;
- Normalmente requer ao usuário do *framework* que este defina subclasses as classes existentes no *framework*, a fim de estender e customizar seus serviços;
- As classes abstratas podem conter métodos abstratos e métodos concretos;
- Os métodos abstratos definidos no *framework* enviam mensagens as subclasses definidas pelo usuário, ou seja, métodos *templates* definidos cujas operações primitivas são definidas nas subclasses pelo usuário do *framework*.

Como foi visto, os *frameworks* fornecem um alto grau de reuso, muito mais do que classes individuais. Consequentemente, os desenvolvedores que estejam interessados em aumentar o grau de reuso de seus softwares, devem enfatizar a criação de seus próprios *frameworks*.

Em particular, um *framework* de persistência deve fornecer as seguintes funções:

- Armazenar e recuperar objetos em um mecanismo de armazenamento persistente;
- Efetuar o controle sobre as transações, com a utilização de comandos de *commit* e *rollback*.

Além disso, o projeto de um *framework* deve suportar algumas qualidades, como:

- Ser extensível para suportar outros tipos de mecanismos de armazenamento;
- Requerer um mínimo de modificações ao código existente;
- Ser de fácil utilização pelo desenvolvedor;
- Ser o mais transparente possível.

Uma aplicação com *framework* pode ser vista como a figura abaixo [CAM94]. O nível superior representa as classes do *framework*, as quais definem a estrutura de controle da aplicação. O nível inferior representa as subclasses concretas da aplicação do usuário, o qual fornece a implementação de operações específicas cuja ativação é modelada no nível superior.

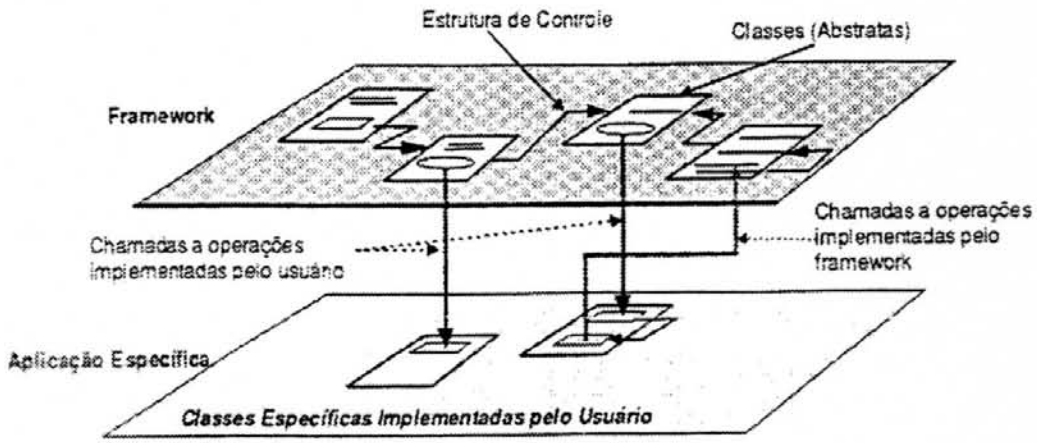

FIGURA 5.1 - Visão de uma aplicação construída com frameworks [ZAN97]

Uma das soluções de projeto para fornecer persistência a objetos é a criação de uma superclasse de objetos persistentes, que fornecerá através de herança métodos que supram os serviços de persistência. Esta opção de projeto foi utilizada na arquitetura que está sendo proposta neste trabalho. Porém, existem autores que afirmam, que apesar de não estar incorreta, esta opção de projeto provoca um forte acoplamento das subclasses do domínio do problema com a superclasse genérica, o que seria um ponto desfavorável[LAR98].

5.2 Projeto do *Framework*

Está seção descreve duas das características essências no projeto do *framework* de persistência. Uma delas é determinar quem deve ser responsável pela materialização e desmaterialização dos objetos a partir de um dispositivo de armazenamento persistente. O capítulo anterior descreveu com detalhes as responsabilidades da camada Gerenciadora de dados. Como foi visto, esta camada ficará responsável pelo acesso aos dados, seguindo o padrão *Database Broker* [BRO96], que propõem a criação de uma classe que é responsável pela materialização, desmaterialização e pelo cache de objetos. O padrão também sugere a utilização de uma classe para cada classe de objeto persistente.

A segunda característica de projeto é baseada no padrão *Template Method*[GHJ95]. Este padrão é a peça fundamental do projeto do *framework*. A idéia é definir um método na superclasse que define o esqueleto de um algoritmo, com suas partes variáveis e invariáveis. O método *Template* invoca outros métodos, alguns dos quais são operações que podem ser sobrescritas nas subclasses. Desta forma, as subclasses podem rescrever a parte variável dos métodos, adicionando seu comportamento próprio. A figura 5.2 abaixo ilustra o padrão descrito. A classe concreta rescreve o método OperaçãoPrimitiva( ). Este será chamado automaticamente quando o método herdado MétodoTemplate( ) for invocado.

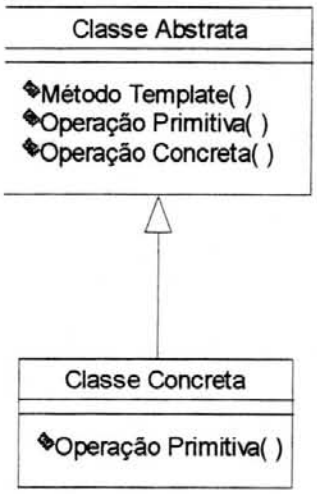

FIGURA 5.2 - Padrão *Template*

No método OperaçãoConcreta( ), será colocado todo o comportamento padrão do *framework*. Ele pode ou não ser rescrito na subclasse herdada. Se ele for rescrito, fica então denominado de método hook [LAR98]. A implementação do método template poderia ser:

```
MétodoTemplate
{
....
OperaçãoPrimitiva( )
....
OperaçãoConcreta( )
....
}
```

5.3 O Processo de Materialização

Conforme é mostrado na figura abaixo, o processo de materialização envolve basicamente a criação de uma instância de uma classe apropriada, com a passagem de dados do registro para os atributos da nova instância. Por exemplo, o campo data de uma linha da tabela de Vendas pode ser copiado para o atributo data da instância Venda associada.

A hierarquia de classes da camada gerenciadora de dados é uma parte essencial do *framework*, pois novas subclasses podem ser adicionadas pelo desenvolvedor da aplicação para customizar um novo tipo de dispositivo de armazenamento persistente, ou um outro sistema gerenciador de banco de dados, ou ainda, simplesmente novas tabelas que necessitem os serviços de persistência. Por exemplo, pode existir a mesma versão da aplicação, rodando uma sobre o banco de dados Oracle e outra sobre o banco de dados Sybase, trocando apenas a camada gerenciadora de dados.

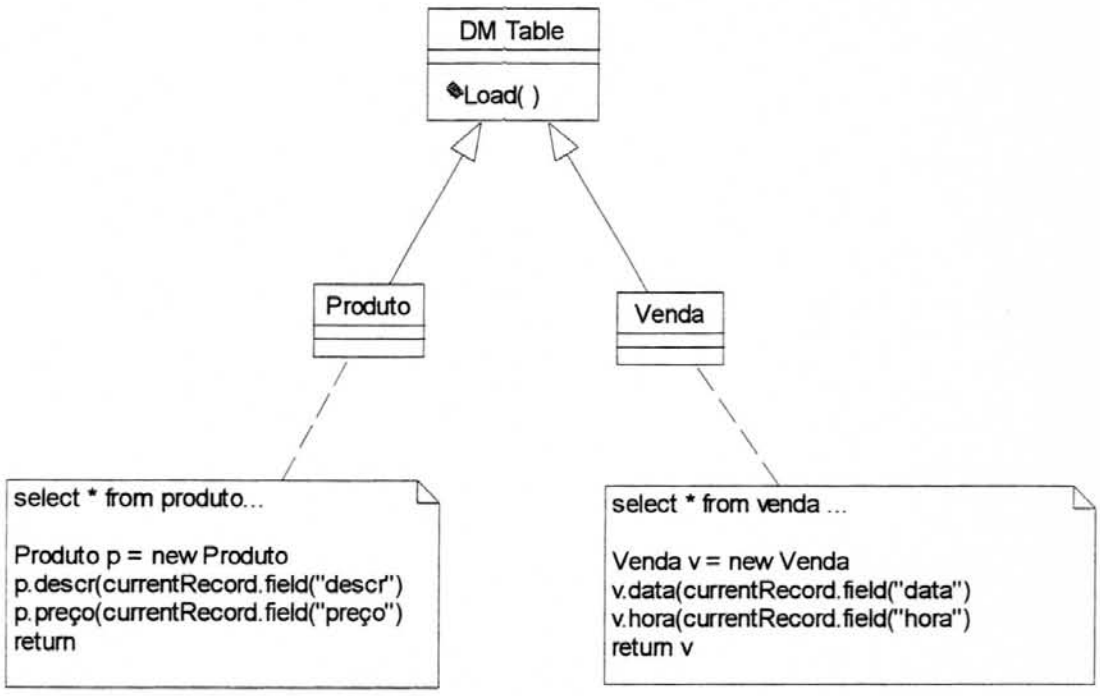

FIGURA 5.3 - Método *Template* na camada Gerenciadora de Dados

Os comandos necessários para a materialização do objeto produto e objeto venda são adicionados nas subclasses de DMTable, que conhece todas as classes que representam tabelas do banco de dados.

5.4 Cache de Objetos

É desejável manter os objetos materializados em um cache local para poder melhorar a performance, pois o processo de materialização é relativamente lento, e para poder suportar operações de gerência de transações, tais como um commit.

O Padrão *Cache Management* [BRO96] propõem que a camada gerenciadora se torne responsável pela manutenção deste cache. Se for utilizado uma gerenciadora para cada classe de objeto persistente, então cada gerenciador mantém seu próprio cache. Este padrão é utilizado integralmente, pois cada DMTable possui associado um DMTuple, que apresenta as funções de cache na arquitetura do *framework*.

Quando objetos são materializados, eles são colocados dentro deste cache, com sua chave primária, que é seu identificador. Requisições subsequentes a um objeto já materializado, fará com que o gerenciador primeiro pesquise na área de cache, evitando com isto materializações desnecessárias.

A área de cache também pode ser utilizada para a gerência de transações. Manter os objetos em diferentes caches, dependendo de seu estado no contexto de uma transação corrente, é uma pequena variação que pode ser utilizada no uso de caches.

Dentro desta abordagem, seis caches podem ser mantidos pelo gerenciador [LAR98]. Isto fornece a base para determinar como os objetos devem ser atualizados ou não na base de dados ao término de uma transação. As seis variações de caches são:

- Cache nova e limpa, para os objetos criados novos e não modificados;
- Cache velha e limpa, para os objetos velhos, ou seja, aqueles que foram materializados a partir do banco de dados e não foram modificados;
- Cache nova e suja, para objetos criados novos e já modificados;
- Cache velha e suja, para objetos velhos e modificados;
- Cache nova e removida, para objetos novos e já removidos da memória;
- Cache velha e removida, para objetos materializados e removidos.

As operações que acontecem em cada situação acima serão descritas mais adiante neste capítulo.

5.5 Materialização sob Demanda

Como foi citado anteriormente, normalmente é desejável atrasar a materialização de um objeto, até que esta seja absolutamente necessária. Isto é conhecido como materialização sob demanda. Para se implementar este conceito usa-se o padrão *Virtual Proxy*, que é uma das variações do padrão *Proxy* [GHV95]. Este padrão também é uma especialização de outro padrão, conhecido como padrão *Bridge* [GHJ95].

O padrão *Bridge* é uma referência inteligente para outro objeto, sendo que este outro objeto é o objeto real. A materialização do objeto real ocorre quando ele é referenciado pela primeira vez, ou seja, o objeto *bridge* pode referenciar um objeto que pode estar ou não materializado.

Na arquitetura proposta, a funcionalidade de um objeto *bridge* está incorporada na classe PDPersist, que é superclasse de todos os objetos do domínio do problema.

Como é mostrado na figura 5.4, um objeto de interface possui uma referência para o objeto PDPersist, que por sua vez se relaciona com objeto DMTable. Para a interface, não existe a preocupação se o objeto está ou não materializado, pois está é uma das responsabilidades do objeto PDPersist.

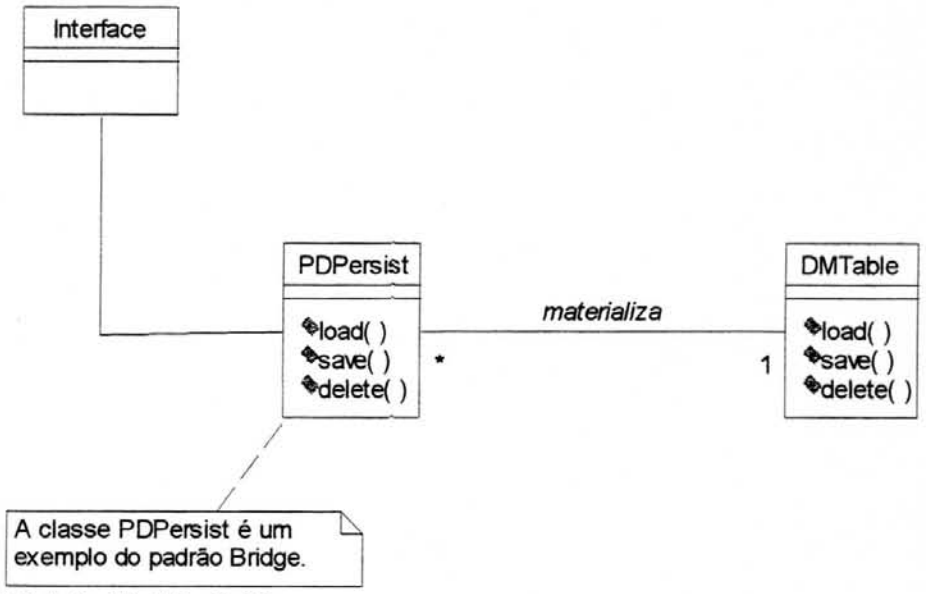

FIGURA 5.4 - Padrão *Bridge*

Um exemplo concreto da utilização deste padrão pode ser visto na figura 5.5. Este exemplo utiliza as classes ItemPedido e Produto. Este tipo de projeto parte do princípio de que o objeto herdado de DPPersist conhece o identificador de seu objeto GD, e quando a materialização é requerida, o identificador será utilizado para recuperar o objeto da base de dados.

Note que a classe ItemVenda possui um atributo que necessita de uma instância da classe Produto, e esta por sua vez, pode ainda não estar materializada na memória. Quando o objeto ItemVenda envia a mensagem descrição para o objeto PDProduto, este materializa o Produto, usando o seu identificador.

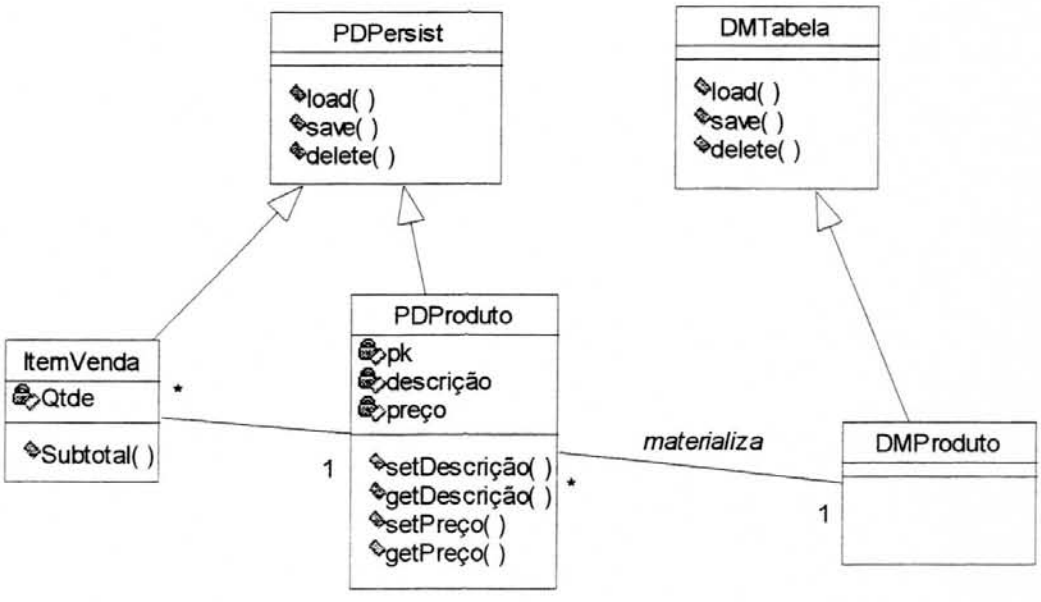

FIGURA 5.5 - Aplicação exemplo para o padrão *bridge*

5.6 O Componente Domínio do Problema e o Gerenciador de dados

O componente DP pode colaborar com o componente gerenciador com a finalidade de materializar um objeto, baseado no identificador mantido no DP, conforme pode ser visto na figura 5.6 abaixo.

FIGURA 5.6 - Colaboração entre o DP e o GD

Quando o método getObject( ) for acionado, ele verifica se o objeto real já foi materializado, caso contrário, ele dispara o método load( ), responsável pela materialização, que será executado no gerenciador de dados.

Um dos problemas a serem resolvidos neste momento, é como uma classe concreta DPpersist sabe qual é sua classe gerenciadora em particular. Para resolver este problema, é aplicado o padrão *Factory*[GHJ95]. Este padrão, mostrado na figura 5.7, é um caso especial do padrão *Template Method*, na qual, uma operação primitiva, que será o método *Factory*, é responsável pela criação da instância. A utilização deste padrão na construção de *frameworks* é bastante comum[LAR98]. O método *Factory* createGD usa o padrão *Singleton*[GHJ95] para retornar o objeto gerenciador, uma vez que somente uma instância de cada gerenciador é desejável.

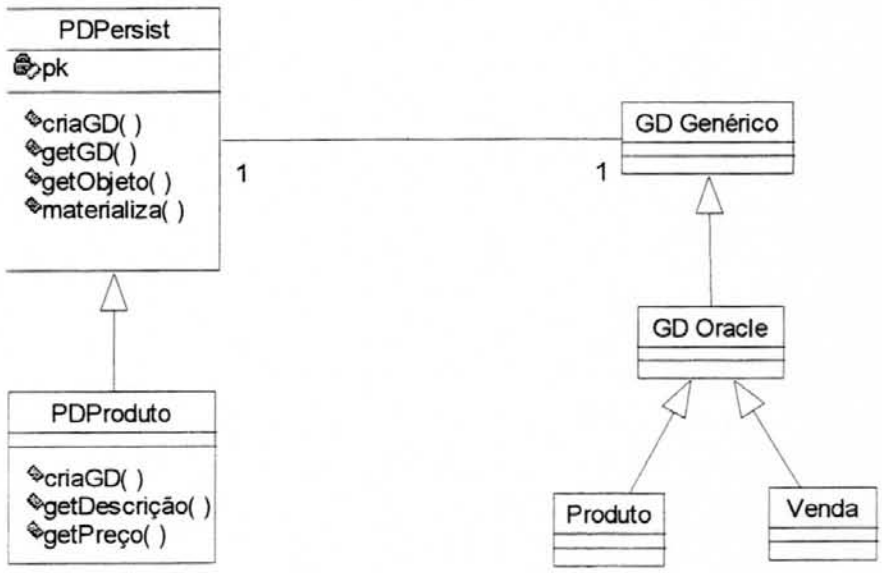

FIGURA 5.7 - Padrão *Factory*

5.7 O Componente Gerenciador de Dados

Cada objeto GD possui atributos e é capaz de realizar ações. Cada um conhece todos os objetos dentro de sua classe Domínio do Problema (DP) correspondente. Exemplificando, a classe Cliente_GD conhece todos os objetos da classe Cliente_PD.

Os objetos deste componente sabem pesquisar, e são capazes de responder perguntas sobre um objeto específico. São capazes também de carregar e salvar informações nos atributos. Eles carregam informações dos dispositivos de armazenamento, criando e inicializando objetos na memória principal. Eles salvam objetos, convertendo-os para formatos específicos de armazenamento.

Normalmente, se usa os objetos GD quando:

- for necessária a persistência do objeto DP;
- for necessário pesquisar e recuperar objetos DP persistentes;
- for necessário trabalhar com algum outro sistema gerenciador de banco de dados, que não seja um sistema gerenciador de banco de dados orientado a objetos.

Os objetos GD isolam a complexidade do gerenciamento de dados do resto da aplicação, através do encapsulamento de como armazenar e como recuperar objetos. Ele interage com o meio de armazenamento, e quando necessário, ele interage com objetos presentes em outros sistemas.

5.7.1 Composição do Componente GD

Os objetos que fazem parte do gerenciador de dados são:

- os objetos GD ;
- objeto servidor de GD;

5.7.1.1 Objetos GD

O propósito principal dos objetos GD é de que cada objeto GD serve a uma classe específica de objeto do DP, fornecendo:

- armazenamento persistente para uma única classe;
- isolamento do mecanismo de armazenamento de dados;
- um lugar para a colocação de índices, mecanismos de pesquisa;
- um buffer de objetos já carregados da fonte de dados;
- conexão com outros objetos.

As necessidades que devem ser supridas por um objeto GD, dependem muito das necessidades da aplicação e da tecnologia de gerenciamento de dados que está disponível.

Cada objeto GD deve ser responsável por:

- quais são seus conhecimentos:

  - índices, dicionários
  - informações necessárias para conversão de um objeto para outro formato, como linhas, registros e outros.

- com quem se relaciona:

  - a classe que está sendo servida
  - todos os objetos desta classe
  - servidor de objetos

- quais suas funções:

  - pesquisar e buscar objetos simples, coleções de objetos, pesquisar por múltiplos tipos de critérios, manter índices.
  - salvar em dispositivos persistentes
  - remover destes dispositivos
  - fazer a carga de todos os objetos
  - liberar a memória destes objetos
  - converter objetos para linhas e registros
  - converter linhas e registros para objetos
  - fazer o trabalho sobre coleções de objetos usando o suporte de um gerenciador de dados

- quando incluir os objetos GD em um cenário:

  - quando os objetos do DP necessitem de suporte para salvar, recuperar e pesquisar
  - quando os objetos da IH necessitem deste mesmo suporte

5.7.1.2 Objeto Servidor de GD

O Objeto Servidor de GD é um, e somente um, dentro de um sistema. Seus propósitos são:

- criar os objetos GD para cada classe suportada
- manter um conjunto de objetos GD
- fornecer um ponto único de acesso para obter objeto GD para uma classe específica.

As responsabilidades atribuídas ao Objeto Servidor de GD são:

- conhecer o próximo ID do objeto disponível
- conhecer todos os objetos servidores
- obter objetos GD para as classes
- fornecer um controle para todas as transações sobre objetos, com gerência sobre os comandos de start, rollback e commit
- obter o próximo ID do objeto

Além disso, o objeto servidor também participa dos cenários, quando um objeto DP ou IH necessitam saber qual é o objeto GD para uma determinada classe, ou quando um objeto DP ou IH necessitam suporte para controle de transação (*start*, *rollback*, *commit*).

5.7.2 Usando a Arquitetura GD com um SGBDR

A seguir estão descritas algumas observações que devem ser levadas em consideração em relação a implementação de persistência, pesquisas, e observações com o uso da arquitetura cliente servidor.

Em relação a implementação da persistência, observar:

- quanto ao ato de criar:

  - criar, exatamente como qualquer outro objeto.
  - nenhuma persistência envolvida.
  - uma classe de objetos necessitando suporte de GD herda quaisquer capacidades que ela necessite.

- quanto ao ato de salvar:

  - normalmente feito manualmente.
  - feito quando existe uma requisição específica a um objeto GD, ou a um objeto servidor GD.
  - requer conversão para linhas em uma ou mais tabelas.
  - usa transações e controles de commit fornecidos pelo SGBDR.
  - deve ser adicionado uma maneira de fazer rollback em objetos, pois não existe uma maneira automática para fazer isto.

- quanto ao ato de carregar:

  - normalmente feito manualmente.
  - feito quando existe uma requisição específica a um objeto GD, ou a um objeto servidor GD.
  - requer chamadas SQL.
  - reconstrói objetos a partir de linhas em uma ou mais tabelas.

Em relação a implementação da pesquisa, observar:

- criar pedidos de pesquisa em SQL.
- usar a pesquisa do SGBDR.
- necessidade de criar uma maneira para converter linhas de uma ou mais tabelas em objetos.
- necessidade de criar uma maneira de manusear o retorno de múltiplas linhas.

Em relação a implementação da arquitetura cliente-servidor, observar:

- requer muito pouca ou nenhuma alteração para implementar.
- necessidade de se adicionar cuidados com concorrência no acesso multiusuário.
- somente move a comunicação SQL para um sistema remoto.
- suporte de comunicação é fornecido pelo banco de dados.
- não há necessidade de um ambiente orientado a objetos no servidor.

5.8 Detalhamento do componente Gerenciador de Dados

Neste componente, como já foi visto, ficam definidos e armazenados os mapeamentos entre as classes e o banco de dados, ou seja, o mapeamento OO-Relacional. Este componente é formado por classes genéricas, que estabelecem a associação com classes da componente do domínio do problema, e por classes que correspondem a construções do banco de dados relacional, como tabelas e tuplas. A tarefa de mapear as classes deve ser desenvolvida de forma manual, por um administrador de dados.

A principal utilidade do componente gerenciador de dados é permitir que se possua uma implementação da classe do domínio de problema independente da implementação de sua persistência. Os métodos abstratos de Load( ), Save( ), Delete( ), são chamados pelos métodos homônimos da classe DPPersist. A seguir será descrito a estrutura de classes deste componente, e seus principais comportamentos.

5.8.1 Estrutura de Classes do Componente GD

As classes que compõe o componente GD-genérico são as classes abstratas DMDatabase, DMTransação, DMTable, DMTuple e DMFK, das quais são derivadas todas as classes persistentes da GD-específica, definindo suas interfaces.

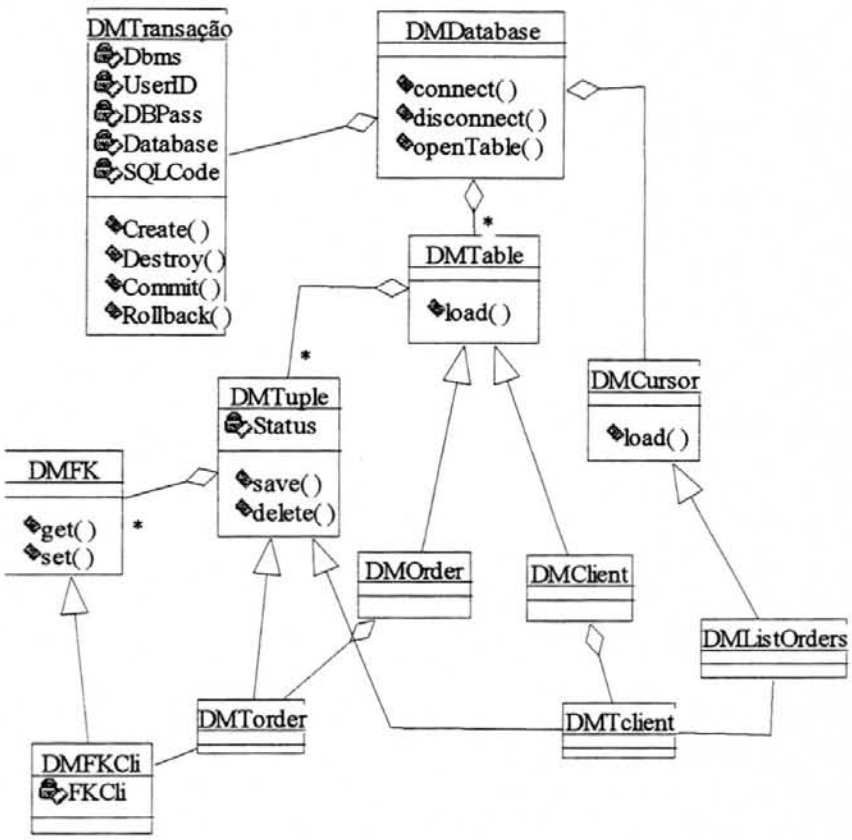

FIGURA 5.8 - Componente GD

A figura 5.8 mostrada acima apresenta as classes abstratas do componente gerenciador de dados, e as classes de usuários que devem ser implementadas por um administrador de dados, pois contém o mapeamento entre os modelos OO e relacional. A seguir, é realizada uma breve descrição das classes e dos métodos (funções membro abstratas) que aparecem a nível de Gerenciadora de Dados.

Classe DMDatabase

A classe DMDatabase é a classe de nível mais alto do componente gerenciador de dados. Possui uma agregação com todas as tabelas do banco de dados em uso pela aplicação. Tem a função de abrir e fechar o banco de dados, estabelecendo uma conexão ou a desconexão ao banco de dados, através dos métodos connect( ) e disconnect( ), respectivamente. Sua codificação está assim definida:

```
class DMDataBase {
        void Connect(void)      ;
        void Disconnect(void)   ;
        void OpenTable(void)    ;
} ;
```

Classe DMTable

A classe DMTable é a classe do componente gerenciador de dados de onde cada classe persistente do modelo de objetos deve herdar os métodos abstratos para leitura nas

tabelas relacionais. Esta classe possui somente o método abstrato Load( ), que será explicado logo a seguir. Sua codificação está assim definida:

```
class DMTable {
      void Load( )          ;
} ;
```

Classe DMTuple

A classe DMTuple é responsável por manter as informações sobre as tuplas lidas no banco de dados e trazidas para memória. Será criado um objeto DMTuple para cada tupla lida do banco de dados. No exemplo da figura anterior, as classes DMTOrder e DMTClient especializadas de DMTuple, mantém a chave primária de cada tupla em seus respectivos objetos DMTuple, e o status, que é um atributo que varia de acordo com as modificações feitas sobre os dados da aplicação, podendo conter as seguintes informações:

- NotModified - o conteúdo da tupla não foi modificado;
- Modified - conteúdo da tupla foi modificado e deverá ser atualizado no banco de dados;
- New - foi instanciado um novo objeto na aplicação, que dará origem a uma nova tupla no banco de dados.

Estas foram as opções implementadas, podendo existir outras. A definição da classe no *framework:*

```
class DMTuple {
      void Save(void)   ;
      void Delete(void)       ;
} ;
```

Classe DMFK

A classe DMFK é a classe do componente gerenciador de dados de onde cada classe de relacionamento deve herdar os métodos abstratos de obtenção e atribuição de suas chaves estrangeiras, através dos métodos get( ) e set( ). Deve também manter as chaves estrangeiras da tabela, pois estas serão mantidas no modelo OO como ponteiros, para que seja possível recuperar dados no banco. A definição da classe DMFK no *framework:*

```
class DMTFK {
      void Get(void)    ;
      bool Set(int)     ;
} ;
```

Classe DMTransação

A classe DMTransação especifica os parâmetros que a aplicação usa para se conectar ao banco de dados. Antes de utilizar um objeto desta classe, deve-se dar valores aos atributos que serão usados. Os atributos da classe DMTransação são descritos a seguir:

- Dbms - Identificação do servidor de banco de dados.
- UserId - O nome ou a identificação do usuário que se conectará ao banco de dados.
- DBPass - A senha que será usada para conectar-se ao banco de dados.

- Database - O nome do banco de dados que se está conectando.
- SqlCode - Mantém o código de erro do banco de dados da operação mais recente.

Os métodos pertencentes a classe DMTransação são :

- Create - Utilizado para criar um objeto de transação, gerando uma nova instância.
- Destroy - Utilizado para destruir um objeto de transação
- Commit - Comando que fecha uma transação sobre o banco de dados com sucesso.
- Rollback - Utilizado para desfazer operações que não ocorreram com sucesso em uma transação.

5.8.2 Comportamentos do Componente GD

Os métodos que estabelecem o comportamento do componente gerenciador de dados serão descritos a seguir, organizados por classes, uma vez que já se conhece a estrutura das classes do componente.

Classe DMDatabase
Sua principal função é estabelecer a conexão com o banco de dados. Conectar-se a um banco de dados significa poder trabalhar com tabelas e visões armazenadas no banco. Para se estabelecer ou mudar uma conexão a um banco de dados usa-se os métodos connect( ) e disconnect( ). Esta conexão é feita através do ODBC (Open Database Connectivity), que é um padrão de interface de programação de aplicação (API) desenvolvida pela Microsoft. Este padrão permite a uma aplicação acessar a uma variedade de dados para os quais existir um driver ODBC. A aplicação usa o SQL como padrão da linguagem de acesso aos dados. Aplicações que provêem uma interface ODBC, podem acessar dados para quaisquer drivers ODBC existentes.

Após estar conectada a uma base de dados, o método OpenTable( ) deve ser executado. Sua função é instanciar as classes específicas da gerenciadora de dados, preparando as classes deste componente para estabelecer associações com as classes do domínio do problema. No exemplo da figura anterior, as classes instanciadas são DMOrder e DMClient.

Classe DMTable
A classe DMTable apresenta o método abstrato Load( ). Este método será especializado pelas classes específicas e deverá ser então escrito com programação em SQL, seguindo as características específicas de cada banco de dados e contendo as regras de mapeamento entre os modelos relacional e orientado a objetos. A funcionalidade do método Load( ) já foi explicada na seção 5.2. Na figura anterior, as classes DMClient e DMOrder correspondem as classes DPClient e DPOrder do componente domínio do problema, e as tabelas Client e Order armazenadas do banco de dados.

Classe DMTuple
A classe DMTuple apresenta os métodos abstratos Save( ) e Delete( ). Estes métodos implementam as operações possíveis sobre a tupla. Da mesma forma que o método

Load( ) anterior, estes métodos devem ser codificados em linguagem SQL, e suas funcionalidades já foram descritas no componente domínio do problema.

A classe DMTuple possui um atributo chamado Status que varia de acordo com as modificações feitas sobre os dados na aplicação, podendo conter as seguintes informações:

- NotModified - o conteúdo da tupla não foi modificado.
- DataModified - o conteúdo da tupla foi modificado, portanto deverá ser atualizado no banco de dados.
- Deleted - a tupla, que deu origem a um objeto na aplicação, foi removida pelo usuário, devendo ser removida do banco de dados.
- NewModified - foi instanciado um novo objeto na aplicação, que dará origem a uma nova tupla no banco de dados.

Classe DMFK

Chaves estrangeiras do modelo relacional devem ser traduzidas para a arquitetura através da utilização das classes DMFK, ou seja, gerenciadora de chaves estrangeiras. No exemplo da figura anterior, a chave estrangeira CodCli da tabela Pedido foi mapeada para a classe DMFKCli. Estas classes herdam os métodos abstratos get( ) e set( ), que devem ser codificados em linguagem SQL. Estes métodos devem conhecer o relacionamento entre as tabelas do modelo relacional.

Para exemplificar, vejamos a seguinte situação: A classe Pedido do domínio do problema quer instanciar o objeto cliente deste pedido. As seguintes etapas deverão ser cumpridas:

- método RefToCli.get( ) deve ser executado, conforme visto na seção 4.3.3.2 em Tratamento de Associações.
- este método fará a chamada ao método DMFKCli.get( ), que por sua vez deve executar:

    - um teste ao atributo DMFKCli.FKCli. Caso possua um valor, significa que o pedido possui um cliente relacionado.
    - disparar uma chamada ao método Load( ) da classe DMClient.
    - receber o endereço de memória do objeto cliente instanciado.
    - atualizar o ponteiro RefToCli.pMyClient

Com isso, um objeto do domínio do problema pode instanciar qualquer objeto associado, proveniente de um relacionamento entre tabelas do modelo relacional.

Classe DMTransação

Quando uma operação de commit é invocada pela aplicação, objetos serão tratados de forma diferente, dependendo do seu estado na transação. Por exemplo, se um objeto foi materializado do banco de dados, mas não foi modificado, não será necessário rescrevê-lo de volta no banco. De forma contrária, se ele foi modificado, seu registro associado no banco de dados deve ser atualizado.

Os possíveis estados de um objeto durante a transação, correspondem as possíveis caches analisadas na seção 5.4. Seriam:

1. Objeto novo e limpo, ou seja, não modificado;
2. Objeto velho materializado da base e não modificado;
3. Objeto novo e sujo, ou seja, já com modificações;
4. Objeto velho e sujo;
5. Objeto novo e removido;
6. Objeto velho e removido.

Um objeto torna-se sujo se qualquer um de seus atributos for modificado. O meio mais comum para sinalizar que um objeto tornou-se sujo é quando o método de atribuição for utilizado.

O gerenciador será notificado que um determinado objeto tornou-se sujo. Com isto, ele procura pela DMTuple apropriada para esta classe de objeto, e notifica que o objeto sofreu alterações.

Quando for decidido pelo commit da transação, uma mensagem de commit é enviada ao gerenciador. Embora qualquer objeto possa enviar esta mensagem, é mais natural que um gerente de aplicação tenha esta responsabilidade. Com esta mensagem, o gerenciador simplesmente a reenvia para cada DMTuple.

Durante uma transação, objetos podem ser modificados, criados e removidos. Assumindo-se que cada objeto possua seu próprio atributo de estado de transação, a seguir são descritas as ações que devem ser tomadas a partir de uma operação de commit:

Atributo NewModified
- Inserir no banco de dados
- Alterar o atributo para NotModified

Atributo NotModified
- Ignorar, pois ele não sofreu alteração

Atributo NewModified
- Inserir no banco de dados
- Alterar o atributo para NotModified

Atributo DataModified
- Atualizar o banco de dados
- Alterar o atributo para NotModified

Atributo Deleted
- Remover do banco de dados
- Remover da cache

Quando for decidido executar uma operação de *rollback* na transação, uma mensagem de *rollback* é enviada para o gerenciador. Da mesma forma que na operação

commit, esta responsabilidade deveria ser atribuída a um gerente de aplicação, e esta mensagem será redistribuída para todas as DMTuple.

A seguir são descritas as ações que devem ser tomadas a partir de uma operação de *rollback*:

Atributo NotModified
- Ignorar, pois os objetos não sofreram modificações

Qualquer outro estado
- Remover da cache

Uma das vantagens do uso de referência inteligentes a objetos em oposição ao uso de referência direta, é o efeito no caso de uma operação de *rollback*. Depois que a cache é *flushed*, todas as referências apontarão para objetos não materializados, ao invés de referenciar objetos modificados na memória local. Na próxima utilização da referência a um objeto antigo, obrigará o gerenciador a materializar novamente o objeto a partir do banco de dados.

5.8.3 Componente gerenciador de dados específico

O componente GD específico contém classes que, cada uma, corresponde a uma construção (tabela, tupla, chave estrangeira) da base de dados relacional. Os métodos destas classes realizam o mapeamento entre os modelos e contém código SQL específico para cada operação a ser realizada sobre a base de dados. Essas classes são de responsabilidade de um administrador de dados, que deve desenvolver os métodos de forma manual.

As classes do componente GD específico fazem a busca de informações no BD e repassam para a aplicação que disparou seus métodos. Para exemplificar sua utilização, foram criadas duas classes, clientes e pedidos, definidas na aplicação do componente DP específico, e necessárias no componente GD específico. As classes do componente GD específicos são mostradas na figura 5.9.

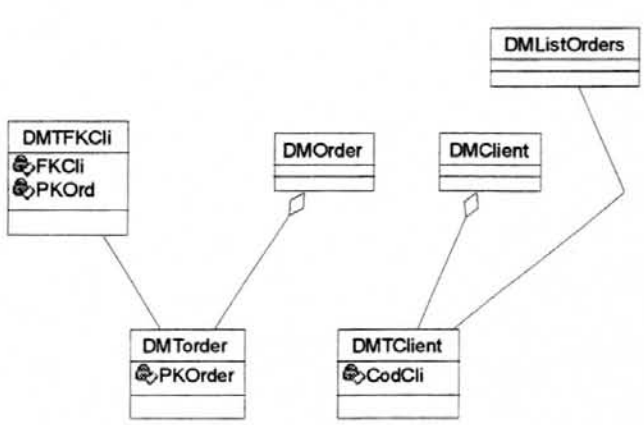

FIGURA 5.9 - Classes do componente gerenciador de dados específico

A classe DMClient faz o mapeamento entre o modelo de objetos e o modelo relacional, fazendo a interface entre a tabela Client do BD relacional e a classe PDClient do componente DP específico.

Esta classe possui métodos específicos, que acessam o banco de dados, carregam as tuplas necessárias e as repassam para o componente DP, bem como fazem a alteração, remoção e inclusão de novas tuplas no BD. A definição da classe DMClient no *framework*:

```
class DMClient : DMTable {
public:
        TListClient * DataCli ;
        DMTClient DMTC ;

        bool Load(int) ;
        bool Save(void)    ;
        bool Destruct(void) ;
        bool Construct(void) ;
        bool Delete(void)   ;
} ;
```

A classe DMTClient representa a tupla da tabela Client que foi carregada para a aplicação. Quando uma operação é feita sobre o banco de dados, é necessário fazer uma consulta a uma determinada tupla e carregá-la para a memória para que a mesma possa ser manipulada. Desta forma, é instanciado um objeto do tipo DMTClient, que contém a chave primária da tupla, bem como o status da mesma.

Esta classe implementa o conceito de cache de objetos, pois se um objeto já estiver instanciado, não será necessário a criação de um novo objeto, e sim a recuperação de um objeto já existente na memória. A definição da classe DMTClient no *framework*:

```
class DMTClient : DMTuple {
public:
        TListPK    PKClient ;
} ;
```

A classe DMTFKCli armazena a chave estrangeira da tabela Order (CodCli). Tal classe se torna necessário quando for preciso obter o cliente de um pedido. Neste caso é feita uma consulta na classe DMTFKCli e o acesso ao banco de dados, baseado na chave estrangeira que tem como atributo. A definição da classe DMTFKCli no *framework*:

```
class DMTFKCli : DMTFK {
public:
        TListFK    FKCli      ;
        DMClient   * pDMC ;
        TListPK * pDMTO   ;

        void Get(void) ;
```

```
        bool Set(int)   ;
};
```

A classe DMListOrder é associada com a classe DMClient, de forma a guardar a lista de todos os pedidos de um determinado cliente. Essa lista possui métodos que buscam no banco de dados os pedidos de um dado cliente, cuja chave primária é passada como parâmetro. A definição da classe DMListOrder no *framework:*

```
class DMListOrders {
public:
        DMOrder * pDMO  ;
        TListPK DMLOPK  ;

        bool GetFirst   (int)     ;
        bool GetNext    (int)     ;
        bool GetPrevious(int)     ;
        bool GetLast    (int)     ;
        bool Remove     (void)    ;
        bool RemoveAll  (void)    ;
};
```

Conforme a definição da classe DMListOrders no *framework,* é possível fazer a busca do primeiro pedido e instanciá-lo na memória, bem como buscar o último, o próximo e o anterior. É possível, também, remover os pedidos da memória, fazendo com que os objetos sejam destruídos depois de utilizados.

Uma lista persistente é uma coleção de objetos persistentes, a qual pode-se carregar, adicionar, modificar ou remover membros, chamando no final o método Save() para garantir todas as modificações.

Nenhuma destas funções operam diretamente sobre o banco de dados. Uma lista persistente é um objeto persistente, com o mesmo comportamento do objeto persistente, não atualizando o banco de dados até que se invoque o método Save(). Isto significa que todos os objetos que foram adicionados, removidos ou modificados, não terão as suas modificações concretizadas no banco de dados, até que o método Save() seja invocado.

Listas persistentes serão sempre usadas como objetos estrangeiros, representando os relacionamentos 1-n. Orders é um exemplo de objeto estrangeiro da classe Client. Os identificadores de objetos Orders formam uma lista associada a um determinado cliente, pois um cliente pode ter vários pedidos, porém um pedido pode ser somente de um cliente.

6 Conclusão

Fornecer a cooperação entre aplicações OO e SGBDR é um dos problemas atuais no desenvolvimento de sistemas de informação [MAR98]. Muitas ferramentas no mercado ajudam a resolver este problema. Uma das alternativas é a utilização de *gateways*. Um *gateway* é uma camada de software intermediária que realiza o mapeamento entre o modelo OO e o modelo relacional, fornecendo um suporte ao modelo de programação OO.

Neste trabalho apresentamos a arquitetura de um *framework* destinado a construir *gateways* de persistência para aplicações OO que utilizam um SGBDR.

Uma aplicação construída pelo *framework* é composta de quatro conjuntos de classes:

- classes domínio de problema (DP) genéricas (classes *template*), providas pelo *framework*
- classes (DP) específicas, construídas pela usuário através de especialização das classes DP genéricas, contendo a funcionalidade da aplicação
- classes de gerência de dados (GD) genéricas (classes abstratas), providas pelo *framework*
- classes (GD) específicas, construídas através de especialização das classes GD genéricas.

A utilização desta arquitetura no desenvolvimento de aplicações orientadas a objetos apresenta as seguintes características:

- Isola o tratamento de persistência, separando o problema de mapeamento do modelo de objetos para o modelo relacional, através da utilização dos componentes de domínio de problema e gerenciador de dados. Esta característica simplifica o projeto do modelo de objetos de sua aplicação

- Provê mecanismos para trabalhar com objetos associados, tanto no relacionamento para um como para muitos.

- Permite separar papéis no desenvolvimento de aplicações, identificando claramente o programador da aplicação e o programador do mapeamento. Esta características é resultante da independência entre as classes persistentes e o banco de dados que as suporta.

- O modelo também ajuda na economia de memória, pois utiliza mecanismos de carga de objetos sob demanda.

- Permite implementar as classes do domínio do problema sem preocupação com a persistência, somente adicionando-as ao modelo como subclasses de PDPersist.

- Aumentam as possibilidades de reuso das classes do domínio do problema da aplicação, pois estas não possuem características nem implementações de persistência em relação a um determinado banco de dados.

- A arquitetura permite a geração automática de código para o componente gerenciador de dados a partir do esquema da base de dados a ser usada.

Na construção do protótipo foram identificados e utilizados padrões de projeto OO. O objetivo dos padrões dentro da comunidade de software é criar uma forma de literatura para auxiliar os desenvolvedores de software na resolução de problemas. Os padrões ajudam a criar um linguagem compartilhada para a comunicação da experiência sobre estes problemas e suas soluções.

Este modelo não tem a intenção de ser uma solução completa ou definitiva para o problema da integração dos modelos de objetos e seus mapeamentos com o modelo relacional, mas sim um novo ponto de vista para a solução de alguns destes problemas.

Como contribuições deste trabalho, destaco uma melhor compreensão da estrutura de um *gateway* e dos serviços de persistência prestados as classes do domínio do problema, e a oportunidade para um estudo de caso da aplicação de padrões de projeto no desenvolvimento de um *framework*.

Como trabalhos futuros, estão em execução as seguintes tarefas:

- Está em projeto uma variante do *framework* que implementa um sub-conjunto de ODMG ao invés de uma interface específica com a definida neste trabalho. Isso tem a vantagem de facilitar o porte das aplicações para outros sistemas ODMG-compatíveis.

- Está sendo definida uma ferramenta de engenharia reversa de esquemas relacionais para ODL de ODMG. Esta ferramenta tem por objetivo permitir a geração automática das classes específicas de domínio de problema e de gerência de dados a partir do esquema de uma base de dados relacional.

Bibliografia

[APP 97] APPLETON, Brad. **Patterns and Software:** essential concepts and terminology. Disponível por WWW em http://www.enteract.com/~bradapp/docs/patterns-intro.html (1997).

[BAT 92] BATINI, C.; CERI, S.; NAVATHE, S.B. **Conceptual database design.** Redwood City, CA: Benjamin/Cummings, 1992.

[BPR 88] BLAHA, M.R.; PREMERLANI, W.J.; RUMBAUGH, J.E. Relational Database Design using an Object-Oriented Methodology. **Communications of the ACM**, New York, v.31, n.4, p.414-427, Apr. 1988.

[BRO 96] BROWN, K. ; WHITENACK, B. **Pattern Languages of Program Design.** Reading, MA: Addison-Wesley, 1996.

[CAM 94] CAMPO, M. **Uma Experiência com Micro-Arquiteturas Orientadas a Objetos:** trabalho individual. Porto Alegre : CPGCC da UFRGS, 1994. (TI-379).

[CAT 94] CATTEL, R.G.G. **Object database management:** object-oriented and extended relational database systems. Reading, MA: Addison-Wesley, 1994.

[CAT 97] CATTEL, R.G.G. et al. **Object Database Standard:** ODMG 2.0.[S.l.]: Morgan Kaufman, 1997.

[CHA 97] CHANG, D.T.; SRINIVASAN, V. Object Persistence in Object-Oriented Applications. **IBM Systems Journal**, [S.l.], v.36, n.1, p.66-87, 1997.

[COA 95] COAD, Peter. **Strategies, Patterns, and Applications.** [S.l.]: Prentice Hall, 1995.

[GHJ 95] GAMMA, E. et al. **Design Patterns Elements of Reusable Object-Oriented Software.** [S.l.]: Addison-Wesley, 1995.

[ISO 94] ISO-ANSI. **Working Draft Database Language SQL/Foundation** (SQL3). New York, NY, 1994.

[JOH 96] JOHNSON, R. **How to Develop Frameworks.** [Linz, AT: s.n.], 1996. Tutorial Notes of European Conference on Object-Oriented Programming, 10., 1996.

[LAE 93] LAENDER, A.H.F.; FLYNN, D.J. A Semantic Comparison of the Modelling Capabilities of the ER and NIAM Models. In: INTERNATIONAL CONFERENCE ON ENITY-RELATIONAL APPROACH, 12., 1993, Arlington, Texas, **Proceedings...**[S.l.:s.n.], 1993.

[LAR 98] LARMAN, C. **Applying UML and Patterns:** An Introduction to Object-Oriented Analysis and Design. NJ: Prentice Hall, 1998.

[LEW 95] LEWIS, Ted et al. **Object – Oriented Application Frameworks**. [S.l.]: Manning Publications, 1995.

[LOO 95] LOOMIS, Mary E.S. **Object Databases – The Essentials**. Reading, MA: Addison-Wesley, 1995.

[MAR 95] MARTIN, Robert. **Designing object oriented C++ applications using the Booch method**. [S.l.]: Prentice-Hall, 1995.

[MAR 98] MARTIN, R.; RIEHLE, D.; BUSCHMANN, F. **Pattern Languages of Program Design.** Reading, MA: Addison-Wesley, 1998. v.3.

[PBR 90] PREMERLANI, W.J. et al. An Object-Oriented Relational Database. **Communications of the ACM**, New York, v.33, n.11, p.99-109, Nov. 1990.

[PRE 95] PREE, Wolfang. **Design Patterns for Object – Oriented Software Development**. Reading, MA: Addison – Wesley, 1995.

[REI 96] REINWALD, B. et al. Storing and using objects in a relational database. **IBM Systems Journal**, [S.l.], v.35, n. 2, p.172-191, 1996.

[ROG 97] ROGERS, Gregory F. **Framework – Based Software Development in C++.** New Jersey: Prentice - Hall, 1997.

[RUM 91] RUMBAUGH, J. et al. **Object-Oriented modeling and design**. [S.l.]: Prentice-Hall, 1991.

[SES 96] SESSIONS, R. **Object persistence beyond object oriented databases**. Englewood Cliffs: Prentice Hall, 1996.

[STO 96] STONEBRAKER, M.; Moore, D. **Object relational DBMS**: the next great wave. [S.l.]: Morgan Kauffman, 1996.

[ZAN 97] ZANCAN, Júlio C. **Um Estudo sobre Requisitos de Ferramentas de Apoio a Instanciação de Frameworks:** Trabalho Individual. Porto Alegre: CPGCC da UFRGS, 1997. (TI -599).

Informática  *UFRGS*

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"Um Framework para Construção de Aplicações OO sobre SGBD Relacional"
por

Kurt Werner Molz

Dissertação apresentada aos Senhores:

Prof. Dr. Cirano Iochpe

Prof. Dr. Clesio Saraiva dos Santos

Prof. Dr. Duncan Dubugras Alcoba Ruiz (PUCRS)

Vista e permitida a impressão.
Porto Alegre, 17 / 12 / 1999.

Prof. Dr. Carlos Alberto Heuser,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora do Programa de Pós-Graduação
em Computação - PPGC
Instituto de Informática - UFRGS