Universidade Federal da Paraíba

Centro de Ciências e Tecnologia
Departamento de Sistemas e Computação
Coordenação de Pós-Graduação em Informática

Dilvan Vitor dos Santos

Transformação de Esquemas de Objetos para um Gerenciador Relacional Estendido, Considerando o Padrão ODMG

Dissertação apresentada ao curso de MESTRADO EM INFORMÁTICA da Universidade Federal da Paraíba, em cumprimento às exigências parciais para obtenção do Grau de Mestre.

Área de Concentração: Banco de Dados

Orientador: Ulrich Schiel

Campina Grande - PB
Setembro de 1998

S237t Santos, Dilvan Vitor dos.
Transformação de esquemas de objetos para um gerenciador relacional estendido, considerando o padrão ODMG / Dilvan Vitor dos Santos. - Campina Grande, 1998.
70 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1998.
"Orientação : Prof. Dr. Ulrich Schiel".
Referências.

1. Banco de Dados. 2. Objetos. 3. Gerenciador Relacional Estendido. 4. Dissertação - Informática. I. Schiel, Ulrich. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.65(043)

TRANSFORMAÇÃO DE ESQUEMAS DE OBJETOS COMPLEXOS PARA UM GERENCIADOR RELACIONAL ESTENDIDO, CONSIDERANDO O PADRÃO ODMG

DILVAN VITOR DOS SANTOS

DISSERTAÇÃO APROVADA EM 12.08.1998

PROF. ULRICH SCHIEL, Dr.
Orientador

PROF. MARCUS COSTA SAMPAIO, Dr.
Examinador

PROF. DÉCIO FONSECA, Dr.
Examinador

CAMPINA GRANDE - PB

Agradecimentos

Ao Prof. Ulrich que mostrou o melhor caminho a ser enveredado para atingir nosso objetivo.

Ao Prof. Rincon, Diretor do Instituto Brasileiro de Informação em Ciência e Tecnologia que especializa o seus funcionários visando a qualidade dos serviços prestados pelo Instituto.

Agradecer a todos os técnicos, profissionais e professores que se interessaram e colaboraram com este trabalho destacando Aninha da UFPb e Zairton do IBICT.

Também ao Marcão com quem pude tomar uns chimarrãos e ao Maurício parceiro de trilha na Serra da Borborema, com os quais pude trocar muitas idéias acerca dos nossos trabalhos.

Gostaria de agradecer também às pessoas que são somente usuárias do computador e que não estão nem um pouco afim de entender as suas complexidades e muito menos a minha tese, mas que me auxiliaram de alguma forma: Valdenir, Norma, Davilene, Tereza Rezende, Júnior e Tatiane, Naná, Paula, Dione, Cristina, aos meus Pais Adelita e Cosmo, Vânio, Sérgio, Vânia Cláudia e Vânia Cristine.

E finalmente ao CNPq que custeiou todo esse trabalho de pesquisa.

Sumário

Lista de Figuras

Resumo

Novas aplicações de banco de dados necessitam de características adicionais para que seja possível o modelamento tanto dos aspectos comportamentais como temporais. Atualmente, novos modelos tem sido proposto para capturar estes aspectos, que muitas vezes não são representados na maioria dos sistemas gerenciadores de banco de dados, mas que suportam eficientemente esses aspectos sem muitas complexidades. Entretanto para implementar esses aplicações usando um SGBD, é necessário prover um mapeamento do modelo de dados temporal para o modelo relacional adotado. Nesta dissertação nos propomos uma abordagem de mapeamento de um modelo dados de objetos temporais chamado **TOM** (*Temporal Object Model*), usando um modelo relacional estendido, considerando o padrão **ODMG** (*Object Database Management Group*).

Com esta abordagem é possível transportar uma aplicação de um ambiente para um outro, que esteja de acordo com as características do ODMG e do SGBD existente. Para ilustrar a abordagem proposta, nos implementamos o mapeador usando SGBD relacional estendido específico, OpenBASE.

Abstract

New database applications require additional features for behavioral and temporal modeling. Recently, new data models have been proposed to capture these aspects although there are yet no database management systems that support efficiently such facilities. Thus, to implement these applications using existing DBMSs, it is necessary to povide a proper mapping from the temporal data model to the underlying model of the adopted system. In this dissertation we propose a mapping approach to the modeling of an object-oriented temporal data model, called **TOM** (**T**emporal **O**bject **M**odel), using relational extended, whose this mapper conside the standard **ODMG** (**O**bject **D**atabase **M**anagement **G**roup).

This approach makes possible to carry out the implementation according to the specific features of the ODMG and the existing DBMS. To illustrate the proposed approach, we discuss the implementation of mapper using the specific relational extended DBMS, OpenBASE.

1. Introdução

1.1 Motivação e Objetivo da Dissertação

A tecnologia de banco de dados tem uma história de mais de 30 anos, iniciada com simples sistemas de arquivos no início dos anos 60. A falta de controle centralizado de dados desses sistemas e outros problemas, teve como conseqüência natural o aparecimento dos sistemas de banco de dados propriamente ditos, tais como os bancos de dados hierárquico e de redes, ainda no início da década de 60.

Esses sistemas ofereciam controle centralizado dos dados e outras propriedades desejadas como controle de dados redundantes e de concorrência. Mas ainda era complicado acessar os dados, pois o usuário era forçado a usar alguns comandos primitivos para navegar em sua estrutura de dados.

O grande salto foi dado por E.F. Codd em 1970 ([Cod70]) propondo o modelo de dados relacional. Neste modelo, conjuntos de entidade e seus relacionamentos são uniformemente representados por tabelas ou relações. Do ponto de vista do usuário desenvolvedor, o grande avanço foi a relativa facilidade com que os dados eram abstraídos e apresentados, isolando esse mesmo usuário da complexidade e representação física dos dados e da forma como eles eram armazenados. Para recuperar esses dados, era possível usar uma linguagem declarativa, para dizer ao sistema o que se estava procurando. Além disso, oferecia a vantagem de ter uma sólida fundamentação teórica. Por estes motivos, o modelo foi adotado pela maioria dos sistemas gerenciadores de banco de dados.

Entretanto, os bancos de dados relacionais carecem de importantes características necessárias às aplicações mais avançadas. Os dados dessas aplicações são muito complexos e evolutivos, sendo necessária a modelagem tanto de sua estrutura quanto de seu comportamento. Para solucionar este problema, começaram a surgir, a partir da segunda metade da década de 70, inúmeras propostas de novos modelos de dados, chamados Modelos Semânticos de Dados ([Pec88]). A esses modelos foram acrescidas características de encapsulamento dos tipos abstratos de dados e herança de tipos, dando origem às linguagens de programação orientadas a objetos. A essas linguagens, a partir da segunda metade da década de 80, foi oferecido suporte a banco de dados, que permitiram que os objetos fossem armazenados, tornando-os persistentes para posteriores recuperações. Duas linguagens que evoluíram nesse sentido foram C++ e

Smalltalk, freqüentemente usadas nos bancos de dados orientados a objetos como linguagens *binding*. Ambas mapeam a sintaxe dos modelos de objetos - persistentes, para a sintaxe das linguagens orientadas a objetos - transientes ([COR98a]).

Com o surgimento dessas novas tecnologias, muitos sistemas foram desenvolvidos tendo como principal objetivo a representação semântica dos objetos. Isso por um lado foi muito positivo, no sentido de que muitas soluções foram apresentadas para os problemas provenientes dos sistemas gerenciadores de banco de dados orientados a objetos, mas por outro lado, a coexistência de muitos sistemas diferentes, apontava para a necessidade de um modelo padrão que permitisse a comunicação e portabilidade das aplicações desses sistemas. Hoje, muitos desenvolvedores de bancos de dados orientados a objetos fundamentam-se no ODMG - *Object Database Management Group*, (Maiores detalhes no Capítulo Modelos Adotados) modelo padrão que apresenta uma arquitetura comum de desenvolvimento de sistemas gerenciadores de banco de dados orientados a objetos ([Cat96]).

Mas o fato é que os SGBD's relacionais ainda representam o estado da arte em termos de sistemas comerciais e apresentam muitos recursos que permitem representar os objetos complexo sem perda semântica. Com o objetivo de preservar todas as características do banco de dados relacionais surge com o papel de *middleware* os banco de dados objeto relacional.

Dessa maneira, nesta dissertação é feita uma abordagem de um modelo semântico de objetos temporal ativo que é conhecido como **TOM** - *Temporal Object Model* ([Sch91], [Dv92], [Fer96]), que utiliza um SGBD relacional estendido para armazenar os objetos, que é o OpenBASE da Tecnocoop Sistemas, o qual nos foi cedido pela mesma, para realizarmos este trabalho.

Mas o objetivo maior do nosso trabalho é estreitar as incompatibilidade existentes entre os modelos semânticos e os modelos relacionais e para tanto, consideramos o modelo ODMG, o qual vem sendo usado como referência por muitos fabricantes de banco de dados. Nesse ponto, e necessário fazer o mapeamento de esquemas de objetos TOM/ODMG para esquemas relacionais OpenBASE. Ilustramos estes aspectos na Figura 1.1 (detalhamos esses processos no Capítulo 3 que trata dos mapeamentos).

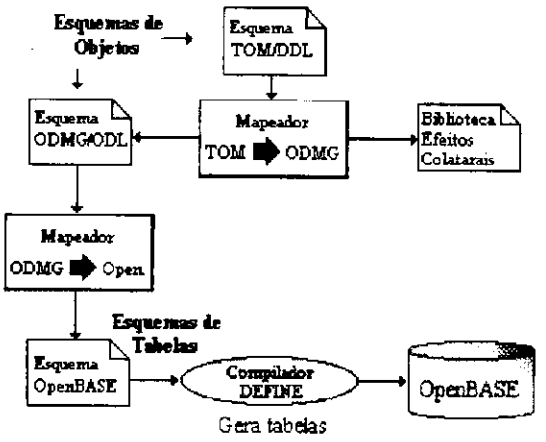


Figura 1.1 - Transformação de objetos TOM em tabelas do OpenBASE considerando o padrão ODMG.

Nesse sentido, estamos diante de um sistema gerenciador de banco de dados objeto-relacional, o que permite que sejam aproveitadas integralmente as funcionalidades desenvolvidas para o ambiente relacional.

A estrutura básica desta dissertação é composta dos seguintes capítulos:

1. **Introdução**
2. **Os Modelos Adotados**
3. **Mapeamentos do Modelos**
4. **Implementação**
5. **Conclusão e Trabalhos Futuros**

Em três apêndices, descrevemos as linguagens dos modelos adotados:

A. Linguagem conceitual do TOM - *Temporal Object Model*

B. Linguagem de Definição de Objetos - ODL - *Object Definition Language* do ODMG

C. Linguagem DEFINE do OpenBASE (Linguagem de definição de esquemas relacionais)

No capítulo **um,** damos introdução a esta dissertação, onde nos situamos fazendo uma abordagem da evolução dos bancos de dados. Está dividido em quatro seções, sendo que a primeira fala da motivação e objetivo do trabalho e a segunda e terceira nos dá uma visão dos bancos de dados relacionais e orientados a objeto, respectivamente, enfatizando as suas principais características. Na seção quatro, abordamos a tecnologia orientada a objeto combinada com a tecnologia relacional.

O capítulo **dois,** está dividido em três seções, nas quais discutimos os modelos adotados, os quais são: o modelo TOM - *Temporal Object Model*; o modelo ODMG - *Object Database Management Group*; e o banco de dados relacional OpenBASE respectivamente.

No capítulo **três**, tratamos dos mapeamentos entre estes modelos, ou seja, TOM para ODMG e ODMG para OpenBASE.

No capítulo **quatro**, especificamos o projeto dos mapeadores implementados em C++.

Finalmente no capítulo **cinco,** é feito um resumo conclusivo, apontando os trabalhos futuros e as perspectivas referentes aos SGBD's.

Além destes capítulos, são também apresentados como parte dessa dissertação os apêndices que tratam das linguagens de definição dos dados dos modelos TOM, ODMG e OpenBASE.

1.2 Banco de Dados Relacional

O modelo de banco de dados relacional de E. Codd, tinha diversas características importantes que resolviam muitos dos problemas provenientes dos sistemas de banco de dados existentes na época da sua proposta. O modelo não se baseava num paradigma de estruturação de dados particular mas, ao contrário, num fundamento matemático específico. O modelo resultante podia expressar uma descrição não redundante dos dados e um conjunto de operadores fixos, com os quais os dados podiam ser formalmente recuperados.

O modelo de dados relacional se popularizou por tornar possível ao usuário modelar ou ter uma concepção do seu sistema ou sua aplicação sem se preocupar com as estruturas dos dados e implementações. Essa foi uma importante meta da tecnologia de banco de dados, pois os dados passaram a ser independentes dos processos.

1.2.1 Estrutura de Dados Relacional

Nos bancos de dados relacionais todos os dados estão armazenados em tabelas bi-dimensionais. Uma aplicação pode geralmente ter uma série de tabelas para conter/manter seus dados.

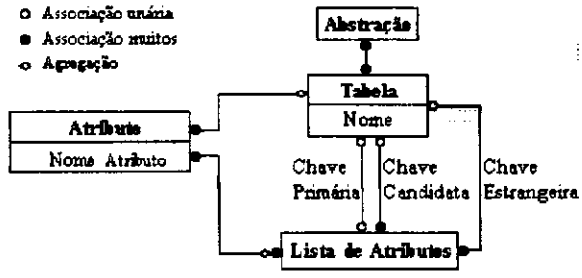


Figura 1.2 - Modelo para descrever a estrutura das tabelas dos SGBDRs - ([Bla94])

A Figura 1.2 apresenta uma base simples para modelar as tabelas e seus relacionamentos. Entre a tabela e a lista de colunas, estão três associações denominadas: chave primária - provê identidade a tabela; chave candidata - uma ou mais colunas para constituir a chave primária da tabela; e chave estrangeira - refere-se a uma chave primária de uma outra tabela, declarada em uma determinada tabela.

O modelo relacional está fundamentado em regras matemáticas bem específicas. Essas regras deram origem à álgebra relacional, que é a base das linguagens de consulta não-procedurais como a **SQL** que foi projetada como uma linguagem de banco de dados para dar acesso a um SGBD baseado no modelo relacional. Atualmente ela passa por adaptações para que também possa manipular objetos. Seu principal objetivo é fornecer

uma linguagem padrão de acesso aos bancos de dados tanto relacional como orientado a objetos ([ISO96]).

1.2.2 Limitações do Modelo Relacional

O modelo relacional, entretanto, apresenta uma desvantagem. É difícil expressar a semântica dos objetos em um modelo de tabelas. Com o objetivo de suprir algumas dessas deficiências do modelo relacional, foram acrescentadas abstrações de generalizações e agregação ao mesmo ([Cod79], [Che76], [Smt77]). Essas extensões podem ser interpretadas como um conjunto de regras semânticas acrescentadas ao modelo relacional básico com o propósito de melhor representar os objetos do mundo real. Na sessão 2.3.2, detalhamos as extensões do Modelo de Entidade e Relacionamento.

1.3 Banco de Dados Orientado a Objetos

As aplicações de engenharia foram as primeiras a utilizar bancos de dados que manipulassem objetos complexos. Aplicações como CAD (*Computer Aided Design*) e CAM (*Computer Aided Manufaturing*) sempre usaram formas não tradicionais de dados, representando com precisão os comportamentos dos objetos em imagens tridimensionais. Essas aplicações armazenam seus dados em arquivos de estruturas específicas, portanto, requerem uma representação dos dados mais rica do que aquela oferecida pelos bancos de dados relacionais.

1.3.1 O Modelo de Objetos

O modelo orientado a objeto é baseado na idéia de representar a semântica e comportamento dos objetos, em estruturas complexas. As propriedades básicas dos objetos são encapsulamento, herança, polimorfismo e identidade.

1.3.1.1 Encapsulamento

O encapsulamento é a propriedade que combina código e dado para formar os objetos e os dados desses objetos, não podem ser acessados ou manipulados, exceto através dos métodos que fazem parte do objeto. Este é um mecanismo de proteção importante em banco de dados, que já está embutido nas linguagens OO.

1.3.1.2 Herança

Herança é um mecanismo que permite que novas classes sejam construídas aproveitando código e dados declarados em classes já existente. Com isso, as características comuns dos objetos não terão mais que ser descritos nos níveis mais baixos. Ou seja, uma subclasse irá herdar a estrutura de dados e os métodos de sua superclasse.

1.3.1.3 Polimorfismo

A fim de atender a uma necessidade particular, cada objeto recebe e responde a uma mensagem comum de maneira apropriada. Desse modo, uma operação pode ser implementada para apresentar resultados diferentes, independentemente do tipo de objeto, considerando que essa operação esta sendo herdada de uma superclasse.

1.3.1.4 Identidade

Os objetos em um sistema orientado a objetos têm seus próprios identificadores. Essa identidade não depende dos valores que estão contidos nos objetos, mas sim do

endereço de memória onde o objeto está localizado. Este endereço é utilizado para estabelecer o relacionamento entre objetos.

1.4 Combinando os Modelos Relacional e Objeto

Os bancos de dados orientados a objetos já estão no mercado há vários anos e fabricantes de banco de dados relacionais estão estendendo seus produtos com capacidades orientadas a objetos. Alguns dos bancos de dados disponíveis como Oracle8, UniSQL, ObjecStore, Informix e outros, já estão combinando capacidades relacionais e orientadas a objeto. Mostraremos, a seguir, a estratégia de unificação da tecnologia orientada a objetos com a tecnologia de banco de dados relacional.

1.4.1 Unificando os Banco de Dados Relacionais e Orientado a Objetos

Os bancos de dados orientados a objetos foram desenvolvidos para armazenar e gerenciar objetos gerados pelas linguagem orientada a objetos. Um banco de dados relacional pode ser usado para armazenar e gerenciar tais objetos. Contudo, um banco de dados relacional não compreende objetos, em particular, métodos e herança. Por essa razão, necessita de uma camada intermediária, que entendemos como um nível orientado a objetos para gerenciar os objetos. Nesse nível os objetos são mapeados para tabelas do relacional, produzindo dessa forma, um banco de dados relacional com características orientada a objetos ([Kim88], [Cat94], [Ban96]).

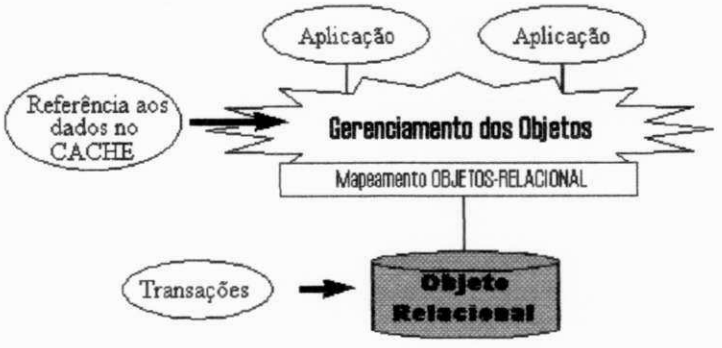


Figura 1.3 - Integração objeto-relacional

A Figura 1.3, mostra uma arquitetura objeto-relacional. Dessa forma, todas as funcionalidades dos bancos de dados relacionais são mantidas.

Porém, o mapeamento de objetos para tabelas compromete a performance do banco devido ao processo de conversão dos objetos em tabelas e vice-versa. Outro fator comprometido é a integridade dos dados, pois os SGBDRs não entendem os métodos que mantém muito da semântica e a integridade dos dados. O mapeador pode forçar essas restrições de duas maneiras: i) gerando um código correspondente utilizando a linguagem do banco, fazendo a restrição diretamente no relacional, ou ii) recompondo os objetos no gerenciador de objetos e utilizar os próprios métodos implementados pela LPOO usada.

1.4.2 Estratégias Básicas para BDOOs

Algumas estratégias que podem ser usadas para combinar banco de dados com orientação a objetos.

1.4.2.1 Estendendo uma linguagem orientada a objeto

Nesse contexto se faz necessário desenvolver as funcionalidades principais de banco de dados (persistência, autorização, concorrência, recuperação) nas linguagens orientadas a objetos ([Atk95]). Com uma LPOO - **L**inguagem de **P**rogramação **O**rientada a **O**bjetos Estendida, pretende-se uma navegação eficiente entre os objetos. Essas linguagens são conhecidas também como *binding* ([Cat94]). Por exemplo Poet da Poet Software.

1.4.2.2 Estendendo um banco de dados relacional

Estender o modelo relacional com novos tipo de dados e métodos de acessos aos objetos, definitivamente acrescenta novas funcionalidades ao modelo relacional ([Row90]). Este tipo de integração implica em algumas mudanças e implementações no núcleo do banco como as extensões dos tipos. O objetivo principal nessa estratégia é não abandonar todo investimento feito no banco de dados relacional. Os desenvolvedores fabricantes defendem também, que o mercado para bancos de dados orientado a objeto puro por enquanto não estão dando lucro. Com esse pensamento a Oracle lançou Oracle8 que dá suporte a objetos.

1.4.2.3 Banco de Dados Orientado a Objeto Puro

Essa abordagem defende que para objetos complexos, os bancos de dados orientados a objetos são mais apropriados, pois repensam os bancos de dados e produzem uma nova arquitetura otimizada para atender às necessidades da tecnologia orientada a objetos ([Mai90]). Por exemplo o O_2 da O_2 System.

2. Modelos Adotados

Nesse capítulo, apresentamos os modelos utilizados nessa dissertação que são os seguintes: **TOM** - *Temporal Object Model* - um modelo semântico de modelamento orientado a objetos; o modelo padrão **ODMG** - *Object Database Manager Group*, que nós dá a completa noção de extensibilidade de uma linguagem de programação orientada a objetos; o modelo relacional estendido **OpenBASE** - da Tecnocoop Sistemas. Nas seções que tratam dos mapeamentos entre estes modelos detalharemos com maior profundidade os mesmos.

Consideramos os fatores técnicos para a escolha dos modelos. O TOM por exemplo apresenta uma técnica de modelamento muito rica e além disso, tem servido como base na Universidade Federal da Paraíba, para muitos trabalho de dissertação. Atualmente está sendo implementado uma ferramenta de modelagem de objetos, como um produto de dissertação a ser defendida.

O modelo ODMG, por apresentar uma proposta de padronização que possibilita que os vários clientes (usuário desenvolvedor) possa escrever suas aplicações através de uma interface comum de modelamento dos objetos.

O OpenBASE apresenta as funcionalidades básicas que um SGBD deve ter e a possibilidade de representar os objetos complexos em tabelas relacionais. Um outro banco poderia ser utilizado e pode, para tanto, basta que as palavras reservadas (dicionário de palavras reservadas) do banco em questão, sejam incorporadas ao sistema de mapeamento para construção do esquema. O OpenBASE também serviu como base para alguns trabalhos de pesquisa desenvolvidos na UFPb, como o Banco de Pesquisas do SEI-BIB.

2.1 TOM

TOM - *Temporal Object Model* é um modelo de dados orientado a objeto com características temporais e ativas, baseado nos conceitos de classe, relacionamento, abstração hierárquicas, método e regra ([Sch91], [Dav92], [Fer96]).

2.1.1 Classe

Cada classe de objetos declarada no modelo apresenta uma estrutura de relacionamentos e métodos que podem ser herdados de outras classes. As classes no modelo são classificadas como: **primitivas**, que correspondem aos objetos cujas instâncias são identificadas por tipos de dados simples (inteiro, *string*, real, *boolean* etc);

não primitivas ou **estruturadas**, que correspondem aos tipos abstratos nos modelos semânticos cujas instâncias são identificadas por um ou mais relacionamentos. A classe primitiva tem sua representação gráfica como uma elipse delineada com um risco simples, a classe não primitiva ou estruturada é delineada com um risco mais espesso e a classe temporal tem a parte superior direita da elipse sombreada (Figura 2.1).

Figura 2.1- Representação gráfica das classes.

2.1.2 Relacionamento

Relacionamentos são associações válidas entre dois objetos. A cada relacionamento é associado uma cardinalidade, expressa pelo par de valores (`min`, `max`), que determina o mínimo e o máximo de instâncias associadas pelo relacionamento. Consideramos os seguintes tipos de relacionamentos:

- **Relacionamento Instância**: É aquele que relaciona instâncias de uma classe com instâncias de outra classe. Na Figura 2.2 expressa que um Modelo de carro pode ser de nenhum carro ou vários carros e carro deve ser de um e somente um modelo.

Figura 2.2 - Relacionamento instância

- **Relacionamento Dinâmico**: É a associação entre objetos através de troca de mensagem.

2.1.3 Abstrações Hierárquicas

As classes são organizadas numa estrutura hierárquica. Os relacionamentos hierárquicos entre classes podem ser considerados abstrações, no sentido de que detalhes de objetos nas classes inferiores são desconsiderados nas classes mais gerais. Existem as seguintes formas de abstração: generalização, agregação e agrupamento.

2.1.3.1 Generalização

Uma nova classe de objeto (superclasse) pode descrever propriedade comuns de várias classes mais específicas. É representada pelo predicado **é_um** (A, B), significando que A é uma subclasse de B (Figura 2.3).

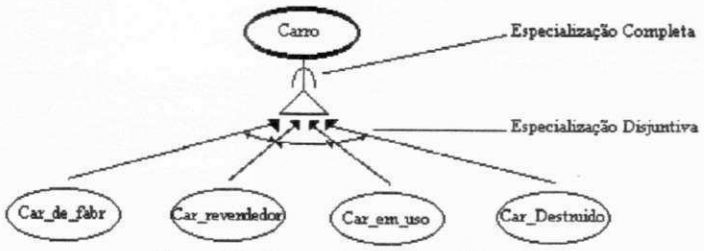


Figura 2.3 - Generalização/Especialização

2.1.3.2 Agregação

Uma agregação é o ato de formar uma nova classe a partir de classes **componentes**. É representado pelo predicado **é_parte_de** (A, B), onde A é um componente da classe agregada B. Figura 2.4

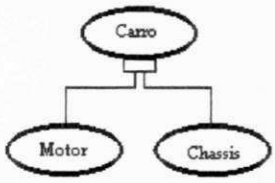


Figura 2.4 - Agregação

2.1.3.3 Agrupamento

O agrupamento é utilizado quando um conjunto de objetos do mesmo tipo, formam um novo objeto. É representado pelo predicado **é_elemento** (A, B), onde as instâncias de B são conjuntos de instâncias de A. B é chamado de classe grupada e A é a classe elemento (Figura 2.5).

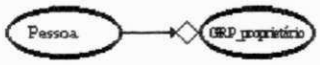


Figura 2.5 - Agrupamento

As abstrações podem ser aplicadas recursivamente na construção de objetos complexos.

2.1.4 Aspectos Temporais

O modelo TOM permite que os aspectos temporais das aplicações sejam modelados através de classes e relacionamentos temporais. Em uma classe temporal, a cada instância está associado o tempo de permanência de uma determinada instância do objeto naquela classe. Este tempo é dado por um (ou vários) intervalos temporais. Para objetos atuais, o limite superior deste intervalo deve ser uma data no futuro ou é uma variável especial, representando o presente.

2.1.5 Método

Um método implementa o comportamento dos objetos. Os métodos de um tipo de objeto referenciam somente as estruturas de dados desse tipo de objeto. Eles não devem acessar diretamente as estruturas de dados de outro objeto. Para acessar a estrutura de dados de outro objeto, eles devem enviar uma mensagem a esse objeto. O objeto que receber a mensagem reagirá a mensagem recebida, realizando dessa maneira um relacionamento dinâmico.

2.1.6 Aspectos Dinâmicos do TOM

Os aspectos dinâmicos do modelo TOM são modelados no TOM-Rules ([And93]), um subsistema automático de regras que representa a extensão do modelo com características ativas. As regras do sistema TOM-Rules baseiam-se no conceito de regras ECA - Evento_Condição_Ação ([Day88]). Essas regras são utilizadas para gerenciar as regras ativas, que especificam ações a serem executadas sempre que certos eventos ocorrem e uma condição é satisfeita. O sistema de regras ativas baseiam-se nos conceitos de evento, trigger e regra.

2.1.6.1 Evento

Evento é uma mensagem que, ao ser detectado por uma regra, pode vir a mudar o estado de um objeto. São considerados três tipos de eventos ([Rol88]):

- **Evento Externo** - é aquele que ocorre com a chegada de uma mensagem do mundo real.
- **Evento Temporal** - é aquele que representa um dado instante do tempo.
- **Evento Interno** - Ocorrem quando determinados métodos são executados através da troca de mensagens entre os objetos do banco, ou ocorrem quando uma certa condição sobre o banco de dados se torna verdadeira.

2.1.6.2 Trigger

Trigger é o mecanismo de execução de uma ou mais ações. Cada *trigger* possui uma condição que faz com que uma determinada ação seja disparada.

2.1.6.3 Regra

Uma regra estabelece em quais situações (eventos) e quais condições certas ações devem ser executadas.

2.2 ODMG

O modelo **ODMG** - *Object Database Management Group* ([Cat96]) foi concebido pelos fabricantes de SGBDs como o modelo padrão de desenvolvimento de bancos de dados orientados a objetos, com o propósito de promover a portabilidade das aplicações desenvolvidas nos vários sistemas de acordo com este padrão. O modelo ODMG baseia-se o **OMG** - *Object Management Group* que especifica o padrão de modelamento e implementação dos objetos em uma arquitetura distribuída cliente/servidor - **CORBA** - *Common Object Request Broker Architecture* - ([COR98]).

2.2.1 Arquitetura

A principal idéia do modelo ODMG está centrada na abordagem de linguagem de programação orientada a objetos com funcionalidade de banco de dados. Dessa forma, as linguagens orientadas a objetos são estendidas para prover dados persistentes, controle de concorrência, recuperação dos dados, consultas e outras capacidades de banco de dados.

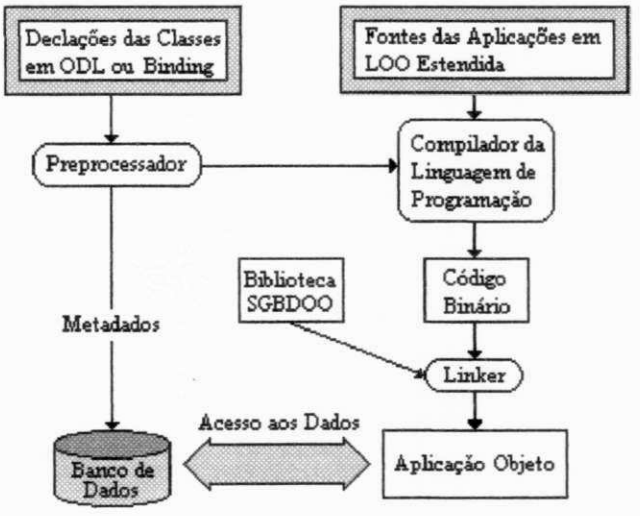


Figura 2.6 - Arquitetura ODMG.

Uma arquitetura (Figura 2.6) foi proposta, onde as declarações dos objetos são feitas usando uma sintaxe de linguagem comum, que é a **ODL** (*Object Definition Language*) ou **IDL** (*Interface Definition Language*) ([COR98]) com os métodos de manipulação desses objetos no banco, implementados usando uma linguagem estendida (Ex. *binding* C++). As declarações dos objetos e os fontes dos programas, são então compilados e *linkados* à biblioteca de métodos do banco de dados ODMG, para gerar a aplicação.

2.2.2 Os Principais Componentes ODMG

2.2.2.1 O Modelo de Objetos

Os elementos básicos do modelo ODMG podem ser descritos da seguinte maneira:

- O modelo ODMG é baseado em objetos, com identificadores, hierarquia e propriedades de objetos;
- O comportamento dos objetos é definido pelo conjunto de métodos que podem ser executados no objeto;
- O estado dos objetos é definido pelos valores das propriedades desses objetos. Essas propriedades são os atributos e os relacionamentos entre um ou mais objetos;
- Atributos são declarados definindo-se o nome e o tipo de seus valores permitidos;
- Relacionamentos são binários e são definidos entre dois tipos de objeto. Nesse contexto, uma função (atributo inverso) é usada para referir-se ao objeto relacionado da classe destino. A cardinalidade do relacionamento pode ser um-para-um, um-para-muitos e muitos-para-muitos;
- O modelo inclui herança de objetos baseado nos relacionamentos dos tipos e subtipos e pode ser entendido como generalização/especialização.

Os objetos podem ser compostos, conforme a hierarquia de tipos da Figura 2.7.

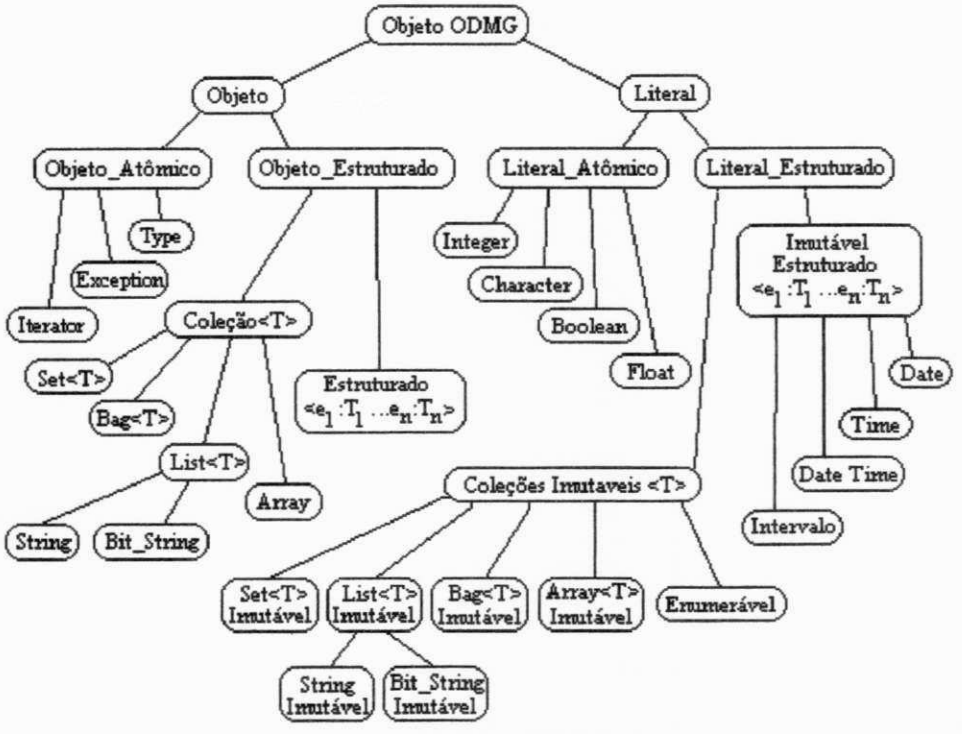


Figura 2.7 - Hierarquia de tipos ODMG

Conforme a hierarquia de tipos, existem no modelo os tipos mutáveis e imutáveis, ou seja, objetos são mutável e literais são imutáveis. A representação da identidade de

um `literal` é tipicamente o padrão de *bits* que codifica seu valor. A identidade de um `objeto` é o seu OID cujo valor está baseado no endereço de memória. No modelo ODMG, os atributos são do tipo `literal` e os relacionamentos são do tipo.

Essas denominações literais e objetos, correspondem respectivamente às classes primitiva e não-primitiva no modelo TOM.

2.2.2.2 Linguagem de Definição de Objetos - *Object Definition Language*-ODL

Chamada de ODL para distinguir das DDL dos bancos de dados tradicionais, a linguagem de definição de objetos, tem uma sintaxe para descrever esquemas de bancos de dados que permite definir tipos de objetos que possam ser implementados em uma variedade de linguagens de programação (*binding*). Veja a sintaxe da Linguagem ODL no Apêndice 2 - ODMG. As principais características da ODL são:

- Permitir que um mesmo banco de dados possa ser compartilhado por várias linguagens de programação - (*Bindings*: C++, Smalltalk e Java);
- O esquema especificado em ODL pode ser usado nos SGBDOOs fundamentados no ODMG.

Um exemplo de esquema definido em ODL encontra-se na seção 3.1.

Ilustraremos graficamente as principais propriedades do modelo. As classes de objetos são representadas como uma elipse (Figura 2.8).

Figura 2.8 - Classe ODMG

Relacionamentos das classes são mostrados como linhas. A cardinalidade permitida pelos tipos de relacionamentos são indicados pelas setas nas extremidades das linhas (Figura 2.9).

Figura 2.9 - Relacionamentos ODMG

Os subtipos ou subclasses são representados com uma grande seta apontando para o tipo genérico ou classe genérica. Na Figura 2.10 representa as subclasses da classe carro.

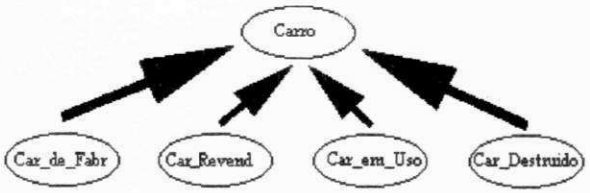


Figura 2.10- Generalização ODMG

2.2.2.3 Linguagem de Consulta a Objetos - *Object Query Language* - OQL

As informações podem ser consultadas e atualizadas no banco de dados tanto através de linguagem de manipulação de objetos (*binding*), ou através de linguagem de consulta de objetos, identificando os objetos através de predicados definidos em suas propriedades.

Em uma consulta simples, um predicado se aplica a uma coleção de objetos, para selecionar os objetos que satisfaçam esse predicado. Entretanto, consultas mais complexas podem ser elaboradas, equivalente aos *joins* do modelo relacional.

As características básicas da OQL são:

- consulta declarativa com primitivas de alto nível para manipular coleções de objetos;
- sintaxe baseada em SQL;
- outras sintaxes OQL podem ser definidas para combinar a linguagem de consulta com uma linguagem de programação possibilitando que OQL fique embutido na linguagem de programação.

2.2.2.4 Linguagens Binding - *Object Manipulation Language* - OML

São previstas várias **L**inguagens de **M**anipulação de **O**bjetos (**OML**), para escrever códigos portáveis para manipular objetos persistentes ou transientes, inclusive as linguagens C++, Smalltalk e Java. Essas *bindings* são uma extensão da sintaxe e semântica das linguagens originais (LPOO estendida).

2.2.3 Outros Componentes

Esses componentes são OMG e CORBA e são importantes para SGBDOOs, mas não estão diretamente associadas aos bancos de dados e sim com os vários aspectos dos objetos.

OMG - *Object Management Group* ([OMG93]) é uma organização internacional de normalização gerida por vários membros, incluindo empresas desenvolvedoras de *software* (por exemplo: 0_2 System e Poet Software e etc.). Fundado em 1989, OMG promove a teoria e prática da tecnologia orientada a objetos em desenvolvimento de software. O Grupo estabelece especificações para prover uma arquitetura comum de desenvolvimento

de aplicações orientadas a objetos. Os objetivos principais são a reusabilidade, portabilidade e interoperabilidade dos software baseados em objetos em ambientes heterogêneos. ([Cat94]). O OMG serve como uma base para o modelo ODMG.

CORBA - *Common Object Request Broker Architecture* - ([COR98]), está de acordo com especificações OMG (Figura 2.11). A estrutura CORBA está baseada em um modelo de referência que tem como componente um *Object Request Broker* - **ORB**, que controla e estabelece conexões entre objetos, recebendo e respondendo chamadas em um ambiente distribuído. Uma coleção de serviços (interface e objetos) que suportam as funções básicas implementáveis em objetos estão disponíveis nos vários servidores que compõem a rede. Esses serviços são necessariamente para construir alguma aplicação distribuída independente do domínio da aplicação. Por exemplo, o Serviço de Ciclo de Vida define convenções para criar, deletar, copiar e mover objetos, e presta esses serviços, independentemente de como os objetos são implementados.

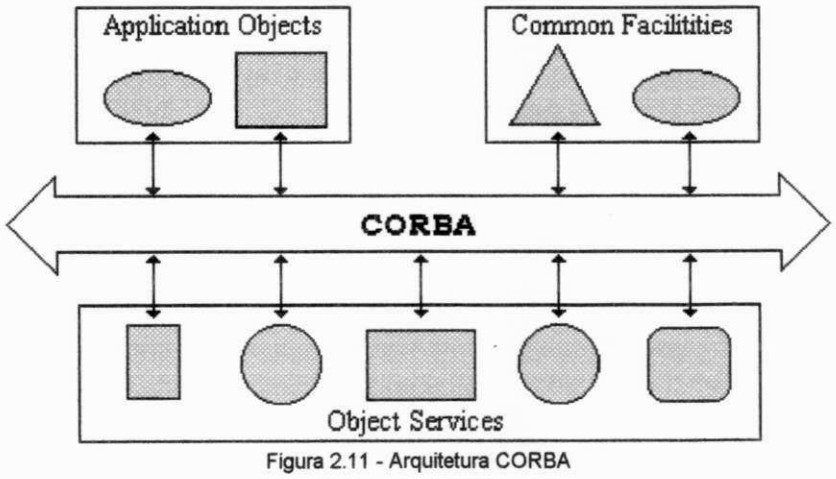


Figura 2.11 - Arquitetura CORBA

Com CORBA, os objetos por serem identificáveis (referência de objeto), podem ser requisitados por uma máquina cliente. Sempre que um cliente solicitar um serviço, o **ORB** é ativado localizando e respondendo a solicitação. Essa requisição pode ser entendida como um evento. Às requisições estão associadas a parâmetros como tipo de operação e localização do objeto.

2.3 OpenBASE

O OpenBASE é um sistema gerenciador de banco de dados relacional extendido, integrado a um completo ambiente de desenvolvimento, manutenção e documentação de Sistemas. ([TEC92]).

As principais extensões oferecidas são:

- **Categorização das Tabelas** (Entidades e Relacionamentos)
  Adota como interface de implementação física o Modelo de Entidades e Relacionamentos - **MER**, na sua forma original, além de permitir, a nível documentacional que sejam utilizadas algumas extensões deste modelo;
- **Integridade Referencial**
  Oferece de forma automática, o controle da integridade referencial entre as associações de dados;
- **Recursos do Modelo Semântico**
  Administra a definição de atributos unívocos, em relação ao conjunto de atributos de um banco de dados, de forma a permitir a recuperação de um ou mais dados, sem que haja a necessidade de qualificar as tabelas onde os mesmos estão armazenados.

2.3.1 Componentes Principais do OpenBASE

O OpenBASE consiste dos seguintes componentes:

- **Sistema de Definição de Banco de Dados** - Denominado DEFINE, processa a descrição de um banco de dados (ESQUEMA) gerada através da sua Linguagem de Definição de Banco de Dados. Linguagem baseada no modelo de Entidade e Relacionamentos, disponibilizando ao desenvolvedor da aplicação um nível de detalhamento do banco de dados bastante amplo. Através do DEFINE o banco de dados é criado ou modificado.
- **Sistema de Rotinas de Manipulação de Banco de Dados** - São as rotinas que compõem o núcleo do OpenBASE. Os métodos de acesso ao banco e gerida por este sistema. Todos os programas que constituem as formas possíveis de acessar um Banco de Dados, utilizam este sistema.
- **Sistema de Utilitários de Banco de Dados** - É um conjunto de programas voltados à administração de banco de dados.
- **Sistema de Consulta e Atualização Interativo** - É um programa chamado GERAL, o qual oferece a seus usuários uma interface interativa, para consultas e atualizações em um banco de dados;
- **Linguagem Estruturada de Consulta (SQL)** - Baseia-se num ambiente chamado TSQL, o qual implementa todos os recursos disponíveis da Linguagem SQL sob o OpenBASE. O TSQL é totalmente compatível com o padrão SQL;
- **Linguagem Compilada** - É uma linguagem chamada OPUS que possibilita ao usuário programar e compilar integralmente suas aplicações.

2.3.2 Modelo Entidade Relacionamento do OpenBASE

O Modelo de Entidade e Relacionamento é uma extensão do modelo relacional. A principal diferença entre os dois modelos é que o MER faz distinções entre entidades e relacionamentos; um relacionamento é considerado um tipo especial de entidade. O OpenBASE adota o MER cujas extensões são:

2.3.2.1 Entidade Autônoma

É a entidade que possui autonomia, ou seja, sua existência não está condicionada à existência de uma outra. Suas características são estritamente particulares conforme Figura 2.12.

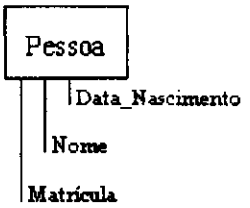


Figura 2.12 - Entidade Autônoma

2.3.2.2 Entidade Fraca

É a entidade que não possui autonomia. Sua existência está condicionada à existência de uma outra no qual mantém relacionamento (1:N) exclusivo com a entidade.

Desta forma, obrigatoriamente, as Entidades Fracas são removidas quando a sua Entidade Autônoma é removida (Figura 2.13).

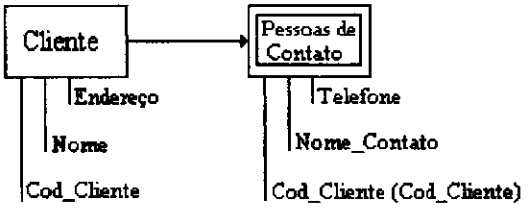


Figura 2.13 - Entidade Fraca.

2.3.2.3 Entidade Associativa (Agrupamento)

São as Entidade que estão subordinadas hierarquicamente a outras. Se uma entidade para existir tem que se relacionar com uma outra, isto demostra dependência hierárquica. Neste caso, esta restrição não impede que a entidade associativa, venha se relacionar a outras entidades além da entidade da qual depende.

A diferença entre o relacionamento da entidade associativa com a entidade principal e os possíveis relacionamento com outras entidades, baseia-se na restrição de que a entidade associativa se relaciona com a entidade principal de forma mandatária (Figura 2.14).

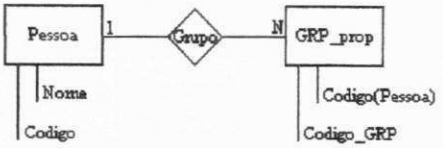


Figura 2.14 - Entidade Associativa.

2.3.2.4 Entidade Composta (Generalização)

É uma entidade composta de sub-entidades.

Uma sub-entidade é parte de um todo que é representado pela entidade, sendo assim, as partes dependem do todo e o todo depende das partes.

Devido ao fato de uma sub-entidade fazer parte de um todo, o relacionamento existente entre ela e o todo será absorvido pela sub-entidade (Figura 2.15).

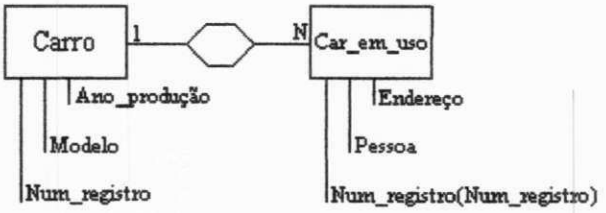


Figura 2.15 - Entidade Composta.

2.3.2.5 Entidade de Ligação ou Agregação

Entidade de Ligação é um Relacionamento que assume o papel de entidade, permitindo ser reconhecida como tal, toda vez que houver necessidade de associar um relacionamento a outro ou possuir identificador próprio.

Quando um relacionamento assume o papel de entidade e não possui um identificador próprio, é necessário eleger um atributo determinante. Normalmente a concatenação dos atributos associativos resolve este caso (Figura 2.16).

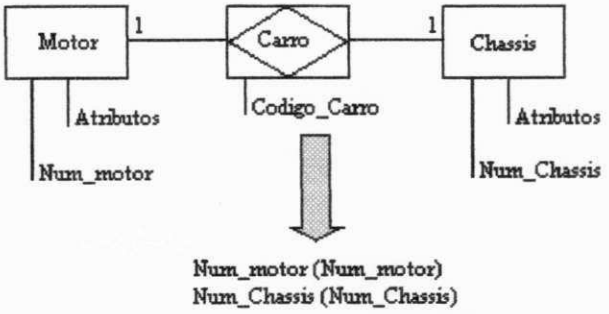


Figura 2.16 - Entidade de Ligação ou Agregação.

3. Mapeamentos

Considerando que uma aplicação de banco de dados é uma representação de vários objetos de nosso mundo e que o comportamento desses objetos vai depender dos objetivos da aplicação, então a busca de uma representação comum desses objetos significaria para o modelista o compartilhamento de conceitos daqueles objetos ([Mar95]). Entendendo dessa maneira e aceitando o modelo ODMG como um modelo padrão de objetos e considerando que muitos modelos tem suas particularidades, o que dificulta a comunicação entre esses objetos, implementamos um mapeador que traduz esquemas de outros modelos OO para esquemas ODMG. Nesse sentido, optamos pelo modelo TOM. Vale ressaltar que o mapeamento engloba somente os aspectos estáticos do modelo TOM.

Visando tornar persistentes os objetos declarados nos esquemas ODMG, este capítulo apresenta um segundo mapeador que traduz os objetos para tabelas do banco de dados relacional, mais especificamente o OpenBASE. Para tornar a aplicação do trabalho mais genérica, o mapeamento foi dividido em dois passos: i) mapeamento de um modelo específico TOM para o modelo padrão ODMG; ii) mapeamento do ODMG para um esquema relacional estendido OpenBASE.

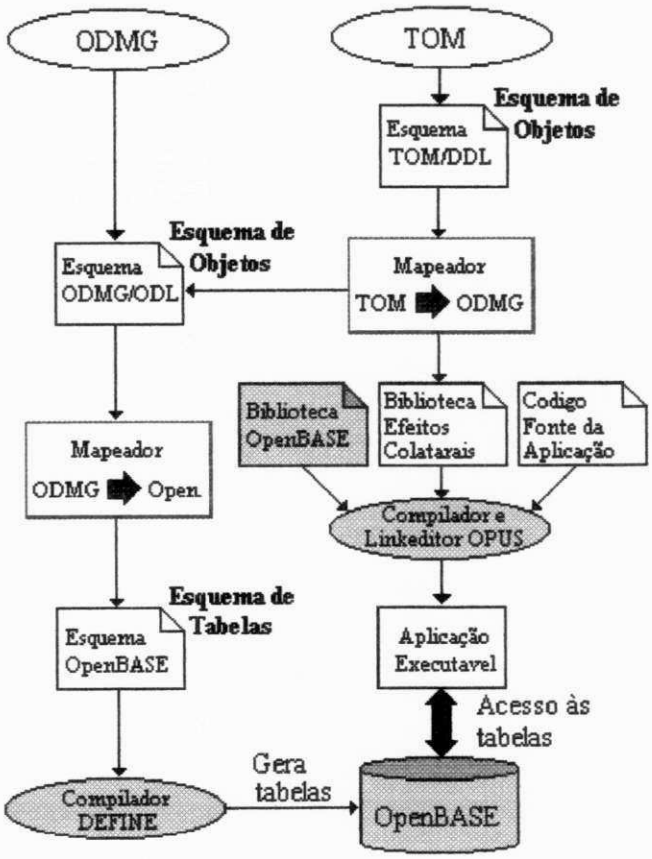


Figura 3.1 - Processo de mapeamento

A Figura 3.1 esclarece como os esquemas ODMG e TOM são processados. Os mapeadores são dois: i) Mapeador TOM ⇨ ODMG, faz a transformação do esquema TOM para o esquema ODMG e ii) Mapeador ODMG ⇨ OpenBASE faz a transformação do esquema ODMG para o esquema OpenBASE. O Mapeador TOM ⇨ ODMG além da transformação, gera os procedimentos na linguagem do banco de dados, que controlam o tempo e a cardinalidade das relações entre os objetos. Estes procedimentos posteriormente são compilados e armazenados em uma biblioteca pelo compilador dos programas fontes do OpenBASE OPUS. Estes procedimentos poderiam ser gerados para uma outra linguagem tais como C, C++ e etc. O esquema OpenBASE gerado é compilado pelo DEFINE, compilador de esquemas do OpenBASE. Dessa maneira, um esquema ODL - *Object Database Language* - qualquer pode ser transformado em um esquema OpenBASE.

Os esquemas lidos são arquivos do tipo ASCII e um *parse* é feito para identificar as palavras reservadas de cada modelo (TOM e ODMG), para tanto um dicionário de palavras reservadas de ambos os modelos são definidos.

Para melhor elucidar os exemplos, utilizaremos um modelo reduzido de uma aplicação de **Registro de Carros**, apresentando inicialmente a representação gráfica na Figura 3.2. Consideramos esta aplicação suficiente (qualquer outra aplicação modelada no TOM poderia ser usada) para expressar as semânticas dos objetos. Enunciamos o seguinte para o domínio da aplicação e está dividida em duas partes:

i) **primeira parte** refere-se ao controle dos carros fabricados pelos seus respectivos fabricantes, que atribui uma identificação única para cada carro. Essa identificação é a composição dos identificadores das partes componentes, como chassis e motor. O motor também tem a sua identificação baseada na suas partes componentes. O carro após ser agregado com suas partes e associado a um modelo de um fabricante. Então as classes a serem criadas na primeira parte são: Carro, Fabricante, Modelo, Chassis, Motor, Carburador e Radiador.

ii) **segunda parte** refere-se ao destino dos carros que poderá estar ainda na fábrica, ou na revendedora, ou com um proprietário ou destruído. Um carro poderá estar apenas em uma dessas situações. Em todas essas situações são estabelecidos tempos, ou seja, para todos os carro em uma dessas situações atribui-se um tempo de permanência nessa situação. Então as classes a serem criadas na segunda parte são: Carro_de_Fabricante, Carro_de_Revendendor, Carro_em_Uso, Car_Destruído, Revendedor, Pessoa e Grupo_Proprietário.

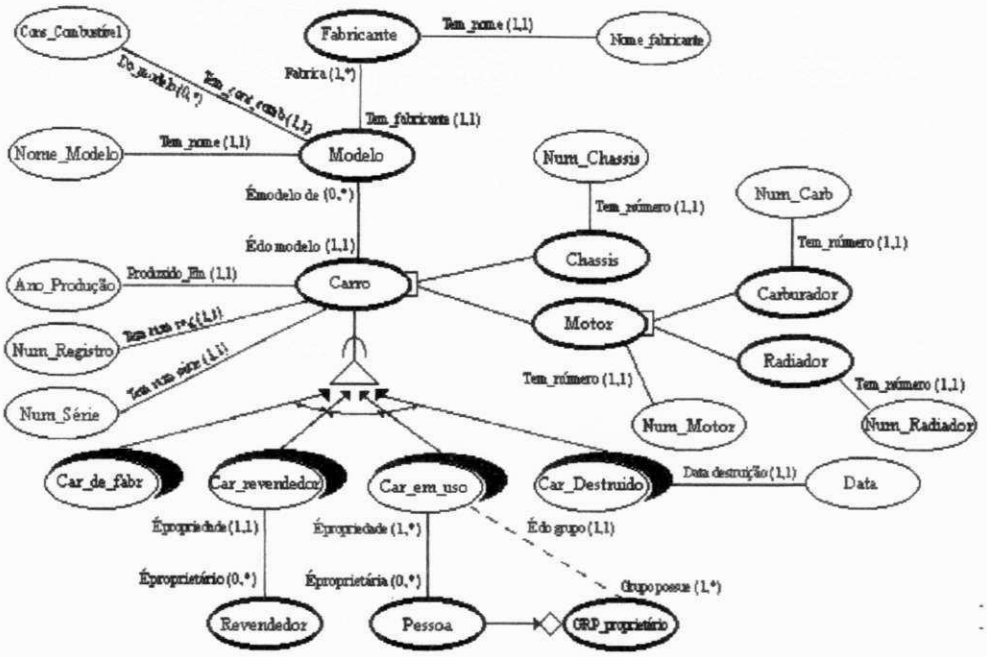


Figura 3.2 - Representação Gráfica do TOM para aplicação **Registro de Carros**.

A descrição dessa aplicação usando a linguagem conceitual do modelo TOM é a seguinte:

```
class_TOM REG_CAR

class CARRO
       instance_relationships
              tem_num_reg     : NUM_REGISTRO (1,1)
              e_do_modelo     : MODELO (1,1)
              produzido_em    : ANO_PRODUCAO (1,1)
              tem_num_ser     : NUM_SERIE (1,1)
       instance_methods
              insere_carro(),
              remove_carro()
       keys_are tem_num_reg
       generalization_of      CAR_DE_FABR, CAR_REVENDEDOR,
                              CAR_EM_USO,  CAR_DESTRUIDO
       aggregation_of MOTOR, CHASSIS

class FABRICANTE
       instance_relationships
              fabricante_nome: NOME (1,1)
              produz          : MODELO (1,*)
       instance_methods
              insere_fabricante(),
              remove_fabricante()
       keys_are fabricante_nome

class CAR_DE_FABR
       class_relationships
              post_class      : CAR_REVENDEDOR exclusive
       keys_are inherited
       parameters_with_time and life_time 3 year

class CAR_REVENDEDOR
       class_relationships
              pre_class       : CAR_DE_FABR
              pre_class       : CAR_EM_USO
              post_class      : CAR_EM_USO exclusive
       instance_relationships
              e_da_empresa    : REVENDEDOR (1,1)
       keys_are inherited
       parameters_with_time and life_time 2 year
```

```
class REVENDEDOR
       instance_relationships
              revendedor_nome : NOME (1,1)
              emp_proprietaria: CAR_REVENDEDOR (0,*)
       keys_are revendedor_nome

class CAR_EM_USO
       class_relationships
              pre_class       : CAR_REVENDEDOR exclusive
              post_class      : CAR_DESTRUIDO
              post_class      : CAR_REVENDEDOR
       instance_relationships
              car_uso_propried: PESSOA (1,10) with 2 old values
       keys_are inherited
       parameters_with_time and life_time 3 year

class CAR_DESTRUIDO
       class_relationships
              pre_class       : CAR_EM_USO exclusive
       instance_relationships
              data_destruicao: DATA (1,1)
       keys_are inherited
       parameters_with_time and life_time 3 year

class PESSOA
       instance_relationships
              pessoa_nome     : NOME (1,1)
              pes_proprietaria: CAR_EM_USO (1,*)
       keys_are pessoa_nome

class GRUPO_PROPRIETARIO
       instance_relationships
              grupo_nome      : NOME (1,1)
       keys_are grupo_nome
       grouping_of PESSOA using pes_proprietaria

class MODELO
       instance_relationships
              modelo_nome     : NOME (1,1)
              tem_cons_comb   : CONS_COMBUSTIVEL (1,1)
              e_modelo_de     : CARRO (0,*)
              produzido_por   : FABRICANTE (1,1)
       keys_are modelo_nome

class MOTOR
       instance_relationships
              tem_num_motor   : NUM_MOTOR (1,1)
              tem_cilindrada  : CILINDRADA (1,1)
       keys_are tem_num_motor
       aggregation_of CARBURADOR, RADIADOR

class CHASSIS
       instance_relationships
              tem_num_chassis  : NUM_CHASSIS (1,1)
       quantidade_porta : QUANT_PORTA (1,1)
       keys_are tem_num_chassis

class CARBURADOR
       instance_relationships
              tem_num_carb    : NUM_CARBURADOR (1,1)
              tem_marca_carb  : NOME (1,1)
       keys_are tem_num_carb

class RADIADOR
       instance_relationships
              tem_num_rad     : NUM_RADIADOR (1,1)
              tem_marca_rad   : NOME (1,1)
       keys_are tem_num_rad

//CLASSES PRIMITIVAS
class NOME
       type string

class CONS_COMBUSTIVEL
       type integer

class NUM_REGISTRO
       type integer
```

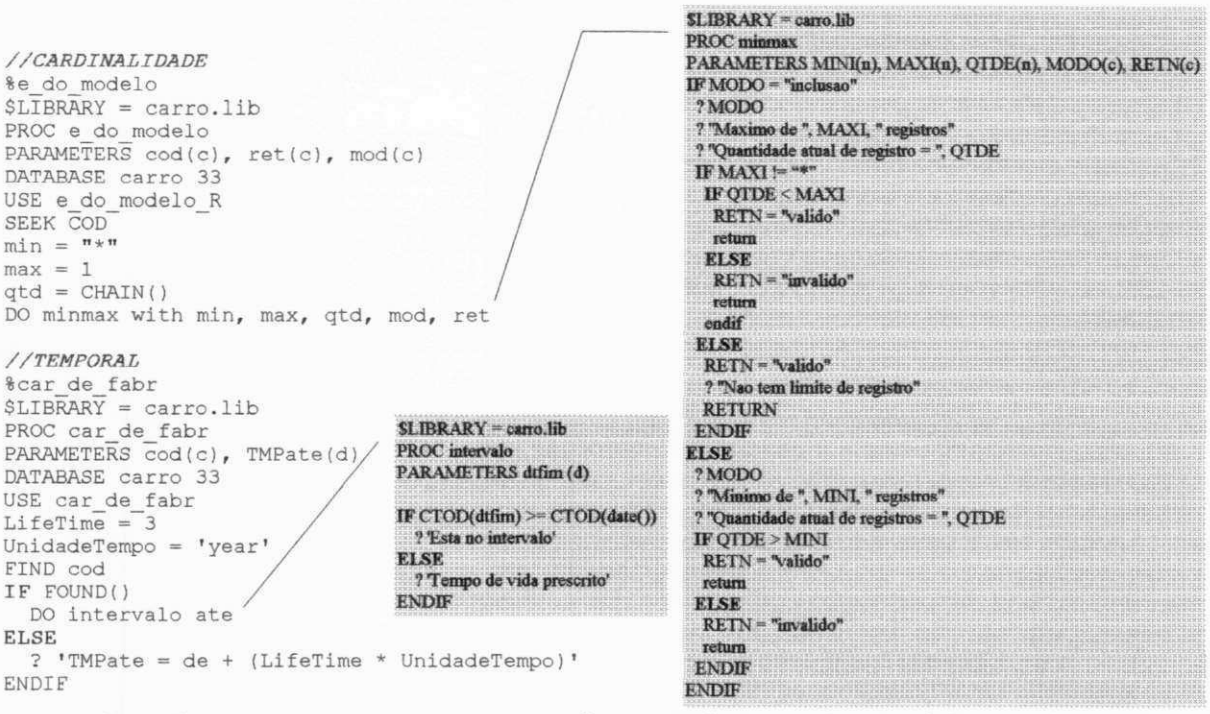
```
class ANO_PRODUCAO
       type integer

class DATA
       type integer

class NUM_SERIE
       type integer

class NUM_MOTOR
       type integer

class NUM_CHASSIS
       type integer

class NUM_CARBURADOR
       type integer

class NUM_RADIADOR
       type integer

class QUANTIDADE
       type integer

class QUANT_CARRO
       type integer

class CILINDRADA
       type integer

class QUANT_PORTA
```

3.1 TOM para ODMG

TOM e ODMG têm muitas similaridades em seus modelos, pois ambos adotam a tecnologia da orientação a objeto. Porém são dois modelos que acrescentam características especiais em cada um dos seus projetos. O modelo TOM possui, além das facilidades de um modelo orientado a objetos, regras de integridade associadas a cada método através de pré-condições e pós-condições, controle de eventos e regras, representação de tempo e de versões o que o caracteriza como um modelo temporal ativo. O modelo ODMG, tem em sua arquitetura todos os componentes desejáveis para manipular bem os objetos em uma arquitetura cliente/servidor, e por estar inserido em um ambiente padrão de desenvolvimento de aplicações orientadas objetos, facilita a comunicação entre estes objetos. Então estes modelos se complementam e pretendemos que haja uma correspondência entre ambos para representa-los unicamente.

Sintaticamente o mapeador se encarrega de fazer as transformações adequadas entre os dois modelos. As riquezas semânticas como, restrições de cardinalidade e controle do tempo do modelo TOM (efeitos colaterais), que não são possíveis representá-las no ODMG, são expressadas como linhas de código e armazenadas como procedimento, dentro de uma biblioteca que será "*linkada*" à aplicação correspondente do OpenBASE em tempo de compilação. No caso do controle da cardinalidade alguns parâmetros são passados para que o procedimento realize a operação como: código do objeto, retorno (válido ou inválido) e o tipo de operação. As variáveis de controle da cardinalidade `min` e `max` já foram anteriormente registradas no corpo do procedimento durante o mapeamento do modelo TOM para ODMG. Também são passados dois parâmetros para os procedimentos temporais os quais são: código do objeto e tempo. As variáveis do tempo também foram anteriormente registradas no corpo do procedimento durante o mapeamento do modelo TOM para ODMG. De dentro dos procedimento uma chamada é feita para outros procedimentos mais genéricos.

A seguir é mostrado um exemplo desses códigos na linguagem OPUS onde o primeiro é chamado `e_do_modelo`, que faz o controle da cardinalidade da tabela de relacionamento `e_do_modelo_R`, expressando a máxima e mínima cardinalidade definida no esquema TOM. O segundo procedimento tem o nome de `car_de_fabr` que faz o controle do tempo dos objetos da tabela, através dos dados registrados nas variáveis `LifeTime` e `UnidadeTempo` capturados também do esquema TOM.



```
//CARDINALIDADE
%e_do_modelo
$LIBRARY = carro.lib
PROC e_do_modelo
PARAMETERS cod(c), ret(c), mod(c)
DATABASE carro 33
USE e_do_modelo_R
SEEK COD
min = "*"
max = 1
qtd = CHAIN()
DO minmax with min, max, qtd, mod, ret

//TEMPORAL
%car_de_fabr
$LIBRARY = carro.lib
PROC car_de_fabr
PARAMETERS cod(c), TMPate(d)
DATABASE carro 33
USE car_de_fabr
LifeTime = 3
UnidadeTempo = 'year'
FIND cod
IF FOUND()
  DO intervalo ate
ELSE
  ? 'TMPate = de + (LifeTime * UnidadeTempo)'
ENDIF
```

```
$LIBRARY = carro.lib
PROC intervalo
PARAMETERS dtfim (d)

IF CTOD(dtfim) >= CTOD(date())
  ? 'Esta no intervalo'
ELSE
  ? 'Tempo de vida prescrito'
ENDIF
```

```
$LIBRARY = carro.lib
PROC minmax
PARAMETERS MINI(n), MAXI(n), QTDE(n), MODO(c), RETN(c)
IF MODO = "inclusao"
 ? MODO
 ? "Maximo de ", MAXI, " registros"
 ? "Quantidade atual de registro = ", QTDE
 IF MAXI != "*"
  IF QTDE < MAXI
   RETN = "valido"
   return
  ELSE
   RETN = "invalido"
   return
  endif
 ELSE
  RETN = "valido"
  ? "Nao tem limite de registro"
  RETURN
 ENDIF
ELSE
 ? MODO
 ? "Minimo de ", MINI, " registros"
 ? "Quantidade atual de registros = ", QTDE
 IF QTDE > MINI
  RETN = "valido"
  return
 ELSE
  RETN = "invalido"
  return
 ENDIF
ENDIF
```

As rotinas `minmax` e `intervalo` são procedimento de propósitos mais gerais e funciona para qualquer aplicação.

Apresentado o nosso ambiente de trabalho, baseando-nos nas sintaxes das linguagens declarativas, vamos detalhar as transformações feitas pelo mapeador começando pelas classes de objetos. As sintaxe das linguagens declarativas dos objetos estão no formato BNF nos apêndices correspondentes às linguagens dos modelos utilizados.

3.1.1 Mapeamento das Classes

3.1.1.1 Classes TOM

As classes no modelo TOM são de dois tipos: **primitivas**, que correspondem aos objetos imprimíveis nos modelos semânticos ou aos literais no ODMG e cujas instâncias são identificadas por tipos de dados simples; e **estruturadas** que correspondem aos tipos abstratos nos modelos semânticos. O algoritmo que faz esse mapeamento entende quando uma nova classe surge, pois um dicionário das palavras reservadas do TOM, é consultado toda vez que uma nova palavra for lida no esquema TOM. Quando a palavra for igual a "`class`", inicia-se a transformação de uma nova classe TOM para outra nova classe ODMG com todas as propriedades como atributos e relacionamentos. Os dados abstraídos referentes às classes TOM são armazenados em estruturas *`btree`* para serem recuperadas no momento da construção das classes ODMG.

O algoritmo principal que faz as chamadas para cada propriedade do modelo TOM a ser mapeada para o modelo ODMG é o seguinte:

```
construindo_Classes_ODMG ()
  { existir = Proxima_Palavra_do_Esquema_TOM (palavra);
   Enquanto (existir)
          { Se (palavra == "class")
             { existir = Proxima_Palavra_do_Esquema_TOM (palavra);
               Enquanto (palavra != "class")
                    { strcpy (Classe.Nome, palavra);
                      Se (palavra == "instance_relationships")
                          Enquanto ((!PalavraReservada(palavra)) && (existir))
                              relacionamento_instance ();
                      Se (palavra == "generalization_of")
                          Enquanto ((!PalavraReservada(palavra)) && (existir))
                              heranca_classe ();
                      Se (palavra == "grouping_of")
                          Enquanto ((!PalavraReservada(palavra)) && (existir))
                              grupo_classe ();
                      Se (palavra == "aggregation_of")
                          Enquanto ((!PalavraReservada(palavra)) && (existir))
                              agrega_classe ();
                    }
               Insere Classe na Btree; //metodo que registra a ocorrencia de uma classe
             }
          }
  };
```

A partir desse algoritmo é possível construir todas as propriedades das classes TOM, como por exemplo, os relacionamentos, generalizações, agrupamentos, agregações e outros. Para cada uma dessas propriedades, foi criado uma estrutura, onde os dados das respectivas propriedades serão armazenados (Detalhes dessas estruturas estão no Capítulo 4).Estas estruturas são manipuladas utilizando uma Btree. Após a leitura de todo o esquema TOM, os dados das classes declaradas nesse esquema são armazenados nas respectivas estruturas. Em seguida construímos o esquema ODMG, com base nas informações dessas estruturas. O algoritmo construtor abaixo faz o acesso aos dados das estruturas, através de métodos de acesso à Btree. Especificaremos melhor esta Btree no Capítulo 4.

```
escrevendo_esquema_ODMG ()
  {existir = Proxima_Palavra_do_Esquema_TOM(palavra);
   Enquanto (existir)
       {//Verifica se eh uma classe existente
        Se (classe esta presente na Btree(palavra))
           EscreveNomeClasse (TTOM_Classe);

        //Verifica se eh subclasse
        Se (heranca esta presente na Btree(palavra))
           EscreveHeranca(TTOM_Classe_Generica);

        //Verifica se Tipo do atributo é uma classe estruturada
        Se (relacionamento esta presente na Btree(TTOM_Tipo_Atrib))
           {EscreveNomeRelacionamento(TTOM_Nome_Atrib);
            EscreveNomeClasseRelacionada(TTOM_Tipo_Atrib);
           }
        Senao
           {//Se o tipo do atributo for simples
            EscreveNomeAtributo(TTOM_Nome_Atrib);
            EscreveTipoAtributo(TTOM_Tipo_Atrib);
           }
       }
  }
```

3.1.1.1.1 Classes Primitivas

Essas classes suportam os seguintes tipos primitivos:

- *integer*
- *string*
- real
- *boolean*

As classes abaixo são primitivas:

```
class NOME
        type string

class NUM_REGISTRO
        type integer
```

3.1.1.1.2 Classes Estruturadas

Composta do nome da classe, uma lista de chaves, uma lista de relacionamentos, uma lista de métodos, suas subclasses e superclasses e parâmetros de tempo. Veja como fica no exemplo simplificado abaixo, as declarações dessas classes:

```
//Classes estruturadas

class CARRO
        instance_relationships
                tem_num_reg     : NUM_REGISTRO (1,1)
                e_do_modelo     : MODELO (1,1)
        instance_methods
                insere_carro(),
                remove_carro()
        keys_are tem_num_reg

class MODELO
        instance_relationships
                modelo_nome     : NOME (1,1)
                tem_cons_comb   : CONS_COMBUSTIVEL (1,1)
                e_modelo_de     : CARRO (0,*)
        keys_are modelo_nome

//Classes primitivas

class NOME
        type string

class NUM_REGISTRO
        type integer
```

No exemplo acima, as classes NOME e NUM_REGISTRO são do tipo primitivas e as classes CARRO e MODELO são estruturadas.

3.1.1.2 Classes ODMG (interfaces)

Classes no modelo ODMG é entendida como interface, onde também são feitas as declarações das propriedades dos objetos, tais como: nome dos objetos, chaves, relacionamentos, subtipos e supertipos e uma lista de métodos.

Veja no exemplo simplificado, as declarações dessas interfaces no modelo ODMG:

```
interface carro {
  keys tem_num_reg;
  attribute integer tem_num_reg;
  relationship modelo e_do_modelo
       inverse modelo::e_modelo_de;
  void insere_carro();
  void remove_carro();
 };

keys modelo_nome;
  attribute string modelo_nome;
  relationship set<carro> e_modelo_de
       inverse carro::e_do_modelo;
};
```

O exemplo anterior, já é resultado do mapeamento do esquema TOM para ODMG, onde uma das mudanças foram as transformações das classes primitivas em atributos, ou seja, as classes primitivas foram absorvidas pelo ODMG como `literal` capturando apenas os tipos correspondentes nos modelos.

TOM		ODMG
`class NUM_REGISTRO` `  type integer`	⇨	`attribute integer tem_num_reg;`

3.1.2 Mapeamento dos Relacionamentos

Nos modelos semânticos de objetos os relacionamentos binários são definidos entre dois objetos e geralmente expressam sua cardinalidade mínima e máxima. O algoritmo simplificado para capturar os dados de relacionamento do modelo TOM é o seguinte:

```
relacionamento_instancia ()
  {Enquanto ((!palavra_reservada(palavra)) && (existir))
     {Atributo.TOM_Nome_Atrib = palavra;
      existir = Proxima_Palavra_do_Esquema_TOM (palavra);

      Se ((classe esta_presente na Btree (palavra)) && (for primitiva primitiva))
          {//Se classe existir e for classe primitiva
           Atributo.TOM_Tipo_Atributo = TTipo;
          }
      Senao
          {//Se classe existir e for classe estruturada
           Atributo.TOM_Tipo_Atrib = palavra;

           //GERA PROCEDIMENTO DE CARDINALIDADE
           //Esses metodos gera linhas de codigo para formar o procedimento que compoe
           //a biblioteca de efeitos colaterais.

           EscreveNomeProcedimentoCard(Atrib.TOM_Nome_Atrib;
           EscreveNomeBanco ();
           EscreveNomeTabelaRelacionamento ();

           existir = Parse. Proxima_Palavra_do_Esquema_TOM(palavra);

           //valor minimo
           Atributo.TOM_Card_Min = palavra;
           EscreveMinimo (palavra);

           existir = Parse. Proxima_Palavra_do_Esquema_TOM(palavra);

           //valor maximo
           Atributo.TOM_Card_Max = palavra;
           EscreveMaximo (palavra);
          }
     }
   Insere relacionamento na Btree;
  }
```

3.1.2.1 Relacionamento no TOM

Cada relacionamento no TOM possui uma cardinalidade como uma restrição de integridade, expressada pelo par de valores `(min, max)`, os quais determina o mínimo e o máximo de instâncias associadas pelo relacionamento. Cada relacionamento possui um relacionamento inverso, conforme mostra o exemplo abaixo:

```
class MODELO
        instance_relationships
                produzido_por   : FABRICANTE (1,1)

class FABRICANTE
        instance_relationships
                produz          : MODELO (1,*)
```

O relacionamento inverso do relacionamento `produzido_por` na classe `CARRO` é `produz` na classe `FABRICANTE`, onde carro é produzido por um único fabricante e este, produz vários carros.

3.1.2.2 Relacionamento no ODMG

O ODMG suporta, como no TOM somente relacionamentos binário, podendo ser a cardinalidade de um para um, um para muitos, ou muitos para muitos. Para determinar a cardinalidade maior que um no lado destino, um conjunto de tipos (*collection types*) são utilizados. Se essa opção for omitida, a cardinalidade será igual a um. Os tipos coleções são os seguintes:

- set ⇨ coleção desordenada sem duplicidade
- list ⇨ coleção ordenada com duplicidade
- bag ⇨ coleção desordenada com duplicidade

Usando a sintaxe ODL para declarar estes relacionamentos temos:

```
interface modelo {
  relationship fabricante produzido_por
        inverse fabricante::produz;
};

interface fabricante {
  relationship set<modelo> produz
        inverse modelo::produzido_por;
};
```

O relacionamento `produzido_por` na classe `CARRO`, tem seu inverso na classe `FABRICANTE produz`. Na classe `CARRO`, como foi omitido o tipo de coleção, a cardinalidade é igual a um na classe destino `FABRICANTE`. Na classe `FABRICANTE` o relacionamento `produz`, usa o tipo `set` para denotar a cardinalidade maior que um na classe `CARRO`. O ODMG não possui restrição de cardinalidade enumerada, mas pode ser conseguido através de métodos de controle de cardinalidades, que podem ser gerados pelo mapeador TOM para ODMG, utilizando os métodos e primitivas do próprio modelo ([Cat96a]).

3.1.3 Mapeamento das Abstrações Hierárquicas

São três as abstrações hierárquicas, generalização, agregação e agrupamento em ambos os modelos.

3.1.3.1 Generalização TOM

No TOM as propriedades comuns, que estão definidas na superclasse, são passadas às subclasses pela propriedade denominada `generalization_of`

```
class CARRO
        instance_relationships
                tem_num_reg     : NUM_REGISTRO (1,1)
                e_do_modelo     : MODELO (1,1)
        keys_are tem_num_reg
        generalization_of CAR_EM_USO, CAR_DESTRUIDO
```

Construímos o seguinte algoritmo para capturar os dados de heranças do TOM:

```
Heranca_Classe ()
  {//HERANCA
   Heranca.TOM_SubClasse = Classe.TOM_Nome_Classe;
   Heranca.TOM_SuperClasse = palavra;

   Insere heranca na Btree;
  }
```

3.1.3.2 Generalização ODMG

ODMG inclui herança baseada nos relacionamentos supertipo-subtipo e é comumente expressado pelo predicado `e_um_relacionamento` ou relacionamento `generalizacao-especializacao`. Por exemplo, `car_em_uso` e `car_destruido` são subtipos da classe `carro`:

```
interface car_em_uso : carro {
  relationship set<pessoa> car_uso_propried
        inverse pessoa::pes_proprietaria;
  attribute date de;
  attribute date ate;
 };

interface car_destruido : carro {
  attribute integer data_destruicao;
  attribute date de;
  attribute date ate;
 };
```

3.1.3.3 Agregação TOM

Formalmente, uma agregação é definida como um subconjunto do produto cartesiano dos objetos componentes e sua identificação é determinada pelas identificações de seus componentes. Construímos o seguinte algoritmo para capturar os dados de agregações do modelo TOM:

```
agrega_classe ()
  {
   Enquanto ((!palavra_reservada(palavra)) && (existir))
     {//Agregacao no modelo ODMG serah reconhecida como um "traversal path" com
      // cardinalidade um.

      Se ((classe esta_presente na Btree(palavra)) && (for classe primitiva))
        {//Se a classe componente for primitiva sera reconhecida como atributo
         // simples
         Agregacao.TOM_Nome_Class_Componente = TTOM_Tipo;
         Agregacao.TOM_Nome_Agregacao = palavra;
         Agregacao.TOM_Nome_Class_Agregada = Classe.TOM_Classe);
        }
      else
        {//Se a classe componente for estrutuda, faz-se uma concatenacao
         //do nome da classe componente com as iniciais Agr.
         Agregacao.TOM_Nome_Class_Componente = palavra;
         Agregacao.TOM_Nome_Agregacao = concatenacao("agr_", palavra);
         Agregacao.TOM_Nome_Class_Agregada = Classe.TOM_Classe);
        }
     }
   Insere agregacao Btree;
  }
```

No TOM, agregação é declarada pela propriedade `aggregation_of`

```
class CARRO
        instance_relationships
                tem_num_reg     : NUM_REGISTRO (1,1)
                e_do_modelo     : MODELO (1,1)
        keys_are tem_num_reg
        aggregation_of MOTOR, CHASSIS
```

3.1.3.4 Agregação ODMG

No ODMG, atributos ou relacionamentos, podem ser agregados para formar objetos. Cada atributo ou relacionamento nomeado como `parte_de`, devem ser especificado na chave do tipo em definição.

```
interface carro {
  keys agr_chassis, agr_motor, tem_num_reg;
  attribute integer tem_num_reg;
  attribute integer tem_num_ser;
  attribute chassis agr_chassis;
  attribute motor agr_motor;
};
```

3.1.3.5 Agrupamento TOM

Formalmente, o agrupamento é um subconjunto do conjunto dos elementos de uma dada classe e os grupos formados pelas partes deverão possuir uma nova identidade. Para realizar-mos este mapeamento consideramos o seguinte algoritmo para obtermos os dados de agrupamentos:

```
grupo_classe ()
  {//A partir da classe agrupada uma nova classe surge como contenedora dos objetos
   //agrupados. As iniciais grp eh concatenada ao nome da classe agrupada.
   Enquanto ((nao for palavra reservada(palavra)) && (existir))
     {
      Grupo.TOM_Nome_Class_Agrupada = Classe.TOM_Classe;
      Grupo.TOM_Nome_Class_Agrupamento = palavra;

      existir = Proxima_Palavra_do_Esquema_TOM (palavra);
      existir = Proxima_Palavra_do_Esquema_TOM (palavra);

      Grupo.TOM_Nome_Agrupamento = concatena("grp_", palavra);

      Insere agrupamento na Btree;
     }
  }
```

No TOM essa propriedade é declarada em `grouping_of`

```
class PESSOA
        instance_relationships
                pessoa_nome      : NOME (1,1)
                pes_proprietaria: CAR_EM_USO (1,*)
        keys_are pessoa_nome

class GRUPO_PROPRIETARIO
        instance_relationships
                grupo_nome      : NOME (1,1)
        keys_are grupo_nome
        grouping_of PESSOA using pes_proprietaria
```

3.1.3.6 Agrupamento ODMG

No ODMG o agrupamento é formado usando o tipo de coleção `set`. No exemplo abaixo, os elementos da classe `pessoa` serão agrupados na classe `grupo_proprietario`

```
interface pessoa {
  keys pessoa_nome;
  attribute string pessoa_nome;
  relationship set<car_em_uso> pes_proprietaria
        inverse car_em_uso::car_uso_propried;
 };

interface grupo_proprietario {
keys grupo_nome;
  attribute string grupo_nome;
  attribute set<pessoa> grp_pes_proprietaria;
 };
```

3.1.4 Mapeamento dos Aspectos Temporais

O modelo TOM adota o intervalo como o elemento primitivo na representação explícita, representado pelos instantes início e fim (respectivamente `de` e `ate` no esquema TOM) que compõem este intervalo. Os valores início e fim são unidades de tempo pré-estabelecidas ([Sch83]). Abstraímos estes valores temporais no modelo ODMG, como dois atributos do tipo `date` dentro de cada classe temporal. Para abstração do tempo dentro das classes ODMG concebemos o seguinte algoritmo:

```
temporal_classe ()
  {Temp.TOM_Nome_Classe_Temp = Classe.TOM_Classe;

   //Classe com life_time. Instancias podem pertencer a classe
   //somente no periodo de tempo especificado.

   existir = Proxima_Palavra_do_Esquema_TOM (palavra);
   Temp.TOM_LifeTime = palavra;

   existir = Proxima_Palavra_do_Esquema_TOM (palavra);
   Temp.TOM_Tempo = palavra;

   //GERA PROCEDIMENTO TEMPORAL

   EscreveNomeProcedimentoTemp (Temp.TOM_Nome_Classe_Temp);
   EscreveNomeBanco ();
   EscreveNomeTabelaTemporal();
   EscreveLifeTime (Temp.TOM_LifeTime);
   EscreveUnidadeTempo (Temp.TOM_Tempo);
   EscreveCorpoRotina ();

   Insere dados temporais na Btree;
  }
```

Após todo o mapeamento, é gerado um esquema ODL do modelo ODMG descrito em seguida. Esse esquema ODL tem a seguinte representação gráfica:

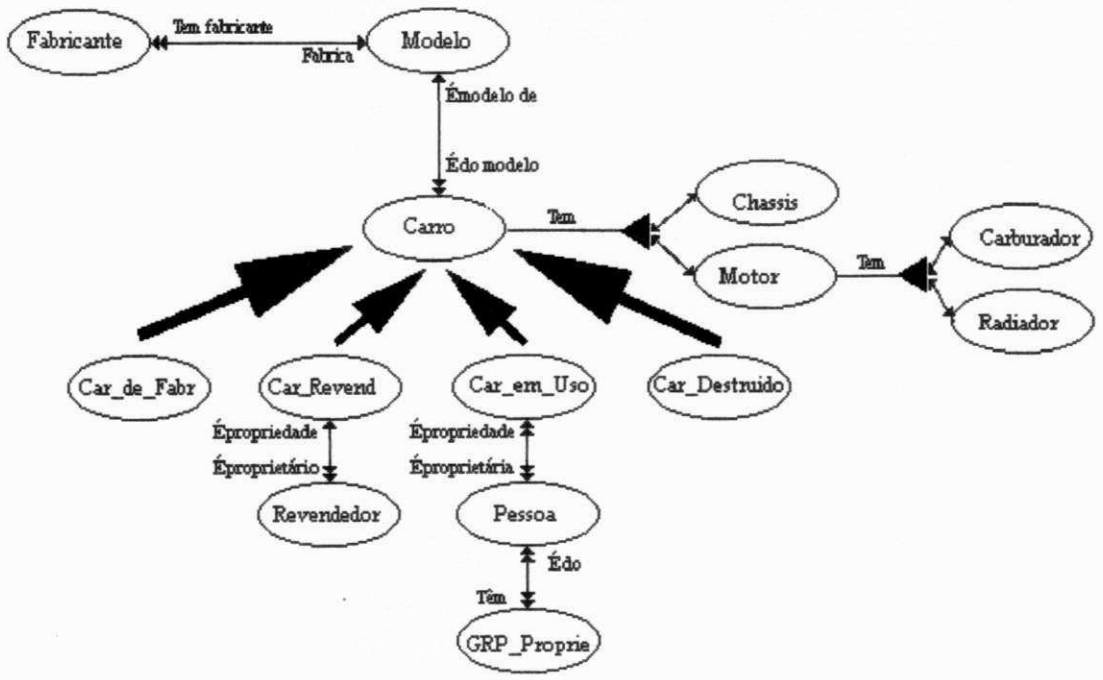


Figura 3.3 - Representação gráfica em ODMG para aplicação **Registro de Carros**

Esquema ODL:

```
module reg_car {

interface carro {
keys agr_chassis, agr_motor, tem_num_reg;
  attribute integer tem_num_reg;
  relationship modelo e_do_modelo
       inverse modelo::e_modelo_de;
  attribute integer produzido_em;
  attribute integer tem_num_ser;
  attribute chassis agr_chassis;
  attribute motor agr_motor;
  void insere_carro();
  void remove_carro();
 };

interface fabricante {
keys fabricante_nome;
  attribute string fabricante_nome;
  relationship set<modelo> produz
       inverse modelo::produzido_por;
  void insere_fabricante();
  void remove_fabricante();
 };

interface car_de_fabr : carro {
  attribute date de;
  attribute date ate;
 };

interface car_revendedor : carro {
  relationship revendedor e_da_empresa
       inverse revendedor::emp_proprietaria;
  attribute date de;
  attribute date ate;
 };

interface revendedor {
keys revendedor_nome;
  attribute string revendedor_nome;
  relationship set<car_revendedor> emp_proprietaria
       inverse car_revendedor::e_da_empresa;
```

```
 };

interface car_em_uso : carro {
  relationship set<pessoa> car_uso_propried
        inverse pessoa::pes_proprietaria;
  attribute date de;
  attribute date ate;
 };

interface car_destruido : carro {
  attribute integer data_destruicao;
  attribute date de;
  attribute date ate;
 };

interface pessoa {
keys pessoa_nome;
  attribute string pessoa_nome;
  relationship set<car_em_uso> pes_proprietaria
        inverse car_em_uso::car_uso_propried;
 };

interface grupo_proprietario {
keys grupo_nome;
  attribute string grupo_nome;
  attribute set<pessoa> grp_pes_proprietaria;
 };

interface modelo {
keys modelo_nome;
  attribute string modelo_nome;
  attribute integer tem_cons_comb;
  relationship set<carro> e_modelo_de
        inverse carro::e_do_modelo;
  relationship fabricante produzido_por
        inverse fabricante::produz;
 };

interface motor {
keys agr_carburador, agr_radiador, tem_num_motor;
  attribute integer tem_num_motor;
  attribute integer tem_cilindrada;
  attribute carburador agr_carburador;
  attribute radiador agr_radiador;
 };

interface chassis {
keys tem_num_chassis;
  attribute integer tem_num_chassis;
  attribute integer quantidade_porta;
 };

interface carburador {
keys tem_num_carb;
  attribute integer tem_num_carb;
  attribute string tem_marca_carb;
 };

interface radiador {
keys tem_num_rad;
  attribute integer tem_num_rad;
  attribute string tem_marca_rad;
 };
 };
```

3.2 ODMG para OpenBASE

Os esquemas na sintaxe ODL do modelo ODMG, serão traduzidos para esquemas relacionais estendidos OpenBASE. Para tanto, um conjunto de regras de mapeamento foram implementadas ([Rum90], [Bla94]). Nos baseamos nesses regras para formalizarmos as seguintes especificações ([Cun90]):

1. Para cada classe defina uma relação com o mesmo nome da classe;
2. Para cada relacionamento 1-1 defina um atributo da tabela;
3. Para cada relacionamento n-m ou n-1 defina uma relação com o nome deste relacionamento e como atributos as chaves das classes relacionadas;
4. A chave de uma agregação é a chave composta das chaves dos componentes;
5. Em uma generalização com chaves distintas, inclua a chave geral nas subclasses;
6. Em um agrupamento, inclua a chave dos elementos que compõem o grupo na relação definida pertinente à classe do agrupamento;
7. Em uma classe com tempo, crie dois atributos '`de`' e '`ate`'.

É importante lembrar, que a aplicação **Registro de Carros** foi originalmente modelada no TOM e também que algumas semânticas, como cardinalidade enumerada e informações temporais, não foram possíveis mapear diretamente para o modelo ODMG. Mas isto não implicou em perdas semânticas nestes dois casos específicos, porque durante o processo de conversão, foram construídas para cada objeto, procedimentos correspondentes com essas funcionalidades (efeitos colaterais). Estes procedimentos conforme mencionamos anteriormente são compilados e armazenados numa biblioteca criada pelo compilador Opus do OpenBASE. Os nomes destes procedimentos e da biblioteca são gerados automaticamente, baseando-se nos nomes das tabelas e do esquema respectivamente. Esses procedimentos podem também serem gerados e armazenados em uma biblioteca ODMG utilizando a linguagem binding C++.

3.2.1 Mapeamento de classe para tabela

Cada classe pode ser mapeada para uma ou mais tabelas, se esta expressa relacionamentos com outras classes, ou essa classe é uma superclasse de uma subclasse. O algoritmo que faz esse mapeamento entende quando uma nova classe surge, pois um dicionário das palavras reservadas do ODMG, é consultado toda vez que

uma nova palavra for lida no esquema ODMG. Quando a palavra for igual a "interface", inicia-se a transformação de uma nova classe ODMG para outra nova tabela do OpenBASE com todas suas propriedades tais como atributos e relacionamentos. Os dados abstraídos referentes às tabelas OpenBASE são armazenados em estruturas *btree* para serem recuperadas no momento da construção das tabelas OpenBASE.

O algoritmo principal que faz as chamadas para cada propriedade do modelo ODMG a ser mapeada para o modelo relacional é o seguinte:

```
Contruindo_tabela_relacional()
  {existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
   Enquanto (existir palavra dentro do esquema)
     {Se (palavra == "interface")
        {Classe.ODMG_Nome_Classe == palavra);
         Proxima_Palavra_do_Esquema_ODMG (palavra);

         Se (nao for palavra reservada)
            heranca_classe ();

         Se (palavra == "keys")
           {Proxima_Palavra_do_Esquema_ODMG (palavra);
            Enquanto ((nao for palavra reservada) && (existir))
               keys_classe ();
           }
            Enquanto ((palavra for diferente de "interface") && (existir))
               {Se (palavra == "attribute")
                  {Proxima_Palavra_do_Esquema_ODMG (palavra);
                   Enquanto ((nao for palavra reservada) && (existir))
                      atributos_classe ();
                  }
                Se (palavra == "relationship")
                  {Proxima_Palavra_do_Esquema_ODMG (palavra);
                   Enquanto ((nao for palavra reservada) && (existir))
                      relacionamentos_classe ();
                  }
               }
         Insere Classe na Btree;
        }
     }
  }
```

A partir desse algoritmo é possível construir todas as tabelas e seus relacionamentos no OpenBASE. Para cada uma dessas tabelas (entidades e relacionamentos), foi criado uma estrutura onde os dados das respectivas tabelas serão armazenados. Estas estruturas são manipuladas utilizando uma Btree. Após a leitura de todo o esquema ODMG e todos os dados das classes declaradas nesse esquema terem sido armazenados nas suas respectivas estruturas, construímos o esquema OpenBASE utilizando o algoritmo seguinte, que os dados das estruturas da Btree.

```
Escrevendo_esquema_OpenBASE ()
  {existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
   Enquanto (existir)
     {
      Se (Classe esta_presente na Btree (palavra))
          // Ao se constatar a existencia dessa classe todas as informacoes sobre
          // ela serao armazenadas em variaveis temporarias com a inicial T.
          {EscreveNomeTabela (TODMG_Nome_Classe);

      Se (TODMG_Key_Classe == "composta")
          {EscreveCodigoClasseAgregada (TODMG_Nome_Classe);
           EscreveChavesComposta ();
          }

      Se (Heranca esta_presente na Btree (TODMG_Nome_Classe))
          {EscreveChavePrimaria (TODMG_Nome_Chave);
           EscreveTipoChavePrimaria (TODMG_Tipo_Chave);
          }

      Enquanto ((palavra for diferente de "interface") && (existir))
          {Enquanto ((nao for palavra reservada) && (existir palavra dentro do esquema))
              { //Verifica se e um atributo e se este nao foi considerado como uma chave.
                //Como chave ele ja eh considerado um atributo, por isso nao eh necessario
                //escreve-lo
                Se ((Atributo esta_presente na Btree) && (Atributo nao for chave primaria))
                   {EscreveNomeAtributo (TODMG_Nome_Atrib);
                    EscreveTipo (TODMG_Tipo_Atrib);
                   }
              }
          }
     }
  }
```

3.2.2 Mapeamento dos relacionamentos para tabelas

Um relacionamento será mapeado para uma única tabela. As chaves primárias de ambas as classes relacionadas se torna a chave primária dessa nova tabela. A forma de mapeamento da Figura 3.4 se aplica a relacionamentos n-m e 1-n. Os relacionamentos 1-1 podem virar atributos.

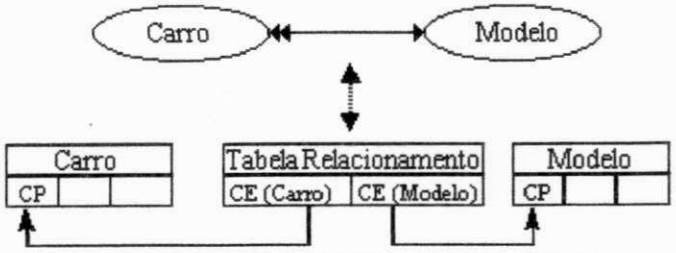


Figura 3.4 - Mapeamento de relacionamento para tabelas

Escrevemos o seguinte algoritmo para abstrair as informações referentes aos relacionamentos:

```
Relacionamentos_Classe ()
  {Enquanto ((nao for palavra reservada) && (existir palavra dentro do esquema))
     {Se ((palavra == "set") || (palavra == "list") || (palavra == "bag"))
         Relacionamento.ODMG_Cardinalidade_Relacionamento = palavra; //set, list ou bag
      Senao
         Relacionamento.ODMG_Cardinalidade_Relacionamento = "ref";

      Relacionamento.ODMG_Classe_Relacionamento_Origem = Classe.ODMG_Nome_Classe;
      existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
      Relacionamento.ODMG_Classe_ Relacionamento_Destino = palavra;
      existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
      Relacionamento.ODMG_Nome_Relacionamento = palavra;
      existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
      Relacionamento.ODMG_Nome_Relacionamento_Inverso = palavra;

      Insere na Btree Relacionamento;

      existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
     }
  }
```

3.2.3 Mapeamento das generalizações para tabelas

Consideramos apenas herança simples em nosso trabalho (embora ambos modelos suportam herança simples e múltiplas) onde a superclasse e cada subclasse são mapeadas para tabelas separadas. As chaves primárias das tabelas originadas das subclasses, são herdadas da tabela gerada a partir da superclasse (Figura 3.5).

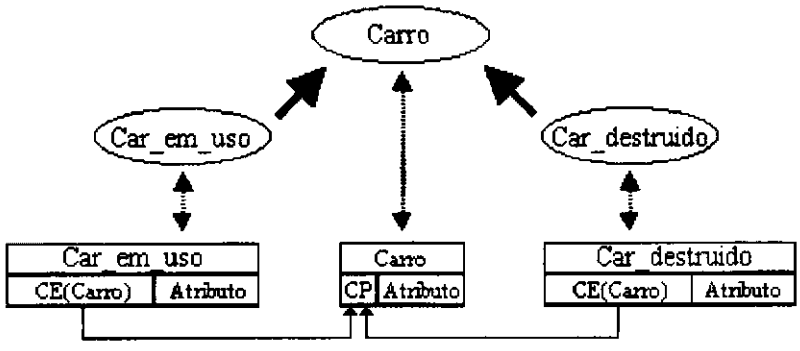


Figura 3.5 - Mapeamento de generalizações para tabelas

Aplicamos o seguinte algoritmo para abstrair os dados de heranças das classes:

```
Heranca_Classe ()
  {//HERANCA
   Heranca.ODMG_SubClasse == Classe.ODMG_Nome_Classe;
   Heranca.ODMG_SuperClasse == palavra;

   Insere Heranca na Btree;
  }
```

3.2.4 Mapeamento das agregações para tabelas

Entendendo a agregação como um subconjunto do produto cartesiano dos objetos componentes, uma chave composta das chaves das tabelas componentes, é criada na tabela agregada (originada da classe agregada), para identificar todas as componentes (Figura 3.6).

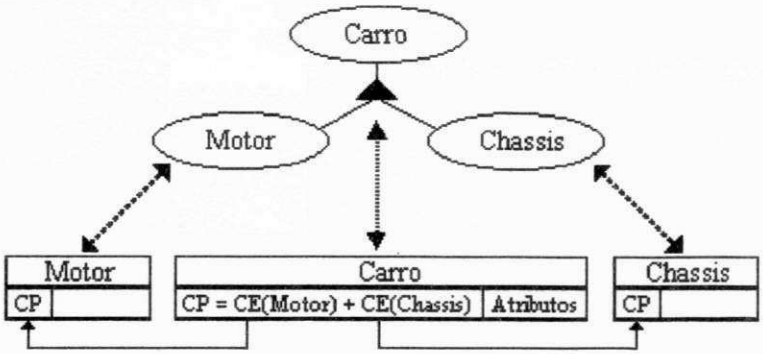


Figura 3.6 - Mapeamento de Agregações para tabelas

Escrevemos o seguinte algoritmo para capturar os dados sobre agregação das classes:

```
Agregacao_Classe ()
  {Enquanto ((nao for palavra reservada) && (existir palavra dentro do esquema))
     {Se ((palavra nao for tipo basico) && (cardinalidade do atributo ODMG == "ref"))
         {//AGREGACAO
          Agregacao.ODMG_Classe_Agregada = Classe.ODMG_Nome_Classe;
          Agregacao.ODMG_Classe_Componente = Nome_da_Classe_Componente; //Tipo
          existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
          Agregacao.ODMG_Nome_Agregacao = Nome_da_Agregacao;

          Insere na Btree Agregacao;
         }
      existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
     }
  }
```

3.2.5 Mapeamento dos agrupamentos para tabelas

Duas tabela são obtida para representar o agrupamento. Uma tabela do tipo entidade para conter os grupos e outra do tipo relacionamento para conter os elementos agrupados de uma outra tabela. A chave da primeira tabela identifica o agrupamento (Figura 3.7).

Figura 3.7 - Mapeamento de agrupamentos para tabelas

Para capturar os dados sobre agrupamento das classes escrevemos o seguinte algoritmo:

```
Agrupamento_Classe ()
  {Enquanto ((nao for palavra reservada) && (existir palavra dentro do esquema))
     {Se ((palavra nao for tipo basico) && (cardinalidade do atributo ODMG == "set"))
        {//GRUPAMENTOS
         Agrupamento.ODMG_Classe_Agrupamento = Classe.ODMG_Nome_Classe;
         existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
         Agrupamento.ODMG_Classe_Agrupada = palavra);

        Insere na Btree Agrupamento;
        }
      existir = Proxima_Palavra_do_Esquema_ODMG (palavra);
     }
  }
```

3.3 Exemplo:

A representação gráfica (diagrama entidade-relacionamento) do esquema **Registro de Carro** no OpenBASE é dada em seguida (Figura 3.8):

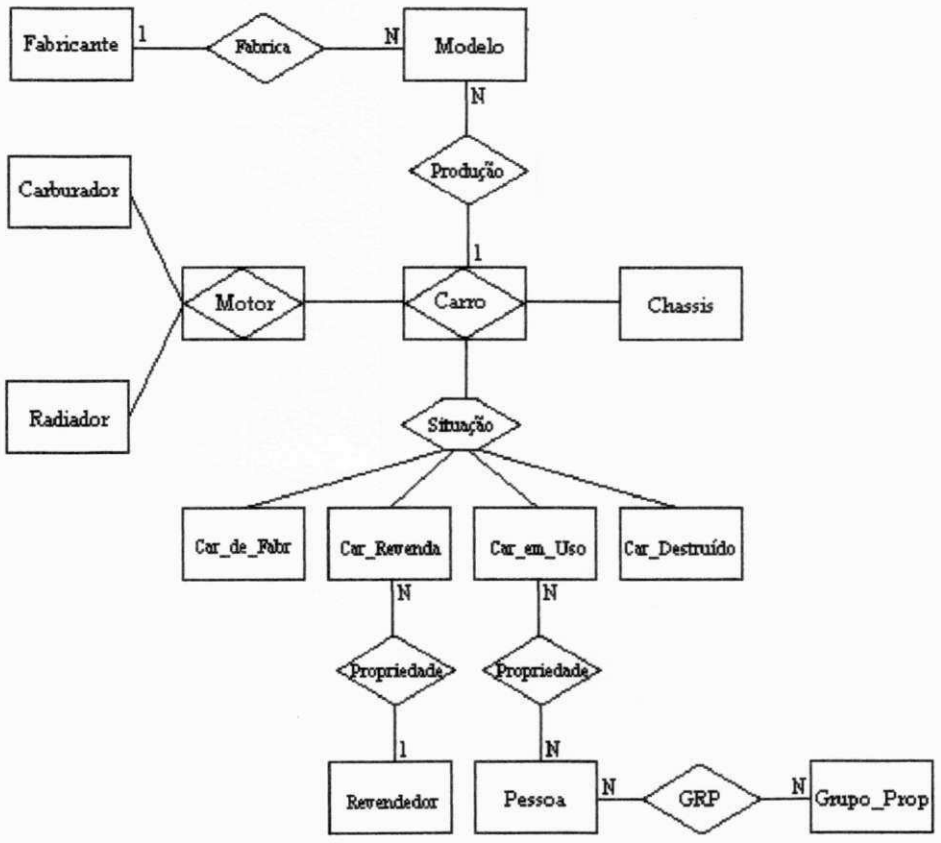


Figura 3.8 - Representação gráfica do esquema OpenBASE.

O esquema ODL da aplicação **Registro de Carro** após o mapeamento, resultou no seguinte esquema OpenBASE.

```
$CONTROLE NOMEIGUAL
BANCO: reg_car 33

NOME: carro E
Cod_carro (1) N10
tem_num_chassis (chassis) N5 POS Cod_carro
Cod_motor (motor) N5 POS Cod_carro + 5
tem_num_reg N5
produzido_em N5
tem_num_ser N5

NOME: fabricante E
fabricante_nome (3) U5

NOME: car_de_fabr E
cod_car_de_fabr (3) N5
Cod_carro (carro) N5 POS cod_car_de_fabr
de D2
ate D2

NOME: car_revendedor E
cod_car_revendedor (3) N5
Cod_carro (carro) N5 POS cod_car_revendedor
de D2
ate D2
```

```
NOME: revendedor E
revendedor_nome (3) U5

NOME: car_em_uso E
cod_car_em_uso (3) N5
Cod_carro (carro) N5 POS cod_car_em_uso
de D2
ate D2

NOME: car_destruido E
cod_car_destruido (3) N5
Cod_carro (carro) N5 POS cod_car_destruido
data_destruicao N6
data_destruicaoDia N2 POS data_destruicao
data_destruicaoMes N2 POS data_destruicao + 2
data_destruicaoAno N2 POS data_destruicao + 4
de D2
ate D2

NOME: pessoa E
pessoa_nome (3) U5

NOME: grupo_proprietario E
grupo_nome (3) U5

NOME: modelo E
modelo_nome (3) U5
tem_cons_comb N5

NOME: motor E
Cod_motor (1) N10
tem_num_carb (carburador) N5 POS Cod_motor
tem_num_rad (radiador) N5 POS Cod_motor + 5
tem_num_motor N5 POS Cod_motor + 10
tem_cilindrada N5

NOME: chassis E
tem_num_chassis (3) N5
quantidade_porta N5

NOME: carburador E
tem_num_carb (3) N5
tem_marca_carb U5

NOME: radiador E
tem_num_rad (3) N5
tem_marca_rad U5

NOME: grp_pes_proprietaria_R
Cgrp_pes_proprietaria (Grupo_Proprietario) U5
grp_pes_proprietaria (pessoa) U5

NOME: e_do_modelo_R R
e_modelo_de (carro)  N5
e_do_modelo (modelo)  U5

NOME: produz_R R
produzido_por (fabricante)  U5
produz (modelo)  U5

NOME: emp_proprietaria_R R
e_da_empresa (revendedor)  U5
emp_proprietaria (car_revendedor)  N5

NOME: pes_proprietaria_R R
pes_proprietaria (car_em_uso)  N5
car_uso_propried (pessoa)  U5
```

4. Implementação

Os mapeadores foram implementados em C++. A implementação em C++, sem dúvida nos deu um maior entendimento dos objetos, tanto conceitual como das duas estruturas. Muitos dos métodos implementados são utilizados nos dois projetos, exceto os métodos das regras de mapeamento. As regras de mapeamento do modelo TOM para o modelo ODMG obedece a sintaxe ODL/IDL (Apêndice B). O mapeamento do ODMG para o modelo relacional, no entanto, implementa uma lógica bem diferente em seu projeto. Haja visto, que já discutimos detalhadamente os métodos específicos dos mapeamentos, tanto do modelo TOM para ODMG, como do modelo ODMG para o Relacional OpenBASE no capítulo anterior, neste capítulo, daremos maior destaque aos métodos comuns a ambos os projetos: TOM/ODMG e ODMG/OpenBASE.

4.1 Especificação do Mapeador do Modelo TOM para ODMG

O projeto desse mapeador está composto de quatro classes as quais são: `palavra`, `btree`, `contrucao_classe_ODMG` e `escreve_Classe_ODMG` - falaremos com mais detalhes posteriormente. A geração é realizada em duas fases. Na primeira faz-se uma leitura para abstrair as classes com suas respectivas propriedades (nome, *keys*, atributos, relacionamentos, heranças, agrupamentos e agregações) que são armazenadas nas *structs* correspondentes (para cada uma dessas propriedades existe uma *struct*). Essas *struct* são armazenadas em uma *Btree*, o que possibilita a recuperação das informações dessas propriedades, para construir as classes do ODMG posteriormente. Na segunda fase escreve-se o esquema ODMG (um arquivo com extensão "`.ODL`" é gerado), seguindo a ordem original das classes do modelo TOM. Os detalhes das estruturas e das classes vêm a seguir.

Estruturas de armazenamento dos dados das classes do esquema TOM

```
//-------------------------------------------------------------
//Estrutura que armazena informacoes sobre uma determinda classe
struct TOM_Classe_Struct
{char  TOM_Classe [30],
}; TOM_Classe_Struct Classe;

//-------------------------------------------------------
//Estrutura que armazena informacoes sobre as subclasses
struct TOM_Heranca_Struct
{char  TOM_Classe_Heranca  [30],
       TOM_Classe_Generica [30];
}; TOM_Heranca_Struct Heranca;
```

```
//-------------------------------------------------
//Estrutura que armazena informacoes sobre os tipos
struct TOM_Atrib_Struct
{char   TOM_Nome_Atrib  [30],
       TOM_Tipo_Atrib  [30], //simples ou estruturado(classe definada pelo usuario)
       TOM_Card_Min    [5],
       TOM_Card_Max    [5],
       TOM_Class_Atrib [30];
}; TOM_Atrib_Struct Atributo;

//-------------------------------------------------------
//Estrutura que armazena informacoes sobre os grupamentos
struct TOM_Grup_Struct
{char   TOM_Nome_Grupamento       [30],
       TOM_Nome_Class_Grupada    [30],
       TOM_Nome_Class_Grupamento [30];
}; TOM_Grup_Struct Grupamento;

//-------------------------------------------------------
//Estrutura que armazena informacoes sobre as agregacoes
struct TOM_Agreg_Struct
{char   TOM_Nome_Class_Componente [30],
       TOM_Nome_Agregacao        [30],
       TOM_Nome_Class_Agregada   [30];
}; TOM_Agreg_Struct Agregacao;

//-------------------------------------------------------
//Estrutura que armazena informacoes das classes temporais
struct TOM_Temp_Struct
{char   TOM_Nome_Classe_Temp [30],
       TOM_LifeTime         [30],
       TOM_Tempo            [30];
}; TOM_Temp_Struct Temporal;
```

Classes declaradas no sistema de mapeamento

A classe `controi_classe_ODMG`, utiliza os método `insere`, `esta_presente` e `remove` da classe `btree` para gerenciar os dados na estrutura *btree*. Os principais métodos dessa classe são os seguintes:

- `void atributos_classe ();`
- `void relacionamentos_classe ();`
- `void heranca_classe ();`
- `void grupo_classe ();`
- `void agrega_classe ();`
- `Temporal_Classe ()`
- `void keys_classe ();`
- `int  palavra_reservada (char *st);`
- `int  tipos_basicos (char *st);`

A classe `escreve_esquema_ODMG` escreve o esquema ODMG e inclui os seguintes métodos com os seus respectivos parâmetros:

- `void EscreveModulo(char *st);`
- `void EscreveNomeClasse(char *st);`

- `void EscreveNomeAtributo(char *st);`
- `void EscreveHeranca(char *st);`
- `void EscreveNomeRelacionamento (char *st);`
- `void EscreveNomeGrupamento (char *st);`
- `void EscreveNomeAgregacao (char *st);`
- `void EscreveAtributosDoTempo (char (*st);`

A classe `palavra` deve incluir os seguintes métodos com os seus respectivos parâmetros:

- Um construtor que aloca memória para um *buffer*. Este *buffer* é usado para armazenar todo ou parte do esquema TOM - *input*.
- `int abre_arquivo (char *filename)` - Este método retorna 1 se o nome do arquivo existir e for aberto com sucesso; caso contrário, retorna 0.
- `int pega_proxima_palavra (char *st)` - Este método retorna 1 se uma próxima palavra, delimitada por caracteres não alfabéticos e não numéricos, com exceção de "_" e "*", é encontrada no esquema TOM, caso contrário, ele retorna 0. Se uma próxima palavra é encontrada, ela é retornada na *string* `st`.

Como não foi feita uma abordagem mais detalhada da classe `palavra` no Capitulo Mapeamento dos Modelos, por ser considerada uma classe auxiliar para realizar os mapeamentos, mostraremos a implementação dos métodos dessa classe a seguir. Essa mesma classe é usada para realizar-mos o mapeamento do esquema ODMG para OpenBASE.

```
//-----------------------------------------------
//Implementacao da Classe Palavra

#include "palavra.h"

const int buffer_size = 15000; //quinze mil bytes

//-----------------------------------------------
palavra::palavra()
  {buffer = new char [buffer_size];
  }

//------------------------------
int palavra::Caracter_Valido(char ch)
  {//Retorna 1 se ch e um caracter valido.

   return((ch >= 'A'  &&  ch <= 'Z') || (ch >= 'a'  &&  ch <= 'z') ||
          (ch >= '0'  &&  ch <= '9') || (ch == ' '));
  }

//-----------------------------------------------
int palavra::abre_arquivo (char*filename)
{if ((fp = fopen (filename, "r")) == 1)
    {Carrega_Buffer ();
     return 1;
    }
}
```

```
//-------------------------------------------
void palavra::Carrega_buffer()
  {size = fread(buffer, 1, buffer_size, fp);
   posicao = 0;
  }

//-----------------------------------------
int palavra::Proxima_Palavra_do_Esquema_ODMG (char *palavra)
  {//Posiciona índice no início de uma palavra

   int index = 0;

   While ((posicao < size) && (!Caracter_Valido (buffer [posicao])))
       posicao++;

   //Posiciona índice no final de uma palavra e acrescenta o caracter de terminacao

   While ((posicao < size) && (Caracter_Valido (buffer [posicao])))
       palavra [index++] = buffer [posicao++];

   palavra [index] = '\0';
  }
```

A classe `btree` deve incluir os seguintes métodos com os seus respectivos parâmetros:

- Um construtor que inicia uma estrutura *btree*.
- `void insere (char *struc)` - Este método insere informações na estrutura.
- `void remove (char *struc)` - Este método remove informações da estrutura, pois cada relacionamento n-m gera duas relações, uma pode ser removida.
- `int esta_presente (char *struct)` - Este método verifica se a informação já está presente na estrutura.

Como não foi feita uma abordagem mais detalhada da classe `btree` no Capitulo Mapeamento dos Modelos, por ser considerada uma classe auxiliar para realizar os mapeamentos, mostraremos a implementação dos métodos dessa classe a seguir. Essa mesma classe é usada para realizar-mos o mapeamento do esquema ODMG para OpenBASE.

```
//-----------------------------
//Implementacao da Classe Btree
#include "btree.h"
void btree::insere (char *palavra)
 {node * parente_proximo, *valor_corrente;
  while (valor_corrente && !encontrou)
       {//a funcao valor_igual foi declarada como um ponteiro para uma funcao
        if ((*valor_igual) (valor_corrente -> conteudo, palavra))
           encontrou = 1;
        else
           {parente_proximo = valor_corrente;
            //a funcao valor maior foi declarada como um ponteiro para uma funcao
            if ((*valor_maior) (palavra, valor_corrente -> conteudo))
               valor_corrente = valor_corrente -> lado_esquerdo;
            else
               valor_corrente = valor_corrente -> lado_direito;
           }
       }
  if (!encontrou)
     {if (!parente_proximo)
         {//Primeiro no na arvore
          root = new node;
          root -> lado_esquerdo = root -> lado_direito = 0;
```

```
            //Aloca espaco para conteudo
            root -> conteudo = new char [size];

            //Executa uma transferencia byte a byte dos dados de a para conteudo
            for (int i = 0; i < size; i++)
                root -> conteudo [i] = palavra[i];
          }
        else
          {if ((*valor_maior)  (palavra,  parente_proximo -> conteudo))
              {//Acrescenta novo no a lado esquerdo do pai
               node *new_node = new node;

               //Aloca espaco para conteudo
               new_node -> conteudo = new char[size];

               //Executa uma transferencia byte a byte dos dados de a para conteudo
               for (int i = 0; i < size; i++)
                   new_node -> conteudo[i] = palavra[i];

               new_node -> lado_esquerdo = new_node -> lado_direito = 0;
               parente_proximo -> lado_esquerdo = new_node;
              }
           else
              {//Acrescenta novo no a direta do pai
               node *new_node = new node;

               //Aloca espaco para conteudo
               new_node -> conteudo = new char[size];

               //Executa uma transferencia byte a byte dos dados de a para conteudo
               for (int i = 0; i < size; i++)
                   new_node -> conteudo[i] = palavra[i];

               new_node -> lado_esquerdo  = new_node -> lado_direito = 0;
               parente_proximo -> lado_direito = new_node;
              }
          }
      }
}
```

4.2 Especificação do Mapeador do Modelo ODMG para OpenBASE

O projeto desse mapeador está composto de cinco classes as quais são `palavra`, `btree`, `constroi_tabela_OpenBASE` e `escreve_tabela_OpenBASE` - falaremos com mais detalhes dessas classes posteriormente. A geração é realizada em duas fases. Na primeira faz-se uma leitura para abstrair as classes com suas respectivas propriedades (nome, *keys*, atributos, relacionamentos, heranças, agrupamentos e agregações) que são armazenadas nas *structs* correspondentes (para cada uma dessas propriedades existe uma *struct*). Essas *struct* são armazenadas em uma *Btree*, o que possibilita a recuperação das informações dessas propriedades, para construir as tabelas do OpenBASE posteriormente. Na segunda fase faz-se uma segunda leitura para escrever o esquema relacional (um arquivo com extensão "`.esq`" é gerado), seguindo a ordem original das classes do modelo ODMG. Os detalhes das estruturas e das classes vêm a seguir.

Estruturas de armazenamento dos dados das classes do esquema ODMG

```
//-------------------------------------------------------------------------
//Estrutura que armazena informacoes sobre uma determinda classe do esquema ODMG

struct ODMG_Classe_Struct
{char ODMG_Nome_Classe  [30],
      ODMG_Key_Classe   [30];
}; ODMG_Classe_Struct Classe;

//-----------------------------------------------------
//Estrutura que armazena informacoes sobre as subclasses

struct ODMG_Heranca_Struct
{char ODMG_SubClasse   [30],
      ODMG_Nome_Chave  [30],
      ODMG_Tipo_Chave  [30],
      ODMG_SuperClasse [30];
}; ODMG_Heranca_Struct Heranca;

//-------------------------------------------------------------------
//Estrutura armazena informacoes sobre as chaves de determinda classe

struct ODMG_Key_Struct
{char ODMG_Nome_Key   [30],
      ODMG_Classe_Key [30];
}; ODMG_Key_Struct Key;

//-----------------------------------------------------
//Estrutura que armazena informacoes sobre os atributos

struct ODMG_Atrib_Struct
{char ODMG_Nome_Atrib   [30],
      ODMG_Tipo_Atrib   [30],
      ODMG_Card_Atrib   [30],
      ODMG_Classe_Atrib [30];
}; ODMG_Atrib_Struct Atributo;

//-----------------------------------------------------------
//Estrutura que armazena informacoes sobre os relacionamentos

struct ODMG_Rel_Struct
{char ODMG_Nome_Rel      [30],
      ODMG_Card_Rel      [30],
      ODMG_Classe_Rel    [30],
      ODMG_Nome_Rel_Inv  [30],
      ODMG_Classe_do_Rel [30];
}; ODMG_Rel_Struct Relacionamento;

//-----------------------------------------------------
//Estrutura que armazena informacoes sobre os Grupamentos

struct ODMG_Grup_Struct
{char  ODMG_Nome_Grupamento   [30],
       ODMG_Card_Grp          [4],
       ODMG_Classe_Grupada    [30],
       ODMG_Classe_Grp_Inv    [30],
       ODMG_Classe_Grupamento [30];
}; ODMG_Grup_Struct Grupo;

//-----------------------------------------------------
//Estrutura que armazena informacoes sobre os Grupamentos

struct ODMG_Agreg_Struct
{char  ODMG_Nome_Agregacao    [30],
       ODMG_Classe_Componente [30],
       ODMG_Classe_Agregada   [30];
}; ODMG_Agreg_Struct Agregacao;
```

Classes declaradas no sistema de mapeamento

A classe `controi_tabela_OpenBASE`, utiliza os método `insere`, `esta_presente` e `remove` da classe `bree` para gerenciar os dados na estrutura *btree*. Os métodos dessa classe já foram explicados no Capitulo Mapeamento dos Modelos. Esses são os seguintes:

- `void montando_tabelas ();`
- `void construindo_esquema ();`
- `void atributos_classe ();`
- `void relacionamentos_classe ();`
- `void faz_relacionamentos ();`
- `void heranca_classe ();`
- `void keys_classe ();`
- `int palavra_reservada (char *st);`
- `int tipos_basicos (char *st);`

A classe `escreve_tabela_OpenBASE` escreve o esquema relacional e inclui os seguintes métodos com os seus respectivos parâmetros:

- `void EscreveCabecalho(char *st);`
- `void EscreveNomeTabela(char *st);`
- `void EscreveChavePrimaria(char *st);`
- `void EscreveTipoChavePrimaria(char *st);`
- `void EscreveNomeChaveComposta (int key);`
- `void EscreveTiposChaveComposta(char *st);`
- `void EscreveNomeChaveCompostaHeranca(char *st);`
- `void EscreveTipoChaveCompostaHeranca(char *st);`
- `void EscreveNomeAtributo(char *st);`
- `void EscreveTipo(char *st);`
- `void EscreveNomeRelacionamento(char *st);`
- `void EscreveNomeClasseRelacionada(char *st);`
- `void EscreveCardinalidade(char *st);`
- `void EscreveNomeGrupamento(char *st);`

- `void EscreveGrupamento(char *st);`
- `void EscreveNomeAgregacao(char *st);`
- `void EscreveNomeClasseComponente(char *st);`

A classe `palavra` deve incluir os seguintes métodos com os respectivos parâmetros: (a implementação é a mesma do mapeamento TOM para ODMG)

- Um construtor que aloca memória para um *buffer*. Este *buffer* é usado para armazenar todo ou parte do esquema TOM - *input*.
- `int abre_arquivo (char *filename)` - Este método retorna 1 se o nome do arquivo existir e for aberto com sucesso; caso contrário, retorna 0.
- `int pega_proxima_palavra (char *st)` - Este método retorna 1 se uma próxima palavra, delimitada por caracteres não alfabéticos e não numéricos, com exceção de "_" e "*", é encontrada no esquema TOM, caso contrário, ele retorna 0. Se uma próxima palavra é encontrada, ela é retornada na *string* `st`.

A classe `btree` deve incluir os seguintes métodos com os seus respectivos parâmetros: (a implementação é a mesma do mapeamento TOM para ODMG)

- Um construtor que inicia uma estrutura *btree*.
- `void insere (char *struc)` - Este método insere informações na estrutura.
- `void remove (char *struc)` - Este método remove informações da estrutura, pois cada relacionamento n-m gera duas relações, uma pode ser removida.
- `int esta_presente (char *struct)` - Este método verifica se a informação já está presente na estrutura.

4.3 Teste da Aplicação

Preparamos esta aplicação para um melhor entendimento de todo o processo de transformação dos objetos em tabelas do relacional. No teste utilizamos o esquema relacional gerado pelo mapeador do sistema **Registro de Carros**, que após ter sido gerado o esquema interno através do compilador DEFINE do OpenBASE, utilizamos o utilitário do OpenBASE TSGERAL para popular as tabelas. Utilizamos também a biblioteca de efeitos colaterais, que assim como o esquema, foram ambos gerados a partir dos esquema TOM. É importante enfatizar que dentro dessa biblioteca existem dois tipos de procedimentos, os quais são: procedimentos de cardinalidade e temporais. No banco de dados objeto-relacional estes procedimentos poderiam ser associados automaticamente às tabelas correspondentes. Por exemplo um procedimento de restrição de cardinalidade mínima e máxima, teria sua expressão na linguagem Opus da seguinte maneira.

```
$LIBRARY = carro.lib
PROG carro
operacao = "exclusao"
DO E_do_Modelo WITH codigo, operacao
```

Considerando que o procedimento é de inclusão, a rotina `E_do_Modelo` estabelece que a cardinalidade mínima é igual a um e a máxima igual a um. Se o procedimento fosse de inclusão o processo seria o mesmo, pois os valores mínimos e máximos estão registrados dentro de cada procedimento. Dois parâmetros `codigo` e `operacao` (valores válidos: inclusão e exclusão) são passados. O parâmetro código no caso da exclusão, é um valor existente, enquanto que o valor a ser incluído pode ou não existir.

Já os procedimentos temporais são associados às entidades com atributos de tempo e tem a seguinte expressão na linguagem Opus do OpenBASE.

```
DO carro_destruido WITH codigo
```

A associação do procedimento à entidade é feita de acordo com o nome da entidade, tanto para os procedimentos de restrição de cardinalidade como para os temporais, pois ambos, entidades e procedimentos, têm o mesmo nome. Apenas o parâmetro `codigo` é passado, o qual identifica o objeto temporal se esse existir, e verifica se o tempo de vida, está no intervalo de tempo especificado (`de` e `ate`). Caso não exista, atribui-se o tempo de vida deste objeto no momento da sua criação. Se não estiver nos limites do intervalo, o procedimento retorna uma mensagem informando que tempo de vida do objeto está prescrito. O procedimento `carro_destruido` foi gerado de acordo com a especificação da classe temporal `carro_destruido`do esquema TOM.

A aplicação funciona, porém algumas mudanças foram necessárias, devido às limitações que o compilador de esquemas `Define` do OpenBase impõe. Por exemplo, os nomes dos atributos podem ter no máximo oito caracteres. Outra restrição do `Define`, é não enxergar as entidades componentes que estão dispostas posteriormente. Estas têm que estar antes da entidade agregada para serem compiladas.

Outra limitação, são os tamanhos dos tipos que também precisam ser revistos após o mapeamento, pois o modelo de objetos não os define. Assumimos tamanhos *defaults* para os inteiros e para as *strings*, mas que as vezes não correspondem com a maioria dos atributos, obrigando um ajustamento pelo desenvolvedor. Essas questões para serem resolvidas, implicariam em algumas mudanças no núcleo do OpenBASE, e não estando disponível (fontes) não foi possível uma análise mais detalhada.

Todas as tabelas da aplicação foram geradas através do compilador de esquemas `Define`. A entrada dos dados, foram feita através do `Geral`, programa interativo do OpenBASE.

No teste, foram geradas oito tabelas do modelo da aplicação **Registro de Carros**, sendo quatro de relacionamentos e quatro temporais. Às tabelas estão associados os procedimentos contidos na biblioteca `carro.lib` cuja função é de restringir a cardinalidade e controlar o tempo de vida dos objetos. As tabelas são:

TABELAS TEMPORAIS

Tabelas		**Procedimentos de controle do tempo**
`carro_de_fabricante`	⇨	`carro_de_fabricante` Parameters with time and life_time 3 year
`carro_de_revendedor`	⇨	`carro_de_revendedor` Parameters with time and life_time 2 year
`carro_em_uso`	⇨	`carro_em_uso` Parameters with time and life_time 3 year
`carro_destruido`	⇨	`carro_destruido` Parameters with time and life_time 3 year

TABELAS DE RELACIONAMENTOS

Tabelas		Procedimentos que controla cardinalidade
e_do_modelo faz o relacionamento entre modelo e carro	⇨	e_do_modelo modelo: (0,*) carro: (1,1)
produz faz o relacionamento entre fabricante e modelo	⇨	produz fabricante: (1,*) modelo: (1,1)
e_da_empresa faz o relacionamento entre carro e revendedor	⇨	e_da_empresa revendedor: (0,*) carro de revendedor: (1,1)
car_uso_proprietario faz o relacionamento entre carro e proprietário	⇨	car_uso_proprietario pessoa: (1,*) carro em uso: (1,10)

A integridade semântica da aplicação é mantida através destes procedimentos. Conceitualmente estes procedimentos são conhecidos como efeitos colaterais, os quais, executam operações auxiliares para manter a integridade do banco de dados. As regras básicas de restrições de integridade do banco de dados, são executadas pelos axiomas dinâmicos. Importante salientar que nessa simulação, estamos supondo que o banco tem implementadas todas essas regras, do lado relacional (as mesmas regras utilizadas no modelo TOM). Entretanto o nosso objetivo foi mostrar que a transformação de objetos complexos para relacional estendido, considerando o ambiente ODMG, é possível.

5. Conclusão

A combinação das tecnologias relacional e objetos incentiva a coexistência dos dois diferentes paradigmas: relacional e objetos ([Ban96]). Bancos de dados relacionais estão firmes e ainda podem permanecer por muito tempo, pelos seguintes motivos: a existência das bases já instaladas, a importância dos dados dessas bases, muitos anos de desenvolvimento de sistemas e aplicações, e ainda a simples resistência às mudanças por todos que têm interesses na existência desses sistemas (desenvolvedores, profissionais com experiências e etc.).

Por outro lado, a tecnologia de objetos está expandindo e se desenvolvendo a passos rápidos: sistemas de banco de dados, linguagens, ferramentas de análises e desenvolvimento e a disponibilidade de bibliotecas de componentes reusáveis, estão conquistando rapidamente a confiança dos profissionais.

A facilidade da reusabilidade, tem sido muito comentada no ambiente orientado a objetos e com o advento dos modelos padrões de objetos, ganha mais poderes no sentido amplo do seu significado. O modelo padrão ODMG, por exemplo, que tem em sua proposta uma arquitetura comum de desenvolvimento orientado a objetos, incentiva a portabilidade de aplicações e consequentemente a reusabilidade. Padrões são vitais para obter uma interação corporativa de computadores. O mundo do futuro será um mundo de sistemas e padrões, onde arquiteturas fechadas e proprietárias darão lugar a arquiteturas projetadas para conectividades abertas, possibilitando portabilidade e acesso a bancos de dados fundamentados nos padrões existentes. Essa abordagem torna-se mais consistente com a arquitetura CORBA - *Common Object Request Broker Architecture* - ([COR98]).

Dessa forma, um sistema de objetos baseado no ambiente CORBA, possibilita ao cliente, solicitar serviços aos provedores (servidores) por meio de uma interface muito bem definida (interfaces são especificadas em IDL - *Interface Definition Language* - OMG). O cliente dependendo de sua localização na rede, usa o mecanismo RPC para estabelecer a comunicação desejada com outros sistemas (banco de dados).

Nesse contexto o modelo ODMG aparece como uma boa alternativa de padronização e além disso, a maioria dos SGBDOOs estão adotando este padrão. Com o mapeador (ODMG ⇒ OpenBASE) o gerenciador OpenBASE também passa a ter uma interface ODMG (Figura 6.1). Esperamos que com esse trabalho possamos dar uma maior

contribuição, no sentido de minimizar as diferenças entre os vários modelos existentes, tornando-os ferramentas com maior portabilidade.

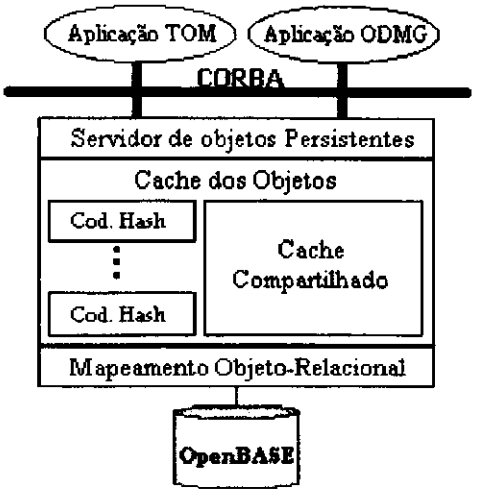


Figura 6.1 - Servidor de banco de dados OpenBASE

5.1 Trabalhos Futuros

- A integração dos dois modelos TOM e ODMG, estendendo uma linguagem *binding* com as semânticas do TOM, reduziria o trabalho de mapeamento. Ou seja, as classes pré-definidas do modelo TOM seriam implementadas nessa *binding* como uma extensão (biblioteca TOM) do modelo ODMG. Por exemplo uma `classe relacionamento` conteria restrições de cardinalidade mínima e máxima de cada relacionamento, ou uma `classe temporal` contendo as informações das classes temporais. Uma linguagem binding candidata poderia ser C++;

- Outro trabalho poderia ser feito no sentido de prover ao sistema, um gerenciamento dos objetos. Nesse sentido, uma interface totalmente orientada a objetos manipularia os objetos. Para tanto, esses objetos teriam que ser recompostos, haja visto, que estes estão persistentes em forma de tabela e somente gerenciados pelo OpenBASE;

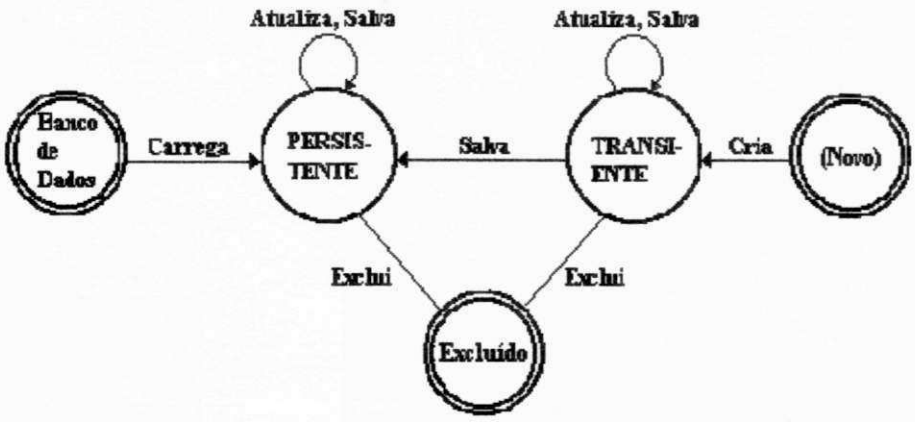


Figura 6.2 - Gerenciamento de objetos

A Figura 6.2 apresenta dois estados de objetos, transiente e persistente, onde transiente é o estado dos objetos que estão sendo criados e portanto estão ainda na memória volátil, ou foram carregados do banco de dados para a memória

- A integração desse sistema com a ferramenta gráfica de modelamento de objetos TOM (CASE/FADO em desenvolvimento), onde na interface dessa ferramenta, o usuário teria a opção de geração dos esquemas TOM ou ODMG.
- A integração desse sistema com o TOM Rules para prover monitoramento dos aspectos temporais do TOM/OpenBASE

Referências Bibliográficas

[Atk95] Atkinson, Malcolm & Morrison, Ronald. Orthogonally Persistent Object Systems. VLDB Journal, 4, 319-401 (1995).

[And93] André Felipe de Carvalho e Silva. TOM Rules - Um Monitor de Eventos, Regras e Gatilhos em um Ambiente Orientado a Objetos. Tese de mestrado, Universidade Federal da Paraíba - Coordenação de Pós-Graduação em Informática. Março de 1993.

[Ban96] François Bancilhon. Object, Relational, Object-Relational & Relational-Object. SIGS Publications, Inc, New York, NY, USA, 1996.

[Bla94] Michael Blaha, William Premerlani, and Hwa Sen Ge Corporate Research and Development. Converting OO Models into RDBMS Schema. IEEE Software V11, n3(may, 1994):28 (12 pages).

[Cat94] Cattell, R. G. G. Object Data Management Object-Oriented and Extended Relacional Database Systems. Addison Wesley Publishing, Inc. USA, 1994.

[Cat96] Cattell, R. G. G. Object Database Standard: ODMG 93. Morgan Kaufmann Publishers, Inc. San Francisco, California, 1996.

[Cat96a] Cattell, R. G. G. Object Database Standard: ODMG 93. - C++ Binding and Smalltalk Binding. Morgan Kaufmann Publishers, Inc. San Francisco, California, 1996. Chapter 5 and 6 - pages 87-164.

[Che76] Chen, P. The entity-relationship model: Toward a unified view of data. ACM Trans. Database System. 1, 1, Mar.,pages 9-36 - 1976.

[Cod70] Codd, E. F. A relational model of data for large shared data banks. Commun. ACM 13, 6 June pages 377-387 - 1970.

[Cod79] Codd, E. F. Extending the database relational model to capture more meaning. ACM Trans. Database System. 4, 4 Dec., pages 397-434 - 1979.

[COR98] CORBA - The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.

[COR98a] CORBA - The Common Object Request Broker: Architecture and Specification, Revision 2.2, pages 617-881. February 1998.

[Cun90] Cunha, Joseluce de Farias. Mapeamento do Modelo Semântico de Dados THM para o Modelo Relacional. Tese de mestrado, Universidade Federal da Paraíba - Coordenação de Pós-Graduação em Informática. Campina Grande, março de 1990.

[Dat91] Date, C. J.. Introdução a Sistemas de Bancos de Dados. Traduzido da 4. Edição Americana. Rio de Janeiro : Editora Campus, 1991. Capítulo 13. Álgebra Relacional.

[Dav92] David, M. Descrição Formal da Estrutura do Modelo Orientado a Objeto Temporal - TOM. Dissertação de mestrado, UFPB/COPIN. Campina Grande, 1992.

[Day88] U. Dayal & Al, "The HIPAC Project: Combining Active Databases and Timing Constraints", in SIGMOD Record, Vol. 17, n. 1, 1988.

[Fer95] Fernandes, Sônia Leila. ConTOM - Um Sistema de Consultas Gráficas a um banco de Dados Orientado a Objetos Temporal, Dissertação de Mestrado, Universidade Federal da Paraíba - Departamento de Sistemas e Computação, Campina Grande, 1995.

[Fer96] Fernandes, Sônia Leila & Schiel, Ulrich. O Modelo Temporal de Objetos - TOM. Relatório Técnico, Universidade Federal da Paraíba - Departamento de Sistemas e Computação. Abril, 1996.

[Fur93] Maria Elizabeth Sucupira Furtado e Ulrich Schiel. Uma Metodologia para Projeto de Banco de Dados Temporal Orientado a Objeto. Tese de mestrado, Universidade Federal da Paraíba - Departamento de Sistemas e Computação. Agosto de 1993.

[Ham91]Hammer, M., and McLeod, D. Database description with SDM: A semantic database model. ACM Trans. Database Syst. 6, 3 - pp. 351-386 - Sept., 1981.

[ISO96]ISO/IEC JTC1/SC21 N10489, ISSO//IEC 9075, Part 2, Committee Draft (CD), Database Language SQL - SQL/Foundation, July 1996.

[Kho90] Setrag Khoshafian and George P. Copeland. Readings In Object Oriented Database Sistems. Object Identity. Morgan Kaufmann Publishers, Inc. San Mateo, California, 1990, pp. 37.

[Kho94] Setrag Khoshafian. Banco de Dados Orientado a Objeto. Infobook, 1994.

[Kim88] Kim, W. et al. Integrating an object oriented programming system with a database system. In Proceedings of OOPSLA '88 Conference (Sept. 25-30, San Diego, Calif.). ACM/SIGPLAN, New York, 1988, pp.142-152.

[Leb93] F. Lebastard. DRIVER: A persistent virtual object layer for reasoning on relational databases. Ph. D.Thesis (in french), CERMICS-INRIA Sophia Antipolis (France), March 1993. 380 pages.

[Mai90] Maier, David & Jacob Stein. Development And Implementation of na Object-Oriented DBMS. Object-Oriented Database Systems. Morgan Kaufmann Publishers, Inc. San Mateo, California, 1990, pp. 167.

[Mar95]Martin, James & Odell, James J.. Análise e Projeto Orientados a Objeto. São Paulo : Makron Books, 1995.

[OMG93]OMG, Object Management Architecture Guide, R. G. Soley, Ed, Object Management Group, Framingham, Massachusetts, 1993.

[Pec88]Peckham, Joan & Maryanski, Fred.. Semantic Data Models. ACM Computing Surveys, Vol. 20, Nr. 3. September 1988.

[Row90] Row, Lawrence A & Stonebraker Michael R.. The POSTGRESS Data Model. Readings in Object Oriented Database Systems. Relational Extensions and Extensible Databases. Morgan Kaufmann Publishers, Inc. San Mateo, California, 1990, pp. 467.

[Ris92] Naphtali Rishe. Database Design: The Semantic Modeling Approach. McGraw Hill, 1992, 528 pp.

[Rum90] W.J. Premerlani, M.R. Blaha, J.E. Rumbaugh, and T.A. Varwing. An Object Oriented Relational Database. Communications Of the ACM, 33(11):99-109, November 1990.

[Rum91]Rumbaugh, M.R. Blaha, W. Premerlini, F. Eddy, and W. Lorensen. Object oriented Modeling and Design. (Chapter 17: Relational Databases), Prentice Hall, 1991.

[Sch83] Schiel, Ulrich. Na abstract introduction to the temporal-hierarchic data model (THM). In Proceedings of the 9 th International Conference on Very Large Data Base (Florence Italy). Very Large Database Endowment, Saratoga, Calif., pp 322-330. - 1983.

[Sch91] Schiel, Ulrich. An Open Environment for Objects with Time and Versioning, proc EastEurOOPe, Bratislava, 1991.

[Sch84] Schiel, Ulrich. A Semantic Database Model and ists Mapping to na Internal Relational Model in Databases: Role and Structure. P. Stocker, M. Atkinson and P. Gray (eds.), Cambridge University Press, 1984.

[Smi77] Smith, J. M., and Smith, D. C. P. Database abstractions: Aggregation and generalization. ACM Trans. Database Syst. 2, 2 (Mar.), pp 105-133. - 1977.

[TEC92] Tecnocoop Sistemas. Manual Projeto de Banco de Dados. INFOPRINT. 1. Ed. Set. 1992.

Web Sites Visitados

-www.odmg.org

-www.odi.com

- www.o2tech.com

- www.gemstone.com

- www.objy.com

- www.poet.com

- www.postgresql.org

- www.unisql.com

- www.persistence.com

- www.iis.com.br/tecnocoop/devtool.htm

Apêndice A

Linguagem TOM

Este apêndice descreve a linguagem, similar à linguagem utilizada no modelo, na definição do esquema de aplicação. Será especificada a sintaxe da linguagem em BNF.

```
class <nome classe>
        [specialization of <nome metaclasse>]
        [instance of <nome metaclasse>]
        [class relationships
                (<nome-rel>":"<nome classe>"("<card-min>","<card-max>")")*
                (pre-class":"<nome classe>[exclusive])*
                (post-class":"<nome classe>[exclusive])* ]
        [class methods
                <def. metodo>+ ]
        [instance relationships
                (<nome rel>":"<classe relacionada>"("<card-min>","<card-max>")"
                [with <n> (valid | transaction | bitemporal) values])+ ]
        [instance methods
                <def. metodo>+ ]
        ((keys are (<chaves>+ | inherited) |
        (type <tipo> [format <especificacao>])
        generalization-of <lista de classes> [by <nome papel>] [explicit] by
        predicate <predicado> | using (<relacionamento> as index [ parameters:
        [disjunctive,] [covering] ]
        grouping of <nome classe> (all | explicit | by predicate <predicado> | using
        <relacionamento>)
        [ parameters: [disjunctive,] [covering,] [ordered] ]
        aggregation [redefinition] of (<lista de classes> (all | explicit) |
        <classe1, classe2> by <relacionamento>) [exclusive]
        with (valid | transaction | bitemporal) time [and lifetime <constante> :
        <unidade tempo>] ]
```

Apêndice B

Linguagem ODL

No modelo ODMG, um esquema de banco de dados pode ser definido usando uma linguagem de definição de dados - ODL, a extensão direta da linguagem de definição de Inteface OMG - IDL, ou usando Smalltalk ou C++.

Neste apêndice será especificada a sintaxe e semântica do modelo ODL em BNF.

```
<specification>::=<definition>
        | <idenition><especification>
<definition>::=<type_dcl>;
        | <const_dcl>;
        | <except_dcl>;
        | <interface>;
        | <module>;
<module>::=module<identifier> {<especification>}
<interface> <interface_dcl>;
        | <forward_dcl>
<interface_dcl>::=<interface_header>
        [<persistence_dcl>] {[<interface_body]}
<persistence_dcl>::=persistent | transient
<forward_dcl>::=interface<identifier>
<interface_header>::=interface<identifier>
        [<inheritance_spec>]
        [<type_property_list]
<type_property_list>]
        ::=([<extent_spec>][<key_spec>])
<extent_spec>::=extent<string>
<key_spec>::=key[s] <key_list>
<key_list>::=<key> | <key>, <key_list>
<key>::=<property_names> | (<property_list>)
<property_list>::=<property_name>
        |<property_name>, <property_list>
<property_name>::=<identifier>
<interface_body::=
        <export> | <export> <interface_body>
<export>::=<type_dcl>;
        | <const_dcl>;
        | <except_dcl>;
        | <attr_dcl>;
        | <rel_dcl>;
        | <op_dcl>;
<inheritance_spec>::=
        :<scoped_name>[,<inheritance_spec>]
<scoped_name>::=<identifier>
        |::<identifier>
```

```
        |<scoped_name>::<identifier>
<const_dcl>::=const<const_type><identifier>=
        <const_exp>
<const_type>::=<integer_type>
        | <char_type>
        | <boolean_type>
        | <floating_pt_type>
        | <string_type>
        | <scoped_name>
<const_exp::=<or_expr>
<or_expr>::=<xor_expr>
        |<or_expr> | <xor_expr>
<xor_expr>::=<and_expr>
        |<xor_expr>^<and_expr>
<and_expr>::=<shift_expr>
        | <and_expr> & <shift_exp>
<shift_expr>::=<add_expr>
        | <shift_expr> >> <add_exp>
        | <shift_expr> << <add_exp>
<add_expr>::=<mult_expr>
        | <add_expr> + <mult_expr>
        | <add_expr> - <mult_expr>
<mult_expr>::=<unary_expr>
        | <mult_expr> * <unary_expr>
        | <mult_expr> / <unary_expr>
        | <mult_expr> % <unary_expr>
<unary_expr>::=<unary_operator><primary_expr>
        |<primary_expr>
<unary_operator>::= -
        | +
        | ~
<primary_expr>::=<scoped_name>
        | <literal>
        | (<const_exp>)
<literal>::=<integer_literal>
        | <string_literal>
        | <character_literal>
        | <floating_pt_literal>
        | <boolean_literal>
<boolean_literal>::=TRUE
        | FALSE
<positive_int_const>::=<const_exp>
<type_dcl::=typedef<type_declarator>
        | <struct_type>
        | <union_type>
        | <enum_type>
<type_declarator>::=<type_spec><declarators>
<type_spec>::=<simple_type_spec>
```

```
        | <constr_type_spec>
<simple_type_spec>::=<base_type_spec>
        | <template_type_spec>
        | <scoped_name>
<base_type_spec>::=<floating_pt_type>
        | <integer_type>
        | <char_type>
        | <boolean_type>
        | <octet_type>
        | <any_type>
<template_type_spec>::=<array_type>
        | <string_type>
        | <coll_type>
<coll_type>::=<coll_spec> < <simple_type_spec> >
<coll_spec> ::= Set | List | Bag
<constr_type_spec>::=<struct_type>
        | <union_type>
        | <enum_type>
<declarators>::=<declarator>
        | <complex_declarator>
<declarator>::=<simple_declarator>
        |<complex_declarator>
<simple_declarator::=<identifier>
<complex_declarator>::=<array_declarator>
<floating_pt_type>::=Float
        | Double
<integer_type>::=<signed_int>
        | <unsigned_int>
<signed_int>::=<signed_long_int>
        | <signed_short_int>
<signed_long_int>::= Long
<signed_short_int>::= Short
<unsigned_int>::=<unsigned_long_int>
        | <unsigned_short_int>
<unsigned_long_int>::=Unsigned Long
<unsigned_short_int>::= Unsigned Short
<char_type>::=Char
<boolean_type::=Boolean
<octet_type>::=Octet
<any_type>::= Any
<struct_type>::=Struct<identifier> {<member_list>}
<member_list>::=<member> | <member>
        <member_list>
<member>::=<type_spec><declarators>;
<union_type>::=union<identifier>switch
        (<switch_type_spec>){<switch_body>}
<switch_type_spec>::=<integer_type>
        | <char_type>
```

```
        | <boolean_type>
        | <enum_type>
        | <scoped_name>
<swithc_body>::=<case> | <case> <switch_body>
<case>::=<case_label_list><element_spec>;
<case_label>::=<case_label>
        |<case_label><case_label_list>
<case_label>::=case<const_exp>:
        | default:
<element_spec>::= <type_spec> <declarator>
<enum_type::=Enum<identifier> {<enumerator_list>}
<enumerator_list>::=<enumerator>
        | <enumerator>, <enumerator_list>
<enumerator>::=<identifier>
<array_type>::=<array_spec> < <simple_type_spec>,
        <positive_int_const> >
        | <array_type>::=<array_spec> < <simple_type_spec>,
<array_spec>::= Array | Sequence
<string_type>::=String < <positive_int_const> >
        | String
<array_declarator>::= <identifier> <array_size_list>
<array_size_list>::= <fixed_array_size>
        | <fixed_array_size> <array_size_list>
<fixed_array_size::= [<positive_int_const>]
<attr_dcl>::=[readonly] attribute
        <domain_type><attribute_name>
        [<fixed_array_size>]
<domain_type>::=<simple_type_spec>
        | <struct_type>
        | <enum_type>
<rel_dcl>:=relationship
        <target_of_path>
        <identifier>
        inverse <inverse_traversal_path>
        [ { order_by <attribute_list> } ]
<target_of_path>::= <identifier>
        | <rel_collection_type> < <identifier> >
<inverse_traversal_path>::=
        <identifier> ::<identifier>
<attribute_list>::=<scoped_name>
        | <scoped_name>, <attribute_list>
<rel_collection_type::= Set | List | Bag | Array
<except_dcl>::= exception <identifier>
        {[<member_list>]}
op_dcl>::= [<op_attribute>] <op_type_spec>
        <identifier> <parameter_dcls>
        [<raises_expr>] [<constext_expr>]
<op_attribute>::=oneway
```

```
<op_type_spec>::=<simple_type_spec>
        | void
<parameter_dcls>::= ([<param_dcl_list>])
<param_dcl_list>::= <param_dcl>
        | <param_dcl>, <param_dcl_list>
<param_dcl<::= <param_attribute> <simple_type_spec>
        <declarator>
<param_attribute>::= in
        | out
        | inout
<raises_expr>::=raises(scoped_name_list>)
<scoped_name_list>::=<scoped_name>
        | <scoped_name>,
        <scoped_name_list>
<context_expr>::=context(<string_literal_list>)
<string_literal_list>::=<string_literal>
        | <string_literal>, <string_literal_list>
```

Apêndice - C

Linguagem DEFINE do OpenBASE

Este apêndice mostra, toda sintaxe da linguagem de definição dos dados do OpenBASE.

```
[<< <comentario> >>]

[$CONTROLE <opтпo>, ..., <opтпo>]

BANCO [<percurso>] <nome_bd> <cѕdigo_de_seguranta>
        ...[{ARQRECUP/DIARIO/DIAREC}]...
        ...[{BLOQARQ/BLOQCHA/BLOQPAG/BLOQRE}]...
        ...[ESQUEMA=[<percurso_bd_origem>]<nome_bd_origem>...
        ...<codigo_de_seguranta_bd_origem>...
        ...[<palavra_de_nivel_bd_origem>]]
        [NIVEIS:<n·mero_nivel> <palavra_nivel>
                <n·mero_nivel> <palavra_nivel>]
        [RELACOES:]
                NOME:[<percurso>] <nome_arquivo> <tipo_arquivo>
                        ...[ESQUEMA=[<percurso_bd_origem>] <nome_bd_origem>...
                        ...<codigo_de_seguranta_bd_origem>...
                        ...[<palavra_de_nivel_bd_origem>]]
        [REGISTROS:]
                <nome_item> [{[<rep>]/(<ligatЈes>)/(<caminho>)/(0)}]...
                ...<tipo>:<tamanho>[,<num_decimais>]...
                ...[(<num.nivel_ler>,<num_nivel_grav)]
                ...[{POS<nome_item_rd>[+<deslocamento>]/...
                ...VIRTUAL(<nome_item_part>, ..., <nome_item_part>)}]...
                ...[UNICA]
```