DEPARTAMENTO DE

ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Estudo Prévio para a Construção de um SGBDO em NeXTSTEP

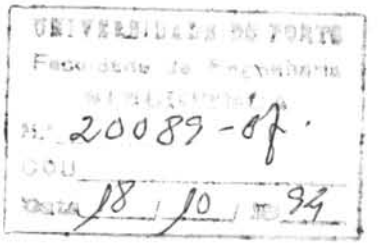
Ademar Manuel Teixeira de Aguiar

FACULDADE DE ENGENHARIA

UNIVERSIDADE DO PORTO

Rua dos Bragas, 4099 Porto Codex – PORTUGAL

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
Departamento de Engenharia Electrotécnica e de Computadores


UNIVERSIDADE DO PORTO
Faculdade de Engenharia
20089-87
18 / 10 / 94

Estudo Prévio para a Construção de um SGBDO em NeXTSTEP

Ademar Manuel Teixeira de Aguiar

Licenciado em Engenharia Electrotécnica e de Computadores
pela Faculdade de Engenharia da Universidade do Porto

Dissertação submetida para satisfação
parcial dos requisitos do programa do curso de

Mestrado em Engenharia Electrotécnica e de Computadores
(Área de especialização em Sistemas)

043 M
A227e
ex. 2

PORTO, Abril 1994

Dissertação realizada sob a supervisão do

Doutor João Bernardo de Sena Esteves Falcão e Cunha

Professor Auxiliar do Departamento de Engenharia Mecânica
da Faculdade de Engenharia da Universidade do Porto

Dissertação realizada por

Ademar Manuel Teixeira de Aguiar

INESC Porto telef: 351-2-2094000 fax:351-2-318692 email:aaguiar@rubens.inescn.pt

Resumo

À medida que a aplicação dos sistemas de computadores se vai alargando a uma cada vez maior variedade de situações, é de esperar que alguns dos seus requisitos representem novos desafios para a tecnologia de bases de dados actualmente disponível. De facto, os Sistemas de Gestão de Bases de Dados (SGBD) tradicionais provaram ser inadequados para diversos tipos de aplicações de utilização cada vez mais corrente, das quais se destacam: as ferramentas CASE, as aplicações médicas e científicas, as aplicações de hipertexto, e as aplicações multimedia, entre outras.

Com o objectivo de enfrentar estes novos desafios, têm sido desenvolvidas novas metodologias de análise e concepção, novas linguagens de programação, e novos sistemas de gestão de bases de dados. Os novos SGBD's, para além da funcionalidade oferecida pelos seus antecessores—os Sistemas de Gestão de Bases de Dados Relacionais, oferecem novas capacidades, como por exemplo: modelo de dados orientado por objectos, encapsulamento, herança, tipos de dados complexos, versões e configurações, e transacções de longa duração.

O presente trabalho sintetiza as características apresentadas por estes novos sistemas, numa perspectiva de construção de um Sistema de Gestão de Bases de Dados de Objectos (SGBDO) para uma ferramenta CASE em desenvolvimento no INESC—Grupo TEIA.

Como um primeiro passo para a futura construção do SGBDO, é delineada neste trabalho uma sua interface com a linguagem Objective-C. Esta interface tem como base o modelo objecto ODMG e as linguagens ODL, OML e OQL que fazem parte da recente proposta de norma para SGBDO's, a ODMG-93.

Abstract

As computer systems are being used in a wider variety of situations, some of the needs of the resulting applications clearly represent new challenges for the currently available database tecnhnology. As a matter of fact, traditional Database Management Systems (DBMS) have proven to be inadequate for several kinds of increasingly spread applications. These applications include: CASE tools, scientific and medical, hypertext, and multimedia, just to mention a few.

In order to face these new challenges, new analysis and design methodologies, programming languages and database management systems are emerging. These new DBMSs provide data management capabilities like their predecessors—the relational DBMSs, and also provide new features such as: an object model, encapsulation, inheritance, complex data types, versions and configurations, and long transactions.

This dissertation surveys the main features and capabilities offered by these new database systems, having in mind the implementation of a new Object Database Management System (ODBMS) for one CASE tool currently being implemented by the TEIA Project at INESC.

As a first step towards the implementation of this new ODBMS, is outlined in this work an interface for the Objective-C language. This interface will be based on the ODMG object model and ODL, OML and OQL languages of the recently published proposal of standard for ODBMSs, the ODMG-93.

Palavras Chave

Gestão de Bases de Dados
Gestão de Bases de Dados de Objectos
Sistemas de Gestão de Bases de Dados de Objectos
Sistemas de Gestão de Bases de Dados
Linguagens de Programação
Orientação por Objectos
Modelos de Dados

Keywords

Data Management
Object Data Management
Object Database Management Systems
Database Management Systems
Programming Languages
Object Orientation
Data Models

Agradecimentos

Ao meu orientador, Professor João Falcão e Cunha, pela disponibilidade sempre demonstrada em colaborar, pela oportunidade dos comentários e críticas a este trabalho, e especialmente pelas excelentes condições de trabalho disponibilizadas para a sua realização.

Aos colegas da equipa do Projecto PDI, Professor João Falcão e Cunha e Eng.José Paulo Espinha, por terem libertado o autor de algumas das suas obrigações nas fases mais cruciais deste trabalho.

Ao Professor Raul Moreira Vidal por todo o apoio e motivação dada ao longo do curso de Mestrado.

Aos colegas do grupo de trabalho do INESC, em especial ao Mário Santos pelas conversas e ideias trocadas sobre o projecto mais global em que se insere este trabalho, e ao José Maria Bonnet e Jorge Simões, pela franca colaboração prestada na revisão de versões preliminares deste trabalho.

Aos colegas José Manuel Moreira, João Vasco Ranito, Luís Neves, João Neto, João Neta, Valter Pinho, João Pascoal Faria e José Magalhães Cruz, pela motivação que de uma forma ou de outra deram para a concretização deste trabalho.

Aos familiares, por todo o apoio prestado, e pela compreensão e paciência demonstrada especialmente na fase conclusiva do trabalho.

O trabalho realizado só foi possível graças à colaboração conjunta da JNICT—Junta Nacional de Investigação Científica e Tecnológica, que concedeu ao autor uma Bolsa de Mestrado (BM/196/90-IA) e do INESC—Instituto de Engenharia de Sistemas e Computadores, que forneceu todo o apoio logístico necessário.

A todos o meu obrigado.

Ademar Manuel Teixeira de Aguiar

à *Kity*

Índice

1 A Necessidade de Novos Sistemas de Bases de Dados 1

1.1 Sistemas de Bases de Dados Actuais 2
1.2 Novos Desafios 2
1.3 Novos Sistemas de Bases de Dados 3
1.4 Componentes dum Sistema de Software 4
1.4.1 Interfaces com o Utilizador 5
1.4.2 Linguagens de Programação 6
1.4.3 Sistemas de Bases de Dados 6
1.4.4 Conceito Unificador: Objectos 7
1.5 Proposta de Norma ODMG-93 7
1.5.1 Importância de uma Norma 8
1.5.2 Objectivos do ODMG-93 8
1.5.3 Componentes do ODMG-93 9
1.6 Motivação para o Presente Trabalho 10
1.6.1 Objectivos 13

1.6.2 Estrutura da Dissertação 13

2 Contexto **15**
2.1 Novas Aplicações 16
2.1.1 Engenharia de Software 16
2.1.2 Engenharia Mecânica, Civil e Electrotécnica 20
2.1.3 Documentos e Aplicações de Escritório Electrónico 25
2.1.4 Outras Aplicações 25
2.1.5 Requisitos em Termos de SGBDOs e SGBDs Tradicionais 28
2.2 Novos Modelos de Dados 31
2.2.1 Revisão do Modelo Relacional 31
2.2.2 Extensões do Modelo Relacional 31
2.2.3 Modelos Orientados por Objectos 32
2.2.4 Modelos Funcionais e Semânticos 32
2.2.5 Modelos de Dados e Arquitecturas de Bases de Dados 33
2.3 Linguagens de Programação Orientada por Objectos 34
2.3.1 Evolução do Modelo Objecto 34
2.3.2 Elementos do Modelo Objecto 37
2.3.3 Benefícios do Modelo Objecto 44

3 A Nova Tecnologia de Bases de Dados **45**
3.1 Sistemas de Gestão de Bases de Dados Tradicionais 46
3.1.1 Perspectiva Histórica 46
3.1.2 Conceitos de Bases de Dados 48
3.1.3 Bases de Dados Relacionais 50
3.1.4 Linguagens de Interrogação (LDD, LMD, Vistas) 55
3.1.5 Concorrência e Recuperação de Erros 57
3.1.6 Bases de Dados Distribuídas 59
3.1.7 Ferramentas de Apoio 60
3.1.8 Desempenho 61
3.2 Conceitos de Bases de Dados de Objectos 63
3.2.1 Arquitecturas duma Base de Dados de Objectos 63
3.2.2 Objectos 66
3.2.3 Relações entre Objectos 69
3.2.4 Objectos Compostos 72
3.2.5 Procedimentos 73
3.2.6 Tipos e Herança 74
3.2.7 Linguagens de Interrogação e de Programação 79
3.2.8 Persistência, Concorrência e Recuperação de Erros 84
3.3 Características Gerais de um SGBDO 88
3.3.1 Funcionalidade de Base de Dados 88
3.3.2 Objectos e Propriedades 89
3.3.3 Sistema de Tipos 90
3.3.4 Linguagens de Programação e Interrogação 92
3.3.5 Arquitectura de Bases de Dados 98
3.3.6 Outras Características 99
3.3.7 Características Essenciais para o SGBDO da MOOT 100

4 Uma Interface para um SGBDO em NeXTSTEP **101**
4.1 Proposta de Norma ODMG-93 102
4.1.1 Objectivos do ODMG-93 102
4.1.2 Componentes do ODMG-93 103
4.1.3 Modelo Objecto ODMG 103
4.1.4 Linguagem de Definição—ODL 109
4.1.5 Linguagem de Interrogação—OQL 110
4.1.6 Mapeamento para as Linguagens de Programação 115
4.2 Interface em Objective-C para um SGBDO 117
4.2.1 Abordagens Possíveis 117
4.2.2 Princípios Orientadores 118
4.2.3 Enquadramento no Ambiente NeXTSTEP 119
4.2.4 Componentes da Interface 119
4.2.5 Mapeamento do Modelo Objecto ODMG para Objective-C 120
4.2.6 Mapeamento das linguagens ODL, OML e OQL 122
4.2.7 Hierarquia de Tipos Base do Modelo 125
4.2.8 Classes *Database* e *Transaction* 127

5 Conclusões **132**
5.1 Contributo Principal 133
5.2 Resultados 133
5.3 Limitações 133
5.4 Questões em Aberto 134
5.5 Futuros Desenvolvimentos 134

F

Lista de Figuras

1.1 A evolução dos sistemas de software 5
1.2 As três gerações dos sistemas de bases de dados 6
1.3 A investigação na ciência da computação 8
1.4 Duas arquitecturas de sistemas de bases de dados 10
1.5 MOOT—recurso a ficheiros para armazenamento dos projectos 12
1.6 MOOT—recurso a uma base de dados para armazenamento dos projectos . 14

2.1 Fases típicas do ciclo de vida do software 18
2.2 Fases típicas do ciclo de vida de projectos eléctricos e mecânicos 22
2.3 A visão interna e externa de uma classe 39

3.1 Modelo de dados hierárquico 46
3.2 Modelo de dados reticulado 47
3.3 Base de dados de documentos. 51
3.4 Tabelas de documentos e capítulos 52
3.5 As arquitecturas de sistemas de bases de dados de objectos 64

3.6 Relações binárias num modelo de dados orientado por objectos 70

4.1 Hierarquia de tipos do Modelo Objecto ODMG 105
4.2 Base de dados de documentos. 111
4.3 Passos para a criação de uma aplicação para um SGBDO 129
4.4 Uma abordagem simples para a interface a definir 130
4.5 As várias ferramentas de desenvolvimento disponíveis em NeXTSTEP . . . 131

1

A Necessidade de Novos Sistemas de Bases de Dados

A investigação na área dos sistemas de bases de dados tem-se revelado, ao longo da sua existência, uma actividade excepcionalmente produtiva e de grande impacto a nível sócio-económico. Desde que considerada uma área de investigação de base decorreram cerca de 20 anos, ao longo dos quais tem alimentado e impulsionado uma indústria de serviços de informação que, segundo estimativas, movimenta actualmente nos Estados Unidos da América cerca de 10 biliões de dólares por ano (aprox. 1500 milhões de contos).

Os resultados obtidos pela investigação na área dos sistemas de bases de dados tem contribuído também para o progresso de outras áreas científicas, da biologia à computação, bem como proporcionado a evolução de diversos sistemas, dos quais se destacam os sistemas de comunicações, os sistemas de gestão financeira, os sistemas de transporte e logística, e entre outros, os sistemas de divulgação e acesso a literatura científica.

A comunidade de investigação em sistemas de bases de dados existe desde finais da década de 60. No começo, modestamente representada quase exclusivamente em

laboratórios industriais, expandiu-se enormemente, durante as duas últimas décadas, por universidades, institutos, e diversas instituições de investigação e desenvolvimento.

No início, as principais questões de investigação relacionavam-se com problemas de gestão de dados em aplicações para fins empresariais, tais como: automatização de operações bancárias, armazenamento de registos de informação e gestão de sistemas de reservas. Embora meramente exemplificativas, estas aplicações reúnem quatro requisitos típicos dos sistemas de bases de dados, a lembrar:

- *Eficiência* no acesso e modificação de muito grandes volumes de dados;
- *Robustez*, ou capacidade de os dados sobreviverem a falhas de hardware e software, sem daí resultarem perdas ou inconsistência;
- *Controlo de Acesso*, para assegurar que o acesso simultâneo aos dados por vários utilizadores, possa ser feito de forma consistente e de acordo com as permissões pré-definidas;
- *Persistência*, ou a manutenção dos dados por longos períodos de tempo, independentemente das aplicações que acedem aos dados.

O estudo de métodos para o projecto e desenvolvimento de sistemas com as características acima, conjuntamente com a criação de linguagens e ferramentas conceptuais para auxiliar os utilizadores no acesso, manipulação e projecto de bases de dados têm sido os objectivos centrais da investigação em sistemas de bases de dados.

1.1 Sistemas de Bases de Dados Actuais

Desde o seu aparecimento, os Sistemas de Gestão de Bases de Dados (SGBDs), têm tido uma enorme divulgação, sendo hoje praticamente utilizados em quase todos os ambientes computacionais, para organizar, criar, e manter importantes volumes de informação.

Os primeiros SGBDs surgem no princípio da década de 70. As abordagens então mais populares tinham como base os modelos de dados hierárquico e reticulado [1].

Após a publicação por Codd em 1970 de uma série de artigos sobre o modelo relacional, aparecem no mercado, por volta dos anos 80, versões comerciais de Sistemas de Gestão de Bases de Dados Relacionais (SGBDRs). Os SGBDRs revolucionaram a indústria de software ao conseguirem oferecer enormes capacidades de gestão de dados, apenas com base em poucos e simples conceitos. Desde então a sua popularidade aumentou consideravelmente, em parte devido à simplicidade e normalização que oferecem. Os SGBDRs hoje lideram o mercado de SGBDs.

1.2 Novos Desafios

Apesar da excelente evolução registada nos sistemas de bases de dados, são já hoje muitas as aplicações com fortes requisitos em termos de gestão de informação não satisfeitas com a actual tecnologia, desafiando por conseguinte o estado actual de conhecimentos. Nestas

[1] Modelo de dados: abstracção matemática de dados e de operações sobre dados

aplicações incluem-se aplicações de CASE [2], de CAD/CAM [3], aplicações médicas e científicas, aplicações para representação gráfica, aplicações de escritório electrónico, hipertexto, represen tação de conhecimento para aplicações de inteligência artificial, novas aplicações empresariais, sistemas de informação geográfica, sistemas de informação urbana, etc.

As aplicações da próxima geração de bases de dados, segundo se prevê [SSU91] [Cat91] pouco terão em comum com as aplicações actuais. As futuras aplicações terão novos requisitos, tais como por exemplo, necessidades de: manipulação de muito maiores quantidades de informação; armazenamento de objectos de grande dimensão e grande complexidade; arquivo de imagens; arquivo de grandes sequências de dados; armazenamento de instruções de programas; fácil acesso e pesquisa a bases de dados históricas de grandes dimensões.

Adicionalmente, a tendência hoje existente de aumento de cooperação entre diferentes organizações para a resolução de problemas comuns (científicos, de engenharia, empresariais) terá implicações ao nível dos sistemas de bases de dados, levando ao aparecimento de bases de dados de grande dimensão, distribuídas, e possivelmente heterogéneas, com os consequentes problemas em termos de inconsistência, segurança, e distribuição em larga escala .

Apesar da aparente maturidade da tecnologia de bases de dados, é hoje cada vez maior a consciência e preocupação, de que até ao momento presente apenas tenham sido aflorados os princípios e técnicas requiridos pela próxima geração de aplicações de gestão de informação, as quais se prevê virão revolucionar as economias industrializadas do princípio do próximo século.

Claramente, os SGBDRs centralizados ou distribuídos, encontram-se hoje muito bem estudados e largamente difundidos, pelo que a sua evolução não parece ser fácil. Novas ideias têm no entanto surgido oriundas da filosofia de programação orientada por objectos, cujo domínio tem nos últimos anos merecido a dedicação de muitos investigadores.

1.3 Novos Sistemas de Bases de Dados

Actualmente, e resultante de quase uma década de investigação em novas direcções, assistimos no momento ao aparecimento de uma Nova Geração de Sistemas de Bases de Dados.

Os novos sistemas oferecem capacidades de gestão de informação idênticas às dos seus predecessores, e adicionalmente oferecem novas capacidades, na sua maioria, inspiradas na programação orientada por objectos. Alguns exemplos destas novas capacidades são: definição de novos tipos por extensão de outros (herança), encapsulamento de informação, gestão de objectos complexos e de grande dimensão, capacidade de armazenamento de informação multimedia, gestão de regras, gestão de procedimentos, gestão de arquivos, gestão de versões e configurações, gestão de transacções de longa duração.

Os novos SGBD surgem com o objectivo de alargar a aplicabilidade da tecnologia de bases de dados a novos tipos de aplicações e a aplicações tradicionais não satisfeitas com os actuais SGBDRs.

Estes novos SGBDs distinguem-se pelo modelo de dados que suportam e pela sua arquitectura, isto é, pela forma como o SGBD é implementado e integrado com outros componentes do sistema.

---

[2] *Computer Aided Software Engineering*

[3] *Computer Aided Design and Manufacturing*

Os diferentes modelos de dados que podemos encontrar nestes novos SGBDs são, paradoxalmente, muito similares em termos das capacidades que oferecem. Os modelos de dados mais representativos são os seguintes:

- os modelos relacionais estendidos;
- os modelos funcionais;
- e os modelos orientados por objectos.

Atendendo à similaridade dos modelos de dados, as diferenças mais importantes encontradas nos novos SGBDs relacionam-se com a arquitectura adoptada. As quatro arquitecturas mais importantes são:

- *Sistemas de Bases de Dados Estendidos:* representam os sistemas que, tendo como base sistemas tradicionais, oferecem nova funcionalidade através de linguagens de interrogação novas ou estendidas;
- *Linguagens de Programação para Bases de Dados:* representam os sistemas que estendem as linguagens de programação, no sentido de as dotar com capacidades típicas de um sistema de bases de dados: persistência, controlo de concorrência, e outras.
- *Gestores de Objectos:* representam produtos que podem ser vistos como extensões de sistemas de ficheiros existentes ou sistemas de memória virtual. Um gestor de objectos possui normalmente um modelo de dados limitado: constituem repositórios para objectos persistentes.
- *Geradores de Sistemas de Bases de Dados:* representam sistemas que permitem a construção de SGBDs para fins muito específicos. Com estes sistemas, o implementador tem possibilidade de definir o modelo de dados e as capacidades a incluir no sistema final.

A transição a que actualmente se assiste, de Sistemas de Gestão de Bases de Dados Relacionais para Sistemas de Gestão de Bases de Dados de Objectos (SGBDO), não constitui em si nada de revolucionário, mas apenas segue as tendências de evolução existentes nos sistemas de software, aliás tal como já se verificou nas outras duas componentes principais dos sistemas de software, concretamente as interfaces com o utilizador e as linguagens de programação. O aparecimento dos SGBDOs não constitui um acontecimento isolado, mas algo perfeitamente integrado num processo mais global de mudança que o software está a sofrer, e durante o qual provavelmente serão definidos os seus contornos para os próximos 10 a 20 anos.

1.4 Componentes dum Sistema de Software

Os sistemas de software possuem três componentes principais:

- as interfaces com o utilizador,
- as linguagens de programação,
- e os sistemas de gestão de dados.

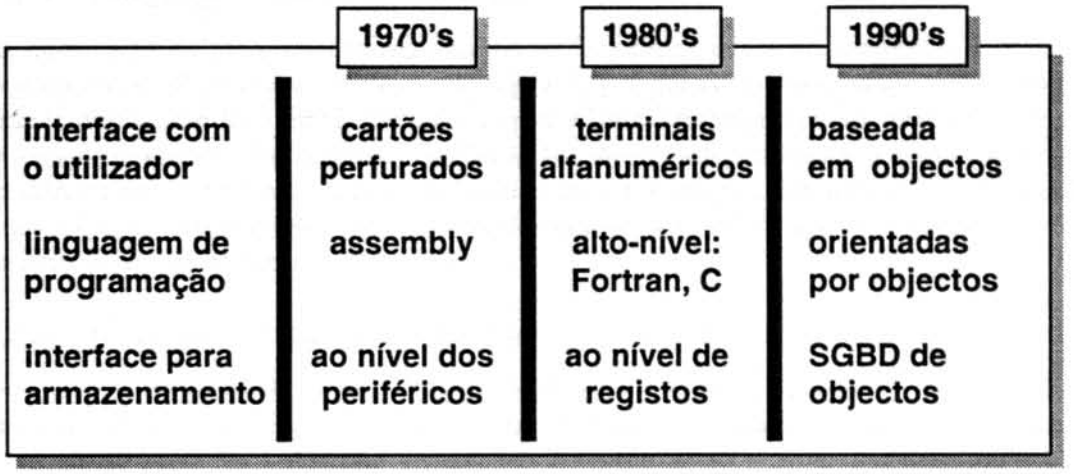


Figura 1.1: A evolução dos sistemas de software

Na figura 1.1 caracterizam-se as três transições mais importantes ocorridas sobre os sistemas de software.

Na década de 60 e princípio da década de 70, a linguagem de programação utilizada reduzia-se à linguagem *assembly*, a interface com o utilizador realizava-se através de cartões perfurados, e a interface para armazenamento dos dados recorria a instruções de muito baixo nível, mais concretamente, a comandos específicos dos periféricos de armazenamento, discos e bandas magnéticas.

No ínicio da década de 80, a interface com o utilizador evolui para a utilização de terminais alfanuméricos, as linguagens de programação de alto-nível são largamente divulgadas, e os sistemas de armazenamento passaram a permitir que o acesso aos dados fosse realizado a um mais alto nível, ao nível de registos lógicos. As razões que levaram a esta evolução foram puramente económicas. O custo de programação de uma aplicação com recurso a uma linguagem de alto-nível, comparativamente com uma linguagem *assembly*, conseguiu ser reduzido para cerca de um terço do valor. Com as actuais técnicas de programação orientadas por objectos, estima-se conseguir obter em média idênticas reduções de custos.

1.4.1 Interfaces com o Utilizador

Os primeiros conceitos de orientação por objectos surgem na área das interfaces com o utilizador, com o Xerox Star e o Macintosh da Apple, a oferecerem interfaces baseadas em écrans bitmap, icons, menus, e uma filosofia de interacção com o écran através de um dispositivo hoje essencial em qualquer computador, o rato. As ideias base desta filosofia, vêm a alcançar enorme popularidade no mercado dos PCs com o posterior aparecimento do sistema Windows da Microsoft.

1.4.2 Linguagens de Programação

As linguagens de programação seguiram-se às interfaces, aderindo também à filosofia de orientação por objectos. A primeira linguagem de programação orientada por objectos com grande divulgação foi o Smalltalk, para a qual a Xerox PARC criou em 1980 um ambiente de desenvolvimento denominado Smalltalk-80. Mais tarde, começam a surgir as versões orientadas por objectos de linguagens tradicionais, das quais, a linguagem C++ se destaca como exemplo de maior sucesso, sucesso este em parte devido à grande popularidade que merecia a sua antecessora, a linguagem C.

1.4.3 Sistemas de Bases de Dados

Os sistemas de bases de dados, os últimos do trio a aderir à filosofia de orientação por objectos, encontram-se actualmente ainda em fase de evolução. Tradicionalmente, os sistemas de bases de dados têm necessitado de cerca de 10 anos de gestação em fase de investigação até ao aparecimento da primeira versão comercial, e cerca de outros 8 a 10 anos para se conseguirem impôr comercialmente. A figura 1.2 ilustra este ciclo para os sistemas relacionais e sistemas de objectos [Atw93, Str93].

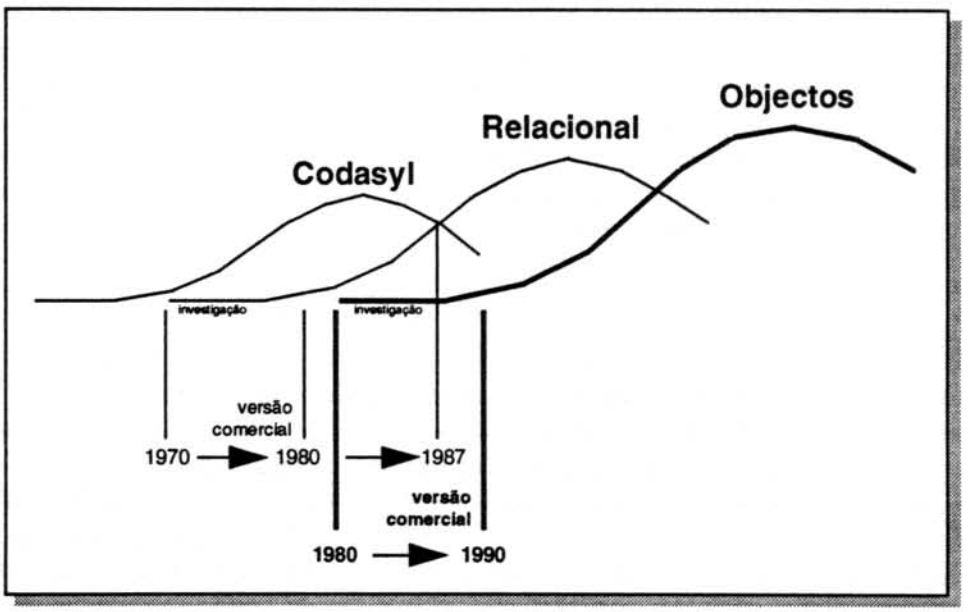


Figura 1.2: As três gerações dos sistemas de bases de dados

Após Ted Codd ter publicado em 1970 os primeiros artigos sobre o modelo relacional, decorreram ainda cerca de 6 a 8 anos até que os primeiros produtos comerciais aparecessem. Foi por volta de 1980, que as firmas líderes em sistemas relacionais—Oracle e Ingres, entraram no mercado. No ano de 1987, as vendas de sistemas relacionais corresponderam a 70% das vendas totais.

Os SGBDOs parecem seguir uma cronologia muito similar. De facto, toda a curva pode

ser deslocada no tempo cerca de 10 anos. O início dos trabalhos de investigação pode-se datar de 1980, com uma tese em modelos de dados semânticos, no MIT. Os primeiros produtos comerciais (Servio e Ontos) aparecem no mercado em 1986-87, e passados três anos, em 1990, aparecem as três firmas que actualmente lideram o mercado—Object Design, Versant e Objectivity. A manter-se o paralelismo, serão necessários 7 anos a contar de 1990, para que 70% dos novos SGBDs vendidos sejam orientados por objectos. Para que tal penetração de mercado seja alcançada, torna-se necessário que os SGBDOs se integrem em diversas áreas de aplicações, desde as aplicações de engenharia até aos sistemas de informação para gestão.

Com vista a que este objectivo possa ser atingido, é de crucial importância que apareçam:

- normas;
- firmas interessadas;
- e produtos concebidos para serem executados em redes locais de computadores pessoais com capacidade de ligação a bases de dados armazenadas em grandes computadores.

No mercado, existem já hoje produtos para ambiente Windows, que por rede permitem o acesso transparente a bases de dados remotas da Oracle e Sybase. A Object Design e a IBM desenvolvem em conjunto ligações para o sistema DB2. A questão que de facto ainda não se encontra resolvida é efectivamente a das normas. Destas normas, resultará a definição dum modelo de objecto comum e a definição de ligações a linguagens de programação que permitirão às equipas de desenvolvimento a escrita de aplicações para armazenamento dos seus dados num SGBDO, com a confiança que as aplicações são compatíveis com qualquer outro SGBDO de outro fornecedor.

1.4.4 Conceito Unificador: Objectos

Ao analisarmos o passado da ciência da computação, nomeadamente em relação à actividade de investigação desenvolvida, claramente identificamos a tendência de unificação de áreas inicialmente consideradas independentes. Na década de 70, os únicos aspectos em que as áreas dos Sistemas Operativos (SOs) e a dos SGBDs se sobrepunham eram os Sistemas de Ficheiros (SFs). Mais tarde, na década de 80, conclui-se que tal sobreposição não se restringe apenas aos SFs.

Actualmente, é de opinião geral que os desenvolvimentos nas áreas das Bases de Dados devem estar ligados às áreas das Linguagens de Programação (LPs) e das Interfaces com o Utilizador (IUs). A figura 1.3 ilustra esta tendência de unificação das diversas áreas da ciência da computação. O conceito que está na base desta unificação é o Modelo Objecto.

1.5 Proposta de Norma ODMG-93

A inexistência de uma norma para as bases de dados de objectos tem sido um obstáculo para o aumento da sua divulgação e utilização. O sucesso dos sistemas relacionais deve-se em muito também à normalização que oferecem. A aceitação de uma norma como o SQL permite um alto grau de portabilidade e interoperabilidade entre sistemas, e simplifica a aprendizagem de novos SGBDRs.

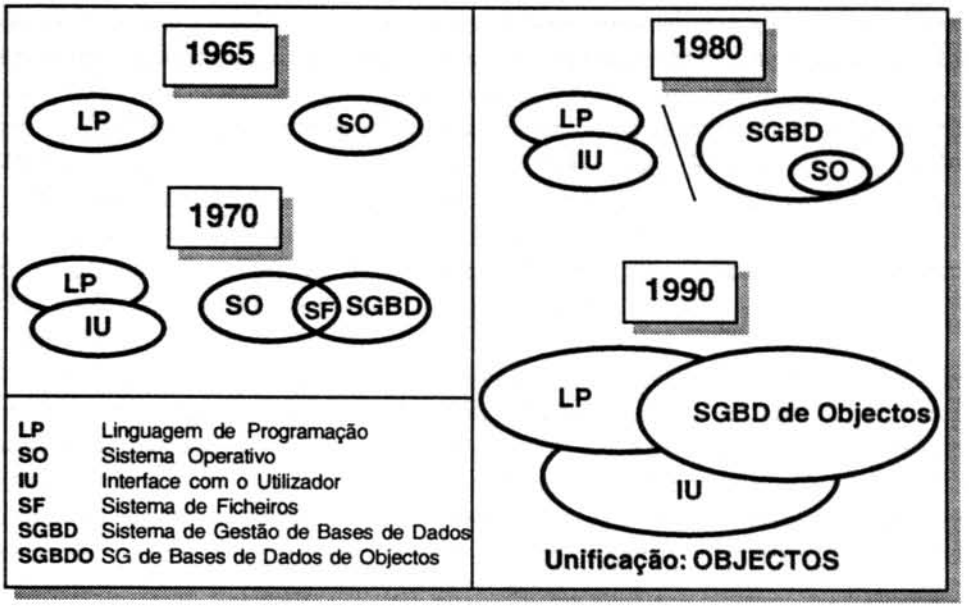


Figura 1.3: A investigação na ciência da computação

1.5.1 Importância de uma Norma

Todos estes factores, são ainda mais importantes para os SGBDOs, atendendo a que a maioria dos produtos desta área são oferecidos por pequenos fornecedores, e que a portabilidade e a aceitação dum produto são qualidades tidas como essenciais para quem compra.

Por outro lado, o âmbito de aplicabilidade dos SGBDOs é muito mais alargado do que o dos SGBDRs, uma vez que pretendem a integração de linguagens de programação e sistemas de bases de dados, pelo que acabam por abranger todos os dados e operações de uma aplicação. A definição de uma norma é por isso muito importante para que tais sistemas possam na prática se impôr comercialmente.

É por conseguinte, de primordial importância para a indústria de Sistemas de Bases de Dados de Objectos, a existência de normas. Com o objectivo de dar um salto neste sentido, os membros do *Object Database Management Group* (ODMG), publicaram em finais de 1993, resultado de cerca de um ano de trabalho, uma proposta de norma para os SGBDOs, o ODMG-93.

1.5.2 Objectivos do ODMG-93

O objectivo principal do ODMG-93 consiste na definição de um conjunto de normas que permita a um cliente dum determinado SGBDO a escrita de aplicações que possam ser transferidas para quaisquer outros produtos SGBDO. Para tal, é forçoso que o esquema de dados, a ligação com a linguagem de programação, e as linguagens de definição, manipulação e de interrogação sejam idênticas. O ODMG espera também que a proposta possa vir eventualmente a permitir interoperabilidade entre vários produtos SGBDO.

O ODMG-93 tenta que a integração das linguagens de programação com os sistemas de bases de dados seja muito superior ao que é hoje possível com os sistemas relacionais. Não se pretende apenas a portabilidade de uma pequena parte da aplicação, a relativa às instruções SQL, mas sim a portabilidade de toda a aplicação.

O grupo ODMG é composto por membros representativos de 90% do mercado existente de SGBDOs: Object Design, Objectivity, O_2 Technology, Versant Object Technology, Ontos, mboxHewlett-Packard, Intelletic, Itasca, Servio, e Texas- Instruments. Alguns dos colaboradores a nível académico são: Stan Zdonik, David Maier, Michael Carey e David Dewitt. O grupo é liderado por Rick Cattell da SunSoft. É intenção dos membros do ODMG implementarem o ODMG-93, pelo que esperam que se torne uma norma de facto para a indústria dos SGBDOs.

Com a definição de normas, não se pretende a criação de produtos SGBDO idênticos. O objectivo consiste na portabilidade do código fonte, deixando em aberto para futura inovação muitas outras áreas. As diferenças entre os produtos serão principalmente ao nível de performance, linguagens suportadas, funcionalidade específica tendo em vista determinados sectores de mercado, ambientes de desenvolvimento de programas, ferramentas para construção de aplicações, pequenos versus grandes sistemas, arquitectura paralelizável, redes, disponibilidade de plataformas, bibliotecas de tipos pré-definidos, ferramentas para construção de interfaces gráficas, ferramentas para apoio ao projecto, etc.

1.5.3 Componentes do ODMG-93

As componentes principais da proposta de norma são a definição dum Modelo Objecto e de um conjunto de interfaces com linguagens de programação. Estas interfaces têm como base as linguagens de definição, interrogação e manipulação de bases de objectos, respectivamente as *Object Definition Language* (ODL), a *Object Query Language* (OQL) e a *Object Manipulation Language* (OML), também definidas pelo ODMG-93.

O Modelo Objecto permite a atribuição dum significado preciso e formal aos conceitos básicos que determinam a estrutura da base de dados, conceitos tais como objecto, atributos de um objecto, relações entre objectos, e operações sobre objectos. O modelo objecto ODMG pode ser sumariamente caracterizado da seguinte maneira:

- A primitiva base do modelo é o *objecto*—algo cujo comportamento e propriedades podem ser definidas de forma precisa.
- Os objectos são categorizados em *tipos*. Todos os objectos de um determinado tipo possuem comportamento e propriedades comuns.
- O estado dos objectos é definido pelo conjunto dos valores das suas *propriedades*. As propriedades podem ser *atributos* do próprio objecto, ou *relações* entre o objecto e outros objectos.
- O comportamento dos objectos é definido por um conjunto de *operações* que lhes podem ser invocadas.
- Os tipos de objectos são organizados em hierarquias *tipo/supertipo*. Subtipos herdam o estado e comportamento definido nos seus supertipos.

As interfaces com as linguagens de programação têm a função de mapear os conceitos do Modelo Objecto para as mais importantes linguagens de programação orientadas por

objectos, nomeadamente para C++ e Smalltalk. Estas interfaces consistem na extensão da linguagem de programação com capacidades de bases de dados, com as quais o programador pode transparentemente criar, apagar, aceder e actualizar objectos armazenados na base de dados. Esta transparência torna desnecessária a aprendizagem de uma linguagem de manipulação de dados separada, evita a necessidade de explicitamente copiar e traduzir os dados entre as representações da base de dados e da linguagem de programação e tem vantagens ao nível de armazenamento temporário de dados nas aplicações (ver figura 1.4).

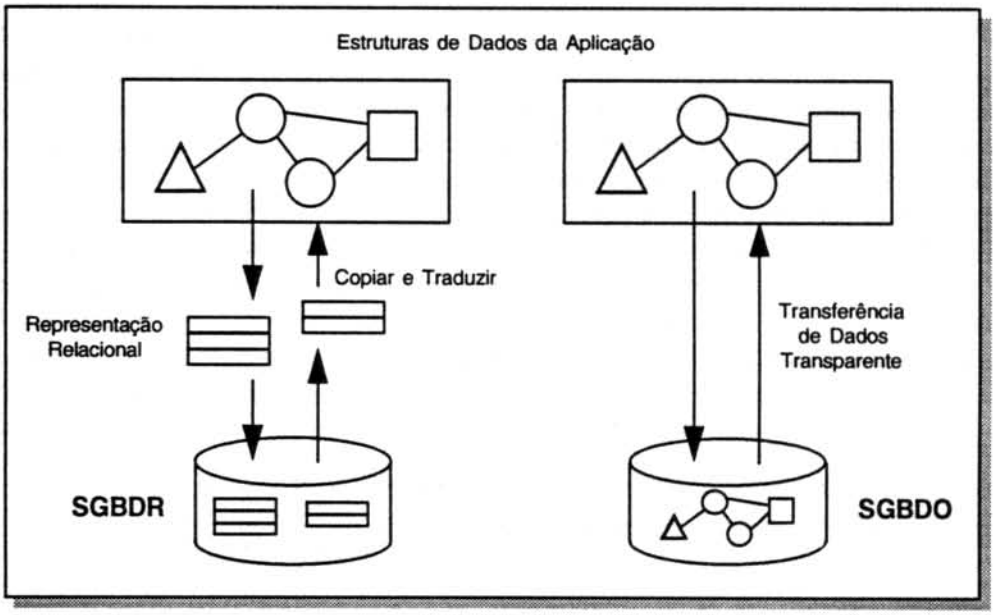


Figura 1.4: Duas arquitecturas de sistemas de bases de dados

Em resumo, o ODMG-93 define um SGBDO como um SGBD que integra capacidades de bases de dados e capacidades de linguagens de programação orientadas por objectos. Um SGBDO faz com que os objectos da base de dados pareçam ser objectos da linguagem de programação, numa ou mais linguagens de programação existentes.

1.6 Motivação para o Presente Trabalho

A motivação para a realização do presente trabalho nasce no INESC, no âmbito do Grupo TEIA—Tecnologias de Informática Avançada, na sequência de outros trabalhos e estudos realizados na área das metodologias de desenvolvimento de software orientadas por objectos e na área dos sistemas de bases de dados.

Tendo como base a experiência e conhecimentos existentes no Grupo na área de sistemas de bases de dados, foi decidido aprofundar conhecimentos em áreas novas relacionadas com o desenvolvimento de sistemas de informação, nomeadamente em: metodologias orientadas por objectos, ferramentas CASE, novos sistemas de bases de

dados, sistemas de bases de dados temporais, redes neuronais, e sistemas distribuídos. Exemplo de alguns dos trabalhos resultantes destes estudos, são: estrutura de diálogo para uma ferramenta CASE-OO baseada em OMT [San93], objectos persistentes em bases de dados relacionais [AS92], definição de dicionários de dados para armazenamento dos diagramas da análise estática, funcional e dinâmica propostas na metodologia OMT (ver [RBL+91]) [Sim92] [Bon92] [Ran92]; ferramenta para navegação em diagramas de classes OMT [Net93]; estudo de arquitecturas de sistemas de bases de dados temporais [Sim93]; predição de séries no tempo [Bon94], etc.

Alguns dos trabalhos anteriores foram desenvolvidos no âmbito dum projecto mais global do Grupo: a criação e exploração de ferramentas CASE para desenvolvimento de aplicações segundo uma metodologia de orientação por objectos.

Como resultado, foi até ao momento desenvolvido no Grupo, em ambiente NeXTSTEP, um protótipo de uma das componentes da ferramenta CASE, de nome MOOT—*Modeling Object Oriented Tool*: a sua estrutura de diálogo. Uma outra componente da ferramenta, igualmente importante, mas ainda não implementada, consiste num sistema de base de dados para armazenamento da informação manipulada pela ferramenta, bem como pelas aplicações com ela desenvolvidas. Nas figuras 1.5 e 1.6 são apresentadas duas versões da estrutura base da referida ferramenta: a estrutura actual e uma sua evolução fruto do recurso a um sistema de bases de dados.

O sistema de bases de dados a implementar deverá facilitar a adopção de um estilo de programação orientada por objectos, e por conseguinte, deverá possuir capacidades típicas de um Sistema de Gestão de Bases de Dados de Objectos [Cat91].

Em termos de desenvolvimento deverá ainda satisfazer as seguintes condições:

- *Arquitectura:* por questões de tempo e custos de desenvolvimento, o sistema deverá ser desenvolvido por extensão de um Sistema de Gestão de Bases de Dados Relacionais (SGBDR) comercialmente disponível;

- *Linguagem de Programação e Linguagens de Bases de Dados:* a integração das linguagens deve ser natural, por forma a diminuir a impedância semântica entre elas, e tornar mais fácil a construção de aplicações de bases de dados;

- *Ambiente de Desenvolvimento:* por questões de facilidade de integração com anteriores trabalhos desenvolvidos no Grupo TEIA, deverá o sistema ser implementável em ambiente NeXTSTEP, recorrendo à linguagem Objective-C como linguagem de programação, e ao Sybase, como SGBDR;

- *Normas:* o sistema deverá, na medida do possível, ter em consideração as normas existentes na área em que se insere, nomeadamente a recente proposta de norma ODMG-93, publicada em finais do ano transacto, e a linguagem SQL— norma existente para as linguagens de interrogação relacional.

A implementação do sistema de bases de dados pretendido é complexa, e envolve conhecimentos em diversas àreas, tais como: sistemas operativos, sistemas de ficheiros, linguagens de programação, compiladores, interfaces gráficas, e sistemas distribuídos. Pela dimensão da actividade de desenvolvimento potencialmente envolvida, o referido sistema não será implementado no âmbito deste trabalho, mas espera-se que este dê um contributo inicial para o posterior desenvolvimento do sistema pretendido, quiçá no âmbito de outros trabalhos de natureza idêntica.

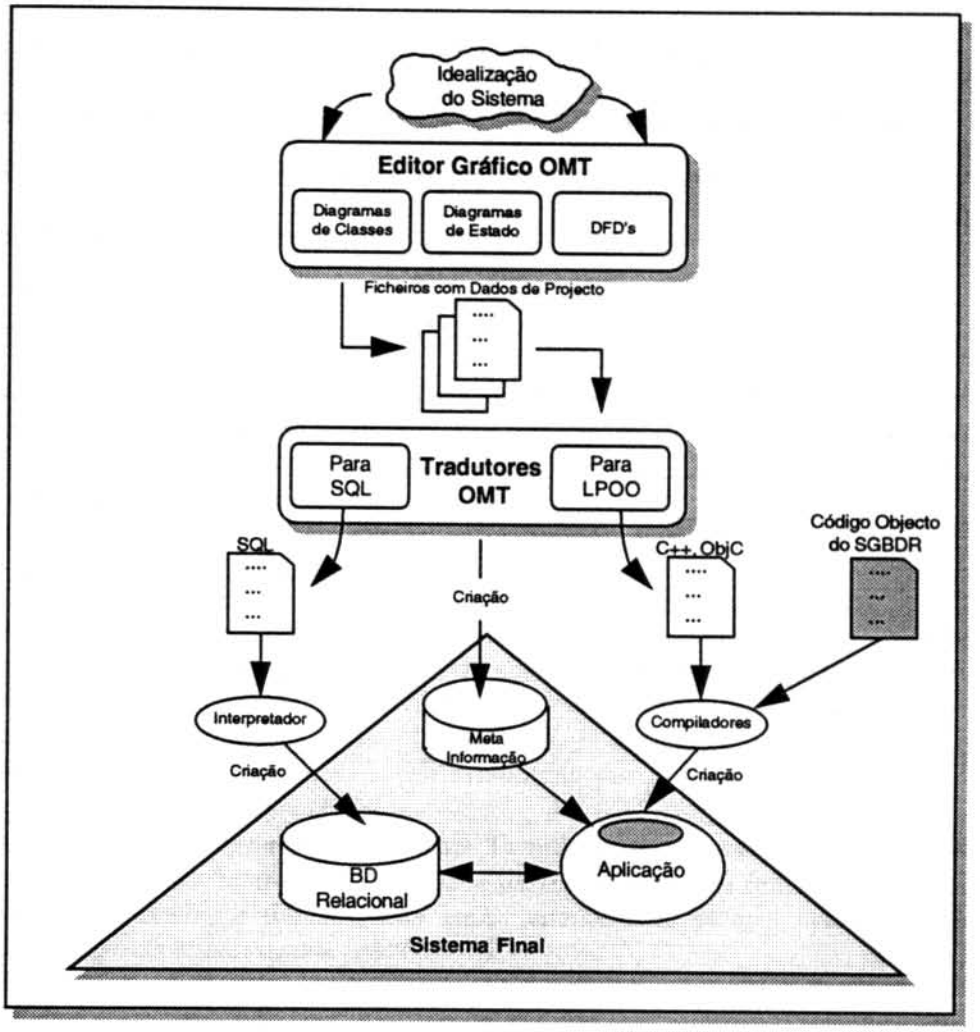


Figura 1.5: MOOT—recurso a ficheiros para armazenamento dos projectos

Assim, e por forma a potenciar futuros desenvolvimentos, é importante que o presente trabalho constitua em si uma boa base para iniciação ao estudo da temática *Novos Sistemas de Bases de Dados*. Um outro contributo igualmente importante, será o da definição de uma interface entre as duas componentes essenciais dos Novos Sistemas de Bases de Dados: o sistema de bases de dados e a linguagem de programação. A definição desta interface irá permitir que os futuros desenvolvimentos do sistema aos diferentes níveis (interfaces com o utilizador, bases de dados, ferramentas CASE, etc.) possam ser realizados independentemente, de forma evolutiva e modular. Tal interface, deixará em aberto questões de implementação das capacidades de bases de dados, prevendo-se no entanto uma sua implementação inicial à custa de um SGBDR. Concretamente, esta interface irá ser definida para a linguagem de programação Objective-C, e terá como base a proposta de norma ODMG-93.

1.6.1 Objectivos

Neste contexto, o presente trabalho tem os seguintes objectivos:

- constituir uma base para o estudo dos Novos Sistemas de Bases de Dados;
- delinear a arquitectura do sistema a implementar futuramente;
- especificar as características pretendidas para o futuro sistema;
- delinear uma interface em linguagem Objective-C para um sistema de bases de dados com base na proposta de norma ODMG-93.

Tal como já foi anteriormente mencionado, estes objectivos enquadram-se num objectivo mais global que transcende o âmbito deste trabalho de dissertação, e que consiste na construção de um Sistema de Gestão de Bases de Dados de Objectos para a ferramenta MOOT e aplicações com ela desenvolvidas.

1.6.2 Estrutura da Dissertação

Após esta não muito breve introdução, será apresentado no próximo capítulo o *Contexto* que está na base do aparecimento dos novos sistemas de bases de dados. Serão examinadas as aplicações de bases de dados que motivaram o aparecimento desta nova tecnologia, e apresentados os novos modelos de dados. Serão também abordados os principais conceitos associados às linguagens de programação orientadas por objectos, das quais os novos SGBDs adoptam diversas capacidades.

Mais à frente, será apresentada a *Nova Tecnologia de Bases de Dados de Objectos*. Começarão por ser revistos os sistemas a partir dos quais os SGBDOs evoluíram—os SGBDs tradicionais, e depois serão então enumerados os principais conceitos e características gerais apresentados pelos SGBDOs.

Tendo em vista delinear uma interface em Objective-C para um Sistema de Gestão de Bases de Dados de Objectos para implementação em ambiente NeXTSTEP, será apresentada a recente proposta de norma para os SGBDOs—a ODMG-93. A interface será delineada com base nesta proposta de norma.

O trabalho terminará com algumas conclusões, e perspectivando futuros trabalhos na sequência deste.

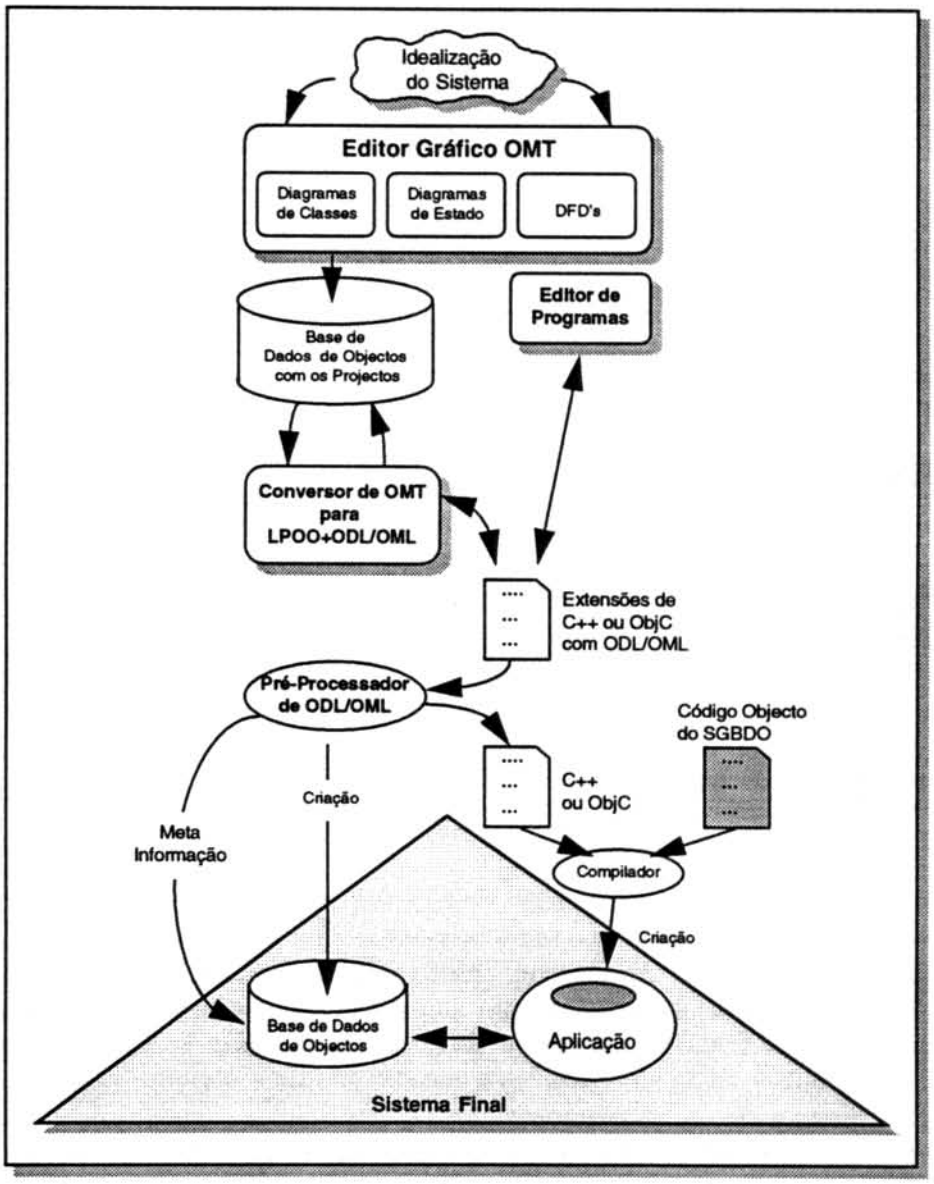


Figura 1.6: MOOT—recurso a uma base de dados para armazenamento dos projectos

2

Contexto

Neste capítulo, pretende-se dar uma visão geral sobre o que está na base do aparecimento dos Novos Sistemas de Bases de Dados. Nesta perspectiva, serão examinadas algumas das novas aplicações de bases de dados que motivaram o desenvolvimento desta nova tecnologia, de entre as quais se destacam as aplicações de CAD, CASE e escritório electrónico, pelos problemas reais que colocam à tecnologia de bases de dados actualmente existente.

Na base dos novos sistemas, encontram-se novos modelos de dados, nomeadamente os modelos de objectos e os modelos relacionais estendidos. Pela sua importância para os novos desenvolvimentos, serão mais à frente apresentadas as suas principais características.

Um dos objectivos dos novos sistemas é a integração de capacidades de sistemas de bases de dados nas linguagens de programação. As linguagens de programação orientada por objectos (LPOO), pela sua proximidade em termos de modelos de dados aos novos sistemas, são as linguagens eleitas para esta integração, pelo que o domínio dos principais conceitos associados a estas linguagens é importante. No final deste capítulo serão por conseguinte revistas as características destas linguagens.

2.1 Novas Aplicações

Das aplicações novas que motivaram o aparecimento da nova tecnologia de bases de dados, as consideradas mais importantes e simultâneamente as mais citadas como sendo as mais beneficiadas pelo recurso a SGBDOs pertencem às seguintes áreas [Cat91, RBL+91]:

- *Engenharia de Software:* produção de pequenos e grandes sistemas de software.
- *Engenharia Mecânica e Electrotécnica:* projecto de produtos físicos e electrónicos.
- *Documentos:* Documentos de texto, hipertexto, gráficos e escritório electrónico.

Para o presente trabalho, serão especialmente consideradas as aplicações pertencentes às áreas acima indicadas, das quais as primeiras duas assumem um interesse especial. Outras áreas em que algumas aplicações específicas poderão também beneficiar com o recurso a novos sistemas de bases de objectos são:

- *Ciência e Medicina:* representação e manipulação de dados complexos, tais como modelos moleculares para compostos orgânicos, análises espectrais, ou codificação genética.
- *Controlo de Processos de Manufactura e Tempo Real:* utilização de computadores para controlo e monitorização de processos fabris.
- *Sistemas Periciais:* representação de bases de conhecimento e regras para aplicações de Inteligência Artificial.
- *Aplicações Comerciais Novas:* extensões de aplicações tradicionais com exigências novas em termos de manipulação de informação.

Após uma análise mais detalhada de cada um dos tipos de aplicações acima enunciados, serão examinados os requisitos que as diversas aplicações têm em comum, tanto requisitos típicos de sistemas tradicionais, como novos requisitos.

2.1.1 Engenharia de Software

O recurso a computadores para apoio ao desenvolvimento e manutenção de sistemas de software é uma actividade a que se dá o nome de *Engenharia de Software Assistida por Computador*, ou mais vulgarmente de CASE—*Computer Aided Software Engineering*. Uma das principais utilizações das estações de trabalho na área de engenharia na última década foi precisamente em actividades de CASE.

Passado

A actividade CASE, Pode-se afirmar que a actividade CASE foi uma das primeiras aplicações dos computadores, no principio da década de 60. Alguns exemplos de aplicações que auxiliaram o desenvolvimento de software, mesmo antes da actividade de programação ser considerada "engenharia de software", são os compiladores de linguagens de alto nível e as ferramentas para desenho de fluxogramas.

Nos finais de 70, as ferramentas de engenharia de software dividiram-se em duas grandes categorias: as destinadas a grandes projectos e as destinadas a pequenos projectos.

Associadas a ferramentas da primeira categoria encontram-se capacidades de coordenação de projectos, como por exemplo: gestão de múltiplas versões de projectos de software, e escalonamento de projectos. As ferramentas da segunda categoria, destinam-se a um único programador, e alguns exemplos são: ferramentas para concepção de programas, editores de programas, e ferramentas para controlo da compilação de módulos interdependentes.

Desde então o termo "CASE" tem evoluído, sendo actualmente utilizado para fazer referência a ferramentas que abrangem diversas fases do ciclo de vida dos sistemas de software de grande dimensão, desde a especificação à manutenção. Iremos analisar mais detalhadamente algumas delas.

Ferramentas CASE

De acordo com os aspectos em que incidem com maior predominância, as ferramentas CASE podem ser classificadas em quatro grandes categorias:

- *Ferramentas CASE Front-end* suportam fases de concepção de alto nível, especificação, e análise de software e de requisitos. Nestas ferramentas estão incluídas as ferramentas para desenho de diagramas segundo uma determinada metodologia de concepção de programas [DeM78, YC78], e que mais recentemente passaram a suportar metodologias de concepção orientada por objectos.

- *Ferramentas CASE Back-end* suportam uma concepção detalhada, produção de código executável e teste de software. Estas ferramentas podem auxiliar o programador directamente; por exemplo, elas incluem ferramentas gráficas para depuração de erros, e capacidade de interrogação e pesquisa de código, por forma a serem encontrados determinados procedimentos ou utilizações de uma variável específica.

- *Ferramentas de Manutenção* apoiam o software nas fases pós-lançamento da primeira versão. Estas ferramentas auxiliam na correcção de erros detectados, na resposta a pedidos de melhoramentos, na adaptação a outras plataformas, e nos processos de criação e gestão de versões.

- *Software de Suporte* às ferramentas descritas nos pontos anteriores. Basicamente, consiste em software que fornece a funcionalidade básica exigida pelas ferramentas referidas anteriormente, e que incluem facilidades desde o nível do sistema operativo até a níveis superiores de gestão de projectos, escalonamento, e sistemas de bases de dados para acesso a diferentes versões e configurações de sistemas de software.

A acrescentar à metodologia CASE baseada nas linguagens e ferramentas de programação tradicional, existem duas abordagens diferentes de CASE, particularmente para ferramentas *Back-end*, e que são:

- *Linguagens de alto nível:* as linguagens de 4^a Geração (4GLs), para auxiliar em várias tarefas, desde o desenho do esquema duma base de dados até à elaboração de formulários para aplicações comerciais de bases de dados em terminais gráficos ou alfanuméricos.

- *Sistemas Periciais:* a aplicação de inteligência artificial à programação, para produção automática de código em domínios restritos, é uma outra abordagem de CASE. No entanto, esta abordagem tem visto a sua aplicação limitada pelo facto de na prática ser extremamente díficil conseguir que uma programação automática seja suficientemente genérica.

Ciclo de Vida de um Projecto de Software

O ciclo de vida completo de um projecto de um sistema de software tem várias fases. Uma visão simplificada dos vários passos do processo é ilustrada na figura 2.1:

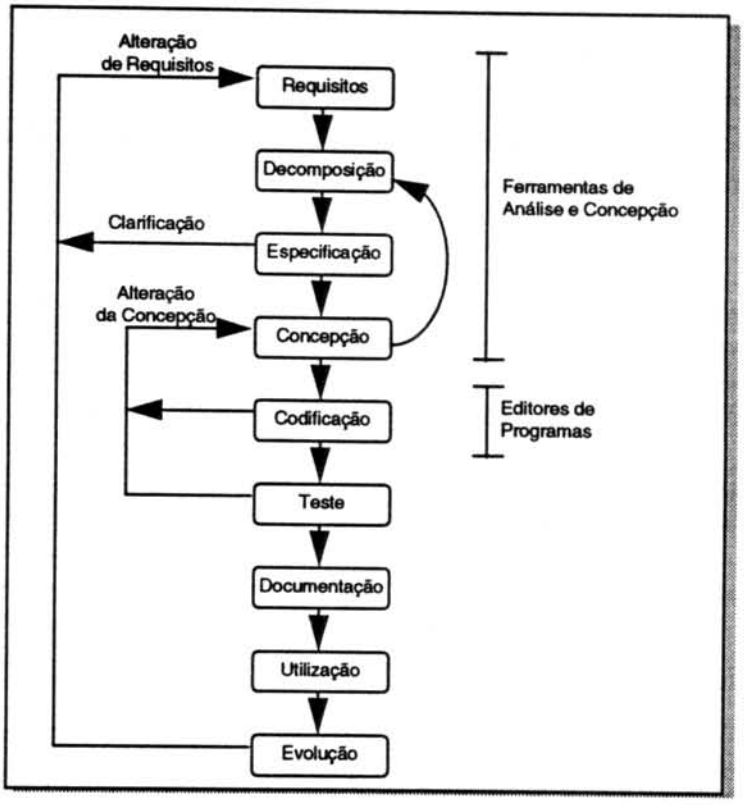


Figura 2.1: Fases típicas do ciclo de vida do software

- *Requisitos:* Resulta normalmente num documento onde são definidos os requisitos a satisfazer pelo software.
- *Decomposição:* O sistema de software é decomposto com vista à redução da sua complexidade, e por forma a poder ser distribuído por vários programadores para ser implementado. O tipo de decomposição poderá ser basicamente de dois tipos: algorítmica ou orientada por objectos.
- *Especificação:* São definidos os subcomponentes a atribuir a cada programador, a funcionalidade exigida, e as respectivas interfaces entre os vários subcomponentes.
- *Codificação:* Implementação de cada subcomponente. Para cada subcomponente, e antes da codificação ser iniciada, poderá ser necessário refinar a especificação, o que eventualmente poderá obrigar a nova passagem pelas fases de decomposição e especificação.

- *Teste e Integração:* Verificação da correcção de cada subcomponente e de todo o sistema face à respectiva especificação.
- *Documentação:* Fornece documentação para o programa. Esta fase pode ser desenvolvida paralelamente com algumas das fases anteriores.
- *Utilização:* Os utilizadores começam a trabalhar com o sistema de software resultante, dando as suas opiniões em termos de performance e funcionalidade. Durante este processo, são elaborados relatórios dos problemas encontrados e são feitas estatísticas de utilização para posterior avaliação.
- *Evolução:* São repetidas algumas das fases anteriores por forma a melhorar a funcionalidade, a corrigir erros e a adaptar o sistema com base nas opiniões emitidas pelos utilizadores.

Na figura 2.1 são também ilustradas algumas das ferramentas utilizadas nas várias fases do ciclo de vida. Pode-se depreender que as fases de requisitos, decomposição, e especificação correspondem a ferramentas *Front-end*; as fases de codificação, teste e documentação correspondem a ferramentas *Back-end*; e finalmente que as fases de utilização e evolução correspondem a ferramentas de manutenção.

A ordem acima mencionada para as diversas fases, nem sempre é respeitada integralmente, sendo normal repetir fases anteriores, ou não cumprir algumas fases. De facto, qualquer ordem é possível, e cada um escolhe aquela que melhor se aplica a um determinado problema. Um aspecto importante a considerar é de que o ciclo de vida dos sistemas de software é complexo, especialmente durante a sua evolução. Por esta razão, aspectos como modularidade, reutilização, e outros do paradigma de orientação por objectos, assumem grande importância em actividades de CASE.

Requisitos das Ferramentas CASE

Para se entender os requisitos das aplicações CASE em termos de tecnologia de bases de dados, iremos analisar alguns exemplos de tipos de informação exigidos para o suporte de ferramentas CASE.

O primeiro exemplo diz respeito a ferramentas do tipo *back-end*, vocacionadas para auxiliar o desenvolvimento de pequenos programas, especialmente no ciclo de edição- compilação-correcção de erros. Este ciclo pode ser melhorado de muitas formas. Recentemente, a tendência tem sido a de colocar à disposição do programador o máximo de informação possível durante a fase de correcção de erros.

Para tal, são colocadas à disposição dos programadores ferramentas gráficas, nas quais, entre outras coisas, se permite: a definição de pontos de paragem de execução dos programas, apontando o rato directamente no texto do programa; a inspecção das estruturas de dados do programa; a observação da pilha de chamadas de procedimentos; a actualização constante da informação relativa aos dados do programa, embora dispersa pelas várias janelas de monitorização de execução do programa, etc.

Outros tipos de ferramentas permitem pesquisar referências a um determinado procedimento ou variável, obter a definição de uma função, ou mais genericamente, permitem que se façam interrogações sobre toda e qualquer informação relacionada com os programas.

Basicamente, estas ferramentas requerem o manuseamento da informação simbólica dos programas, normalmente produzida pelos compiladores e utilizada posteriormente pelas ferramentas para correcção de erros.

Para além dos anteriores tipos de ferramentas, existem dois outros igualmente importantes, para a coordenação das tarefas de programação: gestão de configurações e gestão de projectos.

As ferramentas para gestão de configurações possibilitam aos programadores acesso concorrente ao código, mantendo informação sobre as diferentes versões de código em utilização pelos programadores.

As ferramentas para gestão de projectos auxiliam na gestão de especificações, de dependências entre módulos, e escalonamento de subprojectos num projecto de software. Estas ferramentas podem também auxiliar durante o processo de desenvolvimento, bem como na produção de estimativas de custos e definição de prazos de conclusão.

Ferramenta MOOT

A ferramenta MOOT enquadra-se no âmbito das ferramentas CASE *Front-End* e consiste num sistema gráfico que suporta uma metodologia de análise, concepção e programação. O processo desenvolvimento suportado pela ferramenta tem como a metodologia OMT.

A MOOT permite o desenho de diagramas de classes OMT para as fases de análise e concepção, e apoia o processo de implementação do sistema. Tendo como base a concepção permite a criação automática das classes e bases de dados necessárias à sua implementação. A ferramenta será também útil durante a fase de manutenção ao possibilitar o *reverse-engineering* do sistema, isto é, ao possibilitar a obtenção de grande parte da concepção e análise de um sistema orientado por objectos tendo como base a implementação do sistema.

Em termos de requisitos, a ferramenta MOOT é em tudo idêntica às ferramentas CASE referidas anteriormente.

2.1.2 Engenharia Mecânica, Civil e Electrotécnica

A concepção de sistemas complexos é uma das tarefas mais dificilmente realizáveis. A sua dificuldade deve-se a várias razões:

- A concepção pode ser muito extensa, isto é, pode envolver muitas partes individuais.
- Existem normalmente muitos subsistemas interdependentes cuja concepção tem de ser coordenada. Um bom exemplo disto é a construção de edifícios, em que existem vários projectos desde o nível das estruturas, até ao da segurança, passando pelos eléctricos, hidráulicos e termodinâmicos.
- A concepção é um processo que evolui com o tempo, e como tal está sujeito a muitas alterações, alterações estas que terão de ser propagadas para todos os níveis de representação.
- Por fim, a concepção poderá envolver muitas pessoas: engenheiros, gestores, técnicos e outros trabalhadores. Todos eles deverão trabalhar em paralelo, possivelmente em múltiplas versões de uma mesma concepção, mas tendo em consideração que o produto final deverá ser coerente e bem articulado.

Embora sempre se tivesse recorrido a ferramentas para apoio ao projecto e desenho de grandes sistemas, foram no entanto as estações gráficas que tornaram possível o aparecimento de uma nova geração de ferramentas de desenho.

Desenho Assistido por Computador

O Desenho Assistido por Computador (CAD—*Computer Aided Design*) consiste na utilização de ferramentas de software no processo de desenho. Existem duas grandes categorias de ferramentas de CAD e, engenharia:

- *CAD para Mecânica (MCAD):* destinam-se ao desenho de peças, desde os simples parafusos até a carros e aviões.
- *CAD para Electrónica (ECAD):* destinam-se ao desenho de componentes electrónicos de pequena escala, isto é, circuitos integrados, bem como de placas de componentes e sistemas completos.

Processo de Desenho

As ferramentas MCAD e ECAD devem suportar todo o ciclo de vida de um projecto. O ciclo de vida de um projecto eléctrico ou mecânico é em tudo idêntico ao do software, de facto, CASE tem sido também apelidado de CAD para software. As fases pelas quais passam uma actividade de CAD são:

- *Requisitos:* Determinação dos requisitos que o sistema deve satisfazer, ou do peso relativo dos vários parâmetros do desenho.
- *Decomposição:* Definição dos componentes principais do sistema por forma a satisfazer os requisitos. Os componentes poderão então ser atribuídos a diferentes pessoas ou grupos.
- *Especificação:* É decidida qual a funcionalidade exigida a cada um dos subcomponentes. Este processo decompõe os requisitos globais do sistema em requisitos parciais para cada um dos subcomponentes.
- *Desenho:* É escolhida uma implementação detalhada para cada subcomponente. Este processo normalmente envolve uma passagem recursiva pelas fases de decom posição e especificação, por forma a definir subcomponentes cada vez mais pequenos.
- *Análise e Verificação:* O desenho é examinado a diferentes níveis, para verificação da sua consistência e correcção face à especificação. Para tal verificação, poder-se-á recorrer a simulações ou a outros métodos de verificação.
- *Implementação:* Realização física do sistema e integração dos componentes no sistema final. A esta fase também é dado o nome de prototipagem.
- *Teste:* Verificação de que o protótipo satisfaz os requisitos originais. Idealmente, ter-se-á definido um método de teste durante a fase de desenho.
- *Documentação:* Produção de documentação, quer para utilização para manutenção do sistema. Esta fase poderá ocorrer paralelamente com outras.
- *Utilização:* O sistema final é posto em utilização, sendo ouvidas as opiniões relativas à sua performance e funcionalidade. Relatórios de problemas encontrados e de estatísticas de utilização, podem ser guardados durante esta fase, para posterior avaliação.

- *Evolução:* Repetição de fases anteriores por forma a corrigir problemas encontradas e a refinar o sistema.

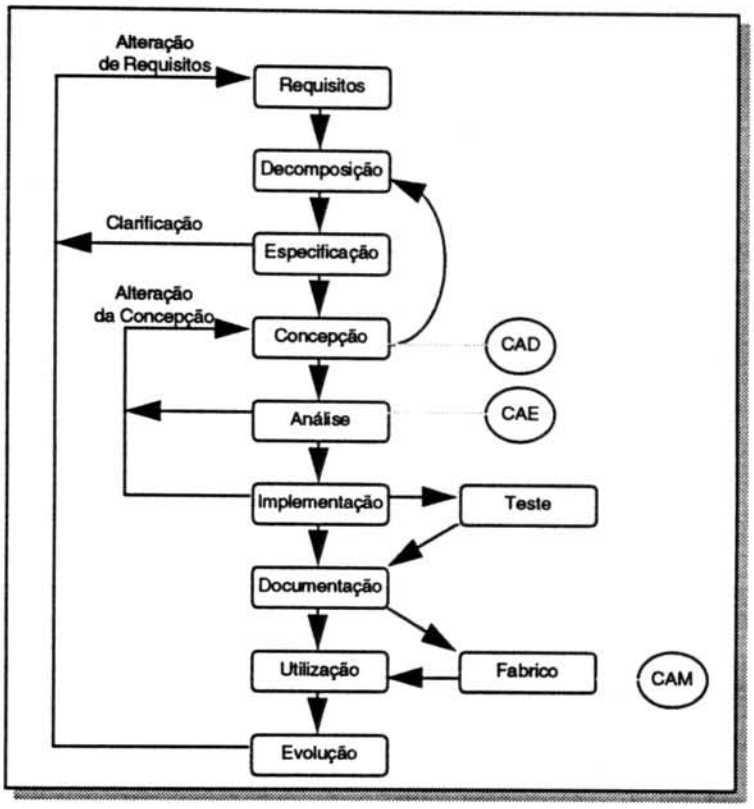


Figura 2.2: Fases típicas do ciclo de vida de projectos eléctricos e mecânicos

Tal como na engenharia de software, o processo de concepção desafia qualquer definição precisa—diz-se ser mais uma arte do que uma ciência. Muitos poucos casos existirão em que os passos anteriormente apresentados serão dados na sequência apresentada. Poder-se-á até dar o caso de se ter que voltar ao primeiro passo, devido aos requisitos do sistema terem sido alterados por alguma razão de força maior. Nestas situações e por forma a aproveitar trabalho realizado, a propagação das alterações de requisitos deverá ser feita de forma inteligente. A utilização de bases de dados, para gerir a propagação de alterações, pode simplificar significativamente o desenho de aplicações, sendo o paradigma de orientação por objectos também importante.

Na figura 2.2 são ilustradas as fases típicas do ciclo de vida de projectos eléctricos ou mecânicos, bem como algumas das aplicaçções utilizadas nas suas várias fases.

Ferramentas de Desenho

Tanto para o desenho de esquemas mecânicos como eléctricos, existem três tipos de ferramentas:

- *Desenho Automático:* Trata-se de ferramentas para derivar automaticamente parte de uma concepção a partir de uma descrição de mais alto-nível dos requisitos e especificações. No limite, deveria ser possível com base numa descrição do comportamento de um circuito, a derivação automática da concepção, conseguindo assim eliminar a maior parte dos passos anteriormente descritos.

- *Desenvolvimento:* Trata-se de ferramentas para editar ou criar um projecto (civil, mecânico ou eléctrico). Estas ferramentas normalmente permitem seleccionar, interrogar, e visualizar informação relativa ao projecto. É este tipo de ferramentas a que normalmente se refere quando se fala em ferramentas CAD.

- *Análise:* Dois exemplos de ferramentas de análise são os simuladores de circuitos eléctricos e os verificadores de regras de projecto.

Todas estas ferramentas recorrem a sistemas de bases de dados para armazenarem a informação que manipulam. Estes sistemas de bases de dados são normalmente fechados, específicos de cada fabricante, e não acessíveis pelos utilizadores, por se encontrarem embebidos nelas. Por outro lado, os utilizadores necessitam quase sempre de ferramentas de diferentes fabricantes, algumas delas com incompatíveis representações internas de informação. Por forma a normalizar estas representações de informação, seria necessário desenvolver um mecanismo comum de interface das ferramentas com as bases de dados, mecanismo este que fosse mais poderoso do que os oferecidos pelos SGBDs actualmente existentes.

Por forma a entendermos melhor as exigências colocadas pelas aplicações de ECAD e MCAD, vamos analisá-las em mais detalhe.

Aplicações de CAD para Electrónica

As ferramentas de ECAD—*Electrical Computer Aided Design* destinam-se a auxiliar no desenvolvimento de circuitos integrados, placas de circuito impresso, ou sistemas electrónicos completos. O desenho de circuitos integrados é provavelmente o mais complexo dos três.

No desenho de circuitos electrónicos, utilizam-se vários níveis de representação:

1. Linguagens de Alto-Nível para descrição do comportamento do circuito.
2. Diagramas de Blocos.
3. Diagramas Lógicos.
4. Diagramas de Circuitos.
5. Subníveis Especializados.
6. Máscaras de Circuitos Integrados.

Os projectistas de circuitos integrados normalmente utilizam todos estes níveis de representação antes de chegarem mesmo à produção física do circuito.

A possibilidade de vários projectistas desenvolverem diferentes partes de um mesmo circuito, acarreta dificuldades acrescidas na propagação das alterações que seguramente irão ser feitas até ser atingido o produto final.

A componente em termos de bases de dados de todo este processo é significativa, envolvendo a manutenção de muitas representações diferentes e de informação diversa sobre cada um dos componentes do projecto.

O tamanho de uma base de dados de uma ferramenta ECAD poderá ser relativamente grande. No entanto, existe normalmente uma pequena biblioteca de componentes base a partir dos quais a base de dados é construída, que tirando partido de uma certa repetição natural existêente na maioria dos circuitos, vai contribuir para que o tamanho da base de dados não cresça de forma exagerada.

Partes do processo de desenho podem ser automatizadas. Em alguns casos, é possível automatizar a conversão de representações a níveis diferentes. Alguns exemplos são:

- Derivação automática dos diagramas lógicos de um circuito a partir de uma descrição mais alto-nível do funcionamento do circuito (passagem do nível 1 ou 2 para o nível 3).
- Geração das máscaras para produção do circuito a partir do diagrama do circuito (passagem do nível 4 para o 6).
- Optimização automática da disposição do circuito (geração de novos diagramas ao nível 3 ou 4).

Apesar de já ter sido desenvolvido muito trabalho na área de automatização de projectos, muito mais há a fazer antes de se conseguir automatizar todo o processo. As ferramentas existentes mais importantes na área de ECAD são as de edição e análise de projectos.

As ferramentas de ECAD de utilização mais alargada são os editores gráficos de circuitos (níveis lógico e físico). Os editores gráficos oferecem capacidades adicionais de verificação de regras de projecto, ou outras para auxiliar no desenho dos circuitos. Outro tipo de ferramentas de igual importância, embora menos utilizadas, são as destinadas à análise e síntese, das quais se destacam os seguintes tipos:

- Ferramentas de simulação.
- Ruído eléctrico e derivação com a temperatura.
- Verificação de regras de projecto.
- Programas de optimização de representações ao nível físico de circuitos.

Os requisitos mais exigentes em termos de bases de dados das ferramentas ECAD são precisamente os impostos pelas ferramentas de análise e síntese. A simulação de um circuito deve ser extremamente rápida por forma a produzir resultados úteis. Porque o software é muitissimo mais lento do que os circuitos equivalentes, mesmo recorrendo à base de dados mais eficiente possível, a melhor solução quase sempre é a de produzir código específico para o circuito, compilá-lo e finalmente fazer a simulação.

O esquema completo de uma base de dados para uma ferramenta ECAD deverá incluir todos os níveis de representação do projecto, não apenas os circuitos, mas também todas as correspondências e relações entre os diferentes níveis.

Aplicações de CAD para Mecânica

As ferramentas de CAD e CAE para mecânica dedicam-se ao projecto de partes e estruturas físicas construídas a partir de uma variedade de materiais. Este tipo de projecto mecânico é utilizado por arquitectos, engenheiros civis, na indústria automóvel, na indústria aeroespacial, etc.

Num projecto de mecânica são utilizadas desde as ferramentas para desenho até às de análise para verificação de consistência—ao nível estrutural, térmico, hidrodinâmico, electromagnético, etc. Para o fabrico, existe uma variedade de ferramentas automáticas, desde as simples máquinas com controlo numérico até a robots sofisticados.

Algumas características importantes que as ferramentas de MCAD—*Mechanical Computer Aided Design*, devem possuir para facilitar a gestão de grandes projectos são: possibilidade de decompôr o projecto em várias camadas, por forma a simplificar a edição do projecto; e dispôr de mecanismos que permitam uma navegação rápida por todo o projecto.

2.1.3 Documentos e Aplicações de Escritório Electrónico

A terceira grande área de aplicações é a de concepção, utilização, e manipulação de *documentos*, especialmente em aplicações de escritório electrónico. Estes documentos, incluem os tradicionais documentos de texto, hipertexto, gráficos, formulários, memorandos, correio electrónico, e qualquer outro formato de agregação de informação utilizada para comunicação entre pessoas.

Existem dois tipos fundamentalmente diferentes de bases de dados de documentos:

- Aplicações em que o próprio documento se encontra decomposto na base de dados; exemplos são os editores de vários tipos, tais como, editores gráficos, ou sistemas hipertexto.
- Aplicações em que os documentos são armazenados como um todo e é utilizada uma base de dados para armazenar informação àcerca dos documentos; exemplos são o correio electrónico e sistemas de bibliotecas para escritório electrónico,

Para o primeiro tipo de aplicações, os sistemas de bases de dados de objectos tornam-se mais importantes à medida que as estruturas de dados dos documentos a armazenar se tornam mais complexas, e que o acesso passa a ser partilhado por vários utilizadores numa rede de computadores.

Os sistemas que lidam com o segundo tipo de documentos, normalmente são apelidados de sistemas de biblioteca ou de escritório electrónico. Estas aplicações permitem o encaminhamento, processamento, e pesquisa de documentos. São normalmente utilizados para estabelecer mecanismos de comunicação entre diversos utilizadores.

2.1.4 Outras Aplicações

Para além das aplicações para projecto de sistemas, existe uma variedade de outras aplicações em que os sistemas tradicionais de bases de dados não são completamente

satisfatórios. Estas aplicações são outros potenciais clientes de um SGBDO:

- *Médicas e Científicas:* Muitas disciplinas (quimica, física, biologia, medicina) manipulam informação suficientemente complexa, razão pela qual beneficiariam com o recurso a um SGBDO.
- *Sistemas Periciais:* As regras e dados das aplicações de Inteligência Artificial podem atingir um volume tal que necessite o recurso a sistemas de bases de dados.
- *Manufactura e Tempo-Real:* Aplicações de monitorização e controlo de processos necessitam aceder a informação partilhada, pelo que poderão tirar benefícios da superior performance oferecida pelos novos SGBDOs.
- *Aplicações Comerciais Mais Complexas:* Muitas aplicações existentes e outras novas poderão também beneficiar das novas capacidades oferecidas pelos SGBDOs.

Aplicações Médicas e Científicas

Uma variedade de aplicações científicas, médicas e outras relacionadas podem beneficiar com o recurso a um SGBDO:

- *Química:* Particularmente na quimica orgânica, uma base de dados de objectos pode ser útil para a representação e manipulação de moléculas de estrutura complexa.
- *Genética:* O mesmo é válido para a representação do DNA, cuja informação é extremamente complexa para poder ser armazenada em bases de dados tradicionais.
- *Cromatografia:* Bases de dados de padrões cromatográficos podem ser bastante grandes e requererem uma eficiente indexação para poderem ser comparados com amostras.
- *Cartografia:* Mapas contendo informação com estruturas complexas, e requerendo estruturas de dados flexíveis.

Todas estas aplicações têm em comum a característica de não poderem ser satisfeitas pelos sistemas de bases de dados tradicionais. Por outro lado, porque manipulam informação suficientemente complexa, torna-se difícil a implementação de um repositório de dados a partir de estruturas de dados existentes em linguagens de programação tradicionais. As estruturas de dados oferecidas pelas linguagens de programação são inadequadas por ser necessário gerir muitos megabytes de informação e utilizar índices e facilidades de processamento de interrogações. Por outro lado, as bases de dados relacionais são inadequadas porque: as aplicações necessitam de ordenar a informação armazenada, de a agrupar em conjuntos, de novos métodos de acesso, e privilegiam, em termos de performance, acessos por um único utilizador relativamente a acessos multi-utilizador.

Sistemas Periciais

Muitas das características incorporadas nos sistemas de bases de dados tiveram a sua origem na área da representação de conhecimento em Inteligência Artificial: objectos, hierarquias de tipos, procedimentos, etc. De uma forma geral, no entanto, os sistemas

para representação de conhecimento não dispõem de capacidades típicas das bases de dados, tais como: armazenamento de grandes quantidades de informação e possibilidade de acesso concorrente à informação que armazenam.

Os SGBDOs podem ser úteis para a implementação das bases de dados de suporte aos sistemas periciais. Estas bases de dados são persistentes, podem tomar dimensões que impeçam o seu armazenamento total em memória principal, podem requerir o acesso concorrente à informação, ou exigir eficiente processamento de interrogações para pesquisar regras ou objectos segundo determinadas condições. Nestas situações, a utilização de um SGBDO que consiga lidar eficientemente com grandes quantidades de informação sem o peso dos SGBDs tradicionais, poderá se manifestar de grande utilidade.

Aplicações de Manufactura e Tempo-Real

Alguns exemplos de aplicações para processos de fabrico (CAM—*Computer Aided Manufacturing*) e tempo-real podem ser: o controlo de uma linha de montagem e o controlo de um processo químico.

- *Controlo de uma linha de fabrico:* Um sistema de computadores poderá ser dedicado à monitorização e controlo de uma linha de fabrico. O sistema deverá manter informação sobre a localização e estado das partes envolvidas, controlar automaticamente o equipamento de teste e, por exemplo, gerir o estado das encomendas dos clientes.

- *Controlo de um processo químico:* Um computador poderá controlar um processo químico, monitorizando sensores de vários tipos, controlar válvulas, aquecer e arrefecer determinados elementos, etc. Um sistema deste tipo necessita de uma base de dados para manter o estado corrente, dados históricos e as medidas a tomar em circunstâncias especiais.

As aplicações de controlo em tempo-real têm exigências em termos de performance que não são satisfeitas pelos SGBDs tradicionais, mas por outro lado, necessitam apenas de pequenas capacidades de gestão de informação.

Por outro lado, as aplicações de CAM têm necessidades a vários níveis: tempo-real, para controlo das linhas de montagem; estruturas de dados complexas transferidas das aplicações de CAD; e exigências típicas das aplicações comerciais tradicionais. Este tipo de aplicações é por conseguinte dificil de implementar, pelo que também poderá beneficiar com o recurso a um SGBDO.

Novas Aplicações Comerciais

Os SGBDOs encontrarão também aplicação na área comercial, apesar de esta se encontrar já dominada pelos sistemas de bases de dados tradicionais.

Como aplicações comerciais entenda-se uma larga variedade delas: aplicações de processamento de transacções, sistemas para controlo de reservas de lugares nas linhas aéreas, máquinas de atendimento automático, aplicações de apoio à decisão, etc. À medida que os computadores vão evoluindo, que os seus preços vão baixando, e que por consequência a variedade de aplicações se alarga cada vez mais, é natural que se esperem alterações nos requisitos deste tipo de aplicações.

Um sistema de reservas para companhias aéreas, poderia beneficiar com a capacidade de, por exemplo, armazenar e modificar informação espacial em diagramas de reservas de lugares.

2.1.5 Requisitos em Termos de SGBDOs e SGBDs Tradicionais

Foram até aqui apresentadas nesta secção uma variedade de aplicações não completamente satisfeitas com a tecnologia actual de SGBDs. Serão seguidamente enumerados os seus requisitos em termos quer de SGBDOs, quer de SGBDs tradicionais, estes últimos por forma a não serem esquecidos na especificação dos SGBDOs.

SGBDOs

Os requisitos em termos de SGBDOs que se identificam a partir das aplicações anteriormente analisadas são:

- *Identificadores de objectos únicos:* A capacidade de criação automática de identificadores para objectos representados na base de dados é importante, especialmente em aplicações de engenharia e desenho, nas quais os objectos possuem identificadores sem qualquer significado para nós.
- *Agregação de objectos:* Muitas aplicações requerem a possibilidade de definição de objectos que contêm outros objectos. Os objectos compostos poderão ser úteis também noutras aplicações (hierarquia de agregações).
- *Referências e integridade referencial:* É muito importante a possibilidade de referenciar um objecto a partir de outro, e esperar que seja o SGBD a automaticamente manter a integridade dessas referências. As referências são de uma forma geral importantes para muitos tipos de aplicações, embora o sejam menos nas aplicações comerciais. Este é um aspecto que não é eficientemente implementado na maioria dos SGBDs tradicionais.
- *Hierarquia de tipos de objectos:* A possibilidade de definição de novos tipos de objectos com base num tipo já existente, adicionando, restringindo ou redefinindo alguma informação ou comportamento é importante para todas as aplicações que utilizam estruturas de dados complexas (hierarquia de generalizações).
- *Encapsulamento:* Devido à importância de conceitos tais como reutilização, interoperabilidade e modularidade, é importante que haja a possibilidade de associar procedimentos a tipos de objectos. Apesar de as linguagens de programação orientada por objectos (LPOO) suportarem este conceito, os SGBDOs deverão também fazê-lo, por forma a aproximar os modelos das bases de dados e das linguagens de programação.
- *Conjuntos ordenados e referências:* Quase todas as aplicações apresentadas necessitam de ordenar as relações para um objecto, e ter como atributos de um objecto não só conjuntos de valores, mas também listas de valores. É dificil representar conjuntos ordenados em bases de dados relacionais.
- *Grandes blocos de dados:* A gestão de grandes blocos de informação, tais como imagens ou texto, armazenados como valores de um atributo, é uma capacidade

reconhecida como muito útil para muitas aplicações; muitos SGBDs tradicionais incluem já esta capacidade.

- *Acesso remoto eficiente:* Acesso transparente a bases de dados armazenadas em outras máquinas de uma rede é algo desejável para a quase totalidade das aplicações de engenharia e projecto baseadas em estações de trabalho. Apesar dos SGBDs tradicionais permitirem o acesso remoto, não o fazem contudo de forma eficiente para transacções de longa duração.

- *Facilidade de alteração do esquema:* Em quase todas as aplicações é importante a possibilidade de alterar o esquema com o mínimo de impacto nas aplicações já desenvolvidas, bem como facilitar a migração de dados antigos para os novos esquemas.

- *Interface com linguagem de programação:* A integração das linguagens de base de dados do SGBD com a linguagem de programação é útil e desejável, pois permite diminuir a impedância semântica entre ambos os sistemas. Os SGBDs tradicionais utilizam diferentes linguagens de base de dados e de programação o que obriga a conversão de dados entre as duas linguagens.

- *Múltiplas versões de bases de dados:* A possibilidade de manter mais do que uma versão de determinados objectos ou de uma base de dados inteira é muito importante nas aplicações de projecto, por motivos de manutenção simultânea de diferentes versões ou para coordenação de subprojectos a decorrer em paralelo.

- *Locks e Transacções de Longa Duração (mecanismos checkout-checkin):* A possibilidade de bloquear o acesso a blocos de informação a diferentes níveis de granularidade, desde um simples objecto a um projecto inteiro, é importante para muitas das aplicações anteriormente apresentadas.

- *Granularidade versus Performance:* Devido ao acesso concorrente a pequenos itens de informação ser pequeno nas novas aplicações, a granularidade dos *locks* e a performance de tratamento associada são aspectos importantes. Nas novas aplicações é mais vulgar um único utilizador pretender ter acesso exclusivo a grandes blocos de informação, do que muitos utilizadores pretenderem aceder concorrentemente a um mesmo pequeno item de informação, aliás como acontece por exemplo no processamento de transacções. Por esta razão, em termos de performance, os SGBDOs deverão dedicar maior atenção ao primeiro caso do que ao segundo, ou então dar a possibilidade de escolha da granularidade pretendida.

SGBDs Tradicionais

Os requisitos em termos de SGBDs tradicionais são:

- *Capacidade de Armazenamento de Grandes Volumes de Informação:* A capacidade de armazenamento de grandes volumes de informação persistente é importante para todas as aplicações que foram apresentadas.

- *Linguagem de Interrogação:* Uma linguagem de interrogação simples que permita ao utilizador formular interrogações para examinar a base de dados, apesar de ser de menor importância para algumas das aplicações, é no entanto útil. Compatibilidade

com a norma SQL assume maior importância nas aplicações comerciais, mas é também um ponto a ter em consideração nos SGBDOs.

- *Desenvolvimento de Interfaces:* A maior parte dos SGBDs comerciais oferecem utilitários para a definição de formulários para possibilitar a edição e visualização de campos da base de dados. Estas facilidades são também de importância para as nossas aplicações.

- *Bases de Dados Distribuídas:* A capacidade de transparentemente manipular bases de dados armazenadas em diferentes máquinas, é muito útil para uma grande variedade de aplicações.

- *Execução Automática de Procedimentos:* Muitas aplicações requerem que certas tarefas sejam executadas sempre que uma determinada condição seja verificada. Este mecanismo de regras/gatilhos é importante em aplicações de tempo-real, e em algumas aplicações de projecto.

- *Concorrência:* Nas aplicações comerciais tradicionais, pretende-se a possibilidade de fazer *locks* a pequenas partes da base de dados, limitando o acesso a outros utilizadores. Por outro lado, nas novas aplicações, os utilizadores normalmente fazem um *lock* a uma grande parte da base de dados e por um grande período de tempo, não tolerando a sobrecarga devida à tão baixa granularidade dos *locks* dos SGBDs tradicionais. No entanto, por vezes também é necessária tal granularidade.

- *Recuperação de Erros:* A possibilidade de recuperação do estado de uma base de dados, em consequência de erros de software ou hardware, é algo que é obviamente sempre desejado. No entanto, em muitos casos o custo em termos de performance não é compensador.

- *Independência de Dados:* É essencial garantir independência entre a informação armazenada na base de dados e as aplicações que a utilizam. Nas bases de dados relacionais, esta independência é conseguida através de vistas. Nas bases de dados orientadas por objectos, esta capacidade é oferecida pelos procedimentos que são os únicos atributos externamente visíveis.

Vimos que a maioria das capacidades dos SGBDs tradicionais são importantes, embora o sejam menos nas aplicações a que se destinam os SGBDOs. É no entanto importante que os SGBDOs continuem a oferecer estas capacidades adicionalmente às novas anteriormente referidas.

2.2 Novos Modelos de Dados

Numa base de dados existem dois tipos de informação: os dados e o esquema dos dados. O esquema constitui a meta-informação que descreve a estrutura dos dados armazenados. O conceito análogo nas linguagens de programação são as declarações de tipos de variáveis. Os dados são no entanto o essencial da base de dados, e dependendo do SGBD, poderão ir desde simples inteiros e caracteres até a estruturas de dados complexas. As restrições que os SGBDs impõem em termos de especificação de tipos de dados e definição de esquemas depende do Modelo de Dados que implementam.

Os modelos de dados mais importantes a considerar para o presente trabalho são:

- o modelo relacional;
- os modelos relacionais estendidos;
- os modelos orientados por objectos;
- e os modelos funcionais e semânticos.

Vamos analisar a seguir quais as principais características de cada um destes modelos.

2.2.1 Revisão do Modelo Relacional

O modelo relacional é bastante importante, devido à sua popularidade quer junto da comunidade cíentifica quer na área comercial, onde superou com enorme sucesso sistemas baseados nos modelos seus antecessores. Por conseguinte, o modelo relacional constitui um bom ponto de partida para o desenvolvimento de novos sistemas, servindo como um bom termo de comparação com sistemas mais avançados.

A popularidade do modelo relacional deve-se sobretudo à sua simplicidade. Existe apenas uma estrutura de dados: uma tabela com linhas e colunas contendo dados de diversos tipos, tais como inteiros e caracteres. A linguagem de interrogação baseia-se em poucas operações sobre tabelas, e as características mais complexas, tais como integridade referencial e definições de vistas, não precisam ser entendidas pelo utilizador final.

A simplicidade do modelo relacional traduz-se na sua facilidade de utilização e tratamento matemático, mas é também uma limitação. Como pudemos já constatar nas muitas aplicações descritas na secção anterior, certas aplicações manipulam informação com estruturas demasiadamente complexas para poderem ser representadas em tabelas. Os SGBDOs são concebidos por forma a obviar estas limitações, embora à custa de um aumento da complexidade do modelo de dados.

2.2.2 Extensões do Modelo Relacional

Uma forma óbvia de remediar as limitações do modelo relacional é estendendo o modelo. Os Modelos Relacionais Estendidos são usados em evoluções de sistemas baseados no modelo relacional por forma a incorporar procedimentos, objectos, versões, e outras novas capacidades. Uma vez que os sistemas relacionais são simples, bem entendidos e populares, tanto nas áreas de investigação como na indústria de software, os sistemas relacionais estendidos herdam estas características de popularidade.

Existem vários modelos relacionais estendidos, cujas características dependem de como foram estendidos. Todos eles, no entanto, suportam as mesmas tabelas relacionais e linguagem de interrogação, incorporam algum conceito de "objecto", e todos eles possibilitam o armazenamento quer de dados quer de procedimentos.

2.2.3 Modelos Orientados por Objectos

Os sistemas baseados nos modelos orientados por objectos têm a sua origem no paradigma de programação orientada por objectos (POO). As linguagens de programação que originaram e popularizaram o paradigma de orientação por objectos foram o Simula e o Smalltalk [Cat91].

O paradigma de POO incorpora o conceito de tipos abstractos de dados (TAD) nas linguagens de programação. Uma declaração de um tipo abstracto de dados define explicitamente a parte pública e privada da estrutura de dados, ou objecto. Os TADs nas linguagens orientadas por objectos—as classes, encapsulam a informação privada do objecto com procedimentos públicos—os métodos. O encapsulamento simplifica a construção e manutenção dos programas porque incentiva a modularização. Um objecto é algo que tem identidade e que pode ser construído e modificado independentemente do resto do sistema, desde que se mantenha inalterada a definição da sua interface pública.

Porque não existe apenas um paradigma de orientação por objectos, existe uma variedade de modelos de dados orientados por objectos. Para além do encapsulamento, as LPOO têm em comum conceitos tais como hierarquias de tipos de objectos, com herança de atributos e métodos. No entanto, as LPOO diferem em muitas características específicas, inclusivé no âmbito do conceito de encapsulamento.

Um sistema de gestão de bases de dados que implemente um modelo de dados de objectos denomina-se de SGBD de Objectos (SGBDO), muito embora vários autores utilizem este termo para sistemas de bases de dados que não suportam sequer o conceito de encapsulamento [Cat91].

2.2.4 Modelos Funcionais e Semânticos

Os modelos funcionais são uma forma particularmente elegante de representação de bases de dados. Eles fazem recurso a uma linguagem de acesso aos dados baseada na notação funcional matemática, e oferecem para definição dos valores das funções uma linguagem de interrogação declarativa e funcional análoga às linguagens de interrogação relacionais. Se a linguagem é estendida para permitir uma especificação procedimental das funções, apelidamos o SGBD de Sistema de Gestão de Bases de Dados Funcional Estendido.

As bases de um modelo funcional são os objectos e as funções. As funções mapeam objectos noutros objectos e valores. As relações entre objectos, atributos de objectos, e procedimentos associados a objectos são todos representados por funções. Merece a pena estudar o modelo funcional devido às vantagens que oferece, das quais se destaca a simplicidade da notação e capacidade de definição de informação virtual.

Outros modelos foram propostos nos anos 80, nomeadamente modelos de dados semânticos, assim chamados por incorporarem mais semântica do que o modelo relacional. Estes modelos contribuíram com vários conceitos novos, muitos dos quais foram depois incorporados noutros modelos.

2.2.5 Modelos de Dados e Arquitecturas de Bases de Dados

Os novos modelos de dados—relacional estendido, funcional, semântico e orientado por objectos, oferecem uma funcionalidade semelhante, sendo por esta razão por vezes chamados conjuntamente de "modelos de dados semânticos" pois incorporam mais semântica dos dados do que a abordagem relacional. O termo "orientado por objectos" tem por outro lado também sido usado para referir todos estes modelos, pela razão de que todos eles, duma forma ou de outra, incorporam conceitos oriundos do paradigma de orientação por objectos.

A conclusão a tirar daqui, é a de que os diversos modelos estão a convergir para um único modelo que incorporará uma combinação das características de todos eles. Devido à convergência das diferentes abordagens, uma classificação dos sistemas com base no modelo de dados não deverá ser levada muito a sério.

Um outro aspecto que permite fazer a distinção entre os vários sistemas de bases de dados é a sua arquitectura, isto é, a forma como o SGBDO é implementado e combinado com a linguagem de programação das aplicações.

2.3 Linguagens de Programação Orientada por Objectos

A programação orientada por objectos (POO) tem como base um conjunto de princípios a que colectivamente se dá o nome de Modelo Objecto. O Modelo Objecto engloba os princípios de abstracção, encapsulamento, modularidade, hierarquia, tipos, concorrência e persistência. Nenhum destes princípios em si é novo, mas a sua utilização conjunta é de facto nova [Boo91].

A POO é fundamentalmente diferente da programação estruturada tradicional. Requer uma maneira diferente de decomposição e dá origem a arquitecturas de software bem diferentes das resultantes das metodologias tradicionais.

2.3.1 Evolução do Modelo Objecto

Os sistemas de software são cada vez maiores e mais complexos do que eram os seus predecessores de apenas há alguns anos atrás. A necessidade crescente de construção de sistemas de software complexos tem estado na origem de avanços na engenharia de software, nomeadamente em questões relacionadas com decomposição, abstracção e hierarquias.

O desenvolvimento de linguagens de programação com crescente capacidade de expressão tem sido um dos resultados visíveis destes sucessivos avanços.

Gerações das Linguagens de Programação

As linguagens de programação de alto nível, de acordo com as capacidades que introduziram, podem ser agrupadas nas seguintes gerações:

- *Linguagens de 1º Geração (1954-1958):* FORTRAN I, ALGOL 58, Flowmatic; expressões matemáticas;
- *Linguagens de 2º Geração (1959-1961):* FORTRAN II, ALGOL 60, COBOL, Lisp; subrotinas, compilação separada; estrutura de blocos, tipos de dados; descrição de dados, ficheiros; processamento de listas, apontadores;
- *Linguagens de 3º Geração (1962-1970):* PL/1, ALGOL 68, Pascal, Simula;classes, abstracção de dados.
- *última Geração (1970-1980)*

Ao longo das várias gerações, o tipo de mecanismo de abstracção que cada linguagem suporta evoluiu. As linguagens da 1^a geração eram usadas principalmente para aplicações cientificas e de engenharia, aplicações essas em que o vocabulário do domínio do problema era essencialmente matemático.

Nas linguagens da 2^a geração foi dada maior ênfase às abstracções algorítmicas. O *hardware* pelo facto de se ter tornado mais poderoso, proporcionou que mais tipos de problemas começassem a poder ser automatizados. O objectivo nesta altura consistia em instruir o computador sobre o que fazer. Tal como a anterior, esta nova geração de linguagens de programação de alto nível fez com que o software se aproximasse mais do domínio do problema e o libertasse mais da especificidade da máquina.

Por volta do final dos anos 60, os computadores aumentaram a sua capacidade de processamento quase exponencialmente, devido ao aparecimento dos transístores e

circuitos integrados a preços muitissimo mais baixos. Surgem então novas linguagens de programação com capacidade de manipulação de mais tipos de informação e que permitem ao programador a descrição de tipos de dados mais complexos, cuja implementação é deixada ao cuidado da linguagem de programação.

Nos anos 70 muitas linguagens foram inventadas, mas poucas atingiram a maturidade. Fruto duma enorme pesquisa no campo das linguagens de programação, são criadas perto de duas mil linguagens novas diferentes. A necessidade de programas cada vez maiores, evidenciou as insuficiências das linguagens anteriores e por conseguinte, deu origem ao desenvolvimento de muitos mecanismos novos que obviassem estas limitações. Muitas destas linguagens não sobreviveram. No entanto, os conceitos por elas introduzidos encontraram o seu caminho nas sucessoras das primeiras linguagens.

Assim, por exemplo, temos hoje as linguagens Ada (um sucessor do ALGOL 68 e do Pascal, com contribuições de Simula e outras) e C++ (derivado do casamento de C e Simula).

Topologia das Linguagens Orientadas por Objectos

A natureza das abstracções que podem ser conseguidas através do uso de procedimentos é suficiente para a descrição de operações abstractas, mas não é de forma alguma suficiente para a descrição de objectos abstractos [Sha84]. Em desenvolvimentos recentes de várias linguagens de programação, como Smalltalk, C++, e Eiffel, nota-se a influência destas ideias. A construção fundamental nestas linguagens é o módulo, representativo de uma colecção lógica de classes e objectos, em vez de subprogramas como acontecia nas linguagens anteriores. Adicionalmente, existe pouca ou mesmo nenhuma informação global. Dados e operações são organizados de forma que as peças fundamentais dos nossos sistemas deixam de ser os algoritmos, para dar lugar às classes e objectos. Em sistemas de grande dimensão, encontramos conjuntos de abstracções construídas em várias camadas que se sobrepõem. A qualquer nível de abstracção, podemos encontrar colecções significativas de objectos que cooperam entre si para atingir algum comportamento superior. Se olharmos para dentro dum destes conjuntos, para ver uma implementação, descobrimos um outro conjunto de abstracções cooperantes, e assim sucessivamente.

Origem do Modelo Objecto

Os métodos de concepção estruturada foram desenvolvidos para auxiliar os engenheiros de software na construção de sistemas complexos utilizando os algoritmos como peças fundamentais. Analogamente, os métodos de concepção orientada por objectos foram desenvolvidos para ajudar os engenheiros de software na exploração da capacidade de expressão das linguagens de programação orientada por objectos, utilizando as classes e objectos como as suas peças fundamentais.

Na realidade, o Modelo Objecto foi influenciado por inúmeros factores, e não apenas pela programação orientada por objectos. Tal como já foi anteriormente dito, o Modelo Objecto provou ser um conceito que unifica a ciência da computação, aplicável não só às linguagens de programação, mas também à concepção de interfaces homem-máquina, bases de dados, bases de conhecimento, e mesmo arquitecturas de computadores.

A concepção orientada por objectos aparece como fruto duma evolução, e não como um conceito revolucionário; não rejeita os avanços conseguidos no passado, antes porém baseia-se nalguns deles. No entanto, a complexidade que conseguimos manipular utilizando uma

decomposição algoritmica tem os seus limites, surgindo então a necessidade de utilizar uma decomposição orientada por objectos.

Análise, Concepção e Programação Orientada por Objectos

Devido ao Modelo Objecto ser fruto de várias influências, a terminologia que o tem acompanhado é imprecisa. Um programador em Smalltalk diz-se que utiliza métodos, em C++ diz-se que utiliza funções membras, e em Eiffel características, etc. Para minimizar a confusão, vamos propôr uma definição do que é orientado por objectos e o que não é.

A primeira coisa que podemos concordar é a de que o conceito de objecto é central a tudo o que é orientado por objectos. Um objecto é uma entidade palpável que exibe um comportamento bem definido, e unifica as ideias de algoritmos e abstracção de dados. Os objectos têm uma determinada integridade que não deve, e de facto, não pode, ser violada. Um objecto só pode alterar o seu estado, comportar-se, ser manipulado, ou relacionar-se com outros objectos de forma apropriada. Por outras palavras, existem propriedades invariantes que caracterizam um objecto e o seu comportamento, e que se manifestam durante toda a sua vida.

Programação Orientada por Objectos (POO)

Programação orientada por objectos é um método de implementação no qual os programas são organizados como colecções de objectos que entre si cooperam, cada um dos quais representando uma instância duma determinada classe, classes estas organizadas numa hierarquia que as interliga por relações de herança [Boo91].

Desta definição importa realçar três aspectos:

1. A POO utiliza objectos e não algoritmos, como bloco lógico fundamental (hierarquia de agregações);
2. Cada objecto é uma instância de uma classe;
3. As classes encontram-se ligadas por relações de herança (hierarquia de generalizações ou especializações).

A respeito de linguagens orientadas por objectos, Stroustrup sugere que "se linguagem orientada por objectos significa algo, então significa uma linguagem que tem mecanismos que suportam bem um estilo de programação orientado por objectos .... Diz-se que uma linguagem suporta bem um estilo de programação se tem facilidades que tornem conveniente o uso desse estilo" [Str88].

Por outro lado diz-se que uma linguagem é orientada por objectos se e só se satisfaz os seguintes requisitos:

- Suporta objectos que constituem abstracções de dados e que têm uma interface de operações e um estado local secreto;
- Os objectos têm um tipo associado (classe);
- Os tipos (classes) podem herdar atributos de supertipos (superclasses).

Numa linguagem que suporte herança é possível exprimir relações de generalização entre classes, tais como: uma rosa vermelha é um tipo de rosa, uma rosa é um tipo de flôr, e uma flôr é um tipo de planta. Se uma linguagem não fornece suporte directo para o mecanismo de herança, então não é orientada por objectos. Há quem distinga estas linguagens e as denomine de baseadas em objectos em vez de orientada por objectos. De acordo com esta definição, Smalltalk, C++ e Eiffel são orientadas por objectos e Ada é baseada em objectos.

O objectivo dos métodos de programação consiste no uso correcto e efectivo dos mecanismos particulares de uma determinada linguagem. Por outro lado, os métodos de concepção incentivam a estruturação correcta e efectiva dum sistema complexo.

Concepção Orientada por Objectos (COO)

A concepção orientada por objectos é um tipo de concepção que consiste numa de composição orientada por objectos e utiliza uma determinada notação para ilustrar os modelos lógico e físico, estático e dinâmico do sistema em consideração.

O que diferencia uma COO duma concepção estruturada é o facto da primeira permitir uma decomposição orientada por objectos: a primeira utiliza abstracções de classes e objectos para estruturar logicamente os sistemas, enquanto que a última utiliza abstracções algorítmicas.

Análise Orientada por Objectos (AOO)

Por fim, a Análise Orientada por Objectos é um tipo de análise que incentiva a construção de modelos do mundo real, utilizando uma visão do mundo orientada por objectos. AOO é um método de análise, que examina os requisitos do ponto de vista de classes e objectos encontrados a partir do vocabulário do domínio do problema. Como é que AOO, COO, e POO se relacionam? Basicamente, os produtos da AOO servem como modelos a partir dos quais podemos iniciar uma COO; os produtos da COO podem por sua vez então ser utilizados como linhas mestre para a implementação completa do sistema recorrendo a métodos de POO.

2.3.2 Elementos do Modelo Objecto

Existem vários estilos de programação, que adoptando tipos de abstracções diferentes orientam na implementação das aplicações. No entanto, não há nenhum estilo de programação que possa ser considerado o melhor para todos os tipos de aplicações. O estilo de programação orientada por objectos, é tido como o mais bem sucedido para a maioria das aplicações, nomeadamente em sistemas de software à escala industrial, nos quais a complexidade é um aspecto dominante.

Cada um destes estilos de programação baseia-se num contexto conceptual próprio, e requer uma mentalização diferente, isto é, uma maneira diferente de pensar sobre o problema. Para tudo que é orientado por objectos, o contexto conceptual é o Modelo Objecto. Segundo Booch, há quatro elementos deste modelo considerados de *maior* importância, e que são: abstracção, encapsulamento, modularidade, hierarquia. Por elementos de *maior* importância, entenda-se aqueles que são indispensáveis a um modelo que pretenda ser considerado de objectos. Analogamente, existem outros três elementos do Modelo Objecto considerados de *menor* importância, e que são: tipos, concorrência, persistência.

Por elementos de menor importância, entenda-se aqueles que apesar de serem úteis, não constituem, no entanto, partes essenciais do Modelo Objecto.

Abstracção

Define-se *abstracção* da seguinte maneira:

> *A abstracção salienta as características dum objecto consideradas essenciais que o distinguem de todos os outros tipos de objectos, fornecendo portanto na perspectiva do observador contornos muito bem definidos [Boo91].*

Uma abstracção encerra em si todo o comportamento importante de um objecto, nem mais nem menos. A escolha do conjunto certo de abstracções para um determinado domínio é o problema central na COO.

Maior preocupação deve ser posta na construção de abstracções de entidades, porque elas reflectem o vocabulário do domínio dum determinado problema.

Todas as abstracções tanto têm *propriedades* estáticas como dinâmicas. Por exemplo, um ficheiro ocupa um determinado espaço num certo periférico de armazenamento; tem um nome, e tem conteúdo. Todas estas propriedades são propriedades estáticas. O valor de cada uma destas propriedades é uma propriedade dinâmica relativamente à vida do objecto: um ficheiro pode aumentar ou diminuir de tamanho, o seu nome pode ser alterado, o seu conteúdo pode ser modificado, etc.

Num estilo de programação estruturada, a actividade que muda o valor dinâmico dos objectos é a parte central de todos os programas: as coisas acontecem quando os subprogramas são chamados e as instruções são executadas. Num estilo de programação orientada por objectos, as coisas acontecem sempre que operamos sobre um objecto (na terminologia do Smalltalk, sempre que mandamos uma mensagem a um objecto). Invocar uma operação sobre um objecto dá origem a uma reacção por parte dele. Quais as operações aplicáveis a um determinado objecto, e como é que ele reage a elas constitui o seu *comportamento*.

Encapsulamento

A abstracção dum objecto deve preceder a tomada de decisões sobre a sua implementação. Uma vez seleccionada uma implementação, deve ser tratada como algo secreto da abstracção e ocultada à maioria de quem a utiliza. O encapsulamento permite que um programa seja alterado com fiabilidade e pouco esforço.

A abstracção e encapsulamento são conceitos que se complementam: a abstracção incide sobre a visão exterior dum objecto, ao passo que o encapsulamento evita que os seus utilizadores conheçam o seu interior e, por conseguinte, a implementação do seu comportamento. Desta forma é imposta uma barreira entre níveis de abstracção diferentes: o nível lógico e o nível físico da representação.

Em termos práticos, uma classe deve ter então duas partes: uma interface e uma implementação. A interface da classe capta apenas a sua visão externa. A implementação duma classe compreende a representação da abstracção, bem como os mecanismos que originam o comportamento desejado. Uma classe diz-se *abstracta* se não apresentar implementação para algumas das propriedades ou operações definidas na sua interface.

A figura 2.3 ilustra esta separação da interface e implementação.

Resumindo, define-se encapsulamento da seguinte maneira:

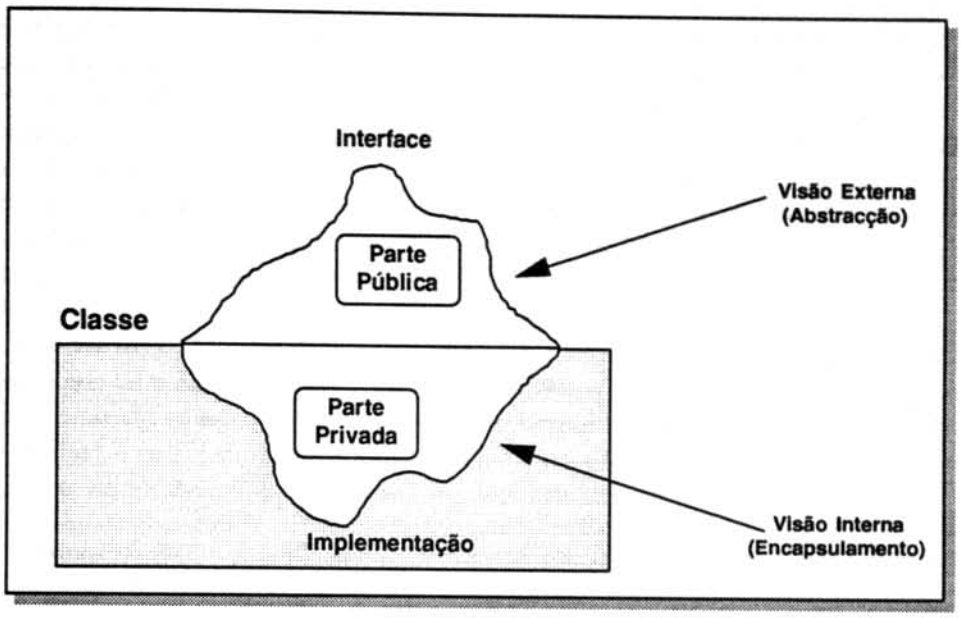


Figura 2.3: A visão interna e externa de uma classe

Encapsulamento é o processo de esconder todos os detalhes dum objecto que não contribuem para a definição das suas características essenciais [Boo91].

Modularidade

O acto de dividir um programa em componentes pode contribuir de forma significativa para a redução da sua complexidade. Em algumas linguagens, tal como o Smalltalk, não existe o conceito de módulo, e por conseguinte, a classe é a única unidade física da decomposição. Especialmente para grandes aplicações, nas quais podemos ter muitas centenas de classes, a ajuda dada pela utilização de módulos é essencial para o domínio da complexidade.

A maioria das linguagens que suportam o módulo como um conceito separado, fazem distinção entre a interface e a implementação do módulo. A modularidade e o encapsulamento são conceitos diferentes mas que andam lado a lado.

Decidir qual o conjunto certo de módulos para um determinado problema é quase tão difícil como decidir qual o conjunto certo de abstracções. Casos há, no entanto, em que uma divisão em módulos feita de forma arbitrária tem consequências piores do que se nenhuma modularização tivesse sido feita.

Os módulos têm que ser vistos como recipientes físicos nos quais nós declaramos as classes e objectos da nossa concepção. Numa concepção estruturada tradicional, a modularização preocupa-se principalmente com o agrupar dos subprogramas. Numa concepção orientada por objectos, o problema é diferente: a tarefa é decidir onde empacotar fisicamente as classes e os objectos da estrutura lógica da concepção, o que é distintamente diferente de agrupar subprogramas.

O objectivo principal da decomposição em módulos é a redução do custo do software ao permitir que os módulos sejam concebidos e revistos independentemente. Cada estru-

tura dum módulo deve ser suficientemente simples por forma a poder ser completamente percebida; deve ser possível alterar a implementação de módulos, sem conhecimento da implementação de outros módulos e sem afectar o comportamento de outros módulos; a maior ou menor facilidade em levar a cabo uma alteração na concepção deve de alguma forma estar relacionada com a maior ou menor necessidade de essa alteração ser feita. Na prática, o custo de recompilar o corpo dum módulo é relativamente pequeno. No entanto, o custo de recompilar a interface dum módulo é relativamente grande. Especialmente em linguagens com verificação estática de tipos, torna-se necessário compilar a interface do módulo, o seu corpo, todos os outros módulos que dependem desta interface, os módulos que dependem destes módulos, e assim sucessivamente.

É necessário estabelecer um compromisso entre o desejo de encapsular as abstracções, e a necessidade de tornar algumas abstracções visíveis a outros módulos. Em suma, devemo-nos empenhar em construir módulos que sejam coesos (agrupando abstracções logicamente relacionadas) e com pequenas ligações (minimizando as dependências entre módulos).

Pode-se assim definir modularidade da seguinte maneira:

> *Modularidade é a propriedade dum sistema que foi decomposto num conjunto de módulos coesos e fracamente interligados [Boo91].*

Conclui-se que os principios de abstracção, encapsulamento, e modularidade são sinergéticos, isto é, contribuem para um mesmo fim. O objecto fornece um contorno muito bem definido à volta duma abstracção, e tanto o encapsulamento como a modularidade colocam barreiras à volta desta abstracção.

A identificação das classes e objectos faz parte da concepção lógica do sistema, mas a identificação dos módulos já faz parte da concepção física do sistema. Não se pode tomar todas as decisões ao nível da concepção lógica, antes de tomar as correspondentes ao nível físico, e vice-versa, pelo que estas decisões têm que ir sendo tomadas duma forma iterativa.

Hierarquia

A abstracção é algo de bom, mas em todas as aplicações, à excepção das mais triviais, podemos encontrar muitas mais abstracções diferentes do que aquelas que simultâneamente podemos compreender. O encapsulamento ajuda a dominar esta complexidade escondendo o interior das abstracções. A modularidade também ajuda, dando-nos uma maneira de agrupar abstracções logicamente relacionadas. Mas estes elementos ainda não são suficientes. Um conjunto de abstracções geralmente forma uma hierarquia, e identificando estas hierarquias na nossa concepção, simplificamos de forma significativa a nossa compreensão do problema. Define-se hierarquia da seguinte maneira:

> *Hierarquia é um conjunto ordenado de abstracções [Boo91].*

As duas hierarquias mais importantes num sistema complexo são a sua estrutura de classes (a hierarquia de generalizações) e a sua estrutura de objectos (a hierarquia de agregações).

A *herança* é a mais importante hierarquia de generalizações e constitui um elemento essencial dos sistemas orientados por objectos. Basicamente, a herança define uma relação entre classes, em que uma classe partilha a estrutura ou comportamento definido numa ou mais classes (herança simples ou múltipla, respectivamente).

A herança representa portanto uma hierarquia de abstracções, na qual uma subclasse herda duma ou mais superclasses. Típicamente, uma subclasse acrescenta ou redefine a estrutura ou comportamento das suas superclasses.

À medida que desenvolvemos a nossa hierarquia de herança, as estruturas e comportamentos comuns a diferentes classes tendem a deslocar-se para superclasses comuns. É por esta razão que se fala de herança como uma hierarquia de generalização/especialização. Superclasses representam abstracções generalizadas, e subclasses representam especializações nas quais são adicionados, modificados, ou mesmo escondidos, propriedades e métodos das superclasses.

Para uma determinada classe, existem normalmente dois tipos de clientes: objectos que invocam operações sobre as instâncias duma classe, e subclasses que herdam dessa classe.

Através da herança, o encapsulamento pode ser violado numa de três maneiras: a subclasse pode aceder a uma variável duma instância das suas superclasses, chamar uma operação privada das suas superclasses, ou referir directamente as superclasses das suas superclasses. Diferentes linguagens de programação tratam este compromisso entre encapsulamento e herança de diferentes maneiras. É no entanto, a linguagem C++, entre as linguagens orientadas por objectos de maior divulgação, aquela que oferece maior flexibilidade [Str88, Mey88]. Especificamente, a interface duma classe pode ter três partes:

- *parte privada*, onde são feitas as declarações do que é apenas visível à própria classe;
- *parte protegida*, onde são feitas as declarações do que é apenas visível à classe e suas subclasses;
- *parte pública*, para tudo que é visível a todos os clientes.

Enquanto a herança denota as relações generalização/especialização entre classes, as estruturas de objectos descrevem relações de agregação entre os objectos.

Tipos

O conceito de tipo é proveniente da teoria de tipos abstractos de dados. Um tipo é uma caracterização precisa das propriedades estruturais e comportamentais partilhadas por uma colecção de entidades.

> *Os tipos são a forma de ligar uma classe a um objecto, de tal modo que objectos de diferentes tipos não possam ser trocados entre si, ou no máximo, possam ser trocados apenas de formas controladas [Boo91].*

Os tipos permitem-nos exprimir as nossas abstracções de tal modo que a linguagem de programação na qual nós as implementamos reforce as nossas decisões de concepção. Este elemento é considerado como um elemento de menor importância, porque uma determinada linguagem de programação pode ter atribuição estática de tipos, atribuição dinâmica de tipos, ou mesmo não ter tipos, e continuar a ser chamada de orientada por objectos ou baseada em objectos.

O recurso a linguagens com atribuição estática de tipos tem as seguintes vantagens:

- Sem atribuição de tipos, um programa em execução pode misteriosamente terminar com erro devido a operações efectuados com operandos de tipos incompatíveis.

- Na maioria dos sistemas, o ciclo de editar, compilar, e depuração de erros é tão aborrecido que uma detecção prévia dos erros se torna indispensável.
- As declarações de tipos ajudam a documentar os programas.
- A maioria dos compiladores consegue gerar código objecto mais eficiente se os tipos dos dados forem declarados.

Por outro lado, as linguagens sem atribuição de tipos oferecem uma maior flexibilidade, mas na prática, a segurança que se perde não justifica, especialmente na programação de grandes sistemas.

Os conceitos de atribuição estática de tipos e associação estática são completamente diferentes. A atribuição estática de tipos tem a ver com a consistência, enquanto que a associação estática tem a ver com a altura em que aos nomes são associados tipos. Associação estática significa que os tipos das variáveis e expressões são definidos na altura da compilação; associação dinâmica significa que os tipos de todas as variáveis e expressões só são conhecidos em execução. Porque atribuição e associação de tipos são conceitos independentes, uma linguagem pode ter atribuição e associação estática de tipos (Ada), atribuição estática e suportar associação dinâmica (C++), ou pura e simplesmente não ter atribuição estática de tipos e suportar associação dinâmica (Smalltalk).

O polimorfismo representa um conceito de teoria de tipos no qual um único nome pode referir objectos de classes diferentes, se relacionadas por superclasses comuns. O oposto ao polimorfismo é o monomorfismo, que podemos encontrar em todas as linguagens que têm simultâneamente verificação estática de tipos e associação estática.

O polimorfismo surge quando a herança e a associação dinâmica interactuam. O polimorfismo é talvez a capacidade mais poderosa das linguagens de programação orientada por objectos, logo a seguir ao suporte da abstracção, e é o que distingue a programação orientada por objectos da programação mais tradicional com tipos abstractos de dados.

Concorrência

Para alguns tipos de problemas, um sistema automatizado poderá ter que lidar com muitos e diferentes eventos simultâneamente. Cada programa tem pelo menos um processo associado, mas um sistema concorrente pode ter muitos mais: alguns transitórios, e outros que durarão todo o tempo de execução de um sistema.

A construção de um sistema de software grande, é por si só, um trabalho bastante árduo; mas a concepção de um sistema que contenha múltiplos processos é muitíssimo mais árdua, porque temos que nos preocupar com pormenores tais como possíveis situações de *deadlock*, *livelock*, exclusão mútua, e pré-condições de entrada em execução. O Modelo Objecto é, no entanto, apropriado para sistemas distribuídos, por implicitamente definir as unidades de distribuição e as entidades que comunicam.

Enquanto a programação orientada por objectos incide sobre a abstracção de dados, encapsulamento e herança, a concorrência incide sobre a abstracção de processos e a sua sincronização. O objecto é um conceito que unifica estas duas diferentes perspectivas: cada objecto pode representar também uma abstracção de um processo. A estes objectos chamam-se objectos *activos*. Num sistema baseado numa concepção orientada por objectos, podemos conceber o mundo como consistindo num conjunto de objectos que entre si

cooperam, alguns dos quais são activos e servem como centros de actividade independente. Surge assim a seguinte definição:

> *Concorrência é a propriedade que distingue um objecto activo de um outro que não o é [Boo91].*

Uma das realidades sobre a concorrência é que uma vez introduzida num sistema, deve-se ter em consideração a forma como os objectos activos sincronizam as suas actividades entre eles, bem como com outros objectos puramente sequenciais. Por exemplo, se dois objectos activos tentam enviar mensagens a um terceiro objecto, devemos usar algum mecanismo de exclusão mútua, de forma a que o estado do objecto que está a ser actuado, não seja corrompido quando os dois objectos activos tentam actualizar o seu estado simultâneamente. É neste ponto, que as ideias de abstracção, encapsulamento e concorrência interactuam.

Tanto a concorrência como a persistência são dois elementos especialmente importantes em sistemas de bases de dados.

Persistência

Um objecto em software ocupa algum espaço e existe durante um intervalo de tempo. Existem objectos, que vão desde os transitórios até aos objectos de uma base de dados, cuja existência não se limita ao tempo de execução dum programa. O espectro de persistência de objectos abrange o seguinte:

- Resultados transitórios na avaliação de expressões
- Variáveis locais na invocação de procedimentos
- Variáveis globais, objectos alocados dinâmicamente
- Dados que existem entre execuções de programas
- Dados que existem entre várias versões dum programa
- Dados que duram mais do que um programa

As linguagens de programação tradicionais apenas consideram os três primeiros tipos de persistência de objectos; os três últimos tipos de persistência são do domínio da tecnologia de bases de dados. Esta divisão dá origem a conflitos que resultam por vezes em situações muito estranhas: programadores que inventam esquemas *ad hoc* para armazenar objectos cujo estado deve ser preservado entre execuções de programas, e programadores de bases de dados que empregam mal a sua tecnologia para lidar com objectos de carácter transitório.

A unificação dos conceitos de concorrência e de objectos dá origem a linguagens de programação concorrente orientada por objectos. Duma maneira análoga, a introdução do conceito de persistência no Modelo Objecto, dá origem às bases de dados de objectos.

Esta unificação simplifica enormemente o desenvolvimento de certas aplicações, e em particular permite que um mesmo método de concepção seja utilizado quer para aplicações de bases de dados, quer para outras aplicações.

> *Persistência é a propriedade dum objecto através da qual a sua existência transcende o tempo (isto é, continua a existir após o seu criador ter deixado de existir) e/ou espaço (isto é, a localização do objecto muda-se do espaço no qual foi criado) [Boo91].*

A persistência lida com algo mais do que a vida dos dados. Em bases de dados de objectos, o que persiste não é apenas o estado do objecto, mas também a sua classe de tal modo que todos os programas interpretem da mesma maneira o estado armazenado.

Em sistemas distribuídos, enfrentamo-nos por vezes com problemas de persistência, não só persistência temporal mas também persistência espacial. Em tais sistemas, é útil poder-se pensar em objectos que se podem mover duma máquina para outra, e mesmo poder ter representações físicas diferentes em máquinas diferentes.

2.3.3 Benefícios do Modelo Objecto

Como tivemos oportunidade de verificar, o Modelo Objecto, modelo base das linguagens de programação orientada por objectos, é fundamentalmente diferente dos modelos seguidos pelos métodos mais tradicionais de análise estruturada, concepção estruturada, e programação estruturada. Isto não significa no entanto, que o Modelo Objecto abandona todos os principios e experiência conquistada por estes modelos mais antigos. Em vez disso, ele introduz muitos elementos novos que se baseiam nestes modelos mais antigos. O Modelo Objecto oferece um número de benefícios significativos que os outros modelos não suportam. Existem quatro benefícios práticos que advêm da aplicação do Modelo Objecto.

Primeiro, o uso do Modelo Objecto auxilia na exploração do poder expressivo de todas as linguagens de programação orientada por objectos.

Segundo, o uso do Modelo Objecto encoraja o reuso não apenas de software, mas também de concepções inteiras. Constata-se que os sistemas orientados por objectos são normalmente menores que os equivalentes com implementações não orientadas por objectos. Graças ao maior reuso, obtem-se menos código para escrever e manter, bem como benefícios em termos de custo e planeamento de produção.

Terceiro, o uso do Modelo Objecto produz sistemas que são construídos sobre formas intermédias estáveis, e por conseguinte mais flexiveis a alterações. Isto significa também que tais sistemas podem evoluir ao longo do tempo, em vez de serem abandonados ou completamente reconcebidos em resposta a grandes alterações nos requisitos.

Finalmente, o Modelo Objecto reduz o risco de desenvolvimento de sistemas complexos e aumenta a confiança na correcção da nossa concepção.

3

A Nova Tecnologia de Bases de Dados

Os novos sistemas de bases de dados aparecem com o objectivo de alargar a aplicabilidade da tecnologia de bases de dados a novas aplicações (secção 2.1). Baseando-se em novos modelos de dados (secção 2.2), têm como objectivo a integração das linguagens de programação (secção 2.3) com os sistemas de bases de dados (secção 3.1), contribuindo assim para a diminuição da impedância semântica actualmente existente entre linguagens de programação (LP) e linguagens de interrogação de bases de dados (LI).

Com o intuito de dar a conhecer as potencialidades dos novos Sistemas de Gestão de Bases de Dados de Objectos (SGBDOs), neste capítulo serão apresentados os principais conceitos que rodeiam tanto os sistemas tradicionais como os novos sistemas. No final do capítulo serão resumidamente enumeradas as características típicas oferecidas pelos SGBDOs.

3.1 Sistemas de Gestão de Bases de Dados Tradicionais

Com o objectivo de simplificar o desenvolvimento de aplicações com fortes necessidades de manipulação de dados, foram criados os Sistemas de Gestão de Bases de Dados (SGBDs). Os SGBDs, dependendo da funcionalidade e ferramentas que colocam ao dispôr do utilizador final, podem mesmo eliminar a necessidade de desenvolvimento.

De uma forma geral, quanto maior fôr a funcionalidade oferecida pelo SGBD, menor será a quantidade de código necessário escrever para implementar as aplicações. No entanto, funcionalidade desnecessária para uma determinada aplicação pode contribuir para a redução do desempenho do sistema, e dificultar a sua compreensão, pelo que uma avaliação criteriosa das necessidades da aplicação em termos de SGBDs é importante.

3.1.1 Perspectiva Histórica

Os primeiros SGBDs surgem no princípio da década de 70. As abordagens então mais populares para a construção de SGBDs tinham como base os modelos de dados hierárquico e reticulado.

A primeira abordagem, tipificada pelo sistema IMS [1] da IBM, possui um modelo de dados em que todos os registos de dados são organizados numa colecção de árvores. Cada uma das árvores é composta por um registo *raíz*, ao qual se encontram hierárquicamente ligados os restantes registos (ver figura 3.1). A linguagem de interrogação disponível nestes sistemas permite que o programador de uma aplicação, acedendo aos registos um a um, navegue desde os registos raíz até aos registos de interesse.

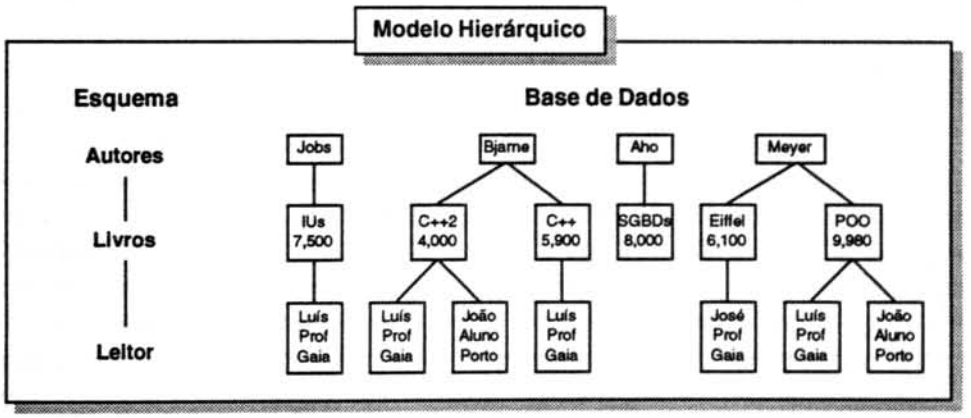


Figura 3.1: Modelo de dados hierárquico

A segunda abordagem, resulta das normas da CODASYL (Conference on Data Systems Languages), e o modelo de dados sugere que a colecção de registos da base de dados seja organizada sob a forma dum grafo dirigido (ver figura 3.2). Tal como na abordagem

[1] Information Management System

anterior, foi também definida uma linguagem de interrogação navegacional a partir da qual um programa pode-se mover de um registo de entrada específico até à informação pretendida.

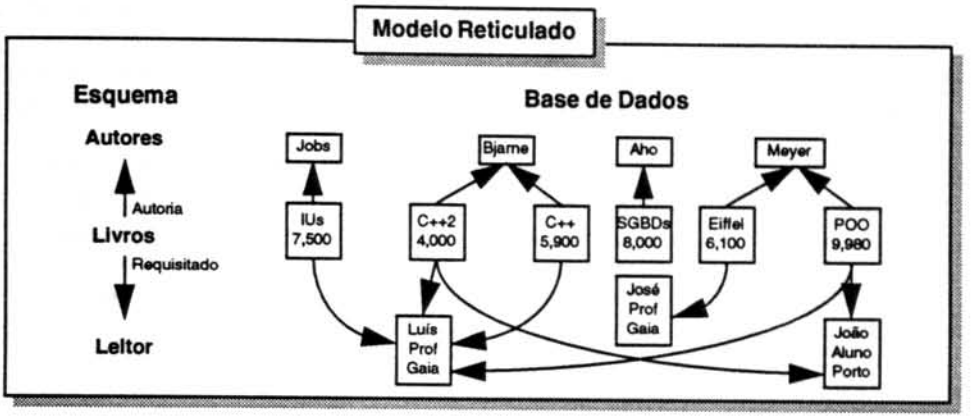


Figura 3.2: Modelo de dados reticulado

Ambas as abordagens, possuem diversas desvantagens, das quais se destacam o caracter navegacional das linguagens de interrogação, e a forte dependência dos programas à estrutura da base de dados, resultando na necessidade de alteração dos programas de aplicação sempre que esta é modificada. Como resultado, os sistemas de bases de dados dos anos 1970 eram de utilização difícil e manutenção dispendiosa, em grande parte devido ao baixo nível a que era feito o interface entre o programa de aplicação e o SGBD.

O modelo relacional, apresentado por Codd numa série de artigos em 1970-72, apresenta diferenças fundamentais relativamente às abordagens anteriores. Codd sugere que toda a informação seja conceptualmente representada por simples estruturas de dados tabulares (relações), e que o acesso aos dados seja feito através de uma linguagem de interrogação de alto-nível e declarativa (ou não-procedimental). Com este tipo de linguagem de interrogação, o programador em vez de ter de escrever um algoritmo para ir obtendo os registos desejados um a um, tem apenas de especificar um predicado que identifique os registos ou conjuntos de registos pretendidos. Um optimizador de interrogações do SGBD fará posteriormente a tradução da especificação do predicado para um algoritmo que se encarregará de fazer o acesso à base de dados por forma a dar resposta à interrogação formulada. Estas linguagens não-procedimentais são muito mais fáceis de utilizar do que as linguagens navegacionais do IMS e CODASYL, resultam numa maior produtividade de programação e facilitam o acesso directo à base de dados pelos utilizadores finais.

Quando o modelo relacional foi inicialmente proposto, era considerado uma elegante construção teórica, mas na prática apenas implementável por brincadeira, isto é, sem viabiliadde para ser utilizado em situações reais. Foi apenas após considerável investigação, na sua grande maioria sobre os fundamentos das bases de dados relacionais, que as implementações em larga escala foram conseguidas.

Durante os anos 1970, a investigação em sistemas de bases de dados incidiu extensivamente sobre os Sistemas de Gestão de Bases de Dados Relacionais (SGBDRs). Alguns dos mais importantes resultados alcançados foram:

- Criação de linguagens de interrogação relacional de alto-nível. Estudo aprofundado da teoria associada às linguagens de interrogação de alto-nível.
- Desenvolvimento da teoria e algoritmos para optimização de interrogações.
- Formulação da teoria da *normalização* para auxiliar o projecto de bases de dados, eliminando a redundância e anomalias lógicas da informação a armazenar.
- Construção de algoritmos para alocação de tuplos de uma relação em *páginas* de ficheiros em memória secundária, com vista à diminuição do tempo médio de acesso a esses tuplos.
- Construção de técnicas de indexação para permitir o acesso rápido a um ou mais registos.
- Implementação de protótipos de SGBDs, que constituíram o núcleo de muitos dos actuais SGBDRs comerciais.

Como resultado da actividade de investigação dos anos 1970, aparecem nos anos 1980 inúmeros produtos comerciais baseados no modelo relacional, muitos deles desenvolvidos com base nos protótipos dos projectos de investigação. Hoje em dia, os SGBDRs encontram-se disponíveis em praticamente todas as plataformas de hardware, desde o simples computador pessoal até ao poderoso mainframe, e constituem uma norma em todos os computadores dos anos 1990.

A sua grande divulgação deve-se em parte à grande simplicidade do modelo relacional, e à consequente superior facilidade de compreensão relativamente aos sistemas anteriores. A simplicidade do modelo relacional baseia-se em poucos conceitos: relações, linguagens de interrogação, e transacções. Estas capacidades libertam o utilizador de detalhes de organização física dos dados a baixo-nível, recuperação de erros, e coordenação entre utilizadores. O modelo relacional permite também um elevado grau de independência dos dados, e normalização ao nível de linguagem de interrogação e interface com diferentes SGBDs, devido à norma SQL—*Structured Query Language.*

A tecnologia que torna hoje possível a existência destes sistemas é um dos mais directos resultados da actividade de investigação realizada.

3.1.2 Conceitos de Bases de Dados

Os conceitos básicos de bases de dados, tais como, independência lógica e física dos dados, serão sucintamente apresentados [Dat90, Ull89a, Wie77].

Arquitectura de Três Níveis

A informação de um SGBD deve ser vista a três níveis diferentes de abstracção:

- *Interno:* A organização dos dados no suporte físico de armazenamento.
- *Conceptual:* O nível do modelo de dados, isto é, os tipos de registos e campos, e suas inter-relações.

- *Externo:* O significado atribuído aos dados pela aplicação, ou grupos de aplicações.

Ao nível interno é também por vezes dado o nome de nível físico, e aos outros dois, de níveis lógicos. A concepção de uma base de dados, isto é, o planeamento e criação de um esquema de base de dados é um processo que parte do nível externo e conceptual e termina no nível físico.

Independência dos Dados

As bases de dados, concretamente as relacionais, permitem que sejam feitas alterações na organização dos dados sem que isto obrigue à modificação dos programas que os utilizam. Esta independência é importante tanto ao nível interno como conceptual:

- *Independência Física* é a capacidade de alterar a organização interna dos dados, índices ou estrutura dos registos, sem ter de alterar as aplicações. Na arquitectura de três níveis atrás referida esta independência situa-se entre os níveis interno e conceptual.

- *Independência Lógica* é a capacidade de alterar o esquema dos dados— adicionando relações ou atributos—sem ser necessário modificar as aplicações. Na arquitectura de três níveis, esta independência situa-se entre os níveis conceptual e externo.

Os esquemas conceptual e interno de uma base de dados podem ter de ser modificados por muitas razões: necessidade de introdução de nova informação, reestruturação da informação para melhorar eficiência, etc. Dependendo do projecto, as alterações ao esquema da base de dados poderão ter de ser semanais, diárias ou ainda mais frequentes. Se as modificações do esquema da base de dados implicarem a alteração de todas as aplicações associadas, o processo torna-se bastante pesado, especialmente em projectos de grande dimensão. A existência de independência dos dados é pois uma vantagem importante dos SGBDs, nomeadamente dos SGBDs Relacionais.

Tipos de Utilizadores de uma Base de Dados

A administração dos diferentes níveis da arquitectura de uma base de dados pode colocar alguns problemas, por diferentes programadores terem de coordenar alterações aos esquemas que afectam outros programadores.

Por forma a resolver este problema, é normal eleger-se um administrador da base de dados: um responsável pela manutenção dos níveis interno e conceptual da base de dados. O administrador deverá coordenar as actualizações do esquema conceptual que afecta utilizadores e programadores, e deverá tentar encontrar a organização física da base de dados que melhor serve o conjunto das aplicações usadas.

Genericamente, existem três tipos de utilizadores de um SGBD:

- *Administrador:* O administrador é responsável pelos níveis interno e conceptual da arquitectura, e é ele que define os esquemas interno e conceptual para todas as aplicações que utilizam a base de dados.

- *Programador:* O programador é responsável pelo nível externo da arquitectura, definindo o esquema externo e os comandos de manipulação necessários para as aplicações específicas. O programador é como que um cliente do administrador.

- *Utilizador Final:* Os utilizadores finais são aqueles que fazem algo de útil com a base de dados, correndo aplicações. Eles são clientes dos programadores, e geralmente não sabem programar, nem são peritos em bases de dados.

Normalmente utiliza-se o termo de utilizador sem especificar a qual destes três tipos nos estamos a referir. Poder-se-ia definir ainda um quarto tipo de utilizadores, que seriam aqueles que implementam o próprio SGBD, isto é, aqueles que definem qual o modelo de dados e os métodos possíveis de utilizar, mas por existirem em muito menor número não são aqui considerados.

3.1.3 Bases de Dados Relacionais

As bases de dados relacionais têm como base o Modelo Relacional.

De uma forma geral, os modelos de dados, com base nalguma teoria formal, definem estruturas de dados, um conjunto de operações sobre essas estruturas, e um conjunto de regras de integridade do modelo.

Relativamente ao Modelo Relacional, são nove os principais conceitos envolvidos:

- Relações e Atributos
- Domínios
- Chaves Primárias
- Chaves Externas
- Operadores de Domínios
- Álgebra Completa: operadores de conjuntos e relações
- Integridade de Domínio
- Integridade Relacional
- Integridade Referencial

O Modelo Relacional, tem como conceito base a relação—subconjunto do produto cartesiano de n domínios. Uma relação é composta por um conjunto de atributos, em que cada atributo pertence a um determinado domínio, isto é, só pode tomar valores dentro do conjunto de valores de que é composto o domínio a que pertence.

Uma base de dados relacional consiste num conjunto de tabelas, em que cada tabela é composta por um conjunto de colunas de diversos tipos de dados, tais como inteiros, caracteres ou datas. O esquema da base de dados consiste simplesmente num conjunto de nomes de tabelas, num conjunto de nomes de colunas para cada tabela, e num tipo de dados para cada coluna.

Uma das melhores características do modelo relacional é a sua simplicidade e consistência nas suas diversas implementações. Os sistemas mais avançados que posteriormente estudaremos, incluindo as extensões do próprio modelo relacional, possuem modelos de dados mais complexos, e são menos consistentes nas suas implementações. O modelo relacional é pois um bom ponto de partida para o estudo e comparação de novos sistemas de bases de dados.

Exemplo de uma Aplicação

Por forma a melhor ilustrar os conceitos de bases de dados, será utilizada como exemplo parte de uma base de dados para armazenamento de informação sobre documentos hipertexto.

Na figura 3.3, apresenta-se parte de um diagrama de classes OMT—*Object Modeling Technique* [RBL+91] ilustrando os tipos de informação a armazenar na referida base de dados. Com base na figura 3.3, podemos definir um documento como algo composto por um ou mais capítulos—relação de agregação entre documento e capítulo. Os documentos podem ser de dois tipos: programas ou textos—relação de generalização entre `Documento`, e `Programas` e `Textos`.

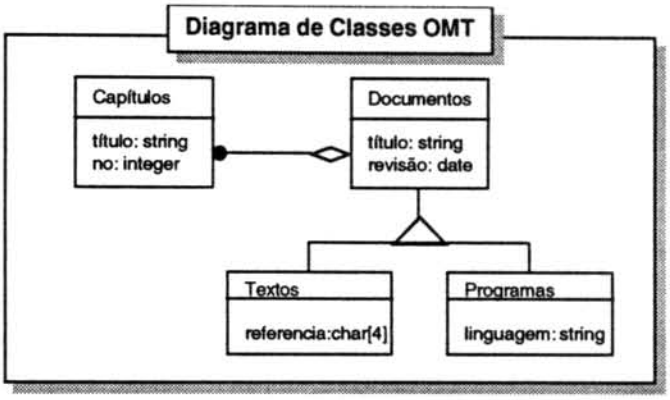


Figura 3.3: Base de dados de documentos.

Apenas para exemplificação, podemos considerar que um documento tem como atributos, um título e uma data da última revisão, e que cada capítulo tem dois atributos, concretamente um título e um número. Atendendo a que um documento pode ter um ou mais capítulos, a cardinalidade da relação entre documento e capítulo é de um- para-muitos.

Tabelas

Para representar a base de dados exemplo serão necessárias duas tabelas (ou relações) no nosso esquema relacional, uma para cada tipo de objectos da aplicação— documentos e capítulos (ver figura 3.4).

Cada documento e capítulo é representada numa linha. A cada linha da tabela é dado o nome de registo ou tuplo. As colunas das tabelas, denominam-se de atributos ou campos, e correspondem exactamente aos atributos dos objectos da aplicação, com a excepção para alguns atributos extra em cada tabela:

- A introdução do atributo `DocID` em `Documentos` serve para garantir a unicidade de cada documento, isto é, documentos diferentes devem ter identificadores diferentes.

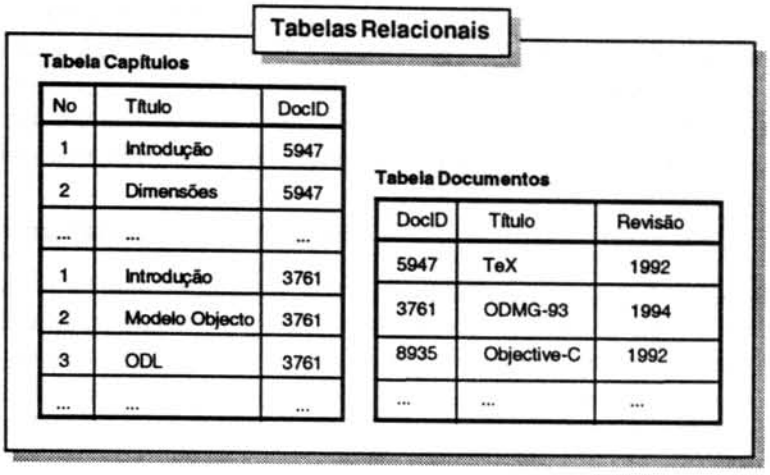


Figura 3.4: Tabelas de documentos e capítulos

É deixado à responsabilidade da aplicação a atribuição de valores únicos a estes campos.

- A introdução do atributo `DocID` em `Capítulos` serve para codificar a relação de agregação existente entre documentos e capítulos. Este atributo basicamente indica a que documento pertence o capítulo.
- A introdução do atributo `TipoDocumento` em `Documentos` serve para codificar a relação de generalização existente entre documentos e programas ou textos. Este atributo basicamente indica que tipo de documento se encontra em determinada linha da tabela.

De salientar, que não era obrigatória a introdução de alguns destes atributos, se partíssemos do princípio que, por exemplo, os nomes dos documentos na base de dados são todos diferentes. Nesta hipótese, e por forma a representar a relação existente entre documentos e capítulos recorrer-se-ia ao nome dos documentos e não ao `DocID`. As desvantagens inerentes a esta alternativa seriam em termos de espaço ocupado e na necessidade da aplicação ter de garantir a unicidade de todos os nomes dos documentos existentes na base de dados.

Chaves Primárias e Chaves Externas

Os atributos introduzidos anteriormente nas tabelas de documentos e capítulos têm funções muito especiais, e são tipícos dos sistemas de bases de dados relacionais.

Ao atributo `DocID` da tabela de documentos é dado o nome de *chave primária*. Uma chave primária é um atributo ou conjunto de atributos cujo valor deve ser único para cada linha da tabela. Por outras palavras, uma chave primária é um identificador único dos objectos reais que as linhas da tabela representam.

Ao atributo `DocID` da tabela de capítulos é dado o nome de *chave externa*. Uma chave externa é um atributo ou grupo de atributos de uma tabela cujos valores estão restritos aos

valores das chaves primárias da tabela que referencia. Uma chave externa é uma referência para objectos (linhas) de outras tabelas.

O modelo relacional não permite a existência de linhas duplicadas numa tabela. Por consequência, todas as relações têm por defeito uma chave primária: todos os atributos de uma relação. No entanto, o caso mais comum é a tabela ter uma chave primária simples, isto é, definida como apenas um atributo ou concatenação de dois ou três.

Tabelas de Entidades e de Relações

Actualmente, é costume fazer-se a distinção entre dois tipos de tabelas. As tabelas com chaves primárias simples são denominadas de tabelas de *entidades*. Estas tabelas representam tipos de objectos, em que cada linha dessa tabela representa um objecto desse tipo. Ao outro tipo de tabelas dá-se o nome de tabelas de *relações*. As tabelas de relações são utilizadas para representar associações entre objectos, com recurso a chaves externas.

Apenas as relações muitos-para-muitos entre objectos obriga à criação de tabelas de relações. No exemplo dado representou-se uma relação de muitos-para-um apenas adicionando uma chave externa a uma tabela de entidades. O mesmo se passa para relações binárias um-para-um.

As tabelas de relações são necessárias sempre que uma relação possui mais do que um atributo "muitos". Por exemplo, a relação *Autores* que conterá a informação relativa à autoria dos *Documentos*, isto é, quem é autor de quê. Esta relação tem dois atributos, que são simultâneamente duas chaves externas: o documento e a pessoa.

Uma vez que um documento pode ter várias pessoas como autores, e que uma pessoa pode ser autora de vários documentos, a relação *Autores* é de muitos-para-muitos, razão pela qual foi necessária a criação de uma tabela de referências.

Domínios e Integridade Referencial

A integridade referencial consiste num conjunto de condições que devem ser satisfeitas pelas chaves externas duma base de dados. Um bom SGBDR faz a verificação destas condições:

- sempre que é atribuído um valor a um atributo relativo a uma chave externa, o SGBDR deverá certificar-se que esse valor corresponde ao valor da chave primária de uma das linhas da tabela referenciada.
- quando um registo é apagado duma tabela, os registos que referenciavam a sua chave primária deverão ser de alguma forma marcados, por forma a poderem ser posteriormente apagados pelo sistema, ou simplesmente considerados inválidos pelas aplicações que os tentem aceder.

O modelo relacional define além dos conceitos de chaves primárias e externas, um outro conceito: o conceito de *domínios*. Um domínio é um tipo de atributos—isto é, um conjunto de valores que um atributo pode tomar. Existem domínios pré-definidos pelo SGBDR, como são o exemplo dos inteiros, datas, caracteres, etc, e domínios de entidades, como é o caso do domínio do atributo `documento` da tabela `Capítulos`.

Os domínios são úteis para a verificação da validade de invocação de determinadas operações a certos atributos. Por exemplo, não faria sentido nenhum, num documento

comparar o número de capítulos que possui com o seu título. No entanto, e apesar da utilidade de tal conceito, a maioria dos SGBDRs não o implementam. A possibilidade de criação de domínios definidos pelo utilizador é também algo que ainda não se encontra na norma actual de SQL, mas que se prevê para as suas sucessoras.

Normalização

A normalização relacional estuda a melhor forma conceptual de representação dos dados de uma determinada aplicação em tabelas relacionais. Este problema pode-se tornar bastante complexo para grandes esquemas, e foi alvo de muita actividade de investigação no princípio dos anos 80.

A base da normalização relacional consiste no partir de tabelas grandes em tabelas mais pequenas sempre que muita informação é representada numa única tabela. Entenda-se por representação com "muita informação", uma que pode dar origem a erros de interpretação por parte das aplicações, por não ser única, ou por associar informação que não se encontra relacionada. Por exemplo, a tabela `Documentos` poderia ser dispensada, se se armazenasse a informação relativa a eles na tabela `Capítulos`. No entanto, para alterar as propriedades de um determinado documento ter-se-ia de o fazer também em todos os registos dos capítulos desse documento. Se por acaso, existisse um documento que ainda não tivesse capítulos, então não seria possível armazenar a informação relativa ao documento. A normalização, ao diminuir a redundância de informação, torna os esquemas lógicos mais flexíveis, com todas as vantagens daqui inerentes.

Existe uma série de formas normais que apareceram com o propósito de eliminar sucessivamente diversas anomalias frequentes nas bases de dados relacionais. A primeira forma normal, por exemplo, requer tabelas em que os atributos apenas armazenam um único valor. As segunda, terceira e outras formas normais dedicam-se às dependências semânticas entre os vários itens de informação existentes na base de dados. O conhecimento destas formas normais pode ser útil para quem desenha esquemas relacionais [Ull89a, Ull89b].

Propriedades das Tabelas

A simplicidade do modelo relacional é em grande parte devida a três características das tabelas relacionais:

- As tabelas são conjuntos, no sentido matemático do termo: não são permitidas linhas duplicadas, e as suas linhas não se encontram segundo nenhuma ordem.
- Os atributos não se encontram ordenados; são sempre referenciados pelo nome, e os nomes dos atributos devem ser únicos na tabela.
- Os valores armazenados nas linhas e colunas são atómicos; isto é, não são conjuntos ou listas.

Estas propriedades são também limitações do modelo relacional. Torna-se por exemplo difícil representar eficientemente em tabelas relacionais objectos que se encontrem originalmente ordenados num vector ou lista. As vantagens desta simplicidade reside no facto de tornar possível a definição de uma linguagem de interrogação igualmente simples que se baseia apenas em operações sobre linhas e colunas.

3.1.4 Linguagens de Interrogação (LDD, LMD, Vistas)

O termo "linguagem de interrogação", tal qual é utilizado actualmente, não é o mais correcto para designar o que realmente significa. De facto, uma linguagem de interrogação numa base de dados relacional é utilizada para interrogar a base de dados, mas também para manipular dados, examinar e modificar o esquema da base de dados. Para se ser mais preciso dever-se-ia utilizar os termos "linguagem de manipulação de dados" (LMD) e "linguagem de definição de dados" (LDD). O termo "linguagem de interrogação" será no entanto usado para referir linguagens com ambas as capacidades.

Linguagens de Manipulação de Dados

Quase todos os SGBDRs fornecem uma variante da norma existente para linguagens de interrogação relacionais—o SQL, embora também existam SGBDRs que fornecem outras diferentes.

As duas características mais importantes de uma linguagem de interrogação são:

1. A linguagem fornece uma síntaxe parecida com a linguagem natural que simplifica o acesso à base de dados, de tal modo que os utilizadores finais e programadores apenas tenham de assimilar o mínimo para utilizar a nova linguagem.

2. A linguagem garante independência dos dados, porque as instruções de alto-nível podem ser compiladas automaticamente para operações de baixo-nível de acordo com a estrutura e organização dos dados.

Como um exemplo, eis uma interrogação em SQL para obter todos os documentos cuja data de revisão esteja compreendida entre 1-Jan-93 e 31-Dez-93:

```
SELECT TÍTULO, REVISÃO FROM DOCUMENTOS
WHERE REVISÃO>= "1-Jan-93" AND REVISÃO<= "31-Dez-93"
```

Uma linguagem de interrogação só se diz relacional se tiver pelo menos três operações básicas: selecção, projecção, e junção. A *selecção* produz um subconjunto de linhas de uma tabela com base numa condição de restrição, que em SQL é especificada após a palavra `WHERE`. A *projecção* produz um subconjunto de colunas da tabela, possível em SQL especificando os nomes dos atributos pretendidos após a palavra `SELECT`. Uma operação de *junção* associa registos de duas tabelas cujos atributos especificados têm valores iguais (equijunção) ou estão de alguma forma relacionados.

Apresenta-se abaixo um exemplo de utilização desta operação, onde se pretendem obter todos os documentos da autoria da pessoa identificada pelo nome "Manuel". Para tal, faz-se uma junção pelos atributos identificadores do documento, `DocID` da tabela `Autores` e `Documentos`:

```
SELECT DOCUMENTOS.TÍTULO, AUTORES.NOME
FROM DOCUMENTOS, AUTORES
WHERE DOCUMENTOS.DOCID=AUTORES.DOCID
AND AUTORES.NOME="Manuel"
```

De salientar que o resultado de cada uma das operações de selecção, projecção e junção é ainda uma relação, propriedade esta que torna o modelo relacional simples e simultâneamente poderoso.

Uma linguagem de interrogação apresenta várias vantagens sobre as linguagens de programação para acesso a bases de dados. É relativamente fácil aos utilizadores finais aprenderem a formular interrogações. Sempre que necessário é possível a compilação de programas com instruções da linguagem de interrogação embebidas na linguagem de programação. O compilador da linguagem, de acordo com o esquema e organização da base de dados de interrogação, escolhe automaticamente a melhor forma de processar a interrogação.

A linguagem SQL tem limitações, e não constitui uma boa linguagem de interrogação relacional [Mir92]:

- *Não é Fechada:* o resultado nem sempre é um conjunto (`group by`), o que traz limitações por exemplo ao nível de manipulação de vistas, na selecção e actualização.
- *Não é Completa:* possui operadores para conjuntos e quantificador existencial mas não possui um quantificador universal.

A linguagem SQL também não é uma boa linguagem de programação, por não ser uma linguagem de programação ortogonal:

- uma função não pode ter outra função como argumento
- depois de `from` não pode ser utilizado o resultado de uma sub-pergunta
- a síntaxe não é homogénea

Para futuras versões da norma SQL, prevê-se a implementação das seguintes caracte rísticas:

- Definição de Tipos Abstractos de Dados
- Hierarquias de Tipos
- Tipos Parametrizados
- União Recursiva

Linguagens de Definição de Dados

A linguagem de interrogação SQL também contém uma linguagem de definição de dados (LDD) utilizada para definir o esquema da base de dados. Eis um exemplo de uma instrução LDD:

```
CREATE TABLE DOCUMENTOS (
      DOCID INTEGER,
      TÍTULO CHAR(40),
      REVISÃO DATE
)
```

Uma LDD é usada para definir outros aspectos relativos aos esquemas interno e conceptual da base de dados, nomeadamente para: especificar quais os índices a criar e sobre que atributos, para definir vistas relacionais, etc.

Vistas Relacionais

Um aspecto importante relacionado com as linguagens de interrogação relacionais é o das vistas relacionais. Uma vista é uma tabela relacional virtual, isto é, uma tabela que não é armazenada directamente, mas apenas definida em termos da linguagem de interrogação.

As vistas relacionais garantem independência lógica dos dados. Se a representação da tabela ou tabelas fôr alterada, o programador pode sempre definir uma vista da nova tabela de modo a igualar a antiga, por forma aos programas continuarem a ser válidos.

As vistas relacionais podem ser ainda utilizadas para definir subconjuntos de uma tabela de acordo com as permissões de leitura e escrita de determinado grupo de utilizadores.

A limitação das vistas relacionais reside no facto de nem sempre ser possível a sua actualização como se de tabelas normais se tratassem, uma vez que a actualização pode dar origem a inconsistências com a própria definição da vista.

3.1.5 Concorrência e Recuperação de Erros

Mecanismos de controlo de concorrência e de recuperação de erros são outros aspectos importantes de um SGBD, por ajudarem a manter a integridade lógica e física da base de dados. Numa base de dados, as implementações de ambos os mecanismos encontram-se muitas vezes interligados, pelo que os iremos abordar simultâneamente.

Recuperação de Erros

Os mecanismos de recuperação de erros permitem que a base de dados seja resposta num estado consistente, após o sistema ter sofrido a ocorrência de erros de hardware ou software. Como a informação de uma base de dados pode durar muitos anos, é essencial que se evite a corrupção de informação, pois tal poderá causar problemas no futuro.

As cópias de segurança são a solução mais simples para o problema de recuperação, e poderão ser feitas diariamente por forma a se perder no máximo apenas a informação relativa a um dia de trabalho. Dependendo do SGBD em questão, poderá existir a possibilidade de fazer uma cópia de segurança incremental, copiando apenas as alterações que foram feitas desde a última cópia.

Este mecanismo de recuperação tem a desvantagem de poder ser bastante custoso: se durante o dia um problema no sistema forçar a repôr a cópia da base de dados do dia anterior, perde-se o trabalho todo desse dia. Actualmente, os SGBDs oferecem mecanismos de recuperação através de transacções, não tirando no entanto às cópias de segurança o mérito que têm como alternativa mais simples para muitas situações.

Concorrência

Quando se permite que vários utilizadores possam ler e escrever simultâneamente numa base de dados, é necessário garantir que todos tenham uma visão consistente da informação nela armazenada. Para tal, existem mecanismos de controlo de concorrência que, ao limitar os acessos, impedem, de uma forma transparente para o utilizador, que as actualizações da base de dados solicitadas por outros utilizadores interfiram na operação lógica em curso.

A utilização de *locks* constitui a solução tradicional para o controlo de concorrência em bases de dados, e constitui um componente importante de quase todos os mecanismos

mais recentes para controlo de concorrência, pelo que iremos começar por analisar em mais detalhe este tipo de mecanismo.

Locks e *Deadlocks*

A utilização mais simples dos *locks* é a de um utilizador os colocar explicitamente em todos os registos importantes para o desenrolar da tarefa em causa, por forma a que o acesso a esses registos possa ser feito apenas por dele.

Existem dois tipos de *locks*: físicos e lógicos. Os lógicos baseiam-se numa expressão da linguagem de interrogação que define o conjunto de dados a utilizar. Os físicos estão associados a dados especificos, e podem ser feitos a vários níveis: registos, páginas, ou tabelas. O *lock* a páginas é mais frequente em SGBDs modernos, e provavelmente constituirão a melhor escolha, porque é eficiente fazer *lock* às mesmas unidades físicas que são lidas e escritas no disco. Com *locks* a registos é possível aumentar o nível de concorrência por envolver unidades físicas mais pequenas. Por outro lado, os *locks* a tabelas poderão ser uma boa escolha quando se pretende aceder a muitos registos de uma mesma tabela, diminuindo com este tipo os custos inerentes à utilização de muitos, embora à custa de impedimento de acesso de outros utilizadores à mesma informação.

Relativamente às estratégias de pedido de *lock*, um pedido é recusado sempre que o mesmo item de informação se encontra já com um *lock*. Nesta situação, o utilizador poderá desistir, ou esperar e voltar a tentar mais tarde. Na situação de vários utilizadores se encontrarem à espera de conseguir fazer *lock* pode ocorrer uma situação de *deadlock*. Isto é, o utilizador A pode estar à espera que B retire o seu *lock*, e por seu turno, o utilizador B estar à espera que A retire o seu. Para que estes mecanismos funcionem surge então a necessidade de existência de mecanismos de transacções e de detecção deste tipo de situações.

Transacções

A maior parte dos SGBDs actuais implementam transacções, o que permite tanto recuperação de erros como controlo de concorrência.

Transacções são unidades lógicas de trabalho de uma aplicação. A base de dados encontra-se num estado logicamente consistente no princípio e fim da transacção, embora possa passar por estados inconsistentes durante a sua execução.

Um mecanismo de transacções de um SGBD garante ao utilizador que cada transacção que ele inicie irá até ao fim (as actualizações são feitas permanentemente no disco), ou que toda a transacção será anulada (como se as actualizações que a durante a transacção foram feitas não tivessem ocorrido). As transacções constituem um útil mecanismo de recuperação por se basearem na manutenção da consistência da base de dados.

Transacções iniciadas por diferentes utilizadores são garantidas que produzem o mesmo efeito que tinham se tivessem sido executadas em série. Uma transacção começa quando um utilizador final ou programador executa o comando de `begin transaction` da linguagem de interrogação. O fim da transacção é explicitamente indicado através de um comando de `commit`. Em alguns sistemas, ambos os comandos de início e fim de transacção podem ser implícitos.

As transacções podem ser explicitamente canceladas através de um comando de **`rollback`**, situação que originará a reposição da base de dados no estado em que estava no início da transacção.

Em muitas implementações de transacções, os utilizadores não necessitam fazer *locks* aos dados explicitamente, pois estes são implicitamente feitos pelo sistema.

O problema dos *deadlocks* é automaticamente tratado pela implementação da transacção, podendo ser detectados automaticamente através de análise de dependências entre pedidos, ou o sistema pode simplesmente desistir de uma transacção após um determinado período de espera. Sempre que é evitado um *deadlock*, a transacção que o originou é cancelada automaticamente pelo SGBD. Uma transacção pode assim ser cancelada devido a *deadlock*, erro de sistema, ou pedido explícito do utilizador.

As transacções fornecem uma solução muito melhor para os problemas de recuperação e controlo de concorrência do que os *locks* manuais e as cópias de segurança. As desvantagens das transacções são: as implementações actuais não tratam bem o caso de transacções de longa-duração, as quais são necessárias nas aplicações de engenharia em que os registos podem ter *locks* durante vários dias seguidos.

3.1.6 Bases de Dados Distribuídas

O termo bases de dados distribuídas tem sido utilizado com vários significados diferentes. As capacidades consideradas mais importantes para este tipo de bases de dados são duas:

- *Acesso Remoto* a bases de dados armazenadas em máquinas diferentes daquela em que se está a correr a aplicação;
- *Distribuição de uma Base de Dados* por várias bases de dados, possivelmente em máquinas diferentes, mas administradas separadamente na mesma máquina.

Acesso Remoto

Muitos SGBDRs actuais suportam acesso remoto a bases de dados, em que o facto da base de dados se encontrar remotamente colocada é transparente para o utilizador ou programa de aplicação.

O acesso remoto é quase universalmente implementado da mesma maneira. Uma pequena parte do SGBD reside com o programa de aplicação na máquina cliente. Esta porção é responsável pela codificação das interrogações—enviando-as pela rede para a máquina servidora onde reside a restante parte do SGBD, e pela recepção dos registos retornados pela rede para descodificação e entrega ao programa cliente.

O acesso remoto embora não sendo difícil de implementar, pode-se tornar complexo devido à heterogeneidade de uma rede máquinas, onde a codificação dos tipos de dados pode ser diferente para clientes e servidores. Neste caso, deve ser usada uma representação dos registos independente da máquina, ou então o servidor deverá traduzir para a representação das máquinas cliente.

A utilização desta arquitectura de cliente-servidor, com acesso remoto a bases de dados, pode alcançar desempenho superior àquele que pode ser obtido por um único processador para várias aplicações: a máquina servidora não é sobrecarregada com a execução das aplicações, e não haverá muito tráfico na rede se a linguagem de interrogação for suficientemente poderosa para exprimir operações do SGBD em poucas chamadas.

Por outro lado, quando se pretende extrair grandes quantidades de informação de uma só vez, ou quando se têm de fazer vários pedidos para alcançar os resultados pretendidos pela aplicação, então o desempenho poderá tornar-se muito menor, Nestes casos, poderá ser melhor a utilização de um único processador, ou encontrar alguma

forma de armazenamento temporário e invisível ao utilizador para melhorar a eficiência (as implementações actuais de acesso remoto a bases de dados não fazem isto).

Distribuição

Define-se uma base de dados distribuída como um conjunto de bases de dados distribuídas na rede ou não, e que aparecem ao utilizador como uma só.

As bases de dados distribuídas são úteis em vários aspectos. Como uma base de dados é normalmente a unidade mais pequena controlada por um administrador de base de dados, as bases de dados distribuídas permitem que dois ou mais departamentos possam conceber e fazer a manutenção da sua própria informação independentemente, podendo no entanto formular interrogações que utilizem toda a informação armazenada nas várias bases de dados. As bases de dados distribuídas contribuem também para uma melhoria de desempenho, ao distribuir a carga de processamento por várias máquinas, sempre que exista uma repartição lógica das bases de dados pelas máquinas que requeira pouca comunicação entre máquinas nas interrogações normais.

Existem dois tipos de repartição. Tabelas diferentes podendo existir em bases de dados diferentes. Uma mesma tabela repartida por várias bases de dados, com algumas linhas numa base de dados e o resto noutra (repartição horizontal), ou algumas colunas numa base de dados e as restantes noutra (repartição vertical). Embora a informação se encontre repartida por várias bases de dados, deve ser possível executar operações de selecção, junções, e outras operações sobre tabelas, transparentemente para o utilizador como se as várias bases de dados fossem apenas uma.

As diferentes implementações diferem na forma de como o esquema da base de dados é distribuído. Algumas permitem que o esquema de cada base de dados seja independente, sendo definido pelo administrador um mapeamento entre tabelas dos diferentes esquemas de bases de dados. Outras implementações requerem a existência de um esquema global que será difundido para cada uma das máquinas intervenientes. Esta alternativa simplifica o acesso distribuído, embora à custa de uma manutenção separada do esquema, e que cada máquina onde reside seja sempre acessível, o que nem sempre acontece.

3.1.7 Ferramentas de Apoio

A razão mais importante que leva à escolha de bases de dados relacionais para implementação de aplicações tem a ver com o leque de ferramentas de apoio que o SGBD oferece tanto ao utilizador final como ao programador. É nas ferramentas de apoio que podem ser obtidos os maiores ganhos em produtividade na utilização de um SGBD, uma vez que o desenvolvimento e a manutenção ao longo do tempo são as fases da vida de uma aplicação que requerem maior esforço.

4GLs e Formulários

Muitos SGBDRs fornecem utilitários para definição de formulários ou linguagens de quarta-geração (4GLs) concebidas para desenvolvimento de aplicações de base de dados, e que são normalmente uma extensão da linguagem de interrogação. Tanto os utilitários como as 4GLs simplificam a definição de écrans, mapeando os formulários de écran com atributos de registos da base de dados, validando entradas de dados, e automatizando outras tarefas, por vezes simples, mas bastante maçadoras.

A utilização de uma 4GL em vez de uma 3GL (linguagem de programação como C, COBOL, ou FORTRAN), pode constituir uma grande diferença em termos de produtividade de programação e utilidade de um SGBD. O código de aplicações escritas em 4GLs é em geral bem mais pequeno, uma vez que todo o suporte para écrans, janelas e menus e verificação de erros faz parte da própria linguagem, isto se a aplicação a ser desenvolvida fôr do tipo de aplicações para as quais a 4GL foi concebida.

Ferramentas de Apoio aos Utilizadores Finais

Com ferramentas apropriadas deste tipo, algumas aplicações passam a dispensar qualquer programação. Esta capacidade é necessária para análises de mercado, relatórios de gestão, e outras aplicações simples de consulta de informação.

Para além da 4GL, muitos SGBDRs oferecem também ferramentas para definição de relatórios e ferramentas para interrogação através de exemplos (query-by- example, QBE). Estas ferramentas geram automaticamente comandos da linguagem de interrogação e do gerador de relatórios por forma a serem obtidos os resultados especificados pelo utilizador.

Recentemente, as ferramentas deste tipo foram melhoradas por forma a melhor explorar as capacidades gráficas das estações de trabalho, simplificando o desenvolvimento de interfaces gráficas com o utilizador, continuando no entanto a suportar a exploração de terminais alfanuméricos.

Ultimamente tem sido dedicada especial atenção a ferramentas destinadas a auxiliar na concepção e manutenção de esquemas de bases de dados, recorrendo a uma notação gráfica. Tais tarefas não são fáceis, por ser necessário incluir toda a informação das aplicações, mas sem violar as regras de normalização. O enorme número de tabelas e atributos pode tornar o esquema difícil de manter e de comunicar a todos os programadores.

3.1.8 Desempenho

O desempenho é quase sempre o factor mais importante na selecção de um SGBD. Por forma a podermos futuramente comparar os SGBDs tradicionais com outros mais avançados vamos ver alguns aspectos que serão importantes para essa comparação.

Interrogações

Com um bom processador de interrogações para compilar os comandos de uma aplicação, o factor limitador para muitas aplicações comerciais, num bom SGBDR, reduz-se à velocidade de transferência do disco.

A implementação de um compilador de interrogações pode ser complexo, mas a experiência actual em optimização de interrogações é boa, embora muitos produtos comerciais ainda não utilizem as últimas técnicas estudadas. Muitos produtos compilam, em vez de interpretarem as interrogações, alcançando factores de 10 vezes ou mais em velocidade de execução analisando e optimizando as interrogações logo na compilação do programa.

Uma representação difícil pode levantar problemas em determinadas interrogações mais complexas, como por exemplo a produção da lista de documentos existentes associados a um determinado programa. O processamento desta interrogação implica a realização de uma união recursiva, operação que, tal como já referido, não existe na linguagem SQL. Os SGBDO tentam resolver estes e outros problemas.

Arquitectura

Os SGBDRs geralmente são implementados utilizando uma arquitectura de dois processos: programas de aplicação fazem chamadas interprocesso a um ou mais processos servidores de base de dados que executam as interrogações. Esta separação de processos traz algumas vantagens; por exemplo, ajuda a proteger a integridade da base de dados impedindo que os programas de aplicação acedam directamente a páginas da base de dados ou estruturas de dados. A desvantagem reside no facto de em alguns sistemas operativos uma chamada interprocesso ser bastante custosa, pelo que os programas que terão necessidade de realizar várias chamadas poderão ver o seu desempenho afectado.

Controlo de concorrência é um factor que pode diminuir significativamente o desempenho do sistema se existirem conflitos entre as aplicações, ou quando apenas existe a possibilidade de fazer *lock* ao nível do registo.

A arquitectura de processos e implementação dos mecanismos de controlo de concorrência nos SGBDOs difere em vários aspectos, tal como veremos adiante.

3.2 Conceitos de Bases de Dados de Objectos

Os SGBDOs pretendem dar suporte a aplicações científicas, de engenharia e a novas gerações de aplicações comerciais mais sofisticadas. Nesta secção, iremos descrever a funcionalidade típica oferecida pelos SGBDOs e as abordagens que são tomadas para implementar essa funcionalidade.

Porque o trabalho na área de gestão de dados de objectos ainda está na sua infância, não existe ainda um único modelo de objecto tal como existe para os sistemas relacionais, e existem ainda muitas questões de investigação em aberto. Os SGBDOs foram construídos a partir de uma variedade de modelos de dados e arquitecturas de implementação, pelo que apenas faz sentido referir as várias abordagens consideradas.

Neste secção serão apresentadas as características de um SGBDO. Um SGBDO mais evoluído possuirá uma grande parte delas, enquanto que um mais simples apenas terá algumas. Na prática, o importante é que o SGBDO possua as características necessárias para a aplicação a que se destina.

As características dos SGBDOs que irão ser apresentadas nesta secção serão agrupadas da seguinte forma:

- Aspectos gerais dos objectos: identificadores, atributos, etc;
- Relações, atributos para referências, e conjuntos;
- Procedimentos e encapsulamento;
- Agregação de objectos e o seu significado;
- Tipos de objectos (classes), hierarquias de tipos, e evolução dos tipos;
- Interfaces com linguagem de interrogação e linguagem de programação;
- Controlo de concorrência, versões, e recuperação de erros.

Antes de apresentar em mais detalhe cada uma destas características, vamos começar por apresentar as diversas abordagens possíveis para a construção de um SGBDO.

3.2.1 Arquitecturas duma Base de Dados de Objectos

Existem basicamente dois aspectos em que os SGBDOs podem ser diferentes: no modelo de dados que suportam (relacional estendido, funcional ou orientado por objectos), e na arquitectura da base de dados, isto é, na forma como o SGBDO é implementado e integrado com outros componentes do sistema. Uma vez que os modelos de dados têm convergido para características idênticas, as diferenças em termos de arquitectura de base de dados ganham ainda maior importância na distinção dos diferentes SGBDOs.

Existem quatro arquitecturas de base de dados importantes a considerar aqui: sistemas de bases de dados estendidos, linguagens de programação para bases de dados, gestores de objectos, e geradores de SGBDs.

As quatro arquitecturas encontram-se fortemente relacionadas:

- *Gestor de Objectos* fornece a funcionalidade mais básica: armazenamento de objectos persistentes.

- *Sistema de Base de Dados Estendido* acrescenta novas capacidades, tais como, nova linguagem de interrogação e controlo de concorrência.
- *Linguagem de Programação para Bases de Dados* faz a integração de uma linguagem de programação persistente com um sistema de bases de dados estendido.
- *Gerador de SGBDs* é um sistema que pode ser usado para construir todos estes diferentes tipos de sistemas.

Apresentadas estas quatro arquitecturas, torna-se mais clara a origem das abordagens existentes para gestão de dados de objectos. As abordagens correspondem exactamente àquelas arquitecturas, excepto para os sistemas de bases de dados estendidos, para os quais se faz a distinção entre os sistemas baseados nos modelos relacionais estendidos e os sistemas baseados nos modelos funcionais e semânticos [Cat91]. Na figura 3.5 são ilustradas as relações existentes entre estas diferentes arquitecturas.

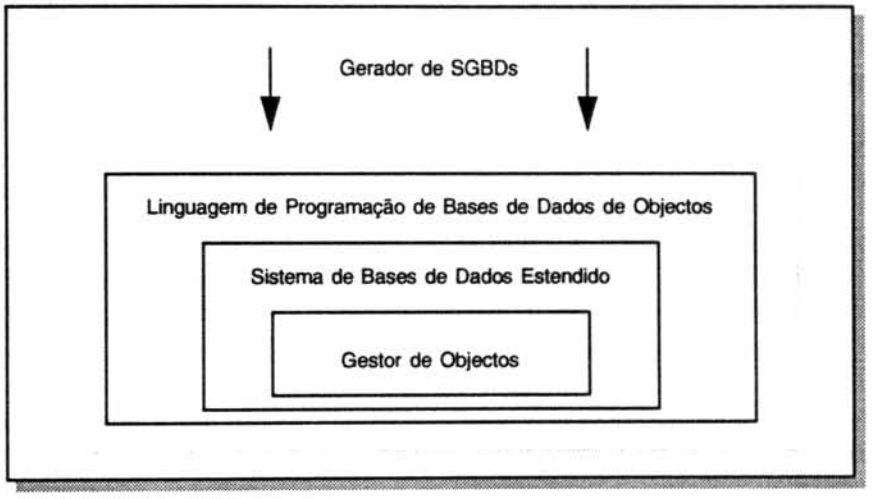


Figura 3.5: As arquitecturas de sistemas de bases de dados de objectos

Existem assim cinco abordagens, das quais iremos incidir especialmente em duas: as baseadas nos modelos de dados de objectos e nos modelos relacionais estendidos, por serem as mais populares em termos de investigação e desenvolvimento.

Linguagens de Programação de Bases de Dados de Objectos

São sistemas de bases de dados que estendem linguagens de programação existentes, tais como C++, para permitirem persistência, controlo de concorrência, e outras capacidades. Estes SGBDOs possuem uma linguagem de interrogação e uma linguagem de programação que partilham um mesmo ambiente de execução. Os únicos SGBDOs deste tipo baseiam-se em modelos de objectos e linguagens de programação orientadas por objectos.

Praticamente todos os SGBDOs baseados no modelo de dados de objectos caiem no âmbito das linguagens de programação de bases de dados de objectos. Estes sistemas são baseados na arquitectura de linguagens de programação de bases de dados: as aplicações são escritas numa extensão de uma linguagem de programação existente, em que a linguagem e sua implementação foi estendida por forma a incorporar a funcionalidade de uma base de dados.

Muitas destas linguagens foram integradas com o C++ ou uma linguagem sua derivada, provavelmente devido à sua já grande popularidade. É intuito em muitos destes SGBDOs a integração de múltiplas linguagens de programação, muito embora o problema de acesso a bases de dados através de múltiplas linguagens de programação seja de difícil resolução, devido à forte ligação existente entre a base de dados e os sistemas de tipos de cada uma das linguagens de programação.

A evolução nos sistemas de bases de dados de objectos tem sido no sentido das linguagens de programação de bases de dados. Cada um dos SGBDOs pode ser transformado facilmente numa linguagem de programação de bases de dados, uma vez que os modelos de dados da programação orientada por objectos e das bases de dados de objectos são muito similares, pelo que assumem assim um interesse especial.

As linguagens de programação persistente são predecessoras das linguagens de programação de bases de dados, não tendo, no entanto, a intenção de fornecer toda a funcionalidade de uma base de dados, ou permitir o acesso a informação persistente a partir de múltiplas linguagens de programação.

Sistemas de Bases de Dados Relacionais Estendidos

Estes sistemas fornecem nova funcionalidade através de uma linguagem de interrogação nova ou estendida que incorpora procedimentos, objectos, hierarquia de tipos, e outras características dos SGBDOs. Nestes sistemas, existem dois ambientes de execução separados: o da linguagem de programação e o da linguagem de interrogação da base de dados. Cada uma das linguagens pode invocar a outra mas são independentes em termos de ambiente de execução, com as inerentes desvantagens ao nível de transferência de dados entre as linguagens. Estes sistemas foram construídos com uma variedade de modelos de dados.

Não existe uma única abordagem de "extensão relacional". De facto, talvez esta seja a abordagem em que haja menos consistência em termos de produtos. É no entanto popular porque potencialmente beneficia do muito que se sabe sobre os sistemas relacionais, e também porque é relativamente fácil, partindo de produtos relacionais existentes, a construção de novos SGBDOs.

Extensões Funcionais de Sistemas de Bases de Dados

São sistemas baseados nos modelos de dados funcionais.

A base dos modelos funcionais são objectos e funções. As funções fazem o mapeamento de objectos noutros objectos, incluindo entidades e valores literais. As funções podem ser definidas procedimentalmente, ou por armazenamento. Os atributos, relações e procedimentos são todos representados como funções, neste modelo. Muitos sistemas permitem que as funções tenham mais do que um valor associado.

Gestores de Objectos

São produtos de software que podem ser vistos como extensões de sistemas de ficheiros ou sistemas de memória virtual existentes. O modelo de dados é limitado e consiste apenas num repositório de objectos. Os gestores de objectos possibilitam o armazenamento persistente de objectos simples, muitas vezes fazem controlo de concorrência, mas não fornecem linguagem de interrogação nem linguagem de programação, podendo ser úteis para aplicações simples que necessitam de capacidades de armazenamento mais sofisticadas.

Geradores de SGBDs

Esta arquitectura representa sistemas que permitem ao implementador a construção de um SGBD vocacionado para as suas necessidades específicas. O implementador começa por definir os modelos de dados conceptual e interno pretendido. O modelo conceptual especifica as capacidades de interrogação e de representação do sistema, tais como tabelas, chaves, relações, e sintaxe para as interrogações. O modelo interno especifica os métodos de acesso e o mapeamento do nível conceptual para o interno.

Um gerador de SGBD poderá ser utilizado para produzir um SGBDO, um SGBD Relacional, ou mesmo um sistema de gestão de ficheiros.

3.2.2 Objectos

Apesar do termo *objecto* ter muitos significados diferentes, todos os SGBDOs incorporam as seguintes duas capacidades básicas dos objectos:

- *Agrupamento de Objectos:* Os objectos podem servir para agrupar informação pertencente a uma entidade real—um documento, uma pessoa, um produto. As implementações do modelo relacional suportam mal esta capacidade, fazendo-o à custa de chaves primárias e externas.

- *Identificadores de Objectos:* Os objectos podem ter uma identidade única e que é independente dos valores que contêm. No modelo relacional uma entidade ou objecto é identificado pela sua chave primária. Existem outros sistemas que permitem que um objecto seja referenciado por um número gerado internamente, um identificador de objecto, e que é independente do valor da sua chave primária, se eventualmente tiver alguma.

Um SGBD que suporte estes dois conceitos diz-se ser estruturalmente orientado por objectos. Um outro tipo de orientação por objectos é a comportamental, em que os objectos são a base do encapsulamento de informação e invocação de métodos.

Identificadores de Objectos

Todo o trabalho desenvolvido na área dos modelos de dados após a introdução dos sistemas relacionais, reconheceu as limitações das chaves primárias e dos modelos que se baseiam em valores, pelo que foram introduzidos identificadores de objectos, também denominados de OIDs—*Object Identifiers*, para remediar as dificuldades. As chaves primárias obrigam a que o programador: gere os seus próprios identificadores únicos, ou então que use nomes inteligíveis como identificadores.

Ambas as soluções não são satisfatórias:

- primeiro por ser mais eficiente, menos aborrecido, e conveniente deixar que seja o próprio SGBD a criar os identificadores, pois este poderá tirar partido disso para operações internas, por exemplo, de pesquisa;
- segundo, porque apesar dos identificadores inteligíveis serem agradáveis, são em média bastante compridos, pelo que em termos de espaço a eficiência será muito menor.

Os OIDs são importantes para referenciar objectos algures na base de dados, pouco importando a forma como os diversos SGBDOs lidam com os OIDs. A única diferença visível reside na acessibilidade ou não aos OIDs. As restrições impostas aos OIDs prendem-se com detalhes de implementação, e com preocupações de desempenho do SGBDO.

Chaves de Objectos

Alguns SGBDOs permitem que os objectos, adicionalmente aos OIDs, possuam também nomes que identificam o objecto. Estes nomes são equivalentes às chaves primárias do modelo relacional, os quais denominamos por chaves de objectos.

Uma das vantagens de se poder definir chaves nos SGBDOs, consiste em não se perder alguma da funcionalidade do modelo relacional, e ser também uma conveniência que pouco acarreta em termos de SGBD. Um SGBD para permitir a utilização de chaves basta oferecer uma sintaxe simples para referenciar objectos, e fazer uma verificação automática da unicidade das chaves.

Atributos Simples

No modelo relacional, os objectos eram representados por registos, e os valores literais armazenados como atributos dos registos (inteiros, reais, caracteres, etc.).

Os objectos podem ter atributos independentemente se são representados num SGBD ou SGBDO, embora estes últimos ofereçam maior flexibilidade nos tipos de valores permitidos. Os atributos de um objecto podem ter valores complexos, tais como conjuntos ou referências para outros objectos, e podem ter tipos definidos pelo próprio utilizador, em que para além da representação do tipo se indicam também as operações permitidas sobre o novo tipo.

Tal como os atributos dos registos relacionais, os atributos de objectos de um SGBDO têm nomes, que podem ser referidos utilizando a notação seguinte, em que `doc` é um objecto e `título` é um nome de um atributo:

```
doc.título
```

Um tipo muito especial de atributos são os atributos de comprimento variável (BLOBs— *Binary Large Objects*) que podendo ocupar bastante espaço, têm por razões de eficiência um tratamento especial, continuando no entanto a ser valores literais.

Atributos que tenham valores literais denominam-se *atributos simples*.

Atributos do tipo Referência

Existem três tipos de atributos complexos: referências, colecções, e procedimentos.

As *referências*, ou associações, são utilizadas para representar relações entre objectos. Os seus valores são objectos—referências para entidades. Este tipo de atributos é análogo aos apontadores de uma linguagem de programação, ou às chaves externas num sistema relacional, embora tenham diferenças importantes:

1. Os atributos do tipo referência não podem assumir valores ilegais, isto é, referências para áreas de memória não concordantes com o seu tipo, enquanto que os apontadores podem. As referências são automaticamente eliminadas sempre que o objecto referenciado é apagado.

2. Os atributos do tipo referência não têm associados valores visíveis aos utilizadores, enquanto que as chaves externas têm. Qualquer valor do objecto referenciado pode ser alterado, e a referência manter-se-á válida.

Atributos do tipo Colecção

O segundo tipo de atributo complexo, as *colecções*, são usados para listas, conjuntos, ou vectores de valores. As colecções de valores podem incluir valores de atributos simples e complexos.

Como exemplo poderíamos definir um atributo para um tipo de objectos `Documento`, que conteria todos os seus capítulos:

```
Documento: {
   título: STRING;
   revisão: DATE;
   capítulos: LIST[Capítulo];
}
```

A primeira forma normal relacional não permite atributos com valores múltiplos, pelo que existe aqui uma grande diferença entre o modelo relacional e o modelo de objectos, no qual conjuntos e listas de objectos fazem parte do modelo.

O tipo `LIST[Capítulo]` é um exemplo de um tipo parametrizado, o que significa que `LIST` (ou `SET`), são tipos que aceitam outros tipos como parâmetros, definindo assim novos tipos com base nos parâmetros.

Os tipos parametrizados foram adoptados das linguagens de programação, desde os vectores até aos próprios conjuntos e listas. Poucas linguagens permitem ao utilizador a definição dos seus tipos parametrizados. Eiffel e C++ são exemplos de linguagens correntes que o permitem.

Atributos Derivados

Os valores dos atributos podem ser armazenados, ou derivados através da execução de procedimentos, se fôr possível associar ao atributo um procedimento que possa ser executado sempre que seja necessário obter o seu valor. Neste último caso, denominam-se os atributos de atributos *derivados*.

Vamos supôr, por exemplo, que em vez de definirmos um campo `total_linhas` para os objectos `Documento` como um inteiro, descreviamos num procedimento a forma de o obter. Esta decisão só traria vantagens, atendendo a que este total, depende de cada um dos capítulos que o compõem, o que implica que, sempre que um ou outro capítulo é alterado, o cálculo do número total de linhas do documento tem de ser recalculado. Com a utilização deste tipo de atributos, o cálculo seria refeito automaticamente, sempre e apenas quando, fosse utilizado o valor do atributo `total_linhas`.

Uma definição possível seria:

```
Documento: {
   título: STRING;
   revisão: DATE;
   capítulos: LIST[Capítulo];
   total_linhas: INTEGER PROCEDURE () {sum_all(capítulos)}
}
```

Sempre que é transparente para o utilizador a forma de como os valores dos atributos são obtidos, por armazenamento ou por cálculo, denominam-se atributos *virtuais*.

A actualização de atributos pode no entanto ser problemática quando estes são virtuais, pois não será correcto permitir a atribuição de valores a este tipo de atributos, atendendo a que semânticamente pode ser incorrecto. A única forma clara de permitir actualizações de atributos virtuais é fornecendo dois métodos: um para leitura e outro para escrita.

Nenhum SGBDO corrente permite a actualização de atributos derivados. Nas linguagens de programação orientadas por objectos não deverá ser possível implementar tal facilidade por promover uma visão diferente dos tipos de objectos e operações sobre eles possíveis, a que os programadores não estão habituados.

3.2.3 Relações entre Objectos

A representação de relações é um dos aspectos fundamentais em que o modelo relacional mais difere do modelo de objectos. Começaremos por analisar a forma como são representadas as relações no modelo de objectos: integridade referencial, relações binárias e não binárias. Depois contrastar-se-ão estas representações com relações nos modelos relacional e relacional estendido.

Relações Binárias

As relações no modelo de objectos são representadas utilizando atributos complexos: referências e colecções de referências, dependendo da multiplicidade da relação.

As relações binárias podem ser representadas da seguinte forma:

- Representa-se uma relação um-para-um entre dois tipos de objectos A e B adicionando em cada tipo um atributo referência para o outro.
- Representa-se uma relação muitos-para-um entre dois tipos de objectos A e B adicionando um atributo referência em A e um atributo conjunto de referências em B.
- Representa-se uma relação muitos-para-muitos entre dois tipos de objectos A e B adicionando em cada tipo um atributo conjunto de referências.

Uma relação entre dois tipos de objectos A e B, pode em todos os casos ser vista como sendo representada por um novo atributo referência em A para um objecto B, e um novo atributo em B que permite a referência inversa para A.

Atributos Inversos

Aos atributos usados para representar as relações entre dois tipos de objectos A e B é dado o nome de atributos *inversos*, por existirem sempre em simetria: referência de A para B, e referência inversa de B para A. Exemplos de atributos inversos são os apresentados abaixo: `instruções` e `DocID` dos tipos `Documento` e `Capítulos`, respectivamente.

```
Documento: {
   título: STRING;
   revisão: DATE;
   capítulos: LIST[Capítulo] <-> documento;
}

Capítulo: {
   título: STRING;
   documento: Documento <-> capítulos;
}
```

Estes pares de atributos inversos devem ser mantidos em sincronismo um com o outro. Por exemplo, quando se adiciona um capítulo ao documento ter-se-á de colocar nesse capítulo uma referência para o documento ao qual pertence. Esta manutenção de integridade referencial deve ser implícita, e suportada ao nível do SGBDO, libertando o programador de tais tarefas. No entanto, os SGBDs diferem no suporte deste conceito. Na figura 3.6 é apresentado um exemplo com objectos e respectivas relações.

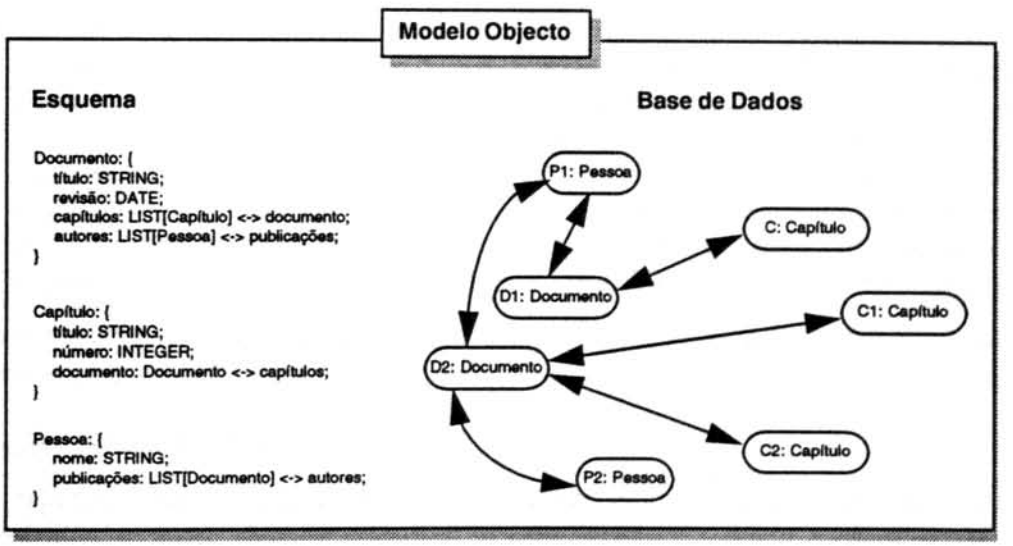


Figura 3.6: Relações binárias num modelo de dados orientado por objectos

Integridade Referencial

Para suportar a integridade referencial numa representação baseada em identificadores de objectos, não basta verificar se os atributos contêm identificadores válidos de objectos de determinado tipo. Existem vários níveis de integridade referencial:

- *Inexistência de verificação de integridade.* O sistema pura e simplesmente não verifica as referências. Esta abordagem é perigosa, pois poderão, por exemplo, ser utilizadas referências para objectos que já não existem. No entanto, muitas linguagens de programação, tais como o C, não garantem a integridade dos apontadores.
- *Validação das referências.* O sistema garante que as referências são para objectos que existem e são do tipo correcto, o que poderá ser conseguido de duas formas:
  1. O sistema pode automaticamente apagar os objectos que já não são referenciados. Estes sistemas normalmente não permitem que um objecto possa ser apagado pelo utilizador, e são eles que em exclusivo se encarregam de o fazer.
  2. O sistema pode exigir que os objectos sejam explicitamente apagados, mas conseguir detectar automaticamente a existência de referências inválidas.
- *Integridade das Relações.* O sistema permite que um objecto seja explicitamente apagado, que sejam modificados objectos e relações, e continuar a manter a correcção automática das relações existentes entre os objectos. Os atributos inversos possibilitam integridade relacional, tal como o modelo relacional permite, mas as alternativas 1 e 2 não o permitem, uma vez que as relações entre as entidades são representadas por atributos que não são mantidos ou sincronizados pelo SGBD. A maioria dos SGBDOs suporta integridade relacional.
- *Semântica de Referências Parametrizável.* Os sistemas mais sofisticados permitem que se especifique a semântica de integridade referencial desejada para cada tipo de objecto ou relação. Os SGBDOs actuais ainda não suportam tais facilidades, excepto para casos muito restritos. Esta semântica poderá ser sempre codificada em métodos associados por objectos. É no entanto muito mais vantajoso o tratamento da integridade referencial pelo SGBDO, pois possibilita que a semântica seja explicitada no esquema da base de dados, e seja vísivel tanto a utilizadores como a programadores, minimizando assim erros de codificação por falta de informação.

A manutenção de múltiplas versões de objectos impõe algumas restrições à integridade referencial, tal como veremos mais adiante.

Relações Não-Binárias

Para representação de relações de ordem superior, nomeadamente ternárias, torna-se necessário criar novos tipos de objectos, mesmo quando o terceiro atributo da relação é um valor literal.

Do ponto de vista do utilizador, a introdução de objectos intermédios para as relações ternárias podem tornar o esquema mais complexo. No entanto, a representação relacional do mesmo esquema requer também uma relação extra. Por outro lado, na notação apresentada anteriormente para acesso a atributos de um objecto, a introdução de um objecto intermédio é conveniente, pois facilita a navegação pelas relações muitos-para-muitos.

Relações Derivadas

As relações, tais como os atributos, podem ser definidas procedimentalmente em vez de explicitamente. Num SGBD com atributos derivados, também podem ser definidas relações derivadas, uma vez que as relações são definidas em termos dos atributos. As relações, no entanto, requerem mais do que um método de leitura—um para cada um dos atributos inversos envolvidos na relação.

Comparação com Relações no Modelo Relacional e suas Extensões

A representação de relações no modelo de objectos é mais poderosa e conveniente do que a representação relacional, pois contém uma semântica acrescida relativamente à relacional (ordenação, navegação). As diferenças mais óbvias na representação relacional de relações são:

1. No modelo relacional, as referências são representadas utilizando chaves primárias em vez de OIDs, embora ambos utilizem referências como base da representação de relações.

2. As tabelas de relações são aproximadamente equivalentes aos objectos intermédios necessários para representação de relações no modelo de objectos, mas elas também são necessárias em mais casos, por forma a ser satisfeita a normalização relacional.

3. O modelo de objectos agrupa todas as relações nas quais um objecto participa como atributos do objecto. Por outro lado, no modelo relacional torna-se necessário examinar todo o esquema, para determinar quais as tabelas que têm chaves externas que envolvem entidades de uma determinada tabela.

4. Modelos de objectos permitem ordenação manual de relações utilizando listas.

5. A sintaxe de acesso e semântica envolvida são diferentes.

Em suma, no modelo relacional relações binárias um-para-um são geralmente representadas juntando duas tabelas de entidades numa única tabela. Num modelo de objectos, os dois objectos poderão igualmente ser agrupados num só.

As relações binárias um-para-muitos são representadas de forma idêntica em ambos os modelos.

Relações binárias muitos-para-muitos podem ser representadas directamente no modelo de objectos através de dois atributos do tipo colecção, mas no modelo relacional exigem uma terceira relação (devido à normalização relacional).

Relações ternárias requerem a introdução de estruturas intermédias nos dois modelos.

As implementações de relações nos modelos relacionais estendidos actuais são as mesmas das do modelo relacional, com apenas a excepção de permitirem a utilização de OIDs em vez de chaves para referenciar objectos.

3.2.4 Objectos Compostos

A agregação consiste em agrupar várias partes num todo. Os objectos que se agregam a outros formam objectos compostos.

A agregação poderá aparecer a vários níveis. Por exemplo, um documento pode ser composto por outros documentos, que por sua vez, também poderão ser compostos por outros, e assim sucessivamente.

Semântica da Agregação

A razão que leva os SGBDOs a adoptarem um tratamento especial face à relação de agregação, tem a ver com o facto de se poder tirar partido da sua semântica para automatizar certas operações comuns. Por exemplo, quando um objecto composto é apagado (copiado) deverão ser apagados (copiados) automaticamente todos os seus componentes, e subcomponentes, e assim sucessivamente.

Os objectos compostos podem também servir de critério para a localização físico dos objectos em disco, por exemplo.

Relações de Agregação

Os SGBDs diferem na forma de definição das relações de agregação. Os SGBDOs normalmente fornecem um mecanismo de declaração de tipos de relações e semântica associada (copiar, apagar, etc.), tipos estes que depois poderão ser especificados nas relações correspondentes à agregação.

Um mesmo objecto composto poderá participar em mais do que uma hierarquia de agregação, e a maior parte dos SGBDOs permitem-no, muito embora este problema possa ser sempre resolvido no caso do SGBDO não o permitir.

3.2.5 Procedimentos

Até ao momento têm-se focado apenas aspectos estruturais da orientação por objectos. Passaremos agora a lidar com os procedimentos que operam sobre os objectos.

Os SGBDOs fornecem uma linguagem de acesso à base de dados que é computacionalmente completa, isto é, pode realizar as mesmas operações que uma linguagem de programação pode. Para além disso, os SGBDOs permitem associar procedimentos aos objectos da base de dados, e armazená-los na base de dados.

Existe uma variedade de abordagens para suportar procedimentos em bases de dados, nomeadamente os sistemas comportamentalmente orientados por objectos, onde os procedimentos são utilizados para encapsular a semântica do objecto.

Procedimentos e Encapsulamento

O paradigma de orientação por objectos é derivado do conceito de tipos abstractos de dados. As declarações de tipos abstractos de dados definem explicitamente as partes pública, protegida e privada da informação de um objecto (ver secção 2.3.2).

Os métodos são procedimentos utilizados nos sistemas orientados por objectos para encapsular os atributos de um objecto. Os atributos associados com um objecto são normalmente privados, e apenas os métodos do objecto podem examinar ou actualizar a sua informação; os métodos podem ser públicos.

O encapsulamento tem um papel idêntico à independência de dados no modelo relacional: permite que uma implementação seja alterada sem afectar os programas que utilizam o tipo de dados.

A definição de encapsulamento é por vezes muita restritiva, pelo que se propõe uma definição em que tanto dados como procedimentos possam constar nas partes privada, protegida e pública.

Alguns SGBDOs fornecem o encapsulamento como uma opção do programador, permitindo que sejam declarados procedimentos associados com objectos, mas permitindo

também que sejam definidos procedimentos livres, isto é, definidos independentemente de qualquer tipo de objectos.

Procedimentos nos Modelos Relacionais Estendidos

Uma outra forma de incorporar procedimentos num modelo de dados, para além das abordagens orientadas por objectos, consiste em estender a linguagem de interrogação relacional para procedimental, ou permitir escrever procedimentos numa linguagem de programação que serão invocados pelo SGBD Relacional.

Dados Activos

Os procedimentos podem ser associados com os objectos ou seus atributos para definir dados activos. Os atributos derivados são apenas um de muitos tipos de dados activos que podem ser armazenados numa base de dados.

Os tipos mais importantes de dados activos são os seguintes:

- *Dados derivados:* Atributos de objectos ou relações podem ser definidos através de procedimentos num modelo de objectos, funcional, ou relacional estendido.
- *Regras:* Um par padrão-acção pode ser activado por uma actualização da base de dados que muda o valor do predicado para verdadeiro. As restrições de integridade são semelhantes às regras, mas não fazem actualizações da base de dados, apenas sinalizam erros devidos a modificações incorrectas de atributos.
- *Agentes:* As acções podem ser espontâneamente iniciadas por regras, mas executam como processos independentes, em paralelo com o processo que dispara as regras. As acções iniciadas por agentes podem executar as mais diversas operações, não estando portanto limitados a actualizações da base de dados.

Numa linguagem de programação de base de dados, todas as acções são implementadas na mesma linguagem e ambiente do resto da aplicação. Nos sistemas de bases de dados estendidos, que têm um ambiente de execução diferente para a base de dados e aplicação, a sua utilidade poderá, por razões óbvias, ser mais reduzida.

3.2.6 Tipos e Herança

Depois de termos definido objectos e suas capacidades, nomeadamente como podemos associar por objectos propriedades tais como: atributos, relações, e procedimentos, vamos agora analisar como é que são definidos os tipos dos objectos.

Tipos

No modelo relacional, as tabelas podem ser consideradas como sendo tipos, onde as definições dos atributos da tabela constituem a definição do tipo, e as linhas da tabela são instâncias do tipo.

Os tipos de objectos são normalmente designados por classes nas linguagens orientadas por objectos, mas o termo classe tem associado pelo menos três significados diferentes:

1. Uma classe define um tipo de objecto—estrutura (atributos e relações dos objectos deste tipo) e comportamento (quais os métodos associados a este tipo).

2. Uma classe define o conjunto de objectos que têm um determinado tipo.

3. Uma classe define um objecto representativo de um determinado tipo de objectos, isto é, uma classe é um meta-objecto. Este meta-objecto tem propriedades (atributos, relações e métodos) que são usadas para definir os objectos desse tipo.

O nome de objecto representativo de um tipo é normalmente usado para conciliar estes três aspectos.

É possível associar valores de atributos por defeito e partilhados ao objecto representativo. Um valor por defeito é lido do objecto representativo sempre que a aplicação tenta ler o atributo duma instância. Um atributo partilhado aparece em todas as instâncias do tipo, mas todas elas referenciam a mesma cópia aramazenada no seu objecto representativo.

Os tipos são normalmente definidos através de linguagens de definição de dados.

Declarações e Variáveis

Num SGBDO, a distinção tradicional entre uma LDD e uma LMD torna-se menos clara. A LMD contém normalmente uma linguagem de interrogação e uma linguagem de programação. A LDD define tipos de objectos em sintaxe idêntica à utilizada no modelo relacional. No entanto, o utilizador também pode declarar procedimentos e variáveis associadas aos tipos de objectos. Para complicar ainda mais, alguns sistemas permitem operações que alteram os tipos em plena execução.

Num modelo de objectos, o utilizador pode definir múltiplas variáveis que são colecções de objectos de um determinado tipo. Na maior parte das implementações relacionais, a única colecção de objectos de um determinado tipo que pode ser referenciada permanentemente, é a própria tabela.

Na maioria dos SGBDOs os nomes das variáveis são transitórios, mas existem SGBDOs em que as variáveis são persistentes, mantendo o seu valor entre execuções dos programas. As variáveis persistentes tornam o SGBDO ainda mais poderoso, permitindo que sejam dados nomes a colecções de objectos, sem se ter de recorrer à criação de novos tipos de objectos para representar determinadas colecções.

Em algumas linguagens de programação orientadas por objectos, as variáveis não são declaradas, sendo o seu tipo determinado em tempo de execução, o que aumenta a flexibilidade mas penaliza em termos de desempenho (Smalltalk).

O âmbito de uma variável depende do tipo de declaração e varia de sistema para sistema. Alguns sistemas permitem a declaração de variáveis cujo âmbito pode ser global, local, ou ainda o de determinado grupo de utilizadores. Isto constitui uma extensão natural do sistema de tipos, aliás tal como acontece nas bases de dados relacionais com a definição de grupos de utilizadores como donos de determinadas tabelas.

Tipos de Dados Literais

Existem dois tipos de dados: literais e entidades. Dados literais são valores simples tais como inteiros, datas, caracteres; entidades são grupos de valores, a que chamamos objectos, e que são referenciados por OIDs. No modelo de objectos, a distinção entre eles é pequena, pois são manipulados por uma linguagem de programação que permite a declaração uniforme dos dois tipos. Os sistemas relacionais estendidos oferecem dois mecanismos distintos para definição de novos tipos, um para literais, e outro para entidades.

A definição de novos tipos de dados literais necessita de informação sobre o espaço necessário para armazenar os valores, procedimentos para codificar e descodificar os valores, e outras características para utilização pelos métodos de acesso, etc. Por estas razões, a definição de novos tipos literais não pode ser encarada como uma tarefa fácil, muito embora seja muito útil.

Hierarquia de Tipos

Muitos modelos de dados avançados incluem o conceito de generalização (herança, definição de subtipos, ou de subclasses), que permite definir uma hierarquia de tipos de objectos.

Devido aos diferentes significados atribuídos ao termo classe, existem por consequência diferentes tipos de generalizações:

- *Especificação:* Os subtipos podem ser definidos com base em valores de determinados atributos. Por exemplo, podemos definir subtipos de `Documento`: Textos ou Programas.
- *Classificação:* Os subtipos podem ser usados como conjuntos para classificar objectos. A classificação é um caso particular de especificação, mas por ser tão frequente é considerado separadamente.
- *Especialização:* Os subtipos podem adicionar atributos ou métodos a um supertipo.
- *Implementação:* Subtipos podem oferecer diferentes implementações de determinados métodos definidos num supertipo. Os subtipos podem ter a mesma especificação (são idênticas para quem os invoca) mas diferentes implementações, ou a semântica dos procedimentos ser igual, mas os operandos diferirem—caso particular do conceito de *overloading* de operadores, no qual o efeito de uma operação depende dos (tipos dos) operandos [Boo91].

Todos estes diferentes tipos de generalização não são mútuamente exclusivos, e são normalmente suportados por um mesmo mecanismo. Os subtipos são usados em combinação com herança por forma a fornecer um mecanismo de abstracção a todos os quatro tipos de generalização. Um subtipo herda a definição de todas as propriedades associadas ao seu supertipo.

Em alguns SGBDOs, um subtipo pode alterar a definição de uma propriedade de um supertipo, simplesmente definindo um componente com o mesmo nome. Para aqueles subtipos que não modificam as propriedades herdadas, alguns sistemas permitem a concatenação da sua definição com as dos seus supertipos.

Hierarquias Avançadas

As hierarquias de tipos podem ter um significado mais sofisticado do que aquele até agora mencionado. Os tipos podem ter múltiplos supertipos, ou objectos podem ter tipos múltiplos. Os SGBDOs possuem diferentes mecanismos para classificar objectos em subtipos, e mecanismos diferentes para a definição dos próprios subtipos.

A maioria dos SGBDOs só permite que os objectos tenham um único tipo. No entanto, algumas sistemas permitem que os objectos possam ter mais do que um tipo, de acordo com predicados que os objectos devem satisfazer. Podendo os objectos ter tipos múltiplos

ou serem definidos tipos de objectos através de predicados, tornam-se necessárias restrições de integridade adicionais no sistema de tipos, por forma a não permitir o aparecimento de situações anómalas.

Muitos SGBDOs suportam herança múltipla, isto é, um tipo poder herdar propriedades de mais do que um supertipo. A herança múltipla contribui para uma redução do número de tipos e complexidade de um esquema de base de dados e programas de aplicação. A herança múltipla é especialmente importante quando existe mais do que uma hipótese de classificação ortogonal dos objectos. Como a herança múltipla é algo mais raramente necessário do que a herança simples, alguns sistemas não o suportam por esta razão.

Evolução do Esquema

Muitos SGBDOs permitem ao utilizador a modificação das definições de tipos, embora diferindo no apoio dado ao utilizador para levar a cabo tais modificações.

Existem três possíveis implicações da modificação de um esquema de dados: modifi cação dos programas que utilizam o esquema antigo, modificação das instâncias dos tipos modificados, e alterações no resto do esquema.

Destas implicações as mais significativas são as relativas aos programas. Mesmo no caso de serem feitas apenas alterações a um atributo de um tipo de objecto, no pior dos casos, precisarão ser alterados os métodos associados com o tipo de objectos modificado. Mas como a alteração da interface de um destes métodos poderá por sua vez invalidar código relativo a outros tipos de objectos, a necessidade de alteração de muitos programas pode assim ser propagada.

A maioria dos SGBDOs não auxiliam na modificação de programas existentes devido a actualizações de esquemas.

As alterações possíveis a um esquema podem ser classificadas da seguinte forma:

1. Alteração das componentes de um tipo: atributos e métodos (acrescentar, tirar, alterar o nome, alterar o tipo, herdar uma definição modificada).

2. Alteração do grafo de herança (acrescentar, tirar, ou alterar ordem de relações de herança entre tipos).

3. Alteração dos próprios tipos (acrescentar, tirar, alterar).

A maioria dos sistemas permitem alterações do tipo 3. Aqueles que têm uma hierarquia de tipos permitem alterações do tipo 2 apenas para tipos que não tenham instâncias. Sendo possíveis estas alterações, consegue-se qualquer uma das outras criando novos tipos, novas instâncias, copiando valores das instâncias antigas, etc.

Alterações de nomes e introdução ou eliminação de métodos, são alterações que apenas envolvem modificações do próprio esquema.

As alterações que afectem determinados objectos poderão ser feitas imediatamente na altura da alteração do esquema, ou apenas quando as instâncias forem utilizadas. Esta alternativa implica que seja utilizada uma representação de tipos que permita a coexistência de instâncias de um tipo com diferentes conjuntos de atributos e relações. Embora mais difícil de implementar, esta hipótese poderá ser útil pois evita que toda a reconfiguração do esquema seja feita imediatamente após a alteração do esquema.

Muitas aplicações necessitam de lidar com transformações de esquemas arbitrariamente complexas, só definiveis recorrendo a linguagens de programação ou linguagens de interrogação.

Mais importante do que automaticamente fazer as modificações dos programas e dados, é os sistemas disponibilizarem boas ferramentas de apoio ao utilizador final que possam auxiliar nestas transformações de esquemas, dados, e programas existentes.

Definição Dinâmica de Tipos

Um tópico relacionado com a evolução de esquemas é a definição dinâmica de tipos. Sem definição dinâmica de tipos, é necessário declarar todos os tipos que o programa vai usar antes deste ser executado.

Existem dois problemas com a definição dinâmica de tipos: definição do novo esquema, e manipulação de objectos utilizando o novo esquema. O primeiro problema é de simples resolução, bastando que para isso o SGBDO disponibilize operações sobre o esquema que possam dinâmicamente ser invocadas. O segundo problema pode ser mais dificil, por duas razões:

1. Os programas antigos eventualmente afectados pelas modificações do esquema não deverão ser executados na nova estrutura da base de dados, por poderem originar resultados inesperados. Tal poderá ser conseguido se o SGBDO mantiver informação sobre datas e/ou versões de esquemas e programas. Neste caso, e porque um pedido de alteração dum determinado tipo do esquema pode ser feito simultâneamente com a utilização de objectos desse mesmo tipo, deverão existir mecanismos de controlo de concorrência para o esquema ou controlo de versões.

2. As interfaces de programação usuais para a maioria dos SGBDOs são compiladas, pelo que se torna impossível a manipulação de objectos utilizando um novo esquema, excepto se através de uma interface dinâmica especial, interface esta que alguns SGBDOs de facto oferecem especialmente para este propósito.

Associação Dinâmica

Associação dinâmica é a capacidade de associar em tempo de execução os símbolos de um programa escrito numa determinada linguagem às suas definições (ou declarações de tipos).

Associação dinâmica pode ser aplicada a tipos, variáveis, atributos, métodos, ou qualquer outro tipo de símbolos de uma linguagem. Como os nomes dos atributos e métodos são interpretados no contexto de uma hierarquia de tipos, a associação poderá envolver a visita a toda a hierarquia para encontrar a definição.

A associação dinâmica é um conceito das linguagens de programação, não um conceito das bases de dados, mas deve ser suportado pelos SGBDOs. Em sistemas de bases de dados estendidas, e em implementações de linguagens de programação de bases de dados não baseadas em compiladores, a associação dinâmica pode ser fornecida pela implementação do SGBDO. Tipos dinâmicos e associação dinâmica poderão ser muito difíceis de implementar em linguagens de programação de bases de dados que não suportem estas capacidades. No entanto, tal problema poderá ser resolvido se estiver disponível uma linguagem de interrogação interpretada, que poderá então ser usada para manipular os novos tipos e símbolos.

3.2.7 Linguagens de Interrogação e de Programação

Uma das mais importantes características de um sistema de base de dados é a linguagem que fornece para extracção, manipulação, e definição de dados. Nos sistemas de bases de dados relacionais, utilizamos o termo linguagem de interrogação para referir aquela linguagem, da qual existiam duas sublinguagens LMD e LDD. Um SGBDO fornece uma linguagem de base de dados que inclui capacidades de linguagem de interrogação e capacidades de linguagem de programação, para além de outras.

Para melhor entendermos as capacidades das linguagens de interrogação e programação dos SGBDOs, iremos confrontar estas linguagens em relação aos seguintes sete aspectos :

- *Ambiente:* As linguagens de programação de bases de dados são extensões das linguagens de programação existentes. Extensões de sistemas de bases de dados, por outro lado, têm uma linguagem de base de dados separada da linguagem de programação da aplicação—as duas linguagens são executadas em ambientes diferentes.

- *Resultados de interrogações:* Os SGBDOs diferem no tipo de resultados que podem ser obtidos por interrogações declarativas—como em SQL, onde o utilizador apenas diz o que pretende obter e não como o pretende obter. Uns SGBDOs não fornecem linguagem de interrogação declarativa; outros fornecem uma linguagem que produz conjuntos, listas de objectos, relações; e outros uma linguagem que produz qualquer tipo de resultados.

- *Encapsulamento:* Alguns SGBDOs suportam na íntegra o conceito de encapsulamento, pelo que apenas os métodos declarados como públicos são visíveis para os utilizadores da linguagem da base de dados. Outros sistemas violam o conceito apenas em relação à linguagem de interrogação. Por último existem sistemas que não têm nenhum mecanismo de encapsulamento.

- *Dados virtuais:* Dados definidos usando atributos derivados e relações, quando transparente para o utilizador a forma de obtenção dos seus valores (armazenamento ou derivação) denominam-se de dados virtuais. Os dados virtuais promovem indepêndencia dos dados, permitindo que o esquema conceptual seja alterado sem necessidade de modificação dos programas existentes.

- *Modelo de dados:* O modelo de dados afecta a sintaxe e semântica da linguagem de base de dados, consoante seja relacional, funcional, ou orientado por objectos.

- *Operações estendidas:* Os SGBDOs normalmente estendem as linguagens de in terrogação convencionais com novas operações tais como fecho transitivo, recursividade, e regras.

- *Normas:* Um aspecto importante na utilização de um SGBDO é a compatibilidade com normas existentes de linguagens de interrogação (SQL), e compatibilidade com outros SGBDOs.

Como os gestores de objectos não costumam incluir nenhuma linguagem, e os geradores de bases de dados deixam a definição da linguagem ao cuidado do implementador, apenas nos interessarão as duas arquitecturas de sistemas de bases de dados estendidas e linguagens de programação de bases de dados.

Ambientes de Linguagens

Os SGBDOs apresentam abordagens diferentes para os aspectos anteriores dependendo do ponto de partida: linguagem de interrogação ou linguagem de programação. Analisemos em que difere a concepção da linguagem nas duas arquitecturas acima mencionadas:

- *Sistemas de bases de dados estendidos* fornecem uma linguagem de interrogação e linguagem de programação, com diferentes sistemas de tipos e ambientes de execução. Ambas são computacionalmente completas, podendo-se incluir instruções da linguagem de interrogação na linguagem de programação por forma a transferir dados entre os dois ambientes.

- *Linguagens de programação de bases de dados* têm apenas um ambiente de execução, linguagem procedimental, e um sistema de tipos. A sintaxe da linguagem de programação é normalmente estendida com sintaxe da linguagem de interrogação.

As diferenças entre as duas arquitecturas são ligeiras. A maior diferenciação consiste na capacidade da linguagem de base de dados ter acesso a todas as variáveis, operações e procedimentos da aplicação, e na linguagem ser completa, quer em termos computacionais, quer em termos de recursos. Se a linguagem falha num destes aspectos, então passamos a ter um sistema com dois ambientes—um sistema de bases de dados estendido.

A existência de dois ambientes nos sistemas de bases de dados estendidos pode ser fonte de problemas para o programador, por tornar necessária, por uma lado, a conversão de dados entre os dois ambientes, o que penalizará em termos de desempenho, e por outro a aprendizagem de duas linguagens, com manuais diferentes e possivelmente de empresas diferentes.

Estes problemas de utilização de dois ambientes têm sido globalmente referidos como impedância semântica entre as linguagens de programação e linguagens de interrogação de base de dados numa aplicação.

Impedância Semântica

O problema de impedância semântica entre linguagens de programação e linguagens de interrogação não é um problema recente. Os sistemas de bases de dados relacionais tipicamente fornecem uma linguagem de interrogação declarativa, sem capacidades de programação. O programador terá que escrever as aplicações numa linguagem de programação recorrendo às estruturas de dados próprias da linguagem, e utilizar a linguagem de interrogação para transferir dados entre os dois ambientes.

Numa base de dados relacional estendida, ou num SGBDO com uma arquitectura de base de dados estendida, os problemas que surgem são idênticos aos anteriores.

As abordagens no sentido de diminuir a impedância semântica são:

- *Interrogações pré-processadas:* A abordagem mais óbvia poderá ser a escrita de interrogações que extraem da base de dados todos os objectos de que a aplicação possa precisar, traduza os objectos para a representação da linguagem de programação, e no fim volte a copiá-los para a base de dados. Esta abordagem permite que os objectos sejam manipulados eficientemente pela linguagem de programação, mas requerem trabalho extra tanto no arranque da aplicação como na sua finalização. Com esta abordagem não se consegue utilizar todas as potencialidades do SGBD durante a execução da aplicação, e limita significativamente o acesso concorrente.

- *Representação da base de dados:* Outra abordagem consiste em copiar apenas um único campo ou objecto da base de dados, operando os dados na base de dados, em vez de ser no programa. Esta abordagem tem o problema de diminuir a eficiência das operações.

- *BLOBs:* Em vez de utilizar um esquema de dados, toda a informação sobre os objectos pode ser codificada numa estrutura de dados da linguagem de programação e armazenada na base de dados como um único campo BLOB. O campo poderá ser eficientemente manipulado, mas muitas das vantagens da utilização de um SGBDO perdem-se, uma vez que interrogações, concorrência, e outras operações deixam de poder ser efectuadas, pois o SGBDO não se preocupa com a estrutura interna do BLOB.

- *Linguagem de base de dados procedimental:* Uma outra abordagem consiste na escrita da maior parte da aplicação na linguagem de interrogação, desde que esta suporte operações de programação. A desvantagem desta abordagem reside no facto de que os programas executados na linguagem de interrogação poderão não ter acesso a informação global ou operações que estão disponíveis na aplicação.

- *Tipos de dados de programação estendidos:* Existem extensões de linguagens de programação que possibilitam a manipulação de relações da mesma forma que uma outra estrutura de dados do programa. A integração dos modelos de dados da base de dados e da linguagem de programação não se encontram integrados, não fornecendo estruturas de dados persistentes, aliás como seria desejável. Esta abordagem também tem os seus problemas em termos de desempenho, dado que o modelo relacional é ineficiente para a representação de estrutras de dados complexas.

- *Linguagens de programação persistentes:* Recentemente, têm sido feitas tentativas no sentido de dotar as linguagens de programação com estruturas de dados persistentes, armazenando os dados num SGBDO relacional estendido. Embora esta abordagem seja a mais prometedora, estes sistemas ainda não conseguem eliminar a impedância semântica, porque os modelos de dados da linguagem de programação e da base de dados são diferentes. Por serem muito semelhantes os modelos de objectos das linguagens de programação e os das bases de dados, tem incidido neles o trabalho feito na área das linguagens de programação de bases de dados.

Apesar das muitas tentativas para eliminação da impedância semântica das linguagens nos sistemas de bases de dados, nenhuma das abordagens resolve o problema. As linguagens de programação de bases de dados são as que conseguem uma diminuição mais significativa, por tornarem a linguagem de base de dados muito mais poderosa.

Resultados de Interrogações

Os SGBDOs de uma forma geral adoptam uma das quatro abordagens seguintes em relação às linguagens de interrogação, abordagens estas que se podem distinguir pela forma dos resultados que permitem:

- *Inexistência de interrogações:* Alguns sistemas ainda não permitem o acesso a conjuntos de objectos através de um linguagem de interrogação—torna-se necessário escrever programas que navegam pelos objectos manualmente. Em algumas

aplicações é desejável a escrita de interrogações em programas, porque é mais conveniente e natural, ou mesmo por ser a única maneira de obter o desempenho desejado.

- *Colecções:* Alguns sistemas permitem a utilização de interrogações que não operam em toda a base de dados, mas apenas numa colecção de objectos, que poderá ser toda uma classe de objectos, ou apenas alguns objectos que satisfazem determinadas condições impostas.
- *Relações:* Os SGBDOs relacionais estendidos fornecem uma linguagem de in terrogação cujos resultados são relações. Os resultados podem ser conseguidos a partir de quaisquer dados da base de dados. Estas linguagens de interrogação igualam ou superam o poder expressivo do SQL ou outras linguagens de interrogação relacionais.
- *Qualquer tipo de resultados:* Existem sistemas com linguagens de interrogação que produzem qualquer tipo de resultados: literais, objectos, listas de objectos, ou vectores de dados.

Os SGBDOs que produzem resultados do tipo colecção podem ser subclassificados conforme permitam junções com base em valores ou apenas junções com base em referências. As junções de valores são junções baseadas em comparações arbitrárias de valores de atributos. Junções de referências são assim chamadas por seguirem as referências entre objectos.

Encapsulamento

Encapsulamento garante independência dos dados através da implementação de métodos, permitindo que a parte privada do objecto seja alterada sem afectar programas que utilizam esse tipo de objectos. Existem dois requisitos para o encapsulamento:

1. Alguns componentes de um objecto devem ser públicos, enquanto outros são privados.
2. Os métodos definem um mapeamento entre os componentes privados e públicos.

A definição restrita de encapsulamento não é a mais conveniente no contexto das bases de dados, pelo que a definição que se utilizará será a seguinte:

1. Procedimentos e dados podem constar da parte pública do objecto.
2. Os métodos podem ser definidos numa linguagem procedimental ou declarativa.
3. Os métodos, embora controladamente, podem manipular dados exteriores ao objecto.

Por forma a permitir o ponto 3 mas não quebrar o encapsulamento, aumentou-se o número de partes da definição de um tipo de objecto: parte pública, protegida e privada. A parte pública é visivel globalmente. A parte protegida é visivel apenas a subtipos. A parte privada é visivel apenas aos métodos definidos na classe, ou classes ou procedimentos declarados como "amigos" da classe. A noção de amigos e de parte protegida permite que grupos de classes cooperem de uma forma mais flexível na implementação de uma abstracção.

Dados Virtuais

A noção de dados virtuais é análoga à das vistas relacionais. Dados derivados podem ser úteis para muitas coisas, mas são-no especialmente para garantir independência de dados. Sempre que um esquema é alterado, podem ser definidos dados derivados para os atributos antigos e relações, por forma a que os programas e interrogações existentes continuem a poder funcionar.

Dados derivados definidos utilizando métodos orientados por objectos numa linguagem procedimental podem ser mais poderosos que as vistas relacionais, porque podem ser especificadas acções diferentes para leitura e escrita, e poderem ser especificadas operações arbitrárias.

Embora os dados virtuais contribuam para independência dos dados, não são no entanto suficientes, porque fazem o mapeamento mas não ocultam a parte privada do objecto.

Uma alternativa para derivar métodos de actualização automática para dados derivados seria permitir ao utilizador especificar explicitamente procedimentos para ler e escrever valores de dados derivados.

Independência de Dados

O factor mais importante para independência física dos dados é a utilização de uma linguagem de interrogação declarativa em programas e métodos, por permitirem especificar o que o utilizador pretende sem ter de especificar como fazer para o obter. Caberá ao optimizador de interrogações, com base na organização dos dados, determinar automaticamente a forma mais eficiente de o fazer.

A independência física dos dados pode-se perder em linguagens de interrogação que combinam facilidades de programação, uma vez que a linguagem resultante não é fácil de optimizar, tal como são as operações relacionais de selecção, projecção e junção. A simplicidade algébrica de optimização de linguagens que são mais declarativas do que procedimentais constitui um factor importante.

O encapsulamento garante uma independência lógica e física dos dados. O encapsulamento permite que estruturas de dados privadas associadas a um objecto possam ser alteradas arbitrariamente, tendo o programador, no entanto, de modificar manualmente todos os métodos afectados pelas alterações.

Modelos de Dados

O modelo de dados utilizado para a linguagem faz com que as linguagens dos SGBDOs apresentem diferenças sobretudo ao nível sintactico. Pela subjectividade do processo de determinação da representação mais natural e sintaxe a utilizar para a definição de uma aplicação, a escolha do modelo de dados de um SGBDO é talvez o aspecto mais controverso.

Operações Estendidas

O poder expressivo de uma linguagem de interrogação varia entre os SGBDOs. Poder expressivo adicional é importante porque permite que maior parte da aplicação seja codificada duma forma concisa ou por garantir um nível de independência física superior à da linguagem de programação. Alguns SGBDOs permitem operações como: fecho transitivo, reconhecimento de padrões de texto, etc.

Sistemas de Regras

Uma outra característica útil dos SGBDOs é um sistema de regras. Um sistema de regras é uma peça fundamental para que um SGBDO possa ser usado para gestão de conhecimento.

Os sistemas de regras podem ser incorporados directamente no SGBDO, ou então existirem como sistemas periciais que têm como base um SGBDO. A vantagem desta alternativa é que permite que o sistema pericial e o SGBDO sejam desenvolvidos, mantidos, e optimizados separadamente.

No futuro, irão ser talvez incorporados nas linguagens de interrogação, operações e sistemas de regras muito mais complexos, eventualmente até todo o poder de inferência lógica das bases de dados dedutivas.

Normas

Ainda não existem nenhumas normas para os SGBDOs, sejam eles baseados nos modelos orientados por objectos, relacional estendido, ou funcional, embora exista já uma proposta para tal, elaborada pelo ODMG, e que será apresentada na secção seguinte [CAD+94].

Relativamente às normas, existem quatro aspectos a considerar para os SGBDOs:

- *Modelo de dados:* Não existe um único modelo de objectos, existem vários, tal como existem vários modelos de dados relacionais estendidos e funcionais.
- *Linguagem de interrogação:* Porque, mesmo para SGBDOs que utilizam um mesmo modelo de dados, não existem linguagens de interrogação comuns, é impossível a escrita de interrogações que possam ser executadas em mais do que um SGBDO.
- *Linguagem de programação:* Existe a necessidade de uma norma em termos de linguagem de programação para os SGBDOs, para permitir que os programas possam ser transferidos entre diferentes sistemas.
- *SQL:* Como o SQL é a única norma universalmente reconhecida para acesso a base de dados, embora indirectamente, tem influência nos novos SGBDOs.

Os sistemas relacionais estendidos têm vantagens em todos estes aspectos, uma vez que utilizam o mesmo modelo relacional apoiado pelos grupos de normalização.

Relativamente ao modelo de objectos, existem grupos a trabalhar no sentido de encontrar normas de acordo com todos estes aspectos, mas os resultados efectivos deverão ainda demorar a aparecer [Mir92, SSU91]

A falta de normas tem-se reflectido de uma forma negativa na aceitação da tecnologia oferecida pelos novos SGBDOs, especialmente devido a problemas de portabilidade.

3.2.8 Persistência, Concorrência e Recuperação de Erros

A gestão de dados de objectos levanta novas questões em relação a concorrência e recuperação de erros para bases de dados, porque os SGBDOs são destinados a utilizadores com novos tipos de requisitos em termos de concorrência: transacções com duração muito superior ao normal, armazenamento de múltiplas versões dos mesmos objectos na base de dados, e a influência da semântica de objectos no tipo de controlo de concorrência.

Nos sistemas relacionais, o mecanismo de transacções suporta os mecanismos de controlo de concorrência e recuperação que se adequam às aplicações comerciais. Os SGBDOs utilizam mecanismos de transacções modificados, bem como novas técnicas— versões, por exemplo.

Persistência

Concorrência, recuperação, integridade física, e integridade lógica são aspectos importantes porque os dados de um SGBD são persistentes. Num SGBDO podem coexistir dados persistentes e transitórios.

Dados transitórios deixam de existir quando um programa termina. Dados persistentes continuam a existir até não mais serem usados. Nos SGBDs os dados transitórios são deixados ao cuidado das linguagens de programação, e os dados persistentes às bases de dados.

As facilidades de linguagem de um SGBDO podem ser utilizadas tanto para dados persistentes como para transitórios. Os SGBDOs podem diferir na forma de tornar um objecto persistente. As hipóteses são as seguintes:

- *Por tipo:* Um objecto pode ser tornado persistente quando criado, com base no seu tipo (tipos persistentes, e tipos transitórios).
- *Por pedido explícito:* O utilizador pode explicitamente especificar a persistência de um objecto, ou quando da sua criação, ou em alguns sistemas, em qualquer outra altura.
- *Por referência:* Alguns sistemas determinam automaticamente a persistência de objectos por serem acessíveis por outros objectos que se sabe serem persistentes.

As duas primeiras hipóteses existem apenas por conveniência, visto os conceitos de tipo e persistência serem ortogonais, razão pela qual a fusão dos dois não será muito correcta. A última hipótese é sem dúvida a mais poderosa pois liberta o utilizador de ter de decidir para cada objecto se é persistente ou não.

Transacções

Os SGBDOs modificam os mecanismos de transacções por forma a permitir transacções longas e transacções dentro de transacções. As primeiras, são transacções que podem durar horas ou mesmo dias. A possibilidade de definição de transacções dentro de uma transacção possibilita que se termine ou interrompa uma transacção sem no entanto afectar aquela na qual se inclui.

Existem duas utilizações para as transacções:

1. As transacções podem ser utilizadas para controlo de concorrência para os casos em que as características de acesso à base de dados forem similares às das aplicações comerciais tradicionais. No entanto, em situações em que as transacções podem durar horas ou dias, são necessários novos mecanismos, por razões de eficiência. As soluções sugeridas são transacções conversacionais, e versões.
2. As transacções são importantes para recuperação e para execução de grupos de operações indivisíveis. As transacções de curta duração são normalmente suficientes para estes casos, mas se forem usadas transacções de longa duração, então será necessário definir transacções dentro de outras transacções ao longo da transacção de longa duração com o objectivo de tornar possível a sua recuperação.

O problema das transacções de longa duração é um problema de implementação, mas com técnicas apropriadas, elas tornam-se possíveis.

Versões de Objectos

Com vista ao controlo de concorrência existe uma alternativa às transacções que tem ganho cada vez maior popularidade: versões. A utilização de versões múltiplas de objectos para permitir a coordenação entre múltiplos utilizadores e diferentes cópias foi já feita numa série de produtos e protótipos, embora a sua semântica ainda seja bastante diferente de caso para caso.

A funcionalidade básica requirida para versões é a criação e destruição de versões de objectos. Deverá também ser possível a destruição ou arquivo de versões antigas de objectos quando deixem de ser precisas.

As versões num SGBDO podem simplificar consideravelmente aplicações que fazem a manutenção de múltiplas versões de dados.

Os históricos de versões podem ser muito complexos. A situação mais normal será a de as novas versões acontecerem numa sequência linear, mas é frequente a existência de pontos de ramificação de versões—duas ou mais versões independentes e de pontos de união—criação de uma nova versão com base em duas ou mais versões independentes.

Todos os SGBDOs com versões fornecem mecanismos para examinar este histórico, e muitos permitem a ocorrência de pontos de ramificação e união. Um SGBDO poderá permitir que as versões sejam utilizadas no próprio esquema, auxiliando portanto no seu processo de evolução.

As versões de objectos introduzem um problema: as referências para um objecto devem ser actualizadas sempre que uma versão do objecto é criada ou destruída.

Configurações

Uma configuração é uma colecção de versões de objectos da base de dados que são mutuamente consistentes. Devem ser mantidas referências entre objectos de uma configuração para as versões apropriadas sempre que são feitas actualizações dos objectos.

Existem duas abordagens para as configurações num SGBDO:

1. O SGBDO pode fornecer um mecanismo mas não uma política para gestão de configurações, deixando a sua gestão a cargo do utilizador.

2. O SGBDO implementa as configurações automaticamente, com algumas opções especificadas pelo utilizador.

Na primeira alternativa, o conjunto de mecanismos que os SGBDOs fornecem para manutenção da consistência das versões de referências, conjuntamente com métodos definidos pelo utilizador, possibilitam que o programador implemente uma variedade de políticas de gestão de configurações. A filosofia geral destes sistemas é: os utilizadores definem a sua própria gestão de configurações com base nas primitivas colocadas à disposição.

A segunda alternativa consiste em incorporar a gestão de configurações no próprio SGBDO. Existem muitas formas de o conseguir, dependendo da política de tratamento de versões, referências e interrogações. Uma forma possível é a seguinte:

1. As configurações são representadas por objectos especiais, com OIDs, nome, e outra informação.

2. Existe sempre uma configuração corrente numa sessão da base de dados; o utilizador pode abrir uma configuração existente ou criar uma nova para alterar a corrente.

3. Sempre que é feita uma alteração à base de dados, é feita no contexto de uma configuração. Novas versões de um objecto criadas como resultado de actualizações durante a sessão são marcadas com a configuração na qual foram criadas.

4. Todas as leituras de dados durante a sessão são feitas no contexto da configuração corrente.

As referências não são a única maneira de acesso a objectos—os objectos podem ser acedidos utilizando uma linguagem de interrogação. É fácil perceber que as interrogações podem ser executadas no contexto de uma determinada configuração, se a configuração fizer parte do SGBDO. No caso da gestão de configurações ser feita pelo utilizador, a linguagem de interrogação necessitará de fornecer mecanismos menos elegantes para dar hipótese ao utilizador de especificar que versões de objectos pretende.

Informação Temporal e de Utilização

Alguns sistemas possibilitam que os objectos sejam marcados por forma a se poder identificar qual o utilizador que pela última vez o alterou e quando é que isso aconteceu.

Esta informação (utilizador-tempo) pode ser lida de um registo como qualquer outro atributo, e pode ser usado em interrogações para examinar a história da base de dados.

Esta capacidade particularmente útil para interrogações históricas e aplicações comerciais tradicionais (sistemas de apoio à decisão), não substitui no entanto versões, transacções, ou outros mecanismos para controlo de concorrência.

Granularidade da Concorrência

Os mecanismos de controlo de concorrência operam a um nível de granularidade de objectos, podendo no entanto operar a outros níveis de granularidade.

Poderá ser mais eficiente operar a níveis de granularidade físicos, como páginas ou segmentos, embora isto não necessite ser percebido pelo utilizador.

3.3 Características Gerais de um SGBDO

Os produtos existentes de Sistemas de Gestão de Bases de Dados de Objectos, apesar de não serem normalizados, reúnem uma série de características comuns. Em [Cat91] são enumeradas as capacidades que segundo o autor todos os SGBDOs devem fornecer.

Embora um SGBDO só possa ser precisamente avaliado tendo em vista uma determinada aplicação, existem no entanto muitas características genericamente pretendidas do ponto de vista de utilização de um SGBDO, e necessidades comuns a um vasto conjunto de aplicações.

Antes de restringirmos a nossa perspectiva à especificidade da aplicação concreta para a qual se pretende especificar o SGBDO—a ferramenta MOOT, vamos analisar as suas características gerais.

A maior parte do debate existente sobre a próxima geração de SGBDs incide essencialmente nos dois modelos mais populares: os modelos objecto e os modelos relacionais estendidos.

Nesta secção serão enunciadas todas as características destes sistemas, e no final serão assinaladas aquelas que se pretendem ver implementadas no SGBDO a desenvolver futuramente.

De todas estas características, algumas são consideradas essenciais; sem elas, não estaremos em presença de um SGBDO. Outras características são apenas desejáveis, no sentido em que facilitam o desenvolvimento de muitas aplicações. Na apresentação das várias características serão diferenciadas as características consideradas essenciais daquelas consideradas apenas desejáveis, assinalando estas últimas com um asterisco (*).

3.3.1 Funcionalidade de Base de Dados

Para termos um SGBDO, devemos começar por ter a funcionalidade mínima de um sistema de base de dados, nomeadamente as capacidades de armazenamento e de acesso a informação. São estas características que permitem distinguir um sistema de base de dados de uma linguagem de programação ou de um sistema operativo, por exemplo.

Armazenamento de Informação

Um sistema de base de dados permite lidar com bases de dados persistentes, estruturadas e possivelmente de grandes dimensões:

- *C1: Persistência.* Um SGBDO deve possibilitar o armazenamento de informação de forma persistente.
- *C2: Esquema.* Um SGBDO deve possuir um esquema da informação que armazena. Os utilizadores devem ser capazes de examinar o esquema como informação persistente, e de criar e destruir tipos de dados.
- *C3: Memória Secundária.* Um SGBDO deve ser capaz de lidar eficientemente com grandes volumes de informação, através da gestão de dispositivos de memória secundária, índices, apontadores, e outras técnicas.

Acesso à Informação

Um sistema de base de dados deve fornecer capacidades básicas para controlo de acesso e controlo de concorrência:

- *C4: Transacções.* Um SGBDO deve possuir um mecanismo de transacções ou outro equivalente para controlo de concorrência e recuperação de erros.
- *C5*: Segurança.* Um SGBDO deve fornecer mecanismos que permitam a associação de permissões a utilizadores e informação e mecanismos para controlo de acesso.

Das cinco características apresentadas relativas a armazenamento de informação e controlo de acesso apenas a última relativa à segurança é considerada opcional num SGBDO. Isto porque muitas das aplicações que consideramos não têm grandes requisitos em termos de segurança. (ver secção 2.1).

3.3.2 Objectos e Propriedades

Primeiro, vamos considerar a definição de objectos e suas propriedades. Os objectos devem ter identificadores únicos, e as suas propriedades podem ser atributos, relações, ou operações.

Identidade de um Objecto

É de opinião geral que os identificadores de objectos (OIDs—*Object Identifiers*) são úteis, mas existe no entanto alguma discordância em relação à obrigatoriedade da sua utilização para acesso aos objectos. Por estas razões, consideramos que os OIDs devem ser opcionais, podendo o acesso ser feito ora por chaves primárias, ora por OIDs.

- *C6: Objectos.* Um SGBDO deve permitir armazenar objectos, e o utilizador deve ser capaz de referir estes objectos tanto através de chaves primárias como através de OIDs únicos gerados automaticamente pelo sistema.

Apesar de o acesso poder ser opcionalmente feito por OIDs ou por chaves primárias, o SGBDO deve no entanto implementar ambos os métodos.

Atributos

Uma grande diferença existente entre SGBDOs e SGBDRs tradicionais consiste no facto dos valores dos atributos de objectos poderem ser de uma vasta variedade de tipos, ao contrário do que acontece com os campos dos registos:

- *C7: Atributos.* Um SGBDO deve permitir que os atributos dos objectos possam ter: valores literais, tais como, inteiros e sequências de caracteres; valores de tamanho variável e de dimensão possivelmente grande, tais como imagens, som e texto; e colecções de valores, como por exemplo, conjuntos, listas e vectores.

Os sistemas orientados por objectos, permitem, de uma forma geral, todos estes tipos de valores. Os sistemas relacionais estendidos, relativamente aos atributos do tipo listas de valores, têm enfrentado alguns problemas, pelo simples facto de as linguagens de interrogação declarativas normalmente não suportarem ordenação. Este facto é particularmente importante para a representação de relações ordenadas, como teremos oportunidade de ver mais à frente em 3.3.2.

Relações

As referências de uns objectos para outros poderá ser feita através de OIDs ou de chaves primárias. Não é necessário que este facto seja transparente para o utilizador, pela simples razão de poder ser sempre conseguida tal transparência através de relações virtuais.

- *C8: Relações.* Um SGBDO deve fornecer um mecanismo para representar as relações entre objectos. Deve ser possível estabelecer relações entre objectos através de OIDs ou chaves primárias.

Se as relações não são representadas por atributos, então C7 deve também ser aplicável a relações: deve ser possível criar relações entre objectos e valores literais tais como inteiros, e deve ser possível especificar uma ordem para as relações.

- *C9: Integridade Referencial.* O mecanismo para representar relações deve manter a integridade referencial, evitando anomalias de actualização na destruição de objectos ou redefinição da relação.

Os sistemas orientados por objectos sem atributos inversos não satisfazem esta exigência porque podem existir duas representações independentes das mesmas relações. Existem alguns produtos relacionais que ainda nem sequer implementam integridade referencial.

Objectos Compostos

A agregação de objectos em objectos compostos é uma capacidade desejável, embora não seja necessária, pelo facto de poder ser conseguida por um programa de aplicação:

- *C10*: Objectos Compostos.* Um SGBDO deve possibilitar a agregação de objectos para operações de destruição, cópia, e controlo de concorrência. Se fôr possível a definição de relações de agregação, poderá ser útil também como critério para localização física dos objectos. A aplicação de operações para o todo o objecto composto deverá ser opcional.

Procedimentos

Para além dos atributos e relações, os objectos também podem ter procedimentos associados:

- *C11*: Procedimentos.* Deverá ser possível associar procedimentos aos tipos de objectos.

Os procedimentos poderão ser invocados directamente, ser utilizados para definir atributos virtuais, ou ser utilizados na linguagem de interrogação.

3.3.3 Sistema de Tipos

No ponto anterior, foram considerados alguns aspectos importantes do sistema de tipos relativamente às propriedades dos objectos. Nesta secção, consideraremos as características importantes de um sistema de tipos para os próprios objectos.

Representação dos Tipos

É um corolário de C2 que o sistema de tipos do SGBDO seja visível tanto da linguagem de programação como da linguagem de interrogação utilizadas pelo programador da aplicação. Por exemplo, um SGBDO pode permitir tipos abstractos de dados na linguagem para serem armazenados como atributos dos objectos, mas se não é possível examinar toda a informação do tipo abstracto de dados numa interrogação, então ainda existe impedância semântica entre as linguagens. Por outro lado, é necessário estender o sistema de tipos da linguagem de programação para que possua um modelo de dados satisfatório para os sistemas de bases de dados. Por exemplo, as linguagens de programação normalmente não permitem a definição de operações associativas sobre um tipo requerido numa linguagem de interrogação, tais como obter todos os objectos de um tipo cujo valor de um campo inteiro esteja entre 4 e 5. De facto, a maioria das linguagens de programação não permite sequer a enumeração de todas as instâncias de um tipo, com ou sem restrições associativas.

Herança

Outra característica em que existe consenso, é a de que a herança de tipos é útil:

- *C12: Herança.* Um SGBDO deve possibilitar a herança de propriedades de supertipos. Opcionalmente, poderá permitir herança múltipla, tipos múltiplos para um objecto, ou tipos com apenas uma instância.

Herança é importante em praticamente todas as aplicações que consideramos anteriormente; sem ela, podem ser necessários muitos mais tipos no esquema, e as operações muito mais difíceis de especificar. As extensões opcionais através de herança enriquecem o esquema, mas todas elas adicionam também complexidade ao modelo de dados e implementação, que poderão não justificar o ganho. Herança múltipla é provavelmente a característica com menos aceitação generalizada; no entanto, e apesar de poder ser útil, a sua utilização terá que se restringir ao estritamente necessário, atendendo a que pode reduzir a eficiência no acesso às propriedades.

Tipos Dinâmicos

Em algumas aplicações, é importante a possibilidade de associar e definir novos tipos durante a execução de uma aplicação. Esta capacidade é também útil na escrita de programas que alteram o esquema de dados.

- *C13*: Tipos Dinâmicos.* Um SGBDO deverá permitir associação dinâmica e pequenas alterações sobre os tipos.

De salientar, que nas linguagens de programação de bases de dados baseadas em linguagens compiladas, poderá se ter que utilizar uma interface de programação especial para aceder aos objectos quando os seus tipos são definidos durante a execução—por exemplo, através de instruções da linguagem de base de dados interpretadas por uma biblioteca de funções especificas para utilização durante a execução. Devido à complexidade adicional associada às alterações de tipos e associação durante a execução, e o pequeno número de aplicações que necessitam destas capacidades, estas são classificadas como desejáveis em vez de obrigatórias.

Evolução do Esquema de Dados

Estando garantida independência lógica num sistema de base de dados, o problema de alteração de um esquema fica bastante simplificado. É no entanto ainda uma questão muito importante atendendo à longevidade das bases de dados. Este problema não surge nas linguagens de programação, porque os tipos perduram para lá dos dados, e não o contrário, como acontece nas bases de dados.

Porque não existe nenhuma capacidade satisfatória para a evolução de um esquema, que inclua muitas das transformações que serão requeridas, não vamos exigir nenhum mecanismo sofisticado para possibilitar a evolução dos esquemas: as capacidades mais importantes serão as que poderão facilitar a programação das transformações do esquema.

- *C14*: Evolução do Esquema.* Um SGBDO deverá providenciar ferramentas para auxiliar o utilizador nas transformações do esquema.

Uma linguagem de interrogação declarativa poderá auxiliar em muito a satisfazer C14, mas as ferramentas gráficas ou linguagens especificamente destinadas à descrição de transformações serão sem dúvida mais úteis. Nenhum SGBDO fornece ferramentas destas, de momento.

Capacidades de alterar nomes, adicionar e remover propriedades de tipos de objectos, e de adicionar e remover tipos de dados, durante a execução, são as capacidades importantes de C13 por forma a que C14 seja possível e que quaisquer programas possam solicitar transformações de esquemas. Sem estas capacidades de transformação durante a execução, será necessário definir estaticamente tanto o esquema novo como o antigo no programa de transformação.

3.3.4 Linguagens de Programação e Interrogação

Porque existem compromissos e interacções entre a utilização de capacidades procedimentais e declarativas, iremos considerar as linguagens de programação e interrogação conjuntamente.

Linguagem de Interrogação

As linguagens de interrogação têm dois grandes objectivos: garantir independência física dos dados, e fornecer uma linguagem de alto nível conveniente para manipulação dos dados.

O acesso directo aos dados numa linguagem de programação não satisfaz nenhum destes requisitos. Por outro lado, uma linguagem de interrogação, embebida numa linguagem de programação, resulta numa linguagem de mais alto nível.

As características base a exigir das linguagens de interrogação são:

- *C15: Linguagem de Interrogação.* Um SGBDO deve ter uma LMD que possa ser usada por um utilizador final para operações *ad hoc*.

- *C16: Independência Física dos Dados.* O processador da linguagem de interrogação deve garantir um alto grau de independência física dos dados, seleccionando automaticamente os métodos físicos de acesso a utilizar para satisfazer uma interrogação.

- *C17: Interrogações em Programas.* Um SGBDO deve suportar uma linguagem de programação que inclua a linguagem de interrogação como um subconjunto. Deve ser possível invocar a linguagem de interrogação da linguagem de programação, e a linguagem de programação da linguagem de interrogação.

Optimização das Interrogações

É difícil escrever um processador de interrogações para um SGBDO que optimize na barreira entre a linguagem de programação e a linguagem de interrogação. Uma linguagem de interrogação declarativa pode ser optimizada através de simples transformações algébricas, enquanto que programas com instruções de controlo não o podem.

- *C18*: Interrogações preferidas.* Os programas devem especificar o acesso aos dados na linguagem de interrogação em vez de o fazerem directamente na linguagem de programação, sempre que fôr prático exprimir o conjunto de operações e objectos desejados na linguagem de interrogação.

A aplicação desta noção pode aumentar significativamente a independência física dos dados, a eficiência, e a legibilidade de um programa de uma aplicação. Quando o acesso aos dados é descrito na linguagem de interrogação, a interrogação pode ser optimizada de acordo com a estrutura da base de dados e enviada para o motor da base de dados (possivelmente remoto), e o resultado da interrogação poderá então ser eficientemente retornado como um bloco único. Numa linguagem de programação, consegue-se garantir independência física dos dados através do encapsulamento de informação. O encapsulamento permite que a implementação de uma propriedade possa ser alterada sem que isso obrigue a alterar os programas que a utilizam, mas é necessário rescrever a propriedade manualmente.

A característica C18 foi uma das motivações da abordagem relacional, em oposição ao acesso navegacional ao nível físico. Acesso navegacional é por vezes citado como sendo um passo atrás na abordagem orientada por objectos, por o modelo relacional tornar transparente para o utilizador os detalhes de navegação. No entanto, e atendendo a que as linguagens de programação orientadas por objectos para bases de dados normalmente disponibilizam linguagens de interrogação com pelo menos o mesmo poder expressivo dos sistemas relacionais, esta critica não é merecida.

Existem casos em que um programa pode ser mais legível se escrito numa linguagem de programação. Existem outros casos em que o programador pode conseguir melhores resultados que o optimizador de interrogações numa parte critica de um CAD, CASE, ou aplicação similar.

Poder da Linguagem de Interrogação

A avaliação do poder da linguagem de interrogação de um SGBDO é algo ainda em evolução. Nos sistemas relacionais, a noção de *ser relacionalmente completa* pode ser usada para avaliar o poder da linguagem de interrogação. Com a introdução de capacidades tais como atributos com colecções de valores, procedimentos, e interrogações sobre tipos arbitrários, não existem ainda modelos para avaliar se uma linguagem de interrogação é ou não completa.

As linguagens de interrogação de alguns SGBDOs operam apenas em colecções, e não permitem o equivalente às junções relacionais entre objectos de tipos diferentes a menos

que um atributo de referência tenha sido explicitamente definido entre os tipos. Existem, no entanto, alguns SGBDOs [D+91] cuja linguagem de interrogação tem poder expressivo muito superior ao cálculo relacional. Permite construtores e operadores sobre conjuntos, listas, tuplos, e podem produzir resultados finais ou intermédios de qualquer tipo.

Em relação ao poder da LMD exige-se que tenha tanto poder expressivo como o SQL relativamente a colecções, e que através da LMD se possa ter acesso a toda a informação.

- *C19*: Linguagem de interrogação completa.* a LMD deve ter pelo menos o poder do cálculo relacional com colecções de resultados, e deve permitir o acesso a todas as estruturas de dados, incluindo colecções.

A noção anterior poderá ser mais enriquecida assim que se disponha de um modelo melhor para avaliar o poder expressivo de uma linguagem de interrogação.

Poder da Linguagem de Programação

Em C17 introduziu-se a noção de que a linguagem de interrogação deverá ser um subconjunto da linguagem de programação; agora, especificaremos as capacidades da componente da linguagem de programação. De salientar que a linguagem de programação poderá ser muito bem uma linguagem de interrogação de base de dados enriquecida com instruções de controlo, chamadas a procedimentos de uma linguagem de programação embebidos numa linguagem de interrogação, ou uma linguagem unificada (parte de programação conjuntamente com uma parte de base de dados).

- *C20: Linguagem de programação completa.* Um SGBDO deve fronecer capacidade de programação completa em termos computacionais e de recursos.

Embora seja desejável a utilização da linguagem de interrogação sempre que possível para exprimir as operações da aplicação, por forma a se obter uma independência física dos dados maior, a linguagem de interrogação declarativa não é suficiente para exprimir todas as operações, pelo que se terá que recorrer à linguagem de programação.

A linguagem de programação ser completa em termos computacionais significa que deve ser capaz de exprimir qualquer operação que possa ser exprimida na linguagem de programação da aplicação, enquanto que completa em termos de recursos significa que a se deverá ter acesso a todo o ambiente que um programa da aplicação, como por exemplo variáveis globais.

Na abordagem de linguagem de programação para base de dados, temos uma única linguagem com capacidades de interrogação e de programação, pelo que esta abordagem satisfaz os requisitos anteriores.

Nos sistemas de bases de dados estendidos, a adição de instruções de controlo à linguagem de interrogação, ou a possibilidade de invocação a partir da linguagem de interrogação de procedimentos da linguagem de programação permite que o requisito de linguagem de programação completa seja satisfeito, embora o outro não.

Integração da Linguagem de Interrogação

A solução ideal para integração das capacidades de linguagem de programação e de linguagem de interrogação é simplesmente a de integrar interrogações e outra funcionalidade de base de dados na linguagem de programação da aplicação disponível.

- *C21*: Integração da Linguagem de Programação.* Um SGBDO deverá ser integrado no ambiente de uma linguagem de programação existente, para permitir a utilização de informação persistente num ambiente já familiar aos utilizadores.

Esta noção refere-se ao tão falado problema de impedância semântica entre linguagem de programação e os SGBDs. Num sistema bem integrado, o programador apenas tem uma linguagem para aprender, e não necessita dividir-se na execução de código em dois ambientes diferentes.

Pela simples razão de ter sido dada pouca atenção à existência de impedância semântica em aplicações comerciais durante muitos anos, esta característica poderá ser menosprezada. No entanto, a integração com uma linguagem de programação de uso geral poderá constituir um aspecto bastante forte em termos comerciais.

Vamos analisar como é que esta integração é conseguida nas diferentes arquitecturas de bases de dados. Em todos os casos, é de salientar que será necessário invocar interrogações a partir da linguagem de programação e invocar procedimentos a partir da linguagem de interrogação. Poderá até ser importante invocar recursivamente código da linguagem de interrogação e da linguagem de programação.

Arquitecturas de Bases de Dados e Integração

A abordagem de linguagem de programação de bases de dados foi especificamente pensada para o problema de integração da linguagem de programação, constituindo de facto, uma das maiores virtudes da abordagem. Nesta abordagem, todo o ambiente de desenvolvimento familiar ao programador passará a fazer parte do ambiente de desenvolvimento da linguagem de programação de base de dados (compilador, detector de erros, bibliotecas, e outras ferramentas desenvolvidas para a linguagem de programação).

Porque nesta abordagem o modelo de dados do SGBDO tem que ser cuidadosamente concebido como uma extensão ao modelo de dados da linguagem de programação, é difícil conseguir uma integração plena com mais do que uma linguagem de programação. A dificuldade de integração com múltiplas linguagens de programação constitui de facto uma das maiores fraquezas desta abordagem. Porque a experiência até à data mostra que o que se tem conseguido é ia ntegração com zero linguagens de programação, podemos ficar satisfeitos com a integração com pelo menos uma linguagem de programação.

Na arquitectura de sistemas de bases de dados estendidos, existem duas maneiras de fornecer capacidade de programação: estender a linguagem de interrogação com construções de controlo por forma a torná-la computacionalmente completa, ou possibilitar a invocação da linguagem de programação a partir de interrogações. Em ambos os casos, não nos encontramos em presença de uma perfeita integração dos dois sistemas de tipos, pelo que o programador deverá explicitamente copiar informação entre as duas representações. Este é o maior defeito desta arquitectura.

As diferenças entre as arquitecturas de linguagens de programação de bases de dados e sistemas de bases de dados estendidos é subtil. A distinção entre as duas arquitecturas deve ser avaliada pelos programadores que confirmarão se a linguagem de programação de base de dados resolve ou não o problema de impedância semântica.

Independência Lógica dos Dados

Independência lógica dos dados é a capacidade de permitir a existência de múltiplas e distintas vistas externas da vista conceptual da base de dados. O nível conceptual é nor-

malmente definido pelo administrador da base de dados, e os níveis externos são definidos pelos programadores da aplicação.

A independência lógica dos dados é muito importante, porque permite que a estrutura conceptual da base de dados possa ser alterada sem se ter que alterar o código dos programas já escritos. Permite ainda que diferentes aplicações vejam a mesma informação de formas variadas.

Um SGBDO deverá possuir duas características relacionadas com a independência lógica dos dados: encapsulamento e mapeamento das propriedades dos objectos.

- *C22: Encapsulamento de propriedades.* Um SGBDO deve permitir que as propriedades (atributos, relações e procedimentos) visíveis externamente possam ser distinguidos das privadas.

Esta característica vai permitir a quem concebe o esquema distinguir as propriedades comuns privadas sobre as quais as propriedades externas podem ser definidas, limitando assim o número de propriedades eventualmente afectadas por alterações na representação.

Esta característica dever-se-á aplicar a todo o tipo de acesso aos dados, seja ele feito pela linguagem de interrogação ou pela linguagem de programação. No entanto, e atendendo a que o principal objectivo é evitar a reescrita de programas, podemos dispensar a linguagem de interrogação utilizada para acesso directo pelos utilizadores finais de satisfazer este requisito.

Muitas das implementações das vistas relacionais não satisfazem este requisito. No mínimo será necessário distinguir as vistas que definem o nível externo das do nível conceptual.

A noção de encapsulamento proveniente das linguagens de programação orientada por objectos satisfaz obviamente C22. As partes privadas dos objectos representam o nível conceptual, e as partes públicas representam o nível externo. De salientar, que será útil poder classificar de públicos ou privados tanto procedimentos como dados.

Um SGBDO deverá permitir a definição de pelo menos dois níveis de acesso (público e privado). Um SGBDO que permita a definição de mais do que os dois níveis poderá ser útil em várias situações. Por exemplo, poderá ser útil em algumas aplicações a definição de três níveis lógicos: um nível partilhado por todas as aplicações, outro nível apenas visto por grupos de aplicações, e um outro apenas para aplicações especificas.

O segundo requisito para a independência lógica dos dados, não é satisfeito pela maioria dos SGBDOs existentes:

- *C23: Mapeamento de propriedades.* Um SGBDO deverá permitir que o utilizador defina procedimentalmente propriedades virtuais dos objectos da base de dados. As propriedades virtuais são propriedades que escondem outras propriedades. Deverá ser transparente para o utilizador se uma propriedade se encontra definida virtualmente ou directamente.

Existem pelo menos três razões que levam a que os SGBDOs existentes falhem na satisfação deste requisito:

- Se existem duas linguagens diferentes num SGBDO, uma linguagem de programação e uma linguagem de interrogação, C23 deverá ser satisfeita em ambos os contextos. Isto é, os mapeamentos definidos em qualquer uma das linguagens deverá ser visível na outra, o que implica que C22 deverá também ser satisfeito em ambos os contextos.

- Muitos sistemas orientados por objectos não permitem atributos virtuais porque possuem diferenças sintácticas para o acesso a atributos e a métodos. Nestes sistemas apenas poderá ser utilizado o mapeamento de procedimentos, o encapsulamento, e a linguagem de interrogação apenas poderá ser utilizada sobre os atributos dos objectos através métodos desse objecto. O encapsulamento é bem mais útil quando os atributos podem ser tratados da mesma forma que os procedimentos e vice-versa, de forma que o utilizador possa utilizar "atributos" sejam os seus valores obtido por procedimentos ou por armazenamento.

- O mapeamento de propriedades pode não ser invertível; isto é, o SGBDO poderá não ser capaz de fazer actualizações das propriedades. As vistas relacionais nem sempre são transparentes ao utilizador por esta simples razão. Nas linguagens de programação de bases de dados será necessário especificar separadamente operações para escrita e leitura dos valores relativos às propriedades para tornar o mapeamento transparente.

Se não fôr possível redefinir transparentemente uma propriedade externamente visível, então as alterações feitas sobre essa propriedade obrigarão a que os programas tenha de ser reescrito. Atendendo a que este requisito não pode ser completamente satisfeito, implica que existe menos independência lógica no sistema, o que já é melhor que nada.

Normas

As normas na área das bases de dados são importantes principalmente porque facilitam a escrita de aplicações que podem ser convertidas para diferentes SGBDOs. A conversão pode ser necessária por razões várias: desempenho, fiabilidade, disponibilidade, falência da empresa fornecedora.

- *C24*: Portabilidade.* Um SGBDO deverá possibilitar a escrita de programas que possam correr também noutros SGBDOs.

Este requisito implica a existência de normas tanto para linguagens de interrogação como para linguagens de programação. Em particular numa linguagem de programação de base de dados, onde a linguagem de programação foi estendida com capacidades de bases de dados, a linguagem toda, ou apenas parte da biblioteca de suporte, deve ser normalizada, por forma a permitir a portabilidade para outro sistema. Mesmo nos sistemas relacionais, onde existe uma linguagem de interrogação norma (SQL), a falta de um norma de como a integrar numa linguagem de programação torna-se inútil.

Os SGBDOs mesmo com funcionalidade superior à dos sistemas relacionais não deverão ignorar o seu norma em termos de linguagem de interrogação, o SQL. SQL é importante por uma série de razões, tais como familiaridade dos utilizadores e interfaces para bases de dados existentes.

- *C25*: Compatibilidade com SQL.* Um SGBDO deverá fornecer compatibilidade com SQL, ou pelo menos fornecer ferramentas que possibilitem a passagem de e para sistemas de bases de dados SQL.

A história tem demonstrado que a conversão para novas tecnologias é sempre muito lenta a menos que exista alguma forma de compatibilidade com tecnologia existente.

Não é prático mover toda a informação de uma empresa para um SGBDO, mesmo que o SGBDO possua capacidades de processamento de transacções, ajuda à decisão, e as mesmas potencialidades de desenvolvimento do SGBD existente. Só há pouco tempo é que a indústria começou a converter-se para os sistemas relacionais !

Infelizmente, algumas das características dos SGBDOs não poderão ser mapeados facilmente para SQL, tais como listas e procedimentos. O acesso a um SGBDR a partir de um SGBDO não será transparente, atendendo a que tem uma funcionalidade menor, sendo o inverso também verdadeiro, uma vez que toda a informação existente no SGBDO não poderá nunca ser passada através de SQL. Este requisito não poderá portanto ser completamente satisfeito, mas apenas parcialmente satisfeito.

3.3.5 Arquitectura de Bases de Dados

A arquitectura e implementação de um SGBDO são quase independentes do modelo de dados, mas são importantes para este trabalho devido a algumas das aplicações anteriormente enumeradas terem fortes requisitos em termos de desempenho e necessitarem de algumas capacidades tais como acesso remoto à informação.

Custos de Acesso

É importante que as linguagens de interrogação e programação fornecidas pelo SGBDO tenham custos mínimos para as operações requeridas pelas aplicações.

- *C26: Custo de acesso mínimo.* Um SGBDO deve minimizar o custo se execução de operações simples sobre os dados, tais como obter um simples objecto.

Verificação estática de tipos, arquitectura mono processo, integração de linguagem de programação, mapeamento eficiente de objectos para os dados, conversão de referências para apontadores frequentemente utilizados, e outras técnicas poderão ser utilizadas para satisfazer C26.

É também desejável a minimização do tempo de execução de interrogações complexas, mas as diferenças de desempenho nesta área não são geralmente tão dramáticas.

Aproveitamento da Memória Principal

O próximo factor mais importante em termos de desempenho é a eliminação de acessos ao disco. As abordagens mais importantes para atingir este objectivo são *caching* e *clustering*:

- *C27: Utilização de memória principal.* Um SGBDO deve maximizar a probabilidade dos dados serem encontrados em memória principal quando acedidos. No mínimo, deverá possuir um mecanismo de *cache* para a informação na memória virtual da aplicação, e a capacidade de agrupar a informação obtida do disco, em páginas ou segmentos .

Este requisito é interdependente com outros requisitos; por exemplo, para que um mecanismo de *cache* possa ser utilizado, o mecanismo de controlo de concorrência deve permitir transacções de longa duração.

Distribuição

Em termos de bases de dados distribuídas, colocam-se dois requisitos, o primeiro dos quais é acesso remoto.

- *C28: Operações remotas.* Deverá ser possível com um SGBDO aceder a informação armazenada noutra máquina. Deve também ser possível executar operações remotamente, se esta capacidade não fôr convenientemente suportada através do sistema operativo.

O segundo requisito que é importante em termos de distribuição é a capacidade de executar interrogações e programas sobre informação existente em mais do que uma base de dados independente.

- *Distribuição de Bases de Dados.* Um SGBDO deverá possibilitar a execução de interrogações e programas que acedem a mais do que uma base de dados. Objectos de uma base de dados deverão ser capazes de referenciar software de outra base de dados.

3.3.6 Outras Características

Nesta secção final, os requisitos apresentados não pertencem às categorias de modelização de dados e de arquitectura de base de dados até agora considerados.

Concorrência

A implementação de transacções já foi apresentado como um requisito a satisfazer pelos SGBDOs. No entanto, em relação a mecanismos mais sofisticados de controlo de concorrência não há uma ideia precisa do que se pretende.

Uns fornecem a possibilidade de utilização de transacções de longa duração, outros propõe mecanismos de concorrência baseados na semântica dos objectos.

- *C30: Locking.* Um SGBDO deverá providenciar um mecanismo de *locks*, com a granularidade de um objecto ou página, que permita que a informação seja reservada durante longos períodos de tempo.

Não se especifica se o mecanismo deverá ser utilizado directamente pelo utilizador ou não, ou em conjunto com um mecanismo de transacções ou concorrência disponibilizado pelo sistema.

A maior parte das aplicações objectivo estudadas têm requisitos em termos de versões e gestão de configurações. Mais uma vez, não existe opinião definida sobre como um mecanismo de versões deve funcionar por forma a satisfazer todas estas aplicações.

- *C31*: Versões.* Um SGBDO deverá disponibilizar primitivas para gestão de múl tiplas versões de informação, de gestão de configurações.

Sistemas de Regras

Embora capacidades de gestão de conhecimento não sejam requeridas por muitas apli cações, outras têm necessidades de regras simples activadas por gatilhos. Alguns autores são da opinião que estas regras podem ser simplesmente incorporadas nos métodos associados ao tipo do objecto, embora nesta abordagem seja mais difícil garantir que determinadas restrições continuam a ser satisfeitas à medida que os sistemas vão evoluindo.

- *C32*: Regras.* Um SGBDO deverá disponibilizar um mecanismo de regras, para executar acções sempre que determinados predicados assumem valores verdadeiros. A acção associada com a regra tanto pode ser um procedimento (gatilho) como uma condição de erro (restrição).

Ferramentas de Alto Nível

O último, mas não menos importante requisito para um SGBDO relaciona-se com as ferramentas colocadas à disposição dos programadores e utilizadores finais. Estas encontram-se entre as mais importantes características de um qualquer sistema de base de dados, com vista ao aumento de produtividade.

- *C33*: Ferramentas.* Um SGBDO deverá fornecer ferramentas de alto nível que permitam acesso ao esquema de dados e aos dados, e que possa auxiliar o programador no desenvolvimento das aplicações.

O desenvolvimento destas ferramentas é um esforço considerável num produto de base de dados. O trabalho existente em termos de ferramentas no âmbito dos sistemas relacionais não é relevante, uma vez que as ferramentas que agora se pretendem deverão ser orientadas para novas interfaces gráficas e modelos de dados mais sofisticados.

3.3.7 Características Essenciais para o SGBDO da MOOT

As características que se consideram essenciais para o SGBDO a desenvolver futuramente para a ferramenta MOOT são praticamente todas as acima enumeradas com a excepção das capacidades de: tipos dinâmicos, evolução do esquema, mapeamento de propriedades, regras e ferramentas. Estas características sãnão são consideradas essenciais, atendendo a que para o efeito a que se destina o SGBDO, apesar de úteis, não são obrigatórias.

4

Uma Interface para um SGBDO em NeXTSTEP

Nos capítulos anteriores foram abordados diversos temas relacionados com os novos sistemas de bases de dados: as novas aplicações de bases de dados, os sistemas de bases de dados tradicionais, os novos modelos de dados, as novas linguagens de programação, e as características gerais dos novos SGBDs. Muito globalmente, podemos afirmar que os SGBDOs têm como objectivo estender a funcionalidade dos SGBDs tradicionais, e diminuir a impedância semântica actualmente existente entre linguagens de programação e linguagens de interrogação de bases de dados. As abordagens seguidas para atingir estes objectivos são várias. Ao longo deste trabalho, tem sido a abordagem relativa às linguagens de programação para bases de dados, aquela que maior atenção tem merecido.

Tendo em vista o futuro desenvolvimento em ambiente NeXT de um SGBDO para os fins anteriormente apresentados (secção 1.6), considerou-se importante começar por definir uma interface entre as suas duas principais componentes em termos de funcionalidade: o sistema de bases de dados e a linguagem de programação.

A definição desta interface é importante porque irá permitir que os futuros desenvolvimentos do sistema, aos seus diferentes níveis—interfaces com o utilizador, bases de dados, ferramentas CASE, etc., possam ocorrer da forma mais independente possível, e assim potenciar uma evolução fácil e contínua do sistema global. Pela complexidade inerente, um sistema desta natureza conseguirá ser mais rapidamente desenvolvido se evoluir por formas intermédias estáveis a partir de um sistema simples.

No sentido de definição de uma norma para o estabelecimento de uma interface entre as LPs e os SGBDs, foi recentemente publicada uma proposta do *ODMG—Object Database Management Group* neste sentido, que no caso de ser aceite, permitirá uma maior adesão e divulgação destes novos SGBDOs. Pela sua importância, tanto para a área em que se insere este trabalho, como para o próprio desenvolvimento do trabalho, o conteúdo desta proposta será sucintamente apresentada.

Tendo como base a proposta de norma ODMG-93, será delineada na segunda parte deste capítulo uma interface em Objective-C para um Sistema de Gestão de Bases de Dados de Objectos a desenvolver.

4.1 Proposta de Norma ODMG-93

Contrariamente ao que se verifica para os Sistemas de Bases de Dados Relacionais, para os quais existe um modelo de dados único e universalmente aceite—o Modelo Relacional, ainda não foi consensualmente definido o modelo base dos Sistemas de Bases de Dados de Objectos—o Modelo de Objecto.

A inexistência de um único Modelo Objecto tem dificultado o aparecimento de normas nesta área, e tem-se reflectido de forma negativa na divulgação deste novo tipo de sistemas de bases de dados.

Se relativamente aos SGBDRs a normalização foi extremamente importante para o seu sucesso, mais importante se revela ainda para os SGBDOs, atendendo a que o âmbito de aplicabilidade destes novos sistemas é muito mais alargado do que o dos SGBDRs (secção 1.5.1). Os SGBDOs conseguirão penetrar no mercado se oferecerem garantia de portabilidade para as suas aplicações [Atw93].

Com o objectivo de acelerar o processo de aparecimento de normas para os SGBDOs, foi publicado em finais de 1993 pelo *Object Database Management Group* (ODMG), uma proposta de norma, o ODMG-93 [CAD+94].

4.1.1 Objectivos do ODMG-93

Esta proposta tem como objectivo principal a definição de um conjunto de normas que garantam a portabilidade das aplicações escritas entre vários SGBDOs. Com este objectivo em vista, são propostas definições para o esquema de dados, linguagens de definição, manipulação e de interrogação, e a sua interface com algumas linguagens de programação, concretamente C++ e Smalltalk.

A definição da interface, ao nível de esquema de dados e dos diversos tipos de linguagens envolvidas, deixa em aberto para futura inovação muitas outras áreas, permitindo assim que os produtos se distingam a vários níveis, tais como: performance, linguagens suportadas, ambientes de desenvolvimento, ferramentas para construção de aplicações, redes, plataformas, bibliotecas de tipos, ferramentas para construção de interfaces gráficas, ferramentas para apoio ao projecto, etc.

O ODMG-93 tem como pretensão conseguir a integração dos sistemas de bases de dados com as linguagens de programação, de forma muito superior ao hoje possível com os sistemas relacionais.

4.1.2 Componentes do ODMG-93

Tal como referido anteriormente (secção 1.5.3) as componentes principais da proposta de norma são a definição dum modelo objecto e do seu mapeamento para algumas linguagens de programação. Estes mapeamentos têm como base as linguagens de definição, interrogação e manipulação de dados de objectos, respectivamente as *Object Definition Language* (ODL), a *Object Query Language* (OQL) e a *Object Manipulation Language* (OML), também definidas pelo ODMG-93.

4.1.3 Modelo Objecto ODMG

O modelo objecto proposto na ODMG-93 caracteriza-se pelo seguinte:

- O *objecto* constitui a primitiva base do modelo.
- Os objectos pertencem a um *tipo*, onde se encontram definidas as propriedades e comportamento comum a todos objectos desse tipo.
- O estado dos objectos é definido pelo conjunto dos valores das suas *propriedades*. As propriedades podem ser *atributos* do próprio objecto, ou *relações* entre o objecto e outros objectos.
- O comportamento dos objectos é definido por um conjunto de *operações*.
- Os tipos de objectos são organizados em hierarquias *tipo/supertipo*. Subtipos herdam o estado e comportamento definido nos seus supertipos.

Os tipos de objectos têm uma *interface* e uma ou mais *implementações*. A interface define a visão externa do tipo, isto é, as suas propriedades e as operações possíveis sobre elas. Cada uma das implementação define as estruturas de dados que permitem representar o seu estado, e os métodos através dos quais é possível alterar esse estado. À combinação da especificação da interface de um tipo com uma das implementações definidas para o tipo é dado o nome de *classe*.

Como exemplo, apresenta-se em baixo a definição da interface de um tipo de objectos para a base de dados do exemplo anteriormente apresentado na secção 3.1.3.

```
interface Documento {
      type properties:
         supertype:  Atomic_Object;
         extent:  documentos; key:  (título);
      instance properties:
         título:  String; // deve ser único
         revisão:  Date;
         autores:  Set<Pessoas> inverse Pessoa::autora_de;
         capítulos:  Set<Capítulo> inverse Capítulo::pertence_a;
      instance operations:
         gravar() raises(erro_de_gravação);
         adicionar_capítulo(capítulo:  Capítulo); };
```

No modelo objecto ODMG, analogamente a algumas LPOOs, os tipos são eles próprios objectos, pelo que também podem ter propriedades. É na definição da interface do tipo que são atribuídos valores a estas propriedades. No exemplo acima são definidos valores para as propriedades relativas às superclasses, conjuntos de instâncias e propriedades chave do tipo.

- *Supertypes.* Lista dos tipos dos quais este herda, ou dito de outra forma, lista dos tipos dos quais este é uma especialização (ver secção 2.3.2).
- *Extents.* Ao conjunto de todas as instâncias de um tipo é dado o nome de *extensão.* Através desta propriedade, o programador pode solicitar ao sistema que seja mantida uma relação de todos os objectos criados para este tipo.
- *Keys.* A possibilidade de definição de chaves para determinados objectos é útil para alguns problemas. As chaves podem ser simples ou compostas—consistindo em uma ou mais propriedades, respectivamente.

Para além das propriedades do próprio tipo, podemos ver no exemplo acima, a declaração das propriedades e operações das instâncias do tipo. Nessas declarações podemos distinguir o seguinte:

- *Operações.* Definem o nome da operação, o nome e tipo dos argumentos, o nome e tipo dos resultados, e o nome das *excepções* que a operação pode desencadear em situação de erro.
- *Atributos.* Definem o nome do atributo e o seu tipo. Um atributo apenas pode assumir valores literais: caracteres, inteiros, reais, etc.
- *Relações.* Definem o tipo de outro objecto ou conjunto de objectos envolvidos na relação, e uma função de navegação utilizada para referenciar o objecto ou conjunto de objectos relacionados. As relações são binárias e a sua cardinalidade pode ser um- para-um, um-para-muitos, ou muitos-para-muitos.

O modelo possui vários tipos pré-definidos: conjuntos, listas, vectores, etc.
A hierarquia dos tipos base do modelo objecto é representada na figura 4.1.

Tipos e Instâncias

Um tipo define as características das suas instâncias: o estado—conjunto de propriedades, e o comportamento—conjunto de operações.

Os tipos podem ser organizados numa hierarquia de generalizações, onde os tipos se encontram ligados por relações de herança. Um subtipo herda todas as características dos seus supertipos, podendo ainda definir características adicionais.

O conjunto de todas as instâncias de um tipo é denominado de *extensão* do tipo. A definição desta propriedade dum tipo indica ao SGBDO que deve manter uma colecção de todas as instâncias do tipo, à medida que estas vão sendo criadas e destruídas. Entre as noções de *tipo* e *extensão* existe uma correspondência directa. Se um objecto `x` é instância de um tipo `X`, então `x` pertence ao conjunto de instâncias de `X`, seja `Xs`. De igual modo, se o tipo `X` é um subtipo de `Y` então `Xs`—o conjunto de todas as instâncias de `X`, é um subconjunto de `Ys`— o conjunto de todas as instâncias de `Y`.

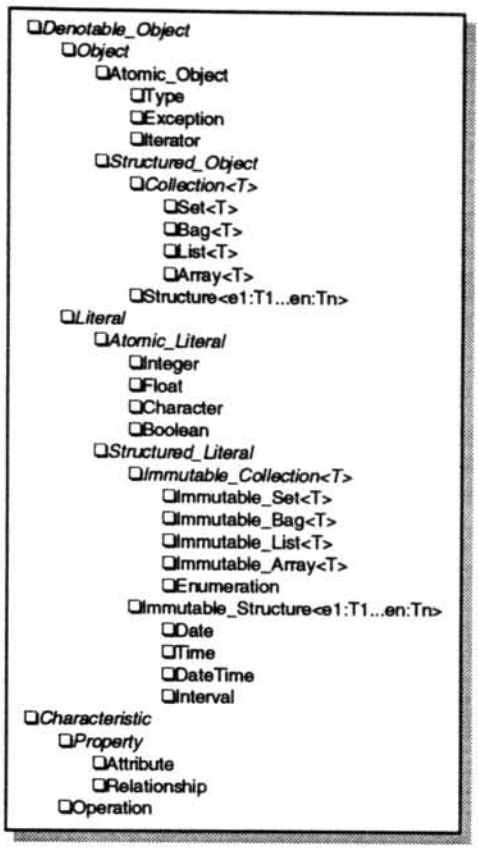


Figura 4.1: Hierarquia de tipos do Modelo Objecto ODMG

Um tipo pode ter uma ou mais implementações. Uma implementação consiste numa representação do estado e num conjunto de métodos que implementam as operações especificadas na interface.

O modelo objecto ODMG é mais rico do que, por exemplo, o modelo de dados de C++, porque permite a existência de mais do que uma implementação para uma mesma interface de um tipo. Do ponto de vista do construtor do SGBDO, isto é útil por facilitar a implementação em diferentes máquinas, arquitecturas, linguagens e ambientes. Do ponto de vista do programador, é útil por permitir ao programador fornecer diferentes implementações para situações com diferentes requisitos em termos de performance e espaço. Sem esta possibilidade, o programador normalmente recorre ao mecanismo de herança, e define diferentes classes, em que as únicas diferenças residem apenas nas implementações.

A escolha da implementação para um determinado tipo tem de ser feita na altura de criação do objecto, o que constitui uma limitação, face à possibilidade de a alterar dinamicamente.

Objectos

Um objecto é algo que possui um estado, um comportamento, e uma identidade, isto é, possui um conjunto de atributos e relações—estado, um conjunto de operações— comportamento, e um identificador (OID) único que o permite distinguir de todos os outros objectos da base de dados a que pertence.

A hierarquia de tipos do modelo objecto ODMG tem duas componentes ortogonais nas quais os vários tipos se decompõem: (1) literais ou objectos e (2) atómicos ou estruturados.

Os valores das instâncias de `Literal` são inalteráveis, e relativamente às instâncias de `Object` são alteráveis tanto os valores dos seus atributos como os das relações em que participa. A identidade destes objectos não é no entanto modificável.

As instâncias de `Literal` têm identidade única, mas não têm OIDs. Um objecto apesar de ter um identificador único, pode ter mais do que um nome associado, desde que este nome no âmbito em que foi declarado referencie univocamente um objecto. Encontram-se definidas no tipo `Object` características que permitem saber em tempo de execução os nomes dados a um objecto e o seu tipo, para além de operações específicas para criar, comparar e destruir objectos.

Em termos de tempo de vida, são definidos no modelo base três âmbitos: procedimento, processo e base de dados.

Estado e Comportamento

O estado de um objecto é representado por atributos e relações.

Os atributos apenas podem assumir valores literais de acordo com o seu tipo. As relações são definidas entre dois tipos de objectos, como por exemplo, entre `Documento` e `Capítulo` como no tipo acima definido.

Para os atributos, encontram-se pré-definidas duas operações: uma para leitura e outra para escrita de valores.

```
set_value(new_value:  Litera)
get_value() -> existing:value:  Literal
```

As relações em si não têm nomes, podendo ser identificadas apenas pelos atributos inversos de cada uma das extremidades da relação. No exemplo abaixo, na interface do tipo `Documento` é declarado que `autores` estabelecem uma relação para um conjunto de objectos do tipo `Pessoa`, e na interface do tipo `Pessoa` é declarado o inverso, isto é, que uma `Pessoa` é autora de um conjunto de objectos do tipo `Documento`.

```
interface Documento { ...
   autores:  Set<Pessoas> inverse Pessoa::autora_de;
};
```

e

```
interface Pessoa { ...
   autora_de:  Set<Documento> inverse Documento::autores;
};
```

A integridade referencial deve ser mantida pelo SGBDO que implemente a norma. As relações podem ser ordenadas numa ou em ambas as extremidades da relação.

Uma instância de uma relação muitos-para-muitos implica a existência de dois conjuntos de relações um-para-muitos, e esta, por sua vez, implica a existência de um conjunto de relações um-para-um.

O modelo tem pré-definidas para todos os tipos de relações uma série de operações para criação, visita, actualização, e destruição [CAD+94].

O comportamento de um determinado tipo de objectos é especificado através de um conjunto de operações para esse tipo. Para cada operação, o programador declarará o nome e tipo de cada um dos seus argumentos e valores retornados, e as excepções potencialmente desencadeadas. As operações são sempre definidas exclusivamente para um único tipo de objectos.

Os nomes das operações têm de ser únicos apenas no âmbito da definição de um tipo, o que permite a definição de operações com nomes iguais desde que em tipos diferentes. Esta possibilidade, conjuntamente com a herança e mecanismos de associação dinâmica, vão permitir o aparecimento de entidades polimórficas.

O modelo possui também mecanismos para tratamento de excepções, em que as excepções são elas próprias objectos do tipo `Exception`.

Objectos Estruturados

O modelo objecto ODMG suporta dois tipos base de objectos estruturados: `Structure` e `Collection`. Os do primeiro tipo caracterizam-se por possuírem um número fixo de objectos ou literais de tipos possivelmente diferentes. Os segundos, por outro lado, contêm um número arbitrário de objectos em que são todos instâncias dum mesmo tipo (ou de um supertipo comum).

Os subtipos de `Collection` definidos no modelo são: `Set`, `Bag`, `List`, e `Array` e são todos tipos parametrizados. Para cada um destes tipos encontram-se definidas uma série de operações para sua criação, manipulação, visita e destruição.

Os objectos do tipo `Structure` podem ser compostos de forma totalmente arbitrária, e indistintamante por objectos ou literais.

São suportados pelo modelo tipos idênticos aos anteriores mas em versões de valores não alteráveis, isto é, uma vez criados, os seus valores não podem ser alterados.

Transacções

O modelo objecto ODMG suporta um modelo de transacções encaixadas, com definição dinâmica de âmbito de visibilidade. O exemplo abaixo ilustra um sequência destas transacções:

```
Transaction::begin() -> p:Transaction
   Transaction::begin() -> x:Transaction
      Transaction::begin() -> y:Transaction
         if minor_error y.abort()
         if major_error y.abort_top_level()
      y.commit()
   x.commit()
p.commit()
```

No exemplo, é apresentada uma transacção principal `p`, que contém duas transacções encaixadas `x` e `y`. Alterações feitas por `t` antes do início de `x` são visíveis tanto a `x` como a `y`. Alterações feitas por `x` são visíveis a ambas as transacções `t` e `y` assim que `x` confirma

a transacção através de `commit()`. De forma análoga, se `t` é interrompida por (`abort()`, então as alterações feitas por `x` e `y` também serão anuladas, independentemente destas já terem feito `commit()` ou não.

Este tipo de transacções permite ao programador uma maneira simples de anular as alterações feitas a informação persistente.

O modelo objecto ODMG utiliza uma abordagem tradicional de *locks* ao nível de objectos, ou a um nível físico superior.

Todas as operações de criação, acesso, modificação, e destruição de objectos persistentes devem ser invocadas no âmbito de uma transacção. As operações definidas no tipo **`Transaction`** são as seguintes:

```
begin([optimistic]) -> t:  Transaction
commit()
abort()
checkpoint()
abort_to_top_level()
```

Na presente norma, os objectos persistentes não são abrangidos pela semântica de transacções, o que significa que o `abort()` de uma transacção não repõe o estado dos objectos transitórios modificados.

O modelo base assume uma sequência de transacções linear dentro de um único processo e para uma única base de dados.

Bases de Dados

Uma base de dados permite o armazenamento de objectos persistentes de um determinado conjunto de tipos. Cada base de dados tem um esquema, que consiste num conjunto de definições de tipos. Uma base de dados lógica pode ser armazenada numa ou mais bases de dados físicas. Cada base de dados é uma instância do tipo **`Database`**. As operações para este tipo são:

```
open()
close()
contains_object?  (oid:Object) -> b:Boolean
lookup_object (oid:  Object) -> b:Boolean
```

Os nomes dos tipos definidos num esquema e os dos respectivos conjuntos de instâncias são globais para a base de dados, pelo que poderão ser acessíveis aos programas depois de abrirem a base de dados. Uma base de dados poderá também conter nomes de objectos, normalmente denominados de objectos-raíz, a partir dos quais, o programador poderá navegar pela base de dados através das relações definidas no esquema.

Futuros Desenvolvimentos

A presente proposta de norma, não constituindo ainda um produto perfeito e acabado, é no entanto uma boa base para iniciar a prototipagem de SGBDOs, e com a experiência daí resultante elaborar versões evoluídas desta proposta ODMG-93.

Nesta proposta, são já indicadas algumas das revisões previstas para apresentação numa futura versão desta proposta, que por razões de especificidade não serão aqui abordadas (ver [CAD+94]).

4.1.4 Linguagem de Definição—ODL

A linguagem de definição de objectos proposta na ODMG-93 (ODL—*Object Definition Language*) é uma linguagem de especificação utilizada para definição das interfaces de tipos de acordo com o modelo objecto ODMG anteriormente apresentado. O principal objectivo da ODL é facilitar a portabilidade de esquemas de bases de dados entre diferentes SGBDOs, e permitir a interoperabilidade de SGBDOs de diferentes fornecedores.

Princípios Orientadores

Os princípios orientadores do desenvolvimento da ODL foram:

- A ODL deverá suportar todas as construções semânticas do modelo objecto ODMG.
- A ODL não deverá ser uma linguagem de programação completa, mas apenas uma linguagem de especificação para interfaces.
- A ODL deverá ser independente da linguagem de programação utilizada.
- A ODL deverá ser extensível, não apenas para futura acrescentar nova funcionalidade, mas também para questões de optimização física.

Os SGBDs sempre ofereceram tanto linguagens para definição de dados (LDDs) como linguagens para manipulação de dados (LMDs) (secção 3.1.4). A ODL descrita nesta secção é uma LDD para tipos de objectos, permitindo a definição das suas características, incluindo propriedades e operações. As implementações destas características não são definidas pela ODL.

A ODMG-93 não propõe uma norma para a linguagem de manipulação de dados de objectos (OML—*Object Manipulation Language*), devendo esta ser definida de acordo com o modelo objecto ODMG e um mapeamento para a linguagem de programação específica. A ODMG-93 propõe mapeamentos para duas linguagens de programação: C++ e Smalltalk. No presente trabalho será definido um mapeamento para a linguagem Objective-C em ambiente NeXTSTEP.

Pretende-se que a ODL defina tipos de objectos que possam ser implementados numa variedade de linguagens de programação, pelo que a ODL não se encontra dependente de nenhuma linguagem de programação específica. Um esquema definido em ODL deverá poder ser implementado por um qualquer SGBDO, independentemente da linguagem de programação que utiliza. A ODL proporciona um isolamento entre as aplicações e a especificidade da linguagem de programação e os SGBDOs.

Uma especificação ODL pode ser na prática implementada numa linguagem de programação orientada por objectos como por exemplo C++.

Especificação

Um tipo é definido através da especificação da sua interface em ODL. As características do próprio tipo são definidas em primeiro lugar, seguindo-se a lista de atributos e relações, e finalmente a lista de operações.

Um exemplo de definição em ODL duma interface para um tipo é apresentado a seguir:

```
interface Texto: Documento {
  extent textos;
```

```
    keys referência;
    ...
    attribute string referência;
    List<Figura> figuras inverse Figura::textos order by número;
    relationship Pessoa revisor inverse Pessoa::textos_revistos;
    alterar_tamanho_página (in altura: integer, in largura: integer);
    obter_no_linhas (out linhas: integer);
};
```

No exemplo acima, as palavras reservadas de ODL encontram-se em **negrito**. As primeiras definições que se encontram na interface dizem respeito às características do próprio tipo: as suas superclasses (`Documento`), o nome da sua extensão, isto é, o conjunto das suas instâncias (`textos`), e as propriedades chave (`referência`). Seguem-se depois as propriedades do tipo: os atributos (`referência`, e as relações (`figuras` e `revisor`). As palavras reservadas `attribute` e `relationship` são opcionais, pelo que podem ser omitidas na declaração.

Finalmente, são declaradas as operações possíveis de invocar para as instâncias do tipo. Na declaração de uma operação é definido o nome da operação, e o nome, tipo e sentido (entrada—`in`, saída—`out`, ou ambas—`inout`) de cada um dos seus argumentos. No exemplo dado, `altura` e `largura` constituem argumentos de entrada para a operação `alterar_tamanho_página`. Para cada operação é ainda possível declarar um ou mais nomes de excepções invocáveis em caso de erro.

Apesar da simplicidade do exemplo dado, os principais aspectos da sintaxe utilizada pela ODL encontra-se nele representada. A sintaxe completa de ODl encontra-se disponível em [CAD+94].

Pelo estudado, pode-se afirmar que ODL é uma linguagem simples, fácil de aprender, e fácil de implementar, factores estes que jogam a seu favor em termos de possibilidade de divulgação em larga escala.

4.1.5 Linguagem de Interrogação—OQL

A linguagem de interrogação de objectos proposta na ODMG-93 (OQL—*Object Query Language*) é uma linguagem simples e completa. A OQL permite a manipulação de objectos complexos sem privilegiar a noção de conjunto ou a cláusula `select-from- where`.

Princípios Orientadores

Os princípios orientadores do desenvolvimento da OQL foram:

- A OQL não é computacionalmente completa. É uma linguagem de interrogação que permite fácil acesso a uma base de dados de objectos.
- A OQL permite um acesso aos objectos de forma declarativa.
- A OQL tem como base o modelo objecto ODMG.
- A OQL tem uma sintaxe abstracta.
- A semântica formal da OQL pode ser facilmente definida.
- A OQL possui construções síntacticas concretas idênticas às de SQL, e outras para integração nas linguagens de programação.

- A OQL possui primitivas para manipulação de conjuntos mas não se limita apenas a este tipo de estruturas de dados. Possui também capacidades de manipulação de listas, vectores, etc.
- Atendendo ao seu caracter declarativo, a OQL pode ser facilmente optimizada.

Exemplos de Interrogações Simples

A OQL pode ser utilizada de duas formas: como linguagem autónoma, ou como uma função da linguagem de programação.

Vamos supôr que, para o exemplo apresentado na secção 3.1.3, temos definidos os tipos `Documento`, `Capítulo`, `Pessoa`, e `Texto`. Os nomes dos atributos destes tipos encontram-se descritos no diagrama de classes OMT da figura 4.2.

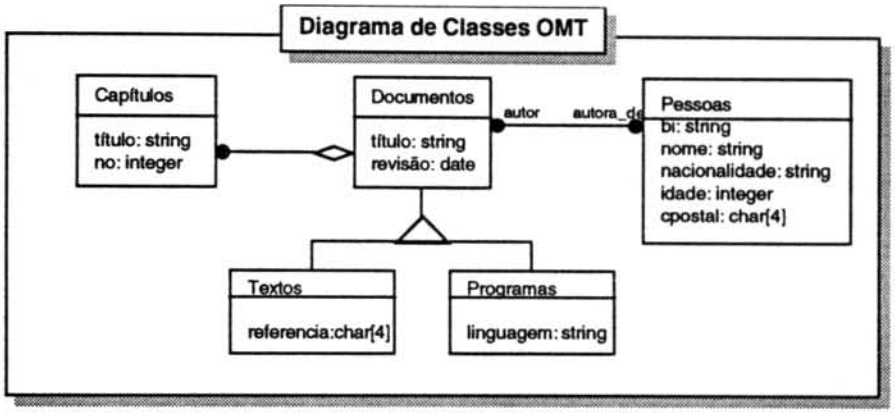


Figura 4.2: Base de dados de documentos.

```
select distinct x.nome
from x in Pessoas
where x.nacionalidade = "Portugal"
```

O resultado desta interrogação será um conjunto de nomes de pessoas portuguesas. Os tipos dos resultados das interrogações serão quase sempre literais, isto é, instâncias de tipos não alteráveis. Neste caso, o resultado será um literal do tipo `set<string>`.

```
select distinct struct(n:  x.nome, m:  x.morada, c:  x.cpostal)
from x in Pessoas
where x.nacionalidade = "Portugal"
```

Esta interrogação construirá para cada pessoa uma estrutura contendo os campos nome, morada, e código postal. Retornará um literal do tipo `set<struct(n: string, m: string; c: char[4])>`.

Contrariamente ao que acontece em SQL, em OQL é possível a utilização de resultados de subperguntas, tanto depois de `select`, como depois de `from`.

```
select distinct struct(t:  x.título,
   auts:  (select y.nome from y in x.autores)
from x in Documentos
```

Esta interrogação construirá para cada documento uma estrutura com o título do livro e o conjunto de nomes dos seus autores. Retornará um literal do tipo **set<struct(t: string, auts: bag<string>)>**.

```
select x.nome
from x in (select y.revisor from y in Textos where revisão > "1-1-94")
where x.nacionalidade="Portuguesa"
```

Esta interrogação permitirá obter o nome dos revisores portugueses dos textos que foram revistos depois de 1 de Janeiro de 1994.

Em OQL, é possível formular interrogações sem ter de recorrer à clausula **select-from-where**, bastando para tal indicar nomes de conjuntos de instâncias ou de objectos. Por exemplo:

```
Pessoas
```

Obtém o conjunto de todos os objectos de pessoas.

Os resultados das interrogações em OQL tanto podem ser objectos seleccionados da base de dados como objectos criados pelo interpretador. A criação de um objecto do tipo **Pessoa** pode ser obtida através da invocação do nome do tipo **Pessoa**.

```
Pessoa(nome:  "Ademar", nacionalidade; "Portuguesa")
```

Entre parentesis opcionalmente poderão ser dados valores de inicialização de algumas das propriedades do objecto.

Poderá ser obtido para o primeiro exemplo apresentado nesta secção um objecto do tipo **set<string>** através de:

```
set(select distinct x.nome
     from x in Pessoas
     where x.nacionalidade = "Portugal")
```

Descrição da Linguagem

Uma interrogação consiste num conjunto possivelmente nulo de expressões de definição de interrogações seguido de uma expressão. Por exemplo:

```
define portugas as select x from x in Pessoas
   where x.nacionalidade="Portuguesa";
select distinct x.nome from x in portugas
```

Esta interrogação define o conjunto **portugas** como um conjunto de todos os objectos do tipo **Pessoa** cuja nacionalidade é portuguesa.

As expressões de OQL podem ser de vários tipos:

- *Elementares:* literais (booleanos, inteiros, reais, caracteres, strings), nomes de objectos raíz, e nomes de interrogações.
- *Aritméticas:* $+$, $-$, $*$, $/$, `mod`, e `abs()`.
- *Comparações:* $=$, $!=$, $<$, $\leq$, $>$, e $\geq$.
- *Booleanas:* negação, conjunção, e disjunção.
- *Construtoras:* de objectos do tipo estrutura, conjunto, lista, vector, ou de qualquer outro tipo.
- *Acesso:* a características dos objectos através da notação '.' ou '$->$', e acesso indexado aos elementos de listas ou vectores através de `first()`, `last()`, e de `[]`.
- *Conversão:* de listas para conjuntos (`listtoset()`), de conjuntos com um elemento para um elemento (`element()`), de colecções de colecções de elementos para colecções de elementos (`flatten()`), e conversão explícita para um determinado tipo.
- *Operações sobre Conjuntos:* intersecção (`intersect`), união (`union`), e diferença (`except`).
- *Operações sobre Colecções:* quantificador universal (`for all ... in ... : ...`), quantificador existencial (`exists ... in ... : ...`), verificação de pertença (`elem in` colecção), cláusula `select-from-where`, ordenação (`sort ... by`), `count()`, `sum()`, `min()`, `max()`, `avg()`, e `group ... in ... by ... with ...`.

Destes tipos todos de expressões merecem especial destaque as expressões para: acesso, conversão, operações sobre conjuntos, e operações sobre colecções.

Exemplos de Interrogações Menos Simples

Para ilustrar a utilização dos tipos de expressões acima destacados serão apresentados alguns exemplos mais elaborados.

```
first(sort x in
      (select y
      from y in Pessoas
      where y.nacionalidade = "Portugal")
   by x.idade).nome
```

A interrogação acima retorna o nome da pessoa portuguesa mais nova. Neste exemplo foi utilizada a função `first()`. Esta função permite seleccionar o primeiro elemento de uma lista ou vector.

```
select w.nome
from w in
   (sort x in
      (select y
      from y in Pessoas
      where y.nacionalidade = "Portugal")
   by x.idade)[0:2]
```

Este exemplo é idêntico ao anterior, mas retorna um literal do tipo `bag<string>` com os nomes das três pessoas portuguesas mais novas. Aqui foram seleccionados os elementos entre os índices 0 e 2 da lista criada por `sort ... by`.

A conversão de expressões é possível através da utilização de algumas funções pré-definidas, nomeadamente de listas para conjuntos.

```
element(select x from x in Pessoas where x.bi="7543047")
```

A interrogação acima retorna um objecto do tipo `Pessoa`, o que é possível porque `element()` permite converter uma colecção com apenas um elemento no elemento em si.

A funcionalidade oferecida pela OQL para objectos do tipo colecção (listas, conjuntos, vectores, etc.) é uma extensão da funcionalidade da sua homóloga dos sistemas relacionais—o SQL.

A OQL possui as operações de intersecção, união e diferença de conjuntos. Adicionalmente, possui primitivas para operar sobre objectos do tipo colecção, tais como, quantificação universal e existencial, teste de pertença, e cláusula `group ... by ... with ...` para a criação de partições de colecções com base em critérios definidos pelo utilizador. Estas características fazem com que a OQL seja linguagem fechada e completa (ver secção 3.1.4).

```
select x.título
from x in Documentos
where (for all y in x.autores:  y.nacionalidade="Portuguesa")
```

Esta interrogação permite, com a simplicidade demonstrada, a obtenção de todos os documentos cujos autores são na sua totalidade portugueses. Só por curiosidade, pode- se adiantar que a interrogação equivalente em SQL não só é mais complexa, como muito mais díficil de interpretar. Tal diferença deve-se sobretudo à não existência de quantificação universal em SQL, e à consequente necessidade de explicitar as junções entre os campos relacionados.

A cláusula `group ... by ... with` em OQL tem uma semântica bem definida e produz resultados de acordo com o modelo objecto ODMG: um conjunto de estruturas.

```
group x in Pessoas by
   (jovens:  x.idade ≥ 12 and x.idade < 30,
   experientes:  x.idade ≥ 30 and x.idade < 55,
   vividos:  x.idade ≥ 50)
```

Esta interrogação dá origem a um conjunto de três elementos, cada um dos quais tem uma propriedade denominada `partition` que contém um conjunto de pessoas que pertencem a cada uma das categorias definidas. O tipo do resultado é:

```
set<struct(jovens:boolean,experientes:boolean,vividos:boolean,
        partition:set<Pessoa>)>
```

Cada estrutura do conjunto resultante, possui as propriedades definidas na cláusula **by** e as definidas na cláusula **with**.

```
group x in Pessoas by
   (jovens:  x.idade ≥ 12 and x.idade < 30,
   experientes:  x.idade ≥ 30 and x.idade < 55,
   vividos:  x.idade ≥ 50)
with (número:  count(partition),
   menor_idade:  min(select x.idade from x in partition),
   maior_idade:  max(select x.idade from x in partition))
```

Esta interrogação permite agrupar as pessoas em três categorias com base na sua idade, e para cada uma delas permite obter ainda o número de pessoas, e as menor e maior idades encontradas em cada categoria. O tipo do resultado é:

```
set<struct(jovens:  boolean, experientes:  boolean, vividos:  boolean,
   número:  integer, menor_idade:  integer, maior_idade:  integer)>
```

Apesar de não ter sido apresentada de forma exaustiva, toda a sintaxe da OQL, foram no entanto apresentados os seus aspectos mais relevantes. A definição da sintaxe completa da OQL encontra-se disponível em [CAD+94].

Pelo apresentado, pode-se já concluir que a OQL é uma linguagem mais poderosa do que o SQL, de fácil aprendizagem, e de sintaxe simples, embora não tão simples como a do SQL.

4.1.6 Mapeamento para as Linguagens de Programação

Apresentadas que estão as linguagens de definição e de interrogação propostas na ODMG-93, poder-se-ia dizer que faltaria ainda apresentar a linguagem de manipulação. Relativamente à linguagem de manipulação (OML—*Object Manipulation Language*) não é feita nenhuma proposta, pelo que ficará a cargo de cada implementação a sua definição no contexto da sintaxe e semântica da linguagem de programação a estender.

A OML será utilizada para criar, apagar, identificar, referenciar, alterar valores das propriedades, e invocar operações para os objectos persistentes. A sintaxe utilizada para tal deverá, na medida do possível, não ser muito diferente, da utilizada na linguagem de programação a estender, para objectos com outros graus de persistência. As expressões deverão poder livremente referenciar tanto objectos persistentes como objectos transitórios.

A proposta de norma define o mapeamento do modelo objecto e linguagens ODMG para as linguagens de programação (LPs) C++ e Smalltalk.

Mapeamento do Modelo Objecto ODMG

Um mapeamento para uma LP deverá começar por mapear o modelo objecto ODMG, concretamente as noções de: objectos e literais, relações, conjuntos de instâncias, chaves, nomes, interface e implementações de um tipo, e hierarquia de tipos pré- definidos.

ODL, OML e OQL

Depois do mapeamento do modelo objecto, seguir-se-á a ODL para descrever o esquema da base de dados. O esquema da base de dados de um SGBDO é um conjunto de classes, em que cada classe tem atributos, relações e operações. Todos estes aspectos têm de ser mapeados.

A OML deverá por fim ser definida para permitir criar, destruir, modificar e referenciar objectos.

Relativamente à OQL, esta será implementada por exemplo através de uma função autónoma que terá como argumentos a interrogação formulada e uma referência para retorno do resultado.

Hierarquia de Tipos Base

Após a implementação do modelo objecto, da ODL, OML, e OQL, o passo seguinte consistirá na implementação das classes para listas, vectores, conjuntos, etc, suportadas pelo modelo objecto.

Operações sobre a Base de Dados e Transacções

A definição de classes para as transacções, e para implementação das operações possíveis sobre a base de dados, definidas no modelo objecto, finalizará o mapeamento para a linguagem de programação escolhida.

Na figura 4.3 são ilustradas os vários passos para a criação de uma aplicação para um SGBDO que tenha como base a ODMG-93.

4.2 Interface em Objective-C para um SGBDO

Nesta secção, será delineada uma interface em Objective-C para um Sistema de Gestão de Bases de Dados de Objectos a desenvolver, tendo como base a proposta de norma ODMG-93.

4.2.1 Abordagens Possíveis

As abordagens consideradas para a definição de tal interface foram várias. Abaixo apresentam-se as abordagens delineadas mais recentemente e que mereceram maior consideração. A figura 4.4 pretende ilustrar essas abordagens.

1. Extensão de Objective-C com uma biblioteca de classes que permitisse o fácil acesso e manipulação de dados armazenados em tabelas duma base de dados relacional [AS92, RBP88, RBL+91].

2. Extensão de Objective-C com novas construções sintácticas e com uma biblioteca de classes para acesso e manipulação de objectos armazenados numa base de dados relacional criada a partir dum diagrama de classes OMT. A linguagem estendida deveria incorporar uma linguagem de interrogação declarativa que, com base no diagrama de classes OMT, fizesse a conversão dos dados armazenados na base de dados relacional para objectos Objective-C. Tal conversão deveria ser transparente para o programador [RBL+91].

3. Extensão de Objective-C tendo como base a proposta de norma ODMG-93, isto é, delinear o mapeamento para Objective-C do modelo objecto e linguagens ODL, OQL e OML constantes da proposta ODMG-93 [CAD+94].

As três abordagens acima, embora todas possuam ohjectivos idênticos, apresentam diferenças significativas, nomeadamente em termos de facilidade de implementação, impedância semântica da interface resultante, e possível compatibilidade com outros desenvolvimentos na área dos SGBDOs.

Assim, a primeira abordagem é aquela cuja facilidade de implementação é maior, embora tenha a desvantagem de não contribuir de forma significativa para a diminuição da impedância semântica. Nesta abordagem, o programador teria de desenvolver em dois ambientes diferentes: definir o esquema e operações sobre a base de dados com base no modelo relacional, e converter explicitamente os dados da base de dados para os objectos da linguagem de programação, para os poder utilizar no âmbito da linguagem de programação. A figura 4.4 pretende ilustrar esta abordagem.

As restantes abordagens contribuem de facto para a diminuição da impedância semântica, embora a sua facilidade de implementação seja menor do que a abordagem anterior. Esta dificuldade adicional é devida essencialmente à necessidade de construção de pré-processadores específicos para a extensão da linguagem Objective- C. As diferenças em termos das duas abordagens, resumem-se à possível compatibilidade com futuros desenvolvimentos na área dos SGBDOs.

Concretamente, a segunda abordagem propõe a extensão da linguagem Objective-C por forma a dotá-la de capacidades de sistemas de bases de dados. A extensão a definir deveria ser realizada de tal forma que o recurso a um SGBDR para armazenamento dos dados de objectos fosse o mais transparente possível para o programador e utilizador final do sistema.

A terceira abordagem, tendo objectivos muito idênticos aos da anterior, tem no entanto a enorme vantagem de não propôr uma extensão qualquer às linguagens de programação, mas sim uma que constitui uma proposta de norma dum grupo de trabalho representativo de cerca de 90% do mercado actual de SGBDOs, e por conseguinte com maior probabilidade de aceitação generalizada. Embora a proposta actual possa ainda não ser perfeita, consegue desde já dar um grande passo em frente relativamente aos sistemas tradicionais, e ao mesmo tempo tem o cuidado de deixar em aberto algumas questões importantes ainda em debate, tais como: os sistemas de bases de dados do futuro serão fruto da *evolução* dos sistemas relacionais actuais (sistemas relacionais estendidos), ou serão sistemas totalmente novos construídos de raíz com base num modelo objecto (ainda actualmente sem consenso...) ? Esta abordagem é a ilustrada na figura 4.3.

Na proposta não são feitas quaisquer referências à forma como devem ser implementadas fisicamente as capacidades de bases de dados disponibilizadas ao nível da linguagem de programação. Este grau de liberdade permitirá o aparecimento de uma diversidade de implementações, certamente diferentes em termos de performance e aplicabilidade, e por conseguinte permitirá também que a inovação continue a impulsionar a evolução neste campo.

A abordagem a utilizar para delinear a interface em Objective-C será assim aquela que tem como base a proposta de norma ODMG-93.

4.2.2 Princípios Orientadores

A extensão da linguagem Objective-C, resultante do mapeamento das linguagens ODL, OML e OQL, será elaborada de modo que a linguagem resultante, do ponto de vista do programador, pareça ser uma única linguagem e não duas linguagens separadas com fronteiras mal definidas entre elas. Para tal, é necessário que se verifiquem as seguintes condições:

1. Deverá existir um sistema de tipos unificado entre a linguagem de programação e a base de dados; as instâncias destes tipos comuns tanto poderão ser transitórias como persistentes.

2. A extensão da linguagem Objective-C com ODL/OML deverá respeitar a sintaxe e semântica da linguagem Objective-C base.

3. A extensão deverá ser estruturada com base em poucas construções sintácticas.

4. As expressões em OML devem poder ser livremente combinadas com expressões da linguagem Objective-C base.

Tendo como base os princípios acima enumerados, garante-se que as principais ferramentas de desenvolvimento disponíveis em ambiente NeXTSTEP poderão continuar a ser utilizadas para as aplicações baseadas na extensão de Objective-C, o que é sem dúvida um excelente resultado. Estas ferramentas são as seguintes:

- *Interface Builder:* ferramenta para o desenvolvimento de interfaces com o utilizador [NeX92d].

- *Project Builder:* ferramenta para desenvolvimento de projectos [NeX92g].

- *AppInspector:* ferramenta para inspeccionar informação associada às aplicações.

- *ProcessMonitor:* ferramenta para examinar processos em execução [NeX92f].

A extensão a delinear com base na ODMG-93 deixará em aberto os detalhes de implementação física das capacidades de bases de dados disponibilizadas ao nível da linguagem de programação. Este facto permitirá que numa primeira abordagem se possa recorrer a um SGBDR disponível em ambiente NeXTSTEP para fornecer a funcionalidade de base de dados. A forma como tal funcionalidade será implementada não será descrita aqui, e constituirá tema para um futuro trabalho na sequência deste.

4.2.3 Enquadramento no Ambiente NeXTSTEP

O SGBDO, para o qual se pretende neste trabalho delinear uma interface em Objective-C, constituirá uma das componentes da ferramenta MOOT anteriormente apresentada na secção 2.1.1.

Para melhor compreensão da função a desempenhar pelo SGBDO no processo de desenvolvimento de sistemas orientados por objectos em NeXTSTEP, serão enumeradas as várias ferramentas envolvidas bem como as funções desempenhadas no processo. Na figura 4.5 é ilustrada a interligação destas ferramentas.

As ferramentas envolvidas são as seguintes:

- *Ferramenta MOOT:* análise e concepção do sistema.
- *Editor de Programas:* edição das classes do sistema durante a fase de codificação.
- *Interface Builder:* concepção e criação da interface com o utilizador da aplicação.
- *Project Builder:* gestão e manutenção do projecto do sistema, auxiliando nas fases de codificação, teste e integração.
- *Pré-processadores e Compiladores:* geração de código.
- *Debugger, AppInspector, ProcessMonitor:* teste e correcção do sistema.
- *SGBDO:* gestão da informação relevante associada às várias fases do projecto: análise, concepção, codificação e documentação; e gestão da informação persistente manipulada pelo próprio sistema desenvolvido.

O SGBDO será utilizado com duas finalidades distintas. A primeira, ainda na fase de desenvolvimento do sistema, para armazenamento da informação relevante associada ao projecto, e manipulada pela ferramenta MOOT. A informação a armazenar será de natureza diversa: diagramas de classes, referências para os ficheiros das classes e seus conteúdos, documentos de análise, erros comunicados, etc.

A segunda corresponde à utilização do SGBDO pela própria aplicação desenvolvida, no caso de esta ter requisitos em termos de gestão de bases de dados.

4.2.4 Componentes da Interface

A interface a desenvolver em NeXTSTEP para o SGBDO será constituída pelas seguintes componentes:

- *Editor Gráfico de Esquemas:* a ferramenta MOOT poderá ser utilizada para este efeito, atendendo a que é possível a geração de código Objective-C estendido com ODL, tendo como base os diagramas de classes OMT. De facto, a linguagem ODL e os diagramas de classes OMT possuem características comuns que tornam possível esta conversão.

- *Pré-processador de ODL/OML:* poderá ser construído por extensão de um analisador léxico/sintáctico existente para Objective-C, ou então concebido desde o zero para a gramática de Objective-C estendida para integrar ODL e OML.

- *Biblioteca de Classes do Modelo Objecto:* por forma a mapear o modelo objecto ODMG para Objective-C, terão de ser criadas as classes base do modelo: vectores, listas, conjuntos, etc.

- *Palette para o Interface Builder:* por forma a facilitar o desenvolvimento de interfaces gráficas com o *Interface Builder* para as aplicações de bases de dados que irão utilizar o SGBDO, considera-se de grande utilidade a construção de uma *palette* específica para este tipo de interfaces. Uma *palette* consiste num conjunto de classes de objectos que podem ser incorporados numa interface gráfica desenvolvida com o *Interface Builder*. Nas bibliotecas NeXTSTEP existem várias *palettes* pré-definidas com: janelas, botões, caixas, menus, campos de texto, etc. As enormes vantagens desta ferramenta estão relacionadas com o seu caracter dinâmico, o que permite o teste da interface, sem ter de a compilar.

- *Funções Específicas de Bases de Dados:* a funcionalidade de base de dados disponibilizada ao nível da linguagem Objective-C estendida, deve ser suportada directa ou indirectamente por um SGBD. Esta funcionalidade será colocada ao nível da linguagem estendida, por intermédio de funções específicas escritas para tal, que serão integradas no executável da aplicação.

4.2.5 Mapeamento do Modelo Objecto ODMG para Objective-C

O mapeamento do modelo objecto ODMG para Objective-C implica o mapeamento das várias noções envolvidas, concretamente as noções de: objectos e literais, relações, extensões, chaves, nomes, interface e implementações de um tipo, e a hierarquia de tipos base do modelo.

O modelo de dados da linguagem Objective-C, pela sua proximidade em termos de conceitos com o modelo objecto ODMG, simplificará de forma significativa este processo de mapeamento.

Seguidamente, e para cada uma das noções acima será então explicado como será feito o seu mapeamento para construções de Objective-C.

Objectos e Literais

Cada tipo ODMG será mapeado para uma classe Objective-C. Dependendo da forma como a classe é instanciada, o resultado poderá ser um objecto ou um literal ODMG. Assim, por exemplo, um objecto `a` incorporado num outro `b` mais alto-nível, será considerado um literal ODMG pelo facto de `b` não poder ser copiado sem que `a` também o seja. Neste sentido, pode-se afirmar que a identidade dos dois se confunde, e que portanto não deve ser

considerada a identidade de a. Se a não possui identidade, então não pode ser considerado um objecto mas apenas um literal.

Um objecto ODMG poderá ser implementado como uma referência para um objecto Objective-C.

Relações

Tal como na generalidade das linguagens de programação, este conceito não é suportado por Objective-C, pelo que será simulado através de métodos criados automaticamente para tal. A integridade das relações será mantida pelo SGBDO.

Para uma dada extremidade *para-um* `tPath` de uma relação, existirão os seguintes métodos:

```
// métodos da classe tPath
- (void)set:TargetClass;
- get;
- (BOOL)exists;
- (BOOL)exists:TargetClass;
- (void)delete;
// métodos da classe toOnePath
- (BOOL)exists;
- (BOOL)exists:TargetClass;
- (void)delete;
```

De igual modo, para uma dada extremidade *para-muitos* `tPath` de uma relação, existirão os seguintes métodos:

```
// métodos da classe tPath
- (void)insert:TargetClass;
- (void)remove:TargetClass;
- (Iterator *)create_iterator:(BOOL)unStable;
- (BOOL)exists;
- (BOOL)exists:TargetClass;
- (void)delete;
```

Extensões

Este conceito, não sendo directamente suportado pela linguagem, pode no entanto ser facilmente integrado através da declaração de variáveis estáticas no âmbito do ficheiro que contém a implementação da classe, conjuntamente com a redefinição dos métodos para criação das instâncias da classe (ver classe `Object` em [NeX92a]).

Chaves

O conceito de chave também não é suportado pela linguagem Objective-C. Poderá mais uma vez ser simulado por recurso a métodos definidos eventualmente com base nas classes que compõem o *Indexing Kit* disponível no NeXTSTEP [NeX92c].

Nomes

Esta noção poderá ser implementada adicionando à classe raíz de Objective-C—a classe `Object`, os métodos que permitem atribuir nomes aos objectos e questioná- los sobre os

seus nomes. Os nomes dos objectos são definidos no âmbito global da base de dados, pelo que não é possível a atribuição de dois nomes iguais a objectos diferentes.

Interface e Implementação

A linguagem Objective-C tem implícita a noção de separar a interface de uma classe da sua implementação. Uma interface não pode ter no entanto mais do que uma implementação em Objective-C.

Hierarquia de Tipos Base

Da hierarquia de tipos base do modelo objecto ODMG, o tipo `Array` será mapeado directamente pela sintaxe da linguagem Objective-C. Para os restantes tipos terão de ser criadas classes, classes estas que não poderão ser parametrizadas, uma vez que Objective-C não o permite.

4.2.6 Mapeamento das linguagens ODL, OML e OQL

Nesta secção será delineada a extensão da linguagem Objective-C para mapeamento das linguagens de definição, manipulação e interrogação.

ODL

A linguagem ODL destina-se à definição do esquema da base de dados. O esquema da base de dados de um SGBDO é um conjunto de classes, em que cada classe tem atributos, relações e operações. A linguagem ODL terá assim de possibilitar a declaração dos três tipos de propriedades dos tipos do modelo ODMG.

Tanto a declaração dos atributos como operações não levanta qualquer problema em termos sintáticos, pois podem ser feitas de forma idêntica às declarações normais de variáveis e métodos em Objective-C ODL.

Para que seja possível a declaração dos atributos inversos de uma relação torna-se necessária a introdução de uma nova palavra reservada, **inverse**. Assim, a definição de uma relação consistirá numa declaração do tipo e nome dos objectos envolvidos na relação, seguida da palavra *inverse* e da declaração da outra extremidade da relação, isto é, novamente o tipo e nome do atributo.

Para melhor ilustrar estas ideias, apresenta-se em baixo um exemplo de definição de uma interface de um tipo.

```
/* conjuntamente com a redefinição do método (-init)
   permite implementar as extensões da classe */
static List textos;
@interface Texto:  Documento {
   char *referência;
   List figuras inverse [Figura textos] order by número;
   Pessoa revisor inverse [Pessoa textos_revistos];
};
- init;
- (void)alterar_tamanho_página:  (int)altura:  (int)largura;
- (int) obter_no_linhas;
@end
```

No exemplo é declarada uma variável estática que conjuntamente com a redefinição do método de criação de objectos - `init` permitirá simular o conceito de extensão e coleccionar o conjunto de todas as instâncias que vão sendo criadas para esta classe. Nas relações os atributos inversos são declarados após a nova palavra reservada `inverse` com a sintaxe de mensagem em Objective-C (`[`*receptor mensagem*`]`).

A declaração de atributos inversos é opcional, pelo que a sua omissão resultará em relações unidireccionais, isto é, relações cuja consistência fica à responsabilidade da aplicação e não do SGBDO.

O tratamento de excepções poderá ser implementado com o recurso à biblioteca de funções existentes em NeXTSTEP para este fim (ver [NeX92b]).

OML

Um princípio orientador da definição da Objective-C OML foi a de utilização de sintaxe para criação, destruição, modificação de propriedades, invocação de métodos, em tudo idêntica à utilizada para objectos não-persistentes.

Assim, a criação de objectos em Objective-C OML será realizada através dos métodos de classe + `alloc` ou + `allocInDatabase:`, respectivamente para objectos transitórios ou objectos persistentes. A persistência de um objecto pode assim ser decidida na altura de criação do objecto, ou, depois de criado o objecto, através do método de classe + `givePersistence:`. Deste modo, garante-se a ortogonalidade dos conceitos de persistência e tipos. Isto é, qualquer tipo poderá ter instâncias persistentes e transitórias simultaneamente.

Os métodos de classe em Objective-C distinguem-se dos métodos de instâncias pela utilização respectiva de + ou − como prefixo da assinatura do método. Em baixo indicam-se as assinaturas dos métodos acima referidos:

```
+ alloc; // para objectos transitórios
+ allocInDatabase:  (char *)dbname // para objectos persistentes
+ givePersistence:  (id)tothisObject // torna os objectos persistentes
```

A destruição de objectos persistentes será realizada através de métodos em tudo simétricos aos anteriores para criação:

```
- free; // para objectos transitórios
- freeFromDatabase // para objectos persistentes
+ removePersistence // para tornar objectos transitórios
```

Através da definição de uma categoria `Database` para a classe `Object` da biblioteca de classes de NeXTSTEP estes métodos poderão ser tornados acessíveis a todos os objectos de uma aplicação de base de dados [NeX92e].

Ao nível de Objective-C OML, serão tratadas de forma idêntica as operações de modificação e referenciação de atributos bem como a invocação de métodos, para objectos transitórios ou persistentes. Ficará a cargo da linguagem estendida a distinção entre estas situações.

As relações entre instâncias, por outro lado, poderão ser criadas, destruídas e percorridas através de métodos definidos para tal, tanto para as extremidades *para-um* das relações como para as extremidades *para-muitos*. Estes métodos deverão ser implementados com bae nas operações definidas no modelo objecto ODMG, e que pela sua extensão não serão aqui formalmente definidos mas apenas ilustrados através de um

exemplo. Em [CAD+94] encontram-se enumeradas todas as operações que devem ser implementadas para dar suporte ao conceito de relações.

```
Texto *texto; Figura *figura; Pessoa *pessoa;
...
// coloca uma referência para a pessoa que reviu o texto e
// adiciona automaticamente um texto
// à lista de textos revistos por essa pessoa
texto->revisor = pessoa;
// adiciona uma figura à lista de figuras do texto e automaticamente
// o sistema adiciona um texto à lista de textos da figura respectiva
[texto- >figuras insert:figura];
```

No exemplo, podemos ver a utilização de dois tipos de relações: uma de um-para-muitos e outra de muitos-para-muitos, respectivamente entre `Pessoa` e `Texto`, e entre `Texto` e `Figura`.

OQL

A linguagem Objective-C OQL poderá ser implementada de três maneiras diferentes, com crescente dificuldade de implementação mas também crescente facilidade de utilização:

- um método definido na classe genérica `Collection`
- uma função `oql` independente e não associada a nenhuma classe
- ou integrando em Objective-C uma sublinguagem de interrogação, a OQL.

É a útlima abordagem, aquela que melhor contribui para a diminuição de impedância semântica ao permitir que expressões de interrogações sejam livremente misturadas com expressões nativas da linguagem. A abordagem de definição de uma função autónoma também é agradável, muito embora obrigue a que a optimização das interrogações só possa ser feita em tempo de execução, e dificulte um pouco mais a utilização livre de expressões nativas e resultados de interrogações.

Para a interface a desenvolver pretende-se seguir a última abordagem das acima referidas.

```
Set autores;
int idade;
...
idade=65;
autores = (select distinct y from x in Textos, y in x.autores
              where y.idade=idade);
if(![autores empty]) reformar(autores);
```

O exemplo acima ilustra a utilização de dados de Objective-C na formulação de interrogações em OQL.

4.2.7 Hierarquia de Tipos Base do Modelo

Após a implementação do modelo objecto, da ODL, e da OML, o passo seguinte consistirá na implementação das classes para listas, vectores, conjuntos, etc, suportadas pelo modelo objecto.

A hierarquia de tipos colecção definida no modelo objecto ODMG pode ser directamente codificada para classes de Objective-C. Nesta hierarquia o tipo `Collection` é o tipo a partir do qual os subtipos `Array`, `Set`, `Bag`, e `List` são definidos.

O modelo objecto ODMG permite a definição de tipos genéricos, isto é, tipos definidos com base em parâmetros, que por sua vez também são tipos. Assim, é possível definir um tipo genérico como por exemplo:

```
Set<T>
```

e com base nele definir um objecto do tipo "conjuntos de conjuntos de inteiros", instanciando o parâmetro `T` com `Set<int>`:

```
Set<Set<int>> umConjunto;
```

A linguagem Objective-C não possui esta capacidade de definição de classes genéricas, pelo que as colecções genéricas do modelo serão mapeadas para colecções de objectos da classe raíz de Objective-C—a classe `Object`. A desvantagem desta definição reside na impossibilidade de com ela impedir o compilador de fazer a verificação estática de tipos, e assim adiar para a execução a eventual detecção de utilizações erradas de instâncias desses tipos.

A classe `Array` será directamente suportada por Objective-C, pelo que seguidamente apenas serão apresentadas as definições das classes `Collection`, `Set`, `Bag` e `List`.

Collection

```
@interface Collection
+ alloc
- init
- free
- (unsigned long)cardinality;
- (BOOL)empty;
- (BOOL)ordered;
- (BOOL)allows_duplicates;
- (BOOL)is_equal_to:(Collection)anOtherCollection;
- (BOOL)is_different_from:(Collection)anOtherCollection;
- (Collection *)copy;
- (Iterator *)create_iterator:(BOOL)unStable;
- (void)insert_element:theElement;
- (void)remove_element:theElement;
- (void)remove_element_at:(Iterator *)position;
- (void)replace_element_at:(Iterator *)position:by;
- retrieve_element_at:(Iterator *)position;
- select_element:(const char *)predicate;
- (Iterator *)select:(const char *)predicate;
- (int)query:(Collection *)result:(const char *)fullOQLpredicate;
- (BOOL)exists_element:(const char *)predicate;
- (BOOL)contains_element:theElement;
@end
```

Set

```
@interface Set:  Collection
+ alloc
- init
- free
- (int)query:(Set *)result:(const char *)fullOQLpredicate;
- (Set *)union:(Set *)with;
- (Set *)intersection:(Set *)with;
- (Set *)difference:(Set *)with;
- (Set *)copy;
- (BOOL)is_subset_of:(Set *)set;
- (BOOL)is_proper_subset_of:(Set *)set;
- (BOOL)is_superset_of:(Set *)set;
- (BOOL)is_proper_superset_of:(Set *)set;
@end
```

Bag

```
@interface Bag:  Collection
+ alloc
- init
- free
- (int)query:(Bag *)result:(const char *)fullOQLpredicate;
- (Bag *)union:(Bag*)with;
- (Bag *)intersection:(Bag *)with;
- (Bag *)difference:(Bag *)with;
@end
```

List

```
@interface List:  Collection
+ alloc
- init
- free
- (void)replace_element_at:(int)position:by;
- (List *)concat:(List*)with;
- (void)append:theElement;
- (void)insert_element_after:(int)position:theElement;
- (void)insert_element_before:(int)position:theElement;
- (void)insert_element_first:theElement;
- (void)insert_element_last:theElement;
- (void)remove_element_at:(int)position;
- (void)remove_first_element;
- (void)remove_last_element;
- retrieve_element_at:(int)position;
- retrieve_first_element;
- retrieve_last_element;
- (int)find_element:theElement;
- (int)query:(List *)result:(const char *)fullOQLpredicate;
@end
```

Iterator

A classe `Iterator` é utilizada pelas classes colecção para definir o comportamento de iteração sobre as colecções. Cada objecto desta classe deve ser criado através da invocação do método `create_iterator` da classe `Collection`

```
@interface Iterator
+ alloc
- init
- free
- (void)reset;
- (BOOL)next:(id *)theNextElement;
- (void)close;
@end
```

O método foi definido como retornando o próximo elemento da colecção através do parâmetro passado por referência e retorna simultâneamente um valor lógico que indica se foi retornado ou não um novo elemento. Um pequeno exemplo é apresentado a seguir:

```
Iterator it = [textos create_iterator];
Texto *t;
...
while([it next:&t]) { ...}
```

4.2.8 Classes *Database* e *Transaction*

A funcionalidade de base de dados é disponibilizada ao nível da linguagem de programação através de objectos das classes `Database` e `Transaction`.

Os objectos das classes `Database` e `Transaction` são objectos transitórios. Antes de poderem ser iniciadas quaisquer transacções que utilizem uma base de dados, esta deve ser aberta. De igual modo, a base de dados deve ser fechada depois de todas as transacções terem sido terminadas ou interrompidas. O acesso, criação, modificação e destruição de objectos persistentes só pode ser feito no contexto de uma transacção.

A criação de uma transacção não inicia a transacção, pelo que as transacções devem ser explicitamente iniciadas. As transacções podem também ser encaixadas, isto é, podem ser criadas transacções dentro de transacções.

Transaction

As transacções serão implementadas em Objective-C à custa de objectos da classe `Transaction`. Nesta classe são definidos métodos para iniciar, terminar, e interromper transacções.

A definição da classe `Transaction` é a seguinte:

```
enum lock_type {read_lock, write_lock, default_lock };
@interface Transaction
+ alloc;
- init;
- free;
- (void)start;
- (void)commit;
- (void)abort;
- (void)checkpoint;
@end
```

Database

Os métodos definidos na classe `Database` permitem abrir, fechar, e destruir bases de dados, e ainda uma série de operações para atribuição de nomes a objectos, modificação de nomes de objectos, e acesso a objectos através dos seus nomes.

A definição da classe `Database` é a seguinte:

```
enum AccessStatus {NotOpen, ReadWrite, ReadOnly, Exclusive };
@interface Database
- (void)open:(const char*)dbname:(AccessStatus)withAccessMode;
- (void)close;
- (void)delete;
- (void)name:theObject:(const char*)withName;
- (void)name:theObject:(const char*)withName;
- (const char *)name:oftheObject;
- (void)rename:(const char*)oldName:(const char*)newName;
- (void)lookup:theObject:(const char*)withName:(lock_type)lockType;
@end
```

A definição da classes para as transacções e para as bases de dados, finaliza o mapeamento para a linguagem de programação Objective-C.

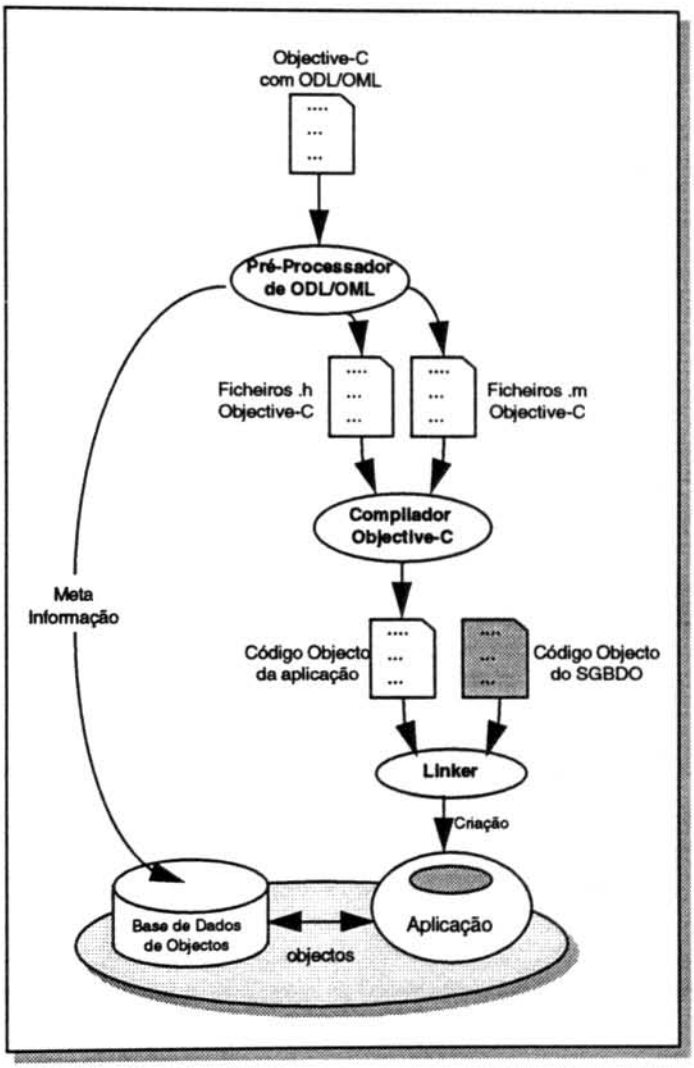


Figura 4.3: Passos para a criação de uma aplicação para um SGBDO

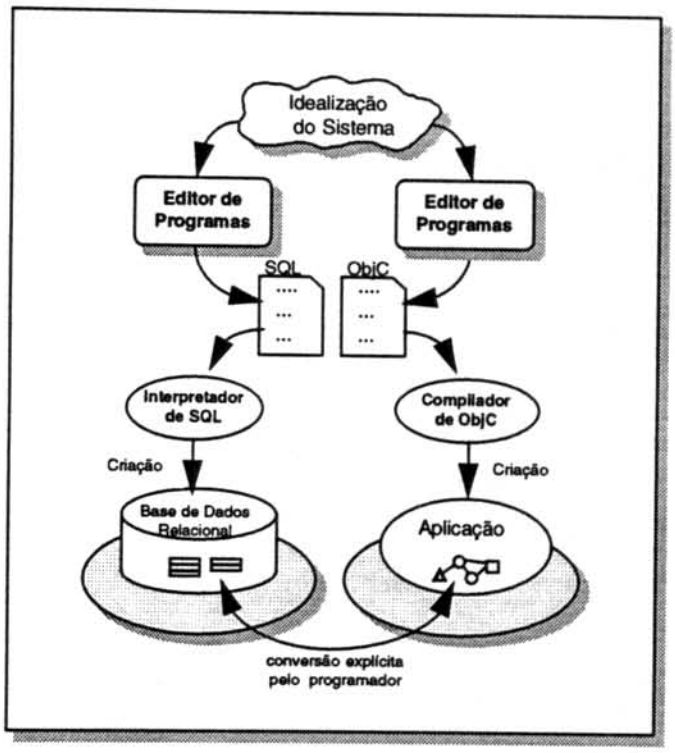


Figura 4.4: Uma abordagem simples para a interface a definir

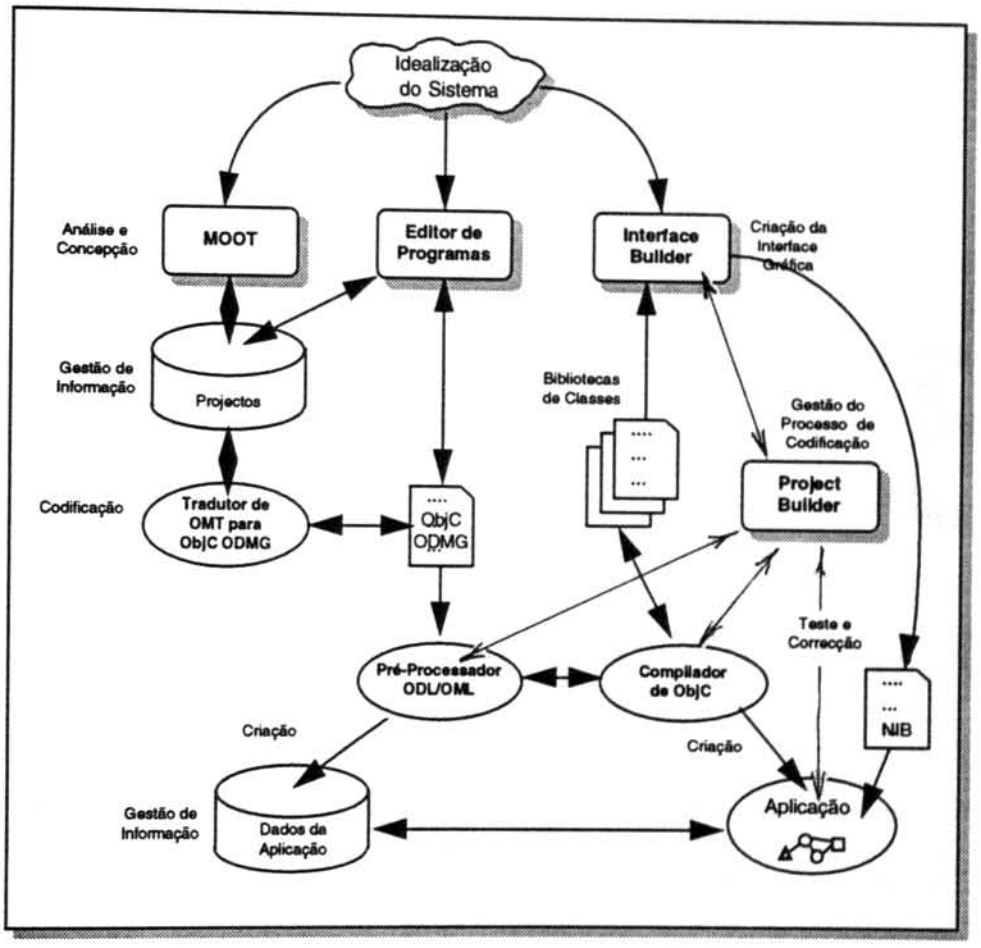


Figura 4.5: As várias ferramentas de desenvolvimento disponíveis em NeXTSTEP

5

Conclusões

Ano a ano, os sucessivos avanços da tecnologia de *hardware* têm proporcionado reduções significativas nos preços dos CPU's, memórias e discos, tornando os sistemas de computadores acessíveis a um cada vez maior número de aplicações. À medida que a aplicabilidade dos computadores aumenta, novos requisitos vão surgindo e novos problemas se levantam em termos de complexidade e dimensão dos sistemas a desenvolver.

Os Sistemas de Gestão de Bases de Dados tradicionais (SGBD) demonstraram ser inadequados para diversos tipos de aplicações de utilização cada vez mais corrente: ferramentas CASE, aplicações de CAD/CAM, aplicações de hipertexto, sistemas multimedia e sistemas de informação urbana e geográfica.

A engenharia de software, acompanhando a necessidade crescente de desenvolvimento de sistemas complexos, tem obtido resultados e originado o aparecimento de novas metodologias de desenvolvimento de software, novas linguagens de programação e novos sistemas de bases de dados.

O presente trabalho enquadrando-se globalmente na área das bases de dados, incidiu sobretudo no estudo dos novos sistemas de bases de dados, tendo sido estudadas as novas

aplicações de bases de dados, os novos modelos de dados, as linguagens de programação orientada por objectos, e as novas características oferecidas pelos sistemas de bases de dados.

5.1 Contributo Principal

No âmbito do Grupo TEIA, encontra-se em desenvolvimento um protótipo de ferramenta CASE para apoiar o processo de desenvolvimento de sistemas de software orientados por objectos: a ferramenta MOOT—*Modeling Object Oriented Tool*. Esta ferramenta CASE tipo *Front-end*, apoiará sobretudo as fases de análise e concepção, e codificação inicial do sistema.

Uma componente da ferramenta, ainda por desenvolver, é a sua componente de bases de dados. Na versão actual, os dados persistentes dos projectos são armazenados em ficheiros, pretendendo-se a sua substituição por um SGBD. Pelas limitações inerentes aos SGBD's tradicionais, e na indisponibilidade de aquisição de um novo sistema de bases de dados para a MOOT, decidiu-se desenvolver um sistema com funcionalidade típica de um SGBDO. Foi tendo em vista a criação deste SGBDO, que o presente trabalho foi desenvolvido.

Tendo em consideração a dimensão e complexidade inerente ao projecto e desenvolvimento do sistema em questão, o problema foi decomposto em partes.

O objectivo principal deste trabalho consistiu em delinear a implementação uma interface em ambiente NexXTSTEP para um sistema de bases de dados. Por razões de normalização do sistema a desenvolver futuramente, de entre as várias abordagens possíveis seguir optou-se por definir a referida interface com base numa proposta de norma para os SGBDO's, a ODMG-93.

5.2 Resultados

Um dos resultados importantes deste trabalho consiste no levantamento e selecção da imensa bibliografia existente sobre o tema central do trabalho, o qual resultou no estudo de diversas questões directa e indirectamente relacionadas com o desenvolvimento e utilização da nova geração de sistemas de gestão de bases de dados de objectos: aplicações, modelos, linguagens, ferramentas, características de bases de dados e seus pormenores de implementação, etc.

A definição da interface para Objective-C, com base na proposta de norma ODMG-93, bem como o seu enquadramento no processo de desenvolvimento do futuro SGBDO, constitui sem dúvida, um outro resultado igualmente importante do trabalho, atendendo a que potencia a elaboração de vários trabalhos na área deste e em consonância com os objectivos deste: a construção de um Sistema de Gestão de Bases de Dados de Objectos.

5.3 Limitações

A maior limitação deste trabalho reside no facto de ter sido desenvolvido tendo em vista a linguagem Objective-C, e não C++. Considera-se ser uma limitação não pelo facto de Objective-C ser pior ou melhor do que C++, mas por razões de facilidade de migração do software desenvolvido para outras plataformas, como por exemplo ambiente Windows.

A linguagem Objective-C em ambiente NeXTSTEP, embora possua algumas limitações relativamente a C++, como por exemplo não permitir a redefinição de operadores e não ter classes genéricas, apresenta no entanto algumas vantagens no que diz respeito a ferramentas de desenvolvimento, bibliotecas de classes, e de capacidades oferecidas pela própria linguagem, nomeadamente as categorias e os protocolos.

Esta limitação não é considerada muito negativa para o projecto em questão, atendendo a que se considera mais importante conceber e prototipar facilmente o sistema num ambiente de desenvolvimento minoritário em termos de divulgação, do que o fazer num ambiente de utilização quase universal mas que levanta inúmeros problemas de desenvolvimento, em grande parte devido à sua concepção.

5.4 Questões em Aberto

Atendendo a que o presente trabalho é um dos primeiros trabalhos realizados no sentido de concretização do objectivo mais global de construção de um SGBDO para integração na ferramenta MOOT, as questões deixadas em aberto pelo presente trabalho são forçosamente muitas mais do que aquelas nele resolvidas. De facto, o presente trabalho pretende também constituir um ponto de partida para a resolução dessas questões agora deixadas em aberto.

5.5 Futuros Desenvolvimentos

Com vista à continuação deste trabalho e à concretização das ideias nele delineadas, é essencial que venham a ser realizados os trabalhos seguintes:

- pré-processador de Objective-C ODL/OML
- tradutor de diagramas de classes OMT para Objective-C ODL
- biblioteca de classes em Objective-C para o Modelo Objecto ODMG
- *palette* para o Interface Builder para aplicações de bases de dados
- funções específicas de bases de dados para implementar a funcionalidade disponibilizada ao nível da linguagem Objective-C OML.

B

Bibliografia

[Agu92] Ademar Aguiar. Sistemas de gestão de dados de objectos para apoio ao desenvolvimento de aplicações. Trabalho da disciplina de mestrado: Projecto, FEUP/INESC, Junho 1992.

[AS92] Ademar Aguiar and Mário Santos. Objectos persistentes e bases de dados relacionais. Trabalho da disciplina de mestrado: Sistemas de informação e dases de dados, FEUP/INESC, Março 1992.

[Atw93] Thomas Atwood. The object dbms standard. *In Object Magazine*, pages 37–44, Outubro 1993.

[Bar91] Thierry Barsalou. Toward expert database systems:integration of object-oriented methods with a relational architecture. In *Conference on "Databases in Modern System Architectures"*, Stanford, California, Agosto 1991.

[Bee92] Catriel Beeri. Object-oriented databases-models and query languages. In *3rd International Conference on "Extending Database Technology"*, Vienna, Austria, Mar o 1992.

[Bon92] José Maria Bonnet. Especificação do dicionário de dados para a análise funcional em omt. Trabalho da disciplina de mestrado: Sistemas de informação e dases de dados, FEUP/INESC, Abril 1992.

[Bon94] José Maria Bonnet. *Predição de Séries no Tempo Utilizando Redes Neuronais.* FEUP, FEUP/INESC, Abril 1994. Dissertação submetida para satisfação parcial dos requisitos do programa do curso de Mestrado em Engenharia Electrótecnica e de Computadores.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin / Cummings Publishing Company Inc., Redwood City; California, 1991.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The gemstone object database management system. *In Communicatios of the ACM*, 34(10):64–77, Outubro 1991.

[CAD+94] R. G. G. Cattell, Tom Atwood, Joshua Duhl, Guy Ferran, Mary Loomis, and Drew Wade. *The Object Database Standard, ODMG-93.* Morgan Kaufmann Publishers, 1994.

[Cat91] R. G. G. Cattell. *Object Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley Publishing Company, 1991.

[D+91] O. Deux et al. The o2 system. *In Communicatios of the ACM*, 34(10):34–39, Outubro 1991.

[Dat90] C. J. Date. *Introduction to Database Systems.* Addison-Wesley, Massachusetts, 1990.

[DeM78] T. DeMarco. *Structured Design and System Specification.* Prentice Hall,Englewood Cliffs, New Jersey, 1978.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company; AT&T Bell Laboratories, Massachusetts, 1990.

[F+89] D. H. Fishman et al. Overview of the iris dbms. Technical report HPL-SAL-89-15, Hewlett-Packard Company, Janeiro 1989.

[Far90] João Pascoal Faria. *SAGA - Uma Ferramenta Interactiva para a Produção e Manutenção de Aplicações de Bases de Dados.* Outubro 1990. Trabalho de síntese elaborado para as provas de aptidão pedagógica e capacidade científica, apresentado à Faculdade de Engenharia da Universidade do Porto.

[G+91] Rajiv Gupta et al. *Object-Oriented Databases with Applications to CASE,Networks, and VLSI CAD.* Series in Data and Knowledge Base Systems. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[KL89] W. Kim and F. Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, Addison-Wesley Publishing Company, 1989.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The object store database system. *In Communicatios of the ACM*, 34(10):50–63, Outubro 1991.

[LLPS91] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to starburst: Objects, types, functions, and rules. *In Communicatios of the ACM*, 34(10):94–109, Outubro 1991.

[Mey88] Bertarnd Meyer. *Object-Oriented Software Construction.* Prentice Hall International (UK), Hertfordshire, 1988.

[Mir92] Serge Miranda. Relational data model and sql revisited. In *Next Generation Database Systems.* EuroPace, Maio 1992.

[MZ90] David Maier and Stanley B. Zdonik. *Readings in Object-Oriented Database Systems.* The Morgan Kaufmann Publishers, San Mateo, California, 1990.

[MZ91] David Maier and Stanley Zdonik. The object-oriented database system manifesto. In *Conference on "Databases in Modern System Architectures"*, Stanford, California, Agosto 1991.

[Net93] João Neto. Ferramenta para navegação em diagramas de classes omt. Trabalho da disciplina de mestrado: Sistemas de informação e dases de dados, FEUP/INESC, Abril 1993.

[NeX92a] NeXTComputer. Application kit, 1992. online manual.

[NeX92b] NeXTComputer. Exception handling, 1992. online manual.

[NeX92c] NeXTComputer. Indexing kit, 1992. online manual.

[NeX92d] NeXTComputer. The interface builder application, 1992. online manual.

[NeX92e] NeXTComputer. Objective-c, 1992. online manual.

[NeX92f] NeXTComputer. The process monitor, 1992. online manual.

[NeX92g] NeXTComputer. The projectbuilder application, 1992. online manual.

[NeX92h] NeXTComputer. System overview, 1992. online manual.

[Ran92] João Vasco Ranito. Especificação do dicionário de dados para a análise dinmica da metodologia omt. Trabalho da disciplina de mestrado: Sistemas de informação e dases de dados, FEUP/INESC, Abril 1992.

[RBL+91] James Rumbaugh, Michael Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs; New Jersey, 1991.

[RBP88] James E. Rumbaugh, Michael R. Blaha, and William J. Premerlani. Relational database design using an object-oriented methodology. *In Communicatios of the ACM*, 31(4):414–427, Abril 1988.

[RBPV90a] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, and Thomas A. Varwig. An object-oriented relational database. *In Communicatios of the ACM*, 33(11):99–107, Novembro 1990.

[RBPV90b] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, and Thomas A. Varwig. An object-oriented relational database. *In Communications of the ACM*, 33(11):99–107, Novembro 1990.

[RF91] João Vasco Ranito and João Pascoal Faria. Saga - sistema assistido de gestão/geração de aplicações : uma ferramenta interactiva para o desenvolvimento expedito e fácil manutenção de aplicações de bases de dados. *Endiel 91*, 1991. Lisboa.

[Ris91] Tore Risch. Introduction to object-oriented database management systems and iris. In *Conference on "Databases in Modern System Architectures"*, Stanford, California, Agosto 1991.

[San93] Mário Santos. *Estrutura de Diálogo para um CASE-OO de Modelização Estática baseado em OMT.* FEUP, FEUP/INESC, Novembro 1993. Dissertação submetida para satisfação parcial dos requisitos do programa do curso de Mestrado em Engenharia Electrótecnica e de Computadores.

[Sha84] K. Shankar. *Handbook of Software Engineering.* Van Nostrand Reinhold, New York, 1984.

[Sim92] Jorge Manuel Simões. Especificação de um dicionário de dados para a análise estática em omt. Trabalho da disciplina de mestrado: Sistemas de informação e dases de dados, FEUP/INESC, Abril 1992.

[Sim93] Jorge Manuel Simões. *Estudo de Arquitecturas de Sistemas de Gestão de Bases de Dados que Suportem Controlo de Versões e Históricos.* FEUP, FEUP/INESC, Julho 1993. Dissertação submetida para satisfação parcial dos requisitos do programa do curso de Mestrado em Engenharia Electrótecnica e de Computadores.

[SK91] Michael Stonebraker and Greg Kemnitz. The postgres next-generation database management system. *In Communicatios of the ACM*, 34(10):78–93, Outubro 1991.

[SM92] J.W. Schmidt and F. Matthes. Database programming: Buld types and languages. In *3rd International Conference on "Extending Database Technology"*, Vienna, Austria, Mar o 1992.

[SMOW88] S. Shlaer, S. J. Mellor, D. Ohlsen, and H. Wayne. *Object-Oriented System Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[SSU91] Avi Silberschatz, Michael Stonebraker, and Jeff Ullman. Database systems: Achievements and opportunities. *In Communicatios of the ACM*, 34(10):110–120, Outubro 1991.

[Sto88] Michael Stonebraker. *Readings in Database Systems.* The Morgan Kaufmann Publishers, San Mateo, California, 1988.

[Str88] B. Stroustrup. What is object-oriented programming ? *IEEE Software*, 5(10):10, Maio 1988.

[Str93] Henry Strickland. Odmg-93 the object database standard for c++. *In C++ Report*, pages 45–48, Outubro 1993.

[Ull89a] Jeffrey D. Ullman. *Principles of Database and Knowledge-Based Systems*, volume 1. Computer Science Press, Rockyville, Maryland, 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Based Systems*, volume 2. Computer Science Press, Rockyville, Maryland, 1989.

[Vel92] F. Velez. Object-oriented database systems-implementation issues. In *3rd International Conference on "Extending Database Technology"*, Vienna, Austria, Mar o 1992.

[VRF93] R. Moreira Vidal, João Vasco Ranito, and João Pascoal Faria. Building interactive database applications with views and rules. (INESC), Julho 1993. (Submetido à "Sixth International Conference on Software Engineering & Its Applications").

[Wie77] G. Wiederhold. *Database Design.* McGraw-Hill, New York, 1977.

[YC78] E. Yourdon and L. Constantine. *Structured Design.* Yourdon Press, Englewood Cliffs, New Jersey, 1978.

FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO
BIBLIOTECA
0000065073