**Integração sob demanda de esquemas
de aplicações desenvolvidas utilizando
bibliotecas de componentes genéricos**

Autor: Júlio Cesar dos Santos

Orientador: Prof. Dr. Caetano Traina Jr

*Dissertação apresentada ao Instituto de
Ciências Matemáticas e de Computação
de São Carlos - USP, como parte dos
requisitos para a obtenção do título de
Mestre em Ciências - Área de Ciências
de Computação e Matemática
Computacional.*

USP - São Carlos
Junho de 1998

A Deus, sem o qual, NADA é possível.

**Aos meus familiares, pelas oportunidades
oferecidas e pelo apoio.**

Agradecimentos

À Mirla pela paciência, carinho, compreensão e incentivo durante todo tempo.

Ao meu orientador Prof. Dr. Caetano Traina Júnior, pelo apoio, compreensão, amizade e confiança.

Aos membros Grupo de Bases de Dados e Imagens, em especial, ao amigo João Eduardo pelo companherismo e pelo apoio.

Aos funcionários deste Instituto pela colaboração no desenvolvimento deste trabalho.

À CAPES pelo apoio financeiro.

Aos amigos, Tatuí e Gelza, Tanão, Matite, Araxá , Marisa, Ciro, Mário, Fábio, e Rogério (Dei Santi, Mouro, Florian e “Prudentino”) pela oportunidade que me deram de “crescer”.

Capítulo 01 - Introdução	1
1.1. <i>Motivação</i>	2
1.2. <i>Objetivos</i>	3
1.3. <i>Organização do Trabalho</i>	3
Capítulo 02 - Modelos de Dados Orientados a Objetos	5
2.1. <i>Conceitos Iniciais de Sistemas de Gerenciamento de Base de Dados Orientados a Objetos</i>	6
2.2. <i>Modelo ORION</i>	7
2.2.1. <i>Introdução</i>	7
2.2.1.1. <i>Características Gerais</i>	8
2.2.2. <i>Arquitetura do ORION</i>	9
2.2.3. <i>Evolução de Esquemas</i>	10
2.2.3.1. <i>Taxonomia da Evolução de Esquema</i>	11
2.2.3.2. <i>Invariantes de Evolução de Esquemas</i>	12
2.2.3.3. <i>Regras de Resolução de Conflitos</i>	13
2.2.3.4. <i>Regras de Propagação de Propriedades</i>	14
2.2.3.5. <i>Regras de Manipulação de DAG – Rooted and Connected Directed Acyclic Graph</i>	15
2.2.3.6. <i>Regras de Objetos Compostos</i>	16
2.2.4. <i>Controle de Versões</i>	16
2.2.4.1. <i>Versões</i>	17
2.2.5. <i>Objetos Compostos</i>	18
2.3. <i>Modelo O₂</i>	19
2.3.1. <i>Introdução</i>	19
2.3.1.1. <i>Visão geral do Modelo O₂</i>	19
2.3.2. <i>Arquitetura do O₂</i>	20
2.3.3. <i>Descrição Informal do Modelo</i>	21
2.3.3.1. <i>Objetos e Valores</i>	21

2.3.3.2. Tipos e Classes	22
2.4. Considerações Finais	23
Capítulo 03 - Ferramentas para RAD - Rapid Application Development	25
3.1. Considerações Iniciais	26
3.2. Conceitos de Orientação a Objetos	26
3.2.1. Encapsulamento	26
3.2.2. Herança	27
3.2.3. Polimorfismo	30
3.2.4. Identificação de Objetos	30
3.3. Ferramentas para RAD - Rapid Application Development	31
3.3.1. Histórico	31
3.3.2. Características comuns às ferramentas	32
3.3.3. BTO - Biblioteca de Templates de Objetos	32
3.3.3.1. Especialização de Classes	32
3.3.3.2. Objetos Padrão	33
3.3.3.3. Categorias Existentes	33
3.4. Conclusão	37
Capítulo 04 - Ferramentas para RAD Avaliadas	38
4.1. Considerações Iniciais	39
4.2. PowerObjects 1.0 da Oracle	39
4.2.1. Arquitetura de Objetos	40
4.2.1.1. Objetos de Aplicação	40
4.2.1.2. Objetos de Biblioteca	41
4.2.1.3. Objetos de Sessão	41
4.2.2. Aplicação de Conceitos Orientados a Objetos	42
4.2.3. BTO	43
4.3. PowerBuilder 5.0 da PowerSoft	44

4.3.1. Arquitetura de Objetos	44
4.3.2. Aplicação de Conceitos Orientados a Objetos	46
4.3.2.1. Encapsulamento	46
4.3.2.2. Herança	46
4.3.2.3. Polimorfismo	46
4.3.2.4. Identificação de objetos	46
4.3.3. PowerBuilder Foundation Class Library - PFC	47
4.3.3.1. Encapsulamento	47
4.3.3.2. Polimorfismo	47
4.4. PowerBuilder X PowerObjects	49
4.5. Conclusão	50
Capítulo 05 - Integração de Esquemas utilizando uma Estrutura Genérica	51
5.1. Considerações Iniciais	52
5.2. O Processo de Integração	53
5.3. Invariantes de Integração de Esquemas	58
5.3.1. Invariante de hierarquia de classes	59
5.3.2. Invariante de identificador	59
5.3.3. Invariante de herança	59
5.4. Regras de integração de Esquemas	59
5.4.1. Regras de manipulação da hierarquia da estrutura	60
5.4.2. Regras de Atributos	61
5.5. Conclusão	65
Capítulo 06 - Conclusões e Sugestões de Futuras Pesquisas	66
6.1. Conclusões	67
6.2. Resultados Alcançados	68
6.3. Sugestões de Futuras Pesquisas	69

<i>Bibliografia</i>	71
----------------------------	-----------

<i>Referências Bibliográficas</i>	72
--	-----------

Listas de Figuras

<i>Figura 2.1: Arquitetura do ORION [KIM89c]</i>	9
<i>Figura 2.2: Arquitetura O₂ [FIG96]</i>	20
<i>Figura 3.1 : Encapsulamento da classe Funcionário [MON94]</i>	27
<i>Figura 3.2 : Estrutura Hierárquica para Modelagem dos Dados da Classe Veículo -Herança Simples [MON94]</i>	28
<i>Figura 3.3 : Hierarquia com estruturas Generalização/Especialização na classificação de Bens[MON94]</i>	29
<i>Figura 3.4 : Generalização/Especialização</i>	34
<i>Figura 3.5 : Generalização/Especialização após a inclusão de uma nova classe</i>	35
<i>Figura 3.6 : Objeto 01</i>	35
<i>Figura 3.7 : Objeto 02</i>	36
<i>Figura 3.8 : Objeto 03</i>	36
<i>Figura 3.9 : Objeto 04</i>	37
<i>Figura 4.1: Hierarquia de Objetos Primitivos [FIN96]</i>	40
<i>Figura 4.2: Objetos de Aplicação – Adaptado de [FIN96]</i>	40
<i>Figura 4.3: Biblioteca de Objetos [FIN96]</i>	41
<i>Figura 4.4: Objetos de Sessão [FIN96]</i>	41
<i>Figura 4.5: Hierarquia de Objetos – Adaptado de [NAN96]</i>	45
<i>Figura 4.6: Objetos Visuais – Adaptado de [NAN96]</i>	45
<i>Figura 4.7: Objetos de Janela -- Adaptado de [NAN96]</i>	45
<i>Figura 4.8: Polimorfismo Operacional – Adaptado de [SYB96d]</i>	47
<i>Figura 4.9: Polimorfismo Inclusional– Adaptado de [SYB96d]</i>	48
<i>Figura 5.1: Processo de Integração</i>	53
<i>Figura 5.2 : Modelagem simplificada de uma cooperativa prestadora de serviços médicos</i>	54
<i>Figura 5.3 : Modelagem simplificada de uma instituição de ensino</i>	56
<i>Figura 5.4 : Estrutura Genérica – Pessoa</i>	57
<i>Figura 5.5 : Representação da estrutura padrão para SI e S2</i>	63

Lista de Tabelas

<i>Tabela 2.1 – ORION x O2 – Adaptado de [CAM92]</i>	24
<i>Tabela 4.1 - Conceitos de Orientação a Objeto</i>	49
<i>Tabela 4.2 - Categorias de BTO</i>	49

Resumo

Atualmente as organizações devem trocar dados entre si, e a tendência é que estas trocas sejam efetuadas cada vez mais através de meios digitais. Os dados são consultados (alimentados) livremente nas bases de dados de organizações independentes entre si, porém quando é necessária a troca de dados, como não existe uma previsão de integração, os dados somente podem ser trocados após uma preparação que impõe alguma forma de intervenção manual, construção de filtros especiais, etc., uma vez que a não existência de um esquema comum impede que os dados de uma base possa ser intercambiados com os de outra base. No entanto, embora as bases de dados de diferentes organizações possam ser construídas de maneira totalmente independentes, a necessidade de uma troca significa que a semântica dos elementos manipulados, em particular daqueles que devem ser compartilhados é, no mínimo, semelhante. Por exemplo, se duas organizações devem trocar dados sobre pessoas, não importa se para as diferentes organizações as pessoas são clientes, empregados, alunos ou pacientes, o significado de “pessoa” é sempre entendido pelos membros das organizações. O mesmo ocorre com qualquer entidade que se deseje trocar informações.

Este trabalho parte da suposição que possa existir alguma forma de definição primitiva para os diversos elementos de dados que devem ser compartilhados, a partir da qual sua instanciação em elementos de um esquema de dados em particular possa ser reconhecido. Assim, busca-se identificar estruturas primitivas, com a finalidade de integrar os sistemas. Contudo, para se chegar a tal estrutura, é preciso definir regras que garantam a preservação de propriedades da mesma com a finalidade de permitir que sempre que o esquema de uma organização A, construído a partir de uma mesma estrutura primitiva que foi utilizada por outro esquema de uma organização B, a integração entre ambas possa ser feita de forma automática, e que isso não provoque inconsistência nas bases de dados que estarão recebendo as informações.

Para atingir esses objetivos, este trabalho propõe que tais estruturas de esquemas primitivos sejam parte das Bibliotecas de *Templates* de Objetos (BTO) que são distribuídas como parte das Ferramentas de Desenvolvimento Rápido de Aplicativos (RAD) comerciais. Como um exemplo de como isso poderia ser feito, apresenta-se um componente que define genericamente um objeto pessoa, desenvolvido a partir de como “pessoas” são tratadas em dois sistemas reais, centrando essa estrutura em uma única Abstração: a Abstração de Generalização. Apresenta também o conjunto de regras que deve ser utilizado para a integração de componentes centrados nessa abstração, que é a mais universalmente compatível entre os diversos modelos de dados orientados a objetos disponíveis atualmente.

Abstract

Nowadays, the organizations need to deliver and receive data mutually, and the tendency is that the interchanges are made ever more through digital means. Although the databases of different organizations can (and must) be totally built in an independent way, when some elements must be interchanged, the semantic of these elements are at least similar. For example, if two organizations should interchange data on people, it does not care, for the different organizations, if the people are customers, employees, students or patient, the means of what are "people" is always understood by each organization.

This work states that some form of primitive, common definition can exist for the data elements that must be shared, from which many elements of a database schema should be recognized. Thus, it searches for primitive structures that should be used by the several systems with the purpose of integrating them. Besides the definition of such structures, is necessary also to define rules that guarantee the preservation of the properties from the data that follow these structure in different organizations, when an interchange is pursued.

To meet these objectives, this work suggests the development of common, generic Schema Components, that could be distributed as part of the Reusable Components Libraries of commercial Rapid Application Development (RAD) tools. As an example of this approach, it is presented one Primitive Component, which defines a generic "people" component. This component was extracted from the analysis of two existing systems, and it was structured using only one abstraction: the generalization one. It presents also the set of rules that must be followed to integrate data from two databases that have a subschema based on that abstraction. This sole abstraction was chosen due to be the most universally compatible concept among the present object-oriented data models.

Capítulo 1.

Introdução

1.1. Motivação

O desenvolvimento de sistemas de informação, em qualquer tipo de organização, tem utilizado cada vez mais o suporte de Sistemas de Gerenciamento de Bases de Dados, cuja frente tecnológica atual apoia-se nos Gerenciadores Relacionais, no projeto e no desenvolvimento baseados no paradigma de Orientação a Objetos. Os aplicativos construídos nas organizações recaem na existência de uma Administração de Dados Central, inclusive quando mais de um Sistema Gerenciador de Dados é utilizado (bases de dados federadas).

Atualmente, as organizações devem trocar dados entre si, e a tendência é que estas trocas sejam efetuadas cada vez mais através de meios digitais. Os dados são consultados/alimentados nas bases de dados de organizações independentes entre si, através de alguma forma de intervenção manual, uma vez que a não existência de um esquema comum impede que os dados de uma base possam ser intercambiados com os de outra base.

Constata-se que mais e mais organizações devem interagir em instantes bem determinados, de maneira efêmera, e sem uma previsão de quando ou quais informações devem ser trocadas. Assim, deve-se preservar a independência do gerenciamento e da estrutura das bases de uma organização, ao mesmo tempo em que é importante propiciar meios para que as bases de dados de duas organizações independentes possam compartilhar informações.

Embora as bases de dados de diferentes organizações possam ser construídas de maneira totalmente independentes, a semântica dos elementos manipulados, em particular daqueles que devem ser compartilhados, é no mínimo semelhante. Por exemplo, se duas organizações devem trocar dados sobre pessoas, não importa se para as diferentes organizações as pessoas são clientes, empregados, donos ou pacientes, o significado de pessoas é sempre entendido pelos membros das organizações e muitos dos dados mantidos para pessoas em ambas as organizações têm o mesmo significado. Assim, parte-se da suposição que, nestes dados, possa existir alguma forma de definição primitiva para os diversos elementos que devem ser compartilhados, a partir do qual sua instanciação em elementos de um esquema de dados numa organização em particular possa ser reconhecido.

1.2. Objetivos

Este projeto de mestrado tem o objetivo de avaliar técnicas que permitam a integração de dados de organizações sob demanda, admitindo-se que, embora desenvolvidos isoladamente, os esquemas de ambas tenham utilizado componentes básicos de construção comum. Tais componentes poderiam, a princípio, estar disponíveis em “Bibliotecas de *Templates* de Objetos” (BTO) que estão se tornando obrigatórias na grande maioria das ferramentas de desenvolvimento rápido de aplicativos (ferramentas para RAD - *Rapid Application Development*). Assim, este trabalho tem como ponto de partida a avaliação de como estruturas de modelagem podem evoluir, preservando propriedades que possam contribuir para uma integração dinâmica dos esquemas de duas bases de dados construídas isoladamente, mas a partir de uma mesma BTO.

É bom salientar que essas ferramentas se apoiam nas técnicas de projeto associadas ao paradigma de orientação a objetos o que corresponde hoje à tendência predominante para o desenvolvimento de novos aplicativos.

Numa aplicação de desenvolvimento utilizando BTO, os programadores instanciam os elementos da biblioteca em elementos do esquema da aplicação, e a seguir personalizam esses objetos, para adequarem-se às necessidades do aplicativo e da organização . Este projeto tem como objetivo estudar ferramentas de desenvolvimento de software disponíveis comercialmente (foram estudadas as ferramentas *PowerObjects 1.0* da *Oracle* e *PowerBuilder 5.0* da *PowerSoft*) para, a partir das estruturas sintáticas pré-definidas nas respectivas bibliotecas, e das operações de edição dessas estruturas, disponibilizadas ao projetista, definir para cada estrutura sintática quais operações preservam que propriedades, com o objetivo de permitir uma futura integração de esquemas desenvolvidos independentemente, mas com as mesmas ferramentas e utilizando as classes definidas nas mesmas BTO.

1.3. Organização do Trabalho

Este trabalho está organizado da seguinte maneira:

No segundo capítulo são tratados os modelos de dados orientados a objetos seguindo a ótica voltada para os conceitos de esquema de dados, visto que o tema do projeto a ser desenvolvido “Integração sob demanda de esquemas de aplicações desenvolvidas utilizando

bibliotecas de componentes genéricos”, está intimamente ligado a esses conceitos. A título de estudo são apresentados os modelos ORION e O₂.

No terceiro capítulo são analisadas as ferramentas RAD - *Rapid Application Development* - de maneira geral, mostrando aspectos históricos das mesmas, bem como os principais conceitos que as caracterizam.

No quarto capítulo são estudadas as ferramentas *PowerObjects* da *Oracle* e *PowerBuilder* da *PowerSoft* buscando levantar características e aspectos relevantes para a escolha da ferramenta que será utilizada para o desenvolvimento do projeto.

No quinto capítulo é mostrado o processo de integração de esquemas sob demanda propriamente dito. Partiu-se de dois sistemas desenvolvidos de forma independente com o intuito de identificar uma estrutura genérica que estivesse presente no mesmo. Depois de identificada a estrutura, definem-se as invariantes e regras de integração necessárias para que se preserve a consistência da informação a ser trocada entre os dois sistemas. A identificação da estrutura comum e a definição de invariantes e de regras de integração representam a base para que seja possível a construção de sistemas independentes, que possam comunicar-se esporadicamente para a troca de informações de maneira automática. Nesse capítulo é mostrada uma estrutura que poderia estar presente nas BTO das ferramentas de desenvolvimento e que poderia ser utilizada para permitir que se reconhecesse automática e mutuamente ao menos parte da estrutura da informação trocada, baseando-se no fato de que ambos os sistemas foram desenvolvidos a partir de uma biblioteca básica, instanciada, de *templates* de objetos.

Por fim, no sexto capítulo são apresentadas as conclusões do trabalho, discutem-se as decisões de projeto que foram adotadas em seu desenvolvimento, e são feitas sugestões no sentido de mostrar como a estrutura identificada pode vir a ser incorporada em uma biblioteca das ferramentas de mercado existentes.

Capítulo 2.

Modelos de Datos Orientados a Objetos

2.1. Conceitos Iniciais de Sistemas de Gerenciamento de Base de Dados Orientados a Objetos

A aplicação do paradigma da “Orientação a Objetos” não é restrita a sua área original de linguagens de programação, atingindo várias áreas da computação tais como: Sistemas de Representação do Conhecimento, Sistemas Gráficos e Sistemas de Bases de Dados [CAM92][CHO93].

Dentro do contexto de um Sistema de Gerenciamento de Base de Dados Orientados a Objetos-SGBDOO, foram incorporados os conceitos derivados de linguagem de programação orientada a objetos [BER91][TAK90] apud [CAM92], entre os quais estão:

Objetos: representa um elemento do mundo real, sendo cada Objeto caracterizado por suas propriedades <Atributo, valor> e possuindo métodos associados que representam seu aspecto operacional. Um Objeto é diferenciado de outros Objetos por um identificador único fornecido pelo sistema ou pelo usuário.

Método: comando ou sequência de instruções associadas a um identificador próprio (seletor). Um método aplica uma operação sobre Objetos associados e atribui a resposta à uma Mensagem. Os Métodos representam a interface do Objeto com o mundo externo, sendo que os detalhes sobre sua implementação são ocultos ao usuário.

Mensagem: elemento criado para solicitar a execução de um Método, ou então criado depois de um Método finalizado, sendo portanto, um elemento de comunicação entre Objetos.

Classe: conjunto de Objetos agrupados por terem as mesmas características estruturais, funcionais e comportamentais.

Além desses conceitos, existem outros que podem ser considerados extensões ou assemelham-se às representações usadas nos Modelos Semânticos, como é o caso de objetos, que correspondem a conjuntos de entidade no ME-R e de:

Relacionamento: associação explícita que estabelece graus de dependência entre os Objetos associados.

Atributos: elemento de caracterização de Objetos e Relacionamentos através de valores. Os Atributos podem ser atualizados pelo usuário ou dinamicamente derivados de outros Atributos por meio de Métodos [CAM92].

Por outro lado, a caracterização de um Gerenciador de Bases de Dados como sendo Orientado a Objetos, deve atender a um conjunto mínimo de requisitos. Devido à ampla

quantidade de variações que se emprega junto ao termo “orientação a objetos”, em particular oriundos de consórcios entre empresas fornecedoras e consumidoras de sistemas desse tipo, têm surgido grupos de definição de padrões, tais como o OMS - *Object Management System* e ODMG - *Object Database Management Group* [BET94][CAT94] que procuram dar uma definição universal dos termos e definição de conceitos. Esses padrões têm seguido as definições obtidas em consenso a partir de discussões que ocorrem desde o final da década passada [ATK89][STO90], onde os seguintes conceitos são caracterizados como constituintes de um sistema orientado a objetos:

- Definição de um objeto através de seu tipo ou da classe a que pertence;
- Identificação de Objetos por seu Identificador unívoco para todo o sistema;
- Suporte a objetos complexos (objetos compostos por outros);
- Encapsulamento e Herança de estrutura e de métodos;
- Suporte a Polimorfismo;
- Completeza computacional em todas as operações, tanto para acesso às instâncias quanto para acesso ao esquema;
- Ortogonalidade entre todos os conceitos envolvidos, permitindo a extensibilidade dos modelos;
- Persistência, concorrência e gerenciamento de dados em disco (embora tais considerações, quanto aos gerenciadores de bases de dados não sejam relevantes);
- Capacidade de recuperação automática de situações de erro, mesmo (e principalmente) aquelas originárias do próprio sistema;
- Suporte para consultas não programadas.

2.2. Modelo ORION

2.2.1. Introdução

O projeto ORION [KIM89c] foi lançado em 1985 pela “*Microelectronics and Computer Technology Corporation*”- MCC, Austin - Texas, tendo como alvo o desenvolvimento de protótipo de Sistemas de Bases de Dados Orientados a Objetos, com o intuito de suportar aplicações CAD/CAE (*Computer Aided Design/Computer Aided Engeneering*), e sistemas de informatização de escritórios. [CAM92]. Dispõe de recursos para o desenvolvimento de

aplicações que utilizam recursos de Inteligência Artificial e de suporte a Documentos Multimídia.

O sistema ORION começou a ser desenvolvido como um banco de dados multitarefa, mono-usuário, rodando em estações de trabalho. Foi originalmente implementado em COMMON LISP em uma máquina *Symbolic 3600 LISP*, na versão denominada ORION-1. Foi posteriormente portado para estações de trabalho SUN sob o sistema operacional UNIX, operando como um sistema cliente-servidor, que se caracterizou como a versão ORION-1SX e operando como um sistema distribuído, que se caracterizou como a versão ORION-2. Atualmente esse sistema está sendo desenvolvido para distribuição comercial, numa nova versão denominada ITASCA, através de uma empresa criada por seus projetores iniciais (denominada *ITASCA Corporation INC.*) [KIM90][KIM95].

2.2.1.1. Características Gerais

É importante ressaltar que o ORION é um modelo e um sistema que se destaca dos demais gerenciadores de objetos, por sua capacidade de gerenciar Versões, Objetos Compostos e Evolução de Esquemas. Essas características tornam-no muito adequado a adaptações estruturalmente sofisticadas, como é o caso de aplicações CAD/CAM, automação industrial e de escritórios.

A concepção ORION e a construção de seus vários protótipos ocorreram principalmente no período de 1985 a 1993. Seus idealizadores passaram a desenvolvê-lo como um produto comercial, que encontra-se disponível em versões β em ambientes de estações de trabalho sob SUN.

Dentre as várias características avançadas presentes no ORION podemos citar: Controle de Versões [CHO86] *apud* [KIM89c], Objetos Compostos [KIM87], Evolução de Esquemas [BAN87b], [BAN87a] *apud* [BAN87b], Gerenciamento de transações e Gerenciamento de dados multimídia [WOE86] *apud* [KIM89c], [WOE87].

Antes de se considerar as características suportadas, será feita uma breve explicação a respeito da arquitetura do modelo ORION, com o intuito de mostrar sua organização em módulos e tratar as funcionalidades de cada um.

2.2.2. Arquitetura do ORION

Figura 2.1: Arquitetura do ORION [KIM89c]

Segundo [KIM89c] a arquitetura apresentada na figura 2.1 descreve o ORION em grandes módulos, os quais têm a seguinte funcionalidade: O manipulador de mensagens - *message handler* - recebe todas as mensagens enviadas ao sistema ORION. Estas mensagens podem ser classificadas como:

- *user-defined methods* : métodos definidos pelo usuário, que são armazenados no ORION;
- *access message*: mensagens que recuperam e modificam o valor de um atributo da classe;
- *system-defined functions* : mensagens compostas por todas as funções disponíveis no ORION para definição de esquemas, criação e remoção de instâncias e gerenciamento de transações.

Todas as mensagens são editadas, criando-se um mapa de alterações geradas em decorrência de cada mensagem. Esse mapa é então validado, e somente se as alterações forem corretas, com efeitos colaterais que não causam inconsistências no sistema, é que as solicitações são atendidas. Esse mecanismo permite suporte ao gerenciamento de versões e alterações na hierarquia de generalização de uma aplicação.

O subsistema de objetos (*object subsystem*) possui funções de alto nível, tais como evolução de esquemas, controle de versões, otimizações de consultas e gerenciamento de informações multimídia.

O subsistema de armazenamento (*storage subsystem*) provê acesso aos objetos do disco. Este subsistema gerencia a alocação e liberação de segmentos de páginas no disco, para possibilitar a referência correta aos objetos desejados. Além disso, gerencia também índices e atributos de classe, com o intuito de aumentar a performance das consultas associativas.

O subsistema de transações (*transaction subsystem*) provê o controle de concorrência e um mecanismo de recuperação para proteger a integridade do banco de dados, visto que permite a execução de múltiplas transações concorrentemente. O controle de concorrência utiliza-se de um protocolo de bloqueio e de um mecanismo de *login* que está disponível para auxiliar a recuperação de quedas sofridas pelo sistema.

Tratou-se da arquitetura do modelo, e em seguida tratar-se-á das características avançadas citadas anteriormente. Contudo características como gerenciamento de transações e gerenciamento de dados multimídia não são consideradas por não fazer parte do contexto do trabalho (conceitos associados a esquemas de dados).

2.2.3. Evolução de Esquemas

Aplicações típicas para as quais o sistema ORION é voltado, requerem flexibilidade considerável na definição e modificação de esquemas de banco de dados de forma dinâmica, isto é, as definições de classes e estrutura de herança na hierarquia de classes (*class lattice*) são também passíveis de alterações [WOE86]. Sistemas orientados a objetos existentes suportam somente alguns poucos tipos de troca de esquema sem precisar reinicializá-los, porque eles são sistemas tipicamente oriundos do desenvolvimento de linguagens de programação. Mesmo os sistemas de banco de dados convencionais existentes permitem somente poucas mudanças no esquema, depois que a base esteja inicializada. Por exemplo o SQL/DS permite somente a criação e remoção dinâmica de relações (classes) e adição de novas colunas (atributos) na relação. Isto ocorre porque as aplicações que eles suportam (aplicações comerciais convencionais orientadas a registros) não requerem muitas mudanças no esquema, e também porque os modelos de dados ao quais eles se destinam não são ricos como os modelos orientados a objetos tendo um conjunto de conseqüências decorrentes de cada mudança relativamente restrito.

Segundo [KIM89c], desenvolveu-se uma taxonomia de operações de troca de esquemas que pode ser suportada pelos sistemas de banco de dados sem que seja preciso sua reinicialização ou mesmo, sua reorganização. Além disso, foi também desenvolvido um “*framework*” formal que consiste de um conjunto de propriedades de esquemas chamadas “*invariants*”, e um conjunto de regras para preservar as mesmas. Os invariantes têm uma base de definição da semântica de troca de esquema significativa, assegurando que a troca não deixará o esquema em um estado inconsistente, isto é, um estado que viole qualquer invariante.

Entretanto para algumas trocas de esquema, os invariantes podem ser preservados de mais de uma maneira. As regras guiam a seleção da maneira mais adequada. Na próxima seção discute-se mais detalhadamente a taxonomia empregada no sistema ORION.

2.2.3.1. Taxonomia da Evolução de Esquema

O sistema ORION permite que uma base de dados já inicializada, com dados já armazenados, possa ter um esquema modificado. Permite também que diferentes versões de um mesmo aspecto modelado possam existir simultaneamente.

O sistema ORION suporta dois tipos de mudanças de esquema, a saber: no conteúdo e na estrutura da hierarquia de classes. As mudanças podem ser melhor referenciadas como: mudança na definição de uma classe (conteúdo do nó) na hierarquia de classes, e mudanças na estrutura (extremidade do nó) da hierarquia de classes. Trocar as definições de classes incluem acrescentar e apagar atributos e métodos. Trocar a estrutura da hierarquia consiste em criar e apagar uma classe, e alterar o relacionamento de generalização entre classes. Essa estrutura com ligações IS_A entre as classes na hierarquia de generalização é que suporta herança [CAM92].

As mudanças no esquema padrão podem afetar fisicamente a Base de Dados. Atualizações dos dados da Base que foram afetados por mudanças do esquema, são tratadas pelo sistema ORION utilizando uma abordagem chamada Conversão Incremental. Esta abordagem consiste em efetuar a atualização das informações à medida que estas vão sendo requisitadas pelo sistema. Contudo, esta abordagem compromete a velocidade de execução das operações, visto que deve-se considerar o tempo gasto na conversão [CAM92].

Além disso, o ORION suporta também a abordagem chamada *Screening* - Atualização Adiada - que consiste na utilização de versões de esquema da Base de Dados, permitindo ao usuário criar várias versões do esquema lógico, de modo que os usuários possam ter diferentes “versões lógicas” da Base de Dados, indicando-se qual a versão a ser utilizada [CAM92]. Esta

abordagem também compromete a velocidade de execução de qualquer operação, uma vez que a informação apresentada aos usuários deve ser modificada de acordo com a versão do esquema de dados.

Na próxima seção discute-se o conjunto de propriedades de esquemas de dados chamadas “*invariants*”. Estas propriedades têm uma base de definição de semântica de troca de esquema significativa, assegurando que a troca não deixará o esquema num estado inconsistente.

2.2.3.2. Invariantes de Evolução de Esquemas

No modelo de dados ORION foram identificados cinco invariantes em esquemas orientados a objetos. Segundo [KIM89c], eles podem ser assim resumidos.

Invariante da Hierarquia de Classes - Class Lattice Invariant

A hierarquia de classe pode ser considerada como um grafo acíclico dirigido conectado (DAG) - em que os nós possuem nomes e suas extremidades são rotuladas. O grafo possui apenas uma raiz intitulada OBJETO e subclasses que definem os domínios primitivos (inteiro, *boolean*, *string*, *float* e *set*). Não existe nó isolado e cada nó possui um nome único dentro da estrutura.

Invariante de Nome Distinto - Distinct Name Invariant

Toda variável instanciada de uma classe, quer seja definida ou herdada, deve possuir um único nome. A mesma regra aplica-se aos métodos de uma classe.

Invariante de Identificador Distinto - Distinct Identify (Origin) Invariant

Toda variável instanciada e métodos de uma classe devem ter um identificador unívoco em todo o sistema.

Invariante de Herança - Full Inheritance Invariant

Uma classe herda todas as variáveis instanciadas e os métodos daquela que é sua superclasse, salvo quando isto viola a regra *distinct name invariant* ou a regra *distinct identify invariant*. Isto é, se duas variáveis instanciadas têm origem distinta mas o mesmo nome nas suas origens (por exemplo, duas superclasses), é a última delas que deverá ser herdada. Por outro lado se duas variáveis instanciadas tem a mesma origem dentro de duas superclasses somente uma delas poderá ser herdada.

Invariante de Compatibilidade de Domínio - Domain Compatibility Invariant

Se uma variável instanciada V2 de uma classe C é herdada de uma variável V1 de uma superclasse de C, então o domínio de V2 é ou o mesmo de V1, ou uma subclasse de V1. Outro ponto a salientar é que o valor compartilhado ou *default* de uma variável instanciada deve ser uma instância da classe que é o domínio de uma variável instanciada.

A hierarquia de classes em um estado deve preservar todas as invariantes. Para alguma troca do esquema entretanto, existe mais de uma maneira de preservar as invariantes. Por exemplo, se existe um conflito de nomes entre variáveis instanciadas herdadas de duas superclasses, a invariante de herança determina que a última das variáveis instanciadas seja herdada. Mas nem sempre isto é o desejado, por isso para definir como são preservadas as invariantes, construiu-se um conjunto de doze regras de evolução de esquema divididos em quatro categorias a saber: regras de resolução de conflitos, regras de propagação de propriedades, regras de manipulação da estrutura da hierarquia e regras de composição de objetos [BAN87b].

2.2.3.3. Regras de Resolução de Conflitos

As regras *default* de resolução de conflitos, segundo [KIM89c], permitem a seleção de uma única maneira de herança quando existe conflito de nome ou identidade. Elas asseguram que as regras de nomes e identidades distintas são garantidas de uma maneira determinística. O usuário ORION, entretanto, deve ignorar estas regras de maneira explícita para resolver os conflitos. As propriedades de uma variável instanciada, definidas ou herdadas de uma classe, podem ser modificadas inúmeras vezes. Particularmente podem ser trocados: seu nome; domínio; valor *default*; valor compartilhado; ou a propriedade de *link* composto. Então uma variável instanciada não valorada e não compartilhada pode mudar de estado, passando a valorada e compartilhada e vice-versa. Além disso, as propriedades de um método pertencente a uma classe devem ser modificadas trocando seu nome e código.

1ª regra

Se uma variável instanciada é definida dentro de uma classe e seu nome é o mesmo de uma definida dentro de suas superclasses, a variável de escopo local prevalece sobre as herdadas. O mesmo vale para os métodos da classe.

2ª regra

Se duas ou mais superclasses de uma classe C têm variáveis instanciadas com o mesmo nome mas origem distinta, a variável escolhida para herança é a da primeira superclasse dentro do conflito de superclasses. O mesmo se aplica aos métodos.

3ª regra

Se duas ou mais superclasses possuem variáveis instanciadas com a mesma origem, a variável instanciada terá o domínio de definição mais restrito correspondente aos domínios disponíveis nas definições dentro das superclasses.

2.3.3.4. Regras de Propagação de Propriedades

As propriedades de uma variável instanciada quer seja definida ou herdada de uma classe, pode ser modificada sem limites de vezes. Em particular, seu nome, domínio, valor *default*, valor compartilhado, ou a propriedade de *link* de composição pode ser trocado. Além disso, uma variável instanciada que não possui a característica de compartilhamento de valor pode ter esta propriedade também modificada.

4ª regra

Quando as propriedades de uma variável instanciada ou método numa classe C são trocadas, as trocas devem ser propagadas para todas as subclasses de C, a menos que estas propriedades tenham sido redefinidas dentro das subclasses.

Contudo, como as alterações das propriedades devem ser propagadas dentro de toda a hierarquia de classes relevantes para a variável instanciada, essa propagação pode introduzir novos conflitos dentro da hierarquia de classes, que afetam as invariantes de esquema.

5ª regra

A adição de uma variável instanciada ou um método, ou a troca de nome de uma variável instanciada ou de um método, é propagada somente para aquelas subclasses onde esta mudança de esquema não gera conflitos de nome.

Às vezes trocas de esquemas devem ser rejeitadas. Em particular no modelo ORION o domínio de uma variável instanciada, quer seja definida ou herdada, pode ser generalizado, isto é, pode-se trocá-lo por qualquer um dos domínios válidos pertencentes a uma de suas superclasses, porém não pode ser especializado. Entretanto esta generalização não pode provocar inconsistência para as propriedades de compartilhamento de valor, valor *default* ou valor definido dentro da classe onde se encontra a variável instanciada.

6ª regra

O domínio de uma variável pode ser apenas generalizado.

2.2.3.5. Regras de Manipulação de DAG – *Rooted and Connected Directed Acyclic Graph*

Precisa-se de um conjunto de regras que controle a adição e remoção de classes e *link* entre classes dentro da hierarquia das mesmas. A criação de um *link* entre uma classe A e uma classe B estabelece um “relacionamento” de herança entre as classes (A seria uma superclasse B).

7ª regra

Se uma classe A é adicionada na hierarquia de classes como uma superclasse de B então esta nova classe deve estar no nível mais alto da hierarquia de classe da classe B.

8ª regra

Se o *link* de uma classe A é removido da hierarquia de classes e este que representava o fato de A ser uma superclasse de B então é necessário que a classe B seja ligada a outra classe para não ferir a invariante de hierarquia de classes. Isto quer dizer que a classe B se tornará subclasse de todas as superclasses de A na mesma ordem válida para superclasse em questão.

Esta regra gera um corolário que é o seguinte. Se a uma classe possui somente a superclasse *OBJECT* dentro da hierarquia de classes então qualquer tentativa de remoção deste *link* será rejeitada.

A adição de uma classe deve obedecer a invariante de hierarquia de classes. Isto é, se uma nova classe for adicionada e ficar isolada, ela violará a invariante de hierarquia. Esta violação é tratada pela regra abaixo.

9ª regra

Se nenhuma classe existente na hierarquia de classe for especificada como superclasse para a classe criada então, por *default*, o sistema assume a classe *OBJECT*.

O processo de remoção de uma classe A não é executado apenas por uma operação atômica. Precisa-se primeiramente apagar todos os *links* entre a classe A e suas subclasses. Em seguida a remoção de todos *links* entre a classe em questão e suas superclasses. E finalmente a remoção da classe em questão.

Para preservar a hierarquia de classes é necessário a seguinte regra:

10ª regra

Para a remoção dos *links* de uma classe A com suas subclasses, a oitava regra só será aplicada se todos os *links* se aplicam a subclasses de A, não sendo permitida a remoção de classes definidas pelo sistema, como por exemplo, a classe *OBJECT*.

2.2.3.6. Regras de Objetos Compostos

Uma variável instanciada composta pode ser trocada por uma variável instanciada não composta, isto é, pode perder a propriedade de *link* de composição. Entretanto, não se pode permitir que uma variável instanciada não composta, mais tarde “evolua” e adquira a propriedade de *link* de composição. A razão é que um objeto deve ser referenciado por qualquer número de instâncias de uma classe através de uma variável instanciada não composta, mas somente uma instância de classe através de uma variável instanciada composta. Para trocar uma variável instanciada não composta por uma variável instanciada composta, seria necessário verificar que instâncias existentes não são referenciadas por mais que uma instância através de uma variável instanciada. Isto exigiria que mantivesse uma lista de referência com cada objeto, uma referência para cada variável através da qual um objeto pode ser referenciado. Em particular esta complexidade não foi incorporada em ORION, por isso ele não permite que uma variável não instanciada composta seja trocada por uma variável instanciada composta.

11ª regra

A propriedade de *link* de composição pode ser removida de uma variável instanciada composta. Entretanto, não se pode acrescentá-la a uma variável não instanciada composta. A integridade de uma composição de objetos lida com o fato de que a existência de um objeto composto está vinculada a todos os objetos dependentes.

12ª regra

Se uma variável instanciada composta de um objeto X é trocada por uma não composta, X “disassocia” Y que era referenciado através da variável instanciada. O objeto X continua referenciando o objeto Y, entretanto, a remoção de X não causará a remoção do objeto Y.

2.2.4. Controle de Versões

Existe um consenso de que o controle de versões é uma das mais importantes funções nos vários domínios de aplicações tais como: sistemas integrados CAD/CAM [KAT84][KAT86] apud [KIM89c] e sistemas de informações [WOE86] que lidam com documentos compostos.

Usuários de tais ambientes precisam gerar e experimentar múltiplas versões de um objeto, antes de selecionar um que satisfaça suas necessidades.

Um objeto geralmente referencia outros objetos num nível mais baixo. Um objeto pode ser referenciado (compartilhado) por um número qualquer de objetos. Quando um objeto é modificado ou apagado, ou uma nova versão do objeto é criada, alguns ou todos os objetos que o referenciam devem tornar-se inválidos e, mais que isso, precisam ser notificados da mudança. As próximas seções destinam-se a elucidar como o sistema ORION trata esse tipo de característica.

2.2.4.1. Versões

Segundo [KIM89c] baseado nos tipos de operações que possam ser empregadas nas versões, pode-se classificar o processo de “versionamento”, em dois tipos:

Versão transiente

- . Pode ser modificada pelo usuário.
- . Pode ser apagada pelo usuário.
- . Uma nova versão transiente deve ser derivada de uma versão transiente existente. A versão transiente existente deve ser “promovida” a versão de trabalho;

Versão de trabalho

- . É considerada estável e não pode ser mudada.
- . Pode ser apagada ou consultada.
- . Uma versão transiente pode ser derivada de uma versão de trabalho.
- . Uma versão transiente pode ser “promovida” a uma versão de trabalho.

Neste ambiente, as promoções podem ser explícitas (definidas pelo usuário) ou implícitas (determinadas pelo sistema).

Segundo [KIM89c], a razão para impor restrições quanto à modificação de versões de trabalho, segundo é que estas são consideradas estáveis e, além disso, as versões transientes podem ser derivadas daquelas. Isto acarreta o seguinte fato: se uma determinada versão de trabalho for modificada depois de uma ou mais versões transientes terem sido derivadas daquela, será necessário um conjunto de algoritmos de atualização (para adicionar, apagar e modificar) que assegure que as versões transientes não verão a modificação ocorrida na versão de trabalho que as originou.

Existe duas maneiras para ligar um objeto a outro objeto “versionado”. Na ligação estática, a referência a um objeto inclui o nome completo do objeto, o identificador do objeto e o

número da versão. Na ligação dinâmica, segundo [KAT84] a referência precisa somente do identificador do objeto, e deve deixar o número de versão não especificado. O sistema se encarregará de selecionar o número de versões *default*.

2.2.5. Objetos Compostos

Os conceitos de variáveis instanciadas, domínios e identificadores de objetos, apesar de serem poderosas, não podem representar o relacionamento IS_PART_OF entre objetos. Este relacionamento captura a noção que um objeto é uma parte de outro, constituindo portanto um conceito fundamental em modelagem de dados. Numa extensão do sistema ORION proposta em [KIM89a], as referências (ou relacionamentos) entre objetos ocorrem quando um objeto referencia outros através do identificador. As referências são classificadas em dois tipos: fracas ou de composição. As referências fracas são relacionamentos padrão que existem nos sistemas orientados a objetos. As referências de composição (IS_PART_OF) representam uma semântica onde o objeto pode ser parte de um ou mais objetos. Devido a isto, podem ser classificadas em referência de composição Exclusiva - objeto faz parte exclusivamente de um outro objeto; e referência de composição compartilhada se um objeto faz parte de muitos objetos.

Muitas aplicações possuem a habilidade de definir e manipular um conjunto de objetos como sendo uma entidade lógica simples para os propósitos de integridade semântica, armazenamento e recuperação eficientes [LOC85]. A definição de um objeto composto segundo [KIM89c] é um objeto com uma hierarquia de objetos componentes exclusivos, e uma referência à hierarquia de classes à qual o objeto pertence como uma hierarquia de objeto composto. Uma classe não-*root* numa hierarquia de objeto composto é chamada uma classe componente. Cada classe não folha, numa hierarquia de objeto composto, tem uma ou mais variáveis instanciadas cujos domínios são as classes componentes. Segundo [KIM89c] tais variáveis instanciadas são intituladas variáveis instanciadas compostas. Um objeto constituinte de um objeto composto refere-se a uma instância de sua classe componente através de uma variável instanciada composta. Tal referência é intitulada, segundo [KIM89c], referência composta ou *link* composto entre um objeto e seu objeto dependente (ou entre uma classe e sua classe dependente).

Um objeto dependente é um objeto cuja instância depende da existência de outro objeto. Um objeto dependente não pode ser criado se seu proprietário ainda não existe. Quando um objeto constituinte de um objeto composto for apagado, todos os objetos dependentes daquele deverão ser apagados [KIM89c].

Num objeto composto, nenhum objeto dependente pode ser referenciado mais do que uma vez. Com isso, pode-se dizer que um objeto composto é uma hierarquia de objetos e não um dígrafo. Entretanto, qualquer objeto dentro de um objeto composto pode ser referenciado através de objetos que não pertencem ao objeto composto através de um *link* não composto. Estas referências podem dar origem a um dígrafo.

No domínio de uma classe pode haver mais do que uma variável instanciada composta. Entretanto, somente um objeto pode ter uma referência composta para qualquer instância da classe de um objeto dependente. O mesmo raciocínio se aplica à referência composta [KIM89c].

A evolução de esquemas numa hierarquia de objetos é possível de ocorrer através de uma propriedade de *link* composto. Numa variável instanciada de uma classe essa propriedade é herdada por todas as subclasses daquela classe. Uma variável instanciada composta pode perder a propriedade de um *link* composto. Isto permite que uma variável instanciada composta perca o *link* e se torne não composta, contudo, ela não poderá novamente se tornar composta [KIM89c].

Apesar do gerenciamento de transações e do gerenciamento de dados multimídia serem mencionados como características do modelo ORION não se discutirá os mesmos aqui, visto que não representam características relevantes para o projeto em questão.

2.3. Modelo O₂

2.3.1. Introdução

O modelo e o sistema O₂ foram projetados e desenvolvidos por um consórcio, composto pelo INRIA - *Institute National de la Recherche Informatique et Automatique*, Siemens - Nixdorf, Bull, CNRS - *Centre National de la Recherche Scientifique* e a Universidade de Paris [FIG96]. Tal consórcio constitui-se em um sub-projeto dentro do projeto ESPRIT da comunidade européia.

2.3.1.1. Visão geral do Modelo O₂

No modelo de dados do O₂ as informações são organizadas em Objetos que possuem um identificador, dados (organizados em tuplas ou *sets*) e Métodos encapsulados que propiciam a manipulação dos Objetos.

No modelo existem duas estruturas: tipos constituídos por inteiro, *string*, *float*, *set*, tupla entre outros, que possibilitam a estruturação dos objetos; e classes estruturadas em uma

hierarquia que descreve a estrutura e o comportamento dos objetos, possibilitando a definição de herança (e múltipla herança), de métodos e atributos.

O Esquema de Dados do modelo é definido pelo conjunto de classes, relacionando-as por ligação de herança e/ou composição. O Gerenciador de Esquemas do O_2 é encarregado de fornecer suporte à criação e remoção de classes, tipos e métodos. O critério de remoção visa manter a consistência do esquema, isto é, só permite a remoção de classes quando estas não possuem objetos e classes dependentes, e só permite a remoção de Métodos, quando estes não são herdados ou não estão ativos.

Como áreas de aplicação do modelo podemos citar: Sistemas /CAM, Sistemas urbanos e geográficos, Sistemas de informação editorial, Automoção de escritórios e Aplicações comerciais.

As funções presentes no modelo O_2 são: uso da tecnologia orientada a objeto, união de interfaces de usuário, linguagem de programação e tecnologia de banco de dados e aderência a padrões [DEU91].

2.3.2. Arquitetura do O_2

A arquitetura do núcleo do sistema apresenta-se em de três níveis, como mostra a figura abaixo:

Figura 2.2: Arquitetura O_2 [FIG96]

O nível mais alto é o Gerenciador de Esquema. As funções presentes neste nível são: criação, acesso, modificação e remoção de classes e variáveis globais. Este nível é também responsável pelo controle da consistência do esquema e pela verificação das regras do subtipo em hierarquias de herança.

O nível intermediário é o Gerenciador de Objetos. Responsável pelo gerenciamento de objetos e valores complexos independentemente de sua persistência. Suporta troca de mensagem e configuração cliente/servidor. Implementa também, todas as funções relacionadas à persistência, coleta de lixo e mecanismos de acesso, tais como índices. Por último disponibiliza todas as funções de gerenciamento de transação.

O nível mais baixo controla todo o armazenamento secundário, sendo constituído por um núcleo de gerenciamento de espaço para objetos não estruturados desenvolvido independentemente, e denominado WiSS (subsistema de armazenamento Wisconsin). Este fornece as funções de persistência, gerenciamento de disco e controle de concorrência para registros [BER93] *apud* [FIG96].

2.3.3. Descrição Informal do Modelo

2.3.3.1. Objetos e Valores

Os objetos no Modelo O₂ são representados por pares identificador-valor. Alguns exemplos de objetos seriam:

```
i0: tuple(nome : "Torre Eiffel",
          endereço: i5,
          descrição: "um famoso monumento em Paris",
          feriados: list of("Natal", "Páscoa", "15 de agosto").
          taxa_adm: 25)
i1: tuple(nome:"Paris",
          map:i2,
          hotéis: set(i3,i4))
i5: tuple(cidade:i1,
          rua: "Champ de Mars",
          número: 1).
```


O primeiro objeto possui identificador i_0 e seu valor é uma tupla estruturada. Os valores podem ser atômicos (“Torre Eiffel”), valores estruturados (valores de feriados) ou outro objeto, através de identificadores (i_1, i_2 , etc).

2.3.3.2. Tipos e Classes

Existem duas espécies de organização:

- classes - cujas instâncias são objetos os quais encapsulam dados e procedimentos;
- tipos - cujas instâncias são valores.

Valores não são encapsulados, isto é, sua estrutura é conhecida pelos usuários, e eles podem se manipulados por operadores. A toda classe é associado um tipo, descrevendo a estrutura de suas instâncias. Classes são criadas explicitamente, usando comandos pré-definidos, e são parte da hierarquia de herança. Tipos aparecem como componentes de classes e são construídos recursivamente usando tipos atômicos (*integer, double, string, char, boolean e bits*).

Exemplo de um tipo em O_2 :

```
tuple(nome: string,  
      map: Bitmap,  
      hotéis: set(Hotel)).
```

O tipo anteriormente citado descreve cidades. O atributo hotéis tem como valor um conjunto estruturado. Abaixo seguem algumas definições de classes a título de ilustração. Dentre elas encontra-se a classe Hotel utilizada na definição do exemplo de tipo citado anteriormente.

```
add class Cidade  
  type tuple(nome:string,  
            map: Bitmap,  
            hotéis: set(Hotel))  
add class Monumento  
  type tuple(nome: string,  
            endereço: Endereços,  
            descrição: string,  
            feriados: list(string),  
            taxa_adm: integer)
```

```
add class Endereços
    type tuple (rua: string,
               cidade: Cidade)
add class Hotel
    type tuple (nome:string,
               endereço: Endereços,
               facilidades: list of(string),
               estrelas: integer,
               taxa: float)
add class Restaurante
    type tuple (nome: string,
               endereço: Endereços,
               menus: set (tuple(nome:string, taxa: float))).
```

Uma classe descreve a estrutura e o procedimento de um conjunto de objetos. A parte estrutural de uma classe é um tipo, e a parte procedimental é um conjunto de métodos. As classes são criadas usando comandos de definição de esquema conforme mostrado anteriormente [FIG94].

2.4. Considerações Finais

Neste capítulo, procurou-se elucidar a maneira como cada modelo trabalha, caracterizando-os segundo critérios que permitissem uma comparação entre os mesmos. Além disso, buscou-se analisá-los segundo uma ótica voltada para os conceitos de Esquema de Dados, visto que o trabalho a ser realizado – Integração sob demanda de esquemas de aplicações desenvolvidas utilizando bibliotecas de componentes genéricos - está intimamente ligado a esses conceitos.

A tabela 2.1 apresenta um quadro comparativo entre os modelos analisados.

Tabela 2.1: ORION x O₂ - Adaptado de [CAM92]

Características	ORION	O ₂
Classe/Tipo	Tipo e Classe	Tipo e Classe
Gerenc.de Objetos Complexos	Refer. de Composição	Atributo
Versões	Objetos e Esquema	-
Utilização de Versões	Histórico e Versões de Esquema	-
Evolução de Esquema	Conversão Incremental e Atualização Adiada	-
Aplicações	CAD/CAM Inteligência Artificial Sistemas de informações de escritório Documentos Multimídia	CAD/CAM Sistemas urbanos e geográficos Sistemas de informação editorial Automação de escritórios Aplicações comerciais

Capítulo 03.

**Ferramentas para RAD - *Rapid Application
Development***

3.1. Considerações Iniciais

As ferramentas para desenvolvimento rápido de aplicações (ferramentas para RAD) se apoiam nas técnicas de projetos associadas ao paradigma de Orientação a Objetos e hoje, correspondem à tendência predominante para o desenvolvimento de novos aplicativos.

É bom salientar que tais ferramentas em geral dispõem de bibliotecas de *templates* de objetos (BTO). Estas bibliotecas geralmente diminuem o tempo de desenvolvimento das aplicações por possuírem objetos reutilizáveis e de características genéricas.

Neste capítulo são apresentados aspectos que caracterizam estas ferramentas. Estes aspectos levantados dão o subsídio necessário para se escolher a ferramenta que será utilizada no desenvolvimento do projeto.

3.2. Conceitos de Orientação a Objetos

Nesta seção não são abordados todos os conceitos de orientação a objetos mas somente aqueles que, dentro do contexto de ferramentas para desenvolvimento rápido de aplicações (*Rapid Application Development* - RAD), são relevantes para a identificação de ferramentas deste tipo.

3.2.1. Encapsulamento

Segundo [MON94], dá-se o nome de **Encapsulamento** à propriedade de se implementar dados e procedimentos correlacionados em um mesmo objeto. A idéia por trás deste conceito é a de que a utilização de um sistema orientado a objetos não deve depender de sua implementação interna, e sim de sua interface. Quando implementa-se o encapsulamento deve-se considerar este fato na definição do acesso aos membros de uma classe.

Este conceito representa uma das grandes vantagens em relação à Programação Estruturada, visto que como os dados e os procedimentos se encontram encapsulados na classe, os usuários de objetos de classe ficam livres dos efeitos colaterais decorrentes da modificação dos procedimentos, importando-lhes apenas a interface dos objetos.

Quando se observa os objetos do mundo real pode-se entender intuitivamente o significado disto, pois atributos e ações são inerentes ao próprio objeto. A título de ilustração considere uma classe intitulada Funcionário. Esta classe (figura 3.1), além dos atributos (data de contratação, nome, salário, etc.), possui também métodos implícitos que podem ser executados

por um objeto que a instancie, por exemplo, a consulta, a impressão e a alteração dos atributos da classe. Todos estes métodos, de acordo com o tipo de acesso que seus membros permitem às demais partes do sistema (público, protegido, privado), podem ser chamados por outros objetos do sistema, mas o detalhamento de seu código é irrelevante para quem os chama, cujo interesse está apenas nos resultados proporcionados pelos métodos.

Figura 3.1 : Encapsulamento da classe Funcionário [MON94]

3.2.2. Herança

Segundo [MON94], somente a definição de classes que modelam tipos semelhantes, e a implementação destas não abrangem o potencial mais significativo da Orientação a Objetos. É necessário considerar as relações entre elas, de modo a permitir o compartilhamento de atributos e métodos semelhantes. Esta necessidade é suprida através da **Herança**. Esta propriedade é fruto de um mecanismo de hierarquia de classes, onde uma classe especializada (classe-filha também denominada subclasse) herda propriedades da classe mais geral (classe-pai ou superclasse), a que ela está imediatamente subordinada na hierarquia.

Esta propriedade caracteriza o mecanismo pelo qual se pode compartilhar automaticamente métodos e dados entre diferentes classes e subclasses objetos. A figura 3.2 mostra uma estrutura hierárquica para a modelagem dos dados de uma classe.

Figura 3.2 : Estrutura Hierárquica para Modelagem dos Dados da Classe Veículo -Herança Simples [MON94]

Uma classe derivada deve ter pelo menos uma diferença (atributos ou métodos em relação à sua superclasse). Além disso não se deve repetir os atributos úteis presentes no nível anterior.

Quando uma classe possui características herdadas de duas ou mais superclasses (estas devem estar no mesmo nível), a herança é dita múltipla.

Portanto herança é uma das propriedades responsáveis pela facilidade de reaproveitamento de código na Programação Orientada a Objetos.

A herança no paradigma de orientação a objetos é definida através das estruturas entre as classes. Uma estrutura consiste em uma forma de ligação entre duas classes, onde as classes estão relacionadas pelo fato de uma ser uma especialização da outra (estrutura Generalização/Especialização). A estrutura do tipo Generalização/Especialização (Gen-Esp) sempre define alguma forma de herança.

Uma estrutura do tipo Gen-Esp surge quando uma classe genérica divide-se em subclasses mais especializadas. A figura 3.3 mostra um exemplo deste tipo de estrutura:

Figura 3.3 : Hierarquia com estruturas Generalização/Especialização na classificação de Bens[MON94]

À medida que se sobe na árvore hierárquica, passa-se por níveis progressivos de generalização. Por outro lado, quando se desce, está-se progressivamente especializando.

É bom lembrar que o que permite a criação desta hierarquia é o fato das classes inferiores sempre terem ao menos uma propriedade das classes superiores. E mais, que uma classe inferior deverá ter, no mínimo, uma propriedade que a distinga da classe imediatamente superior à mesma.

A principal vantagem ao se definir estruturas Gen-Esp está na otimização da definição das classes. Com isso métodos e atributos definidos numa classe estarão disponíveis para todas as classes que se encontrarem num nível de maior especialização.

Costuma-se dizer que uma estrutura do tipo Gen-Esp é uma estrutura “É um tipo de”. No exemplo pode-se dizer que móvel é um tipo de bens. Este tipo de estrutura sempre ocorre de classe para classe, ou seja, não é possível que se tenha uma ligação deste tipo entre classe e objeto, ou entre objeto e objeto.

Para finalizar existem linguagens de programação orientadas a objetos que incorporam ao conceito de herança, além do já mencionado anteriormente, a possibilidade de desvincular a

superclasse de todas as suas subclasses. Além disso outras permitem que a funcionalidade seja mantida ou mesmo estendida.

3.2.3. Polimorfismo

Segundo [MON94], o **polimorfismo** é utilizado dentro do paradigma orientado a objetos para indicar a propriedade de se usar o mesmo nome para métodos diferentes, implementados em diferentes níveis de uma hierarquia de classes. Neste caso, para cada classe tem-se um comportamento específico para o método. Então pode-se dizer que polimorfismo é a propriedade de um programa orientado a objetos discernir, dentre métodos homônimos, aquele que deve ser executado. O ponto crucial para este discernimento está no objeto receptor da mensagem .

A título de exemplo pode-se citar a função de impressão dentro de uma hierarquia de classes. Supondo que várias classes possuam um método de impressão (ex.: `imprima()`), porém em cada nível hierárquico pode-se ter código diferente.

Outro ponto importante que deve ser destacado para que haja polimorfismo é a existência de ligação dinâmica, ou seja, a ligação objeto-função membro deve ser feita em tempo de execução.

Este conceito é um dos responsáveis pela facilidade de extensão de um programa orientado a objetos. Dentro de uma hierarquia de classes a extensão do programa só ocorrerá quando uma nova classe acrescentada a hierarquia necessitar de tratamento de exceções naquele nível, e é o caráter polimórfico do programa que o permitirá.

Com o intuito de diminuir o volume de extensões em um programa, uma das estratégias utilizadas na Programação Orientada a Objetos é a de implementar os métodos o mais alto possível dentro da hierarquia de classes.

3.2.4. Identificação de Objetos

A identidade de um objeto é a propriedade pela qual ele se distingue dos demais dentro do domínio do sistema. Esta unicidade é tipicamente implementada utilizando-se de um sistema de geração automática de identificadores de objetos (OID – *Object Identifier*)[CAT94]. O valor de um OID não é visível para o usuário externo, mas é utilizado internamente pelo sistema para identificar cada objeto individualmente e para criar e gerenciar as inter-relações entre os mesmos.

A característica fundamental de um OID é que ele seja imutável. Isto é, o valor de um OID para um objeto particular não pode ser trocado. Isto para preservar a identidade do objeto sendo representado no mundo real. É também desejado que o OID seja utilizado uma única vez. Neste caso se um objeto for removido do sistema, o valor do seu OID não deve ser associado novamente a nenhum outro objeto. Estas restrições trazem como consequência o fato de que o valor do OID não deve depender de nenhum valor dos atributos do objeto, pois eventualmente os valores dos atributos dos objetos podem ser trocados.

3.3. Ferramentas para RAD - *Rapid Application Development*

3.3.1. Histórico

O primeiro programa de desenvolvimento rápido de aplicações foi o *HyperCard*. Este programa provia um ambiente de desenvolvimento onde programadores não especializados (usuários eventuais) rapidamente criavam janelas, objetos, aos quais acrescentavam pedaços de códigos *HyperTalk* [KOL95].

Objetos como botões podiam ser copiados, colados e reutilizados em outras aplicações *HyperCard* com poucas mudanças ou até mesmo nenhuma. Além disso, os desenvolvedores rapidamente acrescentavam comandos externos (XCMDs - eXtern CoMmanDs) ao *HyperCard* com o intuito de aumentar suas funcionalidades. Com estas características, muitos programadores “novatos” desenvolveram suas aplicações pessoais.

Contudo as aplicações escritas com *HyperCard* tinham suas limitações, como o fato de não poderem ser compiladas em aplicações *stand-alone* e, como consequência desta limitação em particular, tornavam-se extremamente lentas.

A primeira ferramenta RAD *for Windows* largamente usada em produção de software foi o *Microsoft Visual Basic* (VB). É possível acrescentar código a cada objeto criado, tais como botões e campos de dados de entrada com o intuito de controlar o comportamento do objeto. Esta ferramenta é centrada numa variação compilada poderosa da linguagem BASIC, que pode ser usada para prover a funcionalidade das aplicações.

Outro ponto importante da ferramenta VB é o suporte às DLLs (*Dynamic Link Libraries*) que permite aos desenvolvedores acrescentarem extensões funcionais à ferramenta. Além disso, os *custom controls* também são incorporados através dos arquivos de extensão VBX ou OCX que são construídos em outras linguagens como por exemplo C++.

Mais recentemente têm surgido ferramentas voltadas para a criação de interfaces gráficas para as aplicações de banco de dados. Fazendo parte deste grupo de ferramentas pode-se citar: *Delphi*, *SQL/Windows*, *PowerBuilder Enterprise 5.0* e *PowerObjects 1.0 da Oracle*.

3.3.2. Características comuns às ferramentas

Segundo [KOL95][FIN96], ferramentas como as supracitadas compõem uma nova classe de ferramentas para apoio ao desenvolvimento (rápido) de sistemas, e possuem em comum as seguintes características:

- Paradigma de Orientação a Objetos e orientação a eventos.
- Objetos primitivos para desenvolvimento de interface.
- Acesso a um Banco de dados relacional para prototipação de aplicações.
- Linguagem de quarta geração (4GL).
- Criação do *front-end* da aplicação como um conjunto de aplicativos clientes do servidor de Banco de dados.
- Criação de objetos reutilizáveis (chamados classes) .
- Utilização da técnica *drag-drop* para diminuir o tempo de desenvolvimento de aplicações de banco de dados.
- Suporte para a geração de toda a interface usuário/computador do sistema baseado na tecnologia de objeto visual.

3.3.3. BTO - Biblioteca de Templates de Objetos

Bibliotecas de *templates* de objetos podem ser definidas como um conjunto de classes básicas reutilizáveis que podem ser independentes, isto é, constituem um conjunto de objetos “desamarrados” ou podem ser caracterizadas dentro do contexto de um *framework* mantendo uma relação de interdependência entre as classes, com o intuito de formar a base para uma aplicação.

Dentro de BTO são aplicáveis pelo menos dois tipos de conceitos para permitir a reutilização de classes, a saber: especialização de classes e objetos padrão.

3.3.3.1. Especialização de Classes

O conceito de classes de objeto incorpora os recursos para suportar herança e encapsulamento, que estão entre as principais características dos ambientes de orientação a

objetos. Através da especialização de classes de objetos padrão da BTO, pode-se acrescentar funcionalidade adicional para configurar e personalizar objetos e classes que já tem sua funcionalidade básica na BTO.

3.3.3.2. Objetos Padrão

Esse conceito permite que uma classe já definida seja instanciada em objeto que atua como “molde” para ser reproduzido diversas vezes na aplicação, que a seguir são configurados. Esse mecanismo, adotado nas ferramentas avaliadas principalmente sobre objetos visuais, permite propagar às cópias dos objetos “molde” as alterações sofridas pelo mesmo, e definir estruturas e comportamento padronizados. A criação de instâncias pode manter um vínculo com as classes originais ou não.

3.3.3.3. Categorias Existentes

Hoje em dia pode-se verificar que cada ferramenta para RAD possuem uma BTO. Segundo [NAN96], uma BTO pode ser classificada em três categorias:

- **Classes de Propósito Geral**

São usualmente classes onde toda a funcionalidade necessária às mesmas se encontra dentro da classe ancestral de mais alto nível, que inclui qualquer coisa que uma aplicação possa precisar. Isto tipicamente gera ancestrais “gordos” sendo definida muitas das vezes funcionalidade desnecessária para a classe em questão. A funcionalidade do objeto é estabelecida através de variáveis de domínio público dentro da hierarquia de classes no evento *Constructor* do objeto. Por exemplo, seja o objeto O1 definido da seguinte maneira, segundo [NAN96].

```
// Superclasse: O1
// Variáveis instanciadas
Boolean ib_ordena
Boolean ib_filtra
Boolean ib_seleciona
// Funções
    of_ordena()
    of_filtra()
    of_seleciona()
```



```
// Evento Constructor do Objeto  
this.ib_ordena=true  
this.ib_filtra=true  
this.ib_seleciona=true
```

Quando se cria um objeto O2 herdado do O1, automaticamente todas as funções definidas para o ancestral estão disponíveis para o objeto recém-criado. Se um objeto é criado herdado do objeto O1 não é possível escolher o que se quer herdar. Supondo que seja preciso que se crie vários objetos herdados de O1, então a aplicação terá um *overhead* muito grande, visto que a grande maioria dos atributos e métodos herdados não serão utilizados.

- **Generalização Aninhada de classes**

Esta categoria permite que se eliminem os ancestrais “gordos” presentes na categoria anterior pois ela trata as classes como classes de propósito geral. Neste caso as classes ancestrais básicas são criadas apenas com a funcionalidade mais genérica e básica possível e as classes serão progressivamente herdadas até que se tenha todas as combinações necessárias para a aplicação.

Figura 3.4 : Generalização/Especialização

Primeiro criou-se o objeto mais genérico O1 com as funções básicas e então criou-se objetos individuais com funcionalidades específicas, como por exemplo o objeto O2 e o objeto O3 herdados de O1 mostrado na figura 3.4 . Se for necessário criar um objeto que contemple as características do objeto O2 e O3, deverá ser criado mais um nível na hierarquia. Neste ponto é bom salientar que, isto representa o conceito de herança múltipla citado neste capítulo e será mostrado na figura 3.5.

Figura 3.5 : Generalização/Especialização após a inclusão de uma nova classe

- **Classes orientadas a serviços**

Esta categoria é que mais se aproxima dos conceitos de orientação a objetos e sua base está todo no encapsulamento. Esta categoria encapsula funcionalidade em classes personalizadas reutilizáveis chamadas métodos, que executam o trabalho destinado a elas. Com isso, a funcionalidade não está vinculada ao requisitante dos métodos, e estes não são instanciados até que alguém os requisite. Como as classes são baseadas em métodos, o exemplo anterior da classe O1 pode ser representado pela figura 3.6 :

Figura 3.6 : Objeto 01

Além disso a título de curiosidade serão mostrados também os objetos O2 e O3 nas figuras 3.7 e 3.8 .

Figura 3.7 : Objeto 02

Figura 3.8 : Objeto 03

Para encerrar, suponha-se que seja necessário um objeto O4 que tenha características de O2 e O3. A solução consiste em criar um objeto e incorporar os métodos presentes em O2 e O3 ao mesmo, como mostrado na figura 3.9 .

Figura 3.9 : Objeto 04

Portanto conforme mostrado nos exemplos, pode-se perceber que de posse de métodos dentro do sistema, a criação de uma classe desejada nada mais é que selecionar os métodos desejados e incorporá-los à classe.

3.4. Conclusão

Neste capítulo buscou-se caracterizar as ferramentas para RAD - *Rapid Application Development* - apresentando conceitos de orientação a objetos aplicados às mesmas e às diversas categorias de BTO que podem estar presentes numa ferramenta desse tipo.

Essas características foram utilizadas para avaliar as ferramentas disponíveis, com o intuito de determinar qual será objeto de uso para o projeto em questão. O capítulo seguinte apresenta o resultado dessa avaliação.

Capítulo 04.

Ferramentas para RAD Avaliadas

4.1. Considerações Iniciais

Neste capítulo são abordadas ferramentas para RAD sob a ótica dos conceitos de orientação a objetos apresentados no capítulo anterior. Além disso são tratadas as características presentes neste tipo de ferramenta e finalmente qual a categoria de bibliotecas de *templates* de objetos (BTO) que é suportada por cada uma das ferramentas para RAD avaliadas.

No Grupo de Base de Dados e Imagens do ICMC/USP estão disponíveis o *PowerObjects* da Oracle e o *PowerBuilder 5.0* da PowerSoft. O restante deste capítulo apresenta uma avaliação destas ferramentas, segundo os critérios mencionados anteriormente.

4.2. PowerObjects 1.0 da Oracle

PowerObjects é uma ferramenta visual de desenvolvimento orientada a objeto para a criação de aplicações *workgroup* de acesso a base de dados. Sua característica principal é a criação de aplicações *desktops* no ambiente cliente/servidor. Possui drivers nativos de conexão com os principais gerenciadores de dados, por exemplo: *Oracle 7*, *Microsoft SQL Servers*. Oferece também recursos para a migração de plataformas. Pode-se construir uma aplicação no *Macintosh* e movê-la para uma plataforma *Windows* ou vice-versa. Sua versão mais recente *PowerObjects 2.0* suporta controles *OCXs* [REED96][ORA96A][ORA96B].

Atualmente a *Oracle* tem investido num novo ambiente de desenvolvimento para aplicações de missão crítica ou orientadas à *Internet* e aplicações de múltiplas camadas (*n-tier*). Esse ambiente congrega uma ferramenta *CASE (Designer/2000)* e uma ferramenta de desenvolvimento (*Developer/2000*). Essa ferramenta permite o particionamento da aplicação, portabilidade entre ambientes *Windows*, *Macintosh* e *UNIX (MOTIF e modo caracter)* e também o desenvolvimento em equipe [ORA96b].

Antes de abordar as características de orientação a objetos, vale a pena mostrar a hierarquia de objetos da ferramenta *PowerObjects 1.0* sem a preocupação de definir todos os objetos presentes na mesma.

4.2.1. Arquitetura de Objetos

Figura 4.1: Hierarquia de Objetos Primitivos [FIN96]

Pode-se notar três classes de objetos no nível mais alto. No modo de desenvolvimento pode-se editar ou sob-escrever propriedades e métodos de um dado objeto tão bem como acrescentar propriedades e métodos. No modo de execução pode-se modificar apenas propriedades do objeto (cor, tamanho, forma , etc).

4.2.1.1. Objetos de Aplicação

Figura 4.2: Objetos de Aplicação – Adaptado de [FIN96]

Como pode-se notar, estes objetos de aplicação talvez fossem melhor caracterizados como objetos de interface.

4.2.1.2. Objetos de Biblioteca

Figura 4.3: Biblioteca de Objetos [FIN96]

Estes objetos representam objetos criados pelos desenvolvedores com o intuito de permitir sua reutilização em outras aplicações. A partir deles, pode-se criar desde um simples *bitmap* (figura) que represente o logotipo da empresa do próprio desenvolvedor até um *form* (formulário) que represente um ficha completa de cadastramento de cliente.

4.2.1.3. Objetos de Sessão

Estes objetos são responsáveis pela “linkedição” do *front-end* da aplicação com o gerenciador de dados (*back-end*) a ser acessado.

Figura 4.4: Objetos de Sessão [FIN96]

Como foi dito no início desta seção, não se tem aqui o objetivo de descrever detalhadamente cada um dos objetos presentes na ferramenta. Por isso foi apenas mostrada a hierarquia de objetos dentro da mesma.

4.2.2. Aplicação de Conceitos Orientados a Objetos

- **Encapsulamento**

Como foi dito no capítulo anterior, encapsulamento é a habilidade de modificar propriedades e métodos dentro de um objeto sem que tais mudanças afetem os demais objetos que interagem com ele. Por exemplo define-se um botão e este, por se tratar de um objeto padrão de interface, poderá ter suas propriedades visuais e/ou comportamentais modificadas de acordo com a necessidade da aplicação.

Contudo, como o conceito de encapsulamento é suportado pela ferramenta, as alterações sofridas pelo objeto no que tange aos aspectos visuais e/ou comportamentais não afetarão a aplicação [KOL95]. Pode-se mostrar isso da seguinte maneira. Suponha que o desenvolvedor venha a criar um botão e que ele associe a este objeto (botão) uma ação de ordenação ao ser pressionado.

Futuramente o desenvolvedor pode vir a modificar o tamanho, cor do objeto criado (aspectos visuais) e, mais que isso, ele pode modificar o algoritmo de ordenação utilizado pelo objeto (aspecto comportamental). Diante destas mudanças o usuário da aplicação não sofrerá nenhum impacto, pois a maneira como o objeto é “visto” não foi modificada; isto é, a interface do método continua a mesma.

- **Herança**

A própria arquitetura de objetos mostrada no início deste capítulo exemplifica o conceito de herança do mesmo modo como é mostrado no capítulo anterior.

- **Polimorfismo**

Segundo [KOL95], polimorfismo é a habilidade de criar uma subclasse que herde alguns atributos da classe pai. Além disso, a subclasse pode precisar de mais funcionalidades que as existentes na classe pai. De acordo com [KOL95], esta necessidade é atendida pelo polimorfismo presente na ferramenta. Levando em conta este ponto de vista, pode-se afirmar que a ferramenta não utiliza o conceito de polimorfismo mostrado no capítulo anterior.

- **Identificação de objetos**

A ferramenta utiliza um mecanismo de identificação de objetos que respeita a propriedade de identificação de objetos mostrada no capítulo anterior.

4.2.3. BTO

Na versão disponível do *PowerObjects* não foi possível identificar uma biblioteca que tivesse as características de uma BTO. Encontrou-se uma biblioteca de objetos que estava intimamente ligada a um exemplo de uma aplicação desenvolvida com a ferramenta mas não representava uma BTO.

Apesar da ferramenta em questão não possuir uma BTO, pode-se constatar que a mesma com certeza suporta o desenvolvimento de uma. Quanto às categorias identificadas para uma BTO (citadas no capítulo anterior), pode-se dizer que seria possível o desenvolvimento de uma biblioteca que se enquadrasse na categoria de classes de propósito geral bem como, na de Generalização Aninhada de Classes.

4.3. PowerBuilder 5.0 da PowerSoft

PowerBuilder é uma ferramenta visual de desenvolvimento orientada a objeto para a criação de aplicações multi-plataforma de acesso a bases de dados. Avaliada em sua versão 5.0, está disponível nas seguintes versões: *PowerBuilder Desktop*, *PowerBuilder Professional* e *PowerBuilder Enterprise*. A primeira alia a velocidade de desenvolvimento da linguagem de 4GL orientada a objetos à tecnologia de acesso, apresentação e atualização de dados. A ferramenta é acompanhada do *Sybase SQL AnyWhere*, um banco de dados utilizado em pequenas aplicações, que suporta os principais conceitos usados em banco de dados *desktop* do mercado. A segunda configuração é voltada para a construção de aplicações de médio a grande porte, permite conexão com qualquer banco de dados através do padrão ODBC (*Open Database Connectivity*) e permite também o trabalho em equipe com recursos de *check-in/check-out*. Dispõe do *PowerSoft Foundation Class* (PFC), que permite acelerar o desenvolvimento de aplicações cliente/servidor. Por último, a terceira configuração vem suprir a necessidade das aplicações de missão crítica ou orientadas à *Internet*. Essa versão oferece desde *drivers* nativos para acesso aos principais banco de dados do mercado, passando pelo suporte a aplicações em múltiplas camadas (*multi-tier*) até o desenvolvimento de aplicações para WEB [SYB96e][DEY96].

4.3.1. Arquitetura de Objetos

Nesta seção tem-se uma visão geral da arquitetura de objetos da ferramenta em questão sem a preocupação de esgotar toda hierarquia de objetos presente na ferramenta. Na figura 4.5 tem-se a hierarquia de objetos no nível mais alto. Todos os objetos presentes na ferramenta derivam de uma classe abstrata intitulada *PowerObject* [REE96]. Esta classe tem um número de descendentes que constituem objetos padrão disponíveis para o projetista personalizar seus próprios objetos. As figuras 4.6 e 4.7 apresentam os objetos padrão do tipo *GraphicObject* e *WindoObject* respectivamente.

Figura 4.5: Hierarquia de Objetos – Adaptado de [NAN96]

Figura 4.6: Objetos Visuais – Adaptado de [NAN96]

Figura 4.7: Objetos de Janela – Adaptado de [NAN96]

4.3.2. Aplicação de Conceitos Orientados a Objetos

4.3.2.1. Encapsulamento

Um objeto deve ser capaz de se isolar de interferência do ambiente. Encapsulamento é o processo pelo qual os dados e estruturas internas são protegidas e escondidas do ambiente externo. Pode-se pensar num objeto como uma caixa preta. Precisa-se saber apenas três coisas sobre ele:

- O que ele propõe fazer
- Que informação (se necessária) ele requer para fazer seu trabalho
- Como ele comunicará os resultados de suas ações para o meio

A ferramenta implementa o encapsulamento da mesma maneira que a ferramenta *PowerObjects 1.0*.

4.3.2.2. Herança

A própria hierarquia de objetos mostrada no início deste capítulo exemplifica o conceito de herança presente na ferramenta. Nem todos os objetos dentro da hierarquia suportam este conceito. Apenas objetos de acesso à base de dados (*Data Window*) não suportam herança.

4.3.2.3. Polimorfismo

Esta característica é bastante forte na ferramenta. Funções com o mesmo nome se comportam de maneira diferente, dependendo do objeto referenciado e do número de argumentos da função.

4.3.2.4. Identificação de objetos

A ferramenta utiliza um mecanismo de identificação de objetos que respeita a propriedade de identificação de objetos.

Neste ponto já se pode abordar a BTO disponível pela ferramenta. Na versão 5.0 do *PowerBuilder* aqui referenciada existe uma BTO chamada *PowerBuilder Foundation Class Library* (PFC). Esta biblioteca será objeto de estudo mais detalhado na próxima seção.

4.3.3. PowerBuilder Foundation Class Library - PFC

PFC é um conjunto de classes reutilizáveis e extensíveis que trabalham em cooperação para servir de base para a construção de aplicações *PowerBuilder* orientadas a objetos e para o desenvolvimento de bibliotecas de classes [GRE96].

Anteriormente foram vistos os conceitos de orientação a objetos suportados pela ferramenta. Na seção seguinte discute-se como o encapsulamento e polimorfismo são implementados, especificados e tratados pela ferramenta, também alguns exemplos são apresentados.

4.3.3.1. Encapsulamento

PFC implementa encapsulamento em seus objetos através da definição de funções e variáveis instanciadas que poderão ser protegidas ou públicas dependendo do nível de acesso externo que os outros objetos terão.

4.3.3.2. Polimorfismo

PFC implementa este conceito permitindo que funções com o mesmo nome se comportem diferentemente. As funções podem residir dentro de um objeto, numa hierarquia de herança ou até mesmo em vários objetos.

Segundo [SYB96d], PFC usa dois tipos de polimorfismo: polimorfismo operacional e polimorfismo inclusional. Essa divisão é utilizada de maneira particular dentro da ferramenta *PowerBuilder*.

Polimorfismo Operacional

Neste tipo de polimorfismo dois objetos separados definem uma função com o mesmo nome.

Figura 4.8: Polimorfismo Operacional – Adaptado de [SYB96d]

Pela figura 4.8 pode-se perceber que tanto o objeto 1 quanto o objeto 2 possuem uma função `f_ordena()`. É bom salientar que esta característica não representa polimorfismo de fato, como definido no capítulo anterior.

Polimorfismo Inclusional

Neste tipo de polimorfismo vários objetos dentro de uma hierarquia de objetos definem uma função com o mesmo nome mas com argumentos diferentes. É bom salientar que o fato de possuir argumentos diferentes se aplica apenas para funções definidas no mesmo objeto.

PowerBuilder suporta tanto *function overriding* quanto *function overloading*. No processo de ignorar a função do nível mais alto (*function overriding*), a função presente no descendente tem os mesmos argumentos ou os mesmos tipos de dados. No processo de sobrecarregar a função (*function overloading*), a função descendente (ou uma função com o mesmo nome no mesmo objeto) tem argumentos diferentes ou argumentos de tipos de dados diferentes.

Figura 4.9: Polimorfismo Inclusional– Adaptado de [SYB96d]

4.4. PowerBuilder X PowerObjects

Nesta seção são apresentadas duas tabelas comparativas abordando conceitos de orientação a objetos e categorias de BTO.

Tabela 4.1 - Conceitos de Orientação a Objeto

Ferramentas	Conceitos de Orientação a Objeto				
	Encapsulamento	Polimorfismo	Her ¹	Her ²	Identificação de Objetos
PowerObjects	☺	☹	☺	☹	☺
PowerBuilder	☺	☺	☺	☺	☺

A tabela 4.1 mostra um paralelo entre as duas ferramentas RAD avaliadas neste capítulo sob o ponto de vista dos conceitos de orientação a objeto. Pode-se perceber que ambas a ferramenta trazem o emprego dos conceitos de orientação a objeto dentro de seu ambiente de desenvolvimento. Vale ressaltar que a ferramenta *PowerBuilder*, além de incorporar todos os conceitos, traz uma inovação no que tange ao conceito de herança, que permite uma maior flexibilidade no desenvolvimento de aplicações.

Tabela 4.2 - Categorias de BTO

Ferramentas	Categorias de BTO			BTO Disponível
	Classes de Propósito Geral	Generalização Aninhada de Classes	Classes Orientadas a Serviço	
PowerObjects	☺	☺	☹	ausente
PowerBuilder	☺	☺	☺	☺

¹ Suporte a herança no contexto tradicional apresentado no capítulo três.

² Suporte a herança com a possibilidade de ignorar ou estender a funcionalidade existente na super classe.

A tabela 4.2 mostra-se um paralelo entre as duas ferramentas RAD avaliadas neste capítulo sob o ponto de vista das categorias de BTO presentes nas mesmas. Vale ressaltar que a ferramenta *PowerBuilder*, incorpora todas as categorias, com a ressalva para a possibilidade de desenvolver bibliotecas dentro da categoria de classes orientadas a serviços, o que representa uma diferencial para o desenvolvimento de aplicações Cliente/Servidor.

4.5. Conclusão

De acordo com o resumo apresentado anteriormente nas duas tabelas, resolveu-se optar pelo *PowerBuilder* para o desenvolvimento do restante do projeto, visto que o mesmo além de incorporar todos os conceitos de orientação a objetos também traz uma BTO que seria o ponto de partida para o desenvolvimento do trabalho proposto.

Porém após a avaliação das ferramentas e mesmo depois de ter escolhido aquela que seria utilizada para o desenvolvimento do projeto, notou-se que as bibliotecas de *templates* de objetos (BTO) presentes nas mesmas contêm basicamente objetos de interface e portanto, não são adequadas para integrar esquemas de banco de dados.

No próximo capítulo será mostrada uma estrutura de generalização para um problema particular, que deu subsídios para generalizar estruturas de esquemas que visam ser reutilizadas para a solução de outros problemas.

Capítulo 05.

Integração de Esquemas utilizando uma Estrutura Genérica

5.1. Considerações Iniciais

A possibilidade de coexistirem sistemas operados de maneira independente e a necessidade que estes sistemas troquem informações em algum momento fizeram com que várias formas de integração de esquemas fossem propostas. Dentre estas formas está a própria criação de sistemas de banco de dados heterogêneos capazes de acessar banco de dados já existentes, podendo estes estarem em ambientes computacionais diferentes, com modelos de dados distintos, linguagens e representações diversas, muita das vezes para informações similares[LIT90][PER91] *apud* [UCH94]. Dentro deste contexto, dependendo do tipo de integração existente entre os componentes dos sistemas de banco de dados heterogêneos, eles podem ser classificados em sistemas federativos com acoplamento fraco ou com acoplamento forte [RAM91][SHE90] *apud* [UCH94]. O primeiro provê apenas uma interface de negociação entre os diversos componentes da federação. Além disso, o usuário é responsável por compreender toda a semântica do objeto do esquema de exportação e resolver heterogeneidades semânticas e de sistemas de gerenciamento de banco de dados. O segundo tipo (acoplamento forte) garante a transparência de acesso aos diversos componentes da federação. Esta transparência só é conseguida pela tradução dos diferentes esquemas dos componentes para um modelo de dados comum e integrado, compondo um esquema global. Todas as transações que envolvem mais de um banco de dados são definidas em termos do esquema global.

Outra abordagem considera que integração de esquema necessita de informações sobre a semântica dos esquemas a serem integrados. Frequentemente, estas informações não estão disponíveis diretamente no sistema de banco de dados. Neste caso é preciso que exista alguma fase no processo de integração que obtenha estas informações. A obtenção de informações pode ser vista como uma atividade da área da engenharia reversa [DAV87][CH194] *apud* [BEH97]. Contudo, este processo não pode ser completamente automatizado. Situações que apresentem ambigüidades podem inviabilizar a fase de obtenção de informações sobre o esquema e precisar da interferência do usuário.

Até agora, não se viu nenhuma abordagem que investigasse os vários esquemas a serem integrados, com o objetivo de identificar estruturas comuns que, em última análise, pudessem ser utilizadas para efetuar, de fato, trocas de informações de maneira automática entre sistemas desenvolvidos independentemente.

Este capítulo busca descrever o processo de integração de esquemas utilizando-se uma estrutura padrão. Para permitir a integração de esquemas de sistemas desenvolvidos independentemente, além destes terem sido desenvolvidos utilizando-se de uma estrutura padrão, é preciso que se estabeleça um conjunto de propriedades e regras a ser utilizado durante o processo de integração.

5.2. O Processo de Integração

Introduz-se aqui a noção geral do processo de integração de esquema de dados sob demanda. O ponto de partida são dois esquemas desenvolvidos de maneira independente. O objetivo é a identificação de uma estrutura comum a estes esquemas que permita a troca de informações contidas nos mesmos de maneira automática. Partindo deste pressuposto introduz conceito de regras de integração. Uma regra de integração permite que a informação, ou parte dela, presente em um sistema, possa ser “exportada” para outro atendendo a determinadas restrições. O processo pode ser dividido nas seguintes fases: identificação da estrutura, geração de regras, geração de propriedades, aplicação das regras, extração da informações (preservando propriedades) que se deseja exportar e a “exportação” propriamente dita.

Figura 5.1: Processo de Integração

A figura 5.1 indica que as fases poderiam ser seqüencializadas da seguinte maneira. Primeiramente identifica-se a estrutura genérica presente nos dois sistemas. Esta identificação será feita com a intervenção do usuário especializado com o intuito de não inviabilizar o restante do processo. Na verdade poderia se ter um “agente” que identificasse semelhança entre os dois esquemas a serem integrados utilizando-se de alguma documentação a respeito do mesmo. Após a identificação da estrutura, precisa-se definir um conjunto de regras capazes de preservar a validade da informação a ser trocada ente os dois sistemas. A partir deste momento, com a estrutura genérica identificada e o conjunto de regras estabelecido, inicia-se o processo de integração propriamente dito, onde ocorre a extração da informação do sistema origem utilizando a estrutura genérica identificada. Esta extração de informação utilizará uma base de dados que contemple a estrutura genérica que receberá a informação. Finalmente após extrair e armazenar a informação na base genérica, exportar-se-á a informação para o sistema destino.

Figura 5.2 : Modelagem simplificada de uma cooperativa prestadora de serviços médicos

São apresentados aqui dois exemplos de sistemas desenvolvidos independentemente, e que serão utilizados mais tarde para a demonstração de todo o processo de integração. O esquema da figura 5.2 mostra um exemplo simplificado da modelagem de pessoa no contexto de uma cooperativa prestadora de serviços médicos (S1).

Primeiramente dar-se-á um breve explicação sobre a notação utilizada, extraída basicamente da ferramenta *S_Designor* [SYB96a] [SYB96b][SYB96c]. Cada retângulo da figura representa uma classe no modelo. O semicírculo entre as várias classes representa uma especialização da classe situada acima do mesmo. As classes situadas abaixo do semicírculo são as subclasses. Quando um semicírculo estiver vazio o critério de especialização será sobreponível, caso contrário (a existência de um X dentro do semicírculo) o critério será de disjunção. Para ficar mais claro, seja o seguinte exemplo: a classe *pessoa_pri* (pessoa genérica) foi especializada em *pesfis_esp* (pessoa física) e *pesjur_esp* (pessoa jurídica) com a característica de não sobreposição (semicírculo preenchido por um X), isto é, uma dada instância de *pessoa_pri* não pode figurar nos dois lados da especialização (ser *pesfis_esp* e *pesjur_esp*). Esta propriedade confere às classes especializáveis um *link* de integridade com a classe especializada. Este *link* é mantido pelo identificador da classe.

Tem-se também na figura 5.2 junto a cada semicírculo um # seguido por um número que determina qual é o atributo da classe genérica usado para estabelecer o critério de especialização da classe. Essa notação foi feita neste trabalho estendendo a notação da ferramenta *S-Designor*. Tomando a classe *pessoa_pri* como exemplo novamente tem-se que o atributo **bl_pessoa_fisica** é utilizado para mapear a especialização *pesfis_esp/pesjur_esp*. As outras especializações *prest_pri* (prestador da cooperativa médica) e *posate_pri* (posto de atendimento) são determinadas pelos atributos **bl_prestadora** e **bl_posto** respectivamente.

Vale ressaltar que o critério de determinação de uma especialização não se restringe a um único atributo da classe. Isto pode ser comprovado pela especialização de *pesfis_esp* (pessoa física) em *benefc_pri* (beneficiário) *medico_esp* (médico), que utiliza-se de dois atributos (**bl_beneficiario**, **bl_medico**) para determinar a especialização. Para este exemplo a especialização é sobreponível podendo se ter perfeitamente uma pessoa física que seja beneficiário e médico ao mesmo tempo.

Por último a linha presente no *link* entre classe e subclasse denota se a especialização tem participação total ou parcial. Define-se participação total como o seguinte: dada uma instância da classe no nível i a obrigatoriedade dessa instância figurar em pelo menos uma das

sub-classes da classe do nível i. Caso contrário, tem-se participação parcial. Na figura utilizou-se traços duplos para denotar participação total, e traços simples para denotar participação parcial. Essa notação também representa uma extensão à notação da ferramenta *S-Designor*.

A modelagem exemplificada tem o intuito de permitir que se tenha um cadastro de todas as informações relativas a uma prestadora de serviços médicos, isto é, cadastro de beneficiários, médicos vinculados a prestadora, empresas conveniadas e assim por diante.

O esquema da figura 5.3 mostra outro exemplo de modelagem de pessoa, no contexto de uma instituição de ensino (S2).

Figura 5.3 : Modelagem simplificada de uma instituição de ensino

Essa modelagem tem o intuito de permitir que se tenha um cadastro de todas as informações relativas às pessoas dentro de uma instituição de ensino. Nesse exemplo simplificado, têm-se professores, estudantes de graduação, estudantes de pós-graduação e fornecedores.

Partindo destes dois sistemas, desenvolvidos de forma isolada e independente, precisa-se que o esquema S1, mostrado na figura 5.2, troque informações com o esquema S2 da figura 5.3. Em primeira análise estas trocas deveriam ser feitas de forma manual ou semi-automática com a intervenção do usuário.

Para que seja possível a integração automática é necessário identificar similaridades entre as estruturas dos esquemas (S1 e S2). Para identificar essas similaridades é preciso analisar a estrutura não apenas no aspecto de atributos de cada classe, mas também a forma como as classes se relacionam (*links* de herança). Analisando os dois sistemas apresentados pôde-se chegar a uma estrutura comum presente nos mesmos. É bom salientar que este processo (obtenção da estrutura genérica) deve ser feito manualmente pelo desenvolvedor da Biblioteca de *Templates*. A seguir é mostrado na figura 5.4 esta estrutura genérica. Esta estrutura representa uma abstração de generalização nos modelos de dados semânticos.

Figura 5.4 : Estrutura Genérica – Pessoa

A estrutura foi concebida através de um processo de análise (não-automático) dos dois sistemas permitindo identificar a classe *pessoa_pri* em comum e, além disso, um nível de especialização da mesma (*pesfis_esp* / *pesjur_esp*). O conjunto de atributos mostrado em cada classe da estrutura corresponde à operação de intersecção entre o conjunto de atributos das respectivas classes nos dois sistemas utilizados como exemplo.

Após identificada a estrutura genérica que está presente nos dois sistemas distintos, pretende-se definir um *framework* formal para integração de esquemas apoiado na estrutura supracitada. Esse *framework* formal consiste em um conjunto de propriedades de esquemas, chamados invariantes, e em um conjunto de regras, com o intuito de permitir a troca de informações entre dois sistemas desenvolvidos a partir da mesma estrutura.

Neste ponto é necessário delimitar o contexto da validade do processo de integração proposto, para determinar as premissas necessárias para que o mesmo seja factível.

É preciso descartar toda e qualquer informação semântica do ponto de vista do negócio a que se aplicam os sistemas a serem integrados. Tais informações devem restringir-se unicamente à representação sintática das propriedades dos elementos utilizados.

Os exemplos apresentados são centrados em uma única abstração comum que é a Abstração de Generalização. Este trabalho explora apenas esse conceito, como um passo inicial para o processo de integração de esquemas. Essa Abstração foi escolhida por ser a mais estudada, e ao mesmo tempo aquela que apresenta a maior semelhança entre os diversos modelos e ferramentas disponíveis, tanto na literatura quanto comercialmente.

Na próxima seção apresentam-se as invariantes e regras necessárias para a integração de esquema sob demanda para estruturas originadas de Abstrações de Generalização.

5.3. Invariantes de Integração de Esquemas

Primeiramente, buscou-se identificar a essência da estrutura a ser estudada, com o intuito de determinar invariantes para a mesma. Como a estrutura identificada representa uma abstração de generalização no modelo utilizado, a hierarquia terá apenas uma classe no nível zero. Com isso foi possível identificar 3 invariantes de esquemas, apresentadas a seguir.

5.3.1. Invariante de hierarquia de classes

De acordo com a estrutura identificada, a hierarquia de classes pode ser vista como uma dígrafo acíclico conexo, possuindo apenas uma raiz.

5.3.2. Invariante de identificador

Todo objeto da classe C_i do nível i será identificado pelo OID da sua superclasse pertencente ao nível $i - 1$. Com isso, pode-se afirmar que a classe única presente no nível 0, que representa a origem da estrutura (rede), conterá todos os identificadores existentes na estrutura em questão.

5.3.3. Invariante de herança

Os objetos de uma classe herdam o identificador de sua superclasse. Cumpre lembrar que esta invariante é que permitirá de fato que características de um objeto da classe C_j pertencente ao esquema S_i seja herdado por um objeto da classe C_j num esquema S_j ambos construídos a partir de um esquema S_0 .

5.4. Regras de integração de Esquemas

A troca de dados entre esquemas deve, num dado estado, preservar todas as invariantes das estruturas de dados envolvidas. Contudo faz-se necessário determinar regras para que aplicações desenvolvidas de maneira independente, utilizando um mesmo Esquema de *Templates* de Objetos, consigam trocar informações de maneira automática. Com isso foi criado um conjunto de quatorze regras dividido em duas categorias: regras de manipulação da hierarquia da estrutura ou do esquema, e regras de atributos.

Para todas as regras vale a seguinte premissa. Sejam duas estruturas E_i , E_j originadas da mesma estrutura E_0 . O processo de troca de informações ocorre de E_i para E_j e essa troca se restringe às classes presentes na estrutura comum E_0 .

Nos exemplos apresentados pode-se fazer a seguinte analogia. A estrutura Pessoa Genérica identificada como estrutura comum representa a estrutura E_0 . No exemplo S1 a parte do esquema que mapeia a estrutura genérica (E_0) pelo atributo “#1 bl_pessoa_fisica” pode ser tomada como a estrutura $E1$. E por último, no esquema S2 a parte do mesmo que mapeia a estrutura genérica (E_0) pelo atributo “#1 bl_pessoa_fisica” será a estrutura $E2$. Feitas estas

pode-se mostrar o conjunto de regras identificadas para o processo de integração utilizando uma estrutura que representa uma Abstração de Generalização.

5.4.1. Regras de manipulação da hierarquia da estrutura

1ª Regra: se as ocorrências da abstração de generalização da estrutura E_i e da estrutura E_j forem ambas disjuntas, a troca de informações ocorrerá de maneira completa. De acordo com os sistemas mostrados anteriormente, considere E_i sendo $E1$ e E_j sendo $E2$, pode-se ter todas as informações contidas na estrutura $E1$ exportadas para a estrutura $E2$. Analisando a estrutura e os dois exemplos, pôde-se perceber que as mesmas ($E1$ e $E2$) são disjuntas no contexto da especialização da estrutura (pesfis_esp/ pesjur-esp). Portanto a troca de informações entre os sistemas ocorreria de maneira completa.

2ª Regra: se as ocorrências da abstração de generalização da estrutura E_i forem disjuntas e as ocorrências da abstração de generalização da estrutura E_j forem sobreponíveis, a troca de informações de E_i para E_j ocorrerá em sua totalidade. Novamente utilizando o exemplo apresentado, se as ocorrências da abstração de generalização da estrutura $E2$ (E_j) fossem sobreponíveis, a troca poderia ocorrer visto que na estrutura $E1$ (E_i) as ocorrências são disjuntas.

3ª Regra: se as ocorrências da abstração de generalização da estrutura E_i forem sobreponíveis e as ocorrências da abstração de generalização da estrutura E_j forem disjuntas, a troca de informações entre as estruturas obedecerá o critério de disjunção da estrutura E_j . De acordo com os sistemas mostrados anteriormente, considere E_i sendo $E1$ e E_j sendo $E2$, pode-se ter apenas as informações contidas na estrutura $E1$ que estivessem presentes somente num dos lados da especialização de pessoa (física ou jurídica) exportadas para a estrutura $E2$. Isto porque, as ocorrências da abstração de generalização da estrutura $E2$ sendo disjuntas, não permitiriam sobreposição.

4ª Regra: se as ocorrências da abstração de generalização da estrutura E_i e da estrutura E_j forem sobreponíveis, a troca de informações de E_i e E_j ocorrerá em sua totalidade.

5ª Regra: se conjunto de especialização da abstração de generalização da estrutura E_i e da estrutura E_j tiverem ambas participação total, a troca de informações ocorrerá de maneira completa. De acordo com os sistemas mostrados anteriormente, considere E_i sendo $E1$ e E_j sendo $E2$, pode-se ter todas as informações contidas na estrutura $E1$ exportadas para a estrutura $E2$. Analisando a estrutura e os dois exemplos, pôde-se perceber que o conjunto de especialização (pesfis_esp/ pesjur-esp) das estruturas $E1$ e $E2$ tem participação total. Portanto a troca de informações entre os sistemas ocorreria de maneira completa.

6ª Regra: se o conjunto de especialização da abstração de generalização da estrutura E_i tiver participação total e o conjunto de especialização da abstração de generalização da estrutura E_j tiver participação parcial, a troca de informações de E_i para E_j ocorrerá em sua totalidade.

7ª Regra: se o conjunto de especialização da abstração de generalização da estrutura E_i tiver participação parcial e o conjunto de especialização da abstração de generalização da estrutura E_j tiver participação total, a troca de informações entre as estruturas obedecerá o critério de partição da estrutura E_j . De acordo com os sistemas mostrados anteriormente, considere E_i sendo $E1$ e E_j sendo $E2$, pode-se ter apenas as informações contidas em uma superclasse do conjunto de especialização da abstração de generalização da estrutura $E1$ exportadas para a estrutura $E2$ se, a informação da superclasse de $E1$ figurar em, pelo menos, uma das especializações da superclasse. Isto porque, o conjunto de especialização da abstração de generalização da estrutura $E2$ sendo total, não permitiria a existência de uma instância de uma superclasse sem um correspondente em pelo menos umas das subclasses.

8ª Regra: se o conjunto de especialização da abstração de generalização da estrutura E_i e da estrutura E_j tiverem ambas participação parcial, a troca de informações ocorrerá de maneira completa

5.4.2. Regras de Atributos

A integração de atributos restringe-se àqueles presentes na estrutura original E_0 . Atributos acrescidos e/ou eliminados nos esquemas derivados não são integrados. Atributos renomeados em esquemas derivados são tratados normalmente, como se não houvessem sido renomeados. Os tipos de dados alterados seguem as regras usuais de conversão de tipos.

9ª Regra: todo atributo chave de uma classe presente numa estrutura E_i será herdado pela classe correspondente na estrutura E_j . Essa regra pode ser exemplificada pelos atributos *id_pessoa_gen* presente na estrutura e que corresponde ao identificador da classe *pessoa_pri*. Esse atributo presente na classe *pessoa_pri* da estrutura E_1 será herdado pela classe *pessoa_pri* da estrutura E_2 .

10ª Regra: se um atributo não chave é obrigatório (*not null*) na estrutura E_i e E_j , a troca da informação (relativa ao atributo) é possível.

11ª Regra: se um atributo não chave é obrigatório (*not null*) na estrutura E_i e não o é em E_j , a troca da informação (relativa ao atributo) é possível.

12ª Regra: se um atributo não chave não é obrigatório (*not null*) na estrutura E_i mas o é na estrutura E_j , a troca da informação (relativa ao atributo) não poderá acontecer.

13ª Regra: se um atributo não chave não é obrigatório (*not null*) na estrutura E_i e não o é em E_j , a troca da informação (relativa ao atributo) poderá ser efetuada.

14ª Regra: um atributo de predicado presente em uma classe C_i só será trocado (integrado), se pelo menos uma das classes especializadas de C_i fizer parte da estrutura genérica de integração. Isto porque o atributo predicado numa dada aplicação pode estabelecer um critério muito particular que em outra aplicação não está sendo considerado.

Para elucidar as regras identificadas para a estrutura de generalização serão apresentados os dois exemplos anteriormente citados levando-se em conta apenas a parte que corresponde a estrutura comum.

Figura 5.5 : Representação da estrutura padrão para S1 e S2

Foi adotada a seguinte convenção na confecção da figura 5.5. As especializações dos dois exemplos (**pesfis_esp/pesjur_esp**) são disjuntas. Os atributos grifados representam atributos identificadores da classe. Os demais atributos, quando em negrito, representam atributos obrigatórios (*not null*) não chave. Por último os atributos em itálico representam os atributos não obrigatórios (*null*) e não chave.

Como mostrado na figura, tem-se que, nos exemplos de modelagem, a estrutura E1 (pertencente ao esquema S1) e a estrutura E2 (pertencente ao esquema S2) apresentam a mesma restrição (disjunção) para o critério de especialização (**pesfis_esp/pesjur_esp**). Com isso, pela primeira regra pode-se concluir que a troca é possível.

Pela **nona regra** tem-se que todo atributo chave (na figura esses atributos aparecem em negrito) é herdado pela classe correspondente. Nesse caso tem-se o atributo **id_pessoa_pri** da classe **pessoa_pri** da estrutura E1 herdado pela classe **pessoa_pri** da estrutura E2.

Sendo o atributo **nc_nome** da classe **pessoa_pri** obrigatório e não chave nas duas estruturas E1 e E2 pela **décima regra** tem-se que o mesmo poderá ser trocado (apenas a informação relativa ao atributo) entre as duas estruturas.

Visto que o atributo **dt_fundacao** da classe **pesjur_esp** é obrigatório na estrutura E1 e não o é na estrutura E2, e considerando a troca de E1 para E2, constata-se, pela **décima primeira regra**, que a troca de informação relativa ao atributo pode ser efetuada. Por outro lado se a troca ocorresse de E2 para E1, pela **décima segunda regra** a informação relativa ao atributo não poderia ser trocada.

Considerando o atributo **tp_estcivil** da classe **pesfis_esp** não obrigatório nas duas estruturas E1 e E2, pela **décima terceira regra** a troca de informação relativa ao atributo poderia ser efetuada.

Para exemplificar a **décima quarta regra**, na figura 5.5 toma-se como base a classe **pesjur_esp** presente nas duas estruturas E1 e E2. O atributo de predicado da mesma é **tp_esppesjur** que está presente em E1 e E2. Contudo, não existe nenhuma classe especializada comum entre as duas estruturas E1 e E2. Logo, este atributo não será trocado entre os sistemas.

É bom salientar que estas regras foram elaboradas para a Abstração de Generalização e portanto, só se aplicam a esta abstração. Estas invariantes e o conjunto de regras definidos não garantem a troca automática de informações quando ocorrer troca no papel do atributo (valor semântico) dentro do esquema de um dos sistemas.

5.5. Conclusão

Neste capítulo é definida uma metodologia de integração de esquemas sob demanda de uma estrutura baseada na abstração de generalização para banco de dados relacionais. Em contraste com as outras abordagens, o processo de integração está intimamente fundamentado no aspecto que, dois sistemas desenvolvidos independentemente poderão trocar informações de maneira automática, se puder ser identificada uma estrutura comum entre eles e, além disso, for possível determinar um conjunto de regras e invariantes válidas para a estrutura identificada, com o intuito de preservar a consistência da base de dados que estará recebendo a informação.

Apresentou-se também um conjunto de regras e invariantes para estruturas originadas de uma Abstração de Generalização, e mostrou-se como podem ser empregadas numa situação de integração de uma aplicação de um sistema de gerenciamento de cooperativa prestadora de serviços médicos, e uma aplicação de controle de pessoal de uma instituição de ensino.

Capítulo 06.

Conclusões e Sugestões de Futuras Pesquisas

6.1. Conclusões

Este projeto partiu do pressuposto que seria possível definir um conjunto de modelagens padrão, um para cada elemento significativo de um conjunto de organizações que possam eventualmente necessitar da troca de dados de maneira não previsível. Havendo um padrão, cada organização deveria então desenvolver seu esquema a partir dos modelos, adaptando-os às suas necessidades. Este trabalho verificou que esta é uma abordagem possível, porém para efetivá-la é ainda necessário que as operações de personalização dos padrões possam ser analisados e as propriedades sintáticas e semânticas dos esquemas resultantes, possam ser rastreados até cada esquema padrão original. Este trabalho apresenta um estudo realizado sobre a preservação dessas propriedades a partir de esquemas padrão desenvolvidos com o emprego de abstrações de generalização.

Embora não exista hoje um suporte explícito para a existência de esquemas padrão, eles existem de fato como parte das bibliotecas básicas de desenvolvimento e suporte dos ambientes para RAD. Assim, este trabalho pode ser aplicado hoje quando ambos os sistemas tenham sido desenvolvidos com o mesmo ambiente de desenvolvimento. No entanto a existência de um conjunto abrangente de regras para o rastreio de propriedades semânticas e sintáticas na evolução de esquemas originados em módulos de esquema padrão, pode servir como incentivo para uma unificação dos módulos de modelagem básicos supridos pelos diversos fornecedores.

Os módulos de modelagem disponíveis hoje em Ambientes RAD foram desenvolvidos com o propósito de acelerar o desenvolvimento de aplicativos, fornecendo protótipos básicos comuns a muitos sistemas. Este trabalho acrescenta o propósito de que a existência de padrões pode facilitar uma futura integração dos esquemas deles derivados.

Diante do estudo realizado pôde-se constatar que as ferramentas avaliadas para RAD não apresentaram uma BTO que pudesse ser utilizada para uma futura integração de esquemas via uma estrutura comum presente na biblioteca. Todos os objetos existentes nas bibliotecas se voltam para aspectos de interface da aplicação o que, sem dúvida, confere às ferramentas a característica RAD, mas apenas do ponto de vista de desenvolvimento de interface e não do ponto de vista de permitir a integração de esquemas de aplicações.

Assim, pode-se considerar este trabalho como uma atividade exploratória para o desenvolvimento de uma nova frente tanto de pesquisa quanto de extensão do suporte ao desenvolvimento de software através de ferramentas adequadas. As BTO existentes atualmente

estão fortemente direcionadas a disponibilizar módulos de software para serem utilizados, através de adaptações, como parte da interface com o usuário. Porém algumas estruturas conceituais, principalmente ligadas a estruturas de dados existem também em estado incipiente. Este trabalho mostrou que é viável o desenvolvimento de estruturas conceituais de dados padrão, juntamente com um conjunto de operações de adaptação dessas estruturas, cujo histórico, se mantido pelas ferramentas junto com as aplicações que as utilizaram, podem permitir a integração de uma parcela considerável dos dados de aplicação com outra, sob demanda, e sem interferência humana, preservando as propriedades estruturais e semânticas dos dados trocados.

6.2. Resultados Alcançados

Além de mostrar a viabilidade teórica e técnica da proposta apresentada inicialmente, este trabalho teve dois outros resultados significativos:

- **a apresentação de uma estrutura conceitual de dados padrão para o super-tipo Pessoa.** Essa estrutura padrão foi derivada a partir de uma estrutura desenvolvida anteriormente e efetivamente utilizada no Sistema de Gerenciamento de Cooperativas Médicas – UniMaster, desenvolvido para a UNIMED-Brasil. Uma estrutura semelhante foi utilizada para o desenvolvimento do sistema Fenix – Sistema de Informações da Pro-reitoria de Pós-Graduação da USP. Ambos os casos são situações reais de sistemas efetivamente desenvolvidos, e no segundo caso, já em operação. Constatou-se que a disponibilidade de uma estrutura conceitual de dados padrão, tal como a apresentada teria efetivamente facilitado o desenvolvimento daqueles sistemas, apoiando a premissa de que tais módulos na BTO de uma ferramenta RAD agilizaria o desenvolvimento de sistemas que a utilizassem. Além disso, com a disponibilidade do histórico da configuração, tal como descrito no quinto capítulo, (embora isso tenha sido feito de fato apenas manualmente, dada a inexistência real de um tal histórico), a integração automática de esquemas dessas duas aplicações poderia permitir a troca de um conjunto abrangente de dados;
- **a definição de um conjunto de regras para a integração de esquemas de uma estrutura baseada na hierarquia de generalização.** Essas regras, apresentadas também no quinto capítulo, podem ser aplicadas em qualquer situação onde ocorram hierarquias de generalização.

6.3. Sugestões de Futuras Pesquisas

Diversas atividades podem dar continuidade ao trabalho aqui apresentado. A seguir, são apresentadas algumas que mais diretamente foram consideradas durante o trabalho, ou que podem dar sustentação e auxiliar a aplicação prática deste trabalho.

- Durante o desenvolvimento aqui descrito, percebeu-se que os módulos desenvolvidos são mais universalmente reutilizáveis quanto mais os elementos semânticos utilizados se restringem e se organizam em uma estrutura básica. Por exemplo, o estudo de caso apoia-se fortemente na existência de uma única hierarquia da abstração de generalização, encabeçada pelo super-tipo Pessoa. Outras hierarquias semelhantes centradas na mesma abstração podem ser facilmente percebidas como amplamente reutilizáveis, tal como hierarquias para Veículos, Documentos Financeiros, Documentos Fiscais, etc. Alguns outros módulos estudados concentram-se também em apenas uma abstração, como por exemplo, históricos utilizam a abstração de agregação em uma estrutura em estrela. Essa avaliação de um conjunto abrangente de situações reais, porém, se confirmada, pode servir tanto como diretriz como elemento de validação para a criação de módulos para bibliotecas de *templates* para estruturas conceituais de dados padrão.
- Desenvolvimento de um editor para a definição conceitual de esquemas de aplicações, onde os módulos originários das BTO pudessem ser adaptados para a utilização nos esquemas das aplicações sendo desenvolvidas, com capacidade para preservar o histórico das atualizações. Esse editor poderia ser parte de uma ferramenta CASE, e incorporaria ao menos o conjunto de regras para a integração de esquemas de uma estrutura baseada na hierarquia de generalização, porém poderia incluir também regras definidas por um estudo realizado sobre outras abstrações, tal como sugerido no tópico anterior.
- Desenvolvimento de uma ferramenta de validação e auxílio (tipo “*wizard*”) para o desenvolvimento de esquemas a partir de módulos pré-existent, para verificar as propriedades e dados que podem ser preservados a partir de cada adaptação, e que possa auxiliar a maximizar tais propriedades e dados. Tal ferramenta poderia apoiar-se nas informações registradas pelo editor de definição conceitual de esquemas de aplicações sugerido no tópico anterior, para gerar a modelagem física de uma aplicação, preservando os dados necessários para atividades de integração sob-demanda com a base de dados já

alimentada. Essa ferramenta de validação e auxílio poderia ser parte da mesma ferramenta CASE onde o editor estivesse incorporado.

- Desenvolvimento de uma ferramenta de integração automática de esquemas sob-demanda, utilizando as informações preservadas pela ferramenta sugerida anteriormente referentes a duas bases de dados a serem integradas, para um gerenciador de bases de dados em particular. Essa ferramenta deveria fazer parte de um módulo de *run-time*, disponibilizado junto com cada aplicação cuja integração fosse pretendida, atuando como um utilitário de *dump* da base de dados extremamente especializado.

Bibliografia

Referências Bibliográficas

[ATK89]	ATKINSON, M. et. al.. The objetct oriented database system manifesto. Rapport Technique Altaïr, INRIA-LRI, October 1989.
[BAN87a]	BANERJEE, J. et al.. "Data model issues for object oriented applications", ACM Transactions on Office Information Systems, , Vol. 5, No. 1, pp. 3-26, January 1987.
[BAN87b]	BANERJEE, J.; KIM, W.; KIM, H.J. & KORTH, H.F.. "Semantics and implementation of schema evolution in object oriented databases", Proceedings of the ACM SIGMOD 87 Conference, Vol. 16, No. 3, 1987.
[BEH97]	BEHM, A.; GEPPERT, A. & DITRICH, K. R. – "On the migration schemas and data to object oriented database systems", In Proceedings of the 5 th International Conference on Re-Technologies in Information Systems, Klagenfurt, Austria, December 1997.
[BER91]	BERTINO, E. & MARTINO, L.. Object oriented database management systems: concepts and issues, IEEE Computer, pp. 33-47, Abril 1991.
[BER93]	BERTINO, E. & MARTINO, L.. Object oriented database systems, International Computer Science Series, Addison-Wesley, 1993.
[BET94]	BETZ, M.. "Omg's corba: an emerging standard for real-world implementations", Dr. Dobb's Special Report, pp. 8-12, 1994.
[CAM92]	CAMOLESI, L. J.. Suporte a acesso multi-usuário em bases de dados orientadas a objetos através de esquemas suplementares, Dissertação de Mestrado, ICMSC-USP, 1992.
[CAT94]	CATTELL, R.. Object data management: object oriented and extended relational, Addison-Wesley, Ed. 1994.
[CHI94]	CHIANG, R.; BARON, T. & STOREY, V.. "Reverse engineering of relational databases: extraction of an EER model from a relational database", Data & Knowledge Engineering, 12: 107-142, 1994.

[CHO86]	CHOU, H.T. & KIM, W.. "A unifying framework for versions in a CAD environment", Proceedings of the International Conference on Very Large Databases, August 1986.
[CHO93]	CHORAFAS, D. N.. Manufacturing databases and computer integrated systems, Crc Press, Inc./Lewis Publishers, p. 320, 1993.
[DAV87]	DAVIS, K. & ARORA. "Converting a relational database model into an entity-relationship model", In 6 th International Conference of the Entity Relationship Approach, 1987.
[DEU90]	DEUX, O. et. al.. The story of O ₂ , IEEE Transactions on Knowledge and Data Engineering – Special Issue on Database Prototype Systems, Vol.2, No.1, pp. 91-107, março 1990.
[DEU91]	DEUX, O.. The O ₂ system, Communications of the ACM, Vol.34, No.10, pp.34-48, October 1991.
[SYB96e]	CONCORD, M.. Distributed objects, multiplatform support, compiled code and internet capabilities strengthen market-leading application development tool, http://www.sybase.com/inc/corpinfo/press/23ae.htm , May 6, 1996.
[DEY96]	DEYHIMI, D. & MOSLEY, D.. Advanced powerbuilder 5 techniques, Ed. Wiley Computer Publishing, 1996.
[FIG94]	FIGUEIREDO, L. R.. Tratamento das implicações semânticas de relacionamentos em modelos de dados orientados a objetos, Mini-dissertação de mestrado - ICMSC-USP, 1994.
[FIG96]	FIGUEIREDO, M. B.. Representação de áudio em banco de dados, Mini-dissertação de mestrado, ICMSC - USP, 1996.
[FIN96]	FINKELSTEIN, R.; GREENWALD, R. & SISTA, K.. Powerobjects developer's guide, Osborne Mc-Graw-Hill, USA, 1996.
[GRE96]	GREEN, W. T.. Building with and extending the powersoft foundation classes (PFC), http://www.pfc_concentric.net/~bgcastle/PAPERS/PRF/pfc_main.htm , August 1996.
[KAT84]	KATZ, R. & LEHMAN, T.. "Database support for versions and

	alternatives of large design files," IEEE Transactions on Software Engineering, Vol. SE-10, No. 2, pp. 191-200, March 1984.
[KAT86]	KATZ, R.; CHANG, E. & BHATEJA, R.. "Version modeling concepts for computer-aided design databases", Proceedings of the ACM SIGMOD 86 International Conference on Management of Data, Washington, D.C., Vol. 15, No. 2, May 1986.
[KIM87]	KIM, W.; CHOU, H.T. & BANERJEE, J.. "Operations and implementation of complex objects", Proceedings of the Data Engineering Conference, Los Angeles, CA, 1987.
[KIM89a]	KIM, W.; BERTINO, E. & GARZA, J. F.. "Composite object revisited", ACM SIGMOD 89 Conference, Portland, Oregon, Vol. 18, No. 2, pp. 337-347, Junho 1989.
[KIM89b]	KIM, W. & LOCHOVSKY, F. H.. Object oriented concepts, databases, and applications, Addison-Wesley, 1989.
[KIM89c]	KIM, W.; BALLOU, N.; CHOU, H. T.; GARZA, J. F. & WOELK, D.. "Features of the ORION object oriented database system", pp. 251-281.
[KIM95]	KIM, W. et. al. The object model, interoperability and beyond, New York, Addison-Wesley Publishing Company, 1995.
[KOL95]	KOLSTE, B. & PETERSEN, D.. Powerobjects handbook, Osborne McGraw_Hill, USA, 1995.
[LIT90]	LITWIN, W.; MARK, L. & ROUSSOPOULOS, N.. "Interoperability on multiple autonomous databases", ACM Computing Surveys. V. 22, No. 3 Set. 1990.
[LOC85]	LOCHOVSKY, F.. Database engineering, IEEE Computer Society, vol. 8, No. 4, December 1985, special issue on object oriented systems.
[MON94]	MONTENEGRO, F. & PACHECO, R. Orientação a objetos em C++, Ed. Ciência Moderna LTDA, Brasil, 1994.
[NAN96]	NANDA, B.; BODEPUDI, P. & HARTWELL, B. Instant

	powerbuilder 5.0 objects, Wrox Press LTDA, Canadá, 1996.
[NAV94]	NAVATHE, S. B. & ELMASRI, R.. Fundamentals of database systems. 2. ed., The Benjamin/Cummings Publishing Company, Inc., 1994.
[ORA96]	ORACLE Corp.. Powerobjects. http://www.oracle.com/products/tools/power_objects/pcmagrev.html , January, 1996.
[ORA96b]	ORACLE Corp.. FAQ : Frequently answer questions, http://www.oracle.com/products/tools/power_objects/html/opofaq.html , Juny, 1996.
[PER91]	PERRIZO, W.; RAJKUMAR, J. & RAM. P.. "HYDRO: a heterogenous, distributed database system", In: International Conference On Management of Data, 1991, Denver, Colorado. Anais ... Colorado, [ACM-SIGMOD], Vol. 20, No. 2, 1991.
[RAM91]	RAM, S.. Guest editor's introduction: heterogenous, distributed database systems. IEEE Computer. Vol. 24, No. 12, Dez. 1991.
[REE96]	REED, P.. Hands on: power objects. http://www.oracle.com/products/tools/power_objects/dbms.html , February 1996.
[SHE90]	SHETH, A. P. & LARSON, J. A.. "Federated database systems to managing distributed heterogeneous and autonomous databases", ACM Computing Surveys. Vol. 22, No. 3, Set. 1990.
[STO90]	STONEBAKER, M. et. al. Third-generation database system manifesto. The Committee for Advanced DBMS Function, ACM SIGMOD Record, Vol.19, No.3, September 1990.
[SYB96a]	DataArchitect user's guide – S-Designor Version 5, Powersoft – Sybase, USA – 1996.

[SYB96b]	AppModeler user's guide – S-Designor Version 5, Powersoft – Sybase, USA – 1996.
[SYB96c]	ProcessAnalyst user's guide – S-Designor Version 5, Powersoft – Sybase, USA – 1996.
[SYB96d]	PowerBuilder foundation class library user's guide, PowerBuider Version 5, Powersoft – Sybase, USA – 1996.
[TAK90]	TAKAHASHI, T. & LIESENBERG, H. K. E.. Programação orientada a objeto, VII Escola de Computação, São Paulo, 1990.
[UCH94]	UCHÔA, E. M.A. ; SILVA, S. D.; MELO, R. N. – Modelo de dados do HERO – sistemas de bancos de dados heterogêneos orientado a objetos”, Anais do IX Simpósio Brasileiro de Banco de Dados, São Carlos, SP, pp. 142-156, Setembro 1994.
[WOE86]	WOELK, D.; KIM, W. & LUTHER, W.. “An object oriented aproach to multimedia databases”, Proceedings of the ACM SIGMOD 86 Conference on Management of Data, Washington D.C., Vol. 15, No. 2, May 1986.