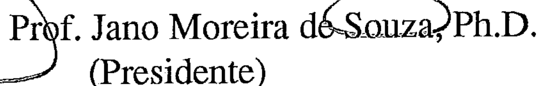
Um Modelo de Objetos para o Sistema de Objetos GEOTABA

LUIZ CARLOS MONTEZ MONTE

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Jano Moreira de Souza, Ph.D.
(Presidente)

Profª Ana Regina C. da Rocha, D.Sc.

Profª Ana Maria de Carvalho Moura, D.Ing.

Profª Regina Célia de Souza Pereira, D.Sc.

Prof. José Lucas Mourão Rangel Netto, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
MAIO DE 1993

MONTE, LUIZ CARLOS MONTEZ
Um Modelo de Objetos para o Sistema de
Objetos GEOTABA [Rio de Janeiro] 1993.
VIII, 250 p. 29,7 cm (COPPE/UFRJ, D.Sc.,
Engenharia de Sistemas e Computação, 1993)
Tese - Universidade Federal do Rio de
Janeiro, COPPE
1. Banco de Dados
I. COPPE/UFRJ II. Título (série)

A Adriana e Lúcia

Agradecimentos

A Adriana, pela sua dedicação e apoio constantes.

A Lúcia, pelo seu carinho.

Aos meus pais e a minha tia Daisy, pela motivação e apoio inestimável.

Ao Professor Jano, pela sua sábia orientação.

À Professora Regina Pereira e ao Professor Rangel pelo apoio e por darem a honra de participarem da minha banca de tese.

Às Professoras Ana Maria Moura e Ana Regina Rocha por me darem a honra de participarem da minha banca de tese.

À Professora Dina e ao Professor Marinho pelo apoio.

Ao colega Jugurta pela implementação do ProtoGEO, entre outras colaborações.

Aos colegas Marta, Trota, Hélcio e Guilherme pela colaboração.

Aos colegas do HiperFicha: Cláudia Susie, Emília, Sandra, Patrícia, Eduardo Chaves, Jaime, Milton, Jorge Rangel e Olavo.

À Professora Miriam, chefe do Departamento de Computação, aos alunos e funcionários do Curso de Informática da UFF, pela compreensão.

Aos demais colegas, professores e funcionários da COPPE/Sistemas.

Ao CNPq, pelo apoio financeiro.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências (D.Sc.)

Um Modelo de Objetos para o Sistema de Objetos GEOTABA

LUIZ CARLOS MONTEZ MONTE

Maio de 1993

Orientador: Jano Moreira de Souza

Programa: Engenharia de Sistemas e Computação

O Projeto TABA, em desenvolvimento no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, visa a criação de uma estação de trabalho para engenharia de software. Em uma estação TABA, o engenheiro de software cria ambientes de desenvolvimento de software utilizáveis inclusive na própria estação. Esta tese propõe que dados, procedimentos e interfaces com usuário, presentes ou desenvolvidos no TABA, sejam gerenciados por um sistema único, integrador de funções de bancos de dados, linguagens de programação e gerência de interfaces com usuário. Este tipo de sistema é aqui definido como um Sistema de Objetos (SOO) e características para um SOO específico, o GEOTABA, para uso pelos ambientes de desenvolvimento de software do TABA, são discutidas. Mais especificamente é visto o modelo de objetos do GEOTABA. Um Modelo de Objetos (MOO) é aqui definido como o modelo de representação da informação em um SOO e integra propriedades de modelos semânticos de dados com modelos de interfaces com usuário e de implementação de programas. Uma representação gráfica para modelagem é definida, juntamente com a linguagem DEMO para descrição e manipulação de objetos. Alguns protótipos envolvendo aspectos do modelo de objetos do GEOTABA são apresentados.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Doctor of Sciences (D.Sc.)

An Object Model to the GEOTABA Object System

LUIZ CARLOS MONTEZ MONTE

May 1993

Thesis Supervisor: Jano Moreira de Souza

Department: Systems and Computing Engineering

The TABA Project, under development in COPPE/UFRJ Systems and Computing Engineering Department, aims at the establishment of a software engineering workstation. On a TABA workstation, a software engineer creates software development environments inclusive to be used in the workstation itself. This Thesis proposes that data, procedures and user interfaces, present or developed with TABA, should be managed by a single system, integrating database, programming language and user interface management functionality. Such kind of system is defined as an Object System. Features of a specific object system, GEOTABA, to be used supporting TABA's software development environments, are discussed. More specifically, the GEOTABA object model is presented. An object model is defined in this Thesis as the information model for an object system, integrating semantic data model features with user interface and programming models. A conceptual data modeling graphic notation - the *Diagramas de Objetos* - is introduced, in the same way that DEMO object description and manipulation language. Some prototypes, envolving GEOTABA object model aspects, are introduced.

Um Modelo de Objetos para o Sistema de Objetos GEOTABA

Introdução **1**
1.1. Apresentação 1
1.2. Motivação 1
1.3. Objetivo e Alcance da Tese 4
1.4. Características Gerais do Taba 5
1.5. Organização da Tese 5
Sistemas de Objetos **7**
2.1. Introdução 7
2.2. Sistemas Correlatos 10
2.3. Sistemas de Objetos e Ambientes de Desenvolvimento de Software 27
2.4. Características Funcionais de um Sistema de Objetos 28
2.5. Sumário 30
Modelos de Objetos para Sistemas de Objetos **31**
3.1. Introdução 31
3.2. Representação da Informação 32
3.3. Conflitos de Representação 79
3.4. Características de Modelos de Objetos para Sistemas de Objetos 80
3.5. Sumário 82
O Modelo de Objetos do GEOTABA **84**
4.1. Introdução 84
4.2. Modelos de Objetos e Ambientes de Desenvolvimento de Software 86
4.3. Características Principais do Modelo 88
4.4. O Nível Conceitual 93
4.5. O Nível de Implementação 195
4.6. O Nível Externo 204
4.7. O Nível de Representação 207
4.8. Sumário 213
Protótipos Relativos ao Modelo de Objetos **216**
5.1. Introdução 216
5.2. ProtoGEO 217
5.3. Flecha 223
5.4. GOA e PARGOA 223
5.5. HiperFicha e Bandar 224
Sumário e Perspectivas Futuras **227**
6.1. Sistemas de Objetos: Integrando Processamento, Armazenamento e Manipulação de Informação 227
6.2. O Modelo de Objetos do GEOTABA 228
6.3. O Nível Conceitual do Modelo de Objetos do GEOTABA 231
6.4. Os Diagramas de Objetos 234
6.5. A Linguagem DEMO 234
6.6. Os Níveis Externo e de Representação e sua Relação com Aspectos 236
6.7. Considerações Finais 237
Anexo A: Exemplo de Modelagem **240**
Referências Bibliográficas **241**

Capítulo 1

Introdução

1.1. Apresentação

Esta tese define Sistema de Objetos como um sistema integrado de gerência de bases de dados, procedimentos e interfaces com usuário. Propõe um sistema de objetos denominado GEOTABA a ser utilizado como núcleo dos ambientes de desenvolvimento de software gerados nas estações de desenvolvimento de software TABA [SOUZ90] [ROCH88]. Define também Modelo de Objetos como a ferramenta conceitual para a modelagem de informação gerenciada por sistemas de objetos e discute características a serem observadas pelo modelo de objetos do GEOTABA.

Esta tese também inclui a descrição de um protótipo que implementa partes do modelo de objetos do GEOTABA voltadas à integração da estruturação conceitual da informação com a programação de rotinas que a manipulam. Uma notação gráfica para a estruturação conceitual da informação também é introduzida. Outros protótipos referentes a implementação de aspectos do modelo de objetos do GEOTABA são sumarizados.

1.2. Motivação

Este trabalho se refere ao desenvolvimento do Sistema de Objetos GEOTABA. Este foi inicialmente idealizado para servir ao projeto TABA [SOUZ90], em andamento no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. O GEOTABA teve seus objetivos ampliados de modo a contemplar outras classes de aplicações não convencionais.

O projeto TABA [ROCH90] tem por objetivo a construção de uma estação de trabalho para o engenheiro de software. Nesta estação, auxiliado por técnicas de inteligência artificial, o engenheiro de software cria e configura

ambientes de desenvolvimento de software para algum projeto específico, considerando o perfil da equipe de desenvolvimento, o tipo da aplicação, as características do usuário, custos e qualidade desejados [AGUI92].

Um ambiente automatizado para desenvolvimento de software reúne uma variedade de ferramentas técnicas e gerenciais. A integração e controle destas ferramentas têm três componentes primordiais:

1. a informação manipulada pelas ferramentas;
2. a interface com usuário;
3. a implementação e execução das ferramentas.

A gerência das informações utilizadas no ambiente poderia ser responsabilidade de um sistema de gerência de bases de dados (SGBD). A interação com usuário, numa hipótese remota, poderia estar sobre o controle de um sistema de gerência de interface com usuário (SGIU). Por fim, as ferramentas em si poderiam estar sob a responsabilidade de um complexo formado pelo sistema operacional, por múltiplos processadores de linguagens de programação e pelo próprio ambiente de desenvolvimento de software. A provável incompatibilidade entre os gerenciadores responsáveis pela integração do ambiente, cada um seguindo um modelo conceitual próprio, seria um fator determinante para a complexidade de construção de ferramentas integráveis aos ambientes de desenvolvimento de software gerados na estação TABA. Os SGBDs tradicionais também demonstram ter limitações quanto ao suporte de dados envolvidos em ambientes de desenvolvimento de software.

Esta tese propõe que a implementação e processamento de ferramentas, as informações por elas manipuladas e a sua interação com usuário sejam gerenciadas por um único sistema, baseado em um modelo de representação da informação que integre estes aspectos. Esta categoria de sistema de gerência é aqui denominada de Sistema de Objetos (SOO). Um sistema de objetos, o GEOTABA, é então proposto para servir de núcleo dos ambientes de desenvolvimento de software do TABA.

A pura integração de SGBDs, SGIUs e linguagens de programação não atende aos objetivos do projeto TABA. Alguns possíveis tipos de ambientes de desenvolvimento de software desejados impõem requisitos especiais para a gerência de dados, interação com usuário e programação de ferramentas.

A tecnologia corrente de banco de dados não suporta adequadamente alguns destes requisitos [HITC85], como, por exemplo, o suporte a versões, objetos complexos e transações de longa duração [MELO87]. Os sistemas de banco de dados atuais se apoiam em modelos de dados voltados para a implementação eficiente em máquinas com recursos ultrapassados. Há necessidade que a gerência de dados passe a adotar modelos que incorporem estruturas capazes de

mapear diretamente a semântica real da informação. A maior capacidade de processamento e armazenamento dos computadores atuais viabiliza tais modelos. Como é inviável a definição de estruturas que atendam todas as aplicações previstas, é desejável que estes modelos sejam extensíveis, ou seja, capazes de gerar novas estruturas de modelagem da informação.

Um ambiente de desenvolvimento de software é um tipo de aplicação não convencional. Seus usuários convivem com a necessidade de definição dinâmica de novos tipos de informação, e a relação entre o número de tipos de informações definidas e o volume de informações manipuladas é muito maior do que nas aplicações convencionais.

Ferramentas, aplicações, procedimentos e funções serão criados, mantidos e alterados nas estações de engenharia de software TABA. Todavia, há fatos que estimulam a encará-los de modo diferente dos demais dados que povoam os ambientes de desenvolvimento de software do TABA, como, por exemplo,

1. eles são "executados";
2. muitas vezes são os responsáveis pela interpretação semântica dos dados;
3. o seu desenvolvimento e manutenção exigem técnicas especiais.

É razoável, portanto, que haja um sistema de gerência de programas embutido nas estações, que suporte adequadamente uma metodologia de desenvolvimento de software reusável e manutenível, e que se integre com os dados e interfaces com usuário da estação.

Os sistemas de gerência de interface com usuário não encontraram até hoje o mesmo sucesso dos sistemas de gerência de bases de dados. Isto a despeito da importância hoje dada à interação com usuário. A razão deste insucesso reside no objetivo paradoxal dos SGIUs de separar a interface com usuário do restante da aplicação, responsável pela semântica da informação processada, o que entra em contradição com as características das interfaces com usuário modernas. De fato, estas se apoiam na chamada "manipulação direta", onde a interface com usuário está fortemente vinculada à semântica da informação manipulada, conforme discutido mais adiante.

O paradigma da orientação a objetos, empregado em linguagens de programação, propõe um mecanismo conceitual onde dados e procedimentos podem ser tratados sob um prisma unificador [GOLD83]. Sistemas de Gerência de Banco de Dados Orientado a Objetos (SGBDOOs) trazem as características da orientação a objetos para o âmbito dos sistemas de bancos de dados. Sistemas de gerência de interface com usuário utilizando o paradigma da orientação a objetos têm sido construídos visando resolver as questões de integração

da interface com usuário com a semântica da aplicação na comunicação homem-computador com manipulação direta. É, portanto, a orientação a objetos a base tecnológica que propiciou a definição de um Sistema de Objetos baseado em um modelo que integrasse programas, dados e interação com usuário.

Uma característica determinante do GEOTABA é a extensibilidade, que permite a criação dinâmica de novos tipos de dados, apropriados a aplicações com requisitos diversificados. O papel do sistema de objetos GEOTABA pode, então, ser estendido para suportar outras aplicações não convencionais, além de ambientes de desenvolvimento de software.

1.3. Objetivo e Alcance da Tese

O TABA é um projeto envolvendo as Linhas de Pesquisa de Engenharia de Software e de Bancos de Dados no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. O porte do projeto envolve uma equipe de dezenas de pesquisadores, incluindo alunos mestrandos e doutorandos. Contido no TABA, o projeto do GEOTABA também mantém dimensões que incluem o trabalho de diversas teses de mestrado e doutorado.

O objetivo desta tese é fixar características a serem suportadas pelo GEOTABA, de modo a que o rumo da pesquisa em torno da construção do GEOTABA a elas sejam submetidas. Estas características podem ser reunidas em dois grupos:

1. Gerais, sobre o Sistema de Objetos. Por exemplo: controle de concorrência, versionamento, programação orientada a objetos e interação com usuário multialinhavada. Este grupo de características é detalhado no Capítulo 2.
2. Específicas, sobre o Modelo de Objetos. Dizem respeito à representação da informação. Por exemplo: identidade de objeto, *late-binding*, relacionamentos, sub-objetos, vistas e representantes. Este grupo de características é detalhado nos Capítulos 3 e 4.

A definição do modelo de objetos envolve profundamente modelos semânticos de dados, linguagens de programação e comunicação homem-computador. Esta tese não tem como objetivo a definição completa do modelo. Novamente, ela se limita a fornecer um arcabouço onde um número de questões específicas possa ser posteriormente tratado em detalhe.

Embora o elaborado nesta tese seja fundamental para a construção definitiva do GEOTABA, tópicos também fundamentais, porém subsequentes a este trabalho, como é o caso da definição da arquitetura final do GEOTABA, não são aqui tratados. Todavia, aqui são descritos uma série de protótipos referentes

ao modelo de objetos que deram subsídios e comprovações ao proposto e que muito contribuirão à versão final do sistema.

1.4. Características Gerais do Taba

O projeto TABA tem por objetivo a construção de uma estação de trabalho para o engenheiro de software. Nesta estação poderão ser configurados vários tipos de ambientes de desenvolvimento. Esta configuração ocorre no chamado meta-ambiente. Através dele, um engenheiro de software pode gerar ambientes específicos de desenvolvimento. O meta-ambiente está sempre evoluindo, pois novas ferramentas podem ser geradas e a ele incorporadas, de modo a que novos tipos de ambientes, utilizando estas novas ferramentas, possam ser gerados. Os ambientes gerados, a partir do meta-ambiente, servirão para o desenvolvimento e execução de software específico. Nestes ambientes, cada ferramenta desenvolvida poderá manipular os mais diversos tipos de modelos utilizados no desenvolvimento de software, como por exemplo: modelos funcionais, modelos de dados, diagramas de transição, modelos para representação de conhecimento, gramáticas e outros.

Mesmo a definição, construção e a simulação de novas ferramentas de software, poderão ser realizadas em ambientes gerados especificamente para este fim. Um aspecto importante destes ambientes para desenvolvimento de ferramentas é a necessidade que algumas ferramentas têm de manipular objetos com os mais diversos graus de complexidade e que, em alguns casos, são definidos melhor em termos de seu comportamento.

O dinamismo com que novas ferramentas de software deverão ser desenvolvidas, aliado à constante evolução do meta-ambiente, demonstram a necessidade de se adotar, no TABA, um gerenciador de informação que seja naturalmente extensível. Ou seja, o modelo deste gerenciador deve ser genérico e suficientemente poderoso e flexível para representar qualquer outro tipo de modelo.

1.5. Organização da Tese

O Capítulo 2 discute aspectos de sistemas de banco de dados, interfaces com usuário e linguagens de programação existentes, visando a definição de um tipo de sistema de gerência de informação - o Sistema de Objetos - adequado à integração destes aspectos. É discutida a dificuldade atualmente existente nesta integração. O papel que um sistema de objetos integrado exerceria em um ambiente de desenvolvimento de software é apresentado. Por fim, são resumidas as

características propostas para um sistema de objetos, deixando para o capítulo seguinte as características referentes ao modelo de objetos do sistema.

O Capítulo 3 inicialmente discorre sobre modelos de dados, programação com objetos e modelos de especificação de interfaces com usuário, observando que estas áreas fornecem o material básico para a criação do modelo de objetos desejado. A seguir são apresentadas algumas exigências gerais impostas por ambientes de desenvolvimento de software ao modelo de objetos. Por último, são resumidos os requisitos globais do modelo, concluídos a partir do apresentado anteriormente.

O Capítulo 4 descreve o modelo de objetos do GEOTABA, que atende os requisitos definidos no Capítulo 3 A apresentação do modelo inicia com as características que envolvem a integração entre a conceituação da informação, sua implementação e sua representação para o usuário. Nisto se inclui a modelagem da informação em níveis. A seguir são descritas as especificidades do modelo referentes a cada nível de modelagem.

O Capítulo 5 mostra alguns protótipos desenvolvidos relacionados a aspectos do modelo de objetos do GEOTABA. Ênfase especial é dada ao METAGEO, que elabora um metaesquema para os níveis conceituais e de implementação do modelo, referente às propriedades de classes, tipos e objetos. O Capítulo 6 apresenta as considerações finais da tese.

Capítulo 2

Sistemas de Objetos

2.1. Introdução

2.1.1. Apresentação

Este capítulo introduz o conceito de Sistemas de Objetos (SOOs), discutindo o seu propósito, as alternativas existentes atualmente, sua função em um ambiente de desenvolvimento de software, características a serem atendidas e lista as propriedades de um SOO, o GEOTABA, específico para as estações de desenvolvimento de software TABA. Alternativas de implementação de SOOs não fazem parte do escopo deste trabalho.

O termo sistema de objetos é aqui usado em substituição a sistema de gerência de objetos (SGO). Este último e, mais especificamente, seu correspondente em inglês - "Object Management System (OMS)" - vêm sendo utilizados na literatura e em pesquisas sem que haja uma definição precisa adequada a todos que utilizam estes termos. É muito tênue a distinção atual entre os termos sistema de gerência de objetos, sistema de gerência de bases de objetos (SGBO)[1] e sistema de gerência de bases de dados orientadas a objetos (SGBDOO)[2].

Para se ter um vocabulário preciso a ser empregado neste trabalho, uma série destes termos são discutidos de modo a estabelecer maior separação entre os conceitos. Algumas definições talvez não possam ser mantidas fora do escopo

---

[1] *Object Base Management System (OBMS)*, em inglês.

[2] *Object Oriented Data Base Management System (OODBMS)*, em inglês.

deste trabalho devido ao grande número de sistemas existentes que se autodenominam com algum destes termos mal definidos.

Um sistema de gerência (SG) se caracteriza pela centralização de soluções de problemas comuns a diversas aplicações. Um SGO, um SGBD ou um SGIU (sistema de gerência de interface com usuário)[3] são tipos de sistemas de gerência. Por exemplo, o acesso eficiente a dados persistentes é provido pelos SGBDs, dispensando-se cada aplicação desta preocupação.

O paradigma da orientação a objetos obteve sucesso em linguagens de programação, bancos de dados e interfaces com usuário. Um sistema de objetos (SOO), conforme aqui proposto, usa este paradigma para a unificação das gerências de dados, de interação com usuário e de aplicações.

Objetos gerenciados por estes SOOs possuem estado e comportamento, correspondendo respectivamente a dados e procedimentos que os manipulam. Além disto, estes objetos possuem representações para interação com usuário. Assim, por exemplo, um objeto que represente um automóvel tem dados, tais como seus componentes (rodas, portas, etc.) e atributos (modelo, cor, ano, etc.), executa ações e pode ser visualizado e manipulado de diversas maneiras na tela ou em outro meio de interação com usuário.

2.1.2. O Problema das Alternativas Existentes

Os SGBDs atuais não têm dispensado as linguagens de programação da tarefa de programar aplicações, transações e ferramentas. Por outro lado, as linguagens de programação não têm se mostrado suficientemente adequadas para a manipulação de dados persistentes, ou seja, que resistem após a execução de um programa. Como solução para estes problemas tem-se proposto tanto Linguagens de Programação com Persistência - estendendo alguma linguagem de programação ou criando uma nova - como Bancos de Dados Orientados a Objetos, que permitem a programação, em alguma linguagem de programação, de métodos, isto é, funções específicas de uma classe de objeto.

O casamento das linguagens de programação com as linguagens de banco de dados tem seus desajustes, causados por diferentes tipos de dados e diferentes níveis de abstração entre os dois tipos de linguagem. Este problema é conhecido como "impedance mismatch"[4] entre estas linguagens.

---

[3]*User Interface Management System (UIMS)*, em inglês.

[4]A tradução poderia ser "desencontro de impedância".

Uma solução diferente se denomina Linguagem de Programação de Banco de Dados (LPBD)[5] [ATKI87], correspondendo a uma linguagem projetada para atender simultaneamente os aspectos de programação de sistemas e manipulação de dados. O paradigma da orientação a objetos é usado na maioria desses projetos pelos motivos já mencionados.

Outro desencontro ocorre entre a especificação de interfaces com usuário e a programação de aplicações. A origem do problema está no crescimento de importância que a comunicação homem-computador teve recentemente. Devido ao aumento da capacidade de computação e visualização disponível a cada usuário, as exigências em relação a qualidade das interfaces com usuário cresceram exponencialmente. Em uma grande parte dos sistemas, a maior fatia dos custos de desenvolvimento fica por conta da criação de software para a interação com usuário.

Dezenas de sistemas de gerência de interface com usuário (SGIUs) foram então propostos, visando a separação do desenvolvimento e processamento das interfaces em relação ao restante da aplicação. Isto estimularia a reutilização e compartilhamento de elementos de interface com usuário, a prototipação e o desenvolvimento de novas interfaces a partir de outras.

Entretanto, os SGIUs não encontraram a aceitação esperada, correspondente à tida pelos SGBDs. Ocorre que a tendência pela utilização de interfaces com "manipulação direta" na tela, pelo usuário, de objetos da aplicação, exige uma colaboração intensa entre a interface com usuário e o cerne da aplicação. A separação de deveres preconizada pelos SGIUs fica mais difícil de ser alcançada.

A orientação a objetos tem sido usada como a base de muitos SGIUs que tratam este problema. Porém, o casamento das interfaces com usuário com as aplicações e os dados persistentes extrapola os objetivos de cada um dos sistemas de gerência em separado. Este casamento deve ser tratado no caso de um sistema de objetos como o GEOTABA.

A abordagem integrada e orientada a objetos em um sistema de objetos permitiria que se percebesse um objeto visualizado na tela como sendo o mesmo objeto cujos dados estão armazenados na base de objetos e como sendo o objeto que reúne métodos e procedimentos de utilização.

---

[5]*Data Base Programming Language (DBPL)*, em inglês.

2.1.3. Organização do Capítulo

A seguir, na Seção 2.2., são apresentados sistemas correlatos a SOOs e definidos termos chaves empregados neste trabalho. A Seção 2.3. relaciona SOOs a ambientes de desenvolvimento de software. A Seção 2.4. enumera as funções esperadas de um SOO e de seu modelo de objetos. Como considerações finais, na Seção 2.5., são apresentados a proposta de desenvolvimento de um SOO e um sumário do capítulo.

2.2. Sistemas Correlatos

Para melhor distinguir sistemas de objetos (SOOs) dos outros sistemas relacionados, esta seção apresenta as características de cada um desses sistemas. Sistemas de objetos, assim como sistemas de gerência de interface com usuário e sistemas de gerência de bases de conhecimento, são conceitos desenvolvidos a partir dos sistemas de gerência de bases de dados (SGBDs). A apresentação destes, a seguir, visa ressaltar o que deve ser preservado, estendido ou retirado nos SOOs.

2.2.1. Sistemas de Gerência de Bases de Dados

Um SGBD é um sistema de computação que permite aos usuários criar e manter bases de dados [ELMA89] [DATE90] [ULLM89]. Dados são fatos conhecidos que podem ser registrados e que têm significado implícito. Uma base de dados é uma coleção de dados logicamente coerente, projetada, construída e populada visando um conjunto de propósitos específicos e representando algum aspecto do mundo real.

Uma característica fundamental dos SGBDs é que estes contêm não apenas a base de dados propriamente dita, mas também uma descrição desta no catálogo do sistema. "No processamento através de arquivos tradicional, a definição dos dados é tipicamente parte dos programas aplicativos. Desse modo, estes programas são restritos a bases de dados cuja estrutura está declarada nestes programas." [ELMA89]. Já um SGBD, para não ser restrito a determinadas aplicações tem que se referir ao catálogo.

A possibilidade de se poder alterar a estrutura dos dados armazenados sem que se tenha que reescrever os programas que os acessam é objetivo dos SGBDs. Esta propriedade é chamada de "independência entre dados e programas" ou simplesmente de **independência de dados**. Isto é caracterizado pela forte separação entre dados e programas, típicas de sistemas baseados em SGBDs.

Para satisfazer esta independência, o SGBD deve fornecer ao usuário uma representação conceitual dos dados: o modelo de dados. Este esconde detalhes de armazenamento que não interessariam ao usuário comum. **Independência de dados física** é a habilidade de se modificar a organização interna dos dados, tal como índices ou leiaute de registros, sem alterar as aplicações.

Um SGBD também deve ser capaz de fornecer, estendendo esta representação conceitual, visões ou perspectivas dos dados específicas para usuários distintos. **Independência de dados lógica** é a habilidade de se modificar o esquema dos dados - por exemplo, acrescentando-se atributos ou coleções - sem modificar as aplicações.

SGBDs são apropriados para o controle de redundância, o compartilhamento dos dados por múltiplos usuário, o controle do acesso concorrente dos dados por estes usuários, a restrição ao acesso a dados não autorizado, a manutenção da integridade dos dados, e a recuperação de falhas de hardware ou software.

C. J. Date ressalta que os dados mantidos pelos SGBDs são "dados operacionais", não incluindo "dados de entrada ou de saída, filas de trabalho e qualquer informação puramente transiente" [DATE90] que surgem no decorrer da execução das aplicações. As linguagens de programação, com as quais as aplicações normalmente são escritas, permitem a criação de estruturas de dados em uma variedade de tipos muito maior do que a existente nos modelos de dados de SGBDs. Isto ocorre porque, se por um lado os SGBDs devem se preocupar prioritariamente com a visão conceitual da informação, as linguagens de programação devem ser capazes de explorar todas as idiossincrasias possíveis na exploração da representação da informação na máquina.

Esta divisão de tarefas entre sistemas de programação e sistemas de gerência de bases de dados, ambos capazes de representar estruturas de informação, se reflete no fato de que os SGBDs não prescindem, na prática, das linguagens de programação para a criação de aplicações. A visão integrada dos dados, operacionais ou não, é tentada, por exemplo, pelas Linguagens de Programação com Banco de Dados (LPBDs).

Uma linguagem de consulta a banco de dados usualmente não permite que qualquer computação arbitrária possa ser realizada [ATKI87]. Por exemplo, o usuário não pode definir uma função para calcular a variância de um conjunto de números. A solução comumente adotada, nesses casos, é a utilização de um "sistema de quarta geração", que fornecem um pacote de alternativas de desenvolvimento cobrindo a maior parte das aplicações de bancos de dados. Quando

o sistema não provê a função desejada, o usuário acaba tendo que utilizar uma linguagem "embutida"[6].

Bancos de dados, quando utilizados em ambientes de desenvolvimento de software, têm demonstrado ser um recurso de grande valor. Entretanto, a tecnologia corrente de banco de dados (sistemas de gerência e modelos de dados) mostra-se ainda inadequada para suportar alguns dos requisitos de alguns tipos de ambientes de desenvolvimento. Um exemplo típico desta inadequação está na dificuldade dos bancos de dados correntes em suportar mudanças incrementais nos dados (e procedimentos) armazenados.

Um SGBD com linguagem de consulta incorporando procedimentos, ou com outras facilidades, pode ser chamado de **sistema de banco de dados estendido** [CATT91]. Tal tipo de SGBD fornece dois ambientes distintos: a linguagem de programação de aplicações e a linguagem de consulta estendida. Embora cada uma dessas linguagens possa fazer chamadas à outra, elas têm sistemas de tipos e ambientes de execução diferentes. Exemplos de **sistemas de banco de dados relacionais estendidos** são o POSTGRES [STON86], ALGRES [CERI90] e Starburst [SCHW86]. Suportando modelos de dados relacionais estendidos para não-primeira-forma-normal (NF^2) tem-se o sistema ODMS [DADA86] e R^2D^2 [KEMP88]. Muitas características desses sistemas já eram encontradas no RM/T [CODD79].

2.2.2. Sistemas de Gerência de Bases de Dados Orientadas a Objetos

Com relação a sistemas de gerência de bases de dados (SGBDs), pode ser feito um paralelo entre o que está ocorrendo no início desta década de 90 e o que ocorria no início dos 80s. As atenções dadas aos sistemas relacionais, naquela época, correspondem hoje ao enorme esforço em pesquisas de sistemas de gerência de bases de dados orientados a objetos. Do mesmo modo, surgem no mercado os primeiros produtos resultantes dessa pesquisa e as principais empresas de software e fabricantes de computadores demonstram interesse pela nova tecnologia.

Um número cada vez maior de tarefas humanas tem passado a contar com computadores e bancos de dados. Usando os sistemas relacionais, diversos tipos de aplicações esbarraram na rigidez de estruturação da informação em tabelas própria dos bancos de dados relacionais. Muitos dados hoje em dia manipulados por computadores não são naturais sob a forma de tabelas. Um considerável esforço é dispendido para transformá-los em um conjunto de tabelas ade-

---

[6]*Embedded language* em inglês.

quadas ao modelo relacional. Atividades de projeto, como a engenharia mecânica ou mesmo a engenharia de software, são exemplos de aplicações onde esses problemas são encontrados. Enxergar um automóvel ou um programa como tabelas pode ser bom para os computadores, mas não para nós, seres humanos.

Observa-se também que os SGBDs convencionais são usados para a gerência de uma quantidade muito maior de dados do que de metadados (dados sobre os dados originais). Ou seja, há muito menos tipos de dados do que exemplares de dados destes tipos. Além disso, é a baixa frequência de alteração desses metadados (esquema, descrição dos tipos de dados). Na atividade de desenvolvimento de software, por exemplo, isto é invertido; novos tipos de dados estão sempre sendo criados, mas poucos tipos de dados terão muitos exemplares.

Outra questão também vem sendo encarada nessas pesquisas. Trata-se de transferir parte do código das aplicações para dentro do próprio banco de dados. Parte-se do fato que uma grande parte das vezes uma informação é tratada sempre pelas mesmas rotinas. Essa parece ser uma evolução natural dos SGBDs, que surgiram retirando das aplicações a responsabilidade pela manipulação dos dados.

Embora haja propostas sobre que características um Sistema de Gerência de Bases de Dados Orientadas a Objetos (SGBDOO) deva possuir, não está absolutamente claro e consensual o que um SGBDOO venha a ser [KIM90b]. Neste trabalho designaremos por SGBDOO um SGBD cujo modelo de dados seja orientado a objetos. O principal a observar nesta definição é que, sendo um SGBD, o seu objetivo é gerenciar dados operacionais.

Os sistemas que implementam o paradigma da orientação a objetos frequentemente definem objeto como uma unidade de informação composta por dados (que representam o estado corrente do objeto) e procedimentos (que formam o comportamento deste). Tendo isto em vista, cabe a questão: um SGBDOO gerencia dados, dados e procedimentos, ou objetos? Como, pelos mesmos motivos, é vago o que seja um "modelo de dados orientado a objetos", qualquer das opções acima podem ser a base de um SGBDOO. O importante é que a sua preocupação fundamental seja gerir dados operacionais.

2.2.2.1. Sistemas de Gerência de Bases de Dados Estruturalmente Orientadas a Objetos

Muitas experiências têm sido realizadas visando criar sistemas de gerência de bases de dados orientados a objetos (SGBDOOs) [ATKI90]. Inicialmente se concentravam na adoção à estruturação dos dados dos conceitos

típicos da orientação a objetos: identidade implícita de objetos, classificação, herança de propriedades (estruturais), objetos compostos e polimorfismo. Estes SGBDOOs não se preocupam com os aspectos comportamentais da informação. A manipulação dos dados é realizada, nestes sistemas, através de comandos de consulta genéricos, isto é, válidos para qualquer tipo de dado (ex.: select, etc.). Não há a possibilidade de se definir operações específicas para um tipo de dado, o que seria característico da orientação a objetos. Sendo a manipulação de dados não orientada a objetos, estes SGBDOOs podem ser chamados de "estruturalmente orientados a objetos" em contraposição aos SGBDOOs "comportamentalmente orientados a objetos" [DITT86a].

Muitos desenvolvedores de SGBDs orientados a objetos só se preocupam com os aspectos estruturais da informação, tratando de oferecer estruturas mais ricas que as simples tabelas do modelo relacional. Não se preocupam em armazenar rotinas e dados conjuntamente. Eles se aproveitam de conceitos elaborados pelas pesquisas em modelos semânticos de dados, incluindo o modelo entidade-relacionamento [CHEN76]. Nestes se incluem o Infoexec SIM da Unisys, o IRIS da Hewlett-Packard, o ZIM da Sterling Software e o Adabas Entire da Software AG. Estes sistemas são chamados de bancos de dados estruturalmente orientados a objetos. Outros caminham pela simples modificação de algum sistema relacional, gerando "sistemas relacionais estendidos". Extensões ao modelo relacional se encontram no Supra da Cincom e no Ingres da RTI.

Os que estão tratando dos aspectos comportamentais da informação, ou seja, das rotinas que manipulam os dados, são os que obtêm maior benefício da orientação a objetos. Conceitos como encapsulamento, herança e polimorfismo são aplicados neste caso.

2.2.2.2. *Sistemas de Gerência de Bases de Dados Comportamentalmente Orientadas a Objetos*

Atualmente, a maioria dos SGBDOOs, além da estruturação dos dados, preocupa-se com aspectos comportamentais da informação, encapsulando nos objetos procedimentos e métodos, podendo serem chamados de SGBDOOs comportamentais [DITT86a] ou "comportamentalmente orientados a objetos". Estes tiram maior proveito da orientação a objetos. Vários autores (por exemplo, [ZDON89]) não consideram um SGBD como orientado a objetos a menos que encapsule a semântica do objeto).

2.2.2.3. *Incorporando Comportamento*

O encapsulamento define que um objeto reúne estrutura e comportamento (rotinas que se aplicam a ele). Isto significa que objetos distintos podem ter comportamento distinto. Uma rotina que se aplica a um objeto pode não ter sentido para um outro. Cada objeto tem seu próprio conjunto de rotinas que o manipulam. É o oposto do modelo relacional, onde só se tem um conjunto de operações totalmente genéricas, que atuam sobre qualquer relação e qualquer relação pode ser usada nestas operações.

Os modelos semânticos de dados já traziam o conceito de herança, algumas vezes com o nome de generalização/especialização. Porém só a estruturação da informação se beneficiava. Por exemplo, como os gerentes de uma empresa devem ser também funcionários, a estrutura da informação sobre qualquer gerente inclui as informações dele como funcionário. Ou seja, pelo fato de gerente ser uma especialização de funcionário, ele herda todos os atributos e relacionamentos deste.

Nos sistemas orientados a objetos comportalmente, trata-se agora de aplicar o conceito de herança ao comportamento das informações. A um gerente pode ser aplicada qualquer operação válida a um funcionário.

Os tipos de entidades representados no banco de dados são organizados em uma hierarquia, onde entidades mais especializadas herdam propriedades estruturais e comportamentais das entidades mais genéricas. A idéia útil de se ter operações aplicáveis a todos os objetos, como ocorre no modelo relacional, não é perdida: definindo-se uma entidade mais geral que todas, as operações aplicáveis a ela serão aplicáveis a todas as outras entidades.

O polimorfismo é a terceira característica da orientação a objetos que vem sendo empregada nesses SGBDs. Com ele, a mesma operação pode executar rotinas diferentes, dependendo do tipo do objeto no qual ela foi aplicada. Isto permite se ter um nível maior de abstração, concebendo-se operações genéricas válidas para entidades totalmente diversas, mas com comportamento distinto para cada entidade. Exemplos triviais dessas operações são: imprimir, duplicar, remover, etc.

As linguagens orientadas a objetos demonstraram que estas propriedades permitem a criação de módulos altamente reutilizáveis. O desenvolvimento de software se torna muito mais rápido.

2.2.3. Sistemas de Banco de Dados Orientados a Objeto

O termo Sistema de Banco de Dados Orientado a Objetos (SBDOO) tenta ser esclarecido no "The Object-Oriented Database System Manifesto"

[ATKI90]. Ele descreve as características obrigatórias, opcionais e possíveis para que um sistema seja assim classificado. Algumas características obrigatórias atendem ao critério que diz que um SBDOO deve ser um SGBD: persistência, gerência de memória secundária, concorrência, recuperação e consultas ad hoc. Outras características procuram uma consistência entre os SBDOOs e linguagens de programação orientadas a objetos: identidade de objetos, encapsulamento, tipos ou classes, herança, ligação dinâmica, extensibilidade, objetos complexos e completeza computacional (capacidade de realizar as mesmas operações que uma linguagem de programação pode realizar).

A definição de que um SBDOO seria um SGBD baseado em um modelo de dados orientado a objetos só seria válida se se ampliasse o conceito de modelos de dados. Definindo-se SBDOOs pelas características acima, ao invés de defini-los pelos objetivos a cumprir, permite dizer que os SGBDOOs, conforme definidos na seção anterior, são SBDOOs. O contrário não precisa ser verdadeiro. Isto porque um SBDOO pode ter objetivos mais amplos que a gerência de dados operacionais, em virtude da sua completeza computacional.

Nos sistemas não orientados a objetos, dados são consultados usando-se linguagens de consulta e atualizados frequentemente por aplicações escritas em linguagens imperativas com comandos da Linguagem de Manipulação de Dados embutidos ou em uma Linguagem de Quarta Geração. Estas aplicações são mantidas em sistemas de arquivo tradicionais e não no banco de dados. O Manifesto acima alerta para a "aguda distinção entre programas e dados, e entre linguagem de consulta (ad hoc) e linguagem de programação (de aplicações)" em um sistema tradicional. Em um SBDOO, "tanto os dados como as operações seriam armazenadas na base de dados".

2.2.4. SGBDs de Terceira Geração

Outro manifesto proclama as características dos Sistemas de Banco de Dados de Terceira Geração [STON90]. Este termo, neste manifesto, é muitas vezes substituído por SGBDs de Terceira Geração. O que distingue fortemente este manifesto do anterior é o suposto que os novos sistemas serão uma evolução dos SGBDs e, portanto, serão predominantemente SGBDs. Neste caso, por exemplo, a completeza computacional não é dada pelo SGBD de terceira geração em si, mas pelo fato dele permitir acesso a múltiplas linguagens de alto-nível. Linguagens de programação com persistência são consideradas úteis e uma "grande idéia" pelo manifesto; todavia devem ser construídas no topo de um SGBD de terceira geração.

A primeira geração dos sistemas de banco de dados corresponde aos sistemas hierárquicos e de rede que prevaleceram nos anos 70. Na década seguinte,

os SGBDs relacionais suplantaram aqueles, estabelecendo uma segunda geração de sistemas de banco de dados. Um maior grau de independência de dados e o uso de linguagens de manipulação de dados não procedimentais foram os principais avanços dessa segunda geração. A terceira geração, de acordo com este manifesto, está surgindo, entre fabricantes e prototipadores de SGBDs avançados, da convergência para um consenso sobre quais extensões e melhorias são desejáveis.

Entre as características fundamentais dos SGBDs de terceira geração e vista apenas como opcional pelos SBDOOs está o suporte a regras e inferência. O manifesto estima que caberá aos sistemas com ausência destas facilidades apenas nichos do mercado. Por estas características, alguns protótipos [SHYY91], que anteriormente seriam classificados como Sistemas de Gerência de Bases de Conhecimento, agora têm se denominado SGBDs de terceira Geração.

O manifesto da terceira geração preconiza uma estruturação da informação mais rica que a dos modelos relacionais atuais, basicamente permitindo um conjunto rico de construtores de coleções. A extensibilidade seria possível a nível dos atributos elementares das entidades, pois tipos abstratos de dados poderiam estender o conjunto de tipos básicos do sistema (os domínios, do modelo relacional). Já os construtores de coleções seriam fixados pelo sistema, não sendo, portanto, uma coleção vista como um objeto. Todavia, poder-se-ia associar funções escritas em uma linguagem de alto-nível a uma coleção. Isto permitiria que a manipulação de um elemento de uma coleção fosse encapsulada nestas funções. A estrutura dos elementos de uma coleção, suas funções e regras seriam herdadas através de uma hierarquia de coleções. A herança múltipla é considerada essencial, enquanto nos SBDOOs é apenas opcional[7]. Identificadores de registros (sic) seriam designados pelo SGBD apenas quando não houvesse chave primária que pudesse exercer este papel.

As divergências mais importantes em relação aos modelos orientados a objetos estão na não exigência de que os sistemas de banco de dados de terceira geração sejam completos computacionalmente e de que o acesso ao SGBD atra-

---

[7]Talvez fosse de se esperar que herança múltipla fosse uma característica mais defendida por quem se denomina "orientado a objetos". Isto não ocorre porque os SBDOOs, ao sofrerem mais influência das linguagens de programação orientadas a objetos, deram mais atenção à implementação de métodos do que à modelagem conceitual da informação. Esta, mais ligada à área de bancos de dados, tira um proveito maior da herança múltipla, do que na implementação de software, onde a herança simples tem se mostrado suficiente

vés de mensagens com ligação dinâmica[8] deve ser evitado. Basicamente, os SGBDs de terceira geração devem ter essencialmente todo acesso aos dados persistentes através de linguagens de consulta não procedimentais, entre elas extensões à SQL. Além disso, os novos SGBDs devem ser abertos a múltiplas linguagens de alto-nível com persistência ou com linguagens de consulta embutidas. Uma linguagem de programação orientada a objetos com persistência baseada em um SGBD de terceira geração poderia ser visto como um Sistema de Banco de Dados Orientado a Objetos (SBDOO) conforme definido na seção anterior.

Por último cabe ressaltar que as funções tradicionais de um SGBD devem ser mantidas como essenciais na nova geração de sistemas. Seu manifesto destaca como facilidades que devem estar obrigatoriamente presentes extensões ao conceito de vistas: vistas atualizáveis e coleções virtuais, isto é, definidos intensionalmente.

2.2.5. Linguagens com Persistência

Nas linguagens de programação comuns, os valores manipulados, em geral, têm um tempo de vida definido implicitamente pelo programa e limitado ao tempo de execução do programa. Valores que perdurem após a execução do programa, ditos persistentes, são nitidamente diferenciados, não havendo uniformidade entre estes e os tipos de valores comuns. Em Pascal, por exemplo, o tipo de dados *file* é empregado para indicar a intenção de que determinados valores persistam fora da execução do programa. Um dado *file* pode conter valores de um tipo de dados designado, porém nem todos os tipos de dados (por exemplo, tipos enumeração) são permitidos.

Uma linguagem de programação onde a persistência é transparente e possível para qualquer tipo de dado é chamada de linguagem de programação com persistência ou **linguagem de programação persistente** [ATKI87]. Exemplos dessas são PS-Algol e Trellis [SCHA86a] [SCHA86b].

Algumas linguagens de programação estendem o alcance da persistência além da execução de programas permitindo o armazenamento de sua memória de trabalho. Este é o caso de Smalltalk, Lisp e Prolog. Pode-se assumir que cada uma destas linguagens manipula uma base de objetos, listas ou predicados, respectivamente. Todavia, o acesso concorrente a essas bases por diferentes usuários não é resolvido por esta abordagem. O problema da persistência após a exe-

---

[8] *Late Binding*, em inglês.

cução de um programa é resolvido, mas fica restando a persistência após a utilização por um usuário.

Alguns **gerenciadores de objetos**, também chamados de **servidores de objetos**, têm sido construídos para fornecer um repositório para objetos persistentes em linguagens de programação. Estes gerenciadores são pacotes que estendem o sistema de arquivos e nem sempre fornecem linguagens de consulta. Exemplos de gerenciadores de objetos incluem POMS [COCK84], Mneme [MOSS88] [MOSS90] e LOOM [KAEH83].

2.2.6. Linguagens de Programação de Banco de Dados

"Functions, including database procedures and methods, and encapsulation are a good idea" [STON90]

Sistemas de gerência de bases de dados de terceira geração devem permitir a definição de funções específicas a determinados tipos de dados. Isto traz à tona o problema de incompatibilidade ("impedance mismatch") [SHYY91] entre linguagens de programação e linguagens de banco de dados. O acoplamento entre linguagens de progamação e linguagens de banco de dados vem sendo resolvido através de extensões às linguagens ou através da criação de biblioteca de funções ou de classes. A diferença conceitual entre os dois tipos de linguagens, com tipos de dados e sintaxes distintas, complica a tarefa de acessar um banco de dados a partir de linguagens de programação.

Linguagens de programação de banco de dados (LPBDs) [BANC90], projetadas de um modo integrado para a programação de sistemas e para o acesso adequado a bases de dados persistentes, são a resposta para este problema. Em uma LPBD, o acesso ao banco de dados é transparente, e os dados armazenados são definidos em estruturas conforme o modelo de dados adotado. As linguagens Pascal/R e Modula/R servem como exemplo de uma "LPBD integrada" por tentar combinar uma linguagem de programação e um modelo de dados ambos existentes. Uma LPBD, indo além de uma linguagem de programação tradicional, deve fornecer persistência, controle de concorrência e outras facilidades de bancos de dados.

2.2.7. Sistemas de Gerência de Interfaces com Usuário

Os SGBDs atuais fornecem facilidades limitadas para a apresentação e manipulação dos dados para o usuário, em geral, sob a forma de tabelas, formulários/fichas e alguns tipos de gráficos. Além disso, a utilização de gráficos está restrita à apresentação de dados para o usuário, não sendo permitindo manipular diretamente [SHNE83] os gráficos.

Tradicionalmente a gerência da comunicação com o usuário é parte integrante de cada aplicativo interativo. Todavia, a abordagem tradicional tem sérias deficiências:

. As linguagens de programação fornecem apenas facilidades primitivas; geralmente, apenas formatação na entrada/saída. Formulários, verificação mais complexa dos dados entrados, opção de desfazer o que foi realizado pelo usuário e qualquer coisa um pouco mais sofisticada têm que ser implementadas pelo programador.

. Instruções para a realização da interação com o usuário são misturadas com as demais instruções do programa. Fica difícil separar as funções da comunicação com o usuário das funções próprias da aplicação. A consistência entre diversos aplicativos fica prejudicada.

. Aplicações fortemente interativas não devem obrigar o usuário a agir seqüencialmente; as linguagens de programação quase que forçam que a interação com o usuário tenha caráter seqüencial.

. A independência de dispositivo também fica por conta do programador.

Programadores acabam sendo os responsáveis pelo projeto da interface. Raramente eles têm conhecimento dos fatores de comunicação necessários, nem estabelecem contatos com os usuários finais.

A comunicação homem-computador (CHC) tem sido assunto de vários "workshops". Um destes, em 1982, definiu um conceito revolucionário na área: os Sistemas de Gerenciamento de Interface com o Usuário (SGIUs) [THOM83].

2.2.7.1. Definição

O termo Sistema de Gerência de Interface com Usuário (SGIU), embora muito utilizado, não tem uma definição aceita por todos. Para todos os efeitos, pode-se encarar um SGIU como sendo um conjunto de ferramentas e técnicas para se construir e controlar a comunicação homem-computador das aplicações. Um SGIU gerencia a interação entre a comunicação homem-computador e o restante da aplicação. Assim como um SGBD retira das aplicações a administração dos dados operacionais, um SGIU delas exclui a administração da interação com usuário. Com isto, objetiva-se o aumento da produtividade na construção de programas [BETT87].

Um SGIU se responsabiliza tanto pelo desenvolvimento quanto pela operação da comunicação homem-computador [COUT85]. Diversos SGIUs visam a prototipagem rápida da comunicação homem–computador e que as interfaces com usuário geradas sejam mais fáceis de serem criadas e mais baratas.

2.2.7.2. *Características*

Um SGIU caracteriza-se, principalmente, pela proposta de independência da interface com a aplicação. Um SGIU encoraja a separação da atividade de projeto da comunicação homem-computador do projeto da aplicação. O SGIU é um conjunto de ferramentas que [BETT87]:

- suporta a definição do diálogo usuário/aplicação;
- impõe controle externo;
- controla (gerencia) a execução da interação usuário/aplicação;
- fornece independência da interface com a aplicação.

O controle externo diz respeito ao controle da interface com o usuário sobre a aplicação. Nele, é a interface que faz chamadas à aplicação. No controle interno (convencional, sem SGIU), a aplicação é que chama a interface quando precisa se comunicar com o usuário.

Os Sistemas de Gerência de Interfaces com Usuário (SGIUs) são compostos para gerenciar a criação e a execução de interfaces com usuário, de um modo independente do restante da aplicação. Os problemas centrais abordados pelos SGIUs são a modelagem e construção de interfaces com usuário e a modelagem do relacionamento delas com as aplicações propriamente ditas.

2.2.7.3. *Vantagens da Utilização de SGIUs*

Os Sistemas de Gerência da Interface com o Usuário (SGIUs) pretendem liberar o programador de aplicativos dos detalhes de baixo-nível, para que ele possa se concentrar nos aspectos específicos da comunicação homem-computador da aplicação. Incluindo, comumente, um pacote de ferramentas para implementação, depuração e avaliação de diálogos homem-máquina, um SGIU procura aumentar a iteratividade do processo de se projetar a comunicação homem-computador.

Permitindo a implementação da comunicação homem-computador independentemente da implementação da aplicação em si, um SGIU fornece meios para a construção de protótipos voltados ao usuário, antecipando a participação deste para fases iniciais do desenvolvimento de um sistema.

A independência entre a funcionalidade da aplicação e o desenvolvimento da comunicação homem-computador também propicia a experimentação de diversas alternativas de interfaces, já que o código da aplicação em si não necessitaria ser alterado.

Um mesmo aplicativo pode ser desenvolvido com interfaces diferentes para usuários de tipos distintos. O caso típico dessa situação é o de um progra-

ma exportável para países de línguas diferentes. A independência da interface em relação a funcionalidade novamente permite a alteração da aparência de uma aplicação sem que se desvirtue o código do programa. O corolário dessa propriedade é a possibilidade de se produzir mais facilmente interfaces adaptativas (que variem dinamicamente conforme o uso).

Para os projetistas da comunicação homem-computador, algumas vezes chamados de *arquitetos da comunicação homem-computador* [FOLE84], que não necessariamente são programadores, o SGIU pretende que a comunicação homem-computador obtenha as seguintes vantagens:

(1) consistência entre a comunicação homem-computador de diversas aplicações relacionadas
(2) consistência na comunicação homem–computador de uma aplicação.
(3) alterabilidade
(4) reusabilidade
(5) independência da aplicação com relação a sua comunicação homem-computador
(6) facilidade de aprendizado
(7) facilidade de uso

Para os programadores de aplicações, um SGIU:

(1) suporta a definição da apresentação dos dados saídos da aplicação
(2) controla (gerencia) a execução da interação entre o usuário final e a aplicação.
(3) fornece independência da aplicação com relação a sua comunicação homem-computador
(4) oferece as seguintes características ao desenvolvimento da comunicação homem-computador:
   - consistência
   - suporte a todos os tipos de usuários, do inexperiente ao experiente
   - suporte a manipulação e recuperação de erros
   - concentração na especificação ao invés de codificação, como em uma linguagem de 4a geração [BETT87]

Embora os usuários finais não precisem perceber a existência de um SGIU entre eles e as aplicações, eles têm as seguintes vantagens:

(1) facilidade e efetividade de uso das aplicações
(2) consistência entre a comunicação homem-computador de diversas aplicações relacionadas

(3) diversos tipos de ajuda
(4) treinamento
(5) ajustabilidade da comunicação homem-computador

Os SGIUs procuram proporcionar as seguintes vantagens no desenvolvimento de aplicações [ENDE84]:

(1) O programador da aplicação não se preocuparia mais com coisas como formatos das saídas e verificação dos valores entrados, se ocupando mais com a produção correta do software.
(2) O projetista da comunicação homem-computador da aplicação se ocuparia especificamente com os aspectos que aumentariam a produtividade do usuário.
(3) O projetista da comunicação homem-computador da aplicação poderia prototipar e implementar rapidamente interfaces com usuário.
(4) A comunicação homem–computador resultante teria uma qualidade maior pois poderia ser o resultado de diversas interações com os usuários.
(5) A mesma aplicação poderia rodar com diferentes interfaces com usuário e em diferentes estações de trabalho.
(6) Diferentes dispositivos e técnicas de interação podem ser usados.
(7) Poderiam ser experimentadas diversas interfaces com usuário diferentes para a mesma aplicação.
(8) A interface com usuário de uma aplicação poderia ser ajustada pelos usuários sem prejuízo para a aplicação.
(9) Estimularia a reusabilidade de módulos de interação por mais de um sistema.
(10) Seria mais fácil especificar, validar e avaliar uma comunicação homem–computador.
(11) A confiabilidade da interface com usuário resultante seria maior, pois partiria de especificações de alto nível.
(12) A manutenção das aplicações interativas seria mais rápida e econômica.

2.2.7.4. Histórico

Algumas idéias que originaram os SGIUs já se encontravam presentes em sistemas anteriores ao uso do termo. Em 1968, William Newman descreveu o Reaction Handler, contendo diversas propriedades do que hoje é conhecido como SGIU. Até 1982, viveu-se a fase dos "SGIUs pioneiros" [HILL87a]. Em 1982, no *Workshop* sobre Técnicas de Entrada e Interação Gráfica ("GIIT -

Graphical Input and Interaction Techniques") [THOM83] foram discutidos os conceitos básicos do que seria então chamado de SGIU. No SIGGRAPH'82, Kasik publica o artigo "**A User Interface Management System**" (Um Sistema de Gerência de Interfaces com Usuários).

Outro *workshop*, desta vez em Seeheim na Alemanha, estudou, principalmente, modelos e estruturas, dando origem ao chamado modelo de Seeheim, definindo os componentes lógicos de um SGIU [GREE85a]. Diversos problemas complexos foram levantados com relação aos SGIUs.

Em novembro de 1986, realizou-se o ACM SIGGRAPH Workshop on Software Tools for User Interface Management, em Seattle. Os SGIUs foram rediscutidos diante da nova realidade (proliferação de ferramentas para interface com usuários baseadas em manipulação direta). A questão base era: por que os SGIUs não são bem sucedidos comercialmente? [LOWG88].

2.2.7.5. *Exemplos*

Alguns exemplos de SGIUs clássicos seriam o ADM [SCHU85], o Menulay [BUXT83], o HutWindows [KOIV88] e o UIMS da Universidade de Alberta [GREE85b]. Muitos SGIUs têm sido implementados recentemente baseados na orientação a objetos, freqüentemente em Smalltalk. Como exemplos destes têm-se o sistema de Grossman e Ege [GROS87], o HIGGENS [HUDS86a] e o IMAGES [SIMO87].

2.2.7.6. *Comunicação Homem-Computador com Manipulação Direta*

Em paralelo com essas novas idéias, a área de comunicação homem-computador evoluiu em outra direção: para o desenvolvimento de Conjuntos de Ferramentas ("toolboxes") de Interação com o Usuário por Manipulação Direta.

Numa interface com manipulação direta, o usuário tem à sua disposição ícones, janelas e cardápios, para representar objetos e efetuar operações manipulando-os diretamente. Assim, a cópia de um arquivo X, para uma unidade de disco A, pode ser realizada movendo, com o *mouse*, o ícone que representa o arquivo X até o ícone representando a unidade de disco A, onde o arquivo é colocado. Para mover um objeto com o *mouse*, move-se o apontador (seta) do *mouse* para cima do objeto, aperta-se um botão do *mouse* (segurando o objeto), move-se o *mouse* (arrastando consigo o objeto) e solta-se o botão (largando o objeto).

2.2.7.7. *Questões Atuais*

Embora a proposta dos SGIUs não seja novidade, eles não têm encontrado sucesso, ao contrário de sistemas voltados ao suporte de execução de interfaces com usuário com manipulação direta [APPL85] [X11 87]. Dentre as razões apontadas [MYER87], está a dificuldade na própria utilização destes sistemas quando eles exigem, como é comum, habilidade de programação para se especificar interfaces com usuário. Também são mencionadas a dificuldade de portabilidade para novas máquinas, sistemas operacionais e sistemas gráficos [GREE87]. Isto ocorre por não haver padronização adequada em serviços que os SGIUs usam, tais como a gerência de dispositivos de entrada (*mouse*, teclado, ...) e a execução de gráficos nos dispositivos de saída.

Há quem se refira a ausência de técnicas de engenharia de software adequadas a produtos de software gerado [OLSE87a] pelos SGIUs ou que os utilizem. Isto provocaria, inclusive, dificuldade de manutenção dos produtos. Como uma aplicação grande utiliza outras ferramentas além de um SGIU (SGBDs, comunicação de dados, ...), um modelo que unifique o uso de diversas ferramentas contribuiria para aumentar a portabilidade e a facilidade de desenvolvimento.

A limitação dos SGIUs quanto aos tipos de interfaces que eles podem criar é, também, um dos motivos mais importantes da sua falta de popularidade. A maioria dos SGIUs existentes não é apropriada para gerar comunicação homem-computador de estilo manipulação direta.

Vários SGIUs foram construídos, porém o despontar da comunicação homem-computador por manipulação direta atropelou estes primeiros sistemas. O que ocorre é que a manipulação direta não se encaixa facilmente com a idéia base dos SGIUs: a separação entre a funcionalidade de uma aplicação e a sua comunicação com o usuário. Isto porque é necessário que o indivíduo manipule diretamente, através dos objetos da interface com o usuário do programa, os objetos da aplicação.

Para que se crie a ilusão de que a representação de um objeto da aplicação *é* o próprio objeto, nem só as ações sobre os objetos exclusivos da interface com o usuário, como cardápios e janelas, devem proporcionar realimentação ao usuário. A realimentação pelas ações sobre os objetos da própria aplicação é chamada de realimentação semântica. Este é o caso de, por exemplo, um objeto sendo arrastado com o *mouse*, mas se movimentando aos saltos para lugares determinados pela aplicação, como no caso de reticulados *("grids")*. Isto exige que a interface com usuário faça acesso às estruturas de dados da aplicação. A pesquisa em SGIUs hoje está voltada para a resolução de tais problemas.

Em face do desenvolvimento dos sistemas de comunicação homem-computador com manipulação direta, algumas questões sobre SGIUs vieram à tona [OLSE87b]:

- Pode uma abordagem SGIU responder às necessidades da comunicação homem-computador com manipulação direta?
- Como um SGIU pode adaptar-se aos sistemas operacionais que usam janelas gráficas como um modelo para controle de processos múltiplos?
- Por que não há mais SGIUs sendo comercializados?
- Qual é o relacionamento entre um SGIU e um Conjunto de Ferramentas para Interação com o Usuário?
- Os SGIUs se propunham a resolver problemas diferentes da realidade?
- Como expandir a capacidade e o poder do software de interface com o usuário?

Outra questão é o compromisso entre a abordagem por manipulação direta e o máximo de independência entre a aplicação e a interface com usuário.

2.2.7.8. Requisitos para Manipulação Direta

SGIUs que atendam a comunicação homem-computador com manipulação direta precisam [MYER87]:

- suportar tipos diferentes de técnicas de interação;
- ter a capacidade de criar novos tipos de técnicas de interação;
- permitir métodos de manipulação direta para especificar a própria interação graficamente;
- ter uma linguagem fácil de usar para programar as partes que não podem ser especificadas graficamente;
- permitir que múltiplas técnicas de interação estejam disponíveis para o usuário ao mesmo tempo.
- permitir que múltiplas técnicas de interação estejam operando ao mesmo tempo.
- ter desempenho satisfatório ao fornecer realimentação, verificação de erros e defaults semânticos.

2.3. Sistemas de Objetos e Ambientes de Desenvolvimento de Software

Conforme mecionado na Seção 1.2., um ambiente de desenvolvimento de software pode se apoiar em um SGBD para gerenciar os dados persistentes e os dados compartilhados por diversos usuários. De modo semelhante, um SGIU poderia controlar toda a interação com usuário no ambiente. O desenvolvimento e a execução das ferramentas e aplicações do ambiente de desenvolvimento de software poderiam utilizar o sistema operacional, linguagens de programação e o próprio ambiente de desenvolvimento de software. A construção de ferramentas utilizáveis no ambiente de desenvolvimento de software seria complicada pela mencionada incompatibilidade entre todos esses gerenciadores.

Entre as aplicações que motivam a pesquisa em SGBDs orientados a objetos, a Engenharia de Software está entre as mais citadas. A Engenharia de Software Auxiliada por Computador (*CASE*) é a aplicação de computadores para auxiliar o desenvolvimento e a manutenção de software. Em [CATT91] é apresentado um estudo do processo de engenharia de software, das ferramentas *CASE* projetadas para auxiliar esse processo e do papel de SGBDs orientados a objetos no suporte a essas ferramentas.

A abordagem proposta por esta tese supõe que a implementação e processamento de ferramentas, as informações por elas manipuladas e a sua interação com usuário sejam gerenciadas por um único sistema, um Sistema de Objetos (SOO), baseado em um modelo de representação da informação que integre todos estes aspectos.

Com um sistema de objetos, todas as ferramentas, aplicações e os próprios ambientes de desenvolvimento de software nada mais seriam que composições de objetos gerenciados pelo SOO. Isto não significa que o SOO faça o papel de ambiente de desenvolvimento de software. Um SOO gerencia bases de objetos, incluindo suas interfaces com usuário, mas não suporta diretamente qualquer metodologia de modelagem de informação, de implementação de programas ou de desenvolvimento da comunicação homem-computador. Pelo contrário, estas tarefas devem ser realizadas com ajuda de ferramentas específicas, que não fazem parte do SOO, embora possam ser gerenciadas por ele.

2.4. Características Funcionais de um Sistema de Objetos

Um Sistema de Objetos (SOO) é um sistema que permite aos usuários criar, manter e manipular bases de objetos. Os objetos de um SOO, simplificadamente, são dados acrescidos do seu comportamento e representação para usuário específicos. Por comportamento entenda-se o conjunto de

procedimentos de manipulação de um tipo de dado. Estes procedimentos podem criar e manipular objetos temporários, isto é, com tempo de vida restrito à ativação do procedimento, ou persistentes, que não estão submetidos a esta restrição.

O conceito de aplicação não é necessário em um SOO; uma aplicação é diluída em procedimentos de manipulação de objetos específicos e que são, juntamente com os objetos e suas interfaces com usuário, gerenciados pelo SOO. A representação para usuário de um objeto é um outro objeto capaz de interfacear o objeto original com o usuário, servindo para representar graficamente o estado do objeto base e para que o usuário possa alterar este estado ou ativar algum procedimento específico do objeto.

A inclusão de procedimentos como itens gerenciados por um SOO não significa que a "independência entre dados e programas", característica dos SGBDs, possa ser perdida. Deve haver uma distinção entre procedimentos que manipulam fisicamente um objeto, denominados de métodos do objeto, e os que se atêm à representação conceitual do objeto. Estes últimos são independentes da implementação física dos dados.

O modelo de objetos de um SOO deve ser projetado para que, além de conter características de modelos de dados de SGBDs, para representar a estrutura conceitual dos objetos, ter o poder computacional das linguagens de programação. Desse modo o modelo de objetos poderá ser implementado por uma ou mais LPBD , não havendo necessidade de se recorrer a linguagens embutidas. Uma LPBD, sendo baseada no modelo de objetos, poderá ser usada para a realização de consultas *ad hoc*.

De maneira semelhante a um SGBD, um sistema de objetos também deve ser capaz de estender a representação conceitual dos objetos, fornecendo visões ou perspectivas destes específicas para usuários distintos. Note-se porém que, indo além dos modelos de dados de SGBDs, a representação conceitual de um objeto pode incluir, não só a estrutura conceitual dos dados do objeto, mas também os seus serviços, ou seja, a representação conceitual de seu comportamento.

Um SOO deve preservar dos SGBDs a habilidade de armazenar grande volume de dados estruturados e que persistam entre a execução de tarefas, o controle de redundância dos dados, o compartilhamento dos dados por múltiplos usuário, o controle do acesso concorrente dos dados por estes usuários, a restrição ao acesso a dados não autorizado, a manutenção da integridade dos dados, acesso remoto aos objetos e a recuperação de falhas de hardware ou software. Mais ainda, os procedimentos dos objetos, vistos como dados que podem ser (ou não) alteráveis, também devem se submeter a estes controles operacionais.

Particularmente a execução de procedimentos dos objetos deve ser submetida a mecanismos de autorização.

É tarefa de um SOO o armazenamento e recuperação de objetos, ou melhor dizendo, seus dados e procedimentos, em memória secundária. O modelo de objetos do SOO pode permitir a especificação de como um objeto é representado em um meio de armazenamento. Correlatamente, também cabe ao SOO a apresentação de objetos ao usuário nos meios de interface com usuário e o controle da interação entre eles.

Um SOO deve propiciar a separação entre os elementos semânticos de uma "aplicação" e a interface com usuário, sem que isto signifique impedimentos para a interação baseada na manipulação direta de objetos. A abordagem orientada a objetos é capaz de produzir soluções adequadas para esta questão.

Além disso, um SOO deve conservar as seguintes características:

- suportar tipos diferentes de técnicas de interação;
- ter a capacidade de criar novos tipos de técnicas de interação;
- permitir que um objeto possa ter múltiplas técnicas de interação.
- permitir que múltiplas técnicas de interação para um dado objeto estejam disponíveis ao mesmo tempo.
- permitir que uma técnica de interação possa ser reaproveitada para diferentes tipos de objetos.
- permitir que múltiplas técnicas de interação estejam disponíveis para o usuário ao mesmo tempo. O usuário não será obrigado a interagir com determinado dispositivo de cada vez, sendo possível escolher qual dispositivo será usado, a menos que a aplicação imponha o uso de um dispositivo específico.
- permitir que múltiplas técnicas de interação estejam operando ao mesmo tempo.
- permitir que múltiplos usuários possam interagir concorrentemente com os mesmos objetos.
- prover independência entre as técnicas de interação e a apresentação dos objetos para os usuários.
- prover independência entre as técnicas de interação e os dispositivos físicos de interface com usuário.
- propiciar a criação de interfaces com usuário que preservem a integridade dos dados.
- suportar o relacionamento entre os objetos e a representação simbólica visual destes. Exemplo: determinada ação sobre determinado objeto da comunicação homem-computador causará chamada à rotina do objeto adequada.

2.5. Sumário

Este capítulo introduziu o conceito de Sistemas de Objetos, discutindo o seu propósito, alternativas, características, sua função em um ambiente de desenvolvimento de software.

Assim como os SGBDs aliviam as linguagens de programação da complexidade da gerência dos dados persistentes e compartilhados, os sistemas de gerência de interface com usuário (SGIUs) se propõem a fazer o mesmo com relação à interação com usuário. Se, por um lado, linguagens de programação e bancos de dados não dispensam uns aos outros, por outro, o acoplamento entre eles não se dá com naturalidade. Semelhantemente, os sistemas de gerência de interface com usuário apresentam dificuldades de se relacionar com aplicações que utilizem comunicação homem-computador por manipulação direta. Um sistema de objetos (SOO) usa o paradigma da orientação a objetos para a unificação das gerências de dados, de interação com usuário e de aplicações, centralizando soluções de problemas comuns a diversas aplicações.

Um sistema de objetos é uma coleção de programas que permite aos usuários criar e manter bases de objetos. Um objeto de um SOO consiste de dados, comportamento e representação para usuário. Objetos podem ser persistentes ou ter seu tempo de vida restrito à ativação de um procedimento.

Um sistema de objetos deve se basear em um modelo de informação que integre a visão conceitual dos objetos com a sua implementação e interação com usuário. Isto sem que haja perda na independência entre as diferentes representações de um objeto.

Este capítulo se deteve nas características dos SOOs como sistemas, deixando para o próximo capítulo as características dos SOOs referentes a representação da informação. Estas formam o Modelo de Objetos do SOO.

Capítulo 3

Modelos de Objetos para Sistemas de Objetos

3.1. Introdução

3.1.1. Apresentação

Este capítulo apresenta modelos de objetos para Sistemas de Objetos, definindo o seu objetivo, estudando áreas com contribuições para a sua definição e listando as características básicas a serem atendidas por um modelo de objetos. As propriedades específicas do modelo de objetos do GEOTABA é assunto do capítulo seguinte.

3.1.2. Motivação

Modelo de objetos, neste trabalho, se refere a representação da informação em um sistema de objetos. O termo é derivado do conceito de modelo de dados, porém, diferentemente destes, não abrange apenas a representação conceitual dos dados e sua manipulação, mas também o comportamento inerente aos dados, em uma visão orientada a objetos, além de sua representação a nível de implementação e sua representação para interação com usuário.

Bancos de dados, linguagens de programação e sistemas de interface com usuário fornecem soluções parciais e não integradas para a representação da informação. Este capítulo inclui um resumo de como modelos de dados, linguagens de programação orientadas a objetos e sistemas de gerência de interface com usuário tratam essa questão e qual a dificuldade em integrar estas soluções.

3.1.3. Objetivo do Capítulo

O objetivo deste capítulo é enumerar quais características devem estar presentes em um modelo de objetos de um SOO e discutir outras características interessantes, porém não obrigatórias. O capítulo seguinte concretiza um modelo de objetos para o GEOTABA definindo as propriedades selecionadas para este.

3.1.4. Organização do Capítulo

A Seção 3.2. seguinte, apresenta como SGBDs, SGIUs e linguagens de programação orientadas a objetos abordam a representação da informação. A Seção 3.3. discute os tipos de conflitos que ocorrem entre estas representações parciais. A Seção 3.4. enumera as características esperadas de um modelo de objetos. Finalmente, a Seção 3.5. apresenta um sumário do capítulo.

3.2. Representação da Informação

Esta seção discute a questão da representação da informação em bancos de dados, linguagens de programação e sistemas de interface com usuário, visando obter subsídios para diversos aspectos de representação da informação em modelos de objetos.

A área de bancos de dados tem apresentado fortes resultados na representação conceitual de dados, expressa nos modelos semânticos de dados. A abordagem orientada a objetos trouxe a necessidade de se incorporar o comportamento associado aos dados à representação conceitual da informação. Quanto a implementação das estruturas de dados, ela tradicionalmente é embutida nos sistemas de gerência de bases de dados, onde não pode ser alterada. Os requisitos de extensibilidade de um SOO, porém, exigem que se possa criar novas estruturas e comportamento dinamicamente. Isto costuma ser responsabilidade de linguagens de programação. Dentre estas, as linguagens de programação orientadas a objetos têm demonstrado alcançar alto grau de alterabilidade e reusabilidade de software. Por fim, a representação de interfaces com usuário e do seu relacionamento com as aplicações (ou, no caso de uma visão orientada a objetos, com os objetos representados) é questão tradicionalmente tratada nos sistemas de gerência de interface com usuário.

Para abranger estas três áreas, esta Seção é dividida em três subseções. A primeira (3.2.1.) trata da implementação de estruturas e comportamento nas linguagens de programação orientadas a objetos. A segunda (3.2.2.) trata da representação conceitual de dados conforme tratada pelos modelos de dados usados para bancos de dados. A terceira subse-

ção (3.2.3.) aborda a modelagem de interfaces com usuário em sistemas de gerência de interface com usuário. Cada uma destas áreas dá uma visão parcial da representação da informação conforme abordada em um SOO. A dificuldade de integração entre estas visões parciais é o assunto da Seção 3.3. seguinte.

3.2.1. Programação com Objetos

O paradigma da orientação a objetos, aqui preconizado como o agente integrador dos diversos aspectos da representação de informação em um SOO, surgiu com as linguagens de programação orientadas a objetos. O progresso na área de linguagens de programação tem sido obtido pela criação de mecanismos de programação mais apropriados ao ser humano, permitindo a abstração de elementos voltados primordialmente ao processamento por computadores. As chamadas linguagens de programação orientadas a objetos (LPOOs) reúnem uma série desses mecanismos cujo objetivo é utilizar na tarefa de programar as mesmas habilidades naturais dos seres humanos. Noções intuitivas, como objetos, classes, estado, ação e reação, utilizadas pelo homem na compreensão de fenômenos do mundo real, são, nas LPOOs, empregadas para se gerar software que possa ser compreendido conceitualmente por estas mesmas noções. As linguagens mais citadas como tendo dado origem à programação orientada a objetos são Simula [DAHL66] e Smalltalk [GOLD83]. Outros exemplos são C++ [ELLI90], CLOS [BOBR88] e CLU [LISK77].

Esta capacidade de fornecer uma visão conceitual do software através de elementos de compreensão do mundo real aproxima as LPOOs dos modelos semânticos de dados e da área de banco de dados. Um modelo semântico de dados visa representar aspectos do mundo real, conforme ferramentas adequadas ao ser humano porém possíveis de serem mapeadas para um banco de dados. Outro aspecto de importância para os SOOs é a perspectiva unificadora de dados e procedimentos encontrada no paradigma da orientação a objetos. Por último, as LPOOs e as interfaces com usuário com manipulação direta estão, desde suas origens históricas, fortemente associadas. Na verdade, ambas se baseiam nos mesmos conceitos cognitivos [MONT92a].

Todavia não se pode encontrar uma definição aceita por todos do que venha a ser uma linguagem de programação orientada a objetos. Porém, constata-se que há linguagens ditas puramente orientadas a objetos e outras chamadas de linguagens de programação orientadas a objetos híbridas. Estas últimas tentam acrescentar a orientação a objetos aos conceitos tradicionais de linguagens de programação, sem detrimento destes. Nem todos os dados que estas linguagens manipulam são tratados como objetos. Exemplos de linguagens híbridas são C++ e Turbo Pascal 5.0. A linguagem orientada a objetos pura mais

típica é Smalltalk, de onde surgiu e se popularizou o termo "orientação a objetos". Como isto é aceito por todos, o paradigma da orientação a objetos apresentado a seguir se refere a como ele é definido pela linguagem Smalltalk. Em uma linguagem de programação orientada a objetos pura, todos os dados são tratados como objetos.

O paradigma da orientação a objetos se baseia nos conceitos de objetos, mensagens e classes. Objetos são usados para representar as entidades de um sistema, mensagens são o meio de comunicação entre objetos e classes descrevem características comuns a grupos de objetos.

3.2.1.1. Objetos

Os objetos caracterizam-se por possuírem uma identidade exclusiva e por encapsularem propriedades estáticas e dinâmicas. A identificação de um objeto, feita de uma forma independente de suas características, permite que este possa ser referenciado sem que se conheça suas propriedades [BLUM88]. Do ponto de vista estático, um objeto é um conjunto de campos muito parecido com uma estrutura de dados registro (*record* do Pascal ou *struct* de C). Estes campos são chamados de variáveis de instância. O valor destas variáveis representa o estado do objeto em um dado instante. Do ponto de vista dinâmico, cada objeto possui um comportamento que é um conjunto de rotinas, denominadas métodos, que define o procedimento que será ativado quando alguém mandar ele executar alguma ação.

Em uma linguagem de programação orientada a objetos pura, as rotinas de um objeto são as únicas que podem alterar os dados contidos nos campos do objeto. Esta propriedade é denominada encapsulamento, pois os objetos seriam "cápsulas", onde as rotinas formariam uma capa que impediria a manipulação direta dos dados contidos no seu interior. O conceito de encapsulamento deriva do conceito de tipos abstratos de dados [STRO88] [STEF86B], onde uma estrutura de dados possui uma porção pública e outra privada.

3.2.1.2. Identidade de Objeto

Cada objeto tem uma identidade própria, distinta dos demais objetos. Isto corresponde a noção intuitiva que se tem de objeto, algo que se pode identificar com clareza. Assim, uma gota de água é um objeto, porém, uma gota de água no oceano não é um objeto. Dois objetos distintos podem ser iguais, isto é, ter valores (e comportamentos) iguais. Um objeto não pode mudar a sua identidade, mas pode ser alterado radicalmente, trocando o seu estado e comportamento.

3.2.1.3. Estado de um Objeto

Os campos de um objeto são chamados de variáveis de instância, ou simplesmente variáveis. Uma variável associa um identificador a um outro objeto qualquer (o valor da variável).

Nas linguagens procedimentais tradicionais (Pascal, C, Fortran, etc.), uma variável contém valores de um único tipo durante toda a execução do programa. Estes tipos podem ser inteiros, reais, estruturas definidas pelo programador, etc. Em algumas linguagens puramente orientadas a objeto, como Smalltalk, uma mesma variável pode conter, em diferentes momentos da execução do programa, objetos de tipos variados.

3.2.1.4. Método

Um método especifica a reação que o objeto deve ter quando receber uma determinada mensagem. Comparando com linguagens de programação tradicionais, um método corresponde à declaração de uma função e uma mensagem é a chamada à esta função. Um método especifica instruções a serem executadas. Estas instruções são compostas por mensagens enviadas a variáveis acessíveis deste método.

Numa linguagem procedimental, uma rotina (procedimento, função ou programa principal) "chama" diretamente outra. É dito, então, que o acoplamento entre as rotinas é estático, ou seja, é resolvido antes da execução do programa. Nas linguagens orientadas a objetos, uma rotina de um objeto envia mensagens a um objeto contido em uma variável. A mesma mensagem, enviada para objetos diferentes, pode ativar rotinas completamente diversas. Para isto ser possível, é necessário haver acoplamento dinâmico[9] entre as rotinas, isto é, a ligação mensagem-método é resolvida durante a execução do programa.

3.2.1.5. Mensagem

Mensagens são o mecanismo de ativação do comportamento de um objeto. É através deste conceito que objetos se intercomunicam. Uma mensagem ativa um determinado método do objeto ao qual tenha sido dirigida. Através desta mensagem podem ser transmitidas, ao objeto destinatário, informações que poderão ser por este utilizadas.

Muito comumente uma mensagem é enviada à uma variável; neste caso, o objeto contido nesta variável é quem receberá a mensagem e executará um de

---

[9]*late-binding*, em inglês.

seus métodos. Em tempo de execução, dependendo do objeto contido na variável, a mesma mensagem pode dar resultados diferentes.

A resposta à mensagem conclui uma comunicação entre dois objetos. Isto pode ser usado para transmitir, ao objeto que iniciou a comunicação, informações sobre o estado do objeto cujo comportamento foi ativado e/ou um indicativo do sucesso (ou insucesso) da operação. Indiretamente, mais de um objeto podem ser atingidos por um processo de comunicação entre objetos. Isto se deve ao fato de que um objeto, ao receber uma mensagem, pode estabelecer novas comunicações enviando mensagens a outros objetos.

3.2.1.6. Encapsulamento

Esta propriedade do modelo é garantida pela regra de visibilidade das variáveis. A princípio, um método só faz acesso às variáveis do próprio objeto. Consequentemente, as variáveis de um objeto, ou seja, a sua estrutura, só são acessíveis pelos métodos do próprio objeto, implementando, assim, o encapsulamento.

Em um sistema orientado a objetos puro, não há procedimentos que não sejam métodos de algum objeto. Linguagens baseadas em C++, para manter compatibilidade com linguagens tradicionais, permite a existência de procedimentos *livres*, isto é, declarados fora de qualquer classe (ver item seguinte: "Classes").

3.2.1.7. Classes

Como cada objeto representa uma estrutura de dados encapsulada, se todos os objetos, além do seu estado, tivessem que incorporar a descrição de sua estrutura e o código para o processamento do seu comportamento, se teria um sistema inviável de ser utilizado e mesmo de ser implementado. Os mecanismos de classificação e herança viabilizam o modelo.

Classes são objetos que têm por objetivo descrever as características de grupos de objetos. Uma classe reúne as propriedades comuns a um dado tipo de objetos. De modo semelhante ao que uma declaração de tipo de dados registro, em uma linguagem convencional, faz, ao definir os campos que qualquer variável deste tipo terá, a definição de uma classe também indica as variáveis de instância que os objetos desta classe terão. Porém, uma classe também define as rotinas que manipulam estes campos. Esta idéias estão presentes nas linguagens procedimentais que permitem a definição de tipos abstratos de dados: Modula 2, Ada, CHILL, CLU, Alphard, etc. Porém nestas não se encontram os conceitos de herança (discutido a seguir) e acoplamento dinâmico.

Classes são um mecanismo de abstração, pois permitem fatorar propriedades (campos e rotinas) comuns a objetos. Porém, propriedades iguais, contidas em classes diferentes, também podem ser fatoradas criando-se superclasses. Esta característica é conhecida como herança, pois as classes podem ser arrumadas em uma hierarquia onde as classes inferiores herdam propriedades das superiores.

Qualquer classe poderá ser usada como superclasse de novas classes que serão especializações da superclasse. A classe especializada é chamada de subclasse daquela genérica. Uma superclasse pode realmente ser uma classe, possuindo instâncias, ou ser uma pseudoclasse, usada apenas para serem superclasses de outras classes.

O relacionamento de generalização/especialização entre classes define que a subclasse herda as propriedades definidas na superclasse. Se a subclasse, por sua vez, define propriedades específicas que conflitam com as suas propriedades herdadas, as propriedades específicas têm ascendência sobre as herdadas, anulando a herança de propriedades conflitantes.

Em Smalltalk, um método de um objeto tem acesso a qualquer variável de intância deste objeto, mesmo que o método seja definido em uma classe e variável em uma superclasse desta classe. Se isto não fosse permitido, aumentaria a independência entre a classe e a implementação da sua superclasse [SNYD86].

Um dos objetivos de se representar objetos através de uma hierarquia semântica de classes é possibilitar a definição de novas classes de objetos a partir da especialização de classes de objetos já existentes. Isto é feito através de programação diferencial e é uma estratégia importante para a reutilização de código.

3.2.1.8. Tipos Abstratos e Implementação

Os conceitos de classe e tipo frequentemente são confundidos. Em linguagens fortemente-tipadas[10], o que não é o caso de Smalltalk, pode-se definir que um **tipo abstrato** é o conjunto de assinaturas[11] de métodos que especificam uma interface de objeto. Uma classe, ou **implementação**, especifica a estrutura implementada de um objeto e o código de seus métodos. Uma classe **implementa** um tipo abstrato se exporta uma interface que satisfaz o tipo, ou seja, que contém a interface definida pelo tipo. Assim são definidos tipos

---

[10] *strongly-typed , em inglês.*

[11] *signatures*, em inglês.

abstratos e implementação no sistema Emerald [BLAC86]. Com esta definição, pode-se ter diversas classes implementando um mesmo tipo abstrato.

O tipo abstrato definido implicitamente por um objeto, ou por uma classe/implementação, é denominado **tipo abstrato induzido**. Um tipo abstrato T1 **está em conformidade com** outro tipo abstrato T2, caso a interface definida por T2 seja um subconjunto da interface de T1. A verificação de conformidade pode ser feita estaticamente e seus algoritmos são fornecidos em [BLAC87]. O conceito de conformidade é próximo às regras de subtipos de Trellis/Owl [SCHA86a] [SCHA86b].

Conformidade permite que se defina generalizações e especializações de tipos de um modo diferente da abordagem convencional de C++ [STRO86], Trellis/Owl [SCHA86a], Orion[12] [KIM90a] [BANE87], GemStone [BUTT91] e O_2 [DEUX91] [LECL88]. Novos supertipos podem ser incluídos na hierarquia de tipos sem que se tenha que modificar os tipos existentes. Isto porque, se a interface definida por um tipo T1 inclui a interface requerida por um outro tipo T2, T1 é implicitamente um subtipo de T2. Um objeto de T1 pode ser usado em qualquer lugar onde seja requerido um objeto de T2.

3.2.1.9. Objetos e Valores

As linguagens de programação orientadas a objetos híbridas, entre elas C++ [STRO86] e Turbo Pascal, definem objeto simplesmente como sendo uma região de memória. Alguns dados, típicos das linguagens de programação tradicionais, como a linguagem C e Pascal ANSI, das quais as linguagens híbridas se originam, não estão relacionados a classes. Este é o caso dos inteiros, reais, caracteres e de constantes de algum tipo de enumeração. Os construtores de matrizes *("arrays")*, cadeias de caracteres e tipos de enumeração também se encontram nessa situação. Uma consequência deste fato é não se poder criar subclasses destes tipos de dados do mesmo modo que se pode fazê-lo para estruturas ("*record*s e *struct*s"). Isto é tão relevante que as bibliotecas de classes distribuídas com ou para estas linguagens criam construtores, equivalentes a esses, sobre a forma de classes. Esta redundância aumenta a complexidade de programação. Quando se precisa definir subtipos dos tipos elementares o tratamento pela linguagem não é uniforme, nem intuitivo. Isto ocorre, por exemplo, ao criar um tipo de dados PESO derivado dos reais. Numa visão orientada a objetos mais severa, pode-se dizer que a informação nestas linguagens pode ser de duas

---

[12] a versão comercial de Orion denomina-se Itasca

espécies: objetos, que podem ser gerados por classes, e valores, que não podem. Daí o rótulo de linguagens híbridas.

3.2.1.10. Alterabilidade e Reusabilidade

Na programação orientada a objetos, os módulos correspondem às classes de objetos. Uma classe agrupa, como submódulos, as rotinas que constituem o comportamento dos objetos desta classe quando eles recebem alguma mensagem.

O principal alvo dos que optam por uma linguagem orientada a objeto é conseguir desenvolver programas mais alteráveis, isto é, modificáveis e extensíveis. A divisão em módulos "visíveis", independentes e reusáveis pode conseguir que um sistema seja mais alterável.

3.2.1.11. Visibilidade

Um software é visível, quando as suas características observáveis dinamicamente são facilmente localizáveis no seu programa fonte. Se for possível descrever naturalmente estas características através dos conceitos de objetos, comportamento destes e ações que eles podem sofrer, será trivial a divisão do software em módulos que incrementem a visibilidade. Isto é o que a orientação a objetos pressupõe que aconteça.

A visibilidade de um software costuma ser medida através da coesão dos seus módulos. Módulos que têm uma razão forte para que seus elementos estejam agrupados nele são considerados coesos. Afirma-se que módulos que implementam um tipo abstrato de dados, reunindo rotinas em torno de uma estrutura de dados, alcançam o maior grau de coesão [FAIR85]. As definições de classes nas linguagens orientadas a objetos proporcionam exatamente isto.

Quando se obtém a alta visibilidade desejada, ao se desejar fazer uma alteração no software, é simples descobrir o que precisa ser alterado. Na direção oposta, também é simples inferir qual o efeito que uma determinada alteração no programa fonte causará no comportamento do software.

3.2.1.12. Independência

Já a independência entre os módulos que constituem um sistema é avaliada através do acoplamento entre os módulos. O encapsulamento das estruturas de dados, proibindo que um módulo possa manipular o interior de outro, diminui o acoplamento entre os módulos, aumentando a independência destes. O mecanismo de troca de mensagens, com parâmetros passados explicitamente nelas,

diminui fortemente o acoplamento entre os módulos. É bom lembrar que uma mensagem é enviada a um objeto, cuja classe/módulo é desconhecida pelo módulo que envia a mensagem, fazendo com que os módulos sejam bastante independentes entre si. Isto significa que pode-se estender um sistema criando novas classes, cujos objetos respondam a mensagens já definidas e usadas em outros módulos. Criando-se objetos desta nova classe, o sistema estendido passa a funcionar sem a necessidade de ter havido alteração de qualquer outro módulo.

3.2.1.13. Reusabilidade

A reusabilidade alcança, nas linguagens orientadas a objetos, um grau muito maior do que o obtido pelas simples bibliotecas de rotinas nas linguagens convencionais. A herança de propriedades, obtidas ao se definir uma nova classe como sendo subclasse de outra, podendo-se estender, modificar ou suprimir características herdadas, faz com que as linguagens orientadas a objetos venham acompanhada de uma ampla biblioteca de classes altamente reaproveitáveis. Com isto, o tempo de desenvolvimento de um sistema é enormemente diminuído. A criação rápida de protótipos, possível nestas linguagens, altera fortemente o modo de se desenvolver sistemas.

O polimorfismo, obtido pelo acoplamento dinâmico e pela herança, permite a definição, pelo programador, de conceitos gerais que, com o mesmo nome, podem se multiplicar em exemplares específicos. Por exemplo, em um programa de 10.000 linhas na linguagem C, o programador facilmente terá que inventar nomes diferentes, de um modo muitas vezes artificial, para um número muito grande de variáveis. De acordo com Plauger, em uma linguagem orientada a objeto, um programa de 100.000 linhas não causaria a mesma dificuldade [TERR89].

3.2.2. Modelos de Dados

O trabalho de definição de um modelo de objetos para sistemas de objetos baseia-se, entre outras coisas, em resultados obtidos por pesquisas feitas na área de modelos de dados. Modelos de dados são ferramentas conceituais através das quais dados podem ser organizados, representados e manipulados [TSIC82]. Além disso, estas representações e manipulações devem poder ser traduzidas para uso em computadores. Modelos de dados são úteis para a modelagem de informações referentes aos objetos relevantes a um determinado ambiente em estudo.

Um modelo de dados se compõe de [CODD80]:

(1) uma coleção de tipos de estruturas de dados;

(2) uma coleção de operadores;

(3) uma coleção de regras gerais de integridade.

Os operadores devem poder ser aplicados a qualquer instância válida dos tipos de estruturas de dados disponíveis, a fim de que se possa recuperar ou derivar dados de qualquer parte destas estruturas, com qualquer combinação desejada. As regras gerais de integridade devem, implícita ou explicitamente, definir o conjunto consistente de estados e/ou modificações de estado de um banco de dados.

Os primeiros modelos de dados a surgir estavam, de alguma forma, relacionados à tecnologia de banco de dados. Dentre eles destaca-se o modelo relacional [CODD70] [DATE90], cuja proposta formal antecedeu às suas implementações. Outros, como o modelo hierárquico e o modelo em rede, surgiram implicitamente em implementações de sistemas de gerência de banco de dados.

Nestes modelos, que são considerados clássicos, a informação deve ser organizada na forma de estruturas de dados capazes de serem entendidas e manipuladas eficientemente por um sistema de gerência de banco de dados. Isto compromete a expressividade semântica com que se consegue representar as informações referentes a uma dada realidade.

O modelo de dados de um SGBD convencional é a ferramenta conceitual que exprime como os dados são representados e operados pelo sistema. O compromisso com a eficiência faz com que nem todo tipo de informação possa ser facilmente representado nesse modelo. Para tal, comumente é necessário usar-se outros modelos de dados, diferentes do empregado pelo SGBD, com maior capacidade de representar naturalmente outros tipos de informação. O mapeamento de uma representação à outra é muitas vezes manual.

Outros modelos de dados têm sido propostos. Nestes, objetos do mundo real e seus interrelacionamentos são representados independentemente de restrições de implementação e com o máximo possível de expressividade semântica. Os **modelos de dados relacionais estendidos** incorporam aperfeiçoamentos no modelo relacional, para incorporar procedimentos, objetos, versões, entre outras facilidades. Os **modelos de dados orientados a objetos** se baseiam no paradigma da programação orientada a objetos (ver 3.2.1.:"Programação com Objetos"). **Modelos de Dados Funcionais** usam linguagens de acesso a dados baseadas na notação de funções matemáticas. Alguns destes possuem extensões para suportar especificação procedimental de funções. Modelos funcionais estendidos, para suportar aplicações não convencionais, são encontrados nos sistemas ADAPLEX, IRIS [FISH88] [FISH87] e PROBE [DAYA86]. Vários outros modelos de dados foram propostos na década de 1980, incorporando mais se-

mântica dos dados do que o modelo relacional, sendo conhecidos como **modelos semânticos de dados** [HAMM81] [HULL87]. Os sistemas CACTIS [HUDS86b] [HUDS89] e SIM são baseados em modelos semânticos.

O principal esforço na definição de um modelo semântico de dados tem sido incrementar a sua capacidade de modelagem. Isto significa dizer que estes modelos devem prover conceitos que sejam mais ricos e mais expressivos para capturar o significado dos objetos e relacionamentos por estes modelados. Em alguns casos, modelos semânticos de dados podem ser usados para representar inclusive o comportamento de tais objetos.

De uma forma geral, modelos semânticos de dados devem suportar as seguintes facilidades:

(1) **Abstração de dados** - é a habilidade de se ocultar, deliberadamente, alguns detalhes sobre os objetos pertencentes a um determinado conjunto, concentrando-se em suas propriedades comuns;

(2) **Semântica comportamental** - é a capacidade de se representar o comportamento dos objetos de um determinado modelo, em função das operações que podem ser efetuadas sobre estes;

(3) **Herança de propriedades** - é a capacidade que se tem de representar objetos cujas propriedades, estruturais e/ou comportamentais, são parcial ou integralmente definidas em função das propriedades de outros objetos;

(4) **Relativismo semântico** - é a habilidade de se interpretar e manipular dados da forma mais apropriada a cada situação.

Nem todos os modelos semânticos de dados suportam, na sua forma original, todas essas facilidades. Entretanto, pode ser observada uma tendência na definição de novos modelos, ou na extensão de modelos existentes, a fim de suportar cada uma das características acima citadas.

Tsichritzis nota que um modelo de dados tem que ser capaz de capturar propriedades estáticas e dinâmicas. Propriedades estáticas são (relativamente) invariantes no tempo e as dinâmicas correspondem à evolução que as coisas sofrem no tempo [TSIC82]. Por essa razão, ele define modelo de dados como um par M = <G, O>.

G é um conjunto de regras de geração de estruturas permitidas, para expressarem as propriedades estáticas. Corresponde a uma linguagem de definição de dados. Estas regras especificam as propriedades que são verdadeiras para todas as ocorrências de um esquema. Note-se que, para a situação descrita para ambientes de desenvolvimento, as duas afirmativas anteriores entram em contradição: esquemas (metadados) não são tão estáticos.

As regras de geração podem ser classificadas em dois tipos: regras de especificação de estrutura (Ge) e regras para especificação de restrições (Gr). As regras do tipo Ge especificam os tipos de objetos suportados pelo modelo, definindo, assim, suas construções básicas. As regras do tipo Gr definem as restrições às quais estão sujeitos tais objetos e, consequentemente, os modelos construídos a partir destes.

As propriedades dinâmicas são expressas pelo conjunto de operações (O), que correspondem a uma linguagem de manipulação de dados. Operações definem as ações que podem ser realizadas sobre as estruturas.

Para que um modelo extensível (metamodelo) possa representar modelos de dados, terá, então, que poder representar operações, ou seja, estímulos e comportamentos.

Recursos para modelagem estrutural e comportamental podem ser encontrados tanto em modelos semânticos [BROD81] [BROD82] [BROD84B] [KING85] [LYNG84] como nos modelos baseados no paradigma de orientação a objetos [GOLD83] [TAKA88]. Assim, a definição de um modelo para um ambiente de desenvolvimento de software deve ser baseada nos conceitos encontrados nestas duas linhas de pesquisa.

3.2.2.1. Modelos de Dados Orientados a Objetos

Como a orientação a objetos fornece abstração de dados e herança, os modelos de dados orientados a objetos muitas vezes são considerados mais ricos em semântica do que os modelos de dados dos SGBDs atuais, predominantemente relacionais [BANE87] [CARE88] [FISH87] [HUDS89] [LECL88] [MAIE86]. Objetos permitem uma modelagem conceitual mais direta por combinarem estado e comportamento em uma única abstração. Todavia, relacionamentos e restrições não são diretamente suportados pelos modelos orientados a objetos. Para manutenção da integridade, as operações de verificação costumam ser programadas explicitamente como métodos em cada uma das classes envolvidas. Se o modelo tivesse consciência da existência do relacionamento, a integridade poderia ser mantida implícita e automaticamente.

O mesmo ocorre com restrições globais envolvendo mais de um tipo de objeto. A mudança de uma restrição poderia implicar em alterações em vários métodos de diversas classes. Se o modelo conhecesse a restrição global, estas alterações seriam implícitas.

O encapsulamento estrito das estruturas de dados, preconizado pelo paradigma da orientação a objetos, é frequentemente considerado restritivo para SGBDs [ATKI90]. A modelagem de relacionamentos e restrições aparece fre-

qüentemente em modelos semânticos de dados. Nestes, a preocupação principal está na modelagem da estrutura conceitual (percebida) dos dados. A conciliação dos dois objetivos é um dos princípios fundamentais a ser alcançado pelo modelo de objetos de um SOO.

3.2.2.1.1. Modelos Híbridos

Assim como as linguagens de programação orientadas a objetos, os modelos de dados orientados a objetos podem ser puros ou híbridos (ver 3.2.1.9.). A distinção entre estes é mais marcante do que nas linguagens de programação. Por exemplo, o sistema O_2 define explicitamente valores, objetos e construtores. Valores são os dados conforme tratados tradicionalmente pelos SGBDs e linguagens de programação. Objetos são unidades de informação conformes com a visão das linguagens de programação orientadas a objetos puras. Os construtores são em número fixo: de tupla, de conjunto e de lista. Conjuntos e listas servem para definir as coleções. Estas não são tratadas como objetos. Toda esta heterogeneidade aumenta a complexidade do sistema, exigindo do programador de aplicações o conhecimento de como uma determinada informação foi definida.

No sistema O_2, objetos são criados a partir de classes, ao contrário de valores e coleções, que são baseados em "tipos de dados" e construtores. Classes herdam propriedades de suas superclasses, mas não há herança para tipos de dados. Novos construtores de coleções não podem ser definidos.

3.2.2.1.2. Identidade de Objetos

Vários modelos de dados, posteriores ao relacional, definem a existência de identificadores de entidade únicos, gerados pelo sistema, frequentemente chamados de ***surrogates*** (algo próximo a "representantes"). Dentre esses há RM/T [CODD79], DAPLEX [SHIP81] e SDM [HAMM81]. Um atributo preenchido com um *surrogate* indica que este atributo faz referência ao objeto identificado pelo *surrogate*. Portanto, *surrogates* são usados no lugar de chaves estrangeiras. Chaves fornecidas pelos seres humanos ou são expressivas, mas longas para serem usadas como chaves estrangeiras, ou são pouco expressivas (números de matrícula, cpf, etc.).

3.2.2.1.3. Procedimentos

Sistemas de bancos de dados comportamentalmente orientados a objetos, isto é, que incorporam procedimentos, têm sido considerados como os que maior proveito tiram da orientação a objetos (ver 2.2.2.2.:"Sistemas de Gerência

de Bases de Dados Comportamentalmente Orientadas a Objetos"). Aos sistemas de bancos de dados orientados a objetos tem sido requisitado que sejam computacionalmente completos (ver 2.2.3.:"Sistemas de Banco de Dados Orientados a Objeto").

Na abordagem adotada por sistemas de banco de dados orientado a objetos, procedimentos são associados a objetos, seguindo o modelo adotado pelas linguagens de programação orientadas a objetos puras (ver 3.2.1.4.:"Método"). Em uma outra abordagem, uma linguagem de consulta a banco de dados é estendida com construções como variáveis, repetições[13] e procedimentos. Um terceiro caminho é o seguido pelo sistema POSTGRES [STON87], onde procedimentos, de uma linguagem de programação separada, são armazenados como atributos e executados quando se tenta acessar esses atributos.

3.2.2.1.4. Classes e Coleções

A divergência mais comum entre linguagens de programação orientadas a objetos e sistemas de banco de dados orientados a objetos é que nestes, frequentemente, se confundem os conceitos de classe e de coleção. A vantagem dessa unificação está, principalmente, na simplicidade do modelo. Já a diferenciação entre classes ou tipos e as coleções provê maior poder de expressão [ATKI87].

3.2.2.1.5. Tipagem

Além da conveniência conceitual das declarações de tipo, elas podem ser usadas para prevenir erros de programação. Uma linguagem é **fortemente tipada** se evita que uma operação possa ser aplicada a valores de tipos não apropriados. Uma linguagem é **estaticamente tipada** se todos os erros de tipo podem ser detectados antes da execução do programa.

3.2.2.1.6. Composição

Relacionamentos de **composição** são tratados especialmente em diversos modelos de dados. Peças podem ser partes de peças compostas, assim como capítulos são partes de um livro. Uma parte de um objeto pode, por sua vez, possuir subpartes [SMIT77]. O objetivo de se explicitar os relacionamentos de composição abrangem o fornecimento de indicações para o agrupamento[14]

---

[13] *loops*, em inglês.

[14] *clustering*, em inglês.

físico de objetos, para a cópia, movimentação e remoção de objetos. Observe-se que, se ao usuário não for permitido deletar objetos, a explicitação da composição destes tem menos importância.

Alguns sistemas atuais especificam o relacionamento de composição qualificando os atributos envolvidos. Há divergências sobre se um objeto pode participar como componente de mais de um objeto. Isto estaria de acordo com situações do mundo real, onde, por exemplo, um prédio pode ter a sua parte elétrica em uma composição, a parte hidráulica em outra, e essas partes terem interseções. Todavia, os objetivos acima citados sobre a necessidade de se destacar a existência de uma composição seriam perdidos.

3.2.2.1.7. Coleções

Atributos multivalorados são representados nos modelos orientados a objetos por atributos-coleções ou por atributos contendo coleções. Neste último caso, o conteúdo do atributo é um objeto que, por sua vez, é uma coleção com os valores finais do atributo. No caso de um atributo-coleção, o conteúdo corresponde aos próprios valores. A diferença entre as duas abordagens é sutil. Um atributo-coleção, em geral, tem um tipo parametrizado escolhido entre um pequeno número de tipos predefinidos (lista, conjunto e *array*, por exemplo). No outro caso, pode-se definir novos tipos específicos de coleções. A abordagem atributo-coleção é mais natural para a modelagem conceitual da informação.

3.2.2.1.8. Atributos Derivados

Atributos derivados são calculados procedimentalmente, no lugar de simplesmente estarem armazenados. Quando um atributo derivado pode ser usado como se fosse um atributo comum, é chamado de **atributo virtual**. Atributos virtuais podem aumentar a independência lógica dos dados. Os sistemas POSTGRES e O_2 permitem a definição de atributos virtuais.

A sintaxe da consulta a um atributo virtual deve ser a mesma da consulta a um atributo comum. Isto não ocorre, por exemplo, com a linguagem C++ [ELLI90], onde a sintaxe para a execução de uma função sem parâmetros difere do acesso a uma variável. O sistema O_2, ao contrário, permite que a consulta a um atributo seja resolvida por um método. Todavia, o sistema O_2 e os demais sistemas de banco de dados orientados a objetos não permitem a atualização de atributos derivados. Algumas linguagens de programação, como CLU [LISK77] e Loops [STEF86a] [STEF86B], e sistemas de gerência de interface com usuário, como Peridot [MYER88] e Chiron [YOUN88], permitem a definição procedimental da operação de atribuição. Estas linguagens, com isso, são

capazes de suportar um outro paradigma de programação, a orientação a acessos (ver 3.2.3.3.8.7.:"Orientação a Acessos"), considerado o paradigma dual da orientação a objetos. A orientação a acessos é importante para a definição de efeitos colaterais gerados na interface com usuário pela atualização de um objeto, para a verificação dinâmica de violação de restrições e para o disparo de gatilhos[15].

3.2.2.1.9. Aspectos

Aspectos foram introduzidos no modelo de dados Melampus [RICH91] para tratar objetos que mudam de tipo ou que tenham múltiplos tipos dentro de um SGBDOO fortemente-tipado. Papéis múltiplos e alteráveis serão típicos nas entidades de vida longa características da próxima geração de sistemas de informação onicientes. Um **aspecto** estende um objeto existente com novos atributos e comportamento, preservando a mesma identidade do objeto. Aspectos também podem ser usados para encapsular o resultado de consultas.

Enquanto o paradigma da orientação a objetos permite a captura da noção de que um estudante *é uma (is a)* pessoa, ele não suporta diretamente a noção de que uma pessoa pode *se tornar* um estudante. E, mais ainda, pode, a qualquer momento, se tornar, simultaneamente, empregado e sócio de um clube e deixar de ser qualquer uma dessas coisas. Em um ambiente de engenharia de software, também há entidades cujos papéis evoluem com o tempo. Por exemplo, um programa pode ser uma cadeia de caracteres com o texto fonte, um documento estruturado, uma árvore sintática e um módulo executável.

O uso de herança múltipla para modelar objetos que são de vários tipos simultaneamente pode levar a uma explosão combinatória [McAl86] de classes interseções [SCIO89], que não acrescentam estado, nem comportamento. Por exemplo, seria necessário se criar a classe *EstudanteEmpregado*, herdando de *Estudante* e *Empregado*, para se ter entidades que fossem simultaneamente estudantes e empregados. Havendo conflito de nomes entre os atributos herdados das duas classes, deve-se optar, de algum modo, pela herança de apenas um dos atributos [BOBR88] [BANE87], o que pode não ser o mais conveniente. Por exemplo, um estudante-empregado poderia ter um departamento onde estuda e um departamento onde trabalha.

Um objeto pode, dinamicamente, adquirir aspectos, e se liberar deles. Toda referência a um objeto indica sob qual aspecto o objeto está sendo

---

[15] *triggers*, em inglês.

observado. Desse modo, o estudante-empregado referenciado como estudante retornará um departamento; como empregado, retornará outro.

O modelo de dados do sistema de banco de dados orientado a objetos Iris [FISH87] permite que um objeto ganhe e perca tipos durante a sua vida, mantendo intacta a sua identidade. Todavia, após ganhar multiplos tipos, pode ocorrer conflitos entre métodos de mesma assinatura definidos em tipos diferentes. Aspectos, ao contrário, particiona uma entidade em diferentes pontos-de-vista independentes, que para serem usados têm que ser referidos explicitamente.

O modelo de dados Melampus propõe o uso de aspectos em um ambiente fortemente-tipado, baseado no conceito de **conformidade**, tal como definido em Emerald [BLAC86] (ver 3.2.1.8.:"Tipos Abstratos e Implementação").

Para se definir um aspecto no modelo definido em [RICH91], define-se uma implementação que estende um tipo abstrato. Todavia, o tipo abstrato definido pelo aspecto não é necessariamente um subtipo de seu tipo base. Isto porque, os métodos do tipo base só fazem parte do tipo abstrato definido pelo aspecto se forem explicitamente exportados pelo aspecto.

Um mesmo objeto pode ter vários aspectos do mesmo tipo (uma pessoa pode ser estudante em várias escolas). Pode-se usar aspectos múltiplos podem representar versões múltiplas de um objeto.

Na sua definição original, aspectos apresentam os seguintes problemas em aberto [RICH91]: como verificar se um objeto possui um dado aspecto e usá-lo mantendo a tipagem forte? como lidar com a remoção de aspectos sem que se gere referências pendentes[16] entre objetos? como compatibilizar aspectos e autorização? como reusar definições de tipos e aspectos? como utilizar aspectos em um modelo de dados real?

Um outro mecanismo, denominado **trevo**[17], possui muitas semelhanças com aspectos [RICH91]. Basicamente, um objeto pode ser estendido por um subtipo de seu tipo, do mesmo modo que um aspecto estenderia um objeto. Entretanto, aspectos não são necessariamente subtipos de seu tipo base. Neste ponto, trevos são mais naturais que aspectos, pois um empregado que assumisse o papel de gerente não deixaria de ser um empregado. Todavia, o conceito de subtipo em trevos passa a exercer um papel dúbio. Compare-se, por exemplo, como um método deve ser escolhido quando se tem subtipos tradicionais e

---

[16] *dangling references*, em inglês.

[17] *clover*, em inglês.

subtipos que funcionam como aspectos. Com subtipos tradicionais, um método deve ser escolhido, dentro do objeto, no tipo mais especializado que definir o método. Em um objeto com aspectos, a menos que a referência ao aspecto desejado seja feita explicitamente, o método do aspecto nunca será escolhido. Estes dois comportamentos são naturais em casos diferentes. Quando um objeto pode ser alterado dinamicamente, aspectos são mais razoáveis. Caso contrário, subtipos serão mais apropriados. Por exemplo, *PeçaBase* e *PeçaComposta* devem ser subtipos de *Peça*, pois não é esperado que uma peça qualquer se transforme em uma peça base ou composta. Porém, *Gerente* pode ser um aspecto de *Empregado*, porque um empregado pode vir a se tornar um gerente.

3.2.2.1.10. Suporte a Relacionamentos

Relacionamentos são representados em modelos de dados orientados a objetos através de atributos colocados em cada objeto envolvido no relacionamento. Relacionamentos muitos-para-um e muitos-para-muitos requerem que atributos com coleções sejam usados. Um relacionamento binário, portanto, é representado por dois atributos, um em cada objeto. Dado um desses atributos, o outro é a sua **inversa**, e vice-versa. Esta representação de relacionamentos, característica da orientação a objetos, é discutida em [RUMB87]. Relacionamentos não-binários necessitam da criação de novos tipos de objetos para representá-los [CATT91].

Linguagens de programação compiladas, como C++, não fornecem mecanismos de verificação da integridade referencial dos relacionamentos. Assim, um atributo pode conter uma referência (um apontador) para um objeto que não mais existe, causando uma referência pendente, ou, pior ainda, para um outro objeto que coincidiu ter a mesma identidade do objeto que era correto, mas foi suprimido. Linguagens como Smalltalk e sistemas como GemStone [BRET88] [BUTT91] usam coletores de lixo para remover objetos não referenciados. Enquanto houver referência a um objeto, ele existirá. Outros sistemas, como IRIS [FISH87], por não reusarem OIDs (identidades de objetos), podem descobrir que uma referência a um objeto é na verdade uma referência a objeto que não mais existe.

Para que a integridade de um relacionamento seja preservada, os atributos inversos, participantes do relacionamento, têm que ser mantidos corretos pelo sistema. Alguns sistemas de banco de dados têm esse cuidado, como é o caso de PROBE [DAYA86].

3.2.2.1.11. Construtores

Linguagens tradicionais, como ALGOL e Pascal, permitem que se crie novos tipos de dados através de construtores predefinidos (*array, record*). Porém novos construtores não podem ser definidos pelo programador. Outras linguagens permitem a definição de tipos *polimórficos* ou parametrizados (Ada [ICHB79], CLU, Miranda [TURN86]). Se isto dá mais poder a uma linguagem, aumenta a sua complexidade. Poder-se-ia tentar definir o conjunto "adequado" de construtores predefinidos. Sistemas como O_2, Adaplex, Galileo [ALBA85] e Pascal/R, mesmo adotando esta abordagem, possuem mecanismos poderosos para a modelagem de dados. Outra questão se refere a se a linguagem fornece os mecanismos adequados para a manipulação das estruturas construídas. Novamente, linguagens com a possibilidade de se criar novos construtores e novas operações para os construtores alcançam um maior grau de extensibilidade e capacidade de ajustamento a necessidades específicas. A capacidade de se definir novos tipos de coleções, além de um conjunto rico de tipos de coleções e não-coleções, é característico das linguagens de programação orientadas a objetos e deve ser preservado em um modelo de objetos para um sistema de objetos.

3.2.2.1.12. Chaves

Frequentemente, modelos orientados a objetos, tendo oferecido identificadores de objetos, únicos e gerados pelo sistema, não permitem a definição de chaves primárias. Entretanto, é comum se ter objetos em que algum de seus atributos seja um texto, ou um número, significativo, próprio a ser usado como nome do objeto. SGBDs de terceira geração (ver 2.2.4.:"SGBDs de Terceira Geração") preconizam que apenas quando não for razoável o uso de chaves primárias, deve-se apelar para o uso de OIDs (identificadores de objetos) [STON90].

A utilização de uma chave, entendida aqui como um nome interno significativo que caracteriza um objeto, no lugar da identidade desse objeto gerada pelo sistema, em um atributo para representar uma referência a este objeto, é muitas vezes mais natural e proporciona maior flexibilidade. Por exemplo, na modelagem do que pode ser um código em alguma linguagem com *overloading*, uma função deve ser representada por sua assinatura (simplificando: o nome da função) e não pela identidade do código desta, pois este só será conhecido em tempo de execução. É claro que, nesse caso, o nome da função não é uma chave no universo de todas as funções, mas é uma chave na coleção de funções que um objeto pode executar. A manutenção da integridade referencial em situações desse tipo é mais complexa do que quando se pode usar diretamente a identidade do objeto.

Dependendo se um modelo confunde ou separa os conceitos de classes e coleções, o significado de chave primária pode variar muito. Um atributo chave garante que, dentro de uma coleção de objetos, não há dois elementos com o mesmo valor nesse atributo (ou que esse valor seja indefinido). Se classes ou tipos forem encarados como coleções, chaves poderão ser definidas para classes ou tipos, significando que não há duas instâncias com o mesmo atributo na classe ou tipo. O modelo de K [SHYY91] é um exemplo desse caso. Modelos que permitem a criação de tipos com extensão, ou seja, com uma coleção, associada ao tipo, contendo todas as suas instancias, podem conseguir o efeito de se ter chaves em tipos, bastando ter a facilidade de se definir chaves em coleções.

Chaves em tipos e classes funcionam como um nome interno possuído pelo objeto, no sentido em que faz parte deste. Nomes de variáveis e de atributos seriam nomes externos com os quais um objeto pode ser referido dentro do escopo da variável ou do atributo.

3.2.2.1.13. Vistas

As dificuldades encontradas ao se tentar reestruturar dados ou integrar bases demonstraram que os modelos de dados orientados a objetos em uso ainda carecem de flexibilidade. Mecanismos de vistas[18], permitindo modos particulares e ajustados de se observar um objeto, para esses modelos têm sido discutidos de modo a superar essas deficiências. Segundo Abiteboul e Bonner [ABIT91], em um dos raros trabalhos sobre o assunto, assim como no modelo relacional, "uma vista é apenas uma consulta"[19]. Vistas devem permitir a modificação do comportamento e da estrutura de objetos, assim como a especificação de atributos derivados (ver 3.2.2.1.12.:"Atributos Derivados"). Também deve ser possível se criar **coleções virtuais**[20], com **objetos imaginários**, que só sejam reais na vista. Se um objeto imaginário deve possuir uma identidade própria, como definido em [ABIT91], ou compartilhar a identidade com um objeto base real, do qual ele é uma vista, é uma questão em aberto, com prós e contras para cada solução. A solução da identidade própria é simples e atende muitos casos comuns de uso de vistas. Assim como uma consulta retorna um objeto, uma vista também resultaria num objeto de primeira classe. Todavia, poder-se-ia esperar que uma alteração nesse objeto imaginário causasse alterações em

---

[18]*views*, em inglês.

[19]*"a view is just a query"*, no original.

[20][ABIT91] propõe *classes virtuais*. Todavia, no modelo aí discutido, classes possuem extensão e são base para as consultas. Coleções virtuais seriam um conceito mais abrangente.

objetos reais dos quais ele foi derivado. Além disso, é ambíguo o significado da atribuição de um objeto imaginário com identidade própria a um atributo de um objeto real.

Abiteboul e Bonner utilizam o modelo de dados do sistema O_2 [LECL89] para definir atributos e classes virtuais, obtendo um resultado mais expressivo do que o descrito para o sistema Orion [KIM89]. É feita uma crítica ao uso de consultas *select,* inspiradas na linguagem SQL, para a definição de vistas. Em uma consulta *select*, são especificados apenas os atributos que se desejam ter na vista. Isto cria dependência entre a vista e o esquema original. Se novos atributos forem criados, eles não estarão na vista. Em sistemas orientados a objetos, a situação é errônea, pois se ocultará indiscriminadamente todos os atributos definidos em subclasses. Portanto, se o objetivo da vista é ocultar um dado atributo, isto deve ser explicitado na definição da vista.

Para criar classes virtuais, Abiteboul e Bonner definem que pode-se selecionar objetos existentes ou criar novos. Através da seleção de objetos, uma classe virtual especializa outras classes. Incluindo classes, uma classe virtual generaliza estas. Não é permitido diretamente em uma classe virtual. Porém, novos atributos virtuais e métodos podem ser acrescentados. O sistema K [SHYY91] também define classes virtuais de maneira semelhante: através de *unions* e de *views* (projeções).

Objetos imaginários, por sua vez, são objetos calculados sob demanda a partir de outros objetos (reais ou imaginários). Por exemplo, dada uma classe *Pessoa*, cujas instâncias possuem um atributo *cônjuge*, pode-se criar objetos imaginários *Família*, com atributos *marido* e *mulher*, calculados a partir de *Pessoa*. *Família* seria uma **classe imaginária** por conter apenas objetos imaginários.

3.2.3. Modelos de Interface com Usuário

A especificação de interfaces com usuário, assunto desta subseção, é fundamental na utilização de SGIUs, onde diversos modelos de especificação foram desenvolvidos. Dentre os objetivos dos SGIUs, destacam-se, pela grande influência nas características destes modelos, a independência de diálogo e a independência de dispositivos. Por sua vez, a comunicação homem-computador por manipulação direta, normalmente não contemplada nos SGIUs tradicionais, também deve ser considerada para a definição de um modelo de especificação de interfaces com usuário. A primeira parte desta subseção trata desses objetivos que servem de balisamento para a análise de modelos de representação de interfaces com usuário.

Geralmente os sistemas de gerência de interface com usuário tratam de como especificar:

1. técnicas de interação com usuário e
2. como uma aplicação relaciona-se com técnicas de interação,

de acordo com a filosofia de separar das aplicações a parte concernente à interface com usuário. Como mencionado anteriormente, nesta separação, típica dos SGIUs, reside a dificuldade destes em prover comunicação homem-computador por manipulação direta. A segunda parte desta subseção apresenta alguns modelos de arquitetura de sistemas interativos, com diferentes abordagens de relacionamento entre aplicações e interfaces com usuário. A terceira parte apresenta modelos de especificação da interação com usuário propriamente dita.

3.2.3.1. Objetivos a Considerar em Sistemas Interativos

3.2.3.1.1. Independência de Diálogo

A interface com usuário é a parte dos sistemas interativos responsável pela comunicação homem-computador. Os sistemas de gerência de interface com usuário (SGIU) têm como finalidade gerenciar as interfaces com usuário das aplicações. A separação lógica entre a interface com usuário e o restante (a semântica) da aplicação, preconizada pelos SGIUs, traz como principal benefício a independência de diálogo, que permite, entre outras vantagens, que a aplicação e a(s) sua(s) interface(s) com usuário possam ser alteradas independentemente.

3.2.3.1.2. Independência de Dispositivos

A independência de dispositivos, por sua vez, permite que a interface com usuário de uma aplicação tenha a possibilidade de trocar o uso de um determinado dispositivo por outro equivalente sem ser alterada substancialmente.

Esta independência sugere que, dentro da interface com usuário, os componentes de controle dos dispositivos de entrada e saída devam ser separados do restante da interface. Isto serve como exemplo de como a arquitetura da interface com usuário, entendida como sendo o modelo de sua composição em módulos, afeta as propriedades das aplicações. Indiretamente, o modelo arquitetural de um SGIU influencia as notações que descrevem a estrutura da interface com usuário, assim como o modelo de representação da estrutura de aplicações interativas pode impor uma arquitetura ao SGIU.

Um SGIU, normalmente, se baseia em um modelo de arquitetura de sistemas interativos. Cada modelo significa um tipo de modularização, afetando a reusabilidade dos componentes, a complexidade do projeto, da prototipação e da manutenção dos sistemas. A fragmentação dos sistemas não pode, porém, isolar do usuário a semântica da aplicação, a ponto dele não ter a chamada realimentação semântica [MONT92a].

3.2.3.1.3. Comunicação Homem-Computador por Manipulação Direta

A interação com o usuário por Manipulação Direta (MD) caracteriza-se por [MYER87]:

- sintaxe pouco exigente;
- uso de múltiplas técnicas de interação ao mesmo tempo;
- uso de dispositivos de entrada de dados diferentes do teclado (ex: "*mouse*");
- gráficos elaborados.
- realimentação semântica nas técnicas de interação;

Realimentação ("*Feedback*") é a resposta visual/auditiva à ação do usuário fornecida imediatamente pela própria interface. Realimentação Semântica é o envolvimento da aplicação com a realimentação durante a movimentação do *mouse* e dentro de "loops internos" das interações. Por exemplo, um objeto sendo arrastado aos saltos para pontos legais definidos pelo estado corrente da aplicação. Este tipo de realimentação é bastante complicado para que o SGIU possa fornecê-lo, localmente, sem fazer acesso às estruturas de dados da aplicação.

Em um sistema puramente orientado a objetos, uma aplicação se dilui em um conjunto de métodos de objetos específicos. Estes objetos serão aqui denominados "objetos da aplicação". Estes formam a semântica da aplicação. Uma interface com usuário com manipulação direta consiste, exatamente, de representar estes objetos na tela (ou em outro meio de interação com o usuário)

e fazê-los responder adequadamente às ações do usuário sobre essas representações.

Estando, nos sistemas orientados a objetos, a aplicação fragmentada em "objetos da aplicação" e a interface com usuário, do mesmo modo, composta de "objetos de interface com usuário" e aqueles se comunicando com estes, a separação entre a semântica da aplicação e a interação com usuário pode ser representada com o seu devido grau de independência mesmo em sistemas com manipulação direta.

3.2.3.1.4. Relacionamento entre o SGIU e a Aplicação

A separação entre a comunicação homem-computador de uma aplicação e a aplicação propriamente dita permite maior controle sobre ambas as partes. Os benefícios, já enumerados, incluem a facilidade de projeto e manutenção, a reusabilidade, entre outros. Porém, a separação mencionada causa problemas de desempenho na realimentação semântica na comunicação homem-computador baseada em manipulação direta [STRU85]. Na manipulação direta, exatamente, a realimentação semântica é extremamente importante e mais se precisa de um excelente desempenho. Uma interface com a aplicação baseada no compartilhamento de determinados objetos é preferível às interfaces convencionais baseadas em chamadas a ações semânticas.

Outra questão relativa ao relacionamento entre a interface com usuário e o restante da aplicação é o controle da seqüência e sincronismo de execução dos módulos do sistema. Com a separação da interface com usuário, fica colocado o problema de qual módulo deve governar esta seqüência. De acordo com essa decisão se terá controle interno, externo, misto ou independente (Figura 1).

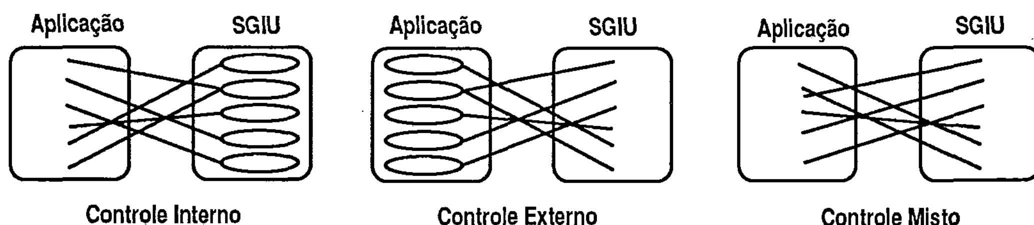


Figura 1: Tipos de Controle

3.2.3.1.4.1. Controle Interno

A solução mais tradicional é o controle interno: a aplicação propriamente dita controla a execução, fazendo chamadas a rotinas da interface com usuário. Neste caso, a interface com usuário se assemelha a uma biblioteca de rotinas ativáveis sob o controle do programa de aplicação.

Este é o sistema empregado pelas aplicações que usam pacotes de ferramentas gráficas ou sistemas de janelas. Apesar de ser uma solução eficiente em termos de tempo de execução, a necessidade de amoldar a interface com usuário à estrutura de controle do programa faz com que a interface seja mais difícil de ser alterada. A construção de protótipos da interface com usuário também é prejudicada, pois eles necessitam, além da interface com usuário, que boa parte da aplicação em si exista para poder controlar a execução do protótipo.

3.2.3.1.4.2. Controle Externo

A necessidade de se construir interfaces com usuário executáveis independentemente do restante da aplicação levou ao chamado controle externo. Neste, a estrutura do diálogo com o usuário é a responsável pelo fluxo de execução do sistema, que passa a depender das entradas do usuário. O restante da aplicação é formado por rotinas, chamadas pela interface com usuário, que executam computação específica da semântica da aplicação. Com isso, a prototipação fica facilitada, bastando, para a interface com usuário poder ser utilizada, que rotinas toscas façam o mínimo de simulação da semântica da aplicação. O controle externo obriga a parte semântica da aplicação a se amoldar ao diálogo com o usuário. A conseqüência é o aumento da complexidade de alterações na parte computacional da aplicação.

3.2.3.1.4.3. Controle Misto

Com o controle misto, tanto a parte funcional da aplicação pode chamar rotinas da interface com usuário, quanto esta pode chamar rotinas da parte funcional. Isto pode significar que a interface entre as duas partes da aplicação será mais complexa e que os projetos da interface com usuário e do restante da aplicação serão mais dependentes um do outro.

3.2.3.1.4.4. Controle Neutro

Há também a possibilidade de se ter o controle de seqüência tanto fora da interface com usuário, quanto da parte funcional da aplicação, como no DMS [HART84]. Neste caso, a aplicação passa a ter uma terceira parte, responsável pelo controle global da execução.

A seguir são então mostrados alguns modelos de arquitetura de software interativo para fornecer subsídios à definição do modelo de objetos do GEOTABA.

3.2.3.2. Modelos de Arquitetura de Sistemas Interativos

3.2.3.2.1. O DMS

O Dialogue Management System (DMS) é parte do projeto de Gerência de Diálogo *(Dialog Management Project)* em Virginia Tech [HART90]. Para o DMS [HART84], a aplicação é quebrada em três componentes: de diálogo, de controle global e computacional. O primeiro é a própria interface com usuário, contendo a lógica do diálogo, telas, mensagens de erro e processamento da entrada de dados. O último contém a semântica da aplicação. O componente de controle global dirige a seqüência de execução chamando o diálogo e a semântica da aplicação conforme necessário.

O componente de diálogo é uma coleção de unidades funcionais de diálogo. O componente computacional é a coleção de unidades computacionais semânticas da funcionalidade da aplicação. O componente de controle global contém a lógica de alto-nível do seqüenciamento das unidades de diálogo e computacional.

A total separação entre o diálogo e a semântica da aplicação, encontrada no DMS, corresponde ao ideal dos SGIUs. Todavia, esta separação só é realmente aplicável nos diálogos seqüenciais.

3.2.3.2.2. Modelo de Seeheim

Em 1984, em Seeheim, na Alemanha, um *workshop* patrocinado pela ACM SIGGRAPH examinou a arquitetura da interface com usuário sob o lema da separação entre os módulos de interface e os da aplicação propriamente dita. O Modelo de Seeheim [GREE85a] define que uma interface com usuário deve consistir de um *pipeline* de três componentes:

- *Apresentação.* É o componente responsável pela representação física da interface com usuários. Representa os aspectos léxicos da comunicação homem-computador, tanto para a entrada quanto para a saída de dados. Contendo as dependências de dispositivo, torna o restante da interface e da aplicação independentes destes. Lida com os dispositivos físicos de entrada e de saída, leiaute da tela, técnicas de interação e técnicas de exibição.
- *Controle de Diálogo.* É o componente responsável pela definição da estrutura e do sincronismo do diálogo entre o usuário e a aplicação. Pode ser visto como o coração da interface com usuário. Representa os aspectos sintáticos da comunicação homem-computador. Controla o fluxo de informação entre as outras duas camadas.

- *Modelo da Interface com a Aplicação.* É o componente responsável pela representação da aplicação no ponto de vista da interface, representando os aspectos semânticos da comunicação homem-computador. A comunicação com a aplicação é realizada através de chamadas a rotinas da aplicação. Este componente define os relacionamentos entre a aplicação e o diálogo e dirige a troca de informação entre entre eles.

O modelo de Seeheim, para obter a independência de diálogo, promove em um componente de controle, o modelo da interface com a aplicação, um fraco acoplamento entre a aplicação e a sua interface com usuário [RHYN87]. A distância entre estas partes prejudica comprovadamente as atividades semânticas da interface, tais como a realimentação e a ajuda dependente do contexto.

3.2.3.2.3. Modelo de Seattle

Em 1986, ocorreu em Seattle, um *workshop* da ACM SIGGRAPH sobre Ferramentas de Software para Gerência de Interface com o Usuário. Preocupado com questões relacionadas à manipulação direta, neste *workshop* foi discutido um modelo [DANC87] baseado no modelo de Seeheim. No novo modelo, o componente de apresentação é substituído pelo agente da estação de trabalho. O controle do diálogo e o modelo da interface com a aplicação, mais algo do componente de apresentação, formam a gerência de diálogo. A gerência de diálogo, por sua vez, inclui um subcomponente de suporte semântico semelhante ao modelo da interface com a aplicação. A finalidade deste módulo seria fornecer informação sobre operações semânticas, tais como, realimentação, valores assumidos implicitamente, verificação de erros e ajuda sobre a aplicação, lidando com informação referente tanto à aplicação, quanto à gerência de diálogo.

A gerência de diálogo coloca à disposição da aplicação serviços de alto nível, como, por exemplo, uma interface independente de mídia. Neste caso, a aplicação pode obter um dado de um determinado tipo, pela gerência de diálogo, sem se preocupar se ele foi obtido através de um menu ou pela digitação de um número.

O restante da aplicação contém as partes do sistema independentes de todas as suas interfaces; contém o que a interface com usuário não precisa conhecer e o que não precisa conhecer a interface com usuário; é responsável pela integridade dos dados, embora os outros componentes também consistam dados.

3.2.3.2.3.1. Suporte Semântico

O componente de suporte semântico é parte da gerência de diálogo e fornece as informações necessárias às operações da interface que envolvem a semântica da aplicação: realimentação, valores assumidos *("default")*, verificação e recuperação de erros, ajuda dependente do contexto e adequação do diálogo ao estado da aplicação (inibição de dispositivos, alteração de menus, etc.). Contém: as partes da interface com usuário específicas àquela aplicação; informações do usuário específicas àquela aplicação (preferências locais, última configuração, etc.); relacionamentos entre as informações manipuladas pela aplicação e a interface com usuário, como, por exemplo, qual trecho de um documento está sendo exibido em uma janela. O componente de suporte semântico, tal como o modelo de interface com a aplicação do Seeheim, especifica a interface com a aplicação, porém definindo como incorporar a semântica da aplicação à interface com usuário.

O restante da gerência de diálogo fica com as partes do sistema que independem das aplicações - por exemplo serviços de macros e gerência de ajuda *("help")* -, preferências e perfis do usuário independentes das aplicações e técnicas de interação comuns.

3.2.3.2.3.2. Agente da Estação de Trabalho

O agente da estação de trabalho (AET), contendo as partes dependentes de dispositivo, fornece protocolos dependentes da mídia, os dispositivos virtuais. Por exemplo, um mesmo conjunto de primitivas pode ser compartilhado por mouses de marcas diferentes, cada um com a necessidade de um software específico. Isto não significa que as idiossincrasias de cada dispositivo devam ser ocultadas, pois, em alguns casos, elas representam mais eficiência na interação. As técnicas de interação são responsabilidade da gerência de diálogo, apesar de baseadas nos dispositivos virtuais. O agente da estação de trabalho também é responsável pela política de multiplexação dos dispositivos pelas aplicações, como a gerência de janelas.

3.2.3.2.3.3. Gerência da Estação de Trabalho

Cada usuário pode ter as suas preferências de volume de sons de alerta, velocidade de repetição de teclas, cores defaults, etc. O metadiálogo com o qual o usuário controla a interface independente das aplicações é responsabilidade de uma aplicação: a gerência da estação de trabalho.

3.2.3.2.3.4. Estrutura do Diálogo

Uma questão deixada em aberto em Seattle é se a gerência de diálogo deve separar aspectos léxicos e sintáticos. Esta divisão é questionada por alguns

pois nem sempre é claro o limite entre o que é léxico e sintático. Sendo as técnicas de interação normalmente encaradas como entidades léxicas, como poderiam existir técnicas de interação compostas de outras técnicas? Se até as técnicas de interação precisam realimentar semânticamente o usuário e, além disso, na manipulação direta, a sintaxe é minimizada, qual a praticidade desta divisão?

3.2.3.2.4. O tríade MVC do Smalltalk

O sistema Smalltalk, com sua abordagem orientada a objetos, propõe a tríade MVC como o modelo arquitetural de objetos interativos. Uma aplicação é um objeto interativo e pode conter outros objetos interativos. O paradigma modelo-vista-controlador (MVC), além de separar a interface com usuário e o restante da aplicação, distingüe, na interface, a apresentação da informação e a interação com o usuário.

3.2.3.2.4.1. Modelo e Vista

O componente modelo da aplicação é um objeto que representa os objetos manipulados pela semântica da aplicação. O componente vista é um objeto que contém a visualização destes objetos na tela. Classes diferentes de vistas podem exibir um mesmo objeto de maneiras diferentes. Por exemplo, a hora corrente pode ser representada em uma vista que simule um relógio com ponteiros ou em uma outra vista que simule um relógio digital. Um dado objeto da aplicação pode estar sendo visualizado simultaneamente em diversas vistas diferentes, com representações diferentes e até de aplicações diferentes. Quando um objeto do modelo da aplicação é modificado, sua(s) vista(s), como efeito colateral, deve(m) receber uma mensagem solicitando modificação e a tela deve ser alterada.

3.2.3.2.4.2. Controlador

Cada vista está associada a um controlador responsável pela recepção de eventos causados pelo usuário através do mouse e teclado. Mensagens adequadas são transmitidas pelo controlador para a sua vista associada indicando a necessidade de alterações na tela e que possivelmente podem ser repassadas aos objetos da aplicação para que sejam modificados. Como há a separação controlador-vista, pode-se ter maneiras diferentes de interagir com o mesmo tipo de representação de uma dada informação. Por exemplo, um texto pode estar sendo visualizado em duas vistas de mesmo tipo; porém em uma ele pode ser editado e na outra a edição pode ser proibida.

É interessante notar que isto ocorre dentro de um sistema orientado a objetos. Objetos modelos, vistas e controladores se associam e se comunicam di-

retamente entre si. Isso significa que não há um módulo que englobe as vistas, por exemplo, fazendo a gerência e triagem da comunicação destas com os modelos e controladores.

Com a manipulação direta, cada objeto apresentado na tela *("output")* pode ser manipulado *("input")* diretamente pelo usuário. A entrada e saída de informação de/para o usuário devem trabalhar em conjunto [HUDS87], objeto a objeto, para haver a ilusão da ação sobre os objetos. A tríade modelo-vista-controlador permite que um dado tipo de apresentação e um tipo de interação sejam associados para manipular um objeto modelo.

3.2.3.2.5. Representações Abstratas em Vistas

Também voltado para a manipulação direta, há o modelo do SGIU Chiron ([YOUN88]). Este também separa as entidades das aplicações (os modelos) de suas representações para o usuário (os artistas). Diferentemente do Smalltalk, cada artista engloba o controle da exibição e da interação com um dado objeto modelo. Porém o que um artista produz é uma representação abstrata da vista *(view abstract depiction)* do objeto modelo. Um terceiro elemento, o arte-finalista da vista *(rendering agent)* é o responsável pela manutenção da correspondência entre a representação abstrata e a representação concreta na tela.

Esta separação permite captar na representação abstrata propriedades perdidas nos bits da imagem na tela, como a estrutura de composição do objeto-vista e quais partes são superpostas por outras. O arte-finalista é então o responsável por saber qual parte de um objeto corresponde a uma dada posição na tela, por exemplo, onde houve um clique no *mouse*.

A existência de uma representação intermediária da informação visualizada, independente do modelo cru desta informação, permite que a visualização leve em conta atributos próprios, independentes dos atributos do modelo. Por exemplo, o modelo não tem informação sobre o leiaute de seus componentes na tela. Vantagens de implementação também são apontadas: maior facilidade para a programação dos artistas, possibilidade de distribuição dos arte-finalistas e representações abstratas nas estações de trabalho e uma gerência mais fina da necessidade de alterações na tela, típica de sistemas para a animação de imagens.

3.2.3.3. Especificação de Diálogo

Para desenvolver a comunicação homem-computador de uma aplicação, um projetista precisa de instrumentos de representação. A variedade destes é muita, indo desde linguagens de programação à simples construção interativa da

interface a partir de elementos predefinidos. Por outro lado, aspectos distintos da comunicação homem-computador precisam ser especificados e precisam de mecanismos distintos para especificá-los.

3.2.3.3.1. O Modelo Lingüístico

O modelo lingüístico divide estes aspectos em três categorias: léxicos, sintáticos e semânticos. Embora esta divisão ainda seja muito utilizada, critica-se o fato deste modelo induzir a arquiteturas *pipelines*, onde um elemento semântico só é ativado após a construção total de um elemento sintático correspondente, e este pode ser composto por diversos elementos léxicos. Questões como a realimentação semântica na manipulação direta esgotam este modelo ao exigir que o usuário seja realimentado de informação semântica antes mesmo que um elemento sintático seja formado. Por exemplo, a movimentação de um objeto, arrastando-o com o *mouse*, pode causar uma reação temporária imediata em outros objetos afetados por cada posição intermediária do objeto sendo arrastado.

Por outro lado, no exemplo de uma linguagem de comando este modelo se mostra adequado. Cada linha de comando lida é analisada lexicamente e sintaticamente, de acordo com uma gramática, reconhecida como correta ou errada, informando isto ao usuário, e a ação semântica correspondente é executada.

3.2.3.3.2. Propriedades Desejáveis

Um instrumento de especificação de diálogo deve ter alguns atributos desejáveis [STRU85]:

- Capacidade de gerar especificações fáceis de serem entendidas. Freqüentemente as especificações não são compreendidas pelas pessoas comuns.
- Facilidade de ser usada. Um mecanismo de especificação deve exigir menos esforço para especificar a comunicação homem-computador do que para implementá-la.
- Capacidade de gerar especificações precisas. Dúvidas sobre o comportamento do sistema não devem restar.
- Capacidade de gerar especificações facilmente verificáveis quanto a consistência.
- Capacidade de expressar comportamentos complexos de um modo o mais simples possível.

- Capacidade de gerar especificações independentes da descrição da implementação, ou seja, deve-se especificar o que fazer e não como fazer.
- Capacidade de gerar protótipos a partir da especificação.

Uma especificação precisa deve ser completa. Para isso acontecer, o instrumento deve prever a especificação de uma grande variedade de detalhes: seqüenciamento das ações do usuário, restrições sintáticas e semânticas, técnicas de interação, elementos gráficos, posicionamento dos elementos na tela, forma dos cursores, restrições na movimentação de cursores, eco, cor, atributos de vídeo, movimentação de objetos, apresentação e conteúdo de mensagens, rolamento, paginação, janelas, etc. Apesar disso, este trabalho se concentra na especificação dos aspectos sintáticos da interação, tais como o seqüenciamento das ações de entrada e saída.

3.2.3.3.3. Especificação Sintática

No modelo lingüístico, a especificação sintática define a estrutura do diálogo. Freqüentemente os instrumentos de especificação sintática são apropriados para diálogos seqüenciais mas não para diálogos não-seqüenciais [BILJ88]. A seguir são listados alguns aspectos não-seqüenciais que devem poder ser tratados pelos instrumentos de especificação sintática de diálogo.

Sentenças entremeadas. Durante a entrada de comandos pelo usuário, as sentenças que formam cada comando podem estar entremeadas, principalmente quando a comunicação homem-computador é imodal. Por exemplo, durante a emissão de um comando, o usuário pode fazer uso de ajuda embutida na aplicação *("help")*, retornando para completar o comando. Neste tipo de permeação, uma sentença interrompe outra, mas é completada antes de ser retomada a sentença original. Um outro tipo diferente de sentenças entremeadas ocorre quando partes de dois comandos são alternadamente fornecidas pelo usuário.

Ordem livre, ou parcialmente livre, das palavras nas sentenças. Os usuários comumente não desejam se preocupar com a ordem em que devem fornecer os parâmetros de um comando.

Entrada concorrente. Em ambientes multitarefas baseados em janelas, as tarefas podem ter entradas de comandos ativas concorrentemente.

Riqueza. Os diálogos podem exigir construções mais complexas que as possíveis de se especificar por máquinas de estados finitos.

Manipulação de condições de interrupção e erros. Os diálogos podem ter a sua seqüência quebrada ou interrompida por eventos ou atitudes errôneas.

Na manipulação direta, a sintaxe é minimizada, usando-se ações físicas diretas (apontar, arrastar...) sobre os objetos individuais. Por isso, a sintaxe deve ser expressa em relação aos próprios objetos e não em relação ao sistema como um todo [HUDS87].

3.2.3.3.3.1. Gramáticas

O uso de gramáticas como metalinguagem para descrever a comunicação homem-computador teve inspiração na área de linguagens de programação, onde são extremamente utilizadas. Enquanto a comunicação homem-computador era essencialmente seqüencial, até o início dos anos 80, as gramáticas foram consideradas úteis. Todavia, com o advento dos diálogos multialinhavados, elas se mostraram inapropriadas. A possibilidade do usuário entrar com várias sentenças simultâneas e entremeadas é difícil de ser especificada através de gramáticas. Entretanto, mesmo nos diálogos alinhavados, as gramáticas podem ser empregadas para descrever cada alinhavo em separado, ou seja, cada sentença, e as partes seqüenciais da comunicação homem-computador.

Por exemplo, o sistema SYNGRAPH usa gramáticas para especificar diálogos com interação gráfica [OLSE83]. O sistema KES, para aplicações em PAC (Projeto Auxiliado por Computador), também é baseado em gramática [HAMM86]. Moran definiu Gramática de Linguagem de Comandos modelando a visão do usuário em níveis [MORA81].

3.2.3.3.4. Linguagem Regular

Uma linguagem é regular se pode ser descrita por alguma gramática regular. Linguagens regulares também podem ser especificadas através de expressões regulares, por autômatos finitos não determinísticos ou determinísticos, grafos e tabelas de transição e diagramas sintáticos regulares.

3.2.3.3.4.1. Expressões Regulares.

As expressões regulares foram estudadas inicialmente por Kleene em 1956. As expressões de fluxo [SHAW80] e as especificações algébricas [CHI85] são extensões formais às expressões regulares para a representação de linguagens gráficas.

3.2.3.3.4.2. Grafos de Transição

Um autômato finito, determinístico ou não, pode ser representado graficamente por um grafo direcionado rotulado, chamado de grafo de transição,

onde os nós são os estados e os arcos representam a função de transição. Em um grafo de transição, os nós são desenhados como círculos e as transições como flechas ligando os nós. Cada transição é rotulada pelos símbolos de entrada que causam a mudança do estado anterior para o posterior da transição. Um dos estados é designado como sendo o estado inicial.

A Figura 2 mostra um grafo de transição de um autômato finito não-determinístico (AFN) que reconhece a linguagem (a|c)*cb. O estado final está representado por um círculo duplo.

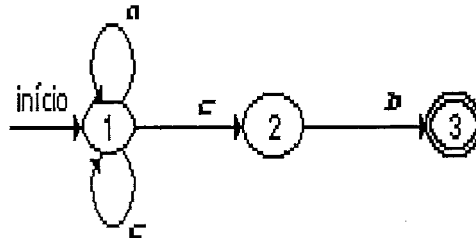


Figura 2: Grafo de Transição de um AFN

3.2.3.3.4.3. Tabela de Transição

Outro modo de se apresentar um autômato finito é através de uma tabela com uma linha para cada estado e uma coluna para cada símbolo de entrada. Uma entrada da tabela contém o conjunto de estados que podem ser alcançados a partir do estado correspondente àquela linha com a entrada do símbolo correspondente àquela coluna. A Figura 3 mostra a tabela de transição do AFN da linguagem (a|c)*cb.

Estados	Entrada		
	a	b	c
0	{0}	-	{0,1}
1	-	{2}	-

Figura 3: Tabela de Transição

3.2.3.3.4.4. Diagrama Sintático

Os diagramas sintáticos são outro modelo de representação de autômatos finitos, com poder de representação equivalente às gramáticas e expressões regulares. Funcionam como uma espécie de fluxograma estilizado. São mais fáceis de se aprender e de se usar que as gramáticas. Em um diagrama sintático, os terminais são representados dentro de círculos e arcos orientados (flechas) definem a ordem em que estes terminais podem aparecer (Figura 4).

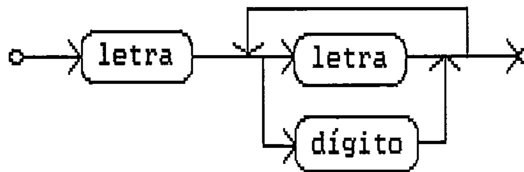


Figura 4: Diagrama Sintático para identificadores

3.2.3.3.5. Linguagem Livre de Contexto

Uma linguagem é livre de contexto (LLC) se puder ser especificada por alguma gramática livre de contexto. O principal uso de gramáticas é para representar linguagens livres de contexto. Mesmo assim, o que costuma ser usado é uma forma simplificada, a BNF *("Backus-Naur Form")*, e suas variações.

As diversas formas de representar linguagens regulares - expressões regulares, grafos, tabelas e diagramas - costumam ser estendidas para poderem especificar linguagens livres de contexto. A idéia base é permitir o emprego de não-terminais em conjunto com os terminais e especificar em separado cada não-terminal empregado.

Por exemplo, os diagramas sintáticos são estendidos permitindo que não-terminais, desenhados como retângulos, sejam usados como se fossem terminais. Subdiagramas definem cada não-terminal.

As expressões regulares também são estendidas de maneira semelhante. Passam a permitir a mistura de terminais com não-terminais em cada expressão e a definição de cada não-terminal através de uma expressão regular deste tipo. A coleção de expressões representando a linguagem é chamada de uma gramática com o lado direito regular *("RRPG - Regular Right Part Grammar")*.

A vantagem destas extensões não é proporcionada apenas por elas poderem definir linguagens não-regulares. Mesmo para linguagens regulares, a possibilidade de se estruturar a linguagem com os não-terminais permite colocar em evidência construções repetitivas da linguagem. Por exemplo, se, para terminar diversos comandos, a linguagem exige do usuário uma atitude indicando uma confirmação ou um cancelamento do comando, este trecho pode ser posto em evidência criando-se um não-terminal para defini-lo.

3.2.3.3.6. Linguagens Sensíveis ao Contexto

Mesmo as linguagens de programação têm aspectos sintáticos sensíveis ao contexto. Um exemplo é a verificação de tipos em uma atribuição, onde os tipos das variáveis dependem do contexto onde se situa a atribuição. Em uma ro-

tina, uma dada variável pode ser do tipo inteiro e, em outra, ela pode estar declarada como real.

3.2.3.3.6.1. Definição Dirigida pela Sintaxe

As definições dirigidas pela sintaxe estendem o conceito de gramática livre de contexto para poderem representar aspectos dependentes do contexto. A estrutura sintática da linguagem ainda é representada por uma gramática livre de contexto. Porém, cada símbolo gramatical, terminal ou não-terminal, é associado a um conjunto de atributos. Por exemplo, um terminal identificador tem atributos indicando se é um identificador de variável ou não, e qual o tipo de dado associado (inteiro, real, etc.). Um não-terminal expressão também precisa de um atributo para indicar o tipo da expressão. Um atributo pode representar qualquer coisa: conteúdo em caracteres, números, posição na tela, etc.

A cada produção são associadas um conjunto de regras semânticas para computar valores com os atributos dos símbolos da produção. As regras semânticas revelam as dependências entre atributos. A avaliação das regras semânticas define os valores dos atributos para uma dada entrada.

A Figura 5 mostra um exemplo de uma definição dirigida pela sintaxe.

Produção	Regras Semânticas
P ::= E	mostre(E.val)
E ::= E1 + T	E.val := E1.val + T.val
E ::= T	E.val := T.val
T ::= T1 * F	T.val := T1.val x F.val
T ::= F	T.val := F.val
F ::= (E)	F.val := E.val
F ::= num	F.val := num.val

Figura 5: Definição Dirigida pela Sintaxe

3.2.3.3.7. Tradução, Diálogo e Interface com a Aplicação

A definição dirigida pela sintaxe é uma notação para gramáticas de atributos. Uma regra semântica pode causar efeitos colaterais, como a saída de informação ou o envio de um sinal. Esta idéia permitiu que, já em 1961, as notações baseadas em gramáticas de atributos fossem usadas para especificar a tradução de uma linguagem para outra [IRON61]. Esta mesma característica permite que gramáticas de atributos possam ser usadas para especificar a relação causa-e-efeito entre as entradas do usuário e a apresentação de informação para ele. Ações como eventos e chamadas de função podem também ser usadas para

definir o relacionamento da interface com a aplicação. A seguir são apresentadas duas notações para especificação de tradução dirigida pela sintaxe que, equivalentemente, podem ser utilizadas para a correspondência entre entrada e saída nas interfaces com usuário.

3.2.3.3.7.1. Esquema de Tradução

Um esquema de tradução é uma definição dirigida pela sintaxe simplificada, onde as ações de saída são enfatizadas. Equivale a uma definição dirigida pela sintaxe, porém a ordem das regras semânticas é explicitada. O esquema de tradução é uma gramática livre de contexto com fragmentos denominados ações semânticas embutidos no lado direito das produções.

Uma ação semântica é representada entre chaves, na posição adequada no lado direito da produção, e deve ser executada durante o reconhecimento da entrada, como em

soma ::= + *soma* {*emite*('+')} *resto*

O lado direito desta fórmula indica que a ação semântica {*emite*('+')} deve ser executada após o reconhecimento de uma soma, porém antes do reconhecimento de resto.

3.2.3.3.7.2. Gramáticas de Transformação

As gramáticas de transformação são usadas para se definir formalmente a tradução de uma linguagem para outra, como feito pelos compiladores ao transformar a linguagem fonte na linguagem de máquina. Elas se diferenciam das gramáticas tradicionais por terem dois lados direitos. Em uma produção, cada lado direito deve estar coordenado com o outro, ou seja, cada não-terminal do primeiro lado direito deve estar presente no segundo e vice-versa. Os terminais do segundo lado direito podem ser de outro vocabulário, o da linguagem destino. Eles também podem representar ações, como eventos ou a gravação de um símbolo em um arquivo.

As gramáticas de transformação livres de contexto são de especial importância. Do mesmo modo que nas gramáticas livres de contexto comuns, o lado esquerdo de cada produção sua será composto por um símbolo não-terminal. A correspondência biunívoca entre os não-terminais dos lados direitos é que permite se derivar duas seqüências simultaneamente. Por exemplo, a gramática de transformação a seguir especifica a tradução de expressões para a forma prefixa:

E ::= E + T ⇒ + E T
E ::= T ⇒ T
T ::= (E) ⇒ E
T ::= a ⇒ a

3.2.3.3.7.3. Diagramas de Transição

Diagramas ou redes de transição têm poder de expressão equivalente ao das gramáticas [PARN69] [WOOD70] [FOLE84] e são usados para especificar interfaces com usuário, pelo menos, desde 1968 no Reaction Handler de Newman.

Um diagrama de transição é um grafo onde cada nó corresponde a um estado na conversação e os arcos definem o que provoca uma transição de estado: entradas que o usuário pode fazer ou saídas para o usuário que o sistema pode dar. Uma variação de diagramas de transição é usada como técnica de especificação de diálogo no sistema Rapid [WASS85A] [WASS85B].

Jacob também sugere o uso de diagramas de transição [JACO86] onde os elementos de entrada *("tokens")* são distingüidos por iniciarem com uma letra "i" minúscula (de *input* - entrada) e os elementos de saída iniciam com um "o" (de *output* - saída). Os outros identificadores representam não-terminais e são associados a subdiagramas [GREE85a] (Figura 6).

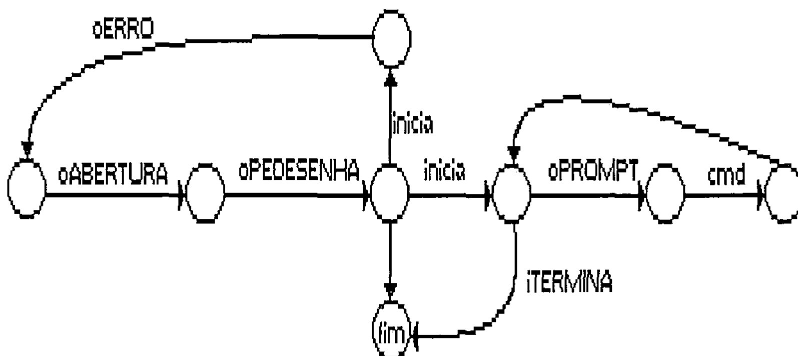


Figura 6: Diagrama de Transição

Em uma comunicação homem-computador baseada na manipulação direta, a cada instante o usuário tem diversas opções de interação; cada opção leva-

ria a um outro estado com outras opções diferentes. Esta situação não seria especificada de modo simples em um modelo baseado em transições de estado, como nas gramáticas e nos diagramas de transição.

3.2.3.3.8. Concorrência

Com relação a interfaces concorrentes pode-se classificar três aspectos: entradas multidispositivos, entradas multialinhavadas e suporte à multitarefa [Tann87]. Quando há entrada multidispositivo, o usuário tem, a cada momento, diversos dispositivos a sua disposição [JACO86].

Com a entrada multialinhavada *("multi-thread input")* [HILL87b], o usuário pode conduzir diversas conversações simultaneamente. Estas conversações podem ser aninhadas ou não. Cada fluxo de entrada é processado léxica e sintáticamente em separado, dependendo do contexto (em geral, da janela). Os contextos também podem ser criados e destruídos dinamicamente, sendo impossível predizer, em tempo de definição de diálogo, o número de contextos. Outro tipo de multialinhavamento ocorre quando há a possibilidade do usuário manusear simultaneamente múltiplos dispositivos, utilizando por exemplo, as duas mãos. Um exemplo de instrumento para a especificação de entrada multialinhavada é a linguagem ALGAE [FLEC87].

Quando há suporte a multitarefa *("multi-tasking support")* [LANT87] cada contexto processa um alinhavo, ou seja, uma conversação, assincronamente. Neste caso, cada janela, por exemplo, processa o fluxo de suas entradas em paralelo com as outras janelas. O sistema Sassafras [HILL86] suporta entrada multialinhavada concorrentemente, tendo sido experimentado um editor gráfico com entrada pelas duas mãos.

3.2.3.3.8.1. Gramáticas Estendidas

Para lidar com interações entremeadas *("multi-thread")* Scott e Yap propuseram uma notação baseada nas gramáticas livres de contexto acrescida de duas extensões: operadores de bifurcação *("forks")* e atributos de contexto [Scot88]. O efeito dos atributos de bifurcação é a criação de subdiálogos que rodam em paralelo com o diálogo original.

3.2.3.3.8.2. Redes de Petri

Por sua aplicabilidade como modelo formal para sistemas concorrentes e assíncronos, Redes de Petri são uma possível técnica de modelagem de diálogos multialinhavados e concorrentes [DEHN80] [PILO83].

Uma Rede de Petri é um grafo direcionado com arcos conectando dois tipos de nós: lugares e transições [PETE81]. Os lugares, representados por círculos, só podem ser conectados a transições, representadas por barras, e vice-versa. Cada lugar pode significar um estado do sistema, enquanto as transições podem significar eventos.

Os lugares podem ser marcados com *tokens*, representados por bolas pretas dentro dos lugares. Um dado arranjo de tokens é chamado de uma marcação da rede. Uma rede pode estar associada a duas marcações especiais: a inicial e a final. Uma rede marcada é conhecida como uma Rede de Petri Marcada (RPM).

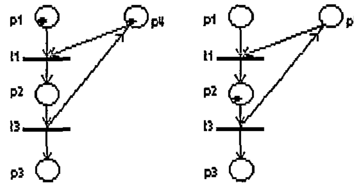


Figura 7: Disparo de uma Transição em uma Rede de Petri

A execução de uma Rede de Petri Marcada se dá dispondo-se os *tokens* na marcação inicial e disparando-se transições até ser alcançada a marcação final. Uma transição dispara caso haja pelo menos um *token* em cada lugar entrando na transição. Quando uma transição dispara, em cada lugar de entrada na transição é retirado um *token* e em cada lugar de saída da transição é colocado um *token*, modificando-se a marcação da rede (Figura 7).

3.2.3.3.8.3. Redes de Petri Rotuladas

Pode-se dizer que uma RPM define (aceita ou gera) uma linguagem quando se associam cadeias desta linguagem às seqüências de disparos. A maneira desta associação ser feita é se rotulando as transições com elementos de um alfabeto *E*.

Definição: Uma Rede de Petri Rotulada é uma tripla composta por uma Rede de Petri Marcada, um alfabeto finito de símbolos contendo ou não a seqüência vazia ε e uma função *R* de rotulação associando um elemento do alfabeto a cada transição.

Uma seqüência de disparos *w=t1t2...tn* define uma sentença *R(w)=R(t1)R(t2)...r(tn)*. A linguagem gerada por uma *Rede de Petri Rotulada RPR* é

$L(RPR) = \{R(w) \mid m_i[w>m_f\}$.

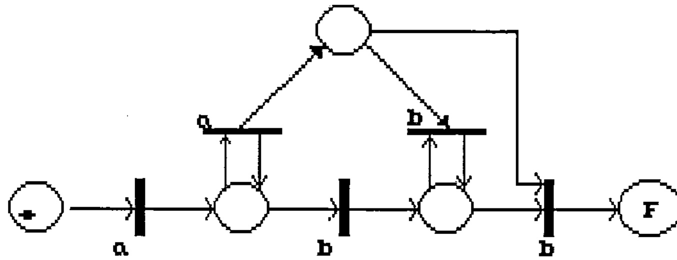


Figura 8: Rede de Petri Rotulada gerando a linguagem $a^n b^n$

Para que as RPRs funcionem como reconhecedores de linguagens, deve-se modificar a regra de disparo, de modo a que ele só se realize se a transição em jogo tiver como rótulo o elemento seguinte da entrada.

> Seja $x=aw$ a seqüência de entrada, onde a pertence ao alfabeto e w é uma seqüência de elementos do alfabeto. A transição t disparará se,
>
> para todo lugar p onde $V(p,t)=1$, $m(p)>0$ e $a=R(t)$.
>
> Se t disparar, haverá uma nova marcação m' como definida anteriormente e a nova seqüência de entrada será w (Figura 8).

Normalmente as Redes de Petri permitem que disparos de transição ocorram simultaneamente. As linguagens resultantes nessa situação, denominadas de <u>linguagens de subconjuntos</u> [ROZE83], são muito mais complexas que as linguagens definidas por RPRs que só permitem um disparo de transição a cada vez. Entretanto, geralmente nada se perde quando se considera que a entrada seja serializada (Figura 9).

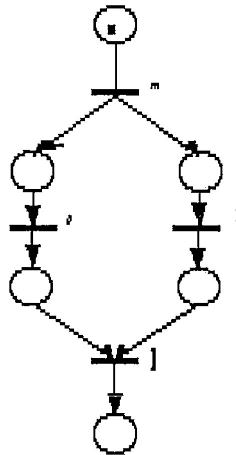


Figura 9: especificando a ordem livre de parâmetros

3.2.3.3.8.4. Redes de Petri Aninhadas

As Redes De Petri Aninhadas (RPAs) foram sugeridas por van Biljon como extensão às Redes de Petri Rotuladas visando dar maior poder de representação de linguagens.

A função de rotulamento de uma RPR é estendida para que possa incluir como rótulos nomes de outras sub-redes. Uma transição com um destes rótulos disparará se a sub-rede chegar ao final com a entrada disponível.

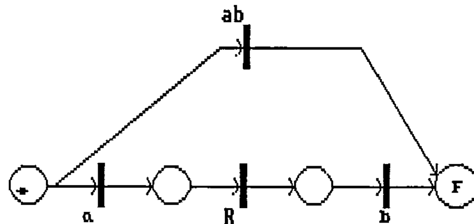


Figura 10: Rede de Petri Aninhada Recursiva R

É fácil notar que as RPAs (Redes de Petri Aninhadas) reconhecem pelo menos as linguagens livres de contexto, bastando fazer uma sub-rede para cada não-terminal. A Figura 10 exemplifica uma RPA recursiva, denominada R, aceitando uma cadeia $a^n b^n$.

3.2.3.3.8.5. Redes de Petri para E/S

Redes de Petri para E/S (RPES) foram sugeridas por van Biljon, como extensão às Redes de Petri Aninhadas, visando a representação dos relacionamentos entre a entrada, a saída de dados e as aplicações. O objetivo desta extensão não é aumentar o poder de reconhecimento de linguagens da rede. O alvo é a modelagem da função de tradução realizada pela interface com usuário dos dados entrados pelo usuário para os dados saídos para o usuário e para as aplicações.

A extensão consiste na modificação da função de rotulamento para que ela inclua não apenas símbolos de entrada, mas também de saída. Como tanto a entrada como a saída de dados podem ser realizadas através de diversos dispositivos diferentes, cada símbolo entrado ou saído estará associado a um dispositivo (ou a uma aplicação).

3.2.3.3.8.6. Interrupções e Erros

A possibilidade de um comando interromper outro e a necessidade de se modelar o comportamento da interface em função de entradas errôneas motivaram van Biljon a sugerir novas extensões a Redes de Petri [BILJ88].

3.2.3.3.8.7. Orientação a Acessos

O termo programação orientada a acessos vem da sua característica de que o acesso a um dado pode causar efeitos colaterais. Em Loops, isto é obtido através do mecanismo de anotações [STEF86a] [STEF86B]. O sistema Peridot [MYER88] usa valores ativos que, ao serem alterados, imediatamente atualizam os objetos gráficos a eles associados. O sistema de depuração simbólica Incense [MYER83] definia o conceito de *artistas*, onde, para cada tipo de dados, eram agrupadas decisões sobre como exibi-lo.

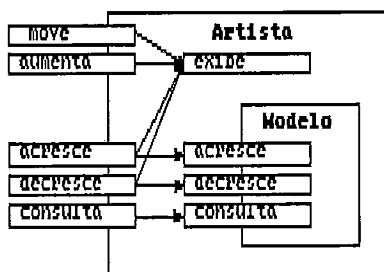


Figura 11: Artistas estendem os tipos abstratos de dados, acrescentando efeitos colaterais ("exibe") às operações

O subsistema de gerência de interface com usuários Chiron, do projeto de ambiente de software Arcadia-1 [YOUN88], usa uma variação de anotações para associar artistas a tipos de dados abstratos. Um artista estende um tipo abstrato de dados com novas operações e novos estados locais (variáveis de instância) para controlar a exibição de um objeto; também estende as antigas operações do tipo abstrato de dados para que elas atualizem a visualização do objeto, quando ele for modificado (Figura 11). O tipo abstrato de dados que receber anotações não terá sua semântica e sintaxe alteradas por elas, de modo que a exibição de um dado não implica na alteração da definição de seu tipo. Um tipo de dados pode ser estendido por vários artistas, permitindo múltiplas visões do mesmo dado.

Embora um artista encapsule a descrição abstrata de uma representação visual de um tipo de dado, a descrição concreta de como essa representação é fisicamente materializada na tela pertence ao SGIU. Esta divisão pode ser exemplificada pela visualização de um diagrama qualquer: o SGIU sabe como cada elemento do diagrama é visualizado; mas onde cada elemento é mostrado está na representação abstrata do objeto. O usuário ao deslocar algum elemento do diagrama está alterando apenas a representação abstrata da visualização do diagrama. O SGIU é o responsável por mapear pontos da tela em objetos correspondentes (por exemplo, para manipulá-los com o mouse).

3.2.3.3.8.8. Modelo por Eventos

Muitas variedades de instrumentos de representação baseados em eventos são empregadas para lidar com a comunicação homem-computador imodal, ou seja, não voltada a estados de interação. Neste caso, a comunicação homem-computador é vista como um conjunto de eventos e de manipuladores de eventos. Os eventos são gerados pelo usuário ao interagir com um dispositivo de entrada. O conjunto de manipuladores de eventos ativos em um dado instante define o leque disponível de ações que o usuário pode tomar neste instante.

No modelo por eventos, as transições de estado não são especificadas explicitamente [HAYE85a]. Quase sempre os manipuladores de eventos são especificados procedimentalmente, de modo semelhante às linguagens de programação orientadas a objeto.

Os manipuladores de eventos são, na grande maioria das vezes, objetos gráficos, tais como cardápios, barras de rolamento, botões, barras de cardápios, folheadores *("browsers")*, etc. A especificação de um manipulador de eventos envolve a presença simultânea de informações vindas do teclado, mouse ou de outros dispositivos de entrada.

Instâncias de manipuladores de eventos são criadas em tempo de execução. Um manipulador de eventos recebe eventos de entrada e gera eventos de saída após alterar seu estado interno. Os manipuladores de eventos, vistos como objetos gráficos de interação com o usuário, são os tijolos da comunicação homem-computador. Muitos sistemas compõem a comunicação homem-computador pela combinação destes objetos criando-se novos objetos [APPL85] [X11 87].

Os modelos por eventos são muito utilizados para representar estilos de diálogo assíncrono. Todavia, um dos seus pontos marcantes é poderem ser usados para representar diálogo seqüencial [GREE86].

O University of Alberta UIMS [GREE85b], o COUSIN [HAYE83] [HAYE85a] [HAYE85b], Sassafras [HILL86] [HILL87b] e IMAGES [SIMO87] servem como exemplos de SGIUs que usam modelos por eventos. O University of Alberta UIMS utiliza uma linguagem de programação de evento com sintaxe baseada em C. A declaração de um manipulador de eventos se compõe de três partes: indicação dos tipos de *tokens* de entrada e de saída tratados por esse manipulador; declarações de variáveis locais; e rotinas tratadoras de eventos, com comandos em C.

Exemplos de modelos de especificação por eventos:

- Benbasat usa tabelas de descrição de eventos equivalentes às máquinas de estados finitas [BENB84].

- A linguagem Squeak [CARD85].
- A ERL *("Event Response Language")*, usada no SGIU Sassafras [HILL86], permite representar atividades concorrentes. É uma linguagem baseada em regras onde são especificadas condições que devem ser mantidas ou ações a serem ativadas quando as condições furarem. Entretanto, a ausência de modularidade da linguagem tem sido criticada.
- Clément e Incerpi experimentam o uso da linguagem Esterel para especificar objetos gráficos como cardápios, botões, etc. [CLÉM88].

3.2.3.3.8.9. Linguagens Orientadas a Eventos

Linguagens voltadas à manipulação de eventos promovem um paradigma de programação diferente, que aqui chamaremos de orientação a eventos (ou sinais). Exemplos deste tipo de linguagem são as linguagens ALGAE [FLEC87] e Esterel. A sigla ALGAE significa uma linguagem para a geração de manipuladores de eventos assíncronos *("A Language for Generating Asynchronous Event handlers")*. A seguir serão mostrados diversos aspectos da linguagem Esterel.

Um programa Esterel é um conjunto de módulos, processando em paralelo, se comunicando por sinais difundidos *("broadcast")*. Os módulos são verdadeiros autômatos de estados que se alternam de acordo com os sinais que chegam e nesse afazer emitem novos sinais. Os sinais carregam dados, de modo semelhante ao que fazem as mensagens das linguagens de programação orientadas a objetos. Mas, diferentemente destas, o estado de uma aplicação é normalmente representado por estes dados que se encontram dentro dos sinais.

A orientação a sinais promove uma programação bastante modular. Um módulo desconhece tanto quem enviou um sinal, como quem o receberá e tratará. A maior parte da informação a ser manipulada deve ser enviada sob a forma de sinais. Os tradicionais **if-then-else** também não são tão populares em Esterel, sendo substituídos por processamento do sinal pelo autômato.

Por exemplo, a especificação de um disparador[21] (objeto gráfico, usado como dispositivo de comunicação homem-computador, que quando acionado com o mouse dispara uma atividade) pode ser apresentada, como a seguir, por uma pseudolinguagem semelhante à Esterel.

---

[21]Corresponderia a *widget* no sistema X-Windows [X11 87] ou a *control* em sistemas como Ms-Windows ou Macintosh [APPL85].

O fundamental do comportamento de um disparador é o fato de que, se um botão do mouse for apertado dentro da região definida pelo desenho do disparador, este se iluminará (ficará em reverso) até o botão do mouse ser solto dentro da região do disparador, quando este voltará ao seu desenho normal e ativará a ação associada ao disparador.

Considerando-se que o botão do mouse está pressionado dentro do disparador, um evento de soltura do botão seria tratado assim:

aguarda SolturaDoBotão, **quando:**
  desilumina o disparador;
  gera EventoAtiveAção(disparador)
  pronto.

Porém, se antes do botão ser solto, o mouse sair da região do disparador, este será desiluminado para informar ao usuário que o disparador não está sentindo o mouse (e para dar uma chance do usuário desistir de ativar a ação). Se o mouse voltar ao disparador, este voltará a se iluminar.

Esta tarefa de iluminar/desiluminar pode ser especificada como outro processo, paralelo ao que espera a soltura do botão do mouse. Considerando-se que a entrada e saída do mouse na região do disparador gere eventos, este processo seria:

repete:
    desilumina o disparador;
    aguarda SaídaDoMouse;
    ilumina o disparador;
    aguarda EntradaDoMouse;

Caso o evento de soltura do botão do mouse ocorra fora do disparador, nada precisará ser feito, a não ser terminar o processamento do disparador. Considerando-se que a soltura do botão do mouse pode ocorrer dentro ou fora do disparador, tem-se:

repete:
    repete até SaídaDoMouse:
        aguarda SolturaDoBotão, **quando:**
            desilumina o disparador;
            gera EventoAtiveAção(disparador)
            pronto.
    repete até EntradaDoMouse:
        aguarda SolturaDoBotão, **quando:**
            pronto.

Uma terceira tarefa deve rodar em paralelo a essas: a que verifica se o mouse está dentro ou fora do disparador, para gerar os eventos adequados. Como esta tarefa é útil não só para disparadores, sendo necessária para qualquer outro objeto gráfico, um novo módulo, mais genérico, deve ser programado.

Algumas construções interessantes da Esterel são:

- **await** sinal;
  suspende a execução de um processo até a chegada do sinal.
- **do** comandos **upto** sinal;
  executa os comandos repetidamente, pelo menos uma vez, até que chegue o sinal.
- **emit** sinal;
  emite por difusão *("broadcast")* o sinal.
- **trap** exceção **in** comandos **end;**
- **exit** exceção;
  o comando **trap** define um bloco de comandos onde pode ocorrer a exceção; o comando **exit** emitido dentro desse bloco terminará a execução do mesmo.
- **every** sinal **do** comandos **end;**
  executa os comandos toda vez que o sinal chegar; enquanto não chega, aguarda.
- **present** sinal
  then comandos
  else comandos **end;**
  executa um grupo de comandos ou outro dependendo de se o sinal foi emitido.
- [comandos || comandos]
  executa os dois grupos de comandos em paralelo.

3.2.3.3.9. Restrições

Outras abordagens mais declarativas têm sido estudadas para a especificação da comunicação homem-computador. Dentre elas há a especificação algébrica e as especificações baseadas em restrições. Uma restrição é uma relação que deve ser mantida verdadeira pelo sistema, mesmo quando dados e objetos se alterarem. O projetista especifica que restrições devem ser observadas e ao sistema cabe resolver como observá-las.

Uma restrição gráfica ocorre entre objetos gráficos, como, por exemplo, um texto centralizado em um retângulo. Uma restrição de dados define um relacionamento entre um objeto gráfico e os valores de um dado.

A declaração de restrições são usadas, por exemplo, nos sistemas Thinglab [BORN86] [BORN81], Sassafras [HILL86], Juno [NELS85] e IDL [FOLE87].

3.2.3.3.10. Especificação Conceitual

As técnicas descritas acima são predominantemente para a especificação sintática da comunicação homem-computador. A linguagem IDL [FOLE87] permite se especificar conceitualmente uma comunicação homem-computador em um nível mais alto, permitindo a abstração da sintaxe e das técnicas de interação. Uma dada especificação pode ser depois materializada em diversas interfaces com usuário, com sintaxes e técnicas de interação diferentes geradas através do auxílio de um SGIU. Além disso, uma especificação pode sofrer alterações, geradas algoritmicamente, que impliquem em interfaces com usuário diferentes, mas funcionalmente equivalentes.

A linguagem IDL utiliza os conceitos de classificação (classes e instâncias), generalização (subclasses e superclasses) e de restrições aplicadas às ações que um objeto pode sofrer (precondições e poscondições). Uma precondição associada a uma ação é um predicado que só permite que esta ação esteja disponível para ser escolhida pelo usuário, e aí ser realizada, caso a precondição seja satisfeita. Uma poscondição associada a uma ação indica que, após a realização desta, a poscondição prevalecerá. Normalmente as poscondições atribuem valores a variáveis usadas nas precondições.

Estes dois tipos de restrições especificam o mínimo necessário da semântica da aplicação para dar sensibilidade ao contexto na apresentação de cardápios, na ajuda interativa e etc. Os tipos de informação manipulados nestas restrições incluem:

- O número de objetos existentes em um dado conjunto.
- O número de objetos e suas classes que formam o conjunto correntemente selecionado.
- O valor de um parâmetro implícito.

Foram abordados nesta seção características gerais de modelos usados para a representação de informação em bancos de dados, linguagens de programação e sistemas de interface com usuário.O objetivo foi reunir alternativas para a definição do modelo de objetos do GEOTABA, que visa proporcionar unidade e coerência entre os diversos prismas com que a informação deve ser manipulada. A seguir são apresentadas algumas questões relativas a esta integração e sumarizadas as características desejáveis que um modelo de objetos deve possuir para atender um sistema de objetos como o GEOTABA.

3.3. Conflitos de Representação

A interface entre uma linguagem de programação e bancos de dados tem sido tratada, tradicionalmente, através de chamadas a sub-rotinas ou através do embutimento de uma linguagem na outra. O uso de duas linguagens apresenta problemas referidos na literatura como *impedance mismatch* [MOSS88b] [FORD88] [BLOO87]. Linguagens de programação de banco de dados têm sido construídas, buscando a solução destes problemas, integrando completamente as facilidades de linguagens de programação e de bancos de dados.

Esta integração passa pela questão de como relacionar os sistemas de tipos de dados para programação e o modelo de dados. Outro problema se refere a como identificar bases de dados, isto é, dados persistentes e compartilháveis. Além disso, deve-se definir como nomes internos ao programa serão relacionados aos nomes externos das bases de dados.

Nas linguagens orientadas a valores, estes possuem um tempo de vida restrito à ativação do segmento de código na qual uma variável que o contém foi declarada. Objetos com tempo de vida independente do escopo de variáveis têm a sua persistência controlada pelo uso de primitivas de alocação e dealocação de memória. A dealocação explícita de um objeto é perigosa [ATKI87]. Para dados que devem durar além da execução do programa, não há uniformidade de tratamento para todos os possíveis tipos de dados. Por exemplo, em Pascal, isto é obtido através do tipo de dados *file*, parametrizável para alguns tipos, não todos. Esta situação não é considerada adequada [ATKI83]. Algumas linguagens de programação suportam persistência apenas para um subconjunto de seus tipos; outras apelam para uma sublinguagem diferente para dados armazenados; outras criam novos tipos para estes dados; outras armazenam toda a sua memória de trabalho. Atkinson e Buneman pretendem que o projeto de linguagens devam seguir os seguintes princípios:

1. Persistência deve ser uma propriedade de valores arbitrários e não ser limitada a determinados tipos;
2. Todos os valores devem ter os mesmos direitos à persistência;
3. Enquanto um valor persistir, sua descrição (tipo) também persistirá.

"Uma linguagem na qual o programador tem que incluir comandos explícitos para iniciar ou transferir objetos de dados não está de acordo com o requisito de que o código para manipular um objeto não deve depender de sua persistência" [ATKI87].

Nas linguagens de programação, um tipo (*integer*, por exemplo) representa um conjunto de valores, porém este conjunto (que pode ser infinito) não pode ser manipulado. Há uma diferença considerável entre o uso de tipos

em linguagens de programação e em diversos bancos de dados. Nestes, por exemplo, o nome *Empregado*, não só descreve um tipo, mas também representa o subconjunto dos valores desse tipo correntemente armazenados na base de dados. Esta definição causa dificuldade na integração entre linguagens de programação e bancos de dados [BUNE86].

3.4. Características de Modelos de Objetos para Sistemas de Objetos

Conforme definido neste trabalho (ver 3.1.2.), o modelo de objetos para um sistema de objetos se caracteriza primordialmente por não abranger apenas a representação conceitual dos dados e de sua manipulação, como é o caso dos modelos de dados para SGBDs, mas também a sua representação a nível de implementação e a sua representação para interação com usuário. Com isso um modelo de objetos cumpre um papel integrador das tarefas normalmente exercidas por SGBDs, linguagens de programação e sistemas de interface com usuário.

Baseado em exigências feitas a SGBDs orientados a objetos para suportar aplicações de Engenharia de Software [CATT91], pode-se listar os seguintes requisitos para o modelo de objetos:

- *Identificação única de objetos.* Essencial para objetos sem identificadores significativos aos seres humanos, ou que possam ser mudados (exemplo: nomes de módulos).
- *Objetos compostos.*
- *Manutenção de relacionamentos.* O sistema deve manter a integridade das referências entre objetos.
- *Hierarquização de Tipos.* Permite a definição de novos tipos através da especificação das diferenças entre tipos já existentes.
- *Procedimentos associados a dados.*
- *Encapsulamento das estruturas de implementação dos objetos.* Visa a reusabilidade, interoperabilidade e modularização.
- *Listas e conjuntos ordenados.*
- *Facilidade para realizar mudanças no esquema.*
- *Capacidade de programação de procedimentos.*
- *Linguagem simples para consultas.* Permite que o usuário faça consultas diretas ao sistema de objetos, para pesquisar, por exemplo, sobre documentos ou programas.
- *Independência de dados.* É desejável que se possa mudar os dados armazenados sem que se tenha que modificar os programas de aplicações existentes.

- *Independência da Persistência de Dados*. (ver 3.4.1.:"Persistência").

3.4.1. Persistência

Embora seja difícil encontrar uma definição precisa do que venha a ser persistência, alguns princípios relativos à persistência são enumerados por Atkinson e Buneman [ATKI87]. Eles objetivam criar independência entre o código que manipula um dado valor, do fato se este valor é ou não persistente. É recomendado que:

1. Persistência seja uma propriedade de um valor ortogonal ao seu tipo. Ou seja, um tipo não indica se os seus valores são persistentes ou não.
2. Se um valor for persistente, toda informação que o descreve também deverá sê-lo. Isto evitará que um valor seja armazenado com um tipo e recuperado com outro.

Uma linguagem deve permitir que o seu usuário se abstraia da movimentação de dados entre meios de armazenamento. Ou seja, a necessidade de se escrever/ler de arquivos explicitamente gera dependência entre o código de manipulação de um objeto e o fato de que ele seja persistente. A leitura/escrita deve ser inferida pelo sistema.

Observe-se porém que, no caso de um sistema que proveja extensibilidade na implementação do armazenamento dos dados, deve haver abertura para que transferências explícitas sejam realizadas. Isto pode ser necessário, por exemplo, quando se deseja modificar a maneira que um objeto pode ser armazenado.

3.4.2. Extensibilidade

Os sistemas de informação baseados em um SOO terão como componentes objetos gerenciados pelo SOO. O desenvolvimento e a manutenção de um número sempre crescente de aplicações, cada uma incorporando operações específicas ao sistema, impõe a este requisitos de extensibilidade. Seu modelo de objetos não pode ser restrito a um número predeterminado de tipos padrão de objetos e de coleções. Do mesmo modo, para um dado tipo de objeto ou coleção sempre se poderá criar novos modelos de implementação, sem prejuízo dos objetos e coleções já existentes. Novas representações destes objetos, para interface com usuário, também poderão ser acrescentadas ao sistema.

3.5. Sumário

A integração da gerência de interface com usuário e da gerência de bases de dados é um dos pontos fortes da idéia de SOO aqui apresentada. Um SOO, ao gerenciar não apenas dados usados pelas aplicações, mas também o comportamento destes dados e as próprias aplicações, deve fornecer soluções para as questões acima relativas a gerência de interfaces com usuário. O modelo de objetos de um SOO deve incluir mecanismos de modelagem de objetos de interface com usuário e de definição do relacionamento destes com os demais objetos das aplicações. O paradigma da orientação a objetos pode ser usado aqui, como em diversos outros sistemas [BANC88] [KIM90a] [FISH88] [MAIE86], por ser adequado para a implementação de interfaces com usuário por manipulação direta [MONT92a].

Mesmo os SGBDs de terceira geração não denominados orientados a objetos incluem as noções de identidade de objetos implícita, encapsulamento, herança e polimorfismo. O modelo de objetos do GEOTABA, do mesmo modo, deve adotar da orientação a objetos conceitos que lhe proporcionem maior manutenibilidade e reusabilidade dos objetos (dados + comportamento) gerenciados.

O trabalho de definição de um modelo de objetos para o Sistema de Gerência de Objetos do TABA envolve o estudo de diversos aspectos referentes à orientação a objetos e aos sistemas orientados a objetos [TAKA88], bem como a sua comparação com modelos de dados e sistemas de banco de dados convencionais [SMIT87].

Os princípios fundamentais de um SOO são os seguintes: o sistema deve gerenciar tanto os dados como os procedimentos, dando ênfase à manutenibilidade e à reusabilidade; deve possibilitar a modelagem conceitual da informação; deve possuir um sistema de gerência de interfaces com usuário incorporado à definição e manipulação de objetos; e ser extensível. Cada um destes princípios, encontra sua vertente tecnológica respectivamente em (1) linguagens de programação de banco de dados, (2) paradigma da orientação a objetos, (3) modelos semânticos de dados, (4) sistemas de gerência de interface com usuário e (5) bancos de dados extensíveis. Os apectos relativos à representação da informação foram discutidos neste capítulo, além de ter sido apresentado um levantamento sobre o tema de casamento entre banco de dados, programas e interfaces com usuário e sumarizada as características de modelos de objetos para sistemas de objetos.

Capítulo 4

O Modelo de Objetos do GEOTABA

4.1. Introdução

Este capítulo define um modelo de objetos específico para o sistema de objetos GEOTABA. O TABA, sendo um ambiente de desenvolvimento configurável por um meta-ambiente nele contido, capaz de gerar ambientes próprios para cada tipo de desenvolvimento, inclusive para a geração de novas ferramentas de desenvolvimento de software, lidará com informações de diversos tipos. Conhecimento de especialistas em desenvolvimento deverá ser manipulado pelo meta-ambiente e outros ambientes. Métodos de desenvolvimento e até modelos de dados terão que ser representados.

Ferramentas podem lidar com suas informações através de modelos próprios. Estes modelos têm que poder ser representados pelo modelo de objetos do TABA. Uma estratégia para a definição de tal modelo envolveria a pesquisa de um modelo no qual pudessem ser unificados todos os conceitos envolvidos nos modelos utilizados pelas ferramentas que compõem, ou viriam a compor, o TABA. Entretanto, como já foi observado, o TABA evolui dinamicamente, gerando suas próprias ferramentas. Isto significa dizer que as ferramentas existentes são usadas para gerar outras ferramentas que, uma vez desenvolvidas, passam a fazer parte também do ambiente.

Assim, a estratégia de definição de um modelo unificado, além de inviável, é inadequada. Com esta abordagem, limitar-se-ia a capacidade do modelo em questão a suportar um conjunto específico de ferramentas que pudessem vir a ser previamente definidas.

Uma outra alternativa para a definição de um modelo envolve a pesquisa de um modelo que possa ser extensível. Um modelo com tais características deve possuir um conjunto básico de conceitos, através dos quais outros modelos possam ser definidos.

A informação gerenciada pelo GEOTABA é organizada de acordo com um Modelo de Objetos [MONT91a] que engloba regras de definição de estruturas e de procedimentos de manipulação da informação, além de regras de definição da representação para o usuário e de técnicas de interação. O Modelo de Objetos do GEOTABA, conforme proposto, é uma ferramenta de alto nível semântico, capaz de ser usada diretamente na modelagem de sistemas e da informação por eles manipulada. Apesar disso, este modelo é implementado diretamente pelo sistema de gerência de objetos GEOTABA. Este atendimento simultâneo ao homem e à máquina deve privilegiar os fatores humanos, correspondendo à tendência atual.

Modelos de objetos definem como, em um determinado sistema de gerência de objetos, pode-se estruturar e manipular objetos. Como objetos são unidades de informação que englobam dados e procedimentos, pode-se ver o conceito de modelos de objetos como sendo uma extensão do conceito de modelos de dados.

O encapsulamento de propriedades oferece independência de implementação e de complexidade tanto a nível estrutural quanto comportamental. O Modelo de Objetos do GEOTABA [MONT91a] é baseado no encapsulamento de objetos em várias camadas, segundo um modelo de referência com quatro níveis de abstração: i) implementação, ii) conceitual, iii) externo e, iv) de representação.

O nível de implementação diz respeito a como o Sistema de Gerência de Objetos efetivamente manipula e implementa um objeto. No nível conceitual (semântico), um objeto é descrito da forma como é percebido no mundo real. O nível externo define as possíveis perspectivas específicas com que cada usuário vê o objeto. O nível de representação de um objeto refere-se às várias formas pelas quais este pode ser exibido e manipulado, nos diversos meios de comunicação com o usuário.

No nível conceitual, os objetos são modelados segundo descrição de seus atributos, serviços, restrições de integridade e relacionamentos. No nível de implementação, os objetos são modelados segundo os conceitos da orientação a objetos, ou seja, através dos conceitos de objeto, classe, variáveis de instância e métodos.

O modelo prevê um mapeamento automático do nível conceitual para o nível de implementação, porém, alterações posteriores poderão ser realizadas diretamente no nível de implementação. Um exemplo de alteração que pode ser efetuada, é quando um atributo não deve ser mapeado em uma variável de instância e sim calculado através de um método. O objetivo desta separação dos

níveis conceitual e de implementação é o de garantir a independência de dados, recurso extremamente vantajoso em SGBDs.

4.2. Modelos de Objetos e Ambientes de Desenvolvimento de Software

A estação de trabalho TABA permitirá o desenvolvimento de software em ambientes de desenvolvimento configuráveis de acordo com as características da aplicação a ser desenvolvida, dos desenvolvedores e dos usuários da aplicação. O processo de desenvolvimento de software envolve, em suas etapas, a construção de diversos tipos de diagramas, documentos e descrições de dados e procedimentos (rotinas).

O Gerente de Objetos do TABA GEOTABA se diferenciará de um SGBD convencional porque, além de gerenciar os dados em uso na estação de desenvolvimento, também controlará as rotinas (funções, ferramentas, etc.) que os manipulam. Na verdade, estas rotinas são, talvez, os principais habitantes do ambiente de desenvolvimento. Muitas vezes as próprias rotinas serão vistas como dados, e também terão outras rotinas especializadas para tratá-las.

A atividade de desenvolvimento de software lida com abstrações. Ao contrário do software desenvolvido para um ambiente convencional, que lida com dados representando entidades pré-existentes, o software para desenvolvimento de software lida com entidades abstratas, inventadas pelos engenheiros, projetistas e programadores de software.

Além disso, o desenvolvimento de software pode ser realizado de forma exploratória. Ou seja, as pessoas inventam entidades, mas também desinventam-nas (desistindo de usá-las) ou alteram-nas com alta frequência (daí o próprio nome: "software").

Estas entidades poderiam ser vistas convencionalmente como dados e programas, ou algo assim. Isto significaria que o controle das propriedades semânticas destas entidades estaria espalhado por todas as rotinas que as utilizassem, e não ficaria centralizado, controlado pelo SOO.

A necessidade de controlar centralizadamente informação, que vai desde dados crus até a representação de conhecimento humano, passando por objetos com comportamento próprio, é a característica fundamental do SOO do TABA. Além disso, ele deve estender a gerência para todos os dados, mesmos os não armazenados. Uma vez que uma informação é entregue a uma aplicação, ferramenta, processo, etc., o SOO não perde o controle sobre ela; ao contrário, toda a informação, mesmo as mais temporárias, podem ser responsabilidade do

SOO, que representa assim o papel de uma máquina virtual da estação de trabalho.

O modelo de dados de um SGBD convencional é a ferramenta conceitual que exprime como os dados são representados e operados pelo sistema. O compromisso com a eficiência faz com que nem todo tipo de informação possa ser facilmente representado em um modelo de dados convencional. Para tal, comumente é necessário usar-se outros modelos de dados, diferentes do empregado pelo SGBD, com maior capacidade de representar naturalmente outros tipos de informação. O mapeamento de uma representação à outra é muitas vezes manual.

Pelo que se pretende do SOO do TABA, o modelo para representação e operação da informação que ele manuseará não poderá ser um modelo de dados de um SGBD convencional. Nem tampouco parece apropriado o termo modelo de dados; sendo adotado o termo modelo de objeto. Este será de nível mais alto que os modelos de dados convencionais (hierárquico, redes, relacional, ...).

Entretanto, o requisito fundamental do modelo de objetos a ser adotado é que ele seja suportado diretamente pelo SOO. Portanto, uma implementação eficiente deste modelo deverá, não só, ser baseada em pesquisas a métodos de acesso, organização de dados, otimizações de acesso, etc., como também na propriedade (antônimo de impropriedade) do modelo ser eficientemente implementável.

Evidentemente, outros objetivos básicos (porém algumas vezes em contradição com o requisito anterior) não podem ser esquecidos:

- no máximo possível, as informações que habitarão o ambiente de desenvolvimento devem poder ser representadas e operadas pelo modelo;
- no máximo possível, esta representação e operação deve ser simples.

As aplicações que se servem de SGBDs convencionais, estão restritas à capacidade de representação dos modelos de dados destes SGBDs. Normalmente, estes são usados para a gerência de uma quantidade muito maior de dados do que de metadados (dados sobre os dados originais). Ou seja, há muito menos tipos de dados do que exemplares de dados destes tipos. Outra característica importante é a baixa frequência de alteração desses metadados (esquema, descrição dos tipos de dados).

Na atividade de desenvolvimento de software isto é invertido; novos tipos de dados estão sempre sendo criados, mas poucos tipos de dados terão muitos exemplares. Isto influiu no modelo de objetos do TABA.

O requisito fundamental do modelo adotado no GEOTABA é dele ser suportado diretamente pelo GEOTABA. Apesar disso, no máximo possível, as informações que habitam o ambiente de desenvolvimento devem poder ser representadas e operadas diretamente, de modo simples, pelo modelo.

As ferramentas, rotinas, funções e procedimentos dos ambientes do TABA também residirão no GEOTABA e por ele serão gerenciadas. O modelo do GEOTABA então precisa poder representar tanto dados como processamento sobre esses dados. Recursos para modelagem estrutural e operacional da informação podem ser encontrados tanto em modelos semânticos [BROD81] [BROD82] [BROD84B] [KING85] [LYNG84] como nos modelos baseados no paradigma de orientação a objetos [GOLD83] [TAKA88]. Assim, a definição de um modelo para um ambiente de desenvolvimento de software deve ser baseada nos conceitos encontrados nestas duas linhas de pesquisa.

4.3. Características Principais do Modelo

4.3.1. Objeto

No modelo de objetos MOO, toda a informação é representada por **objetos**, diferentemente de diversos modelos de dados orientados a objetos, onde o universo é particionado entre objetos e valores [BANC88] [SHYY91] (ver 3.2.1.9."Objetos e Valores" e 3.2.2.1.1."Modelos Híbridos"). Isto significa que inteiros, reais, caracteres e cadeias de caracteres também são objetos. Do mesmo modo, conjuntos, listas e outros tipos de coleções também são objetos. Com isso, o modelo ganha em uniformidade, o que aumenta a facilidade de uso, e em extensibilidade, pois novos tipos de coleções podem ser criados, assim como podem ser definidas especializações dos tipos de dados fundamentais.

Um objeto possui identidade, estado e comportamento. Isto significa que, não só um objeto possui um valor - seu estado -, mas também inclui propriedades dinâmicas, expressas no seu comportamento. Além disso, um objeto é identificável - pode ser distinguido de outros objetos mesmo que tenham valores (e comportamentos) iguais. Isto corresponde a noção intuitiva que se tem de objeto, algo que é possível ser distinguível. Assim, uma gota de água é um objeto, porém, uma gota de água no oceano não é um objeto.

Cada objeto é distinguido dos demais pela sua **identidade**, que é própria, única e imutável (ver 3.2.2.1.2.:"Identidade de Objetos"). Identidades não são objetos em si e portanto não são manipuladas pelos usuários. A identidade de um objeto é implícita e só o sistema de objetos tem acesso a ela. A

maneira de um usuário ter acesso a um objeto é através de um signo, objeto usado para identificar objetos globais do sistema e atributos.

Cada objeto tem a sua própria identidade, distinta dos demais objetos. Se w1 e w2 representam objetos com a mesma identidade, na verdade representam o mesmo objeto. Se eles têm identidades diferentes, os objetos representados são distintos. Isto é representado pelas relações ≡ (é o mesmo que) e ≢ (é distinto de).

Se w1 e w2 são objetos,

(1) $w1 \equiv w2 \Leftrightarrow$
$identidade(w1) = identidade(w2)$.

(2) $w1 \not\equiv w2 \Leftrightarrow$
$identidade(w1) \neq identidade(w2)$.

Objetos sempre possuem estado e comportamento. O **estado** de um objeto pode variar de acordo com as ações que ele sofre. O estado de um objeto representa o relacionamento que este mantém com outros objetos. Observe-se que valores numéricos, textos, datas, etc. também são objetos. Existem **objetos imutáveis**, que, independentemente de qualquer ação que sofram, nunca mudam de estado.

O **comportamento** de um objeto representa o modo com que este reage às ações que sofre. O conjunto de ações que o objeto pode sofrer é chamado de **protocolo do objeto**. A reação de um objeto a uma ação pode incluir uma mudança no seu estado e/ou provocar que este objeto exerça ações sobre outros objetos a ele relacionado (inclusive sobre ele próprio). Além disso, o objeto pode passar uma informação como resposta ao objeto que o acionou. Esta informação, como sempre, é um objeto, que pode ser usado, pelo objeto que exerceu a ação, para alterar o estado deste, criando um relacionamento entre os dois objetos.

Toda **ação** é realizada de um objeto sobre outro. Portanto não existem procedimentos que não façam parte de algum objeto (ver 3.2.1.6.:"Encapsulamento"). Uma ação ocorre em um instante de tempo e não é um objeto. O objeto que realiza a ação é denominado **emissor** e o que sofre é o **receptor**.Toda ação tem uma **assinatura**. Através dela, o receptor identifica que tipo de ação é e reage de modo apropriado. Se o comportamento do objeto não definir uma reação a ações com esta assinatura, estas não fazem parte do protocolo do objeto e, nesse caso, ocorre uma **falha**. Uma ação também possui **argumentos**, que são objetos usados como informação adicional ao receptor. Mais de um objeto pode ser atingido por um processo de comunicação entre objetos iniciado por uma ação. Isto se deve ao fato de que um objeto, ao reagir a

uma ação, pode estabelecer novas comunicações realizando outras ações sobre outros objetos.

Uma ação pode ser síncrona ou assíncrona. Uma ação síncrona é denominada **mensagem** e suspende a atividade do emissor. A reação do receptor a uma mensagem inicia imediatamente após o recebimento desta e termina com uma resposta ao emissor. Esta resposta retoma imediatamente a atividade do emissor e passa, para este, um objeto resultado da ação. Em tempo de execução, a mesma mensagem pode, obviamente, resultar em objetos diferentes. A resposta à mensagem pode ser usada para transmitir, ao objeto que iniciou a comunicação, informações sobre o estado do objeto cujo comportamento foi ativado e/ou um indicativo do sucesso (ou insucesso) da operação.

Uma ação assíncrona recebe o nome de **evento**. O emissor de um evento não é suspenso pelo envio deste. A reação ao evento pode não se dar imediatamente. Em um dado instante, um objeto pode ter diversos eventos recebidos aguardando a sua vez. Ao contrário das mensagens, os eventos não são mecanismos de comunicação bidirecionais: o emissor de um evento não recebe resposta do receptor no próprio evento. A reação a uma ação é implementada por um método. Um método implementado para reagir a um evento é chamado de **tratador de eventos.**

Os exemplos contidos em caixas de bordas duplas, como este, são construções na **Linguagem de Definição e Manipulação de Objetos - DEMO**. Essa linguagem implementa o modelo de objetos do GEOTABA.

A expressão (1)

```
34 + 43
```

define uma ação onde uma mensagem de soma é enviada ao objeto representado pelo literal *34*. Portanto, este é o receptor da ação. O objeto representado pelo literal *43* é o argumento da mensagem. O objeto emissor, que mandou executar esta expressão, é suspenso até que a mensagem seja respondida. O resultado da resposta, que será o inteiro *77*, poderá então ser usado em novas ações ou ser armazenado em alguma variável, como em (2)

```
total := 34 + 43
```

Um **método** especifica a reação que o objeto deve ter quando sofrer uma determinada ação. O comportamento de um objeto é definido por um conjunto de métodos. Um método é composto por uma assinatura, nomes de parâmetros e de variáveis e um corpo. A assinatura identifica unicamente o método dentro

do objeto, ou seja, dois métodos de um mesmo objeto não têm a mesma assinatura. Um método pode ser alterado, porém a sua assinatura não pode, pois, se o for, estará especificando outro método. Além disso, é tendo a mesma assinatura que a ação exercida, que também possui uma assinatura, se casa com o método que implementa a reação correspondente, provocando a execução do método. Caso a ação não tenha um método correspondente, ou seja, com a mesma assinatura, ocorrerá uma falha (exceção). Os parâmetros e variáveis têm finalidade semelhante aos parâmetros e variáveis locais em procedimentos em linguagens de programação tradicionais, como Pascal. Um método não pode ter parâmetros e/ou variáveis locais identificadas pelo mesmo signo. Quando a assinatura da mensagem casa com a assinatura do método, as variáveis parâmetros do método passam a ter como valor os argumentos da mensagem.

O código (3)

```
Tartaruga sentença recomeça {apaga desenho; centraliza}
```

define o método que objetos do tipo *Tartaruga* executarão quando receberem a mensagem *recomeça.* O nome *recomeça* é a assinatura deste método, que não possui parâmetros ou variáveis. O bloco (conjunto de instruções entre chaves), forma o corpo deste método.

O corpo de um método consiste na especificação de uma sequência de comandos a serem realizados. Um **comando** pode ser uma ação que o objeto realizará sobre outro objeto. Existem outras alternativas de comando, restritas a determinados níveis de abstração do objeto. A **especificação de uma ação** é composta pela assinatura da ação, pela especificação do objeto receptor da ação e pela especificação dos argumentos da ação. Tanto os argumentos, como o receptor, seguem as mesmas regras de especificação. Eles podem ser especificados através da referência a (1) um nome de parâmetro ou variável do método, (2) através da referência ao próprio receptor do método ou (3) pela especificação de uma mensagem. Neste último caso, o objeto retornado como resposta da mensagem é que será usado como receptor ou atributo da ação. Existem também outras alternativas de especificação de atributos e receptor das ações, alternativas essas restritas a determinados níveis de abstração do objeto.

O corpo do método do exemplo anterior especifica dois comandos, separados por um ponto-e-vírgula. Os dois comandos são mensagens, sem argumentos, cujas assinaturas são *apaga desenho* e *centraliza*. O receptor dessas mensagens não foi especificado explicitamente e, quando o receptor de uma mensagem é omitido, o **objeto corrente** do bloco será considerado como sendo o receptor da mensagem. O objeto corrente do bloco que constitui o corpo de um método é sempre o receptor deste método. Portanto, essas mensagens serão enviadas ao mesmo objeto receptor de *recomeça*. O literal *isto* também poderia ser usado para representar o receptor do método. Assim, o corpo do método poderia ser reescrito como: (4)

```
{isto apaga desenho; isto centraliza}
```

explicitando os receptores das mensagens.

Ações podem ser executadas na execução de um método ou diretamente pelo usuário. Isto significa que o usuário final pode escrever mensagens do mesmo modo que um programador escreve mensagens em um método. Na verdade, as mensagens que o usuário final envia funcionam como se ativassem métodos de um objeto **ambiente**. Um ambiente é um objeto específico onde o usuário pode executar mensagens.

Os conceitos definidos nesta subseção são válidos em todo o modelo de objetos. Porém este oferece diferentes níveis de abstração para a abordagem destes conceitos. Este é o assunto da subseção seguinte. Linguagens de programação orientadas a objetos e sistemas de bancos de dados orientados a objetos apresentam conceitos semelhantes aos conceitos aqui mencionados, porém circunscritos à visão restrita da informação fornecida por esses sistemas (ver 3.2.1.:"Programação com Objetos"). Acrescente-se a isso que, de um modo geral, o conceito de eventos não faz parte do modelo desses sistemas. Ele pode ser encontrado em alguns sistemas e linguagens para programas com concorrência (ver 3.2.3.3.8.9.:"Linguagens Orientadas a Eventos") e, onde são especialmente importantes, em sistemas de interface com usuário, como o Toolbox do Macintosh [APPL85].

4.3.2. Níveis de Abstração do Modelo

Devido aos requisitos discutidos em 3.4.:"Características de Modelos de Objetos para Sistemas de Objetos", o modelo deve poder representar desde estruturas internas da implementação dos objetos até representações de interface com usuário destes. O objetivo do modelo de objetos do GEOTABA é fornecer

coerência na relação entre estas diferentes formas que os objetos assumem no sistema. Um mesmo objeto tratado pelo sistema, seja ele uma simulação de uma entidade do mundo real ou concebido por algum usuário, apresenta diversos modos de ser encarado: como uma estrutura de dados na memória acoplada a procedimentos - visão adequada ao implementador destas estruturas e procedimentos -, como dados armazenados em disco ou como figuras manipuláveis na tela pelo usuário. Por exemplo, um automóvel pode ser representado por diferentes tipos de estruturas de dados na memória, por outras em disco e ainda sob diversas representações gráficas nos meios de interação com usuário. Todavia estas diferentes abstrações se referem a algum objeto automóvel conceitual. Cabe ao modelo de objetos relacionar coerentemente o objeto conceitual a essas diferentes abstrações. A integração, através da orientação a objetos, destas diferentes abordagens de representação da informação diferencia o modelo de objetos do GEOTABA dos modelos de representação da informação dos SGBDs, das linguagens de programação e dos sistemas de interface com usuário.

O modelo então define quatro níveis de abstrações: (1) implementação; (2) conceitual; (3) externo e (4) representação. Cada objeto é encarado de modo diferente em cada um destes níveis. O nível de **implementação** descreve estruturas de dados e métodos dos objetos conforme implementados. O nível **conceitual** trata da estrutura e comportamento dos objetos conforme percebidos pela comunidade de usuários e pelos outros objetos. O nível **externo** particulariza visões específicas para grupos de usuários criando objetos virtuais. O nível de **representação** se refere a como os objetos são apresentados e manipulados, nos meios de interface com usuário.

Nas seções seguintes, cada um desses níveis de abstração é detalhado, iniciando-se, na seção 4.4., pelo nível conceitual, básico para a modelagem da informação com independência de dados e de representação para o usuário. O nível de implementação é apresentado na seção 4.5. e o nível externo na 4.6.. Por último, na seção 4.7., é discutido o nível de representação.

4.4. O Nível Conceitual

Em aplicações complexas, as principais decisões envolvem definir a estrutura de dados para suportar uma abstração ou dados algoritmos. Como essas decisões podem ser alteradas, tipos abstratos de dados são empregados para esconder as estruturas de dados. Os objetos, conforme definidos pelas linguagens de programação orientadas a objetos puras, encapsulam a sua estrutura através do seu comportamento. Isto significa que os atributos, componentes e relacionamentos de um objeto só são visíveis caso haja métodos que os dêem

como resposta. Assim, as linguagens de programação orientadas a objetos puras permitem a construção de módulos de programa com alta coesão interna e baixo acoplamento entre eles, propiciando maior alterabilidade e visibilidade ao software produzido. O resultado é uma facilidade maior de manutenção e evolução dos sistemas.

Em aplicações convencionais que trabalham com dados do mundo real, como é o caso das aplicações tradicionais de bancos de dados, a estrutura desses dados é bem definida, pouco se altera e, principalmente, é naturalmente percebida por todos que usam esses dados. Nesses casos a ocultação da estrutura da informação pode vir a atrapalhar mais do que a ajudar.

De fato, é comum nas linguagens orientadas a objetos, quando se define uma classe, se definir uma série de métodos que simplesmente consultam ou alteram o valor de uma variável de instância. Isto ocorre porque na verdade esta variável de instância está representando um atributo da estrutura do objeto que é inerentemente visível externamente.

Considerando isso, algumas linguagens orientadas a objetos enfraquecem, ou até eliminam, o mecanismo de encapsulamento, permitindo que algumas variáveis de instância sejam "públicas", isto é, visíveis externamente. Isto não costuma apresentar boas soluções porque confunde a implementação física de um objeto com a sua visão conceitual.

O modelo de objetos do GEOTABA propõe a separação entre os níveis conceitual e implementacional dos objetos, permitindo a definição do mapeamento entre os dois níveis. Esta solução diverge da descrita no parágrafo anterior por impor que tudo o que seja percebido externamente à implementação dos objetos esteja no nível conceitual destes. Com isso, ganha-se a modularização da implementação de programas, característica das linguagens de programação orientadas a objetos, sem sacrifício da naturalidade da descrição semântica da informação e da independência de dados, metas dos sistemas de bancos de dados. Em um modelo puramente orientado a objetos, apenas o protocolo dos objetos corresponderia ao nível conceitual. A constatação importante a ser feita é que, quase sempre, parte deste protocolo é definida para dar acesso a estrutura percebida dos objetos.

O nível conceitual de um objeto diz respeito a como este é percebido pelos demais objetos, ou seja, corresponde ao **protocolo** do objeto. Parte deste protocolo representa a **estrutura percebida** do objeto; o restante é formado pelos **serviços** fornecidos pelo objeto. O nível conceitual também define **restrições** que devem ser obedecidas pela estrutura e serviços do objeto.

4.4.1. Estrutura Percebida ou Conceitual

A estrutura percebida de um objeto compõe-se de **associações**. Cada associação relaciona o objeto em questão a outro, denominado **valor** da associação. Uma associação pode ter, ou não, uma **chave**, que a identifica dentro do objeto. No caso mais comum, todas as associações de um objeto possuem chaves. Duas associações de um mesmo objeto não podem possuir a mesma chave, nem a chave pode ser vazia (objeto nulo). Além disso, uma associação não pode ter a sua chave mudada.

A função da chave é permitir o manuseio da associação. Para cada **associação com chave**, o objeto possui a capacidade de reagir a uma **mensagem de consulta** à associação. A chave é passada como um atributo da mensagem. A resposta a essa mensagem é um objeto que se percebe como associado ao receptor, ou seja, o valor da associação. Observe-se que, internamente, aquele objeto pode estar sendo referido por um campo da estrutura de dados implementada do receptor. Também é possível que não esteja, e, nesse caso, a implementação da consulta implique em cálculos ou consultas a outros objetos.

A expressão (5)

```
Departamento ["Pessoal"]
```

representa uma mensagem de consulta ao objeto designado pelo nome global *Departamento.* A mensagem responderá o valor da associação deste objeto cuja chave é a cadeia de caracteres *"Pessoal"*.

A expressão (6)

```
ramal de Departamento["Pessoal"]
```

consulta a associação, cuja chave é o signo *ramal,* do objeto resultante da expressão anterior. As duas formas seguintes, (7)

```
Departamento["Pessoal"] ramal

Departamento["Pessoal"][$ramal],
```

são equivalentes a anterior. Na última, o *$* de *$ramal* indica lexicamente que este é um signo; este signo é que é usado como chave da associação desejada. Quando a chave é um signo conhecido em tempo de compilação, as formas prefixa e posfixa anteriores podem ser empregadas. Na prefixa é necessária uma preposição, como *de, do, da, of* ou *!*, para fazer a ligação.

Uma associação com chave pode ser **variável** ou **constante**. Apenas associações com chave têm essa classificação. Uma associação constante possui sempre o mesmo valor desde o momento em que foi criada. Atente-se para o

fato que, mesmo que a associação seja constante e, portanto, se refira sempre ao mesmo objeto, este pode não ser um **objeto imutável**. Todas as associações de um objeto imutável são constantes e se referem a outros objetos imutáveis. Associações variáveis são o tipo mais comum e geral de associações. A cada uma delas, o objeto possui a capacidade de reagir a uma **mensagem de atribuição**. Esta mensagem, além da chave como argumento, sempre possui um outro argumento que leva para o receptor um objeto a ser associado a ele. Novamente pode-se notar que, no nível de implementação do objeto, este atributo pode ser armazenado em um campo da estrutura de dados interna do receptor ou receber um tratamento mais complexo.

A expressão (9)

```
Departamento["Pessoal"][$ramal] := "226"
```

é uma mensagem de atribuição ao objeto respondido pela expressão *Departamento["Pessoal"]*. A chave aparece entre colchetes e o objeto a ser atribuído é dado pela expressão que segue o *:=*. Após a execução dessa mensagem, a associação em *Departamento["Pessoal"]*, com chave *$ramal*, passará a ter como valor o objeto cadeia de caracteres *"226"*. As formas prefixa (10)

```
ramal de Departamento["Pessoal"] := "226"
```

e posfixa (11)

```
Departamento["Pessoal"] ramal := "226"
```

também podem ser usadas equivalentemente. O receptor destas mensagens têm que permitir atribuição ao atributo *ramal*. Observe-se que, por exemplo, nestas duas últimas formas a assinatura da mensagem de atribuição é algo semelhante a *ramal:=* e não o *:=* isolado. Este, isoladamente, nunca representa uma mensagem.

Qualquer objeto pode servir como chave. Comumente a chave é um **signo** e, nesse caso, a associação é chamada de **atributo** do objeto. Um signo é um objeto com características textuais cuja finalidade é permitir que o ser humano identifique a associação. Não há necessidade de se diferenciar atributos derivados dos não derivados. Como os atributos estão no nível conceitual, não interessa como eles são implementados, se a partir de um campo que mapeia diretamente o atributo na sua estrutura de dados interna, ou se através de cálculos.

Nos exemplos anteriores, a associação de *Departamento,* cuja chave é *"Pessoal"*, não é um atributo, pois sua chave é uma cadeia de caracteres e não um signo. Já *ramal* é um signo de atributo de *Departamento["Pessoal"]*.

Um objeto pode ter um número variável ou constante de associações. Um objeto com número variável de associações reage a ações de **acréscimo** do objeto criando novas associações nele. Acréscimos a objetos com número constante de associações falham. Um objeto com número variável de associações ou que contém associações que não são atributos é chamado de **coleção.**

Caso o objeto valor do atributo *ramal de Departamento["Pessoal"]* seja uma coleção, um novo ramal pode ser acrescentado a ela. (12)

```
ramal de Departamento["Pessoal"] ⊕= "226"
```

As outras formas de designação do atributo continuam válidas. O símbolo ⊕ =, ao contrário do :=, é um operador, isto é, representa uma mensagem enviada ao objeto resultante da avaliação da expressão *ramal de Departamento["Pessoal"]* .

4.4.2. Serviços

Serviços correspondem às ações que primordialmente produzem processamento e não representam a estrutura percebida do objeto. São usados para descrever cada ação possível de ser executada pelo objeto. Os serviços são implementados por métodos, conforme descrito em 4.3.1.:"Objeto". Os serviços podem ser **dependentes** ou **independentes da implementação** do objeto. Um serviço dependente faz acesso à estrutura interna do objeto e é implementado por método especificado no nível de implementação do objeto. Já um serviço independente é implementado por método também independente da implementação do objeto e nunca acessa diretamente os campos da estrutura interna do objeto. Estes métodos são descritos totalmente no próprio nível conceitual do objeto. Outra classificação para serviços os divide em **síncronos** e **assíncronos,** dependendo de se o serviço serve como reação a uma mensagem ou a um evento.

O método *recomeça* implementa um serviço dos objetos de tipo *Tartaruga*. Este serviço independe da implementação deste tipo, baseando-se apenas em outros serviços de objetos deste tipo.

4.4.3. Restrições e Modelagem Conceitual

Uma restrição é um predicado que deve sempre resultar em um valor verdadeiro para que o estado do sistema seja considerado íntegro. Tanto propriedades estruturais quanto comportamentais podem estar sujeitas a restrições.

As restrições são propriedades de cada objeto. Uma parte da verificação do atendimento as restrições é estática, isto é, realizada durante a especificação de estruturas e de ações, liberando a execução das ações destas verificações. O nível conceitual do modelo de objetos do GEOTABA permite a especificação de restrições através do mecanismo de **tipos de objetos** e de **padrões**. As restrições estáticas são verificadas durante a definição dos tipos e padrões. Já as restrições dinâmicas, quando definidas em um padrão ou tipo, migram destes para os objetos, de modo a poderem controlar o cumprimento da restrição durante as transformações de estado do objeto.

Padrões, tipos, ligações e outros mecanismos de representação de restrições formam o ferramental para a modelagem conceitual da informação no modelo de objetos do GEOTABA. Uma notação gráfica e outra textual são utilizadas para exemplificar como estes mecanismos são empregados na modelagem. A notação gráfica, denominada **Diagrama de Objetos - DOO**, foi desenvolvida para ser utilizada na modelagem conceitual da informação no GEOTABA. A subseção 4.4.11.:"Diagramas de Objetos" é dedicada a descrição desta notação. A notação textual empregada nos exemplos é a **DEMO**, descrita em 4.4.12.:"**Linguagem de Descrição e Manipulação de Objetos**" .

4.4.3.1. Padrões

É comum em sistemas de bancos de dados e em linguagens de programação tradicionais a definição de domínios de atributos e variáveis. A especificação de um domínio sobre o qual um atributo ou uma variável pode ter o seu valor é uma restrição, pois limita as possibilidades de valores possíveis destes. Este mecanismo é comumente designado como tipificação ou tipagem (ver 3.2.2.1.6.:"Tipagem"). Aqui será adotado o termo **padronização**, reservando o termo tipo para uma categoria específica de padrões.

Um **padrão** representa um conjunto de restrições. Quando um objeto atende a todas as restrições especificadas em um padrão, o objeto é dito que

está em conformidade com o padrão. Em um dado instante, um objeto pode estar em conformidade com diversos padrões. Padrões do GEOTABA são semelhantes aos tipos abstratos de Emerald [BLAC86] e a subtipos de Ada [LEDG81].

Padrões também são objetos. É apropriado notar que não é esperado que um padrão saiba responder quais objetos estão em conformidade consigo. Isto significa que um padrão não funciona como um recipiente ou coleção dos objetos em conformidade com ele (ver 3.2.2.1.5.:"Classes e Coleções").

Um padrão pode possuir um nome ou ser um padrão anônimo. O nome de um padrão pode ser usado como uma abreviatura para não se ter que repetir as restrições que formam o padrão.

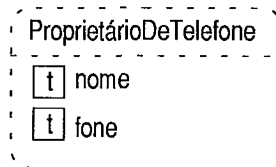


Figura 12: Especificação de um Padrão com Nome

Figura 13: Especificação de um Padrão sem Nome

A Figura 12 exemplifica como se define, em um Diagrama de Objetos (DOO), um padrão *ProprietárioDeTelefone*. Na Figura 13, o atributo *número da coluna* está restrito a um padrão sem nome que define que seu valor deve ser um número inteiro pertencente ao intervalo [1 6].

O trecho de código DEMO abaixo (13) declara uma variável *pt*, restrita ao padrão *ProprietárioDeTelefone*, e atribui um objeto a ela.

```
variável pt: ProprietárioDeTelefone
pt:= ‹      nome:t ⇒ "Fred"
            idade:i ⇒ 27; fone:t ⇒ "555-1830"›
```

Em DEMO, ao contrário das linguagens de programação não conversacionais, como Pascal, o final de linha é significativo, podendo ser substituído equivalentemente por um ponto-e-vírgula. Para que o final-de-linha não seja considerado, ele deve ser precedido pelo caracter escape (\). Portanto, uma linha que termine com este caracter é considerada como emendada à linha seguinte. Na verdade, qualquer elemento léxico precedido imediatamente pelo caracter escape é desconsiderado.

A primeira instrução declara a variável *pt*. Uma variável declarada desse modo é uma **variável da sessão**, que terá seu tempo de vida restrito à duração da sessão do usuário ou até que este remova a variável, ou todas as variáveis da sessão, explicitamente.

A segunda instrução constrói uma tupla (ver 4.4.5.12.:"Tupla"), que é um objeto que permite a adição dinâmica de atributos. Os parênteses quebrados ‹ › funcionam como construtores de tuplas e são caracteres distintos dos operadores relacionais < e >. Cada instrução contida nele acrescenta um novo atributo à tupla. O objeto atribuído à *pt* possuirá três atributos: *nome, idade* e *fone*. Os atributos *nome* e *fone* são restritos ao padrão de nome *t*, e *idade* é restrito ao padrão *i*. O valor de cada atributo da tupla é fornecido após o caracter ⇒ . Os valores fornecidos, duas cadeias de caracteres e um número inteiro, são válidos por estarem em conformidade com os padrões especificados para os atributos, respectivamente: *t* (texto) e *i* (inteiro).

O objeto construído pode ser atribuído à variável *pt* por estar em conformidade com o padrão definido para *pt*: *ProprietárioDeTelefone*. Esta conformidade ocorre porque a tupla possui os dois atributos exigidos no padrão, com as mesmas restrições. Se o usuário enviar a seguir os comandos (14)

```
impressão de nome de pt
impressão de fone de pt
```

serão impressos *Fred* e *555-1830*. Todavia o comando (15)

```
impressão de idade de pt
```

acusará erro de compilação, pois, apesar do objeto contido em *pt* possuir o atributo *idade*, o padrão de *pt* não o possui. Garantindo-se que variáveis, atributos e expressões não serão usadas fora do que prevê seus padrões, erros, como ocorreria se *pt* contivesse um objeto sem o atributo *idade*, podem ser descobertos estaticamente.

4.4.3.1.1. Especialização e Generalização de Padrões

Existe a possibilidade de se criar padrões que são especializações ou generalizações de outros padrões. Por exemplo, pode-se criar dinamicamente um padrão S, com um conjunto de restrições, tal que todo objeto em conformidade com ele esteja também em conformidade com um outro padrão P, ou seja, também atenda as restrições definidas em P. O contrário também é possível: pode-se criar dinamicamente um padrão P, tal que todo objeto em conformidade com um outro padrão S esteja também em conformidade com P. Em ambos os casos, diz-se que S é uma **especialização** de P e que P é uma **generalização** de S. Observe-se que podem existir objetos em conformidade com um padrão P que não estejam em conformidade com qualquer especialização dele. Objetos em conformidade com o padrão mais especializado podem ser usados onde quer que sejam esperados objetos do tipo mais geral.

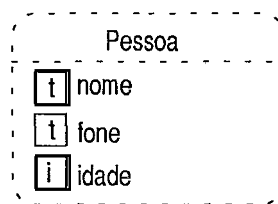


Figura 14: Especificação de uma Especialização do Padrão *ProprietárioDeTelefone*

A Figura 12 especifica que qualquer objeto que possua atributos *nome* e *fone*, ambos restritos a textos (cadeias de caracteres), está em conformidade com o padrão *ProprietárioDeTelefone.* A Figura 14 define o padrão *Pessoa*, que exige, além dos mesmos atributos de *ProprietárioDeTelefone*, que seus objetos possuam também o atributo *idade*, restrito a inteiros, o que está indicado pela letra *i* dentro da caixa. Como todo objeto em conformidade com *Pessoa* estará em conformidade também com *ProprietárioDeTelefone*, segue-se que o padrão *Pessoa* é uma especialização de *ProprietárioDeTelefone* e que este é uma generalização daquele. Observe-se que, neste exemplo, não há qualquer vinculação explícita entre os dois padrões.

No seguinte trecho DEMO(16)

```
variável prop: ProprietarioDeTelefone
variável p: pessoa

p := <       nome ⇒ "José";          idade ⇒ 72
             fone ⇒ "555-1234" >
prop := p    \OK
p := pt      \ERRO
```

são declaradas duas variáveis *prop* e *p*, restritas aos padrões *ProprietárioDeTelefone* e *Pessoa*, respectivamente. Pode-se observar que é indiferente a colocação ou não de acentos e o uso de maiúsculas ou minúsculas. Os padrões dos atributos da tupla criada na terceira instrução não foram fornecidos na construção das associações; cada um deles será induzido a partir do padrão do valor inicial da sua associação. Portanto, *idade* será do tipo *Inteiro*, equivalente ao padrão predefinido *i*; *nome* e *fone* serão do tipo *Texto*, e poderão ser atribuídos a atributos de padrão *t*. O padrão tipo *t* contém cadeias de caracteres imutáveis e o seu subtipo *Texto* acrescenta protocolo que permite a alteração dos caracteres nas cadeias. A tupla, então, poderá ser atribuída a *p* por estar em conformidade com *Pessoa*.

Na instrução seguinte, esta tupla é atribuída a *prop*, o que é permitido pois *p*, que tem conformidade com *Pessoa*, consequentemente tem conformidade também com *ProprietárioDeTelefone*. Pela variável *prop*, não se terá acesso ao atributo *idade* desta tupla, pois ela estará sendo vista aí como um proprietário de telefone. Já por *p* se terá acesso a todos os atributos da tupla que uma pessoa possui. A última instrução não é válida, pois *pt*, que é um proprietário de telefone, não garante que seu objeto possua o atributo *idade*, exigido pelo padrão *Pessoa* de *p*. As palavras *OK e ERRO* funcionam aí como comentário, por estarem precedidas por \.

Este conceito de conformidade é inspirado no encontrado na linguagem Emerald [BLAC86] e no modelo de dados Melampus [RICH91]. A diferença com relação ao conceito de subtipos convencionais, como os das linguagens C++ [STRO86] e de SGBDs O_2 [LECL88] e Orion [BANE87], é que a especialização/generalização pela conformidade é implícita, enquanto, nestes sistemas, o relacionamento entre tipos e seus subtipos é explicitamente declarado. A conformidade permite que se crie novos tipos (padrões no GEOTABA), generalizando ou especializando outros, dinamicamente, sem modificar os já existentes.

Todavia, no modelo de objetos do GEOTABA, a conformidade a padrões representa a obediência a um conjunto de restrições. Em Emerald, a conformidade se refere a tipos, que representam, nesta linguagem, uma interface, conceito correspondente ao de protocolo no GEOTABA. Mais adiante é mostrado que restrição de protocolo é um dos tipos de restrições possíveis de se ter em um padrão e ficará claro que o conceito de conformidade no GEOTABA é mais amplo que o de Emerald, pois engloba-o. Além disso, o modelo de objetos do GEOTABA acrescenta aos padrões o conceito de tipo, superando fragilidades do modelo de conformidade original.

A definição de um padrão generalizando ou especializando outros pode ser feita sem criar qualquer vínculo entre os padrões antigos e o recém criado. O simples fato das restrições de um padrão conterem as restrições do outro é que produz a generalização ou a especialização. Observe-se que, nesta situação, não se pode falar em herança de propriedades de um padrão para o outro, pois eles são totalmente independentes. No momento da criação de um padrão, pode-se até mencionar outros padrões a serem usados como base desta definição, porém, logo após a definição, não restará qualquer vínculo entre o novo padrão e o antigo.

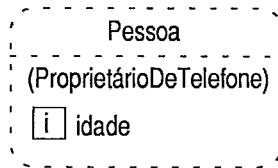


Figura 15: Especialização sem Vínculo

A Figura 15 apresenta uma maneira alternativa de se definir o mesmo padrão que o da Figura 14. Os parênteses em torno do padrão *ProprietárioDeTelefone* indicam que não há vínculo duradouro entre os dois padrões. Apenas no momento da criação do padrão *Pessoa,* as restrições contidas no padrão *ProprietárioDeTelefone* foram repetidas.

Todavia, frequentemente, na definição de uma especialização de padrão deseja-se criar um vínculo com os padrões sendo especializados. Um padrão pode ser vinculado a outros padrões. Neste caso, diz-se que aquele é **subpadrão** destes e estes **superpadrões** daquele. Se S é um subpadrão de P1, P2, ... Pn, então ele é uma especialização de seus superpadrões P1, P2, ... e Pn e, portanto, qualquer objeto em conformidade com S também está em conformidade com P1, P2, ... e Pn. Note-se que podem existir objetos em conformidade simultaneamente com P1, P2, ... e Pn e que não estejam em conformidade com S. O vínculo existente entre um padrão e seus subpadrões implica em que a realização de uma alteração na definição do padrão signifique que as definições dos subpadrões também devem ser consideradas como estando sendo alteradas.

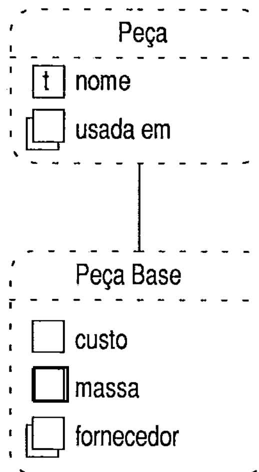


Figura 16: Hierarquização de Padrões

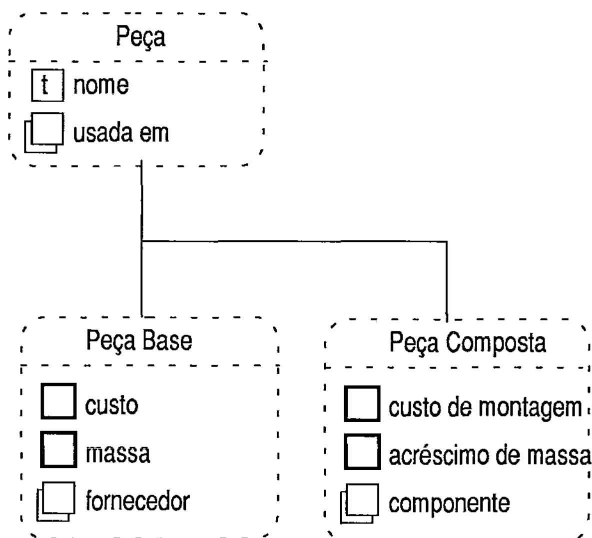


Figura 17: Hierarquização de três Padrões

A Figura 16 define que *Peça Base* é um subpadrão de *Peça*. Para isto bastou-se ligar a parte superior do diagrama de *Peça Base* com a parte inferior de *Peça*. A Figura 17 estende o diagrama acrescentando um novo subpadrão a *Peça*: *Peça Composta*. Se o padrão *Peça* for alterado, esta alteração se refletirá nos seus subpadrões. Pelos diagramas destas figuras, nada impede que haja objetos em conformidade com *Peça* que não estejam em conformidade nem com *Peça Base* nem com *Peça Composta*, ou que estejam em conformidade com os três padrões

Há várias espécies de restrições que podem ser incluídas em um padrão. Uma delas se refere ao protocolo dos objetos em conformidade com o padrão. Uma **restrição de protocolo** define um conjunto de ações - o **protocolo do padrão** - aplicáveis a qualquer objeto em conformidade com o padrão. Apenas os objetos que atendem a todas as ações definidas no protocolo do padrão poderão estar em conformidade com o padrão. Ou seja, toda ação definida no padrão pode ser aplicada a qualquer objeto em conformidade com ele. O protocolo do padrão define a estrutura percebida e os serviços dos objetos quando vistos conforme o padrão. As restrições de protocolo podem ser verificadas estaticamente.

O padrão *ProprietárioDeTelefone*, especificado na Figura 12, se caracteriza por uma restrição de protocolo, que define que um objeto, para estar em conformidade com este padrão, tem que, pelo menos, possuir um atributo textual *nome* e outro atributo, também textual, *fone*. Já o padrão sem nome especificado na Figura 13 possui outros tipos de restrições, mas não inclui restrição de protocolo. A listagem de atributos ou serviços dentro da caixa de definição de um padrão define uma restrição de protocolo. O padrão *Pessoa*, conforme especificado na Figura 14 ou na Figura 15, também são baseados em restrição de protocolo. Cada um dos subpadrões *Peça Base* e *Peça Composta*, definidos nas Figuras 16 e 17, acrescenta uma restrição de protocolo própria, além da restrição de protocolo que eles herdam da definição do padrão *Peça*.

Uma ação definida no protocolo de um padrão possui restrições adicionais chamadas de restrições de padrão de argumento, ou, simplesmente, uma **restrição de padrão**. Uma restrição de padrão em um argumento de uma ação define que aquela ação só será permitida se este seu argumento estiver em conformidade com o padrão designado na restrição. Restrições de padrão podem ser verificadas estaticamente. Como a existência de um determinado atributo no protocolo do padrão significa que os objetos em conformidade com ele aceitam ações de atribuição para este atributo, a restrição de padrão na atribuição especifica que apenas objetos em conformidade com o padrão designado podem ser atribuídos ao atributo. O padrão designado na restrição, neste caso, é chamado de **padrão do atributo**.

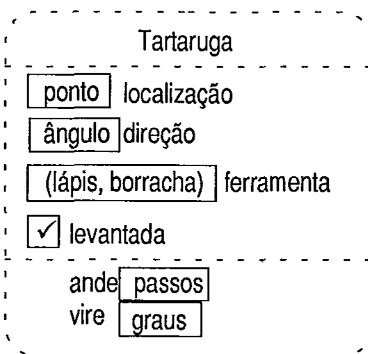


Figura 18: Padrão com Serviços

A Figura 18 mostra a definição de um padrão cujo protocolo inclui, além dos atributos *localização, direção, ferramenta* e *levantada*, os serviços *ande(passos)* e *vire(graus)*. Ambos serviços especificam que as ações para realizá-los exigem um argumento. No serviço *ande(passos)*, o argumento está restrito a um padrão cujo nome é *Passos*; no *vire(graus)*, o argumento é restrito a *Graus*. Portanto, cada um desses argumentos possui uma restrição de padrão. Por sua vez, os atributos definidos também estão restritos a padrões especificados. O padrão do atributo *localização* é *Ponto*. Isto significa que, o objeto obtido como resposta a uma consulta sobre a localização de uma tartaruga sempre estará em conformidade com o padrão *Ponto* e, portanto, poderá sofrer qualquer ação prevista no protocolo de *Ponto*. Além disso, apenas objetos em conformidade com *Ponto* podem ser atribuídos a *localização*. Ou seja, o argumento da atribuição à *localização* de uma tartaruga possui uma restrição de padrão. O padrão do atributo *direção* é *Ângulo* e do atributo *levantada* é *Condição*, especificado através do símbolo ☑. O atributo *ferramenta* é restrito aos objetos designados pelos signos *lápis* e *borracha*.

Como os objetos no sistema são acessados através de atributos e outros tipos de associações, além de ações, todos sujeitas a restrições de padrão, padrões fornecem maneiras de se perceber, no nível conceitual, um objeto implementado. Enquanto um objeto estiver sendo percebido através de um dado padrão, ele não pode se desconformar com este padrão. Por este motivo, no caso do padrão definir alguma restrição dinâmica, no momento em que a atribuição é realizada, além da restrição ser verificada, ela é "copiada" para o objeto, a não ser que ele já a possua. Desse modo, as suas mudanças de estado, independentemente do padrão empregado para realizá-las, garantirão o respeito à restrição. As restrições de padrão são fundamentais na modelagem conceitual. Elas que justificam a existência do conceito de padrões. Elas também são importante para que, para um mesmo objeto receptor, haja ações polimórficas: ações com mesmo "nome", com argumentos diferentes, causem a execução de métodos diferentes, embora com o mesmo "nome". Isto é possível definindo-se que, tanto as assinaturas das ações, quanto as assinaturas dos métodos, possuem informação sobre o padrão de seus argumentos.

vire [graus]
vire [radianos]

Figura 19: Serviços Polimórficos

Se o padrão *Tartaruga* da Figura 18 definisse os serviços *vire(graus)* e *vire(radianos)*, conforme especificado na Figura 19, uma tartaruga poderia receber mensagens *vire* com argumento, tanto em graus, como em radianos. De acordo com o argumento estar conforme *Graus* ou *Radianos*, o método a ser executado será diferente.

Outro tipo de restrição, especialmente importante, possível de ser empregada em padrões, se refere aos tipos de objetos aceitos por aquele padrão. Alguns padrões que possuem restrição de tipo são chamados de padrões tipos ou, simplesmente, de **tipos**. Para que estes possam ser apresentados, mais adiante, em 4.4.3.3.1.:"Padrões Tipos", é apresentado agora o conceito de tipo de objeto.

4.4.3.2. Tipo de Objeto

Um **tipo** define um conjunto de objetos, denominado **domínio do tipo**. Quando um objeto faz parte do domínio de um tipo T, o objeto é dito que **é do** tipo T. Tipos diferentes podem incluir um mesmo objeto, ou seja, em um dado instante, um objeto pode ser de diversos tipos. Todo objeto sabe responder a mensagens que questionam se ele é de um dado tipo, passado como atributo da mensagem. Ou seja, desde o momento em que um objeto é criado, ele possui, internamente, informação que o torna capaz de responder a essas consultas sobre tipos já existentes, ou que venham a ser criado. Um objeto de um tipo não pode dinamicamente deixar de sê-lo, a não ser sob condições muito especiais.

Tipos também são objetos. É apropriado notar que não é comum que um tipo saiba responder quais objetos formam o seu domínio.

É importante distinguir o conceito de tipo do de coleção. Embora um tipo represente um conjunto de objetos (domínio) em conformidade com o tipo, um tipo não sabe responder a ações referentes a quais objetos pertencem ao seu domínio. Um objeto é quem sabe se é de um tipo ou não. Em outras palavras, um tipo não conhece seus objetos. Novos objetos são criados dinamicamente e implicitamente associados aos tipos existentes.

O relacionamento entre coleções e seus objetos **membros**, ou seja, participantes de alguma associação sua, é o inverso do anterior. Uma coleção conhece seus membros e responde consultas referentes a eles. Todavia, um objeto normalmente não sabe informar de que coleções ele é membro (o que só pode ser feito perguntando às coleções). Mais adiante será observado que pode-se definir tipos de coleções e que uma coleção pode definir que seus membros sejam todos de um mesmo tipo.

Um padrão pode definir um tipo especial de restrição estática: uma **restrição de tipo**. Esta define um tipo que um objeto deve ter para que possa estar em conformidade com o padrão.

O padrão do atributo *número da coluna*, na Figura 13, possui uma restrição de tipo, que indica que este atributo está restrito a objetos do tipo *Inteiro*. Adicionalmente, o padrão deste atributo também define uma restrição de intervalo. Já o padrão *ProprietárioDeTelefone*, de acordo com o definido na Figura 12, não estabelece uma restrição de tipo. Qualquer objeto, de qualquer tipo, que possua atributos *nome* e *fone*, conforme definido no padrão, pode ser encarado como um proprietário de telefone. Um atributo que seja restrito ao padrão *ProprietárioDeTelefone* pode conter qualquer objeto destes.

4.4.3.3. Padrões Tipos

Sob outro ponto de vista, um tipo representa também um conjunto de restrições aplicáveis a todos os seus objetos, ou seja, um padrão. Um objeto de um tipo atende a todas as restrições definidas pelo tipo.

Para tal, o modelo define também uma espécie de padrão sempre associado a um tipo, e que define as restrições que o caracterizam. Este padrão é denominado **padrão tipo** ou simplesmente de tipo. Um padrão tipo define, no nível conceitual, a existência de um tipo e lista as restrições que todos os objetos desse tipo atendem por implementação. Assim, pode-se dizer que um objeto de um tipo T está sempre em conformidade com o (padrão) tipo T correspondente.

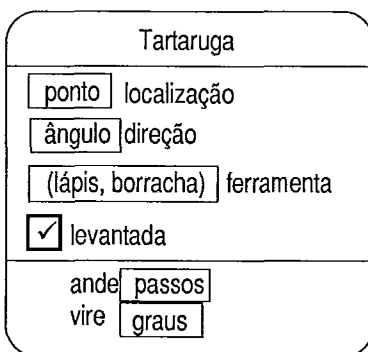


Figura 20: Especificação de um Tipo

A Figura 20 define, em um Diagrama de Objetos (DOO), o tipo *Tartaruga.* A diferença gráfica entre esta figura e a Figura 18, que definia o padrão *Tartaruga,* é que um padrão tipo é desenhado em uma caixa com linha sólida. O padrão *Tartaruga* especificava que qualquer objeto, de qualquer tipo, pois não possuía restrição de tipo, que possuísse um protocolo que contivesse o protocolo definido no padrão, estaria em conformidade com o padrão *Tartaruga.* Já o padrão tipo *Tartaruga* define que podem existir objetos do tipo *Tartaruga,* e que os objetos deste tipo sempre estão em conformidade com este padrão tipo, ou seja, possuem o protocolo e restrições definidas neste padrão tipo.

No modelo de objetos do GEOTABA, a modelagem conceitual da informação é baseada na definição de padrões, tipos e das restrições que os caracterizam. Cada definição de tipo estará associada a, possivelmente várias, definições de classes feitas no nível de implementação.

É conveniente procurar distinguir o conceito de tipo, conforme definido no modelo de objetos do GEOTABA, dos conceitos de classes e tipos frequentemente encontrados em outros modelos. Nestes modelos, é comum se ter apenas um dos conceitos, ou classes, ou tipos, e seja qual for o nome dado, estes costumam ter o poder de gerar objetos. No GEOTABA, como será visto mais adiante, na descrição do nível de implementação, os conceitos de classes e tipos são distintos. Classes, que são definidas no nível de implementação, é que têm o poder de gerar objetos. Um tipo apenas descreve um conjunto de propriedades que os objetos, de diversas classes, mas que estiverem em conformidade com o tipo, possuem. Tipos escondem os detalhes de implementação dos objetos, detalhes estes definidos nas classes de objetos. Mais de uma classe podem gerar objetos em conformidade com um dado tipo[22], isto será definido pelas próprias classes.

Cada tipo definido no nível conceitual implica na criação de uma classe correspondente, no nível de implementação. Esta classe implementa o tipo, por *default.* Outras classes podem ser geradas, a partir desta, implementando o mesmo tipo de modo diferente. Todavia, todas estas implementações têm que ser compatíveis com o que foi definido para o tipo no nível conceitual. Se o tipo não é um pseudotipo, as suas classes correspondentes têm que ser capazes de **criar** objetos compatíveis com o tipo. Portanto, a criação de objetos é feita enviando-se mensagens próprias a classes que implementam tipo, ou ao próprio

[22]Em particular, cada tipo verdadeiro, isto é, que não seja um pseudotipo, tem uma classe associada, com o mesmo nome do tipo, que gera objetos deste tipo.

tipo. Neste último caso, a mensagem será repassada para a classe *default* do tipo.

A expressão DEMO (17)

```
Tartaruga <     localização ⇒ ponto<0;0>
                ferramenta ⇒ borracha
                levantada ⇒ não
                direção ⇒ 45°>
```

instancia um objeto do tipo *Tartaruga*, utilizando uma forma genérica de instanciação. A instanciação é aplicável a classes de implementação, seguindo-se uma tupla após o nome da classe. Quando usado um nome de tipo, como neste exemplo, será aplicada à classe *default* do tipo. Uma classe pode definir outros métodos de instanciação, porém esta forma genérica é gerada para a classe de implementação *default* de cada tipo.

A tupla define os valores iniciais de alguns atributos do novo objeto. Todos os atributos obrigatórios devem ser definidos. Ao seu atributo *ferramenta* será atribuído o objeto designado pelo literal *borracha* e o atributo *direção* receberá o resultado da expressão *45°*. O operador unário posfixo ° , quando aplicado a um número, resulta em um ângulo. O valor de *localização* é uma nova instância de *Ponto*, porém, no lugar de uma tupla, foi fornecido um vetor. Assim fica exemplficado que os valores iniciais podem também ser fornecidos posicionalmente. No vetor apenas os valores são listados, omitindo-se as chaves das associações.

Alguns poucos sistemas separam os conceitos de tipos e classes. A linguagem Emerald [BLAC86] [RICH91] é um exemplo, embora classes sejam chamadas de "implementação". Em Emerald não há herança de propriedades nos tipos, nem nas "implementações", o que não propicia o compartilhamento de código. Uma "implementação" cria implicitamente um tipo correspondente ao seu protocolo, embora possa-se criar tipo que não corresponda a qualquer "implementação". No modelo de objetos do GEOTABA, são os padrões que se assemelhariam aos tipos de Emerald. A existência de vínculos de herança entre padrões e subpadrões e, mais especificamente, entre tipos e subtipos e entre classes e subclasses fornece um modelo mais apropriado para caracterizar o mundo real e para o compartilhamento de código. Outro aspecto importante é que cada tipo gera implicitamente uma classe com a implementação daquele tipo. De qualquer modo, uma implementação (classe) no GEOTABA é consequência da definição de um tipo no nível conceitual e não o contrário, como é comum nas linguagens de programação como Emerald.

4.4.3.3.1. Subtipos

Um tipo pode ser definido a partir de outro(s) tipo(s). Neste caso, diz-se que aquele é **subtipo** deste(s) e este(s) **supertipo(s)** daquele. Se S é um subtipo de T1, T2, ... Tn, então qualquer objeto de S também está em conformidade com seus supertipos T1, T2, ... e Tn. Ou seja, o domínio de S estará contido na interseção dos domínios de seus supertipos. A recíproca, porém, não é verdadeira, pois podem existir objetos de um tipo T que não sejam de qualquer um subtipo dele. Mais ainda, podem existir objetos simultaneamente de T1, T2, ... e Tn e que não sejam de S.

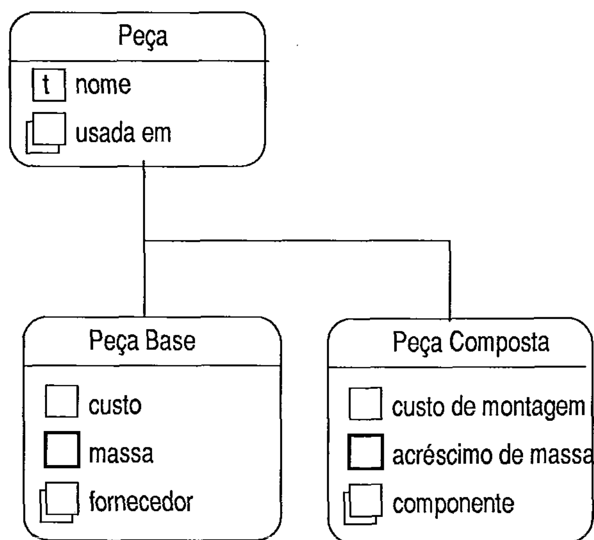


Figura 21: Hierarquização de Tipos

A Figura 21 redefine a Figura 17 transformando *Peça*, *Peça Base* e *Peça Composta* em tipos, sendo que *Peça* é supertipo de *Peça Base* e *Peça Composta*. Toda peça base ou composta é do tipo *Peça*. Todavia, o diagrama desta figura não impede que possam existir peças que não sejam base nem composta. Os subtipos, porém, são mutuamente exclusivos, isto é, não existe objeto que seja simultaneamente uma peça base e composta.

Se todos os objetos de um tipo T forem, obrigatoriamente, de algum subtipo de T, é dito que T é um **pseudotipo**. A função de um pseudotipo é definir propriedades a serem compartilhadas pelos seus subtipos, não havendo objetos que sejam apenas do pseudotipo. O domínio de um pseudotipo é sempre exatamente igual a união dos domínios de seus subtipos.

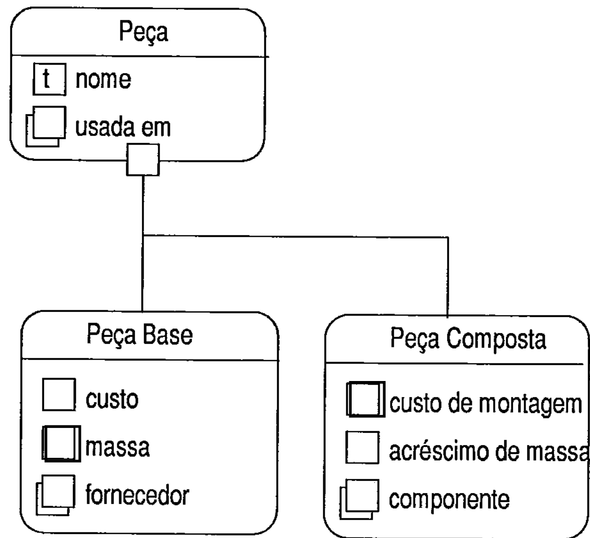


Figura 22: Especificação de um Pseudotipo

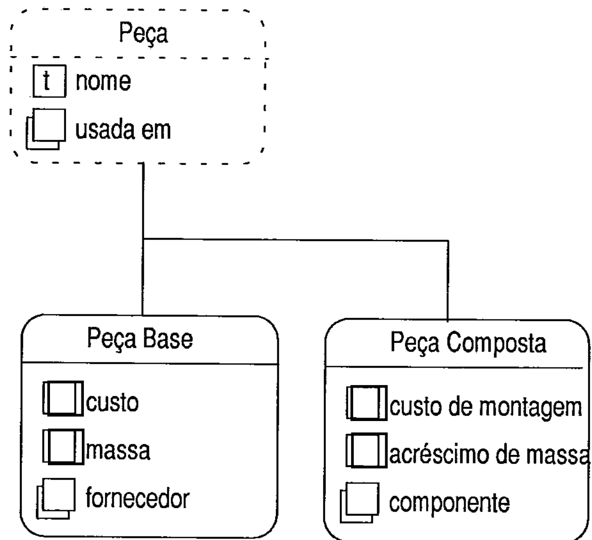


Figura 23: Superpadrão de Tipos

O pequeno quadrado abaixo da caixa de *Peça, n*a Figura 22, especifica que *Peça* é um pseudotipo, ou seja, não há objetos do tipo *Peça* que não sejam do tipo *Peça Base* ou do tipo *Peça Composta.* Já a Figura 23 define que *Peça* não é um tipo, mas apenas um superpadrão de *Peça Base* e *Peça Composta.* Neste caso, para um objeto estar em conformidade com *Peça,* basta possuir os atributos *nome* e *usada em* com as restrições definidas neste padrão, podendo ser de qualquer tipo. De qualquer modo, os objetos dos tipos *Peça Base* e *Peça Composta* sempre estão em conformidade com *Peça.* Modelagens do tipo da Figura 22 devem ser muito mais empregadas do que as do tipo da Figura 23. Padrões tipos são mais comuns do que padrões que não são tipos porque estes devem ser usados apenas quando a verificação de tipo dos objetos não for desejável.

Se todos os objetos em conformidade com um tipo T atendem as restrições definidas neste, os objetos em conformidade com um subtipo S de T, por estarem em conformidade com T, também atendem estas restrições. É dito, portanto, que um tipo **herda** as propriedades de seus supertipos. Todavia, um tipo pode acrescentar novas restrições além das herdadas de seus supertipos. O acréscimo de uma restrição pode ter a forma de uma **redefinição de propriedade.**

O protocolo de um subtipo está contido na união dos protocolos de seus supertipos. Ou seja, um objeto conforme um tipo pode sofrer qualquer das ações definidas em qualquer supertipo do tipo. Um subtipo pode definir uma restrição de protocolo adicional. Uma restrição de protocolo adiciona novas ações ao protocolo do subtipo. Observe-se que, paradoxalmente, uma restrição de protocolo não restringe ou diminui o protocolo do tipo, mas está associada a uma redução do domínio do tipo, estendendo o protocolo.

Um exemplo comum de redefinição de propriedades de um tipo por um seu subtipo é a redefinição do padrão de algum atributo herdado. Esta redefinição só pode ser feita se o padrão do atributo no subtipo for um subtipo do padrão tipo do atributo herdado (ou uma restrição sobre este). As mesmas regras valem para a redefinição de qualquer padrão de argumento de ação.

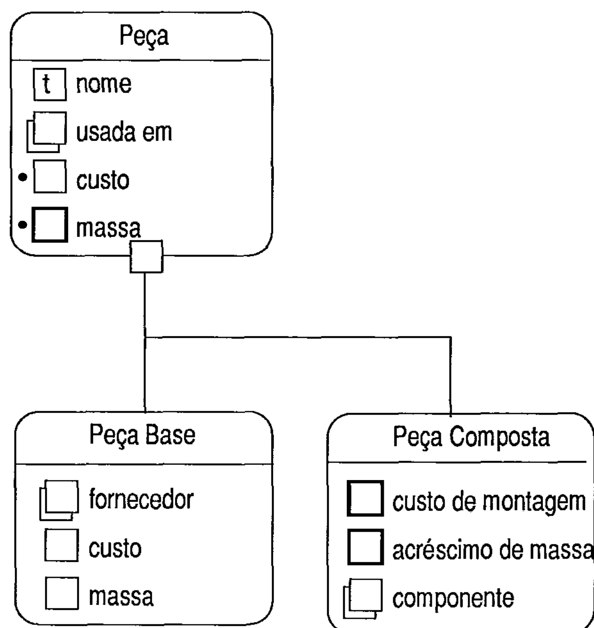


Figura 24: Redefinição de Propriedades

Na Figura 24, os atributos *custo* e *massa*, que, na Figura 22, só existiam em *Peça Base*, passaram a fazer parte, agora, do pseudotipo *Peça*. Isto significará que, tanto *Peça Base*, quanto *Peça Composta*, possuirão estes atributos. Todavia, estes atributos foram definidos como inalteráveis. Isto foi especificado através da colocação de um "•" à esquerda da caixa de atributo. O resultado é que, para peças compostas, pode-se apenas consultar seu custo e massa, não sendo possível fazer uma atribuição a qualquer destes dois atributos. Estes atributos aparecem redefinidos como alteráveis em *Peça Base*, que, por consequência, permite a atribuição a eles no seu protocolo.

É comum que um subtipo queira redefinir algumas ou mesmo todas a-ções de um supertipo. Por exemplo, os tipos *idade* e *velocidade* podem ser subtipos do tipo *número inteiro*. Porém as operações sobre inteiros, como a subtração e as comparações, podem não ser interessantes se aplicadas misturando idades com velocidades. Redefinir essas operações, criando, por e-xemplo, subtração de idades e subtração de velocidades, não resolve o problema. Uma ação que subtraísse uma idade de uma velocidade, casaria com a subtração de inteiros puros, pois idades e velocidades estariam em conformidade com o tipo *número inteiro*. Isto é resolvido definindo-se que *idade* e *velocidade* não são subtipos do tipo *número inteiro*. Elas serão tipos que compartilharão a mesma implementação que os números inteiros, ou seja, a sua classe, como será visto mais adiante. Este é um exemplo de que o conceito

de padrões (similar ao conceito de tipos em Emerald e Melampus) não dispensa a necessidade de se ter tipos.

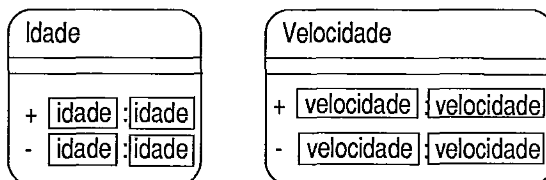


Figura 25: Especificação de Tipos Derivados

A Figura 25 mostra que os Diagramas de Objetos, restritos ao nível conceitual do modelo, não representam que os tipos *Idade* e *Velocidade* compartilham a mesma implementação dos números inteiros. *Inteiro*, *Idade* e *Velocidade* são, neste nível, tipos totalmente independentes entre si. Como não possuem atributos, apenas os serviços que implementam as operações válidas para *Idade* e *Velocidade*, soma e subtração, são representados. Os dois pontos sinalizam que o padrão que se segue é o padrão, com que o objeto retornado como resposta, estará em conformidade. Isto permite a especificação deste padrão ao final, ao invés do início, como usado nos atributos dos exemplos anteriores. Neste caso, preferiu-se usar esta notação por questão de legibilidade, já que os serviços representavam operações binárias e a outra notação induziria o usuário a uma interpretação errônea da especificação.

Na especificação do padrão de uma associação ou de um atributo de uma ação, restrições adicionais podem ser colocadas. Por exemplo, um atributo restrito a inteiros pode ser restringido ainda mais colocando-se um limite inferior e/ou superior para seus valores. Estas restrições adicionais não definem um novo tipo, nem um subtipo, mas sim um novo padrão.

4.4.3.4. Componentes

Um objeto pode possuir componentes, que são outros objetos percebidos como sendo parte do primeiro (ver 3.2.2.1.10.:"Composição"). Não é comum que um objeto seja parte simultaneamente de dois objetos distintos, a não ser que um deles seja parte do outro. Para que este tipo de restrição possa ser especificada, atributos que indicam que os objetos associados por eles são partes do objeto são definidos como **componentes**, isto é, sujeitos à restrição de **composição**. A principal característica da composição é a exclusividade, que indica que um objeto pode ser referenciado em associações de outros objetos; porém não pode ser componente de nenhum outro objeto. A restrição de

composição é verificada dinamicamente, mas apenas nas atribuições para atributos componentes.

A finalidade da distinção destes atributos é dar informação semântica para a movimentação e reprodução dos objetos. Por exemplo, a cópia de objetos deve levar em consideração esta informação e duplicar também os componentes do objeto original.

Figura 26: Atributo Componente

A Figura 26 mostra uma maneira de se representar a composição de objetos em um Diagrama de Objetos. O sublinhado do atributo *dependentes* especifica que cada dependente de um empregado "pertence" a ele, não havendo outro empregado, ou qualquer outro objeto, que o possua simultaneamente. O nível de implementação pode utilizar esta informação para, por exemplo, armazenar junto a cada empregado os dados de seus dependentes.

4.4.3.5. Coleções

Outro tipo comum de restrição estática se refere a capacidade do objeto se associar. Uma **restrição de associação** define se o objeto pode ganhar novas associações e as restrições destas. É muito comum que um tipo de objetos não permita que novas associações, além das definidas no protocolo do padrão tipo, possam ser acrescentadas aos seus objetos. Um padrão tipo que permite que objetos em conformidade com ele adquiram novas associações está encarando cada um destes objetos como sendo uma coleção. Observe-se que, como um objeto pode estar em conformidade com mais de um padrão, um padrão define uma maneira de se interpretar o objeto. Portanto, embora não seja comum, um objeto pode ser visto como sendo uma coleção através de um padrão e como não permitindo adição de novas associações por outro padrão.

O comum é que a definição de coleções utilize um padrão tipo, neste caso denominado **tipo de coleção**. Este define as propriedades comuns a

determinadas coleções. Exemplos de tipos de coleções são conjuntos, dicionários e listas.

Um padrão pode definir uma associação (atributo, componente ou mapeamento) restrita a um padrão, baseado em um tipo de coleção. Por esta maneira se pode modelar atributos multivalorados e relacionamentos 1:n ou n:m. Uma associação restrita a um tipo de coleção define também uma restrição adicional que especifica o tipo dos elementos da coleção.

O ícone do atributo *dependentes* de *Empregado* na Figura 26 especifica que a restrição de padrão para este atributo associa o objeto a uma coleção. Isto equivale a dizer que cada empregado pode ter múltiplos dependentes. Como não há restrição explícita de tipo de elemento da coleção, qualquer objeto pode ser incluído nesta coleção. Ou seja, os elementos são do tipo *Objeto.* Os atributos *usada em, fornecedor* e *componente,* como podem ser vistos na Figura 24, também representam coleções.

No trecho (18)

```
variável emp: Empregado ⇒ <    nome ⇒ "João" >
variável t := < nome ⇒ "José" >
dependentes de emp ⊕= t
```

a primeira instrução declara uma variável de nome *emp*, restrita ao tipo *Empregado* e iniciada com um empregado criado a partir de uma tupla, a qual define que o nome do empregado criado será *"João"*. Outros atributos deste empregado deveriam ter sido fornecidos. Para simplificar o exemplo, não o foram. A segunda instrução define a variável *t*, sem restrições, isto é, podendo conter qualquer objeto, e atribui a ela uma tupla. Se no lugar da atribuição fosse criada uma associação (operador ⇒), *t* seria do padrão tipo *Tupla.* A terceira instrução acrescenta esta tupla à coleção de objetos dependentes do empregado *emp.* O trecho seguinte (19)

```
dependentes de emp ⊖= t
```

remove a tupla dos *dependentes de emp.*

O padrão de uma associação com coleção pode possuir restrições adicionais, por exemplo, definindo as **cardinalidades mínima e máxima** de todos os objetos coleções conforme ele. A cardinalidade mínima, por exemplo, pode definir se uma coleção pode ficar vazia, isto é, sem associações. Uma restrição de cardinalidade é um exemplo típico de uma restrição dinamica, que migra para o objeto no momento em que ele é incorporado à associação. Isto

evita que dois padrões definam cardinalidades conflitantes para o mesmo objeto e garante que cada membro incluído ou removido da coleção provocará a verificação de se as cardinalidades foram respeitadas.

Figura 27: Atributo Coleção Possivelmente Vazia

O ícone do atributo *dependentes* de *Empregado* na Figura 26 foi mudado na Figura 27 para especificar que a coleção de dependentes, que cada empregado possui, pode ser vazia.

Outras restrições podem ser aplicadas a coleções. Uma coleção pode ter restrições adicionais do tipo restrições agregadas da forma "o total de salário não pode exceder um dado valor". Uma restrição desse tipo deve ser expressa sobre a forma de uma expressão, isto é, uma lista de mensagens, que reponda um valor booleano. Uma alteração na coleção só é realmente efetivada se a restrição permanecer satisfeita.

4.4.3.6. Aspectos

A idéia inicial de aspectos foi introduzida como parte do modelo de dados Melampus [RICH91], para a modelagem dos múltiplos papéis que uma entidade pode cumprir durante a sua existência (ver 3.2.2.1.13.:"Aspectos"). No modelo de objetos do GEOTABA, um **aspecto** estende um objeto existente com novas associações e novos serviços, conservando a identidade própria do objeto. Embora subtipos possam definir que um gerente é um empregado, eles falham na modelagem das características dinâmicas e multifacetadas que as entidades do mundo real têm. Por exemplo, um empregado pode se tornar um gerente, ou deixá-lo de ser. Esta mudança de tipo dinâmica deve ser realizada preservando a identidade do objeto, pois o novo gerente é a mesma pessoa que anteriormente era apenas um empregado. Um objeto pode dinamicamente adquirir múltiplos aspectos e se liberar deles.

No modelo de dados Melampus, um aspecto é criado definindo-se a sua implementação. A sua interface implícita corresponderia a visão conceitual do aspecto. No GEOTABA, a abordagem é oposta. Um aspecto, do mesmo modo

que um tipo, é definido conceitualmente, uma implementação padrão sua é deduzida, e alterações nesta implementação, ou criação de outras, são permitidas.

Um **padrão aspecto**, semelhantemente a padrão tipo, é um padrão que define as propriedades de um aspecto. Um padrão aspecto, que algumas vezes pode ser chamado simplesmente de aspecto, estende algum padrão: o seu **padrão base**. O padrão base define quais propriedades um objeto tem que ter para que seja permitido que ele adquira dinamicamente o aspecto. Um padrão aspecto define restrições que o objeto que o adquirir passará a ter que respeitar, incluindo aí restrições de protocolo e de aspecto. A restrição de protocolo definirá uma extensão à interface do objeto, podendo esta definir novos atributos e novos serviços para um objeto quando ele adquirir o aspecto. A **restrição de aspecto** é semelhante à restrição de tipo: indica que apenas objetos com este aspecto poderão ser atribuídos a associações restritas a este padrão aspecto.

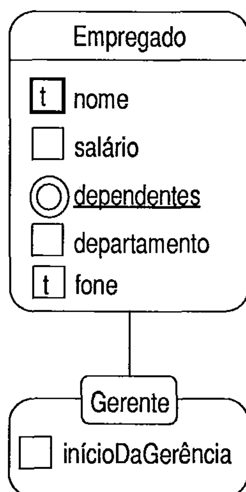


Figura 28: Especificação de um Aspecto

A Figura 28 mostra um fragmento de um Diagrama de Objetos que especifica um aspecto *Gerente* que objetos do tipo *Empregado* podem possuir. A representação de um aspecto difere da representação de subtipo no formato das caixas. Um tipo pode ter simultaneamente ligações de aspectos e de subtipos. No caso de isto acontecer, os subtipos herdam os aspectos de seus supertipos. A figura mostra que um empregado que adquirir um aspecto de gerente passará a possuir um atributo com a data de *início da gerência*. Entretanto, este atributo só é acessível em padrões restritos a este aspecto.

Outra diferença entre os modelos do GEOTABA e Melampus está no fato que, neste último, um aspecto não herda inerentemente as restrições definidas no seu padrão base. Por exemplo, um gerente não possuiria os atributos de empregado, a não ser que cada atributo deste fosse explicitamente exportado na definição do aspecto. No GEOTABA ocorre o oposto, o padrão aspecto estende o padrão base, herdando suas propriedades. Como consequência adicional, um objeto visto conforme um aspecto pode ser atribuído a uma associação restrita ao padrão base deste aspecto. Ou seja, um gerente pode ser atribuído a atributos restritos a empregados.

O trecho seguinte (20) mostra como pode-se estender um objeto com um aspecto.

```
variável empr: empregado
variável ger: gerente

empr := Empregado <
            nome ⇒ "João"
            salário ⇒ 555_000
            departamento⇒ Departamento["Brinquedos"] >

ger := empr gerente <início_da_gerência⇒data<01;01;90>>
```

Inicialmente criou-se um empregado de nome "João", referido pela variável *empr*. O travessão contido no valor do salário não é significativo, sendo usado apenas para dar maior legibilidade. A seguir, *empr* ganhou um aspecto de gerente. Na construção do aspecto foi atribuída uma data ao atributo *inícioDaGerência*. Os travessões aí também não são significativos. O resultado da construção do aspecto respeita o padrão do aspecto *gerente* e, por isso, pode ser atribuído à *ger*. Pode-se dizer que

(21)
```
ger ≡ empr
```

pois eles compartilham a mesma identidade. Entretanto, *empr* enxerga João como sendo um empregado e *ger* o vê como gerente.

O código seguinte (22)

```
variável empr1: empregado
empr1 := ger            \" VÁLIDO NO GEOTABA "

\"Empregado,Gerente: se conformam a ProprietárioDeTelefone
pt := empr1
prop := ger
```

mostra que *ger* pode ser atribuído a uma variável de padrão *Empregado*, pois, no GEOTABA, um aspecto é sempre um subpadrão do seu padrão base. Isto não ocorre no sistema Melampus, onde aspectos foram desenvolvidos pela primeira vez, porque, nele, a definição de um aspecto não herda implicitamente os atributos do seu tipo base.

A cadeia de caracteres precedida do caracter escape funciona como comentário. Uma cadeia de caracteres não ultrapassa o final da linha, a não ser que este seja precedido por escape. Portanto, as aspas finais de uma cadeia, que termine junto com a linha, podem ser dispensadas.

Aspectos são a maneira adequada de se modelar variações de tipo que não sejam excludentes entre si. Por exemplo, uma pessoa pode ser simultaneamente empregado e estudante. Não há necessidade de se definir um subtipo de pessoa empregado-estudante para as pessoas que são as duas coisas simultaneamente. Isto evita a proliferação combinatorial de subtipos dessa espécie. Observe-se que, com aspectos, uma pessoa pode ter um atributo departamento, para quando for encarada conforme seu aspecto de empregado, e outro para estudante. Não há conflito de herança, pois a manipulação de cada atributo só é possível através da utilização explícita do aspecto adequado do objeto. Ou seja, o atributo departamento para empregado só é acessível em associações restritas ao padrão aspecto empregado. Idem para departamento de estudantes.

De um modo geral, um objeto e seus aspectos formam uma árvore, por que um padrão aspecto pode ser usado como base para a definição de novos subaspectos. Um objeto pode ter diversos aspectos do mesmo "tipo". Por exemplo, uma mesma pessoa pode ser estudante em mais de uma escola. Por este motivo, é necessário existir uma operação que responda se dois aspectos do mesmo objeto são distintos. Aspectos podem ser uma ferramenta para se tratar versões de objetos [SCIO89].

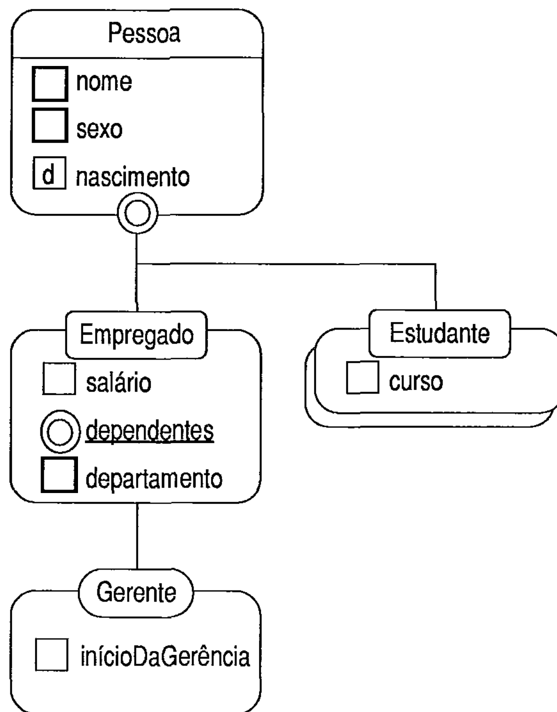


Figura 29: Hierarquia de Aspectos

A Figura 29 especifica que objetos do tipo *Pessoa* podem adquirir dinamicamente aspectos como empregado ou estudante. De acordo com esta modelagem, uma pessoa pode ter vários aspectos *Estudante*, o que é representado pela caixa deste padrão. Uma pessoa pode possuir simultaneamente aspectos como empregado e como estudante, não havendo necessidade de se definir um novo padrão *Empregado-Estudante*. Isto é representado pelo círculo duplo na linha inferior da caixa *Pessoa*. O aspecto *Empregado* possui um subaspecto, *Gerente*, indicando que pessoas que sejam empregados podem passar a ser gerente. Observe-se que as peças compostas e base, do diagrama da Figura 24, são melhores modeladas como subtipos do que como aspectos. Isto porque não é esperado que uma peça se transforme em base ou composta dinamicamente.

O exemplo apresentado em [RICH91], mostrando como aspectos podem ser utilizados para modelar papéis que entidades totalmente diversas podem assumir, é modelado no GEOTABA de maneira bem próxima. O exemplo se refere a uma base de dados para uma companhia telefônica, onde um proprietário de telefone pode ser uma pessoa ou uma empresa. Pessoas e empresas têm poucos atributos em comum: nome e número do telefone. Para cada usuário, devem ser mantidas informações sobre as chamadas realizadas,

além de serviços como emitir conta ou desconectar a linha. Em [RICH91] é definido um "tipo abstrato" *proprietário de telefone*, que define que qualquer objeto que possua atributos *nome* e *número do telefone* pode ser um proprietário de telefone. No GEOTABA, *ProprietárioDeTelefone* seria um padrão. Este padrão poderia definir adicionalmente uma restrição de tipo exigindo que um proprietário fosse do tipo *pessoa* ou do tipo *empresa*. Os "tipos abstratos" *pessoa* e *empresa* definem, entre outros atributos, *nome* e *número do telefone* . Portanto, em [RICH91], *pessoa* e *empresa* são subtipos de *proprietário de telefone*. Outros tipos de objetos, com atributos *nome* e *número do telefone* não podem ser impedidos de serem proprietários de telefone. Com a restrição de tipo acima, o GEOTABA pode evitar este problema. Para isto, *pessoa* e *empresa* têm que ser tipos (padrões tipos) ou aspectos, e não simplesmente padrões, pois objetos em conformidade com o padrão *pessoa* têm que ser do tipo *pessoa*.

Em [RICH91], é criado também o "tipo abstrato" *usuário* que, nos termos do modelo de objetos do GEOTABA, é a visão conceitual do aspecto que um objeto pode ter como usuário de telefone; o aspecto propriamente dito é definido por uma "implementação de aspecto". No GEOTABA, o nível conceitual admite a existência de aspectos e, portanto, *usuário* é definido como um aspecto que herda de *ProprietárioDeTelefone* seus atributos e serviços. Em [RICH91], embora seja dito que *usuário* estende *proprietário de telefone,* estes atributos e serviços têm que ser explicitamente exportados. Além disso, toda a implementação correspondente ao aspecto *usuário* tem que ser construída. No GEOTABA, a implementação de qualquer aspecto é gerada automaticamente, sendo necessário apenas programar eventuais modificações que interessarem. Neste exemplo, o aspecto *usuário* deve definir os serviços para emitir conta e desconectar linha, serviços estes dependentes de implementação. As informações sobre as chamadas realizadas não dizem respeito ao nível conceitual dos objetos; elas, e mais a implementação dos serviços acima, são programadas sobre a implementação gerada do aspecto, no nível de implementação. Mais de uma implementação para o mesmo aspecto podem conviver simultaneamente.

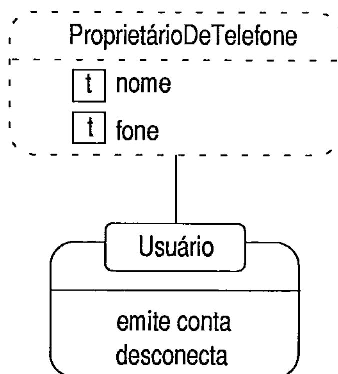


Figura 30: Aspecto de um Padrão

A Figura 30 modela em Diagrama de Objetos para o GEOTABA o exemplo apresentado em [RICH91]. O aspecto *Usuário* pode ser posto em qualquer objeto que esteja em conformidade com o padrão *ProprietárioDeTelefone*, isto é, que possua atributos textuais *nome* e *fone*. Um usuário recebe mensagens para emitir conta e desconectar linha, além de ocultar sua implementação específica.

Qualquer objeto do tipo *pessoa* ou *empresa* estará em conformidade com o padrão *ProprietárioDeTelefone,* por possuírem *nome* e *número do telefone* e serem dos tipos permitidos neste padrão. Qualquer objeto destes poderá, dinamicamente, adquirir o aspecto de *usuário* (de telefone) e, sempre que forem vistos sob este aspecto, fornecerem os serviços específicos a usuários: emitir conta e desconectar linha. Por exemplo, o objeto pode ser atribuído a uma variável de "tipo" *usuário*, ou seja, restrita ao padrão *usuário*, e esta variável será apta a receber mensagens para emitir conta e desconectar linha.

É relevante se observar então que a especificação de um tipo pode prever a existência de aspectos em seus objetos, mas aspectos podem ser colocados em objetos cujo tipo não os previa. É o que ocorre no exemplo anterior, onde foi definido um padrão de aspecto válido a objetos, de qualquer tipo, que tenham conformidade com *ProprietárioDeTelefone*. Consequentemente, padrões tipos, apesar de descreverem propriedades presentes em todos os objetos do tipo, não descrevem todas as propriedades de seus objetos. Se um tipo de um objeto é definido por um padrão que descreve totalmente o objeto (primeiro caso), este padrão tipo é o metaobjeto do objeto em questão. No segundo caso, o metaobjeto não pode ser um padrão tipo, e sim um outro espécime de metaobjeto que, todavia, fará referência ao padrão tipo do objeto, pois as propriedades descritas nele continuam válidas. Uma empresa x estaria descrita pelo seu metaobjeto, que seria inicialmente o padrão tipo *Empresa*. Esta empresa x, por ter conformidade com o padrão *ProprietárioDeTelefone*, poderia

adquirir dinamicamente um aspecto de *usuário* (de telefone). A partir daí o padrão tipo *Empresa* não descreve mais *x* plenamente. O metaobjeto de *x* será outro, que incluirá a descrição do aspecto acrescentado. Como *x* continuará sendo do tipo *Empresa*, o novo metaobjeto se referirá ao padrão tipo *Empresa* para descrever as propriedades do objeto quando visto apenas como uma empresa.

Um objeto pode ser consultado sobre se possui um dado aspecto. Caso ele possua, certamente, a seguir, haverá ações que o encarem sob este aspecto. As linguagens de manipulação de objetos devem ter construções especiais para tratar esta situação.

O modelo de objetos do GEOTABA também define uma maneira própria de tratar a remoção de aspectos. Ao contrário de quando se acrescenta um aspecto a um objeto, a remoção arbitrária de aspectos poderia causar referências pendentes em outros objetos. O GEOTABA só permite a remoção de um aspecto de um objeto se este aspecto não estiver sendo referenciado por nenhum outro objeto, ou seja, se nenhum objeto estiver enxergando o objeto base através deste aspecto.

4.4.3.7. Ligações

O relacionamento entre objetos é submetido a restrições específicas definidas nos tipos dos objetos (ver 3.2.2.1.14.:"Suporte a Relacionamentos"). Por exemplo, seja a ***SITUAÇÃO 1***, onde um objeto que represente um dado departamento de uma empresa possui um atributo que o associa a uma coleção de empregados. Em outras palavras, o departamento possui uma associação com nome, um atributo portanto, que o liga a um objeto coleção, cujos membros são objetos em conformidade com o tipo *empregado*. Por sua vez, cada empregado possui um atributo que o associa ao departamento em que ele está alocado. Essas associações estão sujeitas à seguinte restrição: os objetos membros da coleção de empregados de um departamento D têm que estar alocados ao departamento D e, além disso, o departamento de um empregado E tem que possuir este empregado na sua coleção de empregados.

Este tipo de restrição, chamada de **ligação**, corresponde a um relacionamento, no sentido empregado pelo modelo Entidade-Relacionamento, e vincula uma associação em um objeto (o atributo coleção de empregados de um departamento, por exemplo) a outra associação em outro objeto (o atributo departamento de um empregado). Cada uma dessas associações é um **mapeamento**. A associação da outra extremidade da ligação de uma associação é chamada de **mapeamento inverso** desta. A existência de uma ligação entre objetos implica na existência de associações nos objetos ligados, que

implementem os mapeamentos correspondentes. Os mapeamentos caracterizam um objeto em função de seus relacionamentos com outros objetos. Um mapeamento normalmente é um atributo, ou seja, tem um nome que serve também para identificar as mensagens de recuperação e atribuição dos objetos mapeados.

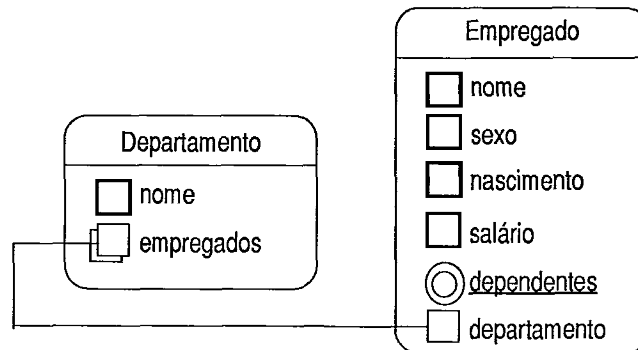


Figura 31: Especificação de uma Ligação

A Figura 31 mostra um Diagrama de Objetos que especifica a ligação descrita na situação 2.

Se *dep* é uma variável de tipo *Departamento* e *emp* outra, de tipo *Empregado*, (23)

```
departamento de emp := dep
```

causará, implicitamente, que, antes da atribuição ser realizada, *emp* será excluído da coleção *empregados de departamento de emp*, e que, após a atribuição, *emp* será acrescentado à coleção *empregados* do novo *departamento de emp*.

Simetricamente, (24)

```
empregados de dep ⊕= emp
```

após acrescentar o empregado *emp* à coleção de *empregados de dep*, implicitamente alterará o *departamento de emp* para *dep*. Consequentemente, antes disso tudo, *emp* será excluído da coleção *empregados de departamento de emp*.

Uma ligação representa uma restrição que ocorre entre objetos e não é específica de qualquer um deles. Se, por um lado, isto quebra a harmonia da o-rientação a objetos, os mapeamentos de uma ligação são as visões orientadas ao objeto dos relacionamentos entre objetos.

Uma ligação pode ser encarada como um objeto. Porém, uma ligação entre objetos só é acessada através dos seus mapeamentos nos objetos envolvidos. Isto significa que praticamente não se manipula diretamente, no nível conceitual, um objeto ligação. Este funciona como uma ponte transparente entre objetos. Como este associa diversos objetos, é necessário que, de cada objeto associado, se possa alcançar os outros objetos envolvidos na associação.

Uma ligação pode possuir atributos, representando os atributos do relacionamento. Como não é desejável que se acesse explicitamente o objeto ligação para se ter acesso a um atributo seu, cada objeto participante da ligação enxerga este atributo como sendo do outro objeto da ligação.

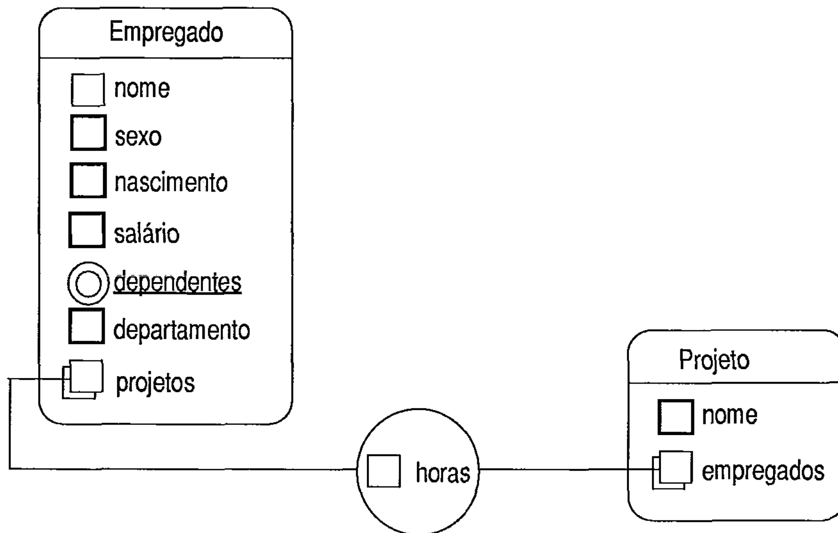


Figura 32: Ligação com Atributo

A Figura 32 mostra um fragmento de Diagrama de Objetos que define a existência de objetos empregados e projetos, de modo a que um empregado possa participar de diversos projetos e um projeto possa ter diversos empregados alocados a ele. O número de horas por semana que um empregado tem que trabalhar em um dado projeto é um atributo da ligação entre o empregado e o projeto, pois em cada projeto o empregado gastará um número de horas diferente e, em um mesmo projeto, cada empregado estará alocado por um número de horas diferente. A ligação, portanto, possui um atributo: *horas*.

De acordo com a Figura 32, uma ligação entre um empregado e um projeto permite que, do ponto de vista de um projeto, este atributo seja visto como sendo do empregado e possa-se consultar o projeto sobre o número de horas alocado de um empregado; do ponto de vista de um empregado, este atributo é visto como sendo do projeto e pode-se consultar o empregado sobre o número de horas alocado a um projeto. O código DEMO (25)

```
impressão de horas de empregados["João"] de proj
impressão de horas de projetos["x.0"] de empr
```

considera que *proj* e *empr* são variáveis de tipo *Empregado* e *Projeto,* respectivamente, e que os atributos *nome* de empregados e de projetos servem como índice das coleções de empregados e projetos, o que não está mostrado na Figura 32. O código mostra como o atributo da ligação (*horas*) é percebido, ora como sendo de empregado, ora como sendo de projeto. Isto é possível porque o padrão dos elementos de *empregados de proj* (ou de *projetos de empr*) não é *Empregado* (ou *Projeto*), mas um padrão que estende este(s) padrões tipos com os atributos da ligação.

Um aspecto ímpar do modelo de objetos do GEOTABA, não encontrado em outros modelos assemelhados, é o do impacto, nos relacionamentos, da redefinição de propriedades pelos subtipos. Seja a ***SITUAÇÃO 2***, derivada da situação 1, porém considerando que cada departamento possui um gerente, que é um empregado. Isto poderia ser modelado definindo-se que cada departamento possui um atributo *gerente.* Pode-se desejar que cada gerente possua um atributo definindo qual departamento ele gerencia. Como todo gerente é um empregado, ele estará alocado a um departamento, como discutido na situação 2. É razoável que um gerente esteja alocado ao mesmo departamento que ele gerencia. Para isto, de algum modo, o atributo *departamento gerenciado* de um gerente deve ser obrigado a ser igual ao seu atributo *departamento alocado.* Isto pode ser modelado definindo-se que gerente é um subtipo de empregado e que, no caso de gerente, o mapeamento inverso de *departamento alocado* é o atributo *gerente* de departamento. Assim, não seria necessário se definir um atributo *departamento gerenciado* em gerente, pois este papel seria cumprido pelo atributo *departamento alocado.*

O atributo *departamento alocado* de um empregado, normalmente, é uma associação com um departamento através da ligação com o atributo *empregados alocados* do departamento. Porém, no caso do empregado ser um gerente, o atributo *departamento alocado* do empregado seria redefinido e estaria ligado ao atributo *gerente* do departamento. Já um departamento terá um

atributo *empregados alocados* com uma coleção de empregados ligados através do atributo *departamento alocado* deles e o atributo *gerente* ligado ao atributo *departamento alocado* de um empregado-gerente.

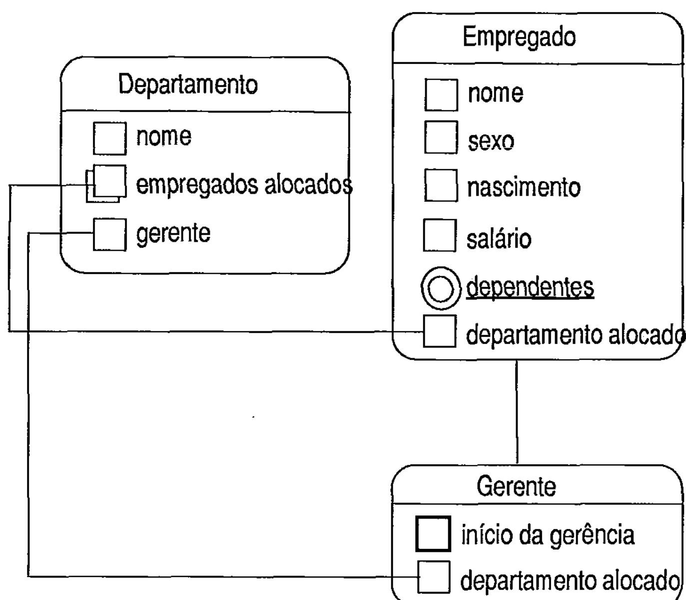


Figura 33: Redefinição de Mapeamento

A Figura 33 representa a situação 2. A ligação do atributo *departamento alocado* de um empregado indica que ele é o mapeamento inverso de *empregados alocados* de *Departamento.* Quando um empregado é um gerente, este atributo é redefinido, passando a ser o mapeamento inverso do atributo *gerente* de *Departamento.* Assim, o departamento alocado de um gerente é o próprio departamento que ele gerencia. Como um subtipo tem que obedecer a todas as restrições definidas no seu supertipo, o *departamento alocado* de um gerente tem que continuar a respeitar a condição de possuir este empregado-gerente na sua coleção de *empregados alocados*, condição esta herdada do supertipo *Empregado.*

Observe-se que se *Gerente* fosse um aspecto, e não um subtipo, o atributo *departamento alocado* não estaria sendo redefinido. Isto porque um gerente possuiria um *departamento alocado,* quando fosse visto como gerente, totalmente independente do seu *departamento alocado,* quando visto como um simples empregado. A consequência disto é que estas duas ligações modelariam corretamente o fato que um gerente tem que estar alocado ao mesmo departamento que ele gerencia, porém estaria em aberto a possibilidade de um departamento ter como gerente um empregado que não estivesse incluído na coleção de empregados alocados do departamento. Se isto fosse o desejado, isto é, os

empregados alocados de um departamento só serem funcionários subalternos, deveria ser apropriado que os departamentos fornecessem, como serviço, o cálculo da coleção total de empregados, incluindo nesta o gerente. É bem razoável que esta seja o resultado de um serviço, e não um atributo, pois a inclusão de um empregado a um departamento deve ser diferente no caso de ser como gerente do departamento ou como um subalterno.

Ainda assim deve ser apropriado que um departamento não permita a inclusão do gerente na sua coleção de empregados subalternos. Isto pode ser modelado criando-se, além de *gerente*, outro aspecto de *empregado*: *funcionário* subalterno. Neste caso, a coleção de empregados subalternos de um departamento estaria restrita a objetos em conformidade com o aspecto *funcionário* subalterno. Um objeto do tipo *Empregado* deve sempre possuir um dos dois aspectos, e apenas um deles, indicando que não há empregados que não sejam ou gerentes ou funcionários subalternos. Tanto *gerente* de *Departamento*, quanto *Funcionários* de *Departamento*, devem ter como mapeamento inverso *departamento* de *Empregado*. Assim, o departamento do gerente de um departamento *D* seria *D*, e o departamento de cada funcionário de *D* seria *D* também. A consequência disso é que *departamento* de *Empregado* teria dois mapeamentos inversos, indicando que um empregado *E* seria, ou o gerente, ou um funcionário, do seu departamento. A ligação do atributo *departamento* de *Empregado* com os atributos *gerente* de *Departamento* e *Funcionários* de *Departamento é* dita possuir uma **bifurcação.**

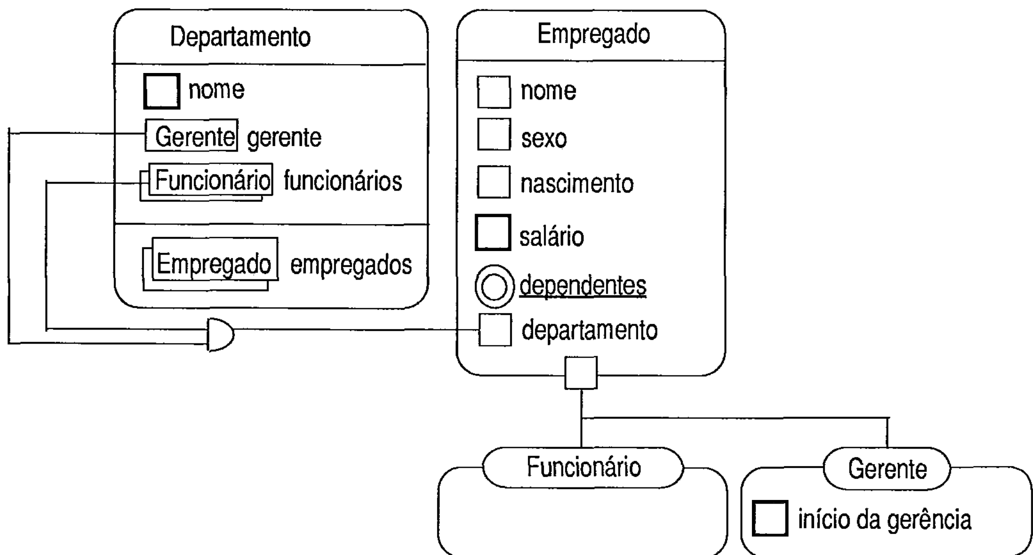


Figura 34: Ligação com Bifurcação

A Figura 34 remodela a situação 2 definindo que um empregado pode possuir aspecto de gerente ou de funcionário. O quadrado sob a caixa de *Empregado* indica que um empregado sempre possuirá um destes aspectos, e não mais do que um. O atributo *gerente* de *Departamento* foi restringido a empregados que possuam o aspecto *Gerente*, e o atributo *funcionários* de *Departamento* a empregados com o aspecto *Funcionário*. Como estes dois atributos possuem como mapeamento inverso o mesmo atributo *departamento* de *Empregado*, eles são ligados a este por uma **ligação com bifurcação**, representada pela meia-lua no diagrama. *Departamento* também oferece um serviço *empregados*, que calcula uma coleção de empregados. Este cálculo, não representado na figura, é totalmente independente do nível de implementação, dando como resposta a união dos atributos *funcionários e gerente* do departamento.

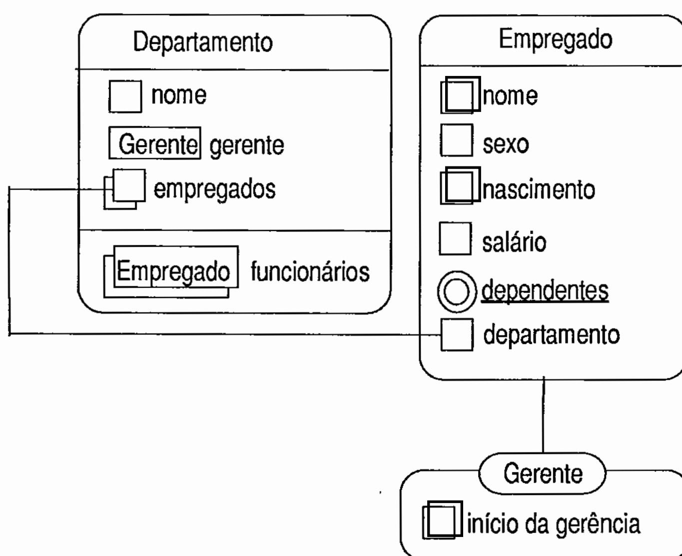


Figura 35: Atributo sem Mapeamento Inverso Explícito

A Figura 35 apresenta outra modelagem alternativa para a situação 2. Aqui *Empregado* possui um único aspecto *Gerente*, opcional e não múltiplo. *Departamento* possui o atributo *empregados* com todos os empregados do departamento. Como o atributo *gerente* de *Departamento* está restrito ao aspecto *Gerente*, cujo atributo *departamento* fornece o mesmo departamento que possui o empregado-gerente na sua coleção de empregados (devido a ligação *empregados-departamento*), é garantido que o gerente está incluído na coleção de empregados do departamento. Um serviço *funcionários*, independente do nível de implementação, foi incluído para responder os empregados de um departamento que não sejam gerentes.

Estas restrições exemplificadas, bastante reais e práticas, podem, como foi mostrado, ser modeladas no modelo de objetos do GEOTABA usando elementos do seu ferramental básico: ligações, aspectos, tipos e subtipos e redefinição de propriedades nos subtipos. Em outros modelos semânticos, como, por exemplo, o modelo entidade-relacionamento esta tarefa não é tão elementar. Mesmo em suas extensões com herança de propriedades, a redefinição de propriedades não encontra o tratamento equivalente ao encontrado nas linguagens de programação orientadas a objetos e no modelo de objetos do GEOTABA.

Um relacionamento binário sem atributos próprios pode ser representado trivialmente por uma ligação, ficando totalmente determinado pelos mapeamentos que levam de um objeto ao outro e vice-versa. O objeto associado por um mapeamento pode ser também uma coleção, o que serve para representar relacionamentos 1:N e N:M. Relacionamentos com atributos próprios e relacionamentos de grau 3 exigem características especiais nas ligações, como atributos, que tornam as ligações parecidas com objetos quaisquer.

As ligações apresentadas nos Diagramas de Objetos das Figuras 31 à 35 todas se referem a relacionamentos binários. Por exemplo, a ligação *Gerente-Departamento* da Figura 33 representa um relacionamento 1:1: cada departamento possui um único gerente e este está alocado a um só departamento. Já a ligação *Empregado-Departamento,* da mesma figura, se refere a um relacionamento 1:N: cada empregado é de um departamento, mas este possui diversos empregados. A ligação *Empregado-Projeto,* da Figura 32, especifica um relacionamento M:N: cada empregado pode estar alocado a vários projetos e cada projeto pode ter vários empregados alocados. O atributo deste relacionamento é representado em uma caixa colocada sobre a ligação. Esta caixa não apareceria no diagrama caso este relacionamento não possuísse atributos.

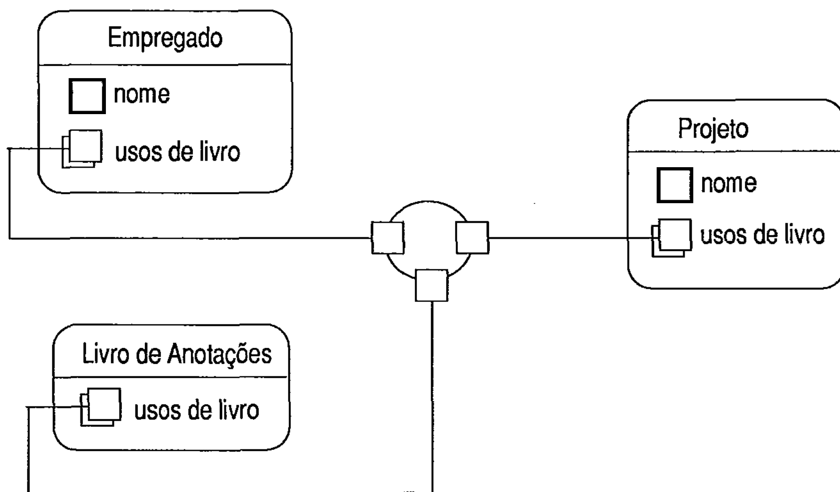


Figura 36: Ligação Representando um Relacionamento Ternário

A Figura 36 mostra a representação de um relacionamento ternário em um Diagrama de Objetos. O exemplo foi adaptado de [TEOR86], onde são discutidos relacionamentos ternários e dependências funcionais. No exemplo, um empregado usa um livro de anotações para um dado projeto. Empregados diferentes usam livros diferentes para o mesmo projeto, mas diferentes engenheiros podem usar o mesmo livro para diferentes projetos. Portanto há as seguintes dependências funcionais: *(empregado, projeto) ---> livro de anotações, (livro de anotações, projeto) ---> empregado, (empregado, livro de anotações) ---> projeto.*

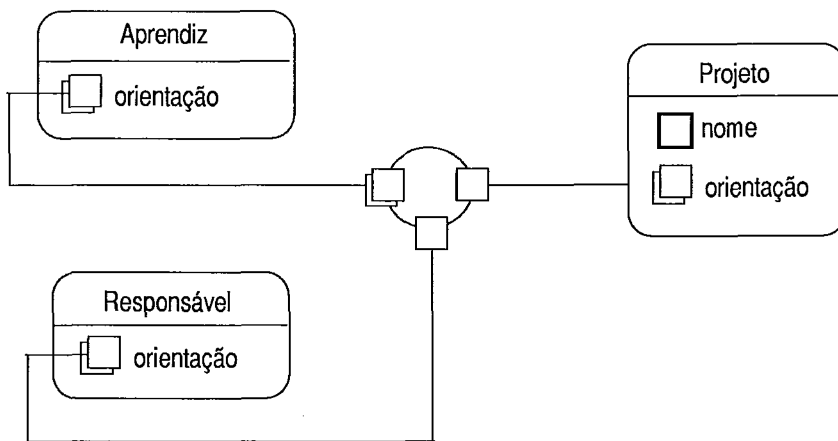


Figura 37 : Outro Exemplo de Ligação Ternária

A Figura 37 mostra outro exemplo de relacionamento ternário extraído de [TEOR86]. Neste exemplo, aprendizes trabalham em projetos sob a supervisão de responsáveis. Nenhum responsável pode orientar qualquer dado aprendiz em mais de um projeto. Nenhum aprendiz pode trabalhar em um dado projeto sob a instrução de mais de um responsável. As dependências funcionais *(responsável, aprendiz)* ---> *projeto* e *(aprendiz, projeto)* ---> *responsável* são representadas pelos quadrados simples que fazem a conexão com *Projeto* e *Responsável* , indicando que, para cada par (aprendiz, responsável), há um único projeto e que, para cada par (aprendiz, projeto), há um único responsável. O quadrado duplo, característico de coleção, indica que para cada par (responsável, projeto), pode haver múltiplos aprendizes. A colocação do quadrado duplo é opcional, pois ele é subentendido na ausência de quadrado simples.

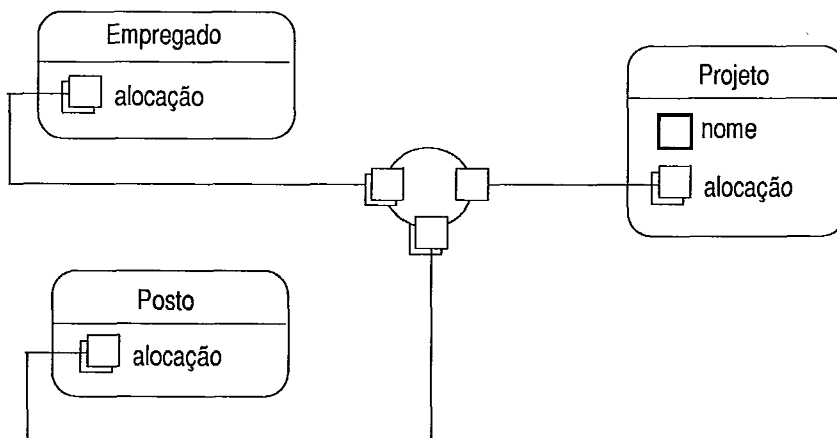


Figura 38: Relacionamento Ternário com Dependência Funcional Única

Um terceiro exemplo de [TEOR86] é apresentado na Figura 38. Neste, cada empregado é designado para um ou mais projetos, mas só pode ser designado para um único projeto em um dado posto de trabalho. O quadrado simples representa a única dependência funcional neste relacionamento: *(empregado, posto)* ---> *Projeto.*

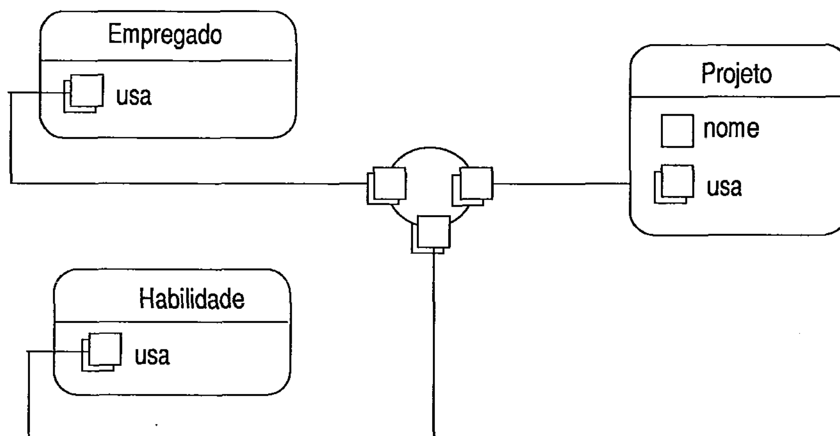


Figura 39: Relacionamento Ternário sem Dependência Funcional

Um último exemplo de relacionamento ternário adaptado de [TEOR86] aparece na Figura 39. Neste caso, cada empregado pode usar cada uma de suas habilidades em diversos projetos; cada projeto pode ter cada um de seus empregados alocados usando diversas habilidades; e cada habilidade necessária em cada projeto pode ser exercida por diversos empregados.

É importante observar que o modelo permite que se modele relacionamentos tanto através de ligações entre objetos, como através da simples associação entre objetos. Neste último caso, a restrição de ligação entre os objetos, que os obriga a se referenciarem mutuamente, não fica representada. Mais ainda, um objeto pode estar associado a outro, por um atributo seu, por e-xemplo, mas este outro pode não estar associado ao primeiro. Isto é o que diferenciará um atributo comum de uma ligação.

Observe-se que, na Figura 35, o relacionamento entre departamentos e gerentes não está representado por uma ligação explícita entre o atributo *gerente* de *Departamento* e algum atributo de *Gerente*. O atributo *localização* de *Tartaruga*, na Figura 18, associa uma tartaruga a um ponto. Uma ligação entre *Tartaruga* e o tipo *Ponto* não é desejável, por não ser esperado que um ponto conheça todos os objetos que o referenciem.

4.4.3.8. Restrições em Ligações

Diversos tipos de restrições podem ser definidos sobre relacionamentos. Um desses é a padronização de mapeamento. Como, do ponto de vista do obje-

to, atributos, componentes e mapeamentos são semelhantes, as mesmas regras de padronização são válidas sobre eles.

Um exemplo de padronização de mapeamentos foi mostrado na Figura 34, onde os padrões dos mapeamentos *gerente* e *funcionários* de *Departamento* possuem, ambos, restrições de aspectos. Assim, um departamento só pode estar associado, pelo atributo *gerente*, a um empregado sendo visto pelo seu aspecto *Gerente*. Obviamente, nem todos empregados possuem esse aspecto. Semelhantemente, *funcionários* só podem associar empregados vistos sob o aspecto *Funcionário*.

Outro tipo de restrição se refere à cardinalidade do mapeamento. Com relação à cardinalidade máxima, a padronização do mapeamento já dá alguma informação. Se a cardinalidade máxima for maior do que 1 (em geral representada por N), o mapeamento se dará para algum tipo de coleção. A especificação de uma cardinalidade máxima finita maior que 1 é realizada como uma restrição de agregação à esta coleção.

O padrão do mapeamento *empregados* de *Departamento*, na Figura 35, é restrito a uma coleção, representada pelo ícone quadrado duplo. Portanto, não há restrição quanto a cardinalidade máxima deste mapeamento. Já o quadrado simples do mapeamento *departamento* de *Empregado* indica que este atributo não se refere a uma coleção, e associa um empregado a um único outro objeto. Portanto, a cardinalidade máxima deste mapeamento é 1.

A indicação de cardinalidade mínima igual a zero significa que o mapeamento pode levar a um elemento nulo ou a uma coleção vazia. O **elemento nulo** é um objeto que indica a ausência de associação. A cardinalidade mínima deve ser encarada como a situação *default*, ou melhor, irrestrita (sem restrição). A proibição de elemento nulo em um mapeamento deve ser explícita. A proibição de coleções vazias pode ser alcançada através de restrições de agregação à coleção.

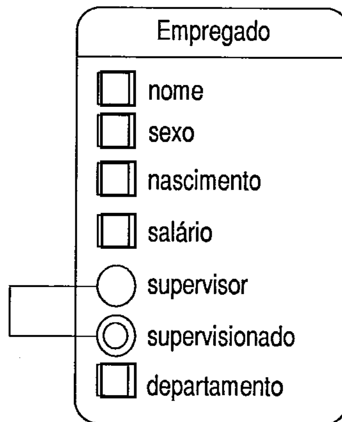


Figura 40: Auto-relacionamento

A Figura 40 mostra a representação, em Diagrama de Objetos, de um auto-relacionamento. Cada empregado tem um atributo *supervisor* que o associa a um outro empregado que o supervisiona. Empregados também possuem um atributo *supervisionado* que o associa aos empregados que ele supervisiona. O círculo duplo representa que este atributo está associado a uma coleção possivelmente vazia, portanto, um empregado pode não supervisionar. Um empregado pode também não possuir supervisor, o que é indicado pelo círculo simples do atributo *supervisor*. Os dois mapeamentos, *supervisor* e *supervisionado*, da ligação *Empregado-Empregado*, portanto, têm cardinalidade mínima igual a zero.

Os mesmos mecanismos para definição de restrições de cardinalidade em mapeamentos podem ser usados em qualquer outro tipo de associação, como atributos e componentes. Estes mecanismos são: a padronização como coleção ou não; restrições de cardinalidade realizadas como restrições de agregação em coleções; e a indicação de proibição de elemento nulo.

4.4.3.9. Restrições em Serviços

Também pode-se querer especificar restrições para os serviços de um objeto. A tipificação dos parâmetros dos métodos de serviço é uma maneira de restringir a ação do serviço.

A especificação de precondições e poscondições a serem verificadas, respectivamente, antes e após a execução do serviço, são outros tipos de restrições empregáveis [BROD81]. Um serviço só é realizado se as suas precondições forem satisfeitas. Se, após a sua realização, as suas poscondições não forem satisfeitas, procedimentos corretivos serão realizados, podendo até desfazer o

trabalho do serviço. Estes procedimentos corretivos são chamados de **tratamento de exceção** e serão discutidos a seguir. As precondições e poscondições podem ser especificadas do mesmo modo que outras restrições.

4.4.3.10. Tratamento de Exceções

A tentativa de violação de uma restrição qualquer, não apenas as pre- e poscondições de serviços, são chamadas de **ocorrência de exceções.** A ocorrência de uma exceção provoca o término antecipado do método em execução e dos métodos chamadores, um a um, até que um método que trate a exceção seja encontrado. Este método é retomado, não no ponto onde estava suspenso, mas na sua rotina de tratamento da exceção ocorrida.

Uma restrição tem sempre um identificador associado que determina a exceção. A princípio, existe um pequeno conjunto de identificadores de exceção predefinidos. Todavia, um método também pode provocar a ocorrência de novos tipo de exceção.

Um método pode definir um **tratamento de exceção.** Este é uma expressão que será executada caso uma dada exceção ocorra dentro do método. Caso ocorra uma exceção que não seja tratada por este método, o método que enviou a mensagem que ativou este poderá tratar a exceção, ou, sucessivamente, deixar para o método que ativou o método, etc. Esta cascata termina no ambiente de execução onde é definido um tratamento default para todos os tipos de exceções.

Observe-se que o tratamento de uma exceção dentro de um método pode provocar uma nova exceção, criando assim um mecanismo de especialização de exceções. Por exemplo, um método pode interceptar a tentativa de violação da cardinalidade máxima de uma coleção e redefinir esta exceção para algo como "turma completa".

O seguinte código DEMO (26)

```
Curso sentença matricula (aluno dado) {
            aluno ⊕= aluno dado
ocorrendo VIOLAÇÃO_DE_RESTRIÇÃO {
            causa exceção TURMA_COMPLETA }
}
```

define um método de nome *matricula* para o tipo *Curso*. Este método exige um argumento, denominado *aluno dado*. Observe-se que o nome do argumento é composto por mais de uma palavra. Nomes de atributos, variáveis e, no nível de implementação, campos, também podem ser assim. A palavra *aluno* será classificada como sendo um identificador de variável, caso já não o seja. Se esta palavra estiver sendo utilizada de outro modo, o argumento não poderá ter o nome começado por *aluno*. A palavra *dado* será um complemento nominal, a não ser que já o seja.

O bloco que representa o corpo do método possui duas partes: seus comandos e seus tratamentos de exceção. Só há um único comando no bloco: o que acrescenta o objeto fornecido como argumento à coleção do atributo *aluno* do curso receptor da mensagem. A análise do código DEMO, quando encontra um nome de variável, verifica se o(s) próximo(s) símbolo(s) são complementos nominais. Em caso positivo, estes são agregados ao nome da variável. Desse modo é feita a distinção dos dois usos da palavra *aluno*.

O bloco define um tratamento para a exceção *VIOLAÇÃO_DE_RESTRIÇÃO* que ocorre quando alguma restrição é violada. Este seria o caso da coleção de alunos do curso ter a sua cardinalidade máxima ultrapassada. O tratamento definido no método do exemplo simplesmente causa uma outra exceção, que irá cancelar os métodos suspensos, até encontrar um que trate esta exceção.

4.4.3.11. Valores Iniciais

O nível conceitual do modelo de objetos permite a especificação de **valores iniciais** em associações. Qualquer criação de objetos de um dado tipo atribuirá convenientemente os valores iniciais definidos para esse tipo. Posteriores atribuições, podem ignorar o valor inicial e definir um novo valor para o atributo.

4.4.3.12. *Valores Compartilhados*

O modelo de objetos do GEOTABA define uma herança de valores pela hierarquia de composição de objetos. Isto pode ser exemplificado da seguinte maneira: imagine que objetos do tipo *Carro* possuam um atributo *cor*. Vários componentes de um carro (capô, capota, portas, etc.) também possuem um atributo *cor*. Ao ser definido um valor para a cor do carro se estará definindo também para cada cor de seus componentes, a não ser que haja uma definição explícita diferente no próprio componente. Valores herdados por componentes são chamados de **valores compartilhados**. Estes diferem dos valores iniciais porque a qualquer momento uma atribuição pode alterar um valor compartilhado e causar a herança deste pela hierarquia de componentes.

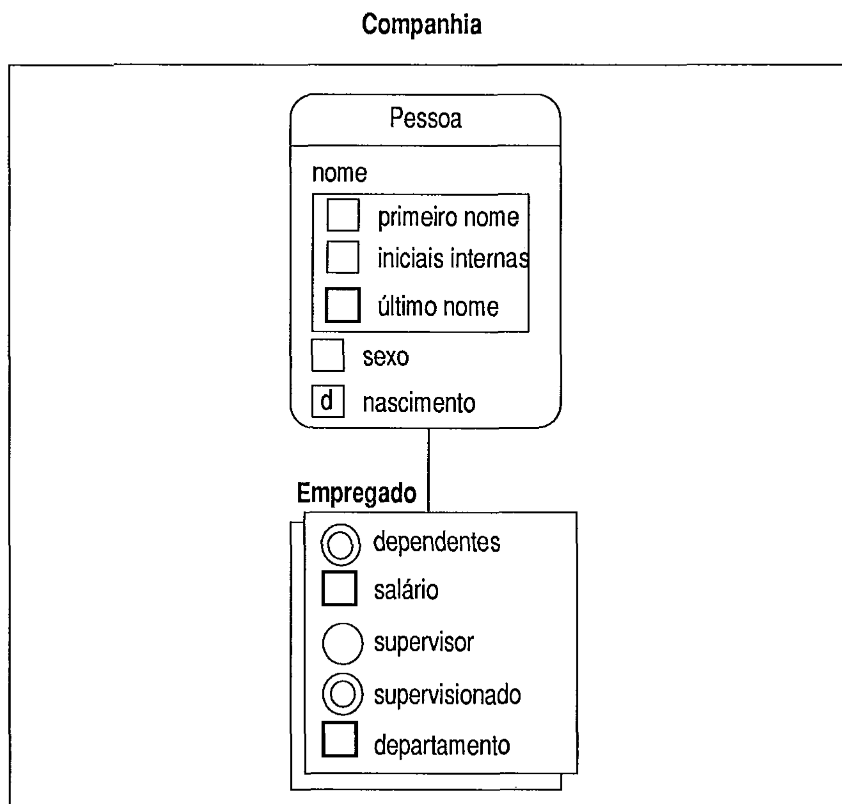


Figura 41: Tipos Implícitos

4.4.4. Tipo Implícito

Há duas maneiras de se definir um tipo: (1) associando-se um nome a um padrão tipo e (2) associando-se um atributo a um padrão tipo. Na primeira maneira, o nome passará a representar o tipo: onde este nome for usado será entendido como sendo uma referência ao tipo a ele associado. No segundo

modo, o atributo fica restrito ao padrão tipo, mas este não possui um nome próprio. Além disso, o domínio do tipo é determinado pelos objetos efetivamente associados ao atributo. O caso do atributo representar uma coleção é o mais comum. Apenas os elementos desta coleção fazem parte do tipo. A criação de um objeto deste tipo automaticamente incluiria-o na coleção. O nome do atributo pode ser usado como se fosse o nome do tipo. Um padrão restrito a este tipo não permite objetos que não estejam associados ao atributo.

A Figura 41 apresenta um Diagrama de Objetos que possui exemplos de tipos definidos implicitamente. No tipo *Pessoa*, o atributo *nome* passou a associar objetos de um tipo que não possui nome, sendo referível apenas por "tipo do atributo *nome*" ou, quando isto não causar confusões, por "tipo *nome*". O tipo dos objetos associáveis a *nome* é descrito dentro da caixa retangular do atributo, que é uma ampliação do quadrado simples usado em atributos que não são coleção. O diagrama mostra que os objetos associáveis a *nome* possuem os atributos: *primeiro nome, iniciais internas* e *último nome.* O segundo exemplo de tipo implícito se refere ao que anteriormente era apresentado como tipo *Empregado.* Nesta figura, *Empregado* é um atributo da base de objetos *Companhia.* Este atributo corresponde a uma coleção, o que é descrito pelo formato de sua caixa retangular dupla, ampliação do quadrado duplo usado para atributos coleção. O tipo dos objetos associáveis a *Empregado* não possui nome próprio, podendo ser referido como "tipo dos elementos de *Empregado*" ou "tipo *Empregado*". Este tipo é um subtipo de *Pessoa* e, além do que ele herda, possui os atributos *dependentes, salário, supervisor, supervisionado* e *departamento.* É importante observar que nunca haverá objetos do tipo *Empregado* que não pertençam à coleção *Empregado.*

4.4.5. Tipos Predefinidos

O GEOTABA inclui um conjunto de tipos de objetos para fornecer uma funcionalidade padronizada (ver 3.2.2.1.15.:"Construtores"). A seguir são apresentados alguns tipos predefinidos dentre os mais importantes.

4.4.5.1. Objeto

O protocolo comum a todos os objetos do sistema é definido no padrão tipo *Objeto.* Todos os objetos respondem que são do tipo *Objeto.* Todo objeto sabe responder: se é ou não de um tipo passado como argumento; se está em conformidade com um padrão; se possui um dado aspecto; se responde a mensagens com a assinatura passada como argumento.

Tanto a equivalência, quanto a igualdade de objetos, podem ser testadas. A **equivalência** testa se dois objetos são o mesmo, isto é, se têm a mesma identidade. A **igualdade** se baseia nos valores de suas associações e é dependente da implementação de cada objeto. É esperado, entretanto, que dois objetos equivalentes sejam sempre considerados iguais, ou seja,

(3) $w1 \equiv w2 \Rightarrow w1 = w2$.

Objeto é um pseudotipo, ou seja, sempre os objetos são de algum subtipo de *Objeto*. Isto significa que *Objeto* define um protocolo padrão para todos os objetos do sistema, mas que a implementação de cada objeto redefine a implementação *default* fornecida. Para se conhecer melhor o significado das operações, é conveniente conhecer algumas definições relativas ao nível de implementação do modelo.

Nos Diagramas de Objetos, o tipo *default* de um atributo é sempre *Objeto*. Isto significa que, não havendo outra indicação em contrário, qualquer objeto pode ser atribuído ao atributo.

4.4.5.2. Condição

O tipo *Condição* é restrito a dois objetos, designados pelos literais *sim* e *não*. A avaliação de uma condição sempre resulta em um destes dois objetos.

O atributo *levantada*, na definição do padrão *Tartaruga*, na Figura 18, é restrito ao tipo *Condição*. Isto é especificado através do símbolo ☑. Os dois únicos valores válidos a esse atributo são *sim* e *não*.

4.4.5.3. Grandeza

O pseudotipo *Grandeza* fornece o protocolo para objetos que podem ser comparados ao longo de uma dimensão linear. Este é o caso de números, datas e mesmo caracteres. As operações *maior que*, *maior ou igual a*, *menor que* e *menor ou igual a* são válidas para qualquer objeto do tipo grandeza. Outras operações adicionais também podem ser disponíveis, como a verificação da inclusão em um intervalo e qual a menor ou a maior entre duas grandezas.

4.4.5.4. Data

Data é um subtipo de *Grandeza*. Não é um pseudotipo, pois novas datas podem ser criadas, fornecendo-se como argumentos o dia, o número do mês e o

ano. A mensagem *hoje* responde o dia em que a mensagem foi enviada. Uma aritmética limitada é fornecida, com subtração de datas e soma e subtração de uma quantidade de dias a uma data. Uma data também sabe informar o seu dia da semana (domingo, segunda-feira, etc.) e o nome do seu mês por extenso (janeiro, fevereiro, etc.).

O atributo *nascimento*, na definição do tipo *Pessoa*, na Figura 41, é restrito ao tipo *Data*. Isto é especificado pela colocação de um *d* dentro da caixa do atributo.

4.4.5.5. Instante

Instante também é um subtipo de *Grandeza*, muito semelhante a *Data*. Um instante pode ser criado fornecendo-se uma hora, os minutos e os segundos (opcionais). A mensagem *agora* responde o instante em que a mensagem foi enviada. A subtração de instantes fornece a quantidade de segundos que os separa. Um instante pode ser somado ou subtraído de segundos.

4.4.5.6. Caracter

Caracter é subtipo de *Grandeza* por manter uma ordem entre os caracteres e, consequentemente, incluir o seu protocolo. Todavia, novos caracteres, além dos predefinidos, não podem ser criados. Os caracteres podem ser referenciados por literais apropriados, definidos por cada linguagem que adote este modelo de objetos. Os caracteres são objetos imutáveis, ou seja, seus estados nunca são alterados. O protocolo de *Caracter* possui mensagens para responder as funções usuais sobre se o caracter é um dígito, ou letra, maiúscula, minúscula, etc. Caracteres não são usados em operações aritméticas, como a soma e a subtração.

4.4.5.7. Número

Número é um subtipo de *Grandeza* que possui diversas implementações de classe correspondentes: *Reais*, *Racionais* e *Inteiros*. Todavia, no nível conceitual, são muitas as situações em que não se precisa conhecer se um dado objeto numérico é um número inteiro, racional ou real. Neste caso, usa-se o tipo *Número*. Assim como caracteres, números são imutáveis, predefinidos e referenciados por literais apropriados. Todos os números, independentemente de sua implementação, têm um protocolo que inclui as operações aritméticas usuais (soma, subtração, multiplicação, divisão, divisão inteira, resto, valor absoluto,

negação e inversão), funções exponenciais, logarítmicas, trigonométricas, de truncamento e arredondamento.

4.4.5.8. Inteiro

O tipo *Inteiro* deve ser usado quando se necessita de alguma ação definida no protocolo adicional que ele acrescenta ao do seu supertipo *Número.* Este protocolo adicional inclui a representação do número em uma base não decimal, a correspondência com o caracter que possui o número como código, a repetição da execução de um bloco de comandos sucessivamente pelo número de vezes especificado, as funções de fatorial, mmc e mdc, e as operações do número como máscara de bits (ou, e, ou exclusivo, etc.). Um literal numérico define, na sua própria representação, se o número é do tipo inteiro ou não.

O atributo *idade*, na definição do padrão *Pessoa*, na Figura 15, é restrito ao tipo *Inteiro*. Isto é especificado pela colocação de um *i* dentro da caixa do atributo.

4.4.5.9. Coleção

Uma coleção pode ser encarada como sendo um grupo de objetos, que são os valores das associações presentes na coleção. Estes objetos são chamados de **elementos** da coleção. Toda coleção é do tipo *Coleção*. Por *default*, uma coleção não impõe uma ordem aos seus elementos. Subtipos de *Coleção* podem impor ordenação de seus elementos. Uma coleção, também por *default*, permite que haja elementos repetidos. Alguns subtipos de *Coleção* podem proibir elementos duplicados na coleção. Por *default,* também, as associações de uma coleção são sem chave. Mais comumente são empregados subtipos de *Coleção* cujas associações têm chave.

A menos que restrições específicas o proíbam, pode-se acrescentar ou remover elementos em uma coleção. Assim, o número de associações que a coleção possui variará. Neste caso, a coleção é uma **coleção variável**; no caso contrário, é uma **coleção fixa**. Ações para acrescentar ou remover elementos podem ser mensagens síncronas ou eventos assícronos. Uma mensagem de **acréscimo** adiciona um elemento a uma coleção e a **anexação** adiciona cada elemento de uma dada coleção à coleção original, que passa a ser equivalente à união das duas coleções. A **remoção** também pode ser de uma ou mais ocorrências de um elemento, ou de todos elementos presentes em uma outra coleção. A **depuração** remove os elementos que não pertencem a outra coleção. A remoção de um objeto não contido na coleção, provoca a ocorrência de uma

exceção. As operações acima retornam como resultado a própria coleção alterada. Uma outra alternativa não altera a coleção receptora da mensagem, mas responde uma outra coleção com o conteúdo da primeira acrescido ou decrescido. Estas operações são a **união**, a **diferença** e a **interseção.** A união e a interseção não são comutativas a não ser quando aplicadas a coleções de mesmo tipo. Caso contrário, o tipo do resultado dependerá do tipo da coleção receptora da mensagem.

Considerando-se que *a* e *b* representam variáveis do tipo *Coleção*, e as seguintes expressões DEMO (27)

```
a ⊕= 145
a ⊕ 145
a += b
a + b
a ⊖= 145
a ⊖ 145
a -= b
a - b
a ∩= b
a ∩ b
```

tem-se que, a primeira acrescenta a *a* o inteiro *145*; o valor resultante da expressão será o novo valor de *a.* A segunda dá o mesmo resultado, porém a coleção *a* permanece inalterada. Na terceira, a coleção *b* é anexada a *a*, isto é, cada valor de *b* é acrescentado a *a*; o novo valor de *a* será o resultado da expressão. A quarta expressão é a união entre *a* e *b* e dá o mesmo resultado que a terceira, porém sem alterar *a.* A quinta remove o inteiro *145* de *a*, respondendo o novo valor de *a.* A sexta expressão, sem alterar *a*, responde como ele ficaria com o *145* removido. A sétima e a oitava respondem como *a* ficaria se todos os elementos de *b* fossem removidos de *a*, sendo que a sétima altera *a* e a oitava, que calcula a diferença entre *a* e *b*, não. A nona e a décima expressões respondem como *a* ficaria se todos os seus elementos que não pertencessem a *b* fossem removidos. Novamente, apenas quando o operador termina com =, a coleção é alterada.

O padrão predefinido *Coleção Fixa* é restrito a objetos coleções, mas não define, no seu protocolo, operações de remoção ou acréscimo de elementos. Isto significa que um atributo restrito a este padrão não permite tais operações.

Uma coleção sem associações com chave só permite consultas sobre se possui um dado elemento, sobre quantos elementos possui, sobre o número de ocorrências de um elemento e se a coleção está vazia.

As operações acima são exemplificadas pelas seguintes expressões (28)

```
a ∋ 145
a possui quantos 145
tamanho de a
a vazia
```

A primeira responde *sim* ou *não*, de acordo com *145* pertencer ou não a *a*. A segunda conta o número de vezes *145* o inteiro aparece em *a*. A terceira mostra os atributos *tamanho* e *vazia*, que toda coleção possui.

Observe-se que na segunda expressão, a mensagem se utiliza de palavras separadas (*possui* e *quantos*). De modo semelhante aos nomes de variáveis, *possui* é classificada como um *identificador de sentença* (procedimento) e *quantos* como complemento verbal. Durante a análise de uma expressão, ao se encontrar um identificador de sentença, deve-se coletar os eventuais complementos verbais que se seguirem imediatamente, para formar o nome da sentença.

Muito importante são as operações de **enumeração** da coleção. Estas executam alguma determinada ação para cada elemento da coleção. As operações de enumeração são: iteração, seleção, rejeição, coleta, deteção e injeção. A operação básica, a **iteração**, é equivalente ao *do* ou *for* das linguagens de programação executa um bloco de comandos para cada elemento da coleção. As operações de seleção, rejeição e coleta resultam na criação de outra coleção de tipo semelhante ao da original. A **seleção** reúne na coleção resultado os elementos da coleção original que atendem a um determinado critério passado como argumento. A **rejeição** faz o oposto: reúne os elementos que não atendem ao critério passado como argumento.

Segue-se um exemplo de iteração (29)

```
digs:=0; cad faz { é dígito? {digs := digs+1} }
```

No primeiro comando da linha, a variável *digs*, recebe o valor zero, para iniciar a contagem de dígitos existentes na cadeia de caracteres *cad.* No outro comando da linha, é realizada uma iteração sobre *cad.* O bloco seguinte à palavra *faz* é executado para cada elemento da coleção (*cad*) que antecede ao operador. O objeto corrente do bloco é o elemento corrente da coleção. A mensagem *é dígito*, então, será enviada ao elemento corrente da cadeia e responderá se o caracter é um dígito ou não. O operador *?* equivale ao *if-then* de outras linguagens; se o que o precede for verdadeiro, o bloco que o segue será executado.

O exemplo poderia ser reescrito do seguinte modo (30)

```
digs := 0
cad faz { λcarac; carac é dígito? { digs:=digs + 1 } }
```

O operador λ cria uma variável, se ainda não houver, de escopo restrito ao bloco que o contém, com o nome que o segue e com valor inicial igual ao seu receptor. Neste caso, o receptor de λ é o objeto corrente do bloco, ou seja, a cada vez, um caracter de *cad* será atribuído à variável *carac.* Assim pode-se definir um nome para o objeto corrente de um bloco.

A seleção é exemplificada a seguir (31)

```
Empregado | { salário >= 10_000 }
```

O operador de seleção (/) avalia o bloco que o segue, para cada elemento do seu receptor, no caso, a coleção referida pelo nome *Empregado.* O objeto corrente do bloco será o elemento corrente da coleção. A seleção cria uma nova coleção, de características semelhantes à coleção receptora, contendo apenas os elementos do receptor que fazem o bloco ser avaliado como verdadeiro. Neste exemplo, cada elemento da coleção *Empregado* terá o seu salário consultado e verificado se é maior ou igual a 10.000. Apenas os empregados com salário acima desse valor estarão selecionados no resultado. Uma expressão equivalente poderia ser escrita usando rejeição no lugar de seleção (32)

```
Empregado ⊥ { salário < 10_000 }
```

A **coleta** cria uma nova coleção cujos elementos foram gerados um a um a partir da avaliação, para cada elemento da coleção original, de uma função. Uma das utilizações da coleta é a realização de projeções: cada elemento da coleção original gera um elemento da coleção resultante com menos atributos ou

com uma vista mais restrita. A junção esporádica de objetos também pode ser realizada pela coleta.

A expressão abaixo (33)

```
Empregado : { salário }
```

reúne os salários dos elementos de *Empregado* em uma nova coleção. Cada elemento dessa coleção foi obtido como resultado da avaliação do bloco para cada elemento de *Empregado*. O bloco do exemplo apenas responde o valor do atributo salário do objeto corrente do bloco, que será, a cada vez, um elemento de *Empregado*. A seguinte (34)

```
Empregado : { '«nome; salário» }
```

cria uma coleção de tuplas com dois atributos: *nome* e *salário*. Cada tupla terá, nesses atributos, o nome e o salário de um empregado. A expressão *«nome; salário»* é uma abreviatura de *‹$nome; $salário›*. Ou seja, os caracteres « », do conjunto ANSI, e que não devem ser confundidos com ‹ ›, nem com < >, permitem a especificação de um vetor de signos. O operador ' faz a projeção de <u>um</u> objeto: cria uma tupla, a partir do receptor, com os atributos designados no vetor de signos que recebe como argumento. Portanto, o exemplo acima mostra uma forma simplificada de se obter o mesmo resultado que a expressão (35)

```
Empregado : { ‹nome ⇒ nome; salário ⇒ salário› }
```

Já (36)

```
Empregado : { ‹  ⇐ nome
                 nomeDep ⇒ nome de departamento› }
```

coleciona tuplas, uma para cada empregado, com o nome do empregado e o nome de seu departamento. Isto é obtido consultando o atributo *departamento* do empregado e o atributo *nome* deste departamento. A expressão ⇐ *nome* serve como abreviatura de *nome* ⇒ *nome*. O operador ⇐ funciona como um construtor de atributo, cujo nome o segue e cujo valor é dado pela avaliação da variável ou atributo pré-existente, com esse nome.

A **deteção** é outra operação de enumeração de coleções. Seu objetivo é responder se algum elemento da coleção atende a um dado critério; se nenhum atender, uma exceção ocorrerá. A **injeção** calcula um resultado a partir da acumulação dos resultados parciais de uma função passada como argumento. A injeção serve para resumir informações a partir da coleção original. Funções agregadoras, como somatório, contadores, máximos e mínimos, são realizadas

por injeção. Todos os elementos da coleção são percorridos e, para cada um, uma função é aplicada relacionando aquele elemento ao resultado da função obtido no elemento anterior.

A seguinte deteção (37)

```
Empregado ¿ { salário > 20_000 }
```

responde *sim* caso exista algum empregado que avalie o bloco como verdadeiro. Isto ocorrerá se houver algum elemento de *Empregado* cujo atributo *salário* seja maior do que 20000. Já a injeção (38)

```
cad injeta 0 em { λ«digs; carac»
               digs + ( (carac é dígito) então{1} senão{0} }
```

conta os dígitos da cadeia *cad*. O bloco será executado uma vez para cada elemento da coleção. A cada vez, o objeto corrente do bloco será um vetor contendo o elemento corrente da coleção e o valor calculado na última iteração, exceto na primeira, quando o valor inicial, contido no primeiro parâmetro, ou seja, após a palavra *injeta*, serve como último valor calculado. O operador λ, quando aplicado a dois vetores de mesma cardinalidade, cria uma variável local para cada elemento do vetor argumento, usando o respectivo elemento do vetor receptor como valor inicial. Portanto, a variável *carac*, será do tipo *caracter* e conterá o elemento corrente de *cad* e *digs* será uma variável inteira contendo o valor acumulado. A cada iteração, o valor resultante será a soma do valor previamente calculado (*digs*) com zero ou um, dependendo se o caracter é um dígito. A fórmula *então{}senão{}* é uma mensagem enviada a um objeto condição, com dois blocos como argumentos. De acordo com a condição, o primeiro, ou o segundo bloco, será avaliado. O valor do segundo bloco tem que ter conformidade com o tipo do valor do primeiro bloco. O resultado dessa mensagem sempre estará em conformidade com o tipo do valor do primeiro bloco. No exemplo, será um número inteiro.

Semelhantemente, o exemplo seguinte (39)

```
Empregado injeta 0 em { λ«emp; tot»
               salário de emp + tot }
```

calcula o somatório dos salários dos elementos da coleção *Empregado*.

O protocolo do tipo *Coleção* vale para qualquer coleção definida por um subtipo de *Coleção*. Portanto, as operações acima são aplicáveis não só a coleções que permitem repetições, sem ordenamento dos elementos e sem chave, mas a qualquer coleção. Além das operações definidas acima, toda coleção deve

permitir a conversão de tipo. A conversão de tipos de coleção equivale à criação de uma coleção do tipo destino contendo todos os elementos da coleção original.

A seguir são listados outros tipos de coleções predefinidos, usáveis como supertipos de novos tipos de coleções. Coleções, por *default*, conforme definido no tipo *Coleção,* não são ordenadas, nem acessíveis por chaves e admitem elementos repetidos. Correspondem aos *bags* de algumas linguagens de programação. O tipo *Conjunto* inclui as coleções onde não há elementos repetidos. O tipo *Dicionário* contém conjuntos cujos elementos podem ser acessados por chaves primárias.

4.4.5.10. Conjunto

Uma coleção pode aceitar a existência de elementos repetidos ou não. Em um conjunto, elementos duplicados, isto é, iguais, são considerados idênticos e apenas um deles é mantido na coleção. O tipo *Conjunto* é um subtipo de *Coleção* que não acrescenta novas ações ao protocolo deste, mas apenas a restrição que impede a existência de elementos repetidos na coleção. A operação de acréscimo de elemento a um conjunto só realmente acrescenta este elemento se o conjunto não o possuir. O padrão predefinido *Conjunto Fixo* não possui o-perações de acréscimo ou remoção de elementos no seu protocolo. Isto significa que um atributo submetido a este padrão não permite acréscimo e remoção de e-lementos.

4.4.5.11. Dicionário

Uma coleção que permite a inclusão de associações com chave é chamada de **dicionário**. Nesta, cada elemento da coleção pode ser acessado em particular, por mensagens de consulta e de atribuição, utilizando a chave da associação cujo valor é o elemento. Se não houver associação com a chave fornecida, uma exceção ocorrerá. A atribuição altera o valor da associação cuja chave foi fornecida ou cria uma nova associação, caso não haja alguma com esta chave.

O código seguinte (40) cria uma variável *antônimo* contendo um dicionário de signos e indexado por signos.

```
variável antônimo: dicionário[signo]=signo
antônimo[$quente] := $frio
antônimo[$frente] := $trás
```

A segunda e a terceira linha exemplificam atribuições a elementos desse dicionário. Como anteriormente não havia elementos de chave *$quente* e *$frente*, essas atribuições implicam na criação de associações e não apenas atribuem um novo valor a alguma associação. No trecho seguinte, (41) a variável *nom* receberá o signo *$frio*, resultado da consulta ao dicionário usando a chave *$quente*.

```
variável nom: signo
nom := antônimo[$quente]
```

As operações comuns a qualquer coleção fixa se aplicam aos valores das associações do dicionário. Por exemplo, a enumeração dos elementos de um dicionário (iteração, seleção, coleta, etc.) é realizada sobre os valores das associações, ignorando as chaves. Um dicionário encarado como uma coleção de valores considera que seus elementos não estão ordenados, porém possuem chaves explicitamente designadas. Um dicionário não possui operações de acréscimo e remoção de valores independentemente de suas chaves. Um dicionário admite uma mensagem que cria uma outra coleção composta só pelos seus valores.

As expressões (42)

```
antônimo : { λa; a }
valor de antônimo
```

são equivalentes. A primeira realiza uma coleta a partir do dicionário *antônimo*. A cada iteração, a variável *a* receberá o valor de uma associação do dicionário e o bloco responderá esse valor, que será coletado na coleção que será o resultado da coleta. A segunda expressão mostra que esta operação poderia ser realizada consultando-se o atributo inalterável *valor* que todo dicionário possui. O método que implementa essa consulta realizará a coleta dos valores do dicionário.

Um dicionário também pode ser visto como sendo um conjunto de objetos associações. Para tal, o tipo *Dicionário* admite uma vista (ver 4.6.1.:"Vistas") que inclui operações, semelhantes às operações comuns sobre

coleções, mas que são aplicadas às associações. As operações envolvendo associações contêm: a referência de associação (**indexação**), onde, dada uma chave, se obtém a associação correspondente; a pertinência de associação, que verifica se uma dada associação faz parte do dicionário; e a remoção de associação. Outras operações considerando um dicionário como um conjunto de associações implementam a enumeração de suas associações.

Considerando-se que *outrasPalavras* é uma coleção de signos, as expressões (43)

```
(associação de antônimo)[$quente]
associação de antônimo ⊕= ( claro ⇒ $escuro )
associação de antônimo ∋  ( claro ⇒ $escuro )
associação de antônimo ⊖= ( claro ⇒ $escuro )
associação de antônimo faz { outrasPalavras ⊖= chave }
```

mostram como o dicionário *antônimo* pode ser manipulado como sendo uma coleção de associações. A primeira expressão faz uma consulta a *antônimo* visto como uma coleção de associações. Seu resultado será equivalente a (44)

```
quente ⇒ $frio
```

A segunda expressão acrescenta uma associação ao dicionário e é equivalente a (45)

```
antônimo[$claro] := $escuro
```

A terceira verifica se a associação dada pertence ao dicionário. A quarta remove esta associação. A quinta expressão executa, percorrendo o dicionário, o bloco fornecido. A cada vez, o objeto corrente será uma das associações do dicionário. A expressão *chave* responde o atributo chave da associação corrente. O bloco, portanto, removerá da coleção *outrasPalavras* todas as chaves presentes em *antônimo*.

Outras operações de dicionário se referem à manipulação das chaves. Neste caso está a **derreferência**, onde, dado um valor, é respondida uma chave associada a este valor. Esta chave pode não ser a única associada ao valor. Outra operação responde um conjunto contendo apenas as chaves do dicionário. A pertinência de chave permite testar se a coleção possui uma associação com a chave dada. A remoção por chave remove a associação da coleção que tem uma determinada chave. A iteração por chave executa mensagens para cada chave da coleção.

Nas expressões abaixo (46)

```
(chave de antônimo)[$frio]
chave de antônimo ∋ $claro
chave de antônimo ⊖= $quente
chave de antônimo faz { λch; outrasPalavras ⊖= ch }
conjunto de chave de antônimo
```

a expressão *chave de antônimo* resulta em uma vista do dicionário *antônimo* onde ele pode ser manipulado como uma coleção de chaves. A indexação, exemplificada na primeira linha, corresponde a derreferência, isto é, responde uma chave, dado um valor. A segunda linha verifica se o dicionário contém uma dada chave, ou melhor, uma associação com esta chave. A terceira linha remove a associação que tem a chave dada. A quarta linha equivale a (47)

```
outrasPalavras -= chave de antônimo
```

A quinta linha cria um conjunto composto pelas chaves do dicionário, evocando o atributo *conjunto* da vista *chave de antônimo.* Note-se que um atributo e qualquer tipo de variável podem ter o mesmo nome que um tipo ou classe. Nos casos onde pode haver confusão (por exemplo, quando se omite o receptor da mensagem de consulta a um atributo), a preferência é a interpretação do nome pelo significado de escopo mais restrito. Como tipos e classes têm nomes globais, eles não teriam a precedência. A qualificação do nome, permitiria eliminar explicitamente a ambiguidade. Nesse caso se diria: *tipo conjunto, classe tabelaHash* ou *variável abc.* Observe-se, porém, que, se esses nomes não existirem com o significado em que foram qualificados, as expressões mencionadas provocarão a criação desses nomes com tais qualidades.

Portanto, um dicionário pode ser visto como uma coleção de valores, como um conjunto de associações ou como um conjunto de chaves. Exemplificando, uma coleção qualquer permite a iteração sobre seus elementos. Em um dicionário esta iteração pode ser feita através de cada valor incluído no dicionário, através de cada chave incluída no dicionário ou, o que é menos comum, através de cada par (chave, valor) incluída. Como uma coleção comum, a iteração se aplicará aos valores do dicionário. Aplicando-se a iteração por chaves, ou a iteração por associações, o dicionário é encarado como sendo um conjunto de associações ou um conjunto de chaves.

4.4.5.12. *Tupla*

Tupla é um subtipo de *Dicionário*, onde as chaves são signos. Tuplas são úteis, por exemplo, para colecionar atributos de outros objetos. Assim pode-se simular a "projeção" de um objeto, selecionando determinados atributos seus, ou a "junção" de dois objetos. O objeto gerado será do tipo *Tupla*, não herdando o tipo dos objetos originais.

A expressão (48)

```
variável t1 ⇒ ‹nome:t ⇒ "Zoraide"; idade ⇒ ›
```

indica que a associação dada (entre o nome *t1* e uma tupla dada) se refere a uma variável. Se esta variável de nome *t1* não existir, ela será criada. Como o padrão da variável (ou seja, o padrão da associação) não foi fornecido (seguindo o seu nome por dois-pontos e a descrição do padrão), mas foi fornecido o valor da associação (após o símbolo ⇒), o padrão da associação que representa a variável *t1* será o tipo induzido do valor da associação. Ou seja, o tipo de *t1* será *tupla*. A tupla definida como valor de *t1* será atribuída a *t1*, quer esta já variável pré-exista, ou não. Se ela já existir com um padrão que tupla não tenha conformidade, um erro, acusado em tempo de compilação, ocorrerá.

Se, no lugar do símbolo ⇒ , fosse usado *:=*, e a variável *t1* tivesse sendo criada, ela seria considerada como restrita ao tipo *Objeto*. Após a criação da variável é que se daria a atribuição.

A tupla usada como valor a ser atribuído a *t1* foi criada com dois atributos. O segundo deles sem um tipo definido e sem um valor inicial. Nesse caso, o valor inicial será o objeto *indefinido* (ver 4.4.7.1.:"Objeto Indefinido"), e o tipo induzido será *Objeto*.

Se *empr* representa um empregado, a expressão (49)

```
empr { ‹ ⇐nome; ⇐fone › }
```

cria uma tupla com os atributos *nome* e *fone* inicializados com os atributos *nome* e *fone* de *empr*, que é o objeto corrente do bloco.

A expressão (50)

```
empr { < ⇐nome; nomeDep ⇒ nome de departamento > }
```

também gera uma tupla com dois atributos: *nome* e *nomeDep*. O valor inicial desse último é o resultado da consulta ao atributo *nome* do departamento contido no atributo *departamento* do objeto corrente do bloco, isto é, o empregado *empr*.

4.4.5.13. Sequência

Uma **sequência** define uma ordem pela qual seus objetos podem ser obtidos um a um. O tipo *Sequência* define o protocolo comum a todas as coleções cujos elementos possuem alguma ordenação. Os elementos de uma sequência sempre podem ser associados a números inteiros representando a sua posição na sequência, ou seja, seu **índice**. *Sequência* é um subtipo de *Coleção*. Como os índices funcionam como chaves restritas a números inteiros, uma sequência aceita qualquer ação válida para dicionários, exceto a atribuição. Portanto, a coleção pode informar qual elemento está em uma dada posição. O padrão predefinido *Sequência Fixa* não tem operações de acréscimo e remoção de elementos.

Como *cadeia de caracter* é um subtipo de *sequência*, as operações sobre dicionários (exceto a atribuição) são válidas em cadeias, como na expressão (51)

```
"Dia de luz"[5]
primeiro de "festa de sol"
último de "ilhas do sul"
```

que consulta a associação da cadeia cuja chave é o inteiro 5. A resposta será equivalente a 'd', que é o quinto caracter da cadeia.

Uma sequência mantém sempre os seus elementos em uma determinada ordem. A operação de **ordenação** justamente define esta ordem. Isto é feito através da especificação de uma função booleana, que quando aplicada a dois objetos responde se eles estão ou não na ordem desejada. A redefinição da ordem de uma coleção ordenada provoca a reordenação de toda a coleção.

No trecho abaixo (52)

```
variável outrasPalavras ⇒ conjunto‹ ›
outrasPalavras { ⊕= $cedo; ⊕= $belo; ⊕= $macio }
variável op ⇒ outrasPalavras ordena { λ«a;b»; a > b }
```

a primeira linha cria uma variável *outrasPalavras* com um conjunto vazio como valor inicial. Na segunda, o conjunto contido nesta variável passa a ser o objeto corrente de um bloco, que contém três mensagens que acrescentam os signos *$cedo*, *$belo* e *$macio* ao conjunto. A terceira introduz uma variável *op* cujo valor inicial será uma sequência criada a partir da ordenação do conjunto *outrasPalavras*. O argumento da mensagem *ordena* é um bloco que avalia a condição em que quaisquer dois elementos estarão ordenados. O objeto corrente do bloco será um vetor com dois elementos da coleção receptora de *ordena*. O comando λ denomina esses elementos de *a* e *b*. Este vetor estará ordenado se a avaliação do bloco for 'verdadeiro', ou seja, se $a > b$. Portanto, *op* terá os elementos de *outrasPalavras*, ordenados decrescentemente. Uma forma de se explicitar a função de ordenação, sem introduzir novas variáveis, indexa o próprio vetor objeto corrente, como em (53)

```
op ordena {[1]<[2]}
```

A operação de indexação pode ser realizada dessa maneira, quando o objeto corrente for algum tipo de sequência, como os vetores.

Como há uma ordem entre os seus elementos, o protocolo de *Sequência* inclui mensagens para responder qual o primeiro elemento da sequência e qual o último. As sequências podem realizar buscas de elemento, de subsequências e, generalizando, de padrões. A **busca** de elemento informa qual a posição em que um dado elemento é encontrado pela primeira vez. A busca pode ser realizada na ordem natural ou na ordem reversa. Também é possível se definir uma posição inicial para a busca.

As operações de alteração podem ser aplicadas à coleção original ou a uma cópia dela, mantendo a coleção original intacta. A remoção pode ser aplicada a sequências ou a padrões. Tem-se também a remoção de um elemento dada a sua posição. A remoção do primeiro e a do último elemento podem ser consideradas operações específicas. A **extração** gera uma cópia de uma subcoleção da coleção original.

Novas operações de enumeração podem ser definidas. A **iteração reversa** executa um conjunto de mensagens para cada um dos elementos da coleção, porém estes elementos são percorridos na ordem reversa. A **busca**

condicional procura o primeiro elemento que atende uma dada condição, ou o último, se a busca for reversa. A **iteração simultânea** permite que mais de uma coleção possa ser percorrida simultaneamente e uma série de mensagens possa ser executada utilizando simultaneamente um elemento de cada coleção.

4.4.5.14. Lista

Uma lista é uma sequência sem função de ordenação explícita, seus elementos são mantidos em ordem pela inserção deles em posições arbitrárias. Estas inserções podem, por exemplo, seguir uma política LIFO ou FIFO. *Lista* é um subtipo de *Sequência*. Em uma lista pode-se fazer uma **inserção** de um elemento, ou de todos os elementos de uma outra coleção, antes ou após uma dada uma posição. A inserção antes do primeiro e após o último elemento da coleção merecem operações especiais, para a rápida implementação de filas e pilhas. Idem para a remoção do primeiro e do último elemento. O padrão predefinido *Lista Fixa* permite restringir a utilização de uma lista, previnindo a remoção e o acréscimo de elementos.

A operação de **preenchimento** substitui todos os elementos da lista por um dado outro elemento. A reversão inverte a ordem dos elementos na coleção. A **concatenação** cria uma nova coleção com os elementos da coleção original seguidos pelos elementos de uma outra dada coleção. A operação de **substituição** realiza uma busca e troca os elementos encontrados na sequência original por outros elementos dados. Uma lista é uma sequência pela ordem de chegada de seus elementos. Quando uma lista recebe uma mensagem definindo uma outra ordem entre seus elementos, ela deixa de ser uma lista e passa a ser apenas uma sequência segundo esta ordem definida.

4.4.5.15. Texto

Texto é um subtipo de *Lista*. Na sua implementação mais simples um texto é uma cadeia de caracteres, mas subtipos de *Texto* implementam textos produzidos por processadores especializados, contendo fontes, estilos de caracteres e parágrafos, etc. As linguagens que adotarem o modelo de objetos do GEOTABA deverão definir literais para representar textos simples.

Um texto simples prevê operações que ignoram diferenças entre minúsculas e maiúsculas, ordenam palavras acentuadas adequadamente e realizam conversões minúsculas/maiúsculas. Pattern-matching com coringas ("wild-characters"), tipo ? e *, são possíveis.

Os atributos *nome* e *fone*, na definição do padrão *ProprietárioDeTelefone*, na Figura 30, são restritos ao tipo *t*. Este tipo se refere a textos que não aceitam serem alterados. Na realidade, *t* é um subpadrão de *Lista* e o padrão tipo *Texto* é subpadrão de *t*. O padrão *t* possui uma restrição de tipo que impõe que todo objeto em conformidade com *t* tem que ser também do tipo *Texto*.

4.4.5.16. Signo

Signos são textos simples (cadeias de caracteres) que o sistema garante serem únicos, isto é, dois signos iguais são, por consequência, idênticos. As linguagens que adotarem o modelo de objetos do GEOTABA deverão definir literais próprios para representar signos. Um signo é comumente usado como chave de algum atributo ou variável, de modo a permitir que o ser humano identifique a associação. Um signo é uma coleção fixa e imutável, ou seja, a sua cadeia de caracteres não pode ser alterada.

Estas propriedades tornam os signos aptos a identificar objetos. Se dois signos distintos pudessem ser iguais, o ser humano encontraria dificuldades para determinar se está usando o símbolo correto ou não. Por exemplo, como distinguir versões diferentes de uma mesma cadeia de caracteres? Se um signo pudesse ser alterado, sempre haveria dúvida em relação a semântica desta alteração: as associações onde aparece este símbolo seriam mantidas, ou não?

4.4.5.17. Progressão

Algumas coleções podem ser definidas dando a sua forma intensional, sendo possível se deduzir a sua extensão. Um exemplo seria a coleção dos inteiros ímpares maiores que 10 e menores que 50. Esta é uma coleção fixa, isto é, não pode ter elementos adicionados ou removidos. Um objeto do tipo *Progressão* é uma sequência de tamanho fixo, representando uma progressão, que não precisa manter a sua extensão. Por exemplo, os múltiplos de 7 maiores que 100 e menores que 200 formam uma progressão.

Uma progressão é caracterizada por um objeto inicial, uma condição de término e um método para computar cada objeto sucessivo.

4.4.5.18. Sucessão

Uma sucessão é uma progressão aritmética: o método para a computação do sucessor da progressão consiste na simples adição de um **incremento** e a condição de término é caracterizada por um valor **limite** (máximo ou mínimo)

para o último número computado. Por exemplo, uma sucessão com valor inicial de 100, incremento de -15 e limite de 1, é composto pela sequência 100, 85, 70, 55, 40, 25 e 10. *Sucessão* é um subtipo de *Progressão*.

4.4.5.19. Intervalo

Um intervalo também é uma progressão: o método para a computação do sucessor da progressão é definido pelo tipo dos objetos da coleção e é acionado através de uma operação que obtém o **sucessor** do objeto. No caso de números inteiros, o cálculo do sucessor consiste em somar 1 ao objeto receptor. A condição de término também é caracterizada por um valor limite. Por exemplo, um intervalo iniciando com o inteiro 10 e limite de 15 é composto pela sequência 10, 11, 12, 13, 14 e 15. *Intervalo* também é um subtipo de *Progressão*. Tipos de objetos que possuem a operação para cálculo do sucessor e do predecessor de um objeto são chamados de *tipos discretos*. Um intervalo é uma progressão de objetos de um tipo discreto.

A expressão seguinte (54)

```
6 até índice final de vet faz { λi; vet[i] := 0 }
```

zera os elementos de *vet* a partir do índice 6. A mensagem *até* aplicada a objetos discretos gera um intervalo. O método *faz* executa o bloco para cada elemento do intervalo. No próximo exemplo (55)

```
índice inicial de vet até índice final de vet passo 2 faz{
              λi; vet[i] := 0 }
```

a iteração ocorre sobre uma sucessão e zerará o primeiro elemento de *vet*, o terceiro, o quinto etc. A seguinte expressão (56)

```
3 até 9999 usando {λi; i * i }
```

representa uma progressão com os inteiros 3, 9, 81 e 6561.

4.4.5.20. Vetor

Vetor é um subtipo de Sequência. Um vetor representa uma coleção de elementos, acessíveis por chaves externas restritas a um intervalo de objetos. Estes objetos funcionam como índices da sequência. Um vetor é uma coleção fixa, ou seja, não pode aumentar ou diminuir.

4.4.6. Acessos

O modelo de objetos deve prever a incorporação dinâmica de caminhos de acessos pragmáticos às coleções. Dois tipos de caminhos de acesso são descritos a seguir: os índices e os cursores.

4.4.6.1. Índices

Um **índice** é um mecanismo de acesso eficiente a elementos de uma coleção que atendem a determinadas condições. Normalmente essas condições se referem a valores de atributos. Algumas coleções podem, dinamicamente, adquirir diversos índices. A utilização da coleção não sofre qualquer impacto com isto.

Mesmo sobre uma coleção sem chaves podem ser criados índices. Alguns sistemas de banco de dados não fornecem o conceito de chaves, tendo estas que serem simuladas pela criação adequadas de índices. Esta não parece ser a atitude mais adequada para determinadas coleções que, no mundo real, naturalmente devam ter chaves.

4.4.6.2. Cursores

Um **cursor** permite que se mantenha uma referência a uma posição em uma coleção de objetos. Esta referência é chamada de a **posição corrente** do cursor na coleção. Com isto, é possível percorrer os elementos da coleção, um a um, de um modo menos estruturado que o das operações de enumeração descritas anteriormente. Em um dado instante pode haver qualquer número de cursores atuando sobre uma mesma coleção, cada um com a sua posição corrente própria.

Um cursor pode ser criado para a leitura de uma coleção, para a escrita na coleção e para ambas operações. A operação de leitura responde o elemento correspondente à posição corrente do cursor e incrementa esta posição. Variações de leitura existem para: (1) responder uma subcoleção com os próximos *n* elementos; (2) responder uma cópia de toda a coleção; (3) realizar a leitura do próximo elemento conferindo se ele é igual a um dado objeto.

A escrita armazena um elemento na posição corrente da coleção e incrementa esta posição. Variações da escrita permitem armazenar todos os elementos de uma outra dada coleção (cópia) e armazenar um dado objeto em um dado número de posições (preenchimento).

Uma operação deve indicar se o cursor está posicionado ao final da coleção; outra deve responder um inteiro correspondente à posição corrente.

Algumas operações sobre a coleção podem ser realizadas pelos cursores, pois eles apenas representam maneiras de percorrer uma coleção. Assim, um cursor pode responder se a coleção está vazia. Do mesmo modo, podem ser realizadas operações de enumeração da coleção através do cursor.

O código (57)

```
variável prenome ⇒ «Roberto; David; Eraldo;
                     Francisco; Haroldo; Jaime; Miguel;
                     Pedro; Ricardo»
variável cur ⇒ Leitor‹prenome›
```

cria uma variável *prenome* com um vetor de signos e outra, *cur*, com um cursor para leitura deste vetor. As seguintes expressões, executadas em sequência, responderão os seguintes resultados (58)

Expressão	*Resultado*
`próximo de cur`	`Roberto`
`próximo de cur`	`David`
`corrente de cur = $Roberto`	`não`
`cur terminado`	`não`
`corrente de cur = $Eraldo`	`sim`
`posição de cur`	`3`
`cur pula 1`	`cur`
`próximo de cur`	`Haroldo`
`cur pula até $Ricardo`	`cur`
`cur terminado`	`sim`
`cur recomeça`	`cur`
`próximo de cur`	`Roberto`

Comumente, mas nem sempre, um cursor é posicionável. Isto significa que a sua posição corrente pode ser alterada arbitrariamente. Isto é feito pela operação de **posicionamento**. Operações especiais podem posicionar o cursor no início ou no fim da coleção. Com o **salto**, o cursor pode avançar um dado número de elementos. A **procura** avança o cursor até a próxima ocorrência de um dado objeto.

Com um cursor posicionável pode-se simplesmente olhar o elemento corrente, sem avançar o cursor. Ou o avanço pode ser condicionado a se o elemento é igual a um dado objeto. Outra operação pode responder a subcoleção da posição corrente até a próxima ocorrência de um dado objeto. Através de um cursor posicionável pode-se obter a cópia reversa de uma coleção.

4.4.7. Literais

Um tipo pode definir literais para representar objetos específicos. Os literais definidos pelos usuários têm que ser nomes. O escopo desse literal será idêntico ao do seu tipo. Se a introdução de um literal causar conflito, em algum escopo, com algum literal já existente, ambos só poderão ser utilizados através da **qualificação** do nome, isto é, explicitamente designando-se qual o tipo desejado associado ao literal.

4.4.7.1. Objeto Indefinido

A ausência de valor ou um valor sem significado é representada por um objeto do tipo *Objeto Indefinido.* Esta abordagem, adotada em diversas linguagens de programação orientadas a objetos, permite uma uniformidade do modelo de objetos vantajosa e a possibilidade de se ter um modelo "fechado". É necessária a existência de um literal, designando este objeto indefinido, para que se possa atribuí-lo a uma associação. Observe-se que um campo com um valor indefinido existe e o modelo se comporta diferentemente quando um objeto não possui um dado campo.

4.4.8. Enumeração

É aconselhável que as linguagens que implementem o modelo de objetos forneçam um mecanismo para a criação de "tipos por enumeração", como é comum em diversas linguagens de programação. Este mecanismo será uma abreviação de um conjunto de ações que podem ser descritas pelos recursos básicos do modelo. Por exemplo, a criação de um tipo *Dia* enumerando os seus literais *seg, ter, qua, qui, sex, sab* e *dom,* poderia ser implementada criando-se, na base de objetos, uma coleção do tipo sequência, definindo um tipo implícito para os seus elementos, que não possuiriam associações, criando-se sete objetos deste tipo e atribuindo-se cada um deles ao literal desejado. Acrescentando-se, a seguir, a esta coleção a restrição de que ela será imutável. Note-se que uma notação mais confortável para a definição de tipos por enumeração é conveniente.

Observe-se que o modelo não restringe tipos por enumeração a sequência de literais. Pode-se, por exemplo, basear a enumeração a um conjunto, ao invés de uma sequência, implicando que os literais não definiriam uma ordem entre si. Por exemplo, o domínio do tipo *Cor* pode ser um conjunto definido a partir da enumeração dos literais *branco, vermelho, amarelo, azul, verde, marrom* e *preto.*

4.4.9. Restrições em Padrões

Um padrão pode definir uma **restrição de inalterabilidade** que indicará que uma associação restrita a esse padrão não poderá ser alterada e, consequentemente, não permitirá a operação de atribuição. Um padrão também pode definir uma **restrição de imutabilidade**, indicando que o objeto associado a uma associação com esta restrição será considerado imutável, ou seja, todas as suas associações receberão restrição de imutabilidade. Uma coleção imutável é fixa, não podendo receber novas associações.

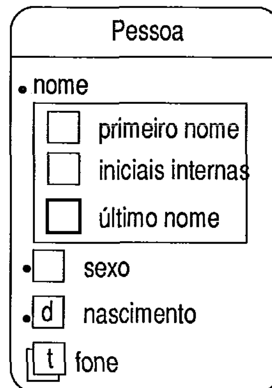


Figura 42: Atributos Constantes

A Figura 42 indica, através da colocação de pontos (•), que os atributos *nome, sexo* e *nascimento* são **constantes**. Isto significa que não pode haver atribuição a nenhum destes atributos e que os seus objetos associados também não podem ser alterados. O atributo *fone* associa uma pessoa a um objeto coleção que também não pode ser alterado, pois este é o significado da caixa dupla. Todavia, o conteúdo dete objeto coleção pode ser alterado, com telefones sendo acrescentados ou removidos. Mais além, cada telefone é um texto que não pode ser alterado, isto é, ter algum caracter modificado, removido ou acrescentado. Isto é representado pelo *t* dentro da caixa. Se um telefone pudesse ser editado, seria usado *texto* no lugar de *t*. Por outro lado, se o atributo *fone* também fosse constante (precedido por •), não se poderia alterar, remover ou acrescentar telefones à coleção. Para que se pudesse atribuir um outro objeto coleção a *fone*, situação menos comum, sua caixa deveria ser simples, indicando associação a um objeto, e seu tipo deveria ser especificado como *coleção de t*, ou algo semelhante.

Uma coleção pode possuir **restrições de unicidade**, que definem que todos os elementos da coleção diferem entre si através de um determinado con-

junto de atributos. Como casos particulares, este conjunto de atributos pode conter um único atributo ou todos os atributos dos elementos. A exigência de que um atributo seja único implica que nunca haverá dois elementos na coleção que consultados sobre este atributo respondam objetos iguais. Se a restrição for aplicada ao conjunto de todos os atributos, significará que dois elementos quaisquer da coleção sempre diferem entre si de alguma maneira. O tipo *Conjunto* define uma coleção genérica que possui restrição de unicidade para a identidade de seus elementos. Todavia, sequências, listas, vetores e dicionários também podem estar restritos por unicidade. Para dicionários, a unicidade se aplica aos valores, pois as chaves de um dicionário já são únicas, por definição.

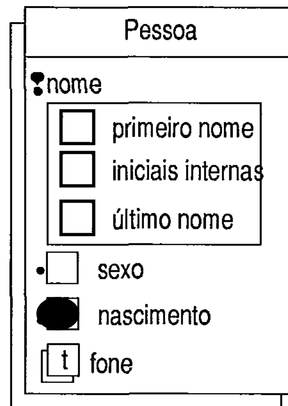
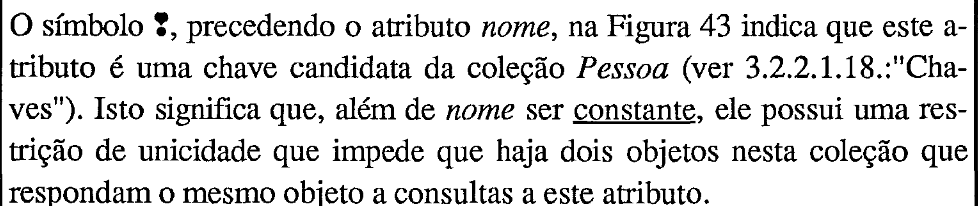


Figura 43: Chave Candidata em uma Coleção


O símbolo ❢, precedendo o atributo *nome*, na Figura 43 indica que este atributo é uma chave candidata da coleção *Pessoa* (ver 3.2.2.1.18.:"Chaves"). Isto significa que, além de *nome* ser constante, ele possui uma restrição de unicidade que impede que haja dois objetos nesta coleção que respondam o mesmo objeto a consultas a este atributo.

Grandezas, como números, datas e caracteres, além de tipos definidos por enumeração de uma sequência, são chamados de **tipos escalares**. Um padrão restrito a um tipo escalar pode ser restringido ainda mais através da definição de uma **restrição de intervalo**. Esta define valores máximo e mínimo que objetos podem ter para estar em conformidade com o padrão.

O padrão do atributo *número da coluna*, da Figura 13, na página 99, possui uma restrição de intervalo, indicando que seus valores, que são do tipo *Inteiro*, têm que pertencer ao intervalo especificado.

4.4.10. Banco de Objetos e Persistência

O **banco de objetos** é um dicionário predefinido e permanente que é o único responsável pela persistência de objetos. A chave de cada associação incluída neste dicionário representa um nome **global**, isto é, acessível por qualquer método[23]. Um objeto existe em um dado momento caso esteja incluído neste dicionário ou esteja associado a algum objeto existente. Só nestes casos um objeto pode ser acessado e usado por algum método. A persistência se refere ao tempo que o objeto existe como sendo um objeto usável.

4.4.11. Diagramas de Objetos

Os **Diagramas de Objetos (DOO)** são uma notação gráfica usável na modelagem conceitual de bases de objetos do GEOTABA, representando visualmente os conceitos contidos no modelo de objetos do GEOTABA. Ao mesmo tempo que os Diagramas de Objetos são adequados à representação de modelagens semântica da informação, eles são mapeados trivialmente no Modelo de Objetos do GEOTABA. Esta subseção apresenta os Diagramas de Objetos, mostrando, através de exemplos, como a semântica da informação contida em uma modelagem tradicional, utilizando um Diagrama de Entidades-Relacionamentos comum, pode ser representada pelos Diagramas de Objetos. É mostrado também como as noções introduzidas neste exemplo são capazes de modelar propriedades não representáveis pelos Diagramas de Entidades-Relacionamentos convencionais. Extensões aos Diagramas de Objetos estão previstas, de modo a incorporá-los a uma linguagem visual de definição e manipulação de objetos para o GEOTABA. Os Diagramas de Objetos têm sido empregados na especificação do metaesquema do GEOTABA e como ferramenta para ajudar a compreensão e o aperfeiçoamento do Modelo de Objetos do GEOTABA.

A utilização intensiva da capacidade gráfica dos computadores tem sido carta fundamental para o aumento da aceitação dos sistemas pelos indivíduos. A criação de uma representação gráfica para o Modelo de Objetos é uma providência que visa incrementar a usabilidade deste modelo.

---

[23]Este acesso deve ser regido por direitos de acesso por autorização, omitidos nesta explanação por não serem fundamentais na proposição do modelo de objetos.

4.4.11.1. Introdução

4.4.11.1.1. Objetivo

A função principal do DOO pode ser comparada à função da representação gráfica DER - Diagramas de Entidades-Relacionamentos. Ambas pretendem poder representar modelagens semânticas de informação. Todavia, sem qualquer prejuízo quanto à capacidade de representar estruturas e restrições, sem impor mecanismos extravagantes que dificultem a modelagem ou a torne artificial demais, o DOO tem que se manter diretamente mapeável no Modelo de Objetos do GEOTABA. Assim, um diagrama deve corresponder exatamente a um esquema de tipos e nomes globais implementável no GEOTABA.

A representação gráfica proposta é adequada à utilização a mão livre, não contendo símbolos difíceis de desenhar. A possibilidade de se ter ferramentas de software para auxílio à confecção desses diagramas permite encarar essa representação gráfica como base para uma verdadeira linguagem visual. Para tal, os DOOs devem poder representar totalmente o Modelo de Objetos, não apenas o nível conceitual dele. Para isto ser possível, os diagramas devem ser estendidos com mecanismos para representação de outros níveis de abstração do modelo: de implementação, externo e de representação. A possibilidade de se representar métodos e de se executar interativamente mensagens aos objetos, completarão a linguagem visual.

4.4.11.1.2. Escopo

Esta subseção se concentra na descrição das características dos Diagramas de Objetos comparáveis com as características dos Diagramas de Entidades-Relacionamentos tradicionais. Outras possibilidades dos Diagramas de Objetos, capazes de representar outros tipos de restrições de integridade e características do Modelo de Objetos não capturáveis pelos Diagramas de Entidades-Relacionamentos convencionais, não são discutidas aqui. O objetivo é mostrar como toda a informação modelável através de um DER pode ser representada por um DO.

4.4.11.1.3. Trabalhos Correlatos

Notações gráficas para representar modelagens orientadas a objetos têm sido propostas e discutidas, ainda que escassamente [BOOC91] [COAD90] [HEND90] [ALAB88]. Modelos de dados semânticos produziram várias notações gráficas para modelagem conceitual, por exemplo, GSM [HULL87], DER

[CHEN76], EER [TEOR86], FDM [SHIP81] [DAYA84]. O aproveitamento de qualquer destas notações é impedido pelas características específicas do Modelo de Objetos do GEOTABA, tais como: distinção entre tipos e coleções (incomum nos modelos semânticos); associações constantes e variáveis; definição de relacionamentos (inexistentes nas notações orientadas a objetos) através de mapeamentos e ligações; e herança e sobreposição de propriedades, inclusive de mapeamentos.

4.4.11.1.4. Organização da Subseção

Os Diagramas de Objetos são ferramentas para modelar conceitualmente informação de acordo com características do Modelo de Objetos do GEOTABA. Portanto, certas peculiaridades do Modelo de Objetos, referentes ao nível conceitual de abstração, impõem requisitos sobre a representação gráfica, isto é, sobre os diagramas. Outros requisitos atendidos pelos Diagramas de Objetos se referem a características desejáveis de manipulação dos diagramas pelo usuário. O item 4.4.11.2.:"Requisitos para os DOOs" enumera características relevantes do Modelo de Objetos e características de utilização pelo usuário e os requisitos correspondentes.

O item 4.4.11.3.:"Um Exemplo de Modelagem" apresenta as noções envolvidas nos Diagramas de Objetos através da modelagem conceitual da informação de um exemplo típico utilizado na literatura para estabelecer as propriedades dos Diagramas de Entidades-Relacionamentos. Finalizando, no item 4.4.11.4.:"Outras Considerações", são apresentados um sumário, perspectivas futuras e conclusões sobre o apresentado.

4.4.11.2. Requisitos para os DOOs

O Modelo de Objetos do GEOTABA é uma ferramenta conceitual equivalente a uma linguagem com semântica definida, porém sem uma sintaxe particular. Para o mesmo modelo de objetos, diferentes sintaxes podem ser estabelecidas. A notação gráfica proposta pode ser vista como uma possível sintaxe para simbolizar alguns conceitos do Modelo de Objetos.

Os Diagramas de Objetos aqui apresentados são uma sublinguagem dessa linguagem gráfica. Apenas a parte declarativa de definição de objetos é contemplada neles. A possibilidade dos diagramas poderem representar esta parte do Modelo de Objetos impôs requisitos específicos ao projeto destes diagramas. A seguir estão enumerados alguns dos requisitos para a representação gráfica relativos ao Modelo de Objetos.

4.4.11.2.1. Tipos e Coleções

O Modelo de Objetos distingue o conceito de tipo de objetos do conceito de coleções de objetos. Classes reúnem apenas descrições de propriedades comuns a determinados objetos. Uma coleção é um depósito de objetos. As consultas, em geral, se referem às coleções.

O diagrama permite a representação da definição de tipos e, coletivamente, de esquemas de tipos. O diagrama, além disso, deve permitir a definição de nomes globais persistentes, em geral coleções que formam bases de objetos, e, coletivamente, de esquemas de nomes globais. O diagrama deve poder ser visto em abstrações diferentes. Pode-se usá-lo tanto para representar um esquema de tipos, como para representar um esquema de globais, ou ainda para representar simultaneamente os dois esquemas.

4.4.11.2.2. Constantes e Variáveis

O Modelo de Objetos utiliza associações de nomes com os objetos para que os usuários possam manipular os objetos através de expressões com significado adequado. O diagrama deve poder representar símbolos como nomes ou expressões em português.

Os diagramas permitem a representação distinta de associações constantes e de variáveis. Em uma associação constante, o valor da associação é sempre o mesmo objeto, e este é imutável, ou seja, seus atributos são também imutáveis. Um tipo de associação variável comum relaciona um nome a uma coleção. Embora esta coleção não seja imutável, a associação do nome com a coleção, em geral, não pode ser modificada. Esta situação requer uma representação própria, devido a sua importância.

4.4.11.2.3. Relacionamentos

As associações implementam relacionamentos entre objetos. Atributos são associações com nomes. Componentes e mapeamentos são tipos especiais de associações. Os Diagramas de Objetos representam distintamente os diversos tipos de associação. Todavia, a semelhança entre estes tipos é notável nas suas representações gráficas.

Um relacionamento binário define um mapeamento de um objeto em outros. Este relacionamento pode definir um outro mapeamento, inverso do anterior. Neste caso, os dois mapeamentos formam um único relacionamento e o vínculo entre eles, nos diagramas, deve ser evidente. Descrevendo um relacionamento entre objetos, em cada objeto envolvido, através de atributos

representando os seus mapeamentos, permite se ter uma visão orientada a objetos dos relacionamentos.

Relacionamentos não binários e relacionamentos com atributos, no Modelo de Objetos, são muito parecidos com os objetos comuns. Nos diagramas, a representação destes relacionamentos se aproxima das representações de objetos comuns.

4.4.11.2.4. Cardinalidade dos Mapeamentos

Também de acordo com o modelo, as associações têm nome e cardinalidade mínima e máxima. Assim, uma associação pode ser opcional ou obrigatória e ser univalorada ou multivalorada. Estas diferenças são explícitas nos diagramas. Uma associação opcional significa que o objeto pode ser associado ao objeto nulo ou a uma coleção vazia. Uma associação multivalorada, de acordo com o Modelo de Objetos, é a associação do objeto com uma coleção de objetos e assim ele é representado no diagrama.

4.4.11.2.5. Domínios Predefinidos

Por último, as associações podem ser feitas para objetos definidos na modelagem ou para objetos de domínios predefinidos do sistema. Os diagramas têm formas diferentes para representar associações em domínios triviais predefinidos pelo sistema, associações em objetos definidos no próprio diagrama e associações em objetos que não sejam do sistema, mas já tenham sido definidos. Este último tipo de representação de mapeamentos é usável no desenvolvimento incremental dos esquemas.

4.4.11.2.6. Serviços

Além dos atributos, componentes e mapeamentos, um objeto pode definir serviços, ou seja, processamento. As assinaturas dos servoços podem ser representadas nos diagramas.

4.4.11.2.7. Tipos e Herança

No Modelo de Objetos, a tipificação relaciona um objeto ao seu tipo. É simples a identificação dos tipos dos objetos representados em um diagrama. A herança de propriedades de um supertipo pelos seus subtipos pode ser representada nos diagramas. Do mesmo modo é possível se representar a redefinição de propriedades em um subtipo. Os pseudotipos são representados distintamente dos demais tipos.

4.4.11.2.8. Chaves

O Modelo de Objetos do GEOTABA suporta a definição de restrições sobre coleções de objetos. Por exemplo, um determinado atributo dos elementos de uma dada coleção pode ser uma chave para esta coleção, ou seja, não aceitar valores nulos, nem ter dois elementos na coleção com valores iguais nesse atributo. Os diagramas provêm mecanismos de definição de quais atributos são chaves em uma coleção.

4.4.11.2.9. Nomes Implícitos

O modelo é bastante flexível quanto a necessidade de se dar nomes às entidades modeladas. Assim, pode-se ter tipos sem nomes e o mesmo nome pode identificar entidades diferentes: tipos, coleções, atributos, mensagens, etc. A representação gráfica deve simplificar ao máximo a tarefa do seu usuário de escolher nomes para as coisas. Isto é alcançado pela existência, na representação, de regras para a definição de nomes *default*, isto é, na ausência de uma especificação de nome feita pelo usuário. Entretanto, o usuário poderá definir nomes adequados quando bem o desejar.

4.4.11.2.10. Níveis de Visualização

Outros requisitos existem independentemente do Modelo de Objetos. Por exemplo, a representação gráfica deve permitir múltiplos níveis de abstração. Uma mesma modelagem realizada com esses diagramas pode ter várias maneiras de ser apresentada, dependendo do tipo de informação que se deseja visualizar.

Os diagramas não obrigam o usuário a fazer declarações e definições que podem ser inferidas. Desse modo, a representação gráfica é lacônica, sem prejuízo para a sua expressividade.

4.4.11.3. Um Exemplo de Modelagem

Características dos Diagramas de Objetos serão mostradas através de um exemplo. Visando a comparação com os Diagramas de Entidades-Relacionamentos, o exemplo escolhido é um já utilizado para exemplificar estes diagramas [ELMA89]. O exemplo descreve uma base de dados COMPANHIA, através do Diagrama Entidade-Relacionamento da Figura 44, que armazena informação sobre os empregados, departamentos e projetos de uma companhia. Os seguintes requisitos são representados no Diagrama Entidade-Relacionamento:

1. A companhia é organizada em departamentos.

2. Todo departamento tem um nome, um número e um empregado que é o gerente do departamento.
3. A data a partir da qual o gerente de um departamento iniciou a sua gerência é relevante.
4. Um departamento pode estar espalhado em diversos locais de trabalho.
5. Um departamento pode controlar projetos.
6. Um projeto tem um nome, um número e se realiza em um determinado local.
7. Todo projeto é controlado por um e somente um departamento.
8. Devem ser mantidos, para cada funcionário, o seu nome, cpf, endereço, salário, sexo e data de nascimento.
9. Todo empregado é alocado a um e somente a um departamento.
10. Todo departamento tem empregados alocados a ele.
11. Um empregado pode trabalhar em diversos projetos controlados por departamentos diferentes.
12. A quantidade de horas que cada empregado trabalha em cada projeto precisa ser registrada.
13. Um empregado pode ter um outro empregado como sendo o seu supervisor.
14. Um empregado pode ser supervisor de diversos empregados.
15. Um empregado pode ter dependentes.
16. Cada dependente tem nome, sexo, data de nascimento e o tipo de dependência que ele tem com o empregado.
17. Nomes de pessoas são compostos por um primeiro nome, as iniciais dos nomes internos e pelo último nome.
18. Dois departamentos não podem ter o mesmo nome.
19. Dois departamentos não podem ter o mesmo número.
20. Dois projetos não têm o mesmo nome.
21. Dois projetos não têm o mesmo número.
22. Dois empregados não têm o mesmo cpf.
23. Um empregado não pode ser gerente de mais de um departamento.
24. Um empregado não tem mais de um dependente com o mesmo nome.

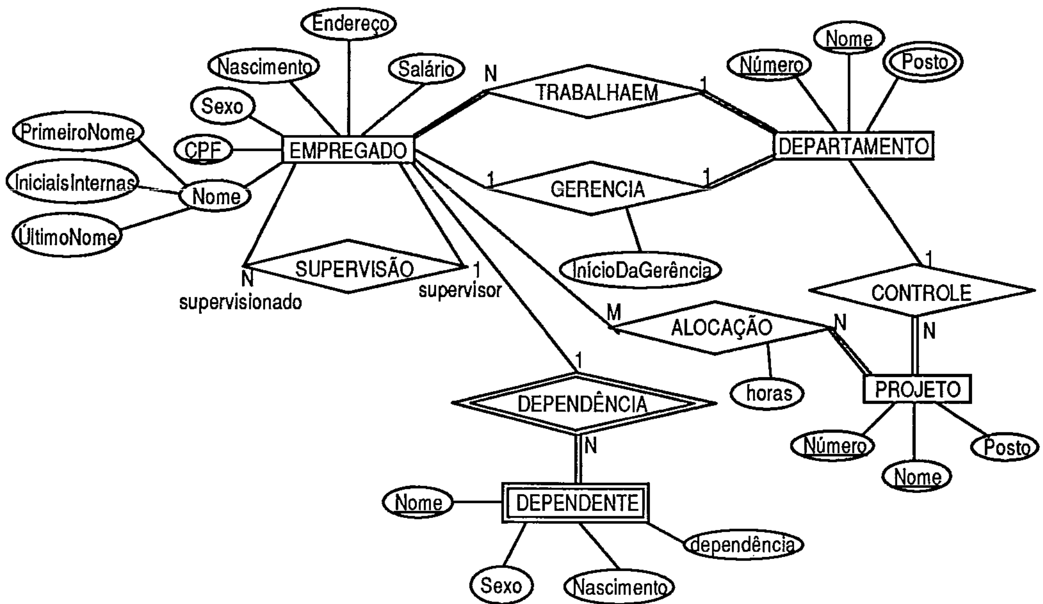


Figura 44: Diagrama Entidade-Relacionamento para a Companhia

4.4.11.3.1. O Diagrama de Objetos

A Figura 45 representa, com um Diagrama de Objetos, o esquema de tipos e nomes globais para uma base de objetos *Companhia*, de acordo com o Modelo de Objetos do GEOTABA. Através desse exemplo, serão mostradas, a seguir, as soluções empregadas pelos Diagramas de Objetos para atender aos requisitos estabelecidos anteriormente.

4.4.11.3.1.1. Notações de Classes e Coleções

O Diagrama de Objetos da Figura 45 representa simultaneamente os esquemas de tipos e de nomes globais. As entidades *Empregado*, *Projeto* e *Departamento* estão representadas como coleções globais de objetos, contrastando com *Pessoa, Dependente, Alocação* e *Gerente* que são tipos cujos objetos não estão reunidos em uma única coleção. A Figura 46 mostra as notações diferentes de tipos e atributos. Com a distinção entre tipos e coleções pode-se representar as propriedades comuns de empregados e dependentes no tipo *Pessoa*, pode-se definir que certos empregados são gerentes e pode-se dispensar uma coleção global de dependentes. Assim, a existência de um dependente depende da sua associação a um empregado.

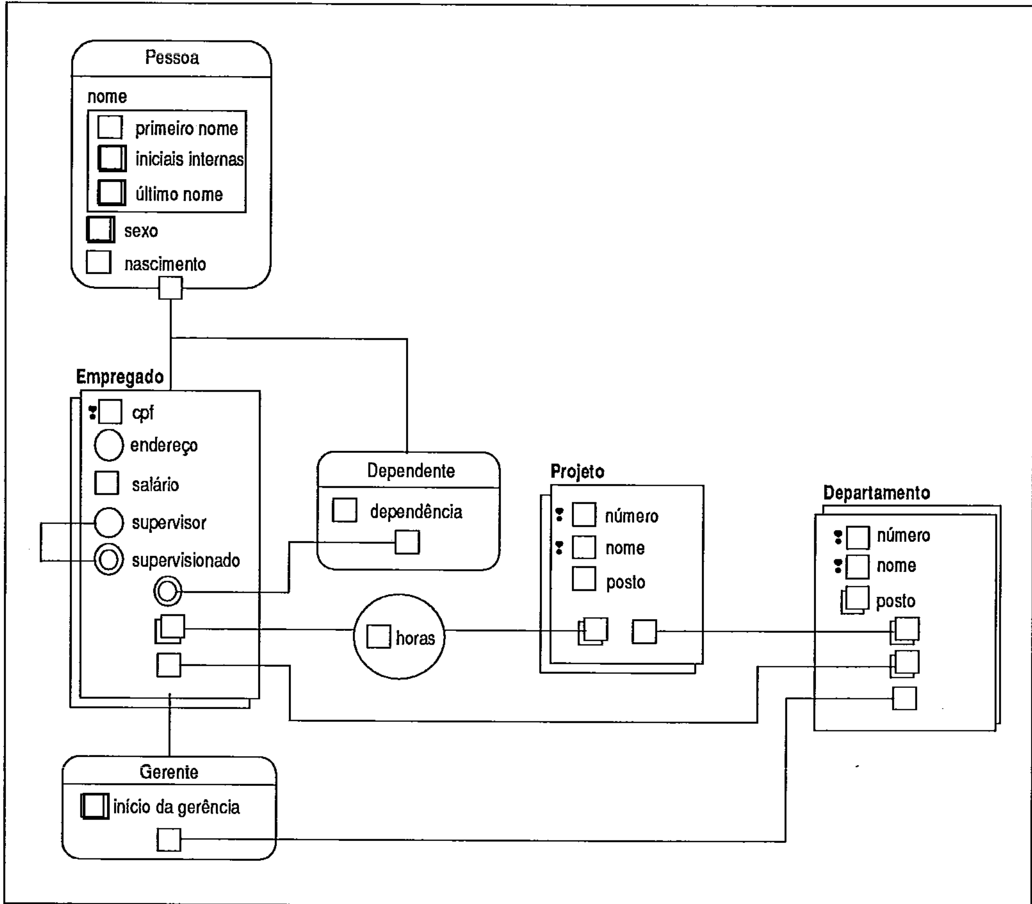


Figura 45: Diagrama de Objetos para a Companhia

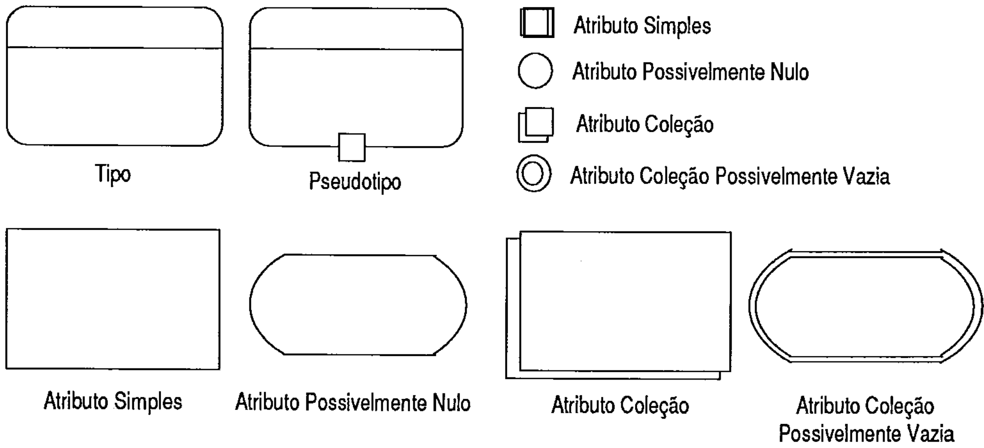


Figura 46: Ícones e Caixas para Tipos e Atributos

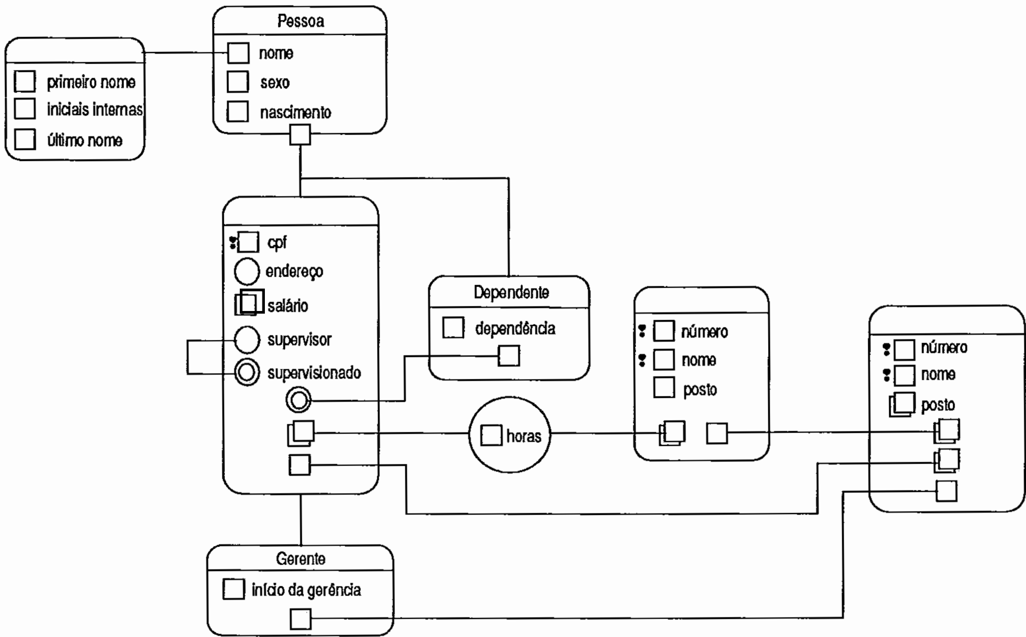


Figura 47: Esquema de Tipos

4.4.11.3.1.2. Esquemas de Tipos e Nomes Globais

A representação gráfica pode ser usada para representar o esquema de tipos e o esquema de nomes globais em diagramas separados. A Figura 47 apresenta o esquema de tipos correspondente ao diagrama da Figura 45. O esquema de globais correspondente é mostrado na Figura 48.

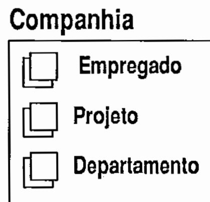


Figura 48: Esquema de Globais

Confrontando-se as três figuras, conclui-se que, no diagrama geral da Figura 45, as três coleções, *Empregado*, *Projeto* e *Departamento*, definem implicitamente três tipos correspondentes. Na verdade, neste diagrama, a notação das coleções com tipos implicitamente definidos é uma abreviatura, conforme mostrado na Figura 49.

O Diagrama de Objetos da Figura 45 é um esquema que pode ser feito para a *Companhia*. Outros esquemas poderiam ser criados para representar a mesma informação. Por exemplo, não há necessidade de se ter três coleções globais (*Empregado*, *Projeto* e *Departamento*), bastando existir uma delas.

Como todos os objetos estarão ligados entre si, uma coleção persistente garante a persistência dos demais objetos.

Todavia, se só existir uma das coleções globais, todo acesso a qualquer informação será feito através desta coleção. Na modelagem adotada foi empregado um outro critério: toda entidade forte corresponde a uma coleção global. Com isto se tem acesso aos empregados da companhia independentemente dos projetos e departamentos; se tem acesso aos projetos independentemente, e aos empregados também.

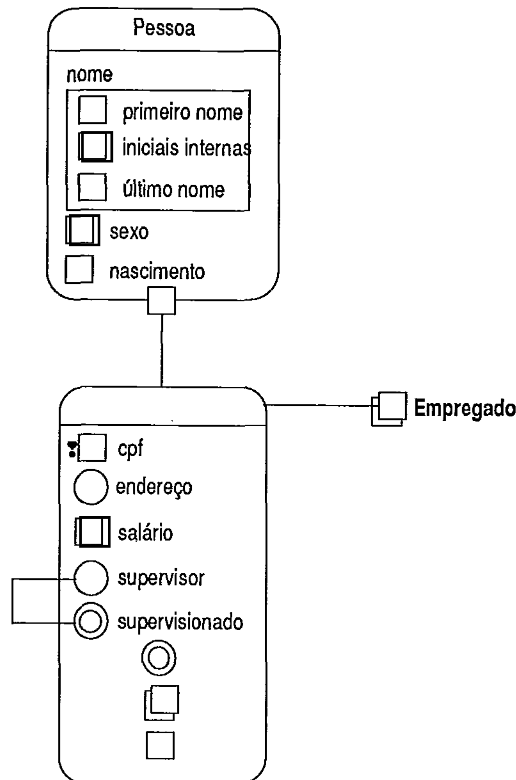


Figura 49: Expansão da Coleção Empregado

4.4.11.3.1.3. Representação de Associações Constantes e Variáveis

Os nomes *Empregado*, *Projeto* e *Departamento*, dentro do escopo da base de objetos da *Companhia*, estão associados a coleções. Estas associações são inalteráveis. Isto significa que, apesar da coleção *Empregado* poder ter o seu conteúdo alterado, colocando-se ou retirando-se empregados, o nome *Empregado* estará permanentemente associado a esta mesma coleção.

Outros atributos foram modelados usando-se associações variáveis. Por exemplo, todo empregado tem um salário, porém este pode ser alterado. Alguns atributos poderiam ser modelados usando associações constantes, como na

Figura 50. Neste diagrama, não é permitido que um empregado mude o seu CPF. Também não é permitido que uma pessoa na companhia mude de sexo ou de data de nascimento. Restrições semelhantes poderiam ser colocadas caso um departamento ou projeto não pudessem mudar de número ou se a data de início de uma gerência não pudesse ser alterada.

4.4.11.3.1.4. Relacionamentos Binários sem Atributos

As ligações entre as entidades representam os tipos de relacionamentos existentes entre elas. Assim vê-se, na Figura 45, que empregados e departamentos se relacionam. As extremidades das ligações representam os mapeamentos.

Observando-se a ligação Empregado-Departamento, interpreta-se que, através desta ligação, um empregado tem um atributo que representa um mapeamento em um departamento. Implicitamente, o nome deste mapeamento é "departamento". Portanto, dado um empregado pode-se falar no "departamento do empregado".

O mapeamento inverso também está especificado. Um departamento tem um atributo, com o nome "empregado" implícito, que representa uma coleção de empregados. Um objeto percebe o seu relacionamento com outro(s) objeto(s) através de um atributo equivalente ao mapeamento do objeto no(s) outros(s). A representação gráfica adotada segue a linha de representar os relacionamentos com uma visão orientada a objetos.

4.4.11.3.1.5. Outros Relacionamentos

Como mencionado anteriormente, relacionamentos não binários ou com atributos têm que ser tratados de modo semelhante a objetos em si. Isto é exemplificado, na Figura 50, pela ligação *Empregado-Projeto*. Um empregado pode trabalhar em vários projetos e um projeto pode ter vários empregados a ele alocados. Isto explica os atributos em *Empregado* e *Projeto*. Entretanto, cada alocação de um empregado a um projeto, representada pela oval da figura, precisa guardar um atributo contendo o número de horas que o empregado está alocado a este projeto.

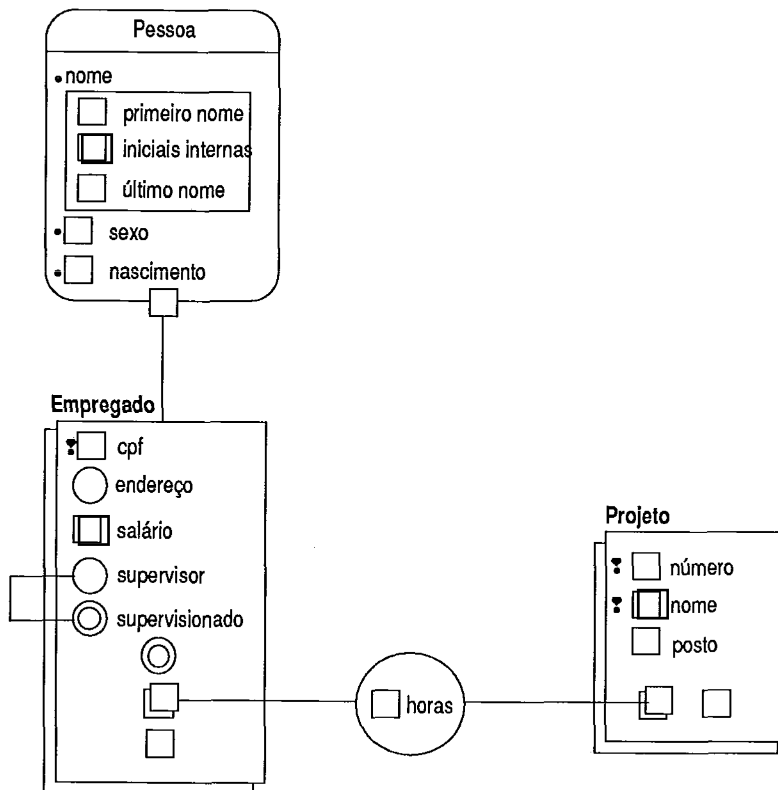


Figura 50: Associações Constantes em Atributos e Ligação Empregado-Projeto através de Alocação

Modelando uma alocação, como se fosse um tipo de objeto, conforme a Figura 51, um empregado tem um conjunto de alocações a projetos e um projeto tem um conjunto de alocações de empregados. Neste diagrama, o relacionamento *Empregado-Departamento* foi partido em dois relacionamentos: *Empregado-Alocação* e *Projeto-Alocação.* O defeito desta quebra é visível quando se nota que nada restringe um empregado a ter duas alocações para um mesmo projeto.

4.4.11.3.1.6. Mapeamento Sem Inverso

Pela Figura 45 observa-se que empregados se relacionam com objetos do tipo *Dependente.* A representação utilizada para o atributo de *Empregado* correspondente ao seu mapeamento em dependentes significa que o valor deste atributo é uma coleção possivelmente vazia de objetos do tipo *Dependente.*

Inversamente, em *Dependente* há um atributo que indica de qual empregado a pessoa é dependente. Esta é uma opção da modelagem da base de objetos. Caso seja assumido que os dependentes só são manipulados a partir dos empregados, o último mapeamento não precisa ser especificado, como na Figura 52.

4.4.11.3.1.7. Nomes de Mapeamentos Explícitos

O auto-relacionamento Empregado-Empregado, indicando que um empregado pode (ou não) ter supervisor e pode (ou não) ter empregados supervisionados, é representado pelos atributos *supervisor* e *supervisionado*. Estes atributos tiveram que receber nomes explicitamente, ao contrário dos demais atributos de mapeamento. Isto foi necessário porque a regra de nomes implícitos estabelece que o nome *default* de um mapeamento é o mesmo do tipo ou objeto para o qual será feito o mapeamento. Assim, o atributo em *Empregado* que mapeia para os seus dependentes se chama *dependente*, pois mapeia para objetos do tipo *Dependente*. Se este nome não fosse adequado para explicar o mapeamento, o projetista poderia especificar um outro explicitamente. Para evitar ambiguidade, o auto-relacionamento não deve usar nomes implícitos.

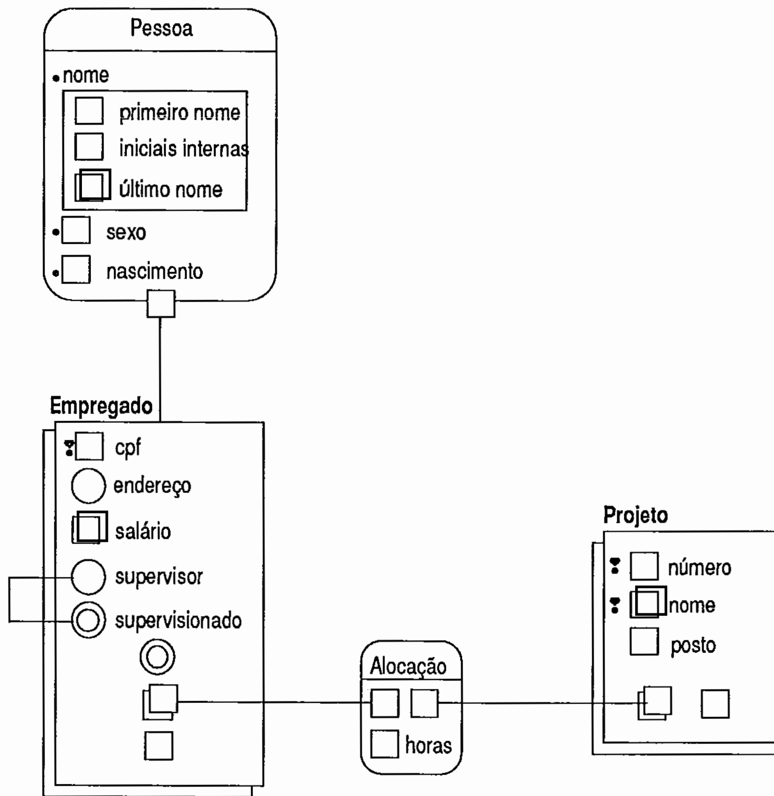


Figura 51: Ligação Quebrada

4.4.11.3.1.8. Padronização

A Figura 52 também apresenta diversas formas de especificação de padrões de atributos. Por exemplo, o atributo *salário* de *Empregado* é restrito ao tipo *valor*, não especificado nesses diagramas. O atributo *dependente* poderia usar a mesma notação, dispensando, assim, a linha que liga o atributo ao

tipo. Quando o atributo e seu tipo possuem o mesmo nome, este pode ser omitido. Esta regra foi usada na especificação dos atributos *sexo, cpf* e *dependência.* Tipos muito usados podem ser representados por abreviaturas. Isto foi utilizado na especificação dos atributos *primeiro nome, iniciais internas, último nome* e *endereço*, onde a letra *t* representa *texto imutável.* A letra *d* no atributo *nascimento* representa o tipo *data.* A especificação do tipo do atributo *nome* é feita *in loco*, listando-se seus subatributos. Por último, as ligações também representam restrições de padrão. Por exemplo, tanto *supervisor*, quanto *supervisionado*, são restritos a empregados, pois estes atributos se ligam a atributos de *Empregado.*

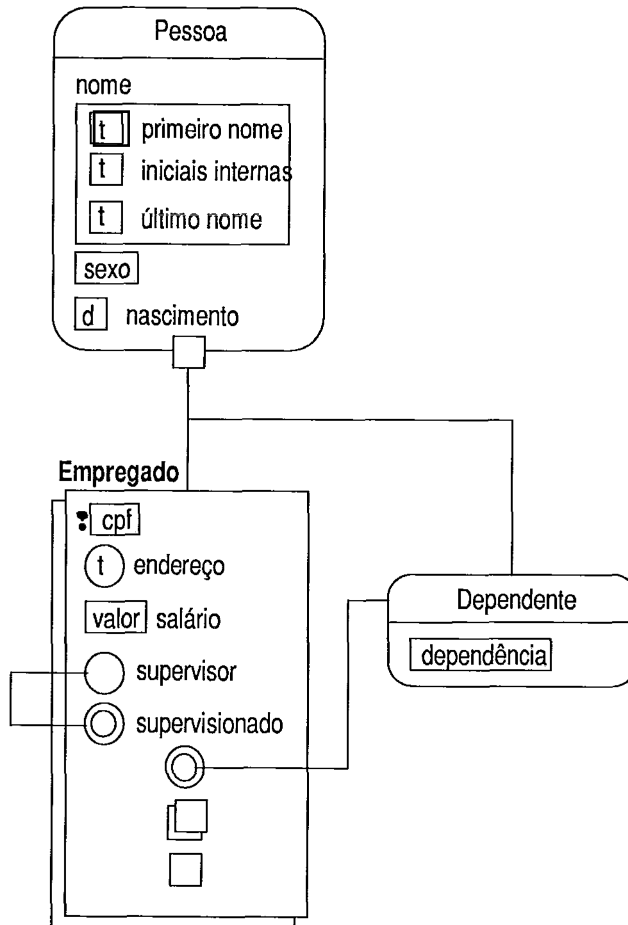


Figura 52: Mapeamento sem Inversa

4.4.11.3.1.9. Herança e Pseudoclasses

Pelo DER da Figura 44 pode-se verificar que tanto empregados como seus dependentes possuem atributos comuns: *nome*, *sexo* e *nascimento.* Nos Diagramas de Objetos esses atributos poderiam ser colocados em evidência com a criação de um supertipo para empregados e dependentes. Isto está

representado na Figura 52, com a criação do tipo *Pessoa*. Tanto *Empregado* como *Dependente* são *Pessoa* e herdam suas propriedades. Como não existe, nesta base, uma pessoa que não seja um empregado ou um dependente, *Pessoa* é um pseudotipo e assim foi representado. Note-se também que os nomes de dependentes passaram a ser vistos do mesmo modo que os nomes de empregados: estruturados em *primeiro nome, iniciais internas* e *último nome.* Em um DER convencional, sem definição de classes, tipos ou domínios, esta possibilidade aumentaria a complexidade do diagrama resultante. A Figura 53 apresenta as diversas possibilidades de um supertipo. O uso das caixas para diferenciar as alternativas é totalmente coerente com seu uso em atributos.

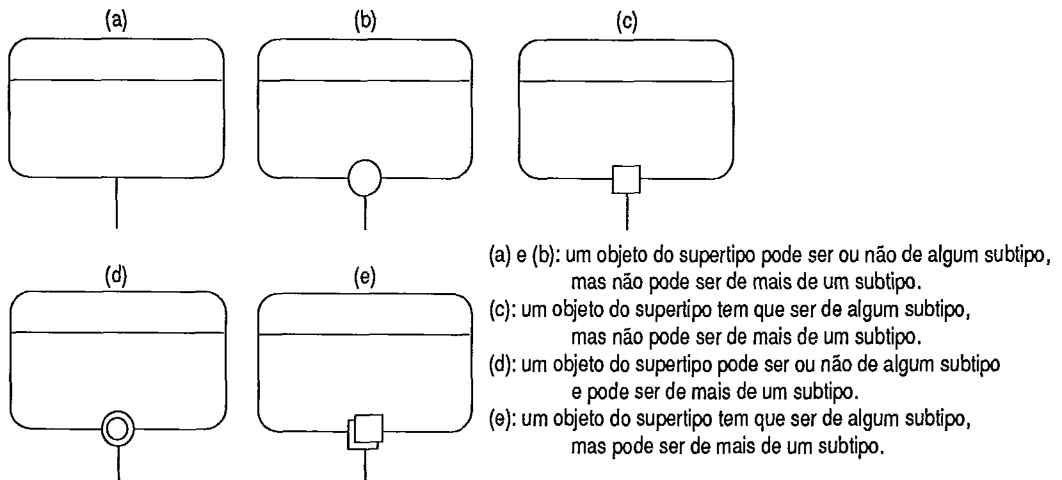


Figura 53: Hierarquização

Na Figura 52, a estruturação de *nome* de Pessoa é realizada dentro da própria caixinha que designa o atributo. Isto é feito assim por comodidade. O tipo que descreve a estrutura dos nomes de pessoas é derivável automaticamente deste diagrama, como pode ser observado na Figura 47, da página 175. Uma modelagem alternativa criaria explicitamente um tipo para nomes de pessoas e faria uma ligação do atributo *nome* de *Pessoa* a esse tipo.

Alguns empregados gerenciam departamentos. Este relacionamento parcial fica melhor modelado com a definição do subtipo *Gerente* que representa os empregados envolvidos neste relacionamento. A data de início da gerência passa a ser um atributo exclusivo desse tipo de empregado.

4.4.11.3.1.10. Sobreposição

Nada impede, no DER da Figura 44, que um gerente trabalhe em um departamento mas gerencie outro. O Diagrama de Objetos apresentado proíbe esta situação usando o fato que uma subclasse pode alterar a definição de

alguma propriedade que ela tenha herdado de suas superclasses. Isto é feito do seguinte modo: o relacionamento *Empregado-Departamento* definiu implicitamente em *Empregado* um atributo *departamento* que indica qual departamento o empregado trabalha. Um gerente ao se relacionar com o departamento que gerencia, na ligação *Gerente-Empregado*, redefine o atributo *departamento*, pois não foi dado um outro nome explicitamente para este mapeamento. Portanto, o departamento de um gerente é o que está envolvido no relacionamento de gerência. Esta característica permite uma representação elegante deste tipo de restrição.

4.4.11.3.1.11. Restrições de Chave

O exemplo define chaves candidatas para as coleções *Empregado* (*cpf*), *Projeto* (*número* e *nome*) e *Departamento* (*número* e *nome*). Os Diagramas de Objetos representam os atributos chaves precedendo-os por um sinal de exclamação ❢.

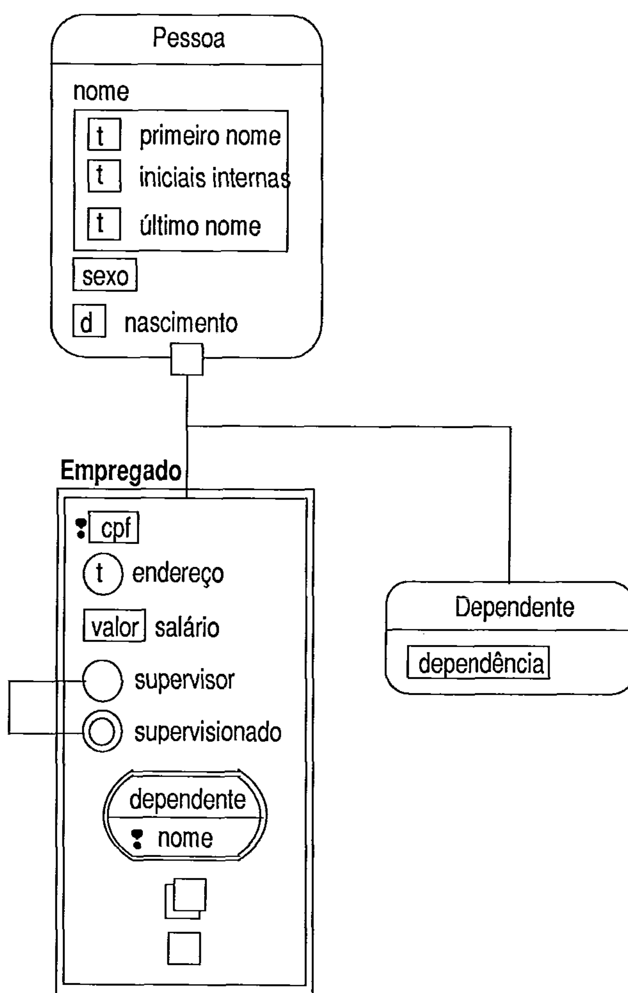


Figura 54: Chave Parcial

A restrição de que, para um mesmo empregado, não há dependentes com o mesmo nome é definida indicando que a coleção de dependentes de cada empregado, representada pelo atributo de nome implícito *dependente*, tem *nome* como chave. Entretanto, esta restrição deve ser colocada apenas na coleção referente ao atributo *dependente*, como na Figura 54.

4.4.11.4. Outras Considerações

4.4.11.4.1. Sumário

Esta subseção apresentou os Diagramas de Objetos, usáveis na modelagem conceitual de bases de objetos do GEOTABA. A propriedade fundamental dos Diagramas de Objetos é atender ao mesmo tempo a representação de modelagens semânticas de informação e serem mapeados trivialmente no Modelo de Objetos do GEOTABA.

Os diagramas espelham diversos conceitos do Modelo de Objetos, tais como: objetos, tipos e coleções; associações constantes e variáveis; reconhecimento da existência de relacionamentos entre objetos e representação destes relacionamentos através de seus mapeamentos; atributos e mapeamentos opcionais e obrigatórios, univalorados e multivalorados; herança e sobreposição de propriedades através de hierarquia de tipos; chaves de coleções.

O foco desta apresentação foi dado em como a semântica da informação contida em uma modelagem tradicional, utilizando um DER comum, pode ser representada pelos Diagramas de Objetos. Isto foi mostrado através de um Diagrama de Objetos que modela a informação descrita em um DER utilizado didaticamente para exemplificar as possibilidades dos Diagramas de Entidades-Relacionamentos. Também foi mostrado que as noções empregadas neste Diagrama de Objetos exemplo são capazes de modelar propriedades não representáveis pelos Diagramas de Entidades-Relacionamentos tradicionais, tais como, associações constantes, herança e sobreposição de atributos e de relacionamentos.

4.4.11.5. Perspectivas Futuras

Uma série de conceitos definidos no Modelo de Objetos do GEOTABA não foram empregados neste exemplo e, consequentemente, não tiveram suas representações introduzidas nos Diagramas de Objetos deste trabalho. Este é o caso da especificação de atributos componentes, da especificação de cardinalidade mínima e máxima pouco usuais, a utilização de padrões, a junção e disjunção de ligações, com a finalidade de se estabelecer condições de integridade que

envolvam mais de uma ligação, e da especificação do protocolo de serviços realizáveis pelos objetos.

Mais além, os Diagramas de Objetos devem ser encarados como um subconjunto de uma linguagem visual de definição e manipulação de objetos do GEOTABA. Para isto ser alcançado, deve ser possível especificar interativamente processamentos e executá-los. Os passos seguintes serão fornecer a possibilidade de se representar não apenas o nível conceitual, mas também os demais níveis de abstração do Modelo de Objetos: nível externo, de representação e de implementação. Isto inclui a especificação dos mapeamentos entre os diferentes níveis de abstração.

4.4.11.6. Conclusões

Os Diagramas de Objetos têm sido empregados na especificação do metaesquema do GEOTABA e como ferramenta para ajudar a compreensão e o aperfeiçoamento do seu Modelo de Objetos. Como os conceitos deste modelo são representados graficamente nos diagramas, as pessoas encontram mais facilidade de fixar as noções envolvidas e seus propósitos.

Como etapa preliminar para a especificação do metaesquema do GEOTABA, foi elaborado um Diagrama de Objetos representando o metaesquema do sistema Smalltalk/V [DIGI86]. Por metaesquema, entenda-se o esquema de classes e objetos predefinidos envolvidos na criação e manipulação básicas de classes e objetos, ou seja, as classes **Object**, **Class**, **Metaclass**, **Behavior**, **CompiledMethod**, etc. Algumas extensões foram necessárias para se poder representar serviços prestados por estas classes (compilar um texto fonte, criar um objeto, executar uma mensagem, etc.) e para representar restrições não convencionais.

Um editor de Diagramas de Objetos, denominado Flecha [CAMA92], foi desenvolvido. Ele é apresentado na seção 5.3.:"Flecha".

4.4.12. Linguagem de Descrição e Manipulação de Objetos

A **Linguagem de Descrição e Manipulação de Objetos** (**DEMO**) é uma notação textual representando os conceitos contidos no modelo de objetos do GEOTABA. A linguagem DEMO é usada para consultas[24] diretas pelo usuário, para a escrita de métodos e para a definição de esquemas (esta última aplicação não foi desenvolvida nesse trabalho). Esta linguagem possui

---

[24] *query*, em inglês.

características essenciais para a representação do modelo de objetos do GEOTABA, como a notação única para atribuição e indexação e a utilização de nomes de objetos com feição de linguagem natural, próprios para serem exibidos diretamente ao usuário final. Esta subseção apresenta algumas características adicionais de DEMO não introduzidas nos exemplos anteriores.

4.4.12.1. Conjunto de Caracteres

A linguagem DEMO requer que seus códigos sejam escritos em editores de texto avançados, com preferência aos dedicados a ela. O conjunto de caracteres na qual a linguagem se baseia é o ANSI. Todavia, a utilização de alguns operadores especiais exige que caracteres de fontes especiais possam ser inseridos. Exemplos desses caracteres são $\oplus$, $\Rightarrow$, $\ominus$, $\equiv$, $\cap$, $\ni$, λ , $\perp$, ⁝, $\Leftarrow$ *e* $\in$. A linguagem fornece formas alternativas à utilização desses caracteres especiais.

4.4.12.2. Elementos Léxicos

Os elementos léxicos da linguagem podem ser classificados em identificador, número, cadeia de caracteres, caracter, operador, pontuadores, signos e parênteses. Entre eles pode haver qualquer número de espaços, entendidos como sendo caracteres brancos, tabuladores ou comentários. Qualquer elemento léxico precedido por um caracter barra-reversa (\) é um comentário e é ignorado pela análise do programa. Entre a barra-reversa e o elemento léxico a ser ignorado pode haver caracteres brancos e tabuladores. O fim de linha é um elemento léxico, não sendo ignorado como se fosse espaço.

Os identificadores são sequências de letras, dígitos, travessões e pontos, não podendo iniciar com um dígito. As formas minúsculas e maiúsculas, acentuadas ou não acentuadas, de uma mesma letra são equivalentes nos identificadores.

Um número segue as convenções da linguagem de programação Pascal. Todavia, a vírgula é usada como vírgula decimal e um número pode conter caracteres pontos, sem efeito quanto ao valor do número e usados apenas para aumentar a legibilidade (exemplo: *32.723,45*). As funções da vírgula e do ponto podem ser trocadas por opção de compilação. Um número não pode iniciar com vírgula ou ponto. Um número inteiro pode ser escrito em hexadecimal, precedendo-o pelo caracter # (exemplo: *#a7.5f*), ou em outra base entre 2 e 9. Neste caso, o número é precedido por um dígito, que indica a base, seguido pelo caracter # (exemplo: *8#12.757*).

Literais cadeias de caracteres são delimitados por aspas. Dentro de um literal cadeia de caracteres não pode ocorrer o caracter aspas. Duas aspas seguidas denotam uma cadeia de caracteres vazia. A ocorrência de um fim-de-linha em um literal cadeia de caracteres termina-o, como se o fim-de-linha tivesse sido precedido por aspas. Ou seja, um literal cadeia de caracteres não pode ocupar mais de uma linha. Operadores de concatenação de cadeias com cadeias e com caracteres isolados podem ser usados para gerar cadeias longas ou contendo aspas.

Um literal caracter é formado cercando-se um caracter entre apóstrofos, como em '%'. O último apóstrofo é opcional.

Operadores são sequências com qualquer número dos seguintes caracteres:

! ? ¿ % & § + - ± ∓ † ‡ ÷ × * / / ⊥ \ : < ≤ = ≠ ≡ ≈ ≅ ≃ > ≥ ^ ~ ′ ¬ ∀
∃ ∧ ∨ λ ↔ ← → ↑ ↓ ↕ ↗ ↘ ↖ ↙ ⇔ ⇐ ⇒ ⇑ ⇓ ⇕ ⊕ ⊖ ⊗ ⊘ ⊙ ⊚ ⊛
⊜ ⦶ ∅ ∩ ∪ ⊃ ⊇ ⊄ ⊂ ⊆ ∈ ∉ ∋ ⊢ ⊣ ∴ ∵ ⁝ △ ▽ ◊

Por exemplo, a sequência ::= é um operador. Todavia, os caracteres + - ± ∓ † ‡ ^ ~ ¬ só podem aparecer no início dessas sequências. Sendo assim, a sequência ++ corresponde a dois operadores. O caracter º, isolado, também forma um operador.

Signos são designados por cadeias de caracteres, identificadores ou operadores precedidos pelo caracter *$*. Os pontuadores são os caracteres dois-pontos e ponto-e-vírgula. O fim-de-linha é equivalente a um ponto-e-vírgula. Diversos tipos de parênteses são empregados: *( ) [ ] { }* « » ‹ ›.

4.4.12.3. Morfologia dos Identificadores e Operadores

Um identificador pode assumir diversos papéis nos programas DEMO. Esses papéis são definidos pelas declarações dos nomes (variáveis, atributos, métodos, etc.) visíveis em cada ponto onde um identificador é usado. Um identificador pode ser classificado como **prenome**, **palavra-chave**, **complemento**, **separador**, **preposição**, **palavra-reservada** e **identificador-não-declarado**.

Os nomes de atributos, variáveis, argumentos e os nomes globais iniciam com um identificador que esteja funcionando como prenome e pode conter, opcionalmente, uma sequência de identificadores funcionando como complemento. Isto pode ser representado pela seguinte forma gramatical:

```
nomeVar ::= prenome { complemento }
```

(as chaves foram empregadas como um metasímbolo indicando repetição opcional).

Uma ação (mensagem ou evento) pode ter a forma de uma **sentença**. Esta sempre inicia com uma palavra-chave seguida opcionalmente de complementos. A seguir poderá vir um primeiro argumento, se houver. Quando há mais de um argumento, eles devem vir intermediados por separadores. Cada separador pode ter complementos seguindo-o imediatamente. A fórmula seguinte resume estas regras.

```
sentença ::=  palavra-chave { complemento }
              [argumento{separador{complemento}argumento}]
```

Estas regras permitem o uso de nomes de variáveis e sentenças bastante naturais, como *Peça Base* e *é vazia*. Quando uma declaração (de variável, sentença, etc.) é feita, os identificadores, introduzidos no nome sendo declarado, têm que ser identificadores-não-declarados ou então estarem sendo usados de modo compatível com a sua classificação anterior. Por exemplo,

```
variável Peça Base
```

declara a variável de nome *Peça Base*. Se o identificador *Peça* foi anteriormente classificado, dentro de um escopo que abranja o desta declaração, como algo diferente de prenome, será acusado um erro de compilação. O mesmo ocorrerá se *Base* não for um complemento ou um identificador-não-declarado. Observe-se que qualquer base de objetos persistente define nomes globais que, por sua vez, estabelecem uma classificação para seus identificadores constituintes. Como esses nomes são globais a todos que usarem essa base, esta classificação não poderá ser mudada por alguma declaração. Para que não surjam conflitos inesperados de uso inadequado de um identificador, pode-se fazer convenções do tipo "os prenomes devem ser substantivos; as palavras-chaves devem ser verbos; os separadores devem ser preposições que não tenham sido preclassificadas como preposição; e os complementos devem ser adjetivos ou substantivos menos importantes".

As preposições podem ser identificadores reservados para tal (sugestão: *de, do, da, dos, das, of* e o operador *!*), ou poderiam ser introduzidas em **declarações de preposição**. Esta matéria ainda não foi definida. Outros identificadores seriam predefinidos como palavra-reservada (exemplo: *retorna, responde, sentença, variável, operação, tipo, classe, consulta, atribuição, atribuidor, conectivo, relação, aditiva, multiplicativa, unária, indexação,* etc.).

Os operadores também recebem uma classificação morfológica de modo semelhante aos identificadores. De acordo com o modo como foi declarado, um operador pode ser **aditivo**, **multiplicativo**, **relacional**, **conectivo** ou **atribuidor**. Esta classificação estabelece a prioridade de avaliação dos operadores em uma expressão com múltiplos operadores e sentenças. Do

mesmo modo que os identificadores, as declarações de operação não podem redefinir a classificação anterior do operador. Por exemplo,

```
conectivo &(cond:condição)
{14()}
```

declara o operador *&* como sendo um conectivo, isto é, com prioridade de avaliação menor que os operadores relacionais. O bloco com o corpo do método só indica que a primitiva *14* deve ser executada. Operações primitivas são operações básicas realizadas pela máquina virtual DEMO. Elas são denotadas por um número inteiro seguido de uma lista de parâmetros entre parênteses.

4.4.12.4. Declarações

Algumas declarações iniciam com uma palavra-reservada que indica o significado que se pretende dar ao nome que está sendo declarado: *variável* para variáveis locais, *global* para nomes globais, etc. Em seguida há a especificação de um atributo ou apenas a especificação do nome. A especificação de um atributo, além de fornecer um valor inicial para o atributo, também define o nome do atributo. A especificação do nome de um atributo é composta pela especificação da sintaxe do nome e do padrão deste. A sintaxe do nome consiste em uma sequência de identificadores que, ou ainda não foram declarados anteriormente, ou têm classificação compatível com a empregada nesta declaração, ou seja, o primeiro será um prenome e os demais serão complementos.

Uma outra maneira de se declarar variáveis locais é através do operador λ. O objeto receptor desta operação será o valor da declaração e à direita do operador é fornecida o nome (sintaxe e padrão) sendo declarado. Uma segunda forma desse operador permite a declaração simultânea de mais de uma variável.

Se o nome empregado em uma declaração já estiver sendo usado com o mesmo significado, a associação a que se referirá a declaração será a mesma que já existia anteriormente. Por exemplo, uma declaração *variável v1*, feita quando já existe uma variável local de nome *v1*, não declara uma nova variável, mas apenas serve como uma referência a variável *v1* já existente. Isto permite que se possa ter um mesmo nome sendo usado com diversos significados. Um caso comum é se ter um atributo de um tipo de objeto com o mesmo nome de um outro tipo, a exemplo de objetos do tipo *Departamento* com atributos de nome *empregado* mesmo quando existe um outro tipo com o nome *Empregado*. Em situações ambíguas, o identificador *empregado* seria considerado como sendo uma referência ao atributo *empregado*, e não o tipo. Isto porque a escolha do significado sempre recai noo que tem o menor escopo. O nome do tipo é global; o nome de um atributo tem seu escopo restrito a expressões cujo receptor é um

objeto que possui esse atributo. Nesta situação, quando se deseja fazer referência ao tipo no lugar do atributo, é necessário se usar uma declaração *tipo Empregado*, qualificando o nome com o significado desejado.

Uma definição de método exige muito mais sofisticação na especificação da sintaxe do seu nome. Esta sintaxe possui duas formas básicas: sentença e operação. A definição de um método cuja expressão de ativação seja uma sentença inicia com a palavra-reservada *sentença*. Em seguida vem a especificação da sintaxe da sentença e a especificação do padrão do resultado do método. Por último, precedido por um ponto-e-vírgula ou fim-de-linha, tem-se o bloco que forma o corpo do método. Na especificação da sintaxe da sentença, os argumentos são apresentados entre parênteses, embora estes parênteses não sejam necessários nas sentenças de ativação do método. A definição do método é semelhante a uma mensagem enviada a alguma classe, tipo, aspecto, vista ou coleção com tipo próprio. Isto significa que o nome do receptor da "mensagem" deve preceder à palavra-reservada *sentença*. Se o nome do receptor não for colocado, a definição do método se referirá à "classe implícita". Esta pode ser alterada através de uma declaração própria. O objetivo da existência de uma classe implícita a qual os método seriam incluídos por *default* é simplificar a escrita de uma sucessão de métodos para a mesma classe e para viabilizar a criação de ambientes de programação aparentemente não orientados a objetos.

A definição abaixo (59)

```
tartaruga sentença ande (:i) para trás; { ande (-i) }
```

acrescenta ao tipo *Tartaruga* um método que implementa o serviço de andar para trás. As sentenças que ativam esse método iniciam com a palavra-chave *ande*, seguida por um argumento (não necessariamente entre parênteses), pelo separador *para* e o complemento *trás*. Por exemplo (60)

```
ande x + 50 para trás
```

O argumento desta sentença deve ser do tipo *Inteiro*, representado pelo *i* após os dois-pontos. O nome do argumento, por não ter sido fornecido, também será *i*. O corpo do método apenas faz chamar outro serviço de *Tartaruga*, usando como argumento o inteiro simétrico de *i*. Os parênteses empregados não são necessários e foram só colocados para dar clareza, evitando que o sinal - fosse confundido com a operação binária de subtração. Observe-se que a definição do método não especifica o tipo do objeto retornado e, coerentemente, o corpo do método não possui uma instrução de retorno. Neste caso, o objeto retornado é o próprio objeto receptor da mensagem (e seu tipo é *Tartaruga*).

No seguinte trecho (61)

```
omite tartaruga
sentença centraliza; { x := 0; y := 0 }
sentença recomeça: indefinido
{
            apaga desenho
            centraliza
            responde nada
}
```

a primeira linha declara que o tipo *Tartaruga* é implícito. Portanto, os dois método definidos em seguida serão incluídos nesse tipo. O segundo método, após a sintaxe do nome (*recomeça*), contém uma especificação do padrão do valor retornado, indicando que ele é do tipo *Objeto Indefinido.* Como nas variáveis, esta especificação inicia com dois-pontos. O literal *nada* representa o único objeto desse tipo.

Observe-se que o método retorna um objeto: *nada.* Isto significa que uma mensagem recomeça pode funcionar como sub-expressão de uma expressão maior. Contanto que as ações realizadas sobre ou com o resultado da recomeça sejam compatíveis com o tipo *Objeto Indefinido.*

As declarações de operações, no lugar da palavra-reservada *sentença,* usam *atribuição, conectivo, relação, operação aditiva, operação multiplicativa, operação unária* ou, simplesmente, *operação.* No caso de ser usada apenas *operação,* a operação definida terá prioridade de operação multiplicativa ou, se a sua sintaxe induzir, de operação unária. Um operador unário pode ter sido anteriormente definido como sendo de qualquer outro tipo de operador. Em caso de ambiguidade, que acontece quando se omite o receptor de uma operação binária, o operador será sempre considerado como unário. Portanto, *(-i)* é interpretada como o simétrico de *i* e não como o objeto corrente menos *i.*

4.4.12.5. Coerção

O objeto resultado de uma expressão pode ser forçado a ser visto conforme determinado padrão. Isto é chamado de **coerção.** Se o padrão da expressão for um subpadrão do padrão dado, e isto sempre pode ser determinado estaticamente, a coerção não falhará.

No trecho (62) a variável *empr* recebe, por indução, o padrão tipo *Empregado*. A expressão *empr:pessoa* faz a coerção de *empr* para o padrão *Pessoa*.

```
variável empr ⇒ empregado ‹nome ⇒ "Patrícia"›
sexo de empr:pessoa := f
```

Como este é um superpadrão de *Empregado*, esta coerção nunca falhará. O atributo *sexo* continuará sendo acessível, pois é definido no tipo *Pessoa*.

Por outro lado, se o padrão da expressão não for um subpadrão do padrão para qual ela está sendo forçada, a coerção poderá falhar dinamicamente.

A expressão (63) poderá causar, dinamicamente, uma exceção, pois nem todo empregado é gerente.

```
impressão de inicio_da_gerência de empr:gerente
```

Se isto não acontecer, o atributo *inicio_da_gerência*, específico do padrão *Gerente*, poderá ser consultado.

É muito comum se desejar testar se uma coerção pode ser feita. Para isso, uma forma de coerção seletiva existe na linguagem.

A expressão abaixo apresenta uma coerção seletiva. (64)

```
p:( peça base { val := custo }
    peça composta { val := custo_de_montagem }
     { val := 0 } )
```

A variável *p* é forçada ao padrão *Peça Base*.Se isto puder ser feito, o bloco seguinte será executado, tendo como objeto corrente *p* visto como uma peça base. Caso contrário, *p* é forçada ao padrão *Peça Composta*, com um bloco semelhante ao anterior, porém onde *p* é visto como uma peça composta. A última alternativa será executada caso nenhuma das coerções anteriores puder ser realizada. Neste caso, o bloco enxerga *p* como não tendo sofrido coerção alguma. Se esta última alternativa não for fornecida, uma exceção ocorrerá.

4.4.12.6. Blocos como Argumentos

Blocos - comandos entre chaves - são objetos e podem ser passados como argumentos de ações. A avaliação de um bloco resulta em um objeto.

Blocos diferentes podem resultar em objetos de tipos diferentes. Para que se possa definir métodos que aceitem blocos como argumentos, o tipo de objeto que o bloco resulta pode ser definido por uma expressão de tipo. Para que métodos com blocos como argumento sejam genéricos, isto é, que não dependam do tipo do valor resultante do bloco, este tipo pode ser omitido.

o seguinte cabeçalho de método (65)

```
condição sentença então (b1:bloco) \
                        senão (b2:padrão de b1): padrão de b1↑
```

define que objetos do tipo *condição*, respondem às mensagens *então { } senão { }*, onde as chaves representam blocos que são passados como argumentos. O primeiro argumento, *b1*, é restrito a ser um bloco. Todavia não foi especificado um padrão para a avaliação do resultado desse bloco. Isto permite que esta mensagem seja genérica, isto é, possa ser usada para blocos que retornam qualquer tipo de valor. Dentro desse método, a expressão *valor de b1*, que causa a avaliação do bloco passado como argumento, é restrita ao padrão *Objeto* e não pode ser usada como objeto de qualquer outro tipo.

O segundo argumento, *b2*, tem o mesmo padrão de *b1*, e, portanto, também é um bloco, que, porém, tem que resultar em um objeto em conformidade com o mesmo padrão do resultado de *b1*. O resultado desse método também tem que estar em conformidade com o padrão do resultado de *b1*. Isto significa que uma mensagem *então { } senão { }*, enviada a uma condição, resulta em um objeto em conformidade com o mesmo padrão do resultado do primeiro bloco-argumento. Além disso, o analisador pode acusar um erro caso o segundo bloco não esteja em conformidade com o primeiro.

A combinação $\approx$ *x* pode ser usada como uma abreviatura de *: padrão de x*. Isto permite que se reescreva o cabeçalho acima assim (66)

```
condição sentença então (b1:bloco) senão (b2≈ b1)≈ b1↑
```

Para se ter métodos genéricos, além dos padrões do resultado de blocos, devem-se poder parametrizar o padrão dos elementos de uma coleção, ou o padrão das instâncias de uma classe.

Um bloco usado como argumento de uma mensagem pode ter o seu objeto corrente redefinido. Isto tem que ser especificado nos cabeçalhos dos métodos correspondentes.

No seguinte cabeçalho (67)

```
coleção sentença faz (:bloco↓(isto↑)): indefinido
```

o termo *isto*↑ significa o padrão de um elemento do receptor (*isto*). A palavra *isto* poderia ser omitida, sendo usada apenas a seta entre parênteses. A seta descendente (↓) indica que o objeto corrente do bloco será o fornecido pela expressão de padrão que o segue. No exemplo, o objeto corrente do bloco passado como argumento terá o mesmo padrão dos elementos da coleção receptora. O método deverá retornar o objeto indefinido *nada*. Dentro do método, o nome do parâmetro, omitido no cabeçalho, será *bloco*, isto é, o nome do tipo fornecido para o parâmetro.

Com a especificação do padrão do objeto corrente de um bloco passado como argumento, a análise das expressões internas ao bloco pode ser feita corretamente. É importante notar que não há *overloading* da especificação do tipo do resultado de um bloco argumento, nem de seu objeto corrente. Ou seja, todos os métodos com a mesma assinatura, da qual estas especificações não fazem parte, têm que ter estas especificações idênticas.

O cabeçalho da operação de seleção é o exemplo seguinte (68)

```
coleção operação | (:bloco↑condição↓(↑))≈ isto
```

Esta é uma operação aplicável a objetos em conformidade com *Coleção* e necessita de um bloco passado como argumento. Este bloco deve avaliar uma condição que indicará se o elemento deve ser selecionado ou não). O padrão do elemento corrente do bloco será o mesmo do padrão dos elementos da coleção receptora. O padrão do resultado da operação de seleção será igual ao da coleção receptora. Novamente, a palavra *isto* poderia ter sido omitida.

Expressões de tipo de argumento podem se tornar bastante complicadas. Porém essas complicações ocorrem em um número pequeno de casos. A extensibilidade e a tipagem proporcionadas pela linguagem, graças a essas regras, compensam estas situações.

Um exemplo de expressão de tipo bastante complicado é encontrável no seguinte cabeçalho do método de injeção para coleções (69)

```
sentença injeta(val) \
em(:bloco↑padrão de val↓(tupla↑‹padrão de val;↑›)≈ val
```

Neste exemplo, o padrão de *val*, o primeiro parâmetro, não foi fornecido. O segundo argumento é um bloco que deve avaliar um objeto em conformidade com o mesmo tipo da expressão passada como primeiro argumento. O objeto corrente deste bloco será uma tupla cujo primeiro elemento será do mesmo padrão de *val* e o segundo será do mesmo padrão dos elementos da coleção receptora. O resultado do método também terá o mesmo tipo que *val*. A expressão (70)

```
cad injeta 0 em { λ«digs; carac»
                digs + ( (carac é dígito) então{1} senão{0} }
```

respeita todas essas restrições.

4.4.12.7. Variáveis Tipo

Objetos persistentes exigem que seus metaobjetos também persistam. Por uniformidade, tipos e outras espécies de metaobjeto são objetos no modelo de objetos do GEOTABA. É possível se ter, por exemplo, uma variável cujo conteúdo seja restrito a objetos tipo. O padrão dos objetos instanciados a partir do conteúdo dessa variável tem que ser determinado estaticamente. Se nenhuma indicação mais precisa for dada, através de uma expressão de tipo, o objeto instanciado será encarado como restrito ao padrão mais geral de todos: o padrão tipo *Objeto*. Nesse caso, o objeto instanciado só poderá ser manipulado pelas operações de *Objeto* ou ser alvo de uma coerção de padrão.

A expressão (71)

```
variável t:tipo↑proprietário_de_telefone
```

declara a variável *t* que conterá objetos tipos que instanciam objetos que podem ser manipulados conforme o padrão *proprietário_de_telefone*. Por exemplo, a expressão (72)

```
t< nome ⇒ "José"; fone ⇒ "555-1234" >
```

representa a criação de um objeto desses. A variável *t1* em (73)

```
variável t1:tipo
```

conterá qualquer tipo de tipo. Não há uma sentença de instanciação de objetos comum a todos os tipos. Portanto, *t1* não poderá ser usada para instanciar objetos. Apenas para ilustração, será considerado que *t1*<> instancia um objeto do tipo contido em *t1*. Este objeto só poderia ser manipulado pelas operações comuns a qualquer objeto. A expressão (74)

```
t1< >:(empregado{salário:=10.000,00};{})
```

permitiria atribuir um valor ao atributo *salário* do objeto instanciado, caso o tipo contido em *t1* gerasse objetos em conformidade com o tipo *Empregado*. Caso contrário, o bloco vazio seria executado. Se fosse esperado que o primeiro caso sempre ocorresse, a expressão poderia ser codificada assim (75)

```
t1< >:empregado{salário:=10.000,00}
```

Uma exceção ocorreria caso *t1* não gerasse objetos compatíveis com *Empregado*.

As mesmas regras válidas para variáveis metaobjetos podem ser usadas para se ter variáveis genéricas de coleção ou blocos.

4.5. O Nível de Implementação

Cada padrão tipo e aspecto definido no nível conceitual pode ter diversas implementações correspondentes no nível de implementação. Evidentemente, essas implementações têm que manter conformidade com os padrões tipos e aspectos que elas implementam, produzindo as mesmas restrições definidas no nível conceitual. Desse modo, a implementação de um tipo tem que gerar, por e-xemplo, o mesmo protocolo definido no padrão tipo.

Cada tipo conceitual definido produz, automaticamente, uma **classe** de implementação *default* correspondente. Esta classe pode ser alterada, contanto

que não fique incompatível com o seu tipo originário. Outras classes podem ser geradas, inclusive a partir desta, implementando o mesmo tipo. Semelhantemente, cada aspecto, definido no nível conceitual, gera uma **extensão** (de implementação de classe) *default*. Essa extensão também pode ser alterada, mantendo compatibildade com o aspecto. Outras extensões podem implementar o mesmo aspecto.

As classes também se organizam em uma hierarquia, conforme encontrado em qualquer linguagem de programação orientada a objetos. A hierarquia de tipos conceituais é distinta da hierarquia de classes de implementação. Isto significa que uma classe que implementa um tipo T não precisa ser obrigatoriamente subclasse das classes que implementam os supertipos de T. É suficiente que a classe e o seu tipo produzam as mesmas restrições.

As classes escondem as decisões de implementação de um tipo: a estrutura de implementação de suas instâncias e a implementação dos seus métodos. As extensões escondem, do mesmo modo, a implementação de um aspecto. O estado do objeto é o conteúdo de sua estrutura de dados interna, preenchida por referências a outros objetos através de suas identidades. Objetos triviais, como os inteiros e os valores booleanos, nem sequer possuem estrutura de dados; suas identidades representam inteiramente o objeto. Os métodos independentes de implementação, definidos no nível conceitual, fazem parte das classes, mas não podem ser alterados por elas.

O nível de implementação de um objeto define a sua estrutura e comportamento, que são ocultos para os demais objetos. A estrutura oculta dos objetos é composta de **campos**, equivalentes às variáveis de instâncias de outros modelos de objetos. O comportamento é uma coleção de métodos. Toda comunicação entre objetos é por troca de mensagens ou eventos. Um objeto pode ter os valores de seus campos alterados. Campos e métodos são os únicos recursos existentes no nível de implementação para representar as associações e serviços definidos no nível conceitual. Assim, atributos, componentes, relacionamentos, ações e restrições são todos mapeados para campos e/ou métodos. Este mapeamento é definido seguindo um conjunto de regras simples.

Um objeto pode não possuir campos. Estes objetos são os nós terminais da rede de objetos. O valor de um objeto, que possua campos, é composto de referências a outros objetos, que, por sua vez, se referem a outros objetos, até que chegam a objetos sem campos. O estado destes objetos não varia nem depende de outros objetos. Como exemplos de objetos invariáveis pode-se citar os números inteiros e os valores verdadeiro (*sim*), falso (*não*) e *indefinido*. Observe que um objeto qualquer pode ter um campo valendo verdadeiro ou

falso, por exemplo. Isto significa que ela estará fazendo referência a um desses objeto, porém estes são objetos invariáveis e terminais.

A maior parte das classes funciona também como geradora de objetos. Ou seja, a criação de objetos é feita enviando-se mensagens adequadas a classes. No nível conceitual, os usuários precisam ter conhecimento de quais classes implementam quais tipos. Assim, para poderem criar objetos de um tipo desejado, têm que enviar uma mensagem de criação de objeto para alguma das classes que implementam o tipo. Como cada tipo possui a sua classe *default*, a mensagem de criação pode ser também enviada ao próprio tipo desejado, pois será repassada à classe *default* correspondente.

Para se ter acesso a uma classe que implementa um tipo, envia-se ao tipo uma mensagem cujo nome é o próprio nome da classe desejada. Por exemplo, se o tipo *Empregado* possui uma classe de implementação *Moderno*, pode-se criar uma instância dessa classe assim: (76)

```
var empr ⇒ Empregado moderno ‹nome ⇒ "João"›
```

A mensagem *moderno* dá acesso a classe *Moderno* de *empregado*. A forma prefixa também é válida, como em (77)

```
(Moderno de Empregado) ‹nome ⇒ "João"›
```

As mesmas regras são válidas para extensões e aspectos. Para que o empregado referido pela variável *empr* se torne um gerente, mas usando a extensão (implementação do aspecto G*erente*) de nome *Esdrúxulo*, pode-se escrever (78)

```
empr gerente esdrúxulo ‹início⇒data‹13;5;89››
```

Em caso de ambiguidade, pode-se usar as palavras-reservadas *aspecto, classe, extensão,* etc., para qualificar um nome, como em: (79)

```
(aspecto gerente esdrúxulo de empr)
                ‹início⇒data‹13;5;89››
```

Neste caso, fica claro que *gerente* é um aspecto de *empr*, e não um atributo.

Comumente, um atributo definido no nível conceitual é mapeado por *default*, no nível de implementação, em um campo (variável de instância), mais um método de atribuição ao campo e outro método de informação do valor atual do campo. Um atributo constante não teria método de atribuição associado. Todavia, a implementação do atributo pode ser refeita manualmente, com seu valor sendo calculado usando-se outros campos.

4.5.1. Campo

Um campo funciona como um atributo interno do objeto, podendo ser consultado e atualizado. Diferentemente dos atributos, para campos, estas operações não correspondem a mensagens ou a eventos.

Mesmo não sendo mensagens, o acesso e a atribuição de valor a um campo, em DEMO, utilizam a mesma sintaxe que o acesso e a atribuição a atributos, que são feitos através de mensagens ou de eventos. É muito comum que uma implementação (classe ou extensão) defina campos com o mesmo nome que atributos do tipo ou aspecto a que se refere a implementação. Isto sempre ocorre, por exemplo, nas implementações *default.* Os qualificadores *atributo* e *campo* podem ser usados para evitar ambiguidades. Por exemplo, se *início* for um campo definido em uma classe que implementa um tipo que define um atributo de mesmo nome, em um método dessa classe poderá aparecer (80)

```
atributo início := início + 1
```

O segundo *início* se refere (pela ausência de qualificador) ao campo, pois campos têm escopo mais restrito que atributos. O objeto referido por esse campo, que é obtido diretamente do campo, sem necessidade de envio de mensagem alguma, receberá uma mensagem de soma. O resultado da soma servirá como argumento para uma mensagem de atribuição, já que o primeiro *início* é um atributo. A mensagem, cujo nome é algo como *início:=*, será enviada ao objeto receptor do método que contém essa expressão. O método que será executado por essa mensagem poderá realizar outras operações diferentes da simples atualização do campo *início.*

4.5.1.1. Campo Indexado

A definição total do nível de implementação do modelo não é o objetivo deste trabalho. Os conceitos definidos acima formam a base do nível de implementação. Outros conceitos podem e devem ser adicionados a esses. O fundamental é que neste nível o encapsulamento dos objetos seja total, de modo a que os objetivos de modularidade, manutenibilidade e reusabilidade das linguagens de programação orientadas a objetos seja alcançado. O não detalhamento permite que várias alternativas de modelo de implementação de objetos possam ser experimentadas no GEOTABA.

A implementação de coleções implica na existência de **campos indexados**, sem nome, mas acessíveis por um índice numérico, a exemplo dos vetores *("arrays")* das linguagens de programação tradicionais. O acesso a esses

campos passa por uma indexação, como nas coleções indexadas do nível conceitual. As restrições a que devem ser submetidos os índices e se os objetos com campos indexados podem ser aumentados ou diminuídos são alternativas em aberto. De fato, cada linguagem de programação aborda estas questões de modo particular. A tomada de partido não representaria qualquer avanço em relação as propostas deste trabalho.

4.5.2. Comportamento

4.5.2.1. Encapsulamento

Uma questão importante, envolvendo modelos de objetos, é o encapsulamento, que faz com que um objeto possa ser encarado como uma intância de um tipo abstrato de dados. A idéia é que a estrutura de um objeto seja inacessível para os outros objetos.

O encapsulamento é considerado por muitos como estando em contradição com o propósito de um sistema de gerência de dados. Se, exatamente, a estrutura desses dados é que precisa ser gerenciada pelo sistema e manipulada pelo usuário, esconder esta estrutura não parece ser uma atitude natural. O modelo de objetos do GEOTABA pressupõe encapsulamento por dois motivos. Primeiro, o seu objetivo é a gerência de objetos, isto é, comportamento + dados, e não apenas de dados. Em segundo, existe uma dualidade entre dados e processamento que permite que se veja o conteúdo de uma informação através do processamento de uma função. Mantendo a abstração de dados por ela ter se mostrado adequada à criação de sistemas complexos, os avanços na estruturação dos dados podem ser mapeados na estruturação do comportamento dos objetos.

O encapsulamento, no modelo de objetos do GEOTABA, é definido pelas regras seguintes. Os métodos independentes do nível de implementação só fazem acesso a literais, nomes globais, aos seus argumentos e aos atributos da estrutura percebida, do próprio objeto que contém o método, ou dos objetos alcançáveis a partir destes. A estrutura percebida de um objeto pode não ser igual a estrutura implementada deste. Na verdade, o acesso à estrutura percebida é realizado através de mensagens de consulta e atribuição que, pela sua aparência, imitam a consulta e a atribuição a variáveis nas linguagens convencionais. Um método dependente do nível de implementação tem a capacidade de, além dos acessos anteriores, fazer acesso aos campos da estrutura implementada do próprio objeto, e apenas dele. Todavia, a consulta e atribuição aos campos não é feita através de mensagens, embora tenham a mesma sintaxe destas.

Resumindo, no GEOTABA, o encapsulamento é total, como nas linguagens de programação orientadas a objetos puras, como Smalltalk e Actor. Um método só acessa as variáveis (campos) do próprio objeto receptor do método. Toda a comunicação com outros objetos é feita sob a forma de mensagens. Como boa parte das mensagens são consulta e atribuição a atributos conceituais dos objetos, pode-se preconizar uma estrutura percebida do objeto. Mantendo-se, para essas mensagens, a mesma sintaxe da atribuição e consulta às variáveis internas, tem-se a impressão que um método acessa a estrutura percebida de qualquer objeto. Porém, apenas os métodos, dependentes do nível de implementação, do próprio objeto têm acesso a seus campos da estrutura de implementação.

Por exemplo, em Smalltalk, um objeto representando um funcionário poderia ter uma variável de instância *cep*. Se for permitido, a outros objetos, consultar o CEP de um funcionário, este deverá possuir um método *cep* para implementar esta consulta. Se este CEP puder ser alterado por outros, haverá também um método *cep:* para realizar a alteração do CEP. Dentro dos métodos de um funcionário, uma atribuição à variável de instância *cep* é realizada através de uma atribuição tradicional: '*cep := x*'. A alteração do CEP de um funcionário *f* é feita por uma expressão do tipo '*f cep: zzz*'. A diferença de sintaxe e a predominância da implementação sobre a conceituação não propiciam a percepção de que o CEP é um atributo percebido de um funcionário.

Já em DEMO, uma linguagem usável no modelo de objetos do GEOTABA, o tipo *Funcionário* definiria *CEP* como um atributo (conceitual) de funcionários. A implementação *default* deste tipo geraria uma classe com um campo *CEP*, um método de consulta também chamado de *CEP* e um método de atribuição *CEP :=*. Dado um funcionário *f*, a consulta de seu CEP seria feita por uma mensagem '*CEP de f*', ou '*f CEP*'. A alteração do CEP seria '*CEP de f := zzz*' ou '*f CEP := zzz*'. Dentro de um método do nível de implementação de *Funcionário*, uma atribuição ao campo *CEP* seria '*CEP := zzz*'. Fica claro, por esta sintaxe, que CEP é um atributo de funcionários. Se o atributo fosse constante, seria possível se ter uma classe implementando o tipo *Funcionário* sem usar um campo *CEP* na sua estrutura de dados, mas definindo que o método *CEP*, que responde o valor deste atributo, fizesse uma consulta a uma tabela, onde dado o endereço do funcionário fosse encontrado o seu CEP.

4.5.3. Classes e Extensões

Como cada objeto representa uma estrutura de dados encapsulada, se todos os objetos, além do seu estado, tivessem que incorporar a descrição de sua estrutura e o código para o processamento do seu comportamento, se teria

um sistema inviável de ser utilizado e mesmo de ser implementado. Os mecanismos de classificação e herança viabilizam o modelo.

Classes são objetos especiais, descritos no nível de implementação, que colecionam propriedades comuns a grupos de objetos. A classe define uma estrutura de dados e um conjunto de métodos para esses objetos. Através das classes pode-se criar novas instâncias de objetos. Depois de criado, um objeto pode adquirir novos campos e métodos específicos, não presentes na classe pelo qual foi criado, não de modo semelhante ao sistema O_2 [BANC88] (ver 4.4.3.5.:"Coleções"), mas através de aspectos [RICH91] (ver 4.4.3.6.:"Aspectos"). Um aspecto reúne campos e métodos adicionais. Todo objeto é descrito por um metaobjeto. Classes e Metaclasses são tipos específicos de metaobjetos.

Um objeto não precisa registrar a sua estrutura e o seu comportamento. Basta conhecer o seu metaobjeto, onde eles estão especificados. Ao objeto resta os valores de seus campos. Quando um objeto acaba de ser instanciado por uma classe, seu metaobjeto é essa classe, que descreve totalmente sua estrutura e comportamento. Se, a seguir, esse objeto adquirir um aspecto, através da incorporação de uma extensão, a classe não será mais o metaobjeto desse objeto, por não mais descrever totalmente o objeto. O novo metaobjeto descreverá que objeto possui a estrutura e comportamento descritos no seu antigo metaobjeto (a classe) estendidos pela descrição contida na extensão. A gerência dos metaobjetos deve ser realizada pelo sistema e ser invisível a todos os usuários comuns.

Uma possível solução de implementação da incorporação de uma extensão a um objeto acrescenta um campo a ele. Este campo contém a identidade de um objeto que, por sua vez, é quem contém os campos (e comportamento) da extensão. Evidentemente, a identidade deste objeto-extensão não deve ser visível para usuários comuns.

A especificação da estrutura engloba a definição do número de campos que os objetos desse metaobjeto devem ter, quais os identificadores e possíveis restrições e propriedades desses campos. Em um modelo simples, como o da linguagem MANO, descrita mais adiante, esta especificação consistiria apenas da lista dos identificadores das variáveis com nome dos objetos e da indicação sobre se esses objetos têm variáveis indexadas ou não.

4.5.3.1. Instâncias

A função de uma classe é esquematizar os seus objetos. Uma classe, todavia, não deve ser encarada como um repositório ou coleção destes. Uma classe é um esquema ou molde pelo qual objetos são criados e interpretados se-

mânticamente. Um objeto não deve ser encarado como membro pertencente a uma classe e sim como uma instância ou exemplar desta classe. Coleções, por sua vez, são mecanismos de reunião de objetos que não acrescentam diretamente semântica a seus membros. As consultas são realizadas comumente sobre as coleções de objetos, não sobre as classes de objetos. Esta separação de papéis não é comumente encontrada em modelos de dados como o relacional, onde o esquema de uma relação define uma coleção e o tipo de seus membros.

4.5.3.2. Metaclasse

Uniformizando o modelo, uma classe deve ser vista também como sendo um objeto. Isto significa que uma classe tem um identificador de objeto, tem um estado e também uma classe. A classe de uma classe é um tipo de classe diferente, denominado metaclasse.

4.5.4. Herança

Um nível adicional de abstração é alcançado através da herança de propriedades entre classes. Propriedades comuns entre classes são fatoradas em classes generalizadas. Uma classe pode, então, ter uma parte de suas propriedades definida em outra classe. Esta será a superclasse daquela. Qualquer classe poderá ser usada como superclasse de novas classes que serão especializações da superclasse. A classe especializada é chamada de subclasse daquela genérica. Uma superclasse pode realmente ser uma classe, possuindo instâncias, ou ser uma pseudoclasse, usada apenas para a definição de subclasses.

O relacionamento de generalização/especialização entre classes define que a subclasse herda as propriedades definidas na superclasse. Se a subclasse, por sua vez, define propriedades específicas que conflitam com as suas propriedades herdadas, as propriedades específicas têm ascendência sobre as herdadas, anulando a herança de propriedades conflitantes.

A hierarquia das classes é independente da hierarquia de tipos, a menos de restrições que o programador impõe relacionando tipos a classes e vice-versa. Todo tipo tem uma classe que o implementa por *default*. Esta classe nunca poderá ser incompatível com o tipo que a originou. Do mesmo modo, uma classe pode ser criada com uma especificação de que ela implementa um dado tipo. Também nesse caso, a compatibilidade tem que ser mantida. Se uma classe mudar de tipo, ou um tipo for alterado, todos os métodos que usarem-nos deverão ser recompilados. A separação das hierarquias, atualmente já

encontrável em poucos modelos, resolve questões em aberto quanto à independência entre o projeto conceitual e lógico.

É possível, e usável, ter-se classes que não implementam tipos. Estas só têm significado, como entidade, no nível de implementação, sendo inexistente para o nível conceitual e para os demais níveis. Não parece ser razoável que essas classes possam criar objetos. Ou seja, são pseudoclasses.

Alguns modelos permitem que uma classe aceite mais de uma superclasse. Neste caso é dito que há uma herança múltipla. Aí as propriedades herdadas de classes diferentes podem conflitar entre si. Cabe ao modelo especificar uma política, ou fornecer mecanismos, para a resolução de conflitos deste tipo. A herança múltipla, no nível de implementação, e seus conflitos não estão estabelecidos por esta definição do modelo.

Toda classe deve ter alguma superclasse. Exceto uma, a classe Objeto, que funciona como a raiz de uma hierarquia de classes. A classe Objeto contém as propriedades comuns a todos os objetos do sistema.

Evidentemente, a hierarquia de classes não pode ter ciclos, onde uma classe herda propriedades de outra classe, que, por sua vez, herda propriedades da primeira. A hierarquia de classes define um grafo direcionado acíclico, que, nos modelos limitados a herança simples, onde cada classe possui apenas uma superclasse, tem a forma de uma árvore.

Um dos objetivos de se representar objetos através de uma hierarquia semântica de classes é possibilitar a definição de novas classes de objetos a partir da especialização de classes de objetos já existentes. Isto é feito através de programação diferencial e é uma estratégia importante para a reutilização de código.

Uma classe, além de definir a estrutura e o comportamento de suas instâncias e de definir o relacionamento de herança entre as suas superclasses e entre as suas subclasses, pode estar associada a um nome. Este é um símbolo que identifica a classe. Esta associação (nome, classe) é vista como sendo um nome global, visível por todos os objetos.

Em aplicações típicas de banco de dados, entidades (tuplas) são criadas dinamicamente sem que seus esquemas tenham sido definidos previamente. É o caso quando se juntam informações de duas entidades diferentes, mas que tenham um relacionamento, e se gera uma nova informação que não é nenhuma entidade ou relacionamento previsto no esquema. No modelo relacional, isto ocorre quando há junções ou projeções. Uma relação projetada em alguns atributos é, quase sempre, um resultado temporário que não altera o esquema original. Para se ter a mesma flexibilidade para se gerar objetos não previstos no

esquema permanente, o Modelo de Objetos do GEOTABA, admite objetos não descritos diretamente por classes, mas por outros tipos de metaobjetos.

4.5.5. Armazenamento

Um grande grupo de classes implementa o armazenamento de objetos e coleções em memória secundária. O nível de implementação de objetos armazenados trata também, normalmente herdando propriedades dessas classes, da estrutura de dados em disco dos objetos e coleções armazenados e do mapeamento de/para a estrutura de dados do objeto na memória. Desse modo, a implementação de objetos na memória e em disco é totalmente extensível.

4.5.6. Restrições

Ao se especificar um campo de objeto define-se o padrão do campo, este pode restringir o tipo de objeto que o campo poderá conter. Padrões e tipos de objetos foram discutidos em 4.4.3.:"Restrições e Modelagem Conceitual". Argumentos e variáveis locais de métodos são submetidos a padrões.

4.6. O Nível Externo

No nível externo, os objetos são descritos em termos das perspectivas pelas quais podem ser vistos. Em grande parte dos modelos de objetos existentes, como no caso do Smalltalk, o protocolo definido por uma classe é totalmente visível pelos usuários desta classe. Alguns outros modelos ajustam a visibilidade a dois níveis, definindo que certos métodos são privativos. Um método privativo pode significar que não é acessível por outras classes. Esta restrição é pouco adequada, pois comumente precisa-se definir métodos usáveis por um conjunto de classes, mas de uso restrito a indivíduos qualificados. A solução sugerida no modelo é a possibilidade de se definir vistas de um objeto. Vistas, como em banco de dados relacionais, são janelas que permitem ver partes do objeto.

4.6.1. Vistas

Um primeiro problema a ser tratado por visões particulares dos objetos se refere a quando determinados atributos de um objeto não devem ser acessíveis a alguns indivíduos. Por exemplo, o atributo *salário* de um empregado não deve ser visto por pessoas não autorizadas. Cada uma delas poderia só ter acesso a empregados através de um padrão que não contivesse este atributo.

Todavia, se esses usuários, apesar de não poderem manipular salários, puderem criar novos empregados, isto não poderia ser realizado através de um simples padrão. Como os padrões não são associados a classes de implementação, eles não têm a capacidade de responder mensagens de criação de objetos. Também não seria apropriado se criar um supertipo *Empregado'* de *Empregado*, pois um objeto empregado', não possuindo o atributo, não poderia ter o salário designado por um usuário autorizado.

Se, por outro lado, fosse possível se definir um tipo de aspecto para empregados, onde se pudesse inibir determinados atributos, o problema estaria resolvido. Esta foi a solução adotada, definindo-se um tipo de aspecto, denominado de **vista**, capaz de manipular a visão conceitual do objeto. Sendo um aspecto, uma vista não é um objeto em si, mas uma outra maneira de se ver um mesmo objeto. Tanto a vista quanto o objeto compartilham a mesma identidade. Se um objeto, sendo percebido através de uma vista, for atribuído a um atributo ou acrescentado a uma coleção, o objeto em si é que terá sido atribuído ou acrescentado, e não a sua vista. Observe-se que se o atributo ou a coleção também estiverem restritos a mesma vista que o objeto, os seus atributos secretos continuarão escondidos, exceto para os usuários que podem usar este atributo ou coleção sem restringir a vista.

Uma vista pode inibir atributos e serviços do objeto. Além disso ela pode definir novos atributos e serviços que sejam implementados por métodos independentes do nível de implementação. Sendo assim, uma vista pode permitir que um objeto seja percebido como possuindo um atributo que, na verdade, foi calculado em função dos seus atributos reais. Por exemplo, pode-se ter uma vista de empregados em que aparece um atributo 'nome do departamento' do empregado, calculado a partir do atributo *departamento* do empregado. Uma vista também pode possuir métodos próprios para a criação de novos objetos, definindo valores *defaults* para os atributos ocultos.

A cada vista corresponde um padrão representando as restrições do objeto visíveis através dela, como, por exemplo, o protocolo (atributos e serviços) e ligações que ela permite acessar. Em função disso, a especificação de uma vista se dá pela especificação de um **padrão vista**. Este, de modo semelhante aos padrões tipos, define que, através desta vista, os objetos só podem ser manipulados de acordo com as propriedades definidas no padrão.

Uma vista é definida sobre um tipo (nível conceitual) de objeto ou sobre outra vista. Uma vista fornece um novo protocolo (atributos e serviços), definindo um novo padrão, construído a partir do protocolo do tipo (ou vista) base, excluindo atributos e serviços ou acrescentando novos serviços e atributos derivados do protocolo base.

Como uma vista é um aspecto, ela pode ser colocada em qualquer lugar onde um aspecto pode aparecer. Assim, pode-se ter vistas sob: um tipo, indicando que objetos deste tipo podem ser vistos conforme esta vista; um aspecto, modificando a visão de objetos que possuem este aspecto; ou uma outra vista, permitindo que se defina vistas a partir de outras vistas. Entretanto, um padrão aspecto não pode ser colocado em uma vista, pois teríamos o nível conceitual definido em função do nível externo.

Quando um padrão vista é colocado sob um padrão tipo, aspecto ou vista, cada aspecto do padrão original é copiado para a vista, a menos que ela explicitamente o desative. Por exemplo, se o padrão tipo *Empregado* possuir um aspecto *Gerente*, um padrão vista *Empregado'*, que inibisse o acesso ao salário do empregado, seria colocado sob *Empregado*. Naturalmente, esta vista permitiria que um empregado visto por ela pudesse ter um aspecto de gerente. Evidentemente, o salário dos gerentes, como o dos demais empregados, não estaria acessível. Assim como a vista inibiu o atributo *salário*, ela poderia proibir o aspecto *Gerente*. Assim, por esta vista, não se acessaria o salário dos empregados, nem os atributos específicos dos empregados que possuissem o aspecto de ser gerente.

No lugar de inibir o aspecto *Gerente*, caso este possua uma vista *Gerente'* própria, a vista *Empregado'* poderia restringir que o aspecto *Gerente*, sob ela, fosse visto por *Gerente'*. Com estas regras, seria efetivo o casamento entre aspectos, que permitem a evolução dinâmica dos objetos, e vistas, criando visões particulares do mesmo objeto. Também é importante observar que, ao contrário dos aspectos, a definição de vistas não afeta o nível de implementação, pois ela se baseia totalmente nos níveis conceitual e externo. O mapeamento das entidades do nível externo em entidades conceituais é completamente controlado pelo sistema, não podendo o usuário modificá-lo.

Vistas acrescentam ou removem atributos, porém estes são apenas um determinado tipo de associação. Coleções podem possuir outros tipos de associações. Outro problema tratado por visões particulares dos objetos é o de que determinadas associações de uma coleção não devem ser acessíveis a alguns indivíduos. Por exemplo, empregados com salário maior do que um dado valor não devem ser vistos por pessoas não autorizadas.

Esta situação também não é resolvida pelo uso de padrões. Por exemplo, a uma variável restrita a um padrão para coleção de empregados com salário menor que *sal,* não poderia ser atribuída uma coleção que já não possuísse esta restrição.

O nível externo permite que se crie uma vista, sobre a coleção de empregados, que só mostre os empregados permitidos. Empregados poderiam

ser acrescentados ou removidos por essa vista, causando acréscimo ou remoção na coleção base. Evidentemente, empregados não visíveis não poderiam ser removidos. Similarmente, não poderia ser acrescentado um empregado ganhando mais do que o salário limite da vista, pois a vista, mas não a coleção base, possuiria uma restrição quanto ao salário dos seus empregados.

Como a base de objetos também é um objeto, onde os nomes globais são os seus atributos, podem-se criar vistas sobre ela. Com isto, determinados objetos e coleções globais podem ser ocultados e novos objetos globais virtuais podem ser criados. Assim, uma vista da base pode conter uma coleção virtual que, na verdade, seria a união de duas coleções globais da base, por exemplo. Isto seria especificado como um método, calculando esta união, pertencente à vista da base. Toda a consulta a esta coleção virtual seria equivalente ao envio da mensagem correspondente para a base, que responderia a união desejada. Opcionalmente, poderia ser especificado um método que permitisse a adição de objetos a esta coleção virtual, distribuindo-os, de fato, nas coleções base. Uma vista sobre a base é também chamada de um **subesquema**.

Vistas atuam apenas na análise (compilação) dos métodos e consultas. Um objeto é visto sob uma vista apenas pelo método ou pela consulta. O objeto não mantém em si nenhum vestígio de que esteja sendo visto por essas ou aquelas vistas. Um método descrito em uma vista tem prioridade sobre os métodos do objeto; isto significa que o acoplamento de um método de vista pode ser resolvido estaticamente, ao contrário dos métodos do objeto.

Se um tipo T1, que permite uma vista V1, tem um subtipo T2, que, por sua vez, pode ser visto por V2, métodos de V2 podem redefinir métodos de V1. Isto sugere que se possa ter métodos virtuais, como em C++, em vistas de pseudotipos, que seriam concretizados em vistas de subtipos do pseudotipo. Isto exigiria acoplamento dinâmico. Este assunto poderá ser no futuro matéria de estudo mais aprofundado.

4.7. O Nível de Representação

O sistema de objetos do TABA deve ser responsável por todos os objetos de software presentes na estação de trabalho. Não só objetos armazenados em bases em disco, como também objetos temporários, existentes apenas em memória principal, e objetos para interfaceamento com os usuários.

O nível de representação do modelo de objetos do GEOTABA inclui as possíveis formas nas quais objetos são apresentados ao usuário e por ele manipulado. De uma maneira geral um objeto pode ser apresentado no formato de ícones, formulários, tabelas ou outros [GIBB83]. Interfaces com manipulação

direta (ver 2.2.7.6.:"Comunicação Homem-Computador com Manipulação Direta") permitem que o usuário veja a informação desejada representada na tela e altere essa informação através de manipulação de sua representação [MONT92a]. Desse modo, cada objeto pode ter formas próprias de representação nos meios de interação com o usuário (tela, impressora, etc.). Estas representações devem simular os objetos base no seu comportamento típico.

Cada apresentação de um objeto é realizada através de um **representante** dele. Um representante é um outro objeto que descreve uma maneira de como o objeto base deve ser apresentado e como o usuário pode manipular essa apresentação. Representantes são objetos distintos dos objetos base, com características próprias e implementados por classes totalmente diferentes. Por exemplo, um estudante pode estar sendo visto sob a forma de um quadro contendo um gráfico de barras comparando as suas notas em diversas matérias. Evidentemente este quadro não é o estudante, porém, para o usuário em uma dada aplicação, ele pode ser o real representante do estudante. Alterar a altura de uma barra do gráfico pode significar, nesta aplicação, alterar realmente a nota do estudante.

Um mesmo objeto pode ser associado dinamicamente a diversos representantes, semelhantes ou não, proporcionando múltiplas visões simultâneas do objeto. Por exemplo, um estudante pode estar sendo representado simultaneamente em uma janela sob a forma de uma ficha com os dados do estudante e em outra janela como uma linha em uma tabela. Novos tipos de representantes podem estar sempre sendo criados e alterados dinamicamente.

Embora o representante seja um objeto distinto do objeto representado, o usuário deve ser levado a "sentir" o representante como sendo o objeto representado. O representante exibe, de alguma maneira, atributos do objeto base e pode permitir alterações nesses atributos ou a execução de serviços do objeto base. Por exemplo, uma ficha, representando na tela um objeto empregado, apresentaria cada atributo do empregado sob a forma de um campo de formulário, preenchido com o valor desse atributo. Na verdade, cada campo deste seria um representante do objeto contido no atributo. Alterações, feitas pelo usuário, no conteúdo desses campos, após serem consistidas e tratadas pelo representante, seriam refletidas no empregado base. Serviços do empregado seriam oferecidos ao usuário, por exemplo, através de menus. É comum que o representante esteja representando uma vista do objeto base (ver 4.6.1.:"Vistas"), e não o objeto conforme visto conceitualmente. Com os representantes, as propriedades estruturais (atributos, componentes e relacionamentos) de um objeto passam a ter uma forma visual, enquanto que as

propriedades comportamentais (os serviços) passam a ser vistas como possíveis operações que o usuário pode executar sobre a representação do objeto.

Além das propriedades do objeto base, o representante pode ter atributos e serviços próprios, manipuláveis pelo usuário. Por exemplo, um dicionário de dados de um projeto de software pode ser representado por um conjunto de fichas e diagramas. A operação de *zoom* na visualização de um diagrama afeta apenas o representante, não alterando o dicionário de dados representado. Um representante também possui tratadores para eventos, em geral originados pelo usuário.

Uma alteração em um atributo do objeto base deve ser transmitida imedatamente a todos os representantes ativos deste objeto. O programador comum não deve precisar se preocupar em gerenciar essa atividade. Do mesmo modo, alterações nos atributos do objeto base, feitas através do representante, devem ser transmitidas ao objeto base. Serviços disparados através do representante devem ser transmitidos ao objeto base.

Um representante pode ser ativado apenas para exibição do objeto, não permitindo alterações. A abertura de um representante que permita alterações pode causar bloqueios de escrita, para outros usuários, nos objetos visualizados por ele (o objeto base, seus atributos, atributos dos atributos, etc.). Esta questão deverá ser melhor desenvolvida em trabalhos futuros.

4.7.1. Tipo Representante

A descrição da representação de um objeto é relacionada com a sua descrição conceitual ou externa. Assim, as propriedades estruturais (atributos, componentes e relacionamentos) passam a ter uma forma no meio de apresentação (vídeo, impressora), enquanto que algumas propriedades comportamentais (os serviços) passam a ser vistas como possíveis operações que o usuário pode executar sobre o objeto.

Representantes são instanciados através de **tipos representantes**. A definição de cada um destes designa um padrão (comumente um padrão tipo ou padrão vista) que será representado. O protocolo do tipo base é importado pelo tipo representante. Portanto, se o tipo base define um atributo *CEP*, o representante também "possuirá" este atributo. Todavia, qualquer referência ou atribuição ao *CEP* do representante será, na realidade, canalizada para o objeto base. Fora estes atributos e serviços importados, um tipo representante pode especificar atributos e serviços próprios e tratadores dos eventos que ele se dispõe a interpretar.

O padrão tipo *Objeto* possui tipos representantes predefinidos que podem ser usados para interação com qualquer objeto. Em geral, subclasses de *Objeto* e outros metaobjetos redefinem o representante, de modo a fornecer informações apropriadas. O mais simples deles é *impressão* (usado como exemplo na expressão 15 na página 103). Este representante imprime o objeto em uma forma textual adequada a impressoras. O método *default*, ou seja, da classe *Objeto*, apenas imprime o nome da classe que implementa o objeto. O representante *tupla* descreve os atributos do objeto em um formato semelhante ao de uma tupla (por exemplo, *tupla de empr* mostraria no vídeo algo como *nome=Luiz; fone=222-2222)*. O representante *linha* descreve os atributos de um objeto em um formato apropriado para fazer parte de uma tabela. A ativação deste representante exige um argumento que descreve parâmetros visuais da tabela. O representante *ficha* descreve um objeto como uma ficha, onde cada atributo é um campo da ficha.

No lugar de simplesmente repassar mensagens, um tipo representante pode redefinir um atributo ou serviço do tipo base, de modo a realizar algumas ações além desse repasse. Estas ações extras podem ser especificadas para serem executadas antes ou depois do repasse. Portanto, se o representante importou um atributo *CEP*, pode-se definir ações a serem executadas antes da consulta ao *CEP*, para depois da consulta, antes da atribuição e para depois da atribuição. Muitas vezes, essas ações especificarão consistências a serem realizadas em valores entrados nos campos ou, então, alguma alteração na aparência do representante para mantê-lo de acordo com o desejado pelo usuário.

Os representantes foram definidos no modelo de objetos do GEOTABA para tratar, principalmente, o problema da atualização imediata da interface com usuário. Alguns pontos ainda permanecem em aberto. Nem sempre o desejado é a atualização imediata. É comum que, antes que os atributos de um objeto sejam alterados, seja desejável uma consistência global, com respeito aos dados entrados pelo usuário que alterarão esse objeto. Nesse caso, o objeto base não deve ser atualizado a cada operação do usuário. Um objeto **anteparo**, réplica do objeto base, pode servir de *buffer* para os valores entrados pelo usuário. Uma alternativa seria o anteparo e o representante serem o mesmo objeto. Isto significaria que os atributos importados do tipo base seriam transformados em campos reais do representante. Nesse caso, deveria ser permitido que o repasse das ações do usuário fosse inibido. A pesquisa em torno desse problema vale a pena, pois esta é uma situação muito encontrada e que sempre tem sido tratada de uma maneira procedimental.

Também é comum, embora frequentemente por motivo de eficiência, que atualizações nos objetos representados não se reflitam imediatamente nos objetos representantes, sendo retardadas até que o objeto base envie aos seus

representantes um comando *atualize*[25]. Isto assume uma importância grande no modelo de objetos do GEOTABA no caso dos atributos derivados. De fato, nos níveis conceitual e externo, a recuperação de valores é feita sob demanda, através de operações de consulta. Isto significa que, se o valor de um atributo derivado for alterado pela alteração dos valores base para o seu cálculo, isto só será percebido quando da próxima vez que este atributo for consultado. Nos sistemas tradicionais, isto não apresenta qualquer problema. Entretanto, a atualização imediata de representantes de objetos com atributos derivados será problemática. Existem possibilidades de implementação que resolvem este problema, porém elas talvez sejam muito custosas. Uma "quase" atualização imediata pode ser satisfatória na maioria dos casos. Basta ver como é comum editores gráficos ou de diagramas que possuem um comando *atualize* ou *redesenhe*[26] para refazer a tela a partir dos dados reais. De qualquer modo, esta é uma área a ser melhor desenvolvida.

Modificações em um objeto são comunicadas aos seus representantes ativos. Se o registro de quais representantes de um objeto estão ativos for feito no interior do objeto, isto é, usando um campo na sua estrutura implementada, todo objeto deverá possuir esse campo. Pior ainda, esta informação é altamente transitória. Em virtude disso, a observação de um objeto através de um representante, assim como ocorre através de uma vista, não causa alterações no objeto em si. Portanto, de alguma maneira, o sistema deve conhecer que representantes estão ativos para cada objeto, por exemplo, mantendo uma tabela de ‹objetos ; representantes›. Esta solução não é orientada a objetos, porém é usada em situações semelhantes até em sistemas orientados a objetos puros, como o Smalltalk.

Quando um (ou mais) representante(s) é(são) colocado(s) sobre um objeto, cada método de atribuição deste deve ser alterado para passar a enviar comunicação ao(s) representante(s). A maneira mais prática disso ser realizado é trocar o metaobjeto do objeto. Este novo metaobjeto indica que as atribuições avisam os representantes, ou seja, os valores dos atributos passam a ser ativos [MYER88]. Fazendo referência ao antigo metaobjeto, o novo não necessita repetir as demais operações. Quando o último representante do objeto for fechado, o que pode ser descoberto pela tabela mencionada, o metaobjeto antigo deve ser retomado.

Um novo ponto em aberto existe quanto a existência de implementações distintas para um mesmo tipo representante. Como um representante é um obje-

---

[25] *update*, em inglês.

[26] *redraw*, em inglês.

to ele possui um nível conceitual e um nível de implementação. É possível que um tipo representante para um dado padrão (exemplo: o tipo *Empregado*) possua diferentes implementações, bastando que elas implementem o mesmo tipo, ou seja, definam os mesmos atributos e serviços. Como a maior parte destes são importados do padrão base, esta situação pode ser comum. A separação entre os níveis conceitual e de implementação dos representantes também é útil na montagem de representantes específicos, graças a independência entre as hierarquias dos dois níveis. No nível de implementação, pode-se compor o representante através da herança de classes de objetos gráficos (dispositivos de interação). O sistema Chiron [YOUN88] também desmembra em dois níveis os seus "objetos" de interface, deixando para o nível mais baixo a tarefa de exposição gráfica do objeto[27] e para o nível mais alto a estrutura lógica do objeto de interface.

4.7.2. Interador

Um representante, além de encapsular as decisões de como representar um tipo de objeto, deve definir como o usuário pode manipular esta representação. Uma mesma representação pode ser manipulada por diversas técnicas de interação. Por exemplo, um gráfico de barras pode ser alterado clicando um botão do mouse ou, se o projetista da comunicação homem-computador assim preferir, movendo com o mouse a borda superior de uma barra. Cada representação deve ser associada dinamicamente a um modelo de interação. Este é ditado por um objeto **interador**.

Um padrão pode ter diversos tipos representantes e um objeto em conformidade com esse padrão pode ser associado dinamicamente a diversos representantes. Um tipo representante define uma apresentação para os objetos do padrão associado, os interadores que podem atuar nos seus representantes e qual deles será o interador *default*. Ao ser ativado um representante, um interador é especificado (ou o default será escolhido).

A definição de um interador especifica quais eventos ele trata e qual o tratamento de cada evento. Existem eventos de baixo-nível, tais como um clique do *mouse* em qualquer posição na tela ou uma tecla digitada pelo usuário, eventos de mais alto-nível, como escolha de um item em um menu ou alterações na barra de rolamento[28], e eventos definidos por programadores comuns,

---

[27] *rendering*, conforme usado em [YOUN88].

[28] *scroll bar*, em inglês.

gerados por qualquer objeto. O protocolo de eventos que um interador sabe tratar é a sua principal característica.

A idéia básica dos representantes vem dos artistas [MONT93], conforme usados no ambiente de desenvolvimento de software do projeto Arcadia-1 [YOUN88] (ver 3.2.3.3.8.7.:"Orientação a Acessos"). O interadores são semelhantes aos dispatchers ou controllers do Smalltalk [MONT93].

4.8. Sumário

Este capítulo definiu o modelo de objetos do GEOTABA (MOO), englobando regras de definição de estruturas e de procedimentos de manipulação da informação, além de regras de definição da representação para o usuário e de técnicas de interação. O modelo de objetos do GEOTABA é baseado no encapsulamento de objetos em quatro níveis de abstração: implementação, conceitual, externo e de representação. Cada objeto é encarado de modo diferente em cada um destes níveis. O nível de implementação (ver 4.5.) descreve estruturas de dados e métodos dos objetos conforme implementados. O nível conceitual (ver 4.4.) trata da estrutura e comportamento dos objetos conforme percebidos pela comunidade de usuários e pelos outros objetos. Há mapeamento automático do nível conceitual para o nível de implementação, onde alterações posteriores poderão ser realizadas. O nível externo (ver 4.6.) particulariza visões específicas para grupos de usuários criando objetos virtuais. O nível de representação (ver 4.7.) de um objeto refere-se a como este pode ser exibido e manipulado, nos meios de interface com o usuário.

No modelo de objetos MOO, toda a informação é representada por objetos. Com isso, o modelo ganha em uniformidade, o que aumenta a facilidade de uso, e em extensibilidade. Toda ação é realizada de um objeto sobre outro. Não existem procedimentos que não façam parte de algum objeto.

O nível conceitual define a estrutura percebida e os serviços fornecidos pelo objeto. O nível conceitual também define restrições que devem ser obedecidas pela estrutura e serviços do objeto. A estrutura percebida de um objeto compõe-se de associações. Uma associação com chave dá ao objeto a capacidade de reagir a mensagens de consulta e de atribuição. Uma associação cuja chave é um signo é chamada de atributo do objeto. Um objeto com número variável de associações é chamado de coleção.

Padrões, tipos, ligações e aspectos formam o ferramental básico para a modelagem conceitual da informação no modelo de objetos. Um padrão (ver 4.4.3.1.) representa um conjunto de restrições. Quando um objeto atende as restrições de um padrão, é dito que está em conformidade com o padrão. Pode-se

criar especializações ou generalizações de padrões. Como os objetos no sistema são acessíveis através de atributos e de ações sujeitos a restrições de padrão, padrões fornecem maneiras de se perceber, no nível conceitual, um objeto implementado.

Todo objeto possui informação que o torna capaz de responder mensagens que questionam se ele é de um dado tipo (ver 4.4.3.2.). Um padrão tipo (ver 4.4.3.3.) define, no nível conceitual, a existência de um tipo e lista as restrições que todos os objetos desse tipo atendem por implementação. Padrões tipos podem ser subtipos ou supertipos de outros padrões tipos, especificando, entre eles, herança de propriedades. Um tipo pode acrescentar novas restrições a uma propriedade de um supertipo, porém não pode contradizê-la.

Atributos que indicam que os objetos associados por eles são partes do objeto são definidos como componentes. A composição indica que um objeto pode ser referenciado em associações de outros objetos; porém não pode ser componente de nenhum outro objeto.

Um padrão pode definir uma associação a uma coleção. Por esta maneira, é possível modelar atributos multivalorados e relacionamentos 1:n ou n:m. O padrão de uma associação com coleção pode possuir restrições definindo as cardinalidades mínima e máxima da coleção.

Um aspecto (ver 4.4.3.6.) estende um objeto existente com novas associações e novos serviços, conservando a identidade própria do objeto. Um objeto pode dinamicamente adquirir múltiplos aspectos e se liberar deles. Um padrão aspecto define as propriedades de um aspecto. Aspectos são a maneira adequada de se modelar variações de tipo que não sejam excludentes entre si. Um padrão aspecto pode ser usado como base para a definição de novos subaspectos. Um objeto pode ter diversos aspectos do mesmo "tipo".

Uma **ligação** (ver 4.4.3.7.) vincula um atributo em um objeto a um atributo em outro objeto. Cada uma desses atributos é um **mapeamento**. O atributo da outra extremidade da ligação de um atributo é chamado de **mapeamento inverso** deste. Uma ligação pode possuir atributos.

Sendo um modelo extensível, além dessas ferramentas básicas, também foram introduzidos alguns tipos predefinidos (ver 4.4.5.) , contendo o protocolo para a manipulação de tipos de objetos mais comuns. Destacam-se os diferentes tipos de coleções apresentados: conjunto, dicionário, tupla, sequência, lista, texto, signo, progressão, sucessão, intervalo e vetor. As operações básicas para lidar com coleções (ver 4.4.5.9.) são a iteração, a seleção, a rejeição, a coleta, a deteção, a injeção e a ordenação.

Uma notação gráfica, denominada Diagrama de Objetos - DOO (ver 4.4.11.), foi desenvolvida para ser utilizada na modelagem conceitual da informação no GEOTABA. A Linguagem de Descrição e Manipulação de Objetos (DEMO) (ver 4.4.12.), uma notação textual para o modelo de objetos, também foi apresentada. A linguagem DEMO é usada para a escrita de consultas, métodos e definição de esquemas.

Mais considerações sobre o modelo de objetos do GEOTABA, sua notação gráfica e a linguagem DEMO, são feitas no Capítulo 6.

Capítulo 5

Protótipos Relativos ao Modelo de Objetos

5.1. Introdução

Este capítulo descreve alguns protótipos relacionados ao modelo de objetos do GEOTABA. O desenvolvimento do sistema de gerência de objetos GEOTABA [MATT89], adequado a ambientes de desenvolvimento de software e baseado no modelo de objetos MOO, depende da especificação fina desse modelo, que vem sendo elaborado desde 1991 [MONT91b]. Uma série de protótipos preliminares foram, estão sendo e serão ainda construídos, com o propósito de obter subsídios para a definição do modelo, validá-lo, aperfeiçoá-lo e testar soluções de implementação. Alguns desses protótipos são descritos nas seções deste capítulo. O ProtoGEO [MONT92b], que inclui a linguagem MANO, versão não tipada de DEMO, é descrito em 5.2. A especificação de MANO é parte integrante deste trabalho de definição do modelo de objetos do GEOTABA e atende a manipulação de objetos conforme o modelo preliminar [MONT91a] definido na época de sua implementação. MANO já continha uma série de características sintáticas de DEMO.

O ProtoGEO cria um ambiente onde objetos são criados, trocam mensagens e métodos e consultas podem ser realizadas. Além do compilador MANO, destaca-se no ProtoGEO a sua máquina virtual, que inclui a gerência de mensagens e classes, a memória de objetos e o conjunto (alterável) de primitivas. A implementação do ProtoGEO mostrou a viabilidade de realização dessas características e contribuiu no aperfeiçoamento da linguagem DEMO.

A seção 5.3. comenta sobre o Flecha [CAMA92] - um Editor Gráfico Cooperativo de Esquemas para o Modelo de Objetos do GEOTABA. A seção 5.4. discorre sobre o GOA e o PARGOA [MATT93] [MATT91], gerentes de ob-

jetos armazenados para o GEOTABA. A seção 5.5. trata do HiperFicha, sistema de hipermídia onde foram originadas algumas propostas para o nível de representação do modelo, e do Bandar, protótipo de um banco de dados relacional com extensões para dar suporte ao HiperFicha.

5.2. ProtoGEO

Esta seção apresenta o sistema protótipo de gerência de objetos ProtoGEO, que contou com a orientação do autor desta tese e fez parte do trabalho de tese de mestrado de J. Lisboa [LISB92]. O ProtoGEO consiste na versão inicial do Gerente de Execução do sistema GEOTABA [MONT92b]. Questões pertinentes a um sistema de gerência de objetos definitivo, do tipo compartilhamento, concorrência, distribuição, segurança e reconstrução, não são contempladas no ProtoGEO. Este se concentra nas características centrais do nível de implementação do modelo de objetos do GEOTABA.

O núcleo do ProtoGEO, escrito em C++, é responsável pelo processamento de mensagens/métodos escritos na linguagem de manipulação de objetos MANO. A linguagem MANO foi definida pelo autor desta tese para atender uma versão preliminar do modelo de objetos [MONT91a]. As principais características da linguagem DEMO, exceto a tipagem, já estavam presentes em MANO. A implementação do ProtoGEO mostrou a viabilidade de realização dessas características e contribuiu no aperfeiçoamento da linguagem DEMO.

A linguagem MANO é utilizada para a implementação dos métodos, que descrevem o comportamento dos objetos no nível de implementação do modelo de objetos do GEOTABA. MANO é uma linguagem orientada a objetos pura, com polimorfismo, encapsulamento, ligação dinâmica, herança de propriedades através de uma hierarquia de classes, etc. O núcleo do protótipo ProtoGEO funciona em um ambiente compatível com o PC AT. O ProtoGEO está sendo transportado para estações Sun, onde fará uso da memória virtual do Gerente de Objetos Armazenados (GOA) do GEOTABA e se comunicará com o Gerente de Meios de Armazenamento do Servidor de Objetos do GOA [MATT93].

Sintaticamente, MANO é uma sublinguagem de DEMO, no sentido em que todo programa em MANO é também um programa em DEMO. Porém MANO não é uma linguagem tipada como DEMO. A avaliação da experiência de programação em MANO contribuiu para a definição de DEMO. Portanto, a versão atualmente implementada de MANO possui algumas diferenças em relação a DEMO conforme descrita aqui (ver 4.4.12.). Estas diferenças serão eliminadas de MANO em futuro próximo. Inicialmente as duas linguagens conviverão e protótipos dos módulos de nível mais alto do GEOTABA serão

escritos em MANO. A proximidade das duas linguagens permitirá a fácil conversão de programas para DEMO. Após uma implementação estável de um processador DEMO, a linguagem MANO perderá a sua importância.

5.2.1. Arquitetura do ProtoGEO

A arquitetura utilizada na construção do ProtoGEO (Figura 55) é uma simplificação da arquitetura que será utilizada na construção do GEOTABA. O Gerente de Execução (GE) é o responsável pelo processamento das mensagens enviadas aos objetos do sistema. Quando um objeto recebe uma mensagem, um método específico deve ser executado com o objetivo de "responder" à mensagem. A identificação do método a ser executado, e sua execução, são tarefas de responsabilidade do gerente de execução. O Gerente de Execução utiliza o Interpretador MANO para executar o método. Durante a execução de um método, muitas vezes, outras mensagens são enviadas a objetos, o que resulta na execução de novos métodos. A busca do método que deve ser executado em decorrência de uma mensagem, é feita pelo Interpretador MANO na hierarquia de classes.

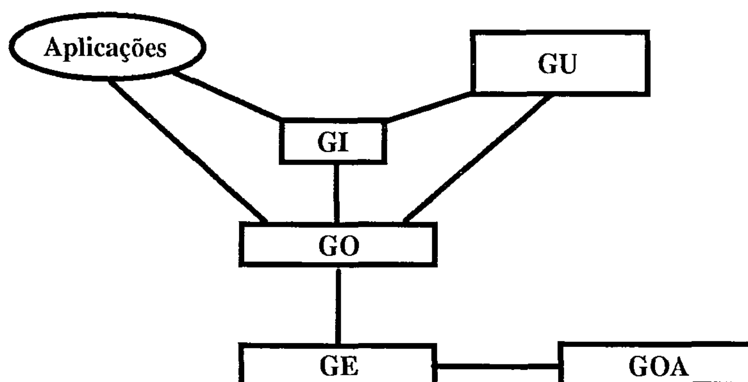


Figura 56: Arquitetura do ProtoGEO

Para executar um método, o interpretador necessita manipular diversos objetos. Estes objetos são obtidos através do Gerente de Objetos Armazenados (GOA), que os coloca disponível na Memória de Objetos.

O ProtoGEO possui ainda, um conjunto de classes predefinidas que são responsáveis pela implementação do metaesquema do modelo de objetos Estas classes formam o Gerente de Objetos (GO). A primeira delas é a classe Objeto que está localizada no topo da hierarquia de classes, significando que todas as demais classes são subclasses da classe Objeto. Esta classe engloba o protocolo que é comum a todos os objetos existentes. Como exemplos de classes

predefinidas, podemos citar também: a classe Classe que descreve a estrutura de dados e o protocolo de todos os objetos que são classes; a classe MétodoInterpretável que descreve a estrutura dos objetos que são os métodos gerados pelo compilador MANO.

As classes responsáveis por objetos de interface com usuário constituem o Gerente de Interação (GI). O Gerente de Uso (GU) possui as classes que permitem que o usuário faça consultas, crie novas classes e programe métodos. Todas estas atividades são realizadas através de mensagens, escritas pelo usuário. O GU ativa o compilador MANO para traduzir essas mensagens para a linguagem da máquina virtual MANO, embutida no GE. O Compilador MANO também faz acesso a objetos, como por exemplo, a classe (e suas superclasses) sob a qual o método está sendo compilado.

5.2.2. Linguagem MANO

A linguagem de Manipulação de Objetos MANO, foi projetada para permitir a definição de métodos e envio de mensagens.

5.2.2.1. Principais Características de MANO

A sintaxe da linguagem MANO é bastante simples, todos os comandos são, na verdade, envio de mensagens a algum objeto. Uma descrição completa da linguagem pode ser obtida em [LISB92]. A seguinte descrição de MANO é proveniente de [MONT92b].

A descrição dos objetos é feita através de suas classes. Quando um objeto recebe uma mensagem, um método de sua classe (ou superclasse) é executado. Este acoplamento mensagem-método é feito em tempo de execução.

MANO foi construída para ser usada através de um editor que utilize atributos de caracteres para realçar a visualização dos programas. Na representação usada nos exemplos, comentários são colocados dentro de caixas, nomes de objetos foram grifados e declarações de variáveis são sublinhadas. A seguir, é mostrado um exemplo de um método escrito em MANO.

Suponha que exista uma classe Aluno em cuja definição constam os seguintes campos: curso, nota_1 e nota_2. O exemplo apresenta o método de seletor média da classe Aluno que devolve como resultado da mensagem, o valor da média das notas do objeto receptor.

```
média;
        | Retorna o valor da média do aluno |
        ^ ( nota_1 + nota_2 ) / 2;
```

Neste exemplo, podemos observar as seguintes características:

1. Os campos não são declarados dentro do método, porque fazem parte da descrição da classe Aluno.
2. Na expressão (nota_1 + nota_2) / 2; existem duas mensagens, uma com seletor / e argumento 2, sendo enviada para o objeto resultante da outra mensagem, de seletor + com o argumento nota_2 que está sendo enviada para o objeto nota_1.
3. Existe uma ordem de precedência entre mensagens para operações aritméticas e booleanas, por isso a necessidade de parênteses.
4. O símbolo ^ significa que o objeto resultante da expressão deve ser retornado.
5. É permitida a utilização do conjunto de caracteres ascii estendidos, por isso o seletor média está acentuado.

O envio de mensagens sem parâmetros tem duas formas: prefixa e posfixa. Para obter o nome do chefe do departamento de um funcionário, podemos escrever:

    func.depto.chefe.nome

Isto significa que a mensagem depto está sendo enviada ao objeto designado por func. Ao objeto resultante desta mensagem, está sendo enviada a mensagem chefe, e da mesma forma a mensagem nome. Esta mesma expressão pode ser escrita usando-se preposições, obtendo-se uma forma mais próxima da língua portuguesa:

    nome do chefe do depto de func;

Em MANO, pode-se escrever os seletores de mensagens, que são compostos de mais de uma palavra, de diversas formas. Vejamos alguns exemplos:

    mova o valor de: nota_1 para: nota_2;
    crie_e_inicialize: aluno com: 0;

No exemplo a seguir, é mostrado o método da classe predefinida Objeto que permite a duplicação de um objeto por inteiro, ou seja, além do objeto que

está recebendo a mensagem ser duplicado, cada objeto referenciado por este também é duplicado.

```
duplique;

        Este método retorna um objeto idêntico ao receptor.
        A cópia é realizada em profundidade

cópia; classe; vars;

classe := classe;
( classe. é Variável )
então:          {    vars := tamBásico;
                     cópia := classe novo: vars; }
senão:          {    vars := 0;
                     cópia := novo de classe; }
( classe.é Apontador )
então:          {    1 até: (vars + tamInst de classe) faça:
                          { i; cópia[i] := duplique a mim[i]; } }
senão:          {    1 até: vars faça:
                          { i; cópia[i] := mim[i]; } }
^cópia;
```

Este exemplo apresenta as seguintes características de MANO:

1. Variáveis temporárias são declaradas logo após o cabeçalho do método.
2. A mensagem então:senão: possibilita a execução condicional de blocos de mensagens.
3. Existe um objeto especial (mim) que representa o objeto receptor da mensagem correspondente ao método em execução.
4. Blocos de mensagens podem ser parametrizados e executados repetitivamente.

Um método para uma classe X é traduzido pelo Compilador MANO para um objeto da classe MétodoInterpretável e armazenado junto com os demais objetos do banco. O método interpretável é acrescentado, através de seu seletor (cadeia de caracteres formada pelo cabeçalho do método), ao vetor de mensagens aceitáveis pela classe X.

5.2.2.2. *O Interpretador MANO*

A função do interpretador é executar os métodos, codificados em instâncias de MétodoInterpretável, que "respondem" às mensagens enviadas aos objetos. Existem basicamente cinco tipos de instruções nos métodos: de armazenamento (usadas em expressões de atribuição); de desvio (alteram o valor do apontador de instruções); de envio de mensagens (ativam outros métodos); de retorno (usadas para finalizar a execução de um método) e instruções que manipulam pilha de execução.

Para a execução de cada método, o interpretador utiliza um objeto da classe Contexto que contém uma pilha de execução. Quando um método ativa outro método, o estado de execução do método anterior é salvo no contexto corrente e um novo contexto é criado para a execução do novo método. Quando encerrada a execução deste novo método, o contexto salvo é reativado e sua execução prossegue normalmente.

Ao interpretar uma instrução de envio de mensagem a um objeto, uma busca deve ser realizada na hierarquia de classes para localizar o método que "responderá" a mensagem enviada. Esta busca é feita verificando-se inicialmente a existência do método no dicionário de mensagens da classe do objeto receptor da mensagem. Caso não seja encontrado, a busca prossegue em sua superclasse até o topo da hierarquia de classes (classe Objeto). Um tratamento de erro é ativado quando o método não é encontrado.

O Interpretador possui um conjunto de rotinas primitivas que executam de forma otimizada, as operações mais utilizadas: aritméticas, lógicas, de manipulação de cadeias, de entrada/saída, entre outras. Estas operações são executadas sem que haja a necessidade de troca de contexto, ou seja, não são criados novos contextos para a execução dos métodos que estiverem associados a rotinas primitivas.

5.2.3. Resumo

A importância do ProtoGEO para o GEOTABA foi constatada pela possibilidade de seu aproveitamento como uma versão inicial do núcleo deste sistema de objetos. Com relação ao modelo de objetos do GEOTABA, o ProtoGEO permitiu que se praticasse e avaliasse peculiaridades sintáticas da linguagem MANO e, em alguns casos, melhoramentos fossem sugeridos. Isto teve influência decisiva na definição da linguagem DEMO. Suas principais características, como a forma única para atribuições e indexação, já estavam presentes em MANO.

Além disso, o processador de MANO pode ser quase que totalmente aproveitado como processador de DEMO. A construção de objetos que implementarão os níveis superiores do modelo, usados por DEMO, será realizada em MANO (preferivelmente após alterações que a colocará de acordo com a definição de DEMO atual). A tradução de MANO para DEMO poderá ser realizada com muita facilidade, consistindo, praticamente, na inclusão de tipos em variáveis e parâmetros.

5.3. Flecha

Os Diagramas de Objetos (ver 4.4.11.) foram desenvolvidos em conjunto com o modelo de objetos do GEOTABA. Esta seção apresenta o editor gráfico cooperativo Flecha de Diagramas de Objetos, para a criação de esquemas no modelo de objetos do GEOTABA.

A implementação deste editor foi parte do trabalho de tese de mestrado C.S.P. Camargo [CAMA92]. Utilizando o Flecha, o usuário projeta os esquemas através de Diagramas de Objetos (DOOs) [MONT91a]. O objetivo do Flecha é auxiliar um grupo de projetistas a trabalhar em conjunto, produtivamente, na criação de diagramas de objetos, trocando idéias e sugestões e, integrarem diagramas complexos.

Propiciando o trabalho cooperativo, esse editor permite a criação de anotações, troca de mensagens, facilidades para a interação e compartilhamento da informação entre colaboradores. É possível que indivíduos possam trabalhar simultaneamente em diferentes partes do mesmo diagrama e tomando conhecimento imediato das alterações feitas por todos estes coautores.

O Flecha reconhece três categorias de usuários: gerentes, comodeladores e leitores. Os comodeladores recebem de gerentes ou de outros comodeladores contextos, onde possuem direito de alteração. Ao leitor é permitido participar apenas com comentários e mensagens. Comentários podem ser públicos, privados (restrito ao autor) e mensagens.

O Flecha funciona em redes locais Novell em estações com Ms-Windows. O Flecha deverá ser integrado ao ProtoGEO [MONT92b] (ver seção anterior).

5.4. GOA e PARGOA

Dois protótipos, GOA e PARGOA [MATT93], foram construídos visando a gerência de armazenamento de objetos do GEOTABA. GOA é o Gerente de Objetos Armazenados do GEOTABA. Nele foram exploradas

diferentes estratégias de resolução de consultas. O PARGOA, Servidor Paralelo de Objetos, é um servidor de objetos que, em uma máquina NCP1, de arquitetura paralela, utiliza algoritmos paralelos na gerência de objetos armazenados. O PARGOA suporta as mesmas operações de gerência e de consultas do GOA, utilizando o mesmo modelo físico de armazenamento de objetos. Tanto o GOA, como o PARGOA, se inserem em uma arquitetura para o GEOTABA, especificada em [MATT93], capaz de comportar o uso de paralelismo. Para dar suportar o modelo de objetos do GEOTABA, esta arquitetura possui características de extensibilidade. Também foi especificada uma plataforma cliente/servidor, com exploração de paralelismo no servidor, para esta arquitetura.

As tarefas de gerência de objetos armazenados incluem a alocação e remoção de objetos, administração de coleções e o acesso associativo a elementos de coleções. O GOA gerencia a memória de objetos, a memória de páginas e o espaço em disco.

O PARGOA concilia a distribuição, para processamento paralelo, dos objetos de uma coleção com o agrupamento para o processamento em conjunto. Ele utiliza um algoritmo para gerência de paralelismo no servidor de objetos e algoritmos paralelos específicos para a avaliação de predicados.

A integração do ProtoGEO ao GOA/PARGOA está em andamento.

5.5. HiperFicha e Bandar

O protótipo HiperFicha foi desenvolvido por alunos de quatro períodos da disciplina Laboratório de Banco de Dados, do curso de mestrado em Engenharia de Sistemas e Computação da COPPE/UFRJ, em Smalltalk/V, sob a orientação do autor desta tese e professor da disciplina. Como discutido em [MONT89], o HiperFicha foi visualizado como a peça central de uma arquitetura proposta para os ambientes de desenvolvimento de software das estações TABA. Nesta arquitetura, as ferramentas de software dos ambientes seriam encaradas como hiperdocumentos ativos. Estes seriam textos e diagramas editáveis e com a capacidade de disparar processos como resposta a ações do usuário, tais como clicar o *mouse* em uma entidade em um diagrama ou escolher uma opção em um menu. Segundo essa proposta, os ambientes de desenvolvimento de software do TABA seriam centrados em documentos e não em ferramentas, como é o caso das arquiteturas tradicionais. Note-se que esta é a visão hoje preconizada por sistemas como o Ms-Windows ao introduzir a facilidade de ligação ativa entre documentos (OLE).

O HiperFicha no TABA seria o gerente da manipulação interativa de um sistema de objetos (o GEOTABA). O HiperFicha foi inspirado nos produtos Hypercard [BORN87] [WILL87] e Notecards [CONK87]. Assim como nesses sistemas, a ficha[29] é a metáfora primordial. Todavia, a proposta do HiperFicha se distinguia em vários pontos, entre estes a forte integração com um sistema de objetos e o oferecimento de mecanismos adicionais de consulta não-navegacionais.

Na arquitetura proposta, os hiperdocumentos seriam suportados pelo HiperFicha, permitindo que o usuário navegasse-os e ativasse ações. O HiperFicha se apoiaria no GEOTABA para obter os objetos e executar as ações. Estas ações seriam especificadas pelos autores de hiperdocumentos na linguagem DEMO. Um outro módulo fazendo uso do GEOTABA seria o CHOC, um Auxiliador de Desenvolvimento de Interfaces com Usuário. Segundo essa proposta, tanto o HiperFicha, quanto o CHOC, seriam escritos em DEMO. O CHOC faria uso do HiperFicha para prover as ferramentas de construção de interfaces e o HiperFicha usaria o CHOC para permitir a criação de novos objetos de interação para os documentos. Dentro dessa proposta, uma discussão sobre os requisitos para o modelo de objetos do GEOTABA poder suportar hiperdocumentos já apontava pela necessidade de integrar dados, interfaces e métodos.

A prototipação do HiperFicha foi menos ambiciosa. Das três formas de uso previstas (folheamento, edição e autoria), apenas o folheamento foi oferecido para todos os dispositivos de interação desenvolvidos (campos de texto, diagramas, menus e tabelas). A edição não foi implementada para menus e apenas parcialmente para tabelas. A autoria interativa não foi implementada. O protótipo não se apoiava em um sistema de objetos. A linguagem em que métodos eram escritos e incorporados aos campos ativos das fichas era a própria Smalltalk usada cmo linguagem de implementação do protótipo.

Tabelas formavam um dos tipos de objetos de interação desenvolvidos no HiperFicha. Por meio de tabelas, o usuário poderia visualizar relações, alterar valores, criar colunas e linhas e preenchê-las pela execução de funções, como em planilhas eletrônicas. Bandar[30] era um gerenciador dos objetos relações representáveis por essas tabelas e manuseáveis por operações da álgebra relacional [CAMA91].

---

[29]chamada de *card*, nesses outros sistemas.

[30]Bandar significa Banco de Dados Relacional Pigmeu.

O construção do protótipo do HiperFicha deu grande contribuição à concepção do modelo de objetos do GEOTABA. Particularmente com relação às necessidades do nível conceitual do modelo. A necessidade de atualização imediata das fichas e de seus campos exigiu o estudo de diversas possibilidades de associação entre objetos de interface e seus objetos bases. Por outro lado, a criação e uso de dados no modelo relacional, proporcionados pelo Bandar, mostrou as limitações do uso de bases relacionais em um ambiente orientado a objetos. Isto serviu para aumentar a percepção de que o modelo do GEOTABA não deveria cair em vícios, comuns em alguns modelos de dados orientados a objetos, oriundos dos modelos orientados a registros, como a não inclusão de métodos, a ênfase em coleções, a confusão entre classes e coleções, o uso generalizado de referências através de chaves estrangeiras, o tratamento distinto entre objetos e valores, a impossibilidade de se criar novos tipos de coleção com comportamento próprio, a distância entre as linguagens de programação de métodos e a linguagem de consultas, a dificuldade dos objetos de responderem qual a sua classe, a linguagem de consulta não extensível e a impossibilidade de um objeto adquirir novos papéis.

Capítulo 6

Sumário e Perspectivas Futuras

6.1. Sistemas de Objetos: Integrando Processamento, Armazenamento e Manipulação de Informação

O projeto TABA, que busca a criação de estações de trabalho para a engenharia de software, tem ensejado diversas pesquisas na área de Bancos de Dados. Dentre estas estão os trabalhos para a definição e implementação do Sistema de Objetos GEOTABA. Especial cuidado há nesse projeto com relação ao desenvolvimento e suporte a aplicações com características hodiernas, onde a facilidade de uso e apredizagem são determinantes para a aceitação de sistemas computadorizados. Estes se apoiam no uso de metáforas de situações, processos e utensílios familiares aos usuários, na interatividade, na utilização de múltiplos meios de armazenamento, apresentação e manipulação da informação e na ajustabilidade. A esta busca pela facilidade de uso, corresponde um aumento exponencial da complexidade interna do software, ou seja, do seu desenvolvimento.

O paradigma da orientação a objetos tem sido aplicado no desenvolvimento de sistemas com essas características. Interfaces com usuário por manipulação direta, chamadas por alguns de interfaces orientadas a objetos, representaram um real avanço quanto a facildade de uso e de aprendizado dos sistemas de computação. Estas interfaces se baseiam nos mesmos conceitos que norteiam a programação orientada a objetos: objetos que reagem às ações, estas são transmitidas diretamente a cada objeto, a mesma ação provoca reações diferentes em objetos de classe diferentes ou com estado interno diferente, etc. Não é por acaso que os sistemas que realmente proporcionam manipulação direta foram programados utilizando características presentes nas linguagens orientadas a objetos. O objetivo desse paradigma é aproveitar os mecanismos

humanos de compreensão do mundo real para a compreensão dos mundos virtuais criados pela computação.

Além da programação e da comunicação homem-computador, a orientação a objetos despontou na gerência de dados dos sistemas e nas próprias fases iniciais do processo de criação de software: análise/especificação e projeto. Os objetivos contiuaram os mesmos.

Todavia, cada uma dessas áreas desenvolveu-se com devido distanciamento do que se passava nas outras. Linguagens de programação orientadas a objetos, SGBDs orientados a objetos e sistemas de gerência de interface com usuário orientados a objetos foram criados isoladamente.

Este trabalho propôs que um sistema único integre, através do paradigma da orientação a objetos, a funcionalidade proporcionada por esses sistemas em separado. Um, então denominado, Sistema de Objeto gerencia a informação para processamento, armazenamento ou manipulação pelo usuário. O casamento entre estas diferentes abordagens se dá através de um Modelo de Objetos. Este fornece um instrumental de representação da informação, baseado nos conceitos da orientação a objetos, integrador dos diversos modos que um sistema de computação encara a informação.

O GEOTABA seria um sistema de objetos que administraria os objetos de software das estações TABA. Ele estaria presente, tanto na criação de ambientes de desenvolvimento de software, suportando o chamado meta-ambiente, quanto na base dos próprios ambientes gerados e, possivelmente, em alguns ambientes de execução dos sistemas de software desenvolvidos por esses ambientes. O GEOTABA se concentra na gerência dos objetos, deixando, para ferramentas e ambientes de desenvolvimento exteriores, o processo de engenharia desses objetos. Com essa definição de papéis aqui apresentada, a arquitetura de estações de engenharia de software do TABA que se basearem no GEOTABA poderá ser especificada em detalhe.

6.2. O Modelo de Objetos do GEOTABA

O modelo de objetos do GEOTABA permite que uma mesma entidade possa ser observada sob diferentes níveis de abstração. No nível conceitual, têm-se os dados dessa entidade e as ações realizadas por ela, conforme percebidos pela comunidade dos usuários dessa entidade. No nível de implementação, tem-se acesso aos dados e as ações como efetivamente foram implementados. No nível externo, tem-se vistas da entidade, apropriadas a cada tipo de usuário. O nível de representação se refere a como a entidade se apresenta nos meios de interação com usuário.

Por exemplo, um automóvel "conceitual" tem a sua real implementação ocultada, aberta apenas no nível de implementação. Automóveis distintos, percebidos como sendo do mesmo tipo de entidade, podem ter implementações totalmente diferentes, contanto que estas proporcionem a mesma visão conceitual definida para automóveis. Diferentes usuários podem perceber um mesmo automóvel de maneiras distintas, apropriadas às especifidades deles. E, ainda mais, um mesmo automóvel, até para um mesmo usuário, pode ser representado de diversas maneiras em um meio de interação com usuário. Todas essas diferentes visões se referiram ao mesmo objeto automóvel. Uma alteração feita com o *mouse* no desenho de alguma parte desse automóvel poderá ter que refletir em alterações nos dados referentes ao automóvel nos demais níveis de representação.

O nível de implementação trata os objetos do modo com que são tratados pelas linguagens de programação. As linguagens com capacidade de definição de tipos abstratos, proporcionam também uma outra visão dos objetos. Dentre essas, poucas fazem uma efetiva separação das duas visões, ocultando rigidamente a estrutura e os serviços implementados. A possibilidade de se ter atributos públicos, isto é, por definição presentes nas duas visões, diminui a independência entre os dois níveis de um objeto. Exemplos dessa abordagem são as linguagens Ada [LEDG81] e C++ [STRO86].

Outras linguagens separam rigorosamente os níveis. Porém, o modelo conceitual que elas oferecem é limitado. No extremo dessas linguagens está Smalltalk [GOLD83], que vê conceitualmente um objeto apenas através de quais mensagens ele pode receber. Essas linguagens não têm a preocupação de fornecer um ferramental de descrição conceitual tão rico quanto os modelos semânticos de dados. Elas partem de um vício de origem: a visão conceitual dos objetos é induzida da definição da implementação desses. Isso parece ser o mais natural para uma linguagem de programação. O modelo de objetos do GEOTABA, além de se preocupar com a independência entre os dois níveis, prioriza a definição da visão conceitual dos objetos, deduzindo uma implementação padrão que pode ser alterada ou servir de base para outras implementações.

O Grupo de Estudos em Sistemas de Gerência de Bases de Dados da ANSI/SPARC propôs que a arquitetura de SGBDs apresente três níveis: interno, conceitual e externo [TSIC78]. É importante salientar a diferença entre o nível interno do modelo ANSI/SPARC e o nível de implementação do GEOTABA, a não existência do nível de representação e a não captura da semântica real dos dados nos modelos de dados de SGBDs convencionais.

O nível interno ANSI/SPARC se refere mais a definição das estruturas físicas dos dados armazenados do que a sua manipulação em memória. Em

resumo, as linguagens atuais para SGBDs convencionais não são computacionalmente completas, no sentido de Turing. A razão disso está na distância semântica entre os modelos computacionalmente completos tradicionais e os modelos de dados dos SGBDs convencionais. Por isso, na prática, os SGBDs não dispensam o apoio de linguagens de programação para embutir determinadas consultas e tarefas. O paradigma da orientação a objetos pode ser usado como uma ponte para ligar as duas margens desse problema. Os sistemas de gerência de bases de dados orientados a objetos baseiam-se em modelos de representação conceitual da informação que seguem o mesmo paradigma que, nas linguagens de programação orientadas a objetos, provê um modelo computacionalmente completo. Em virtude disso, pode-se definir um nível de implementação computacionalmente completo.

Os modelos de dados dos SGBDs orientados a objetos emergentes, como o O_2 [DEUX90], ou não suportam a separação em níveis preconizada pela arquitetura ANSI/SPARC, ou não provêm a independência necessária entre esses níveis. Além disso, o modelo conceitual que eles fornecem é mais próximo das linguagens de programação do que dos modelos semânticos de dados. Característica é a dificuldade que esses modelos orientados a objetos têm em representar o relacionamento entre objetos como algo exterior aos objetos em si.

A não previsão de um nível de representação na arquitetura ANSI/SPARC é típica da menor importância dada à comunicação homem-computador, na época da definição desse padrão, em relação ao que ocorre nos sistemas atuais. Na prática, os SGBDs contêm ferramental de interfaceamento com usuário, provendo apresentação de tabelas, fichas e gráficos. O modelo de dados de um SGBD costuma ser um fator limitante quanto a variedade de representações para o usuário em que a informação pode ser mostrada e quanto a qualidade da interação o usuário pode ter com essas representações. É natural que modelos que encarem a informação através de assemelhados a arquivo de registros só permitam que esses registros sejam vistos sob a forma de fichas ou de linhas em tabelas. Se é viável apresentar alguma informação através de *pie-charts* ou de histogramas, mais difícil é manipular a informação através desses mesmos gráficos. A necessidade de se desenvolver sistemas multimídia tem sido tratada anexando aos SGBDs convencionais capacidade de suportar, nos dois sentidos da palavra, dados não estruturados (crus) e tipos abstratos de dados desenvolvidos em linguagens de programação externas ao banco de dados.

Aqui também a orientação a objetos foi encarada como a ponte entre problemas antes tratados separadamente. As interfaces com usuário modernas têm sido implementadas em sistemas com características da orientação a objetos.

Modelos semânticos de dados foram criados para superar a dificuldade que os modelos de dados tradicionais possuem para representar a semântica real da informação. O paradigma da orientação a objetos possui muitos pontos de contato com esses modelos. O principal deles é a própria intenção de capturar as propriedades de entidades reais. Pode-se lembrar que os primórdios das linguagens de programação orientadas a objetos (Simula [DAHL66]) está ligado a programação de sistemas de simulação. A maior deficiência dos modelos orientados a objetos em relação aos modelos semânticos é que aqueles têm uma dificuldade inerente de explicitar as propriedades dos relacionamentos entre objetos. Do nível conceitual do modelo de objetos do GEOTABA, portanto, foi exigida uma expressividade correspondente a de um modelo semântico de dados, mas que se mantenha tão próximo da orientação a objetos que permita o acoplamento natural com os demais níveis do modelo.

Uma propriedade importante dos modelos orientados a objetos é a extensibilidade alcançada pela possibilidade de se criar, a qualquer momento, novos tipos de objetos. Aplicando a orientação a objetos a todos os níveis do modelo de objetos do GEOTABA pode-se ter, dinamicamente, novos tipos de implementação (e armazenamento) de um dado tipo de objeto, e novos tipos de representação de um dado tipo de objeto.

6.3. O Nível Conceitual do Modelo de Objetos do GEOTABA

Sendo o nível conceitual do modelo de objetos do GEOTABA o responsável por expressar a semântica da informação gerenciada pelo sistema, este é o nível central do modelo. A definição dos demais níveis é função dependente das definições desse nível.

Pressupondo que o paradigma da orientação a objetos se basearia na maneira natural do ser humano apreender as propriedades do mundo real, o nível conceitual do modelo se fundamenta em conceitos típicos desse paradigma. A informação é organizada em objetos que possuem identidade, estado e comportamento próprios. O estado de um objeto pode variar de acordo com as ações que ele sofre. Uma ação é realizada sobre algum objeto, que reagirá a ela de acordo com o seu comportamento, que por sua vez é influenciado pelo seu estado. O estado diz respeito aos relacionamentos que um objeto mantém com os demais objetos.

A separação dos níveis conceitual e de implementação permite que se fale da estrutura percebida de um objeto, que pode ser diferente da sua estrutura implementada. A estrutura percebida consiste de um conjunto de associações deste objeto com outros, que são chamados de valores da associação. Um atri-

buto é o tipo mais comum de associação e se caracteriza por possuir uma chave de identificação, que comumente é um nome ou um índice numérico. Um atributo, parte da estrutura percebida de um objeto, pode não ser um campo físico de sua estrutura implementada, o que garante o grau desejado de independência entre os dois níveis mencionados.

Os modelos orientados a objetos ou são híbridos e tipados, ou são puros mas não tipados. O modelo do GEOTABA é puro, ou seja, toda informação é manipulada como objetos. Números inteiros, reais e cadeias de caracteres também são objetos. Isso proporciona uniformidade e extensibilidade.

Ao contrário do comum, o modelo de objetos do GEOTABA, apesar de não ser híbrido, é tipado. Cada objeto possui na sua implementação informação capaz de identificar se um objeto é de um tipo ou não. Esta verificação pode ser realizada estaticamente graças ao mecanismo de padrões e de padrões-tipos. Um padrão é simplesmente um conjunto de restrições. Cada associação possui um padrão que, pela definição da associação, seus valores têm que respeitar.

A modelagem conceitual é realizada pela definição de padrões e, principalmente, de padrões-tipos. Um padrão-tipo define o padrão que todos os objetos de um tipo têm que respeitar. Os padrões-tipos são definidos em uma hierarquia onde os subtipos herdam as propriedades dos seus supertipos. Um subtipo só pode redefinir uma propriedade de um seu supertipo caso esta redefinição não contrarie a propriedade anterior, ou seja, esta continue valendo. Assim, um subtipo pode apenas acrescentar restrições adicionais.

Ao contrário dos demais modelos orientados a objetos, o do GEOTABA não negligencia a questão do relacionamento entre objetos. Uma <u>ligação</u> entre atributos de objetos de dois (ou mais) padrões-tipos representa um relacionamento entre esses objetos. Embora uma ligação seja exterior aos objetos envolvidos, quebrando a rigidez da orientação a objetos, ela é refletida nos padrões dos atributos envolvidos. Isso permite que, em praticamente todas as situações, apesar da ligação afetar os objetos envolvidos, ela nunca seja manipulada diretamente. De fato, o acesso a um dos atributos envolvidos atravessa a ligação e tem como resposta o objeto associado ao atributo pela ligação.

Outro fator que distingue o modelo do GEOTABA de vários modelos orientados a objetos é o fato que tipos não são repositórios de objetos. Ou seja, são distintos os conceitos de tipo e coleção. Isso é fundamental para se ter a completude computacional, a extensibilidade (permitindo a criação de novos tipos de coleções) e a localização mais apurada de determinadas restrições (diferenciando restrições que são características de um tipo das que são próprias

a uma determinada coleção). Infelizmente, este é mais um dos fatores a aumentar a complexidade do modelo.

A evolução dinâmica dos objetos é tratada através do conceito de aspectos. Um aspecto reúne propriedades que um objeto adquire na sua dinâmica. Assim, um empregado pode passar a ser um gerente, ou sócio de algum clube, sem que ele tenha que mudar de tipo (o que poderia invalidar associações que ele participasse), nem que novos tipos sejam criados combinado os pré-existentes (o tipo empregado-sócio seria desnecessário).

Aspectos foram introduzidos pelo modelo de dados Melampus [RICH91]. Aqui eles foram adaptados aos requisitos do modelo de objetos do GEOTABA. Uma primeira diferença é o fato de que, no GEOTABA, um aspecto é definido conceitualmente e sua implementação é derivada dessa definição, podendo ser posteriormente alterada. No Melampus um aspecto é definido pela sua implementação e seu *tipo* (padrão, no GEOTABA) é gerado implicitamente. Outra diferença importante é que o padrão definido por um aspecto, no GEOTABA, herda, por *default*, as propriedades do padrão base, podendo redefini-las totalmente. No Melampus, um aspecto tem que especificar quais as propriedades do padrão base ele exporta. Isto provoca que um objeto, quando visto sob um dado aspecto seu, não possa ser atribuído a uma variável restrita a seu padrão base. No GEOTABA isto é possível e o objeto atribuído é visto, através dessa variável, conforme definido no seu padrão-base. Isto é mais de acordo com o que parece ser natural, já que o objeto que possui o aspecto, de fato, pode ser visto, tanto conforme o padrão base, como conforme o padrão aspecto. Outra diferença importante é que, no GEOTABA, a remoção de aspectos não pode gerar referências pendentes.

A descrição do nível conceitual apresentada define alguns tópicos que deverão ser mais explorados em trabalhos futuros. Entre esses estão as restrições dinâmicas, restrições em serviços, o tratamento de exceções e os valores iniciais e compartilhados. A definição de novos tipos de bifurcação de ligações permitiria se especificar mais claramente restrições que envolvessem vários objetos. O mapeamento do nível conceitual no nível de implementação necessita de uma descrição extensiva.

Uma representação gráfica, os Diagramas de Objetos, usável na modelagem conceitual da informação para o GEOTABA também foi apresentada. Uma linguagem textual, a DEMO, foi descrita no que se referiu ao seu emprego na manipulação de objetos. O uso dessa linguagem para a descrição de padrões e tipos de objetos não foi definido e precisará ser tratado futuramente. A omissão pode ser feita porque os Diagramas de Objetos suprem essa

necessidade. Os diagramas sempre serão mais adequados a essa tarefa do que a linguagem textual.

6.4. Os Diagramas de Objetos

Os Diagramas de Objetos, usáveis na modelagem conceitual de bases de objetos do GEOTABA, atendem ao mesmo tempo a representação de modelagens semânticas de informação e são mapeados trivialmente no Modelo de Objetos do GEOTABA. Os Diagramas de Objetos foram apresentados na Conferência Latinoamericana de Informática em 1991 [MONT91a]. Um editor cooperativo de Diagramas de Objetos, denominado Flecha [CAMA92], foi desenvolvido.

Os Diagramas de Objetos devem ser encarados como um subconjunto de uma linguagem visual de definição e manipulação de objetos do GEOTABA. Para isto ser alcançado, eles devem ser estendidos para que seja possível especificar interativamente processamentos e executá-los e representar os demais níveis de abstração do Modelo de Objetos.

6.5. A Linguagem DEMO

O modelo de objetos do GEOTABA é materializado em uma linguagem textual: DEMO - Descrição e Manipulação de Objetos. Além de atender as características do modelo de objetos, a DEMO possui diversas peculiaridades sintáticas.

A principal delas se refere às mensagens de consulta e de atribuição. A notação adotada visa proporcionar ao programador a sensação de que, no nível conceitual, mesmo através de mensagens, ele está manipulando atributos percebidos. Para exemplificar o problema, será descrito a seguir como as linguagens Smalltalk [GOLD83] e C++ [STRO86] lidam com a questão.

Em Smalltalk, toda estrutura implementada de um objeto é oculta por sua interface. Para que um campo (variável de instância) desse objeto possa ser consultado, um método de consulta, com o mesmo nome do campo, deve ser criado. Para que possa haver atribuições a esse campo, um método que faça a atribuição também precisa ser criado. Em geral, o nome (seletor) do método consiste em uma palavra-chave derivada do nome do campo. Assim, um campo de nome *idade* precisaria de um método de nome *idade:* para realizar atribuições. A atribuição de uma idade a um objeto *fulano* se daria conforme a expressão *fulano idade: 30*. Ocorre que a atribuição a uma variável, nessa linguagem, é realizada pelo tradicional operador :=, comum nas linguagens de programação

derivadas de Algol. Esta diferença entre as duas atribuições não promove a sensação de que *idade* é um atributo da estrutura percebida do objeto. Isto é agravado pela notação da atribuição a um elemento de uma coleção (um *Array*, por exemplo). Para atribuir a cadeia *'abc'* ao terceiro elemento de um array *vet*, se escreveria *vet at:3 put:'abc'*.

Em C++, existem duas opções. A primeira seria definir o campo (membro) *idade* como sendo público. Isso permitiria que a atribuição pudesse ser escrita como *fulano.idade := 30*. É claro que, com essa notação, o programador perceberia *idade* como fazendo parte da estrutura do objeto. Infelizmente, a independência da especificação conceitual do objeto e a sua implementação fica comprometida. Por exemplo, se o programador deseja criar um efeito colateral na atribuição de um valor a uma idade (atualizando um outro objeto, por exemplo) esta solução não poderá ser adotada. E fortemente não é adotada pelos que pretendem programar realmente orientado a objetos em C++. Nesse caso, o empregado é uma solução próxima, porém pior, da solução Smalltalk: uma função membro para atribuição é criada. O nome, em inglês, da função, costuma ter o prefixo *set*. Neste caso se escreveria *fulano.setidade(30)*.

Em DEMO, para qualquer atribuição se utiliza o operador *:=*. Isto significa que um método para atribuir uma idade teria o nome de *idade:=*. Ao método *at:put:* do Smalltalk, corresponde o método *[]:=*, em DEMO, para permitir a notação de colchetes para a indexação de vetores e dicionários. O analisador consegue reconhecer tanto *fulano idade := 30*, quanto *idade de fulano := 30*. A linguagem MANO, precursora de DEMO no GEOTABA, foi implementada com essas características [LISB92]. A atribuição a um vetor seria denotada por *vet[3]:='abc'*. Este mesmo operador *:=* serve para atribuir um valor a uma variável global, temporária ou parâmetro de um método. Na realidade conceitual estas atribuições também são mensagens enviadas a objetos bases de objetos, contextos de blocos ou contextos de método.

Embora conceitualmente simples, essa solução adotada é fundamental para que o programador tenha a sensação de que a interface de um objeto contém a estrutura percebida desse e não apenas métodos de serviço. A manipulação dos dados contidos na estrutura percebida é essencial numa visão de gerência de dados do sistema, como nos SGBDs. Com essa solução, a independência da implementação, a extensibilidade e a orientação a objetos são garantidas. A essa simplicidade conceitual não corresponde uma trivialidade na implementação de um compilador, como foi demonstrado no sistema ProtoGEO [LISB92].

Outra peculiaridade de DEMO é a noção de objeto corrente. A execução de cada bloco especifica um objeto corrente. Uma mensagem que omite a espe-

cificação do seu receptor é enviada ao objeto corrente do bloco. Essa regra é uma extensão de uma regra semelhante de C++, onde o receptor omitido é o próprio receptor da função onde a mensagem está escrita. Poder mudar esse *default* a cada bloco, permite uma notação mais apropriada para os métodos de coleções. A linguagem DEMO pode então funcionar como uma linguagem multiparadigma [STEF86a], onde, por exemplo, as operações da álgebra relacional podem ser definidas. O operador λ permite batizar o objeto corrente de um bloco. Assim, notações semelhantes ao cálculo relacional podem ser providas.

A linguagem DEMO oferece uma liberdade ímpar na definição de nomes de variáveis e de procedimentos. Sentenças com aparência de linguagem natural são construídas e interpretadas intuitivamente pelos usuários, graças a definição implícita de valores morfológicos dos identificadores. O uso de "preposições" e caracteres acentuados também torna mais "latino" os programas escritos em DEMO. Também é notável a possibilidade de se criar novos operadores, diferentemente de C++ ou Smalltalk, onde apenas podem ser criadas novas operações para os operadores predefinidos da linguagem. Além disso, a especificação da prioridade desses operadores também pode ser especificada de um modo simples.

A extensibilidade da linguagem é bem caracterizada pela possibilidade de se ter parâmetros e variáveis contendo blocos, métodos, classes, tipos e coleções genéricos. O mecanismo de expressões de padrão, juntamente com o de coerção permite que essa extensibilidade possa ser fornecida sem que a tipagem das variáveis seja enfraquecida.

Diversas outras peculiaridades foram descritas no decorrer desse texto. Todavia, quase sempre, a especificação da estrutura percebida foi feita através de Diagramas de Objetos, mais naturais do que uma linguagem textual para essa tarefa. Também não foram contemplados, na descrição da linguagem, os demais níveis do modelo. Apesar disso, a grande base descrita proporcionará que a versão completa definitiva da linguagem DEMO possa ser realizada com pouco esforço.

6.6. Os Níveis Externo e de Representação e sua Relação com Aspectos

A principal utilização de aspectos no GEOTABA está na modelagem de papéis que as entidades podem assumir, ou deixarem de ter, dinamicamente. Outras utilizações são mostradas em [RICH91]: representação de papéis comuns

a objetos de tipos sem relação entre si e para a interpretação de resultados de consultas.

O nível externo do modelo define o conceito de vistas de um objeto. Uma vista, assim como um aspecto, representa uma maneira específica de se ver o objeto conceitual. Tanto vistas como aspectos podem ser definidos em qualquer momento posterior à criação do objeto. Todavia, um aspecto, acrescentado a um objeto, é incorporado a ele. Uma vista, pelo contrário, não altera a visão conceitual do objeto; ela sempre se baseia nesta visão conceitual, ou em outra vista. As semelhanças e diferenças entre aspectos e vistas devem ser esmiuçadas em trabalhos futuros. Algumas combinação entre vistas e aspectos foram aqui discutidas, um trabalho futuro deve descrever todas as possíveis alternativas.

Os representantes, que correspondem a como os objetos são vistos no nível de representação, também têm características comuns a aspectos e vistas. Também fornecem modos particulares de se perceber um objeto e podem ser acrescentados dinamicamente. Semelhantemente a vistas, a criação dinâmica de um representante sobre um objeto não o altera. Este ignora a existência de seus representantes, a não ser pelo fato de que as alterações que o objeto sofre são comunicadas aos seus representantes, e vice-versa. Por outro lado, um representante pode ter métodos e atributos próprios, não relacionados ao seu objeto base. A relação entre representantes e vistas também é assunto para pesquisas futuras.

Este trabalho pretendeu somente fornecer a base sobre a qual os níveis externo e de representação devem ser definidos. Esta base se preocupa em criar uma filosofia coerente para todo o modelo de objetos. Uma definição detalhada desses níveis e exemplos de modelagens externas e de representação fazem parte da sequência deste trabalho.

6.7. Considerações Finais

Os desdobramentos do trabalho apresentado são muitos, boa parte deles relacionada nas seções anteriores do atual capítulo. Outros pontos em aberto, ainda existentes, foram apresentados no decorrer do próprio Capítulo 4:"O Modelo de Objetos do GEOTABA". Esta seção em particular se deterá na proposição de um caminho imediato a ser seguido.

O aproveitamento do ProtoGEO na construção do GEOTABA é possível e recomendável. Sua integração com o GOA/PARGOA, em estações Sun, poderá ser iniciada imediatamente. Para que isto possa ter caráter definitivo, é conveniente que um modelo de armazenamento físico, completo e

em total acordo com os requisitos impostos pelo modelo de objetos aqui descrito, seja estabelecido.

A linguagem MANO é a outra extremidade do ProtoGEO. Dois caminhos podem ser tomados daí. Um deles aponta pela reforma de MANO, de modo a torná-la tipada e sintaticamente compatível com a DEMO atual. Para isto, basta serem feitas alterações nos analisadores léxico, sintáticos e semânticos, que juntos constituem apenas uma pequena parte de todo o processador MANO. O segundo caminho preserva MANO como ela está atualmente para a consecução da tarefa seguinte.

Esta consistirá na escrita em MANO (reformada ou não) de classes predefinidas pelo sistema a serem usadas pela versão inicial de DEMO. Exemplo dessas são as classes *Objeto*, *Classe*, *Metaclasse*, *Metaobjeto* e *MétodoCompilado*. Algumas dessas classes são simuladas artificialmente pelo processador atual de MANO. Por exemplo, para que o programador possa criar uma classe é necessário enviar uma mensagem à classe *Classe*. Esta é simulada por um módulo do processador, escrito em C++. A classe *Classe* definitiva pode então ser criada, escrita na própria MANO. Uma vez compilada, ela será usável até por métodos e consultas em DEMO.

O objetivo da linguagem MANO era escrever métodos e criar classes do nível de implementação do modelo. Com MANO, portanto, se poderia implementar qualquer objeto do GEOTABA. MANO será sempre restrita ao nível de implementação. Tipos, em DEMO, são definidos no nível conceitual e implementações são derivadas deles. A reforma de MANO mencionada acima incluiria tipagem através de padrões. Todavia, a abordagem em MANO seria oposta: classes definem tipos (padrões) implícitos. Desse modo, MANO pode cumprir seu papel adequadamente no *bootstrapping* de DEMO. Basta que cada variável ou congênere especifique como seu padrão o padrão (implícito) de alguma classe. Esta é a abordagem adotada nas linguagens de programação que separam classes e tipos. Com esta reforma, todo programa em MANO seria um programa válido em DEMO.

A seguir viria a criação em MANO dos objetos necessários para a versão inicial de DEMO. Esta versão poderia, por exemplo, omitir aspectos e os níveis externos e de representação, a serem colocados em versões sucessivas posteriores. Em DEMO, ao contrário de MANO, o programador define tipos explicitamente. O compilador DEMO, praticamente poderá ser implementado como uma extensão, ao compilador MANO, para que ele entendesse declarações de tipos. Aspectos, vistas e representantes seriam extensões seguintes.

Tanto vistas, como representantes, para serem inseridos em DEMO, necessitariam ter um detalhamento de suas propriedades maior do que o que foi

dado aqui. Outro lado de pesquisa a ser abordado é a otimização de consultas. Se isto já era crucial em modelos de dados orientados a objetos, mais será no modelo de objetos do GEOTABA, onde não foram feitas concessões para garantir eficiência na consulta a coleções. Por exemplo, uma coleção de objetos de tipo *Pessoa* poderá ter objetos de qualquer classe que implemente esse tipo. Isto significa que um atributo *idade* em alguns desses objetos poderão estar implementados por um campo e em outros estarão implementados por métodos que calculam a idade. Uma consulta que use um predicado envolvendo esse atributo não poderá se valer apenas da comparação de valores armazenados, precisará conhecer o método que implementa *idade*.

Uma outra vertente de trabalhos e pesquisas se refere a criação de técnicas de modelagem baseadas no modelo de objetos proposto.

Concluindo, este trabalho fornece parâmetros e viabiliza pesquisas em diversas áreas de sistemas de programação orientados a objetos, submentendo-as a uma abordagem onde a manipulação semântica da informação é determinante.

Anexo A: Exemplo de Modelagem

Este anexo apresenta, como exemplo de emprego dos Diagramas de Objetos, uma modelagem dos objetos manipulados pelo EDC - Editor de Diagrama de Classes. A descrição do EDC pode ser obtida em [Matt90]. Aí também é encontrada uma especificação deste editor que pode ser usada para comparação com os Diagramas de Objetos.

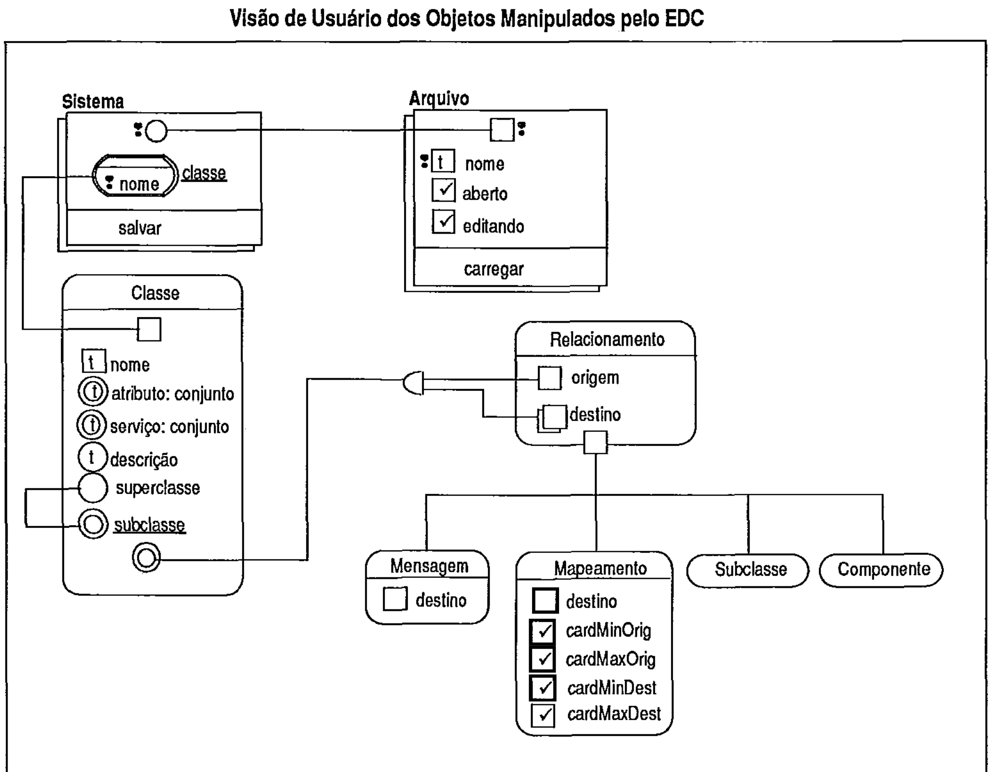


Figura A-1: Modelagem dos Objetos do EDC

A Figura A-1 mostra um Diagrama de Objetos modelando os objetos manipulados pelo usuário através do EDC, conforme descrito em [Matt90]. A classe Sessão não foi representada por não fazer sentido no caso do EDC ser implementado sobre o GEOTABA. Através da atual versão dos Diagramas de Objetos não é possível mostrar graficamente que cada tipo de dados especificado pode ser manipulado pelo usuário por representantes correspondentes. De acordo com o especificado em [Matt90], esses representantes seriam:

Diagrama de Classes, para Sistema;
Elemento Classe e Ficha de Classe, para Classe;
Ligação, para Relacionamento;

Ligação de Mensagem, para Mensagem;
Ligação de Subclasse, para Subclasse;
Ligação de Mapeamento, para Mapeamento;
Ligação de Componente, para Componente.

Os serviços de criação de objetos (*CriarSistema, CriarClasse, CriaArq,* etc.) não precisam ser especificados, pois os tipos *Sistema, Classe* e *Arquivo* herdam a criação padrão do tipo *Objeto*. Assim, para criar um sistema usa-se:

```
Sistema < >
```

Atribuir um nome ao sistema equivale a designar um arquivo para conter o sistema. Portanto, isto pode ser realizado através de uma atribuição ao atributo *arquivo* do sistema:

```
arquivo := Arquivo < nome ⇒ "arqsis1" >
```

Operações de inclusão e exclusão em coleções (*IncluirClasse, ExcluirClasse, AcrescentarAtributo, RemoverAtributo,* etc.) também não precisam ser especificadas. Por exemplo, *IncluirClasse(C)* pode ser realizada assim:

```
classe ⊕= C
```

Também não é necessário definir o protocolo para atribuição a atributos. Por exemplo, *DefinirNomeClasse(N)* seria expresso por

```
nome := N
```

A mesma expressão também substitui *NomeiaArq(N)*, embora esta pressuponha efeito colateral. Como a atribuição, no GEOTABA, também é uma mensagem, o efeito colateral poderá ser implementado no método correspondente à atribuição a nome de arquivo.

A ação de salvamento foi modelada no tipo *Sistema*, no lugar de em *Arquivo* como no original, por pura questão de estilo. É suposto que o usuário deseje salvar o sistema no qual está trabalhando em um arquivo, e não salvar o arquivo. Todavia, espera-se que a classe de implementação de *Arquivo* possua métodos equivalentes ao *SalvaArq* para gravar informação nova no arquivo. Isto não deve ser modelado neste diagrama, pois este se refere à visão (subesquema) que o usuário possui dos objetos manipulados pelo EDC, abstraindo-se da implementação destes. Para implementar o salvamento de sistema, o programador precisa ter uma visão dos objetos arquivos que inclua as ações necessárias ao salvamento. Ambas as visões de arquivo se referirão ao mesmo padrão-tipo *Arquivo*.

A observação acima também explica a omissão de atributos e serviços, justificáveis apenas para a implementação e não visíveis ao usuário, como é o

caso de *TemRelacionamento, Armazenado, IncluirMensagem, RemoverMensagem, IncluirMapeamento,* etc. Para o usuário incluir um relacionamento-mensagem, por exemplo, bastaria criá-lo. Como os atributos *origem* e *destino* não são opcionais, nessa criação eles devem ser inicializados com as classes adequadas, como em

```
Mensagem ‹origem⇒classe1; destino⇒classe2›
```

Pelo modelado no Diagrama da Figura A-1, isto provocaria a inclusão automática do relacionamento-mensagem nas coleções *relacionamento* de *classe1* e *classe2*. Portanto, o usuário não precisa ter acesso a um serviço especial para esta tarefa, como seria o caso do serviço *IncluirMensagem.*

Os atributos *PosInicial* e *PosFinal* também não participam desta modelagem por pertencerem, na verdade, aos representantes de interface com usuário (os elementos de um Diagrama de Classes).

O Diagrama de Objetos definiu duas coleções globais: *Sistema* e *Arquivo*. Os objetos de *Sistema* possuem dois atributos, denominados *arquivo* e *classe*. Como o atributo *arquivo* é designado como "único", cada sistema está associado a um único arquivo. Do mesmo modo, o atributo *sistema* de *Arquivo* indica que cada arquivo está associado a um único sistema. Como este atributo nunca é vazio, ele é uma chave candidata da coleção *Arquivo* e esta é um conjunto.

O atributo de composição (sublinhado) *classe* de *Sistema* indica que um sistema se compõe de classes e uma classe não pode pertencer a mais de um sistema. Além disso, um sistema não possui mais de uma classe com o mesmo nome (chave parcial *nome*). A remoção de classes deve possuir uma precondição, como em [Matt90], indicando que a classe *C* a ser removida não pode possuir subclasses.

O atributo *subclasse* de *Classe* está sublinhado, indicando composição, ou seja, uma classe não pode ser subclasse de mais de uma classe (hierarquia simples). A cardinalidade máxima do atributo *destino* de *Relacionamento* é redefinida nos tipos Mensagem e *Mapeamento*. Isto é permitido porque seria considerado como uma restrição adicional, já que um atributo não coleção inclui o protocolo de um atributo coleção. Isso é possível porque o tipo predefinido *Objeto* sabe responder adequadamente os serviços de *Coleção* (acréscimo, exclusão, teste de inclusão, iteração, etc.).

Os Diagramas de Objetos podem ser alterados facilmente para poderem modelar o nível de representação dos objetos. Esta modelagem, nesse exemplo, seria trivial. É importante notar, também, que a proximidade do ferramental de modelagem do GEOTABA e de especificação contido no método de

desenvolvimento de software com orientação a objetos descrito em [Matt90] aponta para a utilização desse método em ambientes com o GEOTABA.

Referências Bibliográficas

[ABIT91] SERGE ABITEBOUL, ANTHONY BONNER, **Objects and Views.** *Proceedings of the 1991 ACM SIGMOD Conference on Management of Data*, 238-247, 1991.

[AGUI92] T.C. AGUIAR, *A Estação de Trabalho TABA*. Dissertação de Tese de Doutorado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, março de 1992.

[ALAB88] B. ALABISO, **Transformations of Data Flow Analysis Models to Object-Oriented Design,** in *of Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, ACM, New York, pp. 335-353, 1988.

[ALBA85] A. ALBANO, L. CARDELLI, R. ORSINI, **Galileo: A Strongly Typed Interactive Conceptual Language.** *ACM Transactions on Database Systems* **10**:2 (230-260), June 1985.

[APPL85] *Inside Macintosh,* Apple Computer Corp., Addison-Wesley, Reading, Mass., 1985.

[ATKI83] MALCOLM P. ATKINSONS, P.J. BAILEY, K.J. CHISHOLM, W.P. COCKSHOT, R. MORRISON, **An Approach to Persistent Programming Language,** *Comput. J.* **26**:4, November 1984.

[ATKI87] MALCOLM P. ATKINSONS, O. PETER BUNEMAN, **Types and Persistence in Database Programming Language,** *ACM Comp. Surv.* 19:2, June 1987.

[ATKI90] MALCOLM P. ATKINSONS et al. **The Object-Oriented Database System Manifesto,** *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data,* Atlantic City, NJ, May 1990.

[BANC88] F. BANCILHON, **The Design and Implementation of O2, an O-O Database System.** *Proc of the II International Workshop on O-O DB Systems,* 1988.

[BANC90] F. BANCILHON, P. BUNEMAN, *Advances in Database Programming Languages*, ACM Press, Addison-Wesley, Reading,Massachussets, 1990.

[BANE87] J. BANERJEE, H.T. CHOU, J.F. GARZA, W. KIM, D. WOELK, N. BALLOU, **Data Model Issues for Object Oriented Applications.** *ACM Transactions on OIS,* vol 5, n^{0}. 1, jan 1987.

[BENB84] I. BENBASAT, Y. WAND, **A Structured Approach to Designing Human-Computer Dialogues.** *Int. Journal of Man-Machine Studies* **21**, pp. 105-126, 1984

[BETT87] W. BETTS, et al., **Goals and Objectives for User Interface Software,** *Computer Graphics 21,* 2 (Apr 87), 73-78.

[BILJ88] W. BILJON, **Extending Petri Nets for Specifying Man-Machine Dialogues.** *Int. Journal of Man-Machine Studies* **28**, pp. 437-455, 1988.

[BLAC86] A. BLACK, N. HUTCHINSON, E. JUL, H. LEVY, **Object Structure in the Emerald System,** *Object-Oriented Programming Systems, Languages and Applications, Portland, OR, Sept 1986. Proceedings SIGPLAN NOTICES, Nov 1986,* pp. 78-86.

[BLAC87] A. BLACK, N. HUTCHINSON, E. JUL, H. LEVY, L. CARTER, **Distribuction and Abstract Types in Emerald.** *IEEE Transactions on Software Engineering* **13**:1, 65-76, January 1987.

[BLOO87] T. BLOOM, S.B. ZDONIK, **Issues in the Design of Object-Oriented Database Programming Languages.** in [MEYR87].

[BLUM88] H. BLUM, L.A. GONÇALVES, M.A. ROSSATO, L.C. COSTA, *O Desenvolvimento de um Protótipo de Sistema de Banco de Dados Orientado a Objetos,* Relatório Técnico do Programa de Engenharia de Sistemas, COPPE/UFRJ, agosto 1988.

[BOBR88] D.G. BOBROW et al. **Common LISP Object System Specification,** *SIGPLAN Notices* 23, September 1988.

[BOOC91] GRADY BOOCH, *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[BORN81] A. BORNING, **The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory.** *ACM Transactions on Programming Languages and Systems* 3:4 (Outubro), pp. 353-387, 1981.

[BORN86] A. BORNING, R. DUISBERG, **Constraint-Based Tools for Building User Interfaces.** *ACM Trans. on Graphics* **5**:4, 1986.

[BORN87] H. BORNSTEIN, **HYPERCARD: Shaking by association.** *The MACazine,* December 1987.

[BRET88] R. BRETL et al. **The GemStone Data Management System.** in [KIM88].

[BROD81] M. L. BRODIE, **On Modelling Behavioural Semantics of Databases.** *Proceedings of the 6th International Conference on Very Large Data Bases*, pp. 32-42, setembro de 1981.

[BROD82] M. L. BRODIE, E. SILVA, **Active and Passive Component Modelling: ACM/PCM.** *Proceedings of the IFIP TC 8 Working Conference on Comparative Review of Information Systems Design Methodologies*, maio de 1982.

[BROD84] M. L. BRODIE, J. MYLOPOULOS, J. SCHMIDT *On Conceptual Modelling*, Springer-Verlag, 1984.

[BROD84B] M. L. BRODIE, D. RIDJANOVIC, **On the Design and Specification of Database Transactions.** em [BROD84].

[BROD86] M. L. BRODIE, J. MYLOPOULOS (eds.), *On Knowledge Base Management Systems*, Springer-Verlag, Berlin, Germany, 1986.

[BUNE86] O. PETER BUNEMAN, MALCOLM P. ATKINSON, **Inheritance and Persistence in Database Programming Language,** *Proceedings of the 1986 ACM SIGMOD Conference on Management of Data,* 4-15, 1986.

[BUTT91] P. BUTTERWORTH, A. OTIS, J. STEIN, **The GemStone Object Database Management System.** *Communications of the ACM* **34**:10(64-77), October 1991.

[BUXT83] W. BUXTON et al. **Towards a Comprehensive User Interface Management System.** *Computer Graphics 17:3,* Jul 1983, pp. 35-42.

[CAMA91] C.S.P. CAMARGO, J. LISBOA FILHO, L.C.M. MONTE, *Uma Implementação de Relações em Smalltalk.* Relatório Técnico do Programa de Engenharia de Sistemas e Computação. ES-231-90, novembro de 1990, 36 pp.

[CAMA92] CLAUDIA SUSIE PINA CAMARGO, **Flecha: Um Editor Gráfico Cooperativo para o Modelo de Objetos do GEOTABA.** Tese de M.Sc. submetida à COPPE/UFRJ, outubro de 1992.

[CARD85] L. CARDELLI, R. PIKE, **Squeak: a Language for Communicating with Mice.** *Proceedings of SIGGRAPH'85* (San Francisco, Calif., Julho) **19**:3, pp.199-204, 1985.

[CARE88] M.J. CAREY, D.J. DEWITT, S.L. VANDENBERG, **Storage Management for Objects in EXODUS.** *Proceedings of the ACM SIGMOD Conference on Management of Data,* 413-423, Chicago, IL, June 1988.

[CATT91] RODERIC GEOFFREY GALTON CATTELL, *Object Data Management: Object-Oriented and Extended Relational Database Systems,* Addison-Wesley Publishing Company, Inc. 1991.

[CERI90] S. CERI et al. **ALGRES: An Advanced Database Sysem for Complex Applications.** *IEEE Software*, July 1990.

[CHEN76] P.P. CHEN, **The Entity-Relationship Model - Toward a Unified View of Data.** *ACM Trans. Database Syst.* **1**:1, pp. 9-36, março 1976.

[CHI85] U. CHI, **Formal Specification of User Interfaces: a Comparision and Evaluation of Four Axiomatic Approaches.** *IEEE Transaction on Software Engineering* **11**, pp. 671-685, 1985.

[CLÉM88] D. CLÉMENT, J. INCERPI, *Specifying the Behavior of Graphical Objects Using Esterel.* Rapports de Recherce No 836, INRIA, (Abril) 1988.

[COAD90] PETER COAD, EDWARD YOURDON, *Object-Oriented Analysis,* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[COCK84] W. COCKSHOT, M. ATKINSON, K. CHISHOLM, P. BAILEY, R. MORRISON, **Persistent Object Management System.** *Software - Practice and Experience* **14**:1, January 1984.

[CODD70] E.F. CODD, **A Relational Model for Large Shared Data Banks.** *Comunications of ACM* **13**:6, June 1970.

[CODD79] E.F.CODD, **Extending the Database Relational Model to Capture More Meaning.** *ACM Transactions on Database Systems* **4**:4, December 1979.

[CODD80] E.F. CODD, **Data Models in Database Management.** *Proceedings of the ACM SIGMOD Conference on Management of Data,* **11**:2, junho de 1980.

[CONK87] J. CONKLIN, **Hypertext: An Introduction and Survey.** *IEEE Computer* (17-41), September 1987.

[COUT85] J.COUTAZ, **Abstractions for User Interface Design.** *Computer,* Set 1985, pp. 21-34.

[DADA86] P. DADAM et al. **A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies.** in *Proceedings of the ACM SIGMOD Conference*, Washington D.C., May 1986.

[DAHL66] O-J. DAHL, K. NYGAARD, *SIMULA - an Algol Based Simulation Language. Comunications of ACM* **9**:9, 1966.

[DANC87] J. DANCE et al. **The Run-Time Structure of UIMS-Supported Applications.** *Computer Graphics* 21:2, Abr 1987, pp. 97-101.

[DATE90] C.J. DATE, *Introduction to Database Systems,* vol. 1, 5th ed., Addison-Wesley, Reading, Mass., 1990.

[DAYA84] U. DAYAL, H.Y. HWANG, **View Definition and Generalization for Database Integration in a Multidatabase System.** *IEEE Trans. Soft. Eng.* **SE-10**:6, pp. 628-644, 1984.

[DAYA86] U. DAYAL, J. SMITH, **PROBE: A Knowledge-Oriented Database Management System.** in [BROD86].

[DEHN80] W. DEHNING et al. **The Adaption of Virtual Man-Computer Interface to User Requirements in Dialogs.** *Lecture Notes in Computer Science.* Berlin: Springer-Verlag, 1980.

[DERE76] F. DE REMER, **Review of Formalisms and Notations.** in BAUER (ed.) *Compiler Construction*, Springer-Verlag, Berlin, pp. 37-56, 1976.

[DEUX90] O. DEUX et al., **The Story of O_2.** *IEEE Trans. on Knowledge and Data Engineering,* Vol. 2, N. 1, March 1990.

[DEUX91] O. DEUX et al., **The O_2.** *Communications of the ACM* **34**:10(34-48), October 1991.

[DEWI90] DeWilt, D. Maier, D. Fritterssoch, P. Velez, F. "A study of these alternative workstation-server architectures for Object-Oriented Database Systems", in Proc. of the 16th Very Large Databases Conference, pp. 107-121, 1990.

[DIGI86] *Smalltalk/V: Tutorial and Programming Handbook,* Digitalk Inc., Los Angeles, Calif., 1986.

[DITT86a] K.R. DITTRICH, U. DAYAL (eds.): *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* IEEE Computer Science Press, 1986.

[DITT86b] K.R. DITTRICH, **Object-Oriented Database Systems: The Notions and the Issues**, in [DITT86a].

[DITT88] K.R. DITTRICH (ed.), **Advances in Object-Oriented Database Systems: Second International Workshop on Object-Oriented Database Systems,** Bad Münster, Germany, September 1988, Computer Science Lecture Notes 334, Springer-Verlag, Berlin, 1988.

[ELLI90] M. ELLIS, BJARNE STROUSTRUP, *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, Massachussets, 1990.

[ELMA89] RAMEZ ELMASRI, SHAMKANT B. NAVATHE, *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

[ENDE84] G. ENDERLE, **Report on the Interface of the UIMS to the Application.** Working Group Report. *Computer Graphics Forum 3,* pp. 175-179, 1984.

[FAIR85] RICHARD FAIRLEY, *Software Engineering Concepts,* McGraw-Hill, pp. 148-151, 1985.

[FISH87] FISHMAN et al., **Iris: an Object-Oriented Database Management System,** *ACM Transactions on Office Information Systems* **5**:1, January 1987.

[FISH88] FISHMAN et al., **Overview of the Iris DBMS**, in [KIM88].

[FLEC87] M.A. FLECCHIA, R.D. BERGERON, **Specifying Complex Dialogs in ALGAE.** in *Proceedings of the ACM CHI+GI'87 Conference* (Toronto, Ontario, Canadá, Abril), ACM, New York, pp. 229-234, 1987.

[FOLE84] J. FOLEY, **Managing the Design of User-Computer Interfaces.** *Proceedings of the Fifth Annual NCGA Conference and Exposition.* Anaheim, CA, vol II, Mai 1984, pp. 436-451.

[FOLE87] J. FOLEY, **Transformation on a Formal Specification of User-Computer Interfaces.** *Computer Graphics* 21:2, Abr 1987, pp. 109-113, 1987.

[FORD88] S. FORD et al. **ZEITGEIST: Database Support for Object-Oriented Programming.** in [DITT88].

[GIBB83] S. GIBBS, D. TSICHRITZIS, **A Data Modeling Approach for Office Information Systems.** *ACM TOIS* 1:4, pp. 299-319, outubro de 1983.

[GOLD83] A. GOLDBERG, D. ROBSON, *Smalltalk-80 - The language and its implementation*, New-York, Addison-Wesley, 1983.

[GREE85a] M. GREEN, **Report on Dialogue Specification Tools.** in [PFAF85], pp. 9-20.

[GREE85b] M. GREEN, **The University of Alberta User Interface Management System.** *Computer Graphics 19:3,* 1985, pp. 205-213.

[GREE86] M. GREEN, **A Survey of Three Dialog Models.** *ACM Transaction on Graphics* **5**:3 (Julho), pp. 244-275.

[GREE87] M. GREEN, **Directions for User Interface Management System Research.** *Computer Graphics* 21:2, Abr 1987, pp. 113-116.

[GROS87] M. GROSSMAN, R. EGE, **Logical Composition of Object-Oriented Interfaces.** in [MEYR87], Out 1987, pp. 295-306.

[HAMM81] M. HAMMER, D. MCLEOD, **Database Description with SDM: A Semantic Database Model.** *ACM Transaction on Database Systems* 6:3, September 1981.

[HAMM86] K. HAMMER et al. **Automating the Generation of Interactive Interfaces.** *23rd Design Automation Conference,* IEEE, 1986.

[HART84] H. REX HARTSON, DEBORAH HIX (JOHNSON), R.W. EHRICH, **A Human-Computer Dialogue Management System.** In *Proceedings of INTERACT'84, First IFIP Conference on Human-Computer Interaction* (Londres, Setembro), International Federation for Information Processing, pp. 57-61, 1984.

[HART85] H. REX HARTSON (ed) *Advances in Human-Computer Interaction,* Ablex Publishing Corp., Norwood, NJ, 1985.

[HART89] H. REX HARTSON, DEBORAH HIX **Human-Computer Interface Development: Concepts and Systems for its Management,** *Computing Surveys* **21**:1, Mar 1989, pp 5-92.

[HART90] H. REX HARTSON, DEBORAH HIX, T.M. KRALY, **Developing Human-Computer Interface Models and Representation Techniques.** *Software - Practice and Experience,* **20**:5 (Maio), pp. 425-457, 1990.

[HAYE83] P.J. HAYES, P.A. SZEKELY, **Graceful Interaction Through the COUSIN Command Interface.** *International Journal of Man-Machine Studies* **19**:3 (Setembro), pp. 285-305, 1983.

[HAYE85a] P.J. HAYES, P.A. SZEKELY, R.A. LERNER, **Design Alternatives for User Interface Management Systems Based on Experience with COUSIN.** in *CHI'85 Proceedings* (San Francisco, Calif., Abril), ACM, New York, pp. 14-18, 1985.

[HAYE85b] P.J. HAYES, **Executable Interface Definitions Using Formal-Based Interface Abstractions.** in [HART85], pp. 161-190, 1985.

[HEND90] BRIAN HENDERSON-SELLERS, JULIAN M. EDWARDS, **The Object-Oriented Systems Life Cycle.** *Communications of ACM* **33**:9, pp. 142-159, setembro de 1990.

[HILL86] R. HILL, **Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction - The Sassafras UIMS.** *ACM Trans. on Graphics* **5**:3 (Julho), pp. 179-210, 1986.

[HILL87a] R. HILL, **Some Important Features and Issues in User Interface Management Systems.** *Computer Graphics 21:2,* Abr 1987, pp. 116-120.

[HILL87b] R. HILL, **Event-Response Systems - A Technique for Specifying Multi-Threaded Dialogues.** *Proceedings of CHI+GI'87,* Abr 1987.

[HITC85] P. HITCHCOCK, et al., **The Use of Database for Software Engineering.** *IV British Conference on Databases,* 1985.

[HUDS86a] S. HUDSON, R. KING, **A Generator of Direct Manipulation Office Systems.** *ACM Trans. on Office Information Systems 4:2,* Abr 1986, pp. 132-163.

[HUDS86b] S. HUDSON, R. KING, **CACTIS: A Database System for Specifying Functionally-Defined Data.** in [DITT86a].

[HUDS87] S. HUDSON, **UIMS Support for Direct Manipulation Interfaces.** *Computer Graphics* 21:2, Abr 1987, pp. 120-124.

[HUDS89] S. HUDSON, R. KING, **CACTIS: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database System.** *ACM Transactions on Database Systems* **14**:3, September 1989.

[HULL87] RICHARD HULL, ROGER KING, **Semantic Database Modeling: Survey, Applications, and Research Issues,** *ACM Computing Surveys* **19**:3, pp.201-260, Setembro 1987.

[ICHB79] J.H. ICHBIAH, J.G.P. BARNES, J.C. HELIARD, B. KRIEG-BRUCKNER, O. RUBINE, B.A. WICHMANN, **Rationale of the Design of the Programming Language Ada.** *ACM SIGPLAN Notices* **14**:6, 1979.

[IRON61] IRONS, E. **A Syntax Directed Compiler for Algol 60.** *Communications of the ACM* **4**:1, pp. 51-55.

[JACO85] R.J.K. JACOB, **An Executable Specification Technique for Describing Human-Computer Interaction.** in [HART85], pp. 211-244, 1985.

[JACO86] JACOB, R.J.K. **A Specification Language for Direct-Manipulation User Interfaces.** *ACM Trans. on Graphics* **5**:4 (Outubro), pp. 283-317, 1986.

[KAEH83] TED KAEHLER, GLENN KRASNER, **LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems.** *Smalltalk-80 Bits of History, Words of Advice,* Addison-Wesley, Reading, Massachussets, 1983.

[KEMP88] A. KEMPER, M. WALLRATH, **A Uniform Concept for Storing and Manipulating Engineering Objects** in [DITT88][DITT88].

[KIM88] W. KIM AND F. LOCHOVSKY Eds. *Object-Oriented Concepts, Databases, and Applications.* ACM and Addison-Wesley, Reading, Massachussets, 1988.

[KIM89] W. KIM, **A Model for Queries for Object-Oriented Databases.** *Proceedings of the 15th Conference on Very Large Database Systems,* 1989.

[KIM90a] W. KIM, et al., **Architecture of the ORION Next-Generation Database System.** *IEEE Trans. on Knowledge and Data Engeneering* **2**:1(44-62), March 1990.

[KIM90b] W. KIM, et al., *Introduction to Object-Oriented Database,* The MIT Press, 1990.

[KING85] R. KING, D. MCLEOD, **A Database Design Methodology and Tool for Information Systems.** *ACM TOIS* 3:1, janeiro de 1985.

[KOIV88] M. KOIVUNEN, M. MANTYLA, **HutWindows: An Improved Architecture for a User Interface Management System.** *IEEE Computer Graphics & Applications*, Jan 1988, pp. 43-52.

[LANT87] K.A. LANTZ, P.P. TANNER, C. BINDING, K.T. HUANG, A. DWELLY, **Reference Models, Window Systems, and Concurrency.** in [OLSE87b], pp. 87-97, 1987.

[LECL88] C. LECLUSE, P.RICHARD, F. VELEZ, **O_2, an Object-Oriented Data Models.** *Proceedings of the ACM SIGMOD Conference on Management of Data,* 424-433.

[LECL89] C. LECLUSE, P.RICHARD, **The O_2 Database Programming Language.** *Proceedings of the 15th Conference on Very Large Database Systems,* Amsterdam, 1989.

[LEDG81] HENRY LEDGARD, *Ada, an Introduction*. Springer-Verlag, 1981.

[LISB92] JUGURTA LISBOA FILHO, *MANO: Linguagem de Manipulação de Objetos do ProtoGEO e seu Processador.* Dissertação de Tese de Mestrado, Programa de Engenharia de Sistemas e Computação COPPE/UFRJ, outubro de 1992.

[LISK77] B. LISKOV, A. SNYDER, R. ATKINSON, C. SCHAFFERT, **Abstracion Mechanisms in CLU**, *Communications of the ACM*, 20:8, 1977.

[LOWG88] J. LÖWGREN, **History, State and Future of User Interface Management Systems.** *SIGCHI Bulletin* 20:1 (Jul 1988), 32-44.

[LYNG84] P. LYNGBAEK, D., MCLEOD, **Object Management in Distributed Information Systems.** *ACM TOIS* 2:2, abril de 1984.

[McAl86] D. McAllester, R. Zabih, **Boolean Classes.** *Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, Sept 1986. Proceedings SIGPLAN NOTICES, Nov 1986, 472-482.

[MAIE86] D. MAIER, et al. **Development of an object-oriented DBMS.** *Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, Sept 1986. Proceedings SIGPLAN NOTICES, Nov 1986, 214-223.

[MATT89] M.L.Q. MATTOSO, J. M. SOUZA, L. GONÇALVES, C. DEGRAZIA E M. ROSSATTO, *GEOTABA: O Sistema de Gerência de Objetos da Estação TABA*, Relatório técnico Es 179/88 do Programa de Engenharia de Sistemas - COPPE/UFRJ, 1989.

[MATT90] A.L.Q. MATTOSO, *TABA_OBJ: Um Ambiente de Desenvolvimento de Software com Orientação a Objetos,* Tese de M.Sc. submetida ao Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, agosto de 1990.

[MATT91] M.L.Q. MATTOSO, *Banco de Dados e Paralelismo: Um Levantamento.* Relatório Técnico ES-244 Programa de Engenharia de Sistemas e Computação COPPE/UFRJ, 1991.

[MATT93] M.L.Q. MATTOSO, *Aspectos de Paralelismo na Gerência de Dados e Objetos no GEOTABA.*. Tese de D.Sc. submetida ao Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, abril de 1993.

[MELO87] R. N. MELO, *Bancos de Dados Não Convencionais.* VI Jornada de Atualização de Informática - SBC, Salvador, julho de 1987.

[MEYR87] N. MEYROWITZ, (ed) **Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87).** Orlando, Florida, *SIGPLAN Notices* 22:12, Dez 1987.

[MONT89] L.C.M. MONTE, A.L.Q. MATTOSO, **HiperFicha: Hipermídia para Desenvolvimento de Software.** *Anais do XXII Congresso Nacional de Informática*, São Paulo, setembro de 1989, pp. 108-112.

[MONT91a] L.C.M. MONTE, **Uma Representação Gráfica para o Modelo de Objetos do GEOTABA,** *anais da XVII Conferência Latinoamericana de Informática*, Caracas, julho de 1991.

[MONT91b] L.C.M. MONTE, **O Modelo de Objetos do GEOTABA,** *anais do Workshop de Banco de Dados Não Convencionais*, Coppe/UFRJ, Rio de Janeiro, pp. 14-17, julho de 1991.

[MONT92a] L.C.M. MONTE, *Comunicação Homem-Computador por Manipulação Direta.* Relatório Técnico do Departamento de Computação da Universidade Federal Fluminense, 1992.

[MONT92b] L.C.M. MONTE, J. LISBOA FILHO, M.L.Q. MATTOSO, J.M. DE SOUZA, **Contribuições do ProtoGEO ao Projeto do SGBDOO GEOTABA.** *Anais do VII Simpósio Brasileiro de Banco de Dados.* Porto Alegre, maio de 1992.

[MONT93] L.C.M. MONTE, *Modelagem da Comunicação Homem-Computador.* Relatório Técnico do Departamento de Computação da Universidade Federal Fluminense a ser publicado, 1993.

[MORA81] T.P. MORAN, **The Command Language Grammar: a Representation for the User Interface of Interactive Computer Systems.** *Int. Journal of Man-Machine Studies* **15**, pp. 3-51, 1981.

[MOSS88] J.E.B. MOSS, S. SINOFSKY, **Managing Persistent Data with Mneme: Designing a Reliable Shared Object Interface.** in [DITT88].

[MOSS88b] J.E.B. MOSS,**Object-Orientation as Catalyst for Language Database Integration.** in [KIM88].

[MOSS90] J.E.B. MOSS, **Designing of the Mneme Persistent Object Store.** *Transactions on Office Information Systems* **8**:2, April 1990.

[MYER83] B.A. MYERS, **Incense: a System for Displaying Data Structures.** *Computer Graphics* **17**:3 (Julho), pp. 115-125, 1983.

[MYER87] B.A. MYERS, **Gaining General Acceptance for UIMSs,** *Computer Graphics 21*:2, Abr 1987, pp. 130-134.

[MYER88] B.A. MYERS, *Creating User Interfaces by Demonstration.* Academic Press, Inc., San Diego, 1988.

[NELS85] G. NELSON, **Juno, A Constraint-Based Graphics System.** *Computer Graphics* **19**:3, 1985.

[OLSE83] D.R. OLSEN JR., E. DEMPSEY, **SYNGRAPH: A Graphic User Interface Generator.** *Computer Graphics* 17:3, Jul 1983, pp. 43-50.

[OLSE87a] D.R. OLSEN JR., **Larger Issues in User Interface Management.** *Computer Graphics 21:2,* Abr 1987, pp. 134-137.

[OLSE87b] D.R. OLSEN JR., **ACM SIGGRAPH Workshop on Sotware Tools for User Interface Management - workshop summary**, *Computer Graphics 21:2*, Abr 1987, pp. 71-72.

[PARN69] D. PARNAS, **On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System.** *Proc 24th National ACM Conference*, 1969.

[PETE81] J. PETERSON, *Petri Nets Theory and the Modelling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[PFAF85] G. PFAFF (ed), *User Interfaces Management Systems,* Springer-Verlag, Berlin, 1985, 224 pg.

[PILO83] M. PILOTE, **A Programming Language Framework for Designing User Interfaces.** SIGPLAN Symposium on Programming Language Issues in Software Systems. *SIGPLAN Notices* **18**:6, pp. 118-136, 1983.

[RHYN87] J. RHYNE et al. **Tools and Methodology for User Interface Development.** *Computer Graphics* 21:2, Abr 1987, pp. 78-87.

[RICH91] JOEL RICHARDSON, PETER SCHWARCZ, **Aspects: Extending Objects to Support Multiple, Independent Roles.** *Proceedings of the 1991 ACM SIGMOD Conference on Management of Data,*Denver, Colorado, May 1991, 298-307.

[ROCH88] A. R. C. ROCHA, J. M. SOUZA, *TABA: Uma Estação de Trabalho para o Engenheiro de Software.* Relatório Técnico ES-145/88, UFRJ COPPE Sistemas, 1988.

[ROCH90] A.R.C. DA ROCHA, J.M. SOUZA, T.C. AGUIAR, **TABA: A Heuristic Workstation for Software Development,** *COMPEURO 90;* Tel Aviv, Israel, maio 1990.

[ROUS91] Roussopoulos, N. & Delis, A. "Modern Client-Server DBMS Architectures", SIGMOD RECORD, Vol. 20 (3), pp. 52-61, Sept. 1991.

[ROZE83] G. ROZEMBERG, R. VERRAEDT, **Subset Languages of Petri Nets.** in G. ROZEMBERG (ed) *Selected Papers from the 3rd European Workshop on the Theory and Applications of Petri Nets.* Berlin, Springer-Verlag, 1983.

[RUMB87] J. RUMBAUGH, **Relations as Semantic Constructs in an Object-Oriented Programming Language.** *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* Orlando, Florida, October, 1987.

[SCHA86a] C. SCHAFFERT, P. O'BRIEN, B. BULLIS, **Persistent and Shared Objects in Trellis/Owl.** in [DITT86a].

[SCHA86b] C. SCHAFFERT, T. COOPER, B. BULLIS, M. KILIAN, C. WILPORT, **An Introduction to Trellis/Owl.** in *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, September 1986..

[SCHU85] A. SCHULERT, et al. **ADM - a Dialog Manager.** *Proc. CHI 85,* ACM, New York, 1985, pp. 177-183.

[SCHW86] P. SCHWARZ et al. **Extensibility in the Starburst Database System** in [DITT86a].

[SCIO89] E. SCIORE, **Object Specializations.** *ACM TOIS* 7:2, Apr 89, 103-122.

[SCOT88] SCOTT, M. & YAP, S. **A Grammar-Based Approach to the Automatic Generation of User-Interface Dialogues.** *Proceedings of SIGCHI 88*, pp. 73-78, 1988.

[SHAW80] SHAW, A. **On the Specification of Graphics Command Languages and their Processors.** in GUEDJ et al. (eds) *Methodology of Interaction.* Amsterdam: Elsevier, 1980.

[SHIP81] D. SHIPMAN, **The Functional Data Model and the Data Language Daplex.** *ACM Trans. Database Syst.* **6**:1, pp. 140-173, março 1981.

[SHNE83] BEN SHNEIDERMAN, **Direct Manipulation: A Step Beyond Programming Languages,** *IEEE Computer* 16,8, Aug. 1983.

[SHYY91] Y.M. SHYY, S.Y.W. SU, **K: A High-level Knowledge Base Programming Language for Advanced Database Applications,** *SIGMOD RECORD* Vol. 20, (2), Jun 1991.

[SIMO87] L. SIMÕES, J. MARQUES, **IMAGES - An Object Oriented UIMS. in** *Proceedings of INTERACT'87*, **Elsevier Science Publishers B V, 1987.**

[SMIT77] J.B. SMITH, D.C.P. SMITH, **Database Abstractions: Aggregation and Generalization.** *ACM Transactions on Database Systems* 2:2, June 1977.

[SMIT87] K. SMITH, S. B. ZDONIK, **Intermedia: A Case Study of the Differences Between Relational and Object Oriented Database Systems.** in [MEYR87], outubro de 1987.

[SNYD86] A. SNYDER, **Encapsulation and Inheritance in Object-Oriented Programmng Languages.** *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, Portland, Oregon, September 1986.

[SOUZ90] JANO M. DE SOUZA et al., *"Uma Estação de Trabalho para Desenvolvimento de Software"*, Relatório Técnico do Prog. Eng. Sist. COPPE/UFRJ, 1990.

[STEF86a] M.J. STEFIK, D.G. BOBROW, K.M. KAHN, **Integrating Access-Oriented Programming into a Multiparadigm Environment.** *IEEE Software,* Janeiro 86, pp. 170-178, 1986.

[STEF86B] M.J. STEFIK, D.G. BOBROW, **Object-Oriented Programming: Themes and Variations.** *AI Magazine* 6:4, pp. 40-62, 1986.

[STON86] M. STONEBRAKER, L.A. ROWE, **The Design of POSTGRES** in *Proceedings of the ACM SIGMOD Conference on Management of Data,* Washington, D.C., 1986.

[STON87] M. STONEBRAKER, L.A. ROWE, **The POSTGRES Data Model.** *Proceedings of the 13th Conference on Very Large Database Systems (VLDB),* Brighton, England, September 1987.

[STON90] M. STONEBRAKER, **Third Generation Database System Manifesto,** *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data,* Atlantic City, NJ, May 1990.

[STRO86] BJARNE STROUSTRUP, *The C++ Programming Language*, Addison-Wesley, Reading, Massachussets, 1986.

[STRO88] BJARNE STROUSTRUP, **What is Object-Oriented Programming?** *IEEE Software* 5:3, May 1988.

[STRU85] H. STRUBBE, **Report on Role, Model, Structure and Construction of a UIMS.** in [PFAF85], pp.3-8.

[TAKA88] T. TAKAHASHI, *Introdução à Programação Orientada a Objetos*, III EBAI, Curitiba, janeiro de 1988.

[Tann87] P. Tanner, **Multi-Thread Input.** *Computer Graphics* 21:2, Abr 1987, pp. 142-145.

[TEOR86] TOBY J. TEOREY, DONGQING YANG, JAMES P. FRY, **A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model,** *ACM Computing Surveys* 18:2, pp. 197-222, Junho 1986.

[TERR89] CHRIS TERRIS, **Objects Facilitate Modular, Reusable Code,** *EDN*, Nov 9, 1989, 85-90.

[THOM83] J.J. THOMAS, G. HAMLIN, **Graphical Input Interaction Techniques** (GIIT) workshop summary, *Computer Graphics 17,* 1 (Jan 83), 5-30.

[TSIC78] D. C. TSICHRITZIS, A. KLUG (eds.), **The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems.** *Information Systems* 3, 1978.

[TSIC82] D. C. TSICHRITZIS, F. H. LOCHOVSKY, *Data Models.* Prentice-Hall Software Series, Englewood Cliffs, New Jersey, 1982.

[TURN86] D. TURNER, **An Overview of Miranda.** *ACM SIGPLAN Notices* **2**:12, 158-167, December 1986.

[ULLM89] J. ULLMANN, *Principles of Database and Knowledge-Based Systems*, vol. 1 e 2, Computer Science Press, Rockville, Maryland, 1987.

[WASS85A] A.I. WASSERMAN, D.T. SHEWMAKE, **The Role of Prototypes in the User Software Engineering (USE) Methodology.** in [HART85], pp. 191-210, 1985.

[WASS85B] A.I. WASSERMAN, **Extending State Transition Diagrams for the Specification of Human-Computer Interaction.** *IEEE Transaction on Software Engineering* **11**:8 (Agosto), pp. 699-713, 1985.

[WILL87] G. WILLIAMS, **HyperCard.** *BYTE Magazine,* December 1987.

[WOOD70] W.A. WOODS, **Transition Network Grammars for Natural Language Analysis.** *Communications of the ACM* 10:13 (Outubro), pp. 462-471, 1970.

[X11 87] *X11 Toolkit,* MIT, Project Athena, (Fevereiro), 1987.

[YOUN88] M. YOUNG, R. TAYLOR, D. TROUP, C. KELLY, **Design Principles Behind Chiron: A UIMS for Software Environments.** *IEEE 10th Int. Conf. on Software Engineering*, Computer Society Press, Washington, DC, Abr 1988, pp. 367-376.

[ZDON89] S.B. ZDONIK, D. MAIER (eds.), *Readings in Object-Oriented Database Systems,* Morgan-Kaufmann, San Mateo, California, 1989.