Utilização de um Banco de Dados Orientado a Objetos em um Ambiente de Desenvolvimento de Software

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pela Srta. **Carmem Satie Hara** e aprovada pela Comissão Julgadora.

Campinas, 5 de outubro de 1990

Orientador: _____
Prof. Dr. Geovane Cayres Magalhães

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Sumário

Muito tem se pesquisado sobre banco de dados orientados a objetos (BDOO). Alguns protótipos de sistemas de gerenciamento para estes bancos de dados foram recentemente desenvolvidos, porém há uma carência de experiências práticas nesta área. Esta tese procurou justamente experimentar com um destes protótipos utilizando uma aplicação apropriada à orientação a objetos, com o objetivo de identificar problemas na modelagem e implementação, bem como verificar se as facilidades oferecidas pelo protótipo são adequadas à solução do problema.

Damokles é um protótipo de BDOO desenvolvido na Universidade de Karlsruhe com o propósito de dar suporte a ambientes de projeto. Seu modelo de dados estende o modelo entidade-relacionamento com os conceitos de composição de objetos e versões, além de possuir um tipo de dado destinado ao armazenamento de campos longos. Ele foi utilizado na implementação do ADS A_HAND (Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados), que está atualmente sendo desenvolvido pelo Departamento de Ciência da Computação da Unicamp.

Para o projeto conceitual do ambiente, criou-se um modelo de diagramação baseado no modelo de dados Damokles. Representações de alguns conceitos não suportados pelo mesmo foram incorporados a este sistema a fim de tornar a experiência mais rica no sentido de testar como se comporta o mapeamento de um modelo mais poderoso para o do Damokles.

Abstract

Object oriented database systems (OODBS) have been researched in a great deal. Recently, some prototypes of these systems were developed, but there is a lack of practical experiences in this area. This thesis reports an experience in using one of these prototypes for an application suited to the object orientation, in order to identify modelling and implementation problems, as well as to verify if the prototype features were adequate in the problem solution.

Damokles is a prototype of an OODBS developed at the Karlsruhe University. It is directed to design environments and its data model extends the entity-relationship model with the concepts of objects and versions composition, and also the long-field data type. It was used in the implementation of a software development environment (SDE), A_HAND, being developed at the Computer Science Department of UNICAMP.

A diagrammatic model, based on Damokles, was created for the modelling of A_HAND. Concepts not present in the database were used to enrich the experience with the mapping from a more comprehensive model to the Damokles one.

Aos meus pais e
ao Kendi

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este trabalho é uma experiência em utilizar banco de dados orientados a objetos para ambientes de desenvolvimento de software. Neste capítulo é colocado o contexto no qual o trabalho foi desenvolvido, bem como a apresentação do seu conteúdo.

1.1 Definição do Contexto

Bancos de dados tem sido utilizados para o armazenamento e gerenciamento de grande volume de dados em diversos tipos de aplicações. Recentemente, devido o grande avanço na tecnologia de hardware, software antes proibitivos devido ao seu tamanho ou complexidade tornaram-se tecnicamente viáveis. Estamos nos referindo às aplicações ditas não convencionais, tais como sistemas de engenharia e de automação de escritório, que trouxeram novas expectativas quanto às facilidades oferecidas por um banco de dados. Visto o alto custo de desenvolvimento de tais sistemas, para que eles se tornem economicamente factíveis, é necessário que sua vida útil seja mais longa, através da incorporação de características como portabilidade, adaptabilidade e manutenibilidade [DLi87]. Portabilidade é a facilidade com que um software pode ser transportado para máquinas diferentes; adaptabilidade é a qualidade do sistema em adequar-se a novos requisitos; e manutenibilidade corresponde à facilidade com que eventuais erros podem ser corrigidos.

Para garantir a incorporação destas características nos sistemas, é preciso dispormos de ferramentas mais poderosas e melhor elaboradas, além dos tradicionais editores de texto, compiladores e ligadores. Assim, começaram a surgir aplicativos como editores orientados à sintaxe, geradores de diagramas, gerado-

res de esqueleto de programas e muitos outros. Veio à tona, então, a idéia de se automatizar o próprio ambiente de desenvolvimento de software (ADS). Uma vez que grande parte do esforço de desenvolvimento de software consiste de tarefas mecânicas como, por exemplo, verificar se todos os módulos dependentes de um módulo alterado foram recompilados, justifica-se a utilização do computador para automatizar seu próprio ambiente. Além disso, o surgimento de estações de trabalho de baixo custo, com grande capacidade de memória e de processamento e alta resolução gráfica foi determinante na viabilização de ADS's. A interligação destas estações de trabalho através de uma rede possibilitou o desenvolvimento distribuído de sistemas, além de fornecer facilidades individuais com interfaces gráficas e amigáveis e bom tempo de resposta.

As aplicações não convencionais, dentre elas os ADS's, possuem em comum características bastante diferentes dos atuais sistemas comerciais. Dentre elas citamos:

- objetos de estrutura complexa com volume relativamente pequeno de dados de cada tipo, ao contrário dos sistemas convencionais, onde as informações possuem estrutura simples, mas com volume de cada tipo bastante grande;
- transações longas como, por exemplo, a construção de um programa ou redação de uma ata de reunião;
- as pessoas envolvidas no sistema podem estar distribuídas fisicamente e, ao mesmo tempo, executando um trabalho cooperativo;
- muitas informações possuem histórico de desenvolvimento, como é o caso de desenvolvimento de projetos.

O modelo relacional, embora utilizado em muitos sistemas, mostrou-se inadequado no atendimento a alguns requisitos destas aplicações [Pen86] [SZa87]. A principal dificuldade deste modelo é quanto ao armazenamento de objetos complexos. Como todas as informações tem que ser mapeadas para relações compostas de campos simples, partes do objeto ficam "espalhadas" pelo sistema, dificultando sua recuperação. Em outras palavras, o banco de dados não retrata a semântica do minimundo modelado. Para suprir esta deficiência e, motivado pelo paradigma de orientação a objetos, surgiu a idéia do banco de dados orientado a objetos (BDOO). Podemos defini-lo informalmente como sendo um sistema que suporta um modelo de dados que permite representar uma entidade do minimundo (qualquer que seja sua complexidade ou estrutura) por exatamente um objeto do banco de dados [Dit86]. Porém, não há ainda um consenso quanto ao seu modelo de dados [Ban88] [Mai89], muitos preferindo apenas dar uma definição operacional de um BDOO, ou seja, através das facilidades que ele deve

apresentar. Algumas destas características são: suporte a objetos complexos, identificação única, extensibilidade, herança e armazenamento de programas. Estes mecanismos serão discutidos com mais detalhe no capítulo 2, juntamente com a apresentação do papel que um banco de dados pode desempenhar no sentido de que diversos dos requisitos atualmente identificados para um ambiente de desenvolvimento de software sejam atingidos.

1.2 Apresentação do Trabalho

Alguns protótipos de BDOO foram recentemente desenvolvidos, porém há uma carência de experiências práticas que comprovem sua adequação aos problemas que eles se propõe resolver. Este trabalho relata justamente uma experiência com um destes protótipos, utilizando uma aplicação não convencional, no caso, um ambiente de desenvolvimento de software. O objetivo é, através da modelagem e implementação deste ambiente, verificar como as facilidades oferecidas se ajustam frente as características dos objetos da aplicação.

O ambiente que será detalhado é o A_HAND (Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados) [DLi87], atualmente em desenvolvimento no Departamento de Ciência da Computação da Unicamp. Suas facilidades são orientadas para o suporte de suas linguagens Cm [SLD88] e LegoShell [Dru89], além de dedicar atenção especial à questão da reusabilidade do software. Segundo Meyer em [Mey87], a especificação e desenvolvimento de software reutilizável é um problema técnico. Isto é, além de facilidades para recuperação dos módulos, as próprias linguagens devem facilitar a fatoração de partes comuns em todos os níveis de abstração do objeto. Como o próprio nome indica, a criação de níveis diferenciados de abstração é uma linha mestra na concepção do ambiente A_HAND.

O banco de dados utilizado na implementação foi o SGBD Damokles [Abr88]. Ele foi desenvolvido na Universidade de Karlsruhe e surgiu no âmbito do projeto alemão UNIBASE com o objetivo de dar suporte a ambientes de projeto. Ele possui facilidades para a descrição de objetos complexos que são manipulados por operações genéricas, o que caracteriza um banco de dados orientado a objetos com enfoque estrutural. Seu modelo de dados, o *Design Object Data Model* (DODM) é uma extensão do modelo entidade-relacionamento [Che76] acrescido de facilidades como controle de versões e um tipo de dado para o armazenamento de campos longos.

Baseado no modelo DODM, foi criado um modelo de diagramação para ser utilizado na modelagem do ambiente. Alguns conceitos não suportados pelo

mesmo foram acrescentados ao modelo de diagramação a fim de tornar a experiência mais rica no sentido de testar como pode haver um mapeamento de um modelo mais poderoso para outro e quais as suas conseqüências. A metodologia utilizada para a modelagem do ambiente em muitos pontos assemelha-se à proposta de projeto orientado a objetos de Booch [Boo86].

A apresentação do trabalho está disposta na seguinte seqüência: o capítulo 2 faz uma introdução a banco de dados orientado a objetos e sua aplicação em ADS's; o capítulo 3 apresenta a metodologia de projeto que foi utilizada na modelagem do ambiente A_HAND, cujo detalhamento está descrito no capítulo 4; o capítulo 5 mostra a modelagem do ambiente segundo a metodologia proposta, sendo a implementação do modelo descrita no capítulo 6; o capítulo 7 resume as conclusões obtidas ao longo do trabalho.

Capítulo 2

Banco de Dados Orientado a Objetos

O banco de dados de um ambiente de desenvolvimento de software (ADS) deve incorporar algumas facilidades não oferecidas pelos bancos de dados tradicionais a fim de que os requisitos deste tipo de aplicação sejam atendidos. Isso motivou o surgimento de um novo tipo de banco de dados chamado de banco de dados orientado a objetos (BDOO). Neste capítulo, apresentaremos alguns conceitos relacionados a um BDOO e a sua aplicação em ADS's.

2.1 Introdução

As primeiras propostas de ambientes de desenvolvimento de software procuravam dar suporte principalmente à fase de programação dos sistemas e utilizavam arquivos comuns para o armazenamento das informações. As vantagens de se manter programas fonte e outros documentos como especificação de configurações e documentação na forma de texto deve-se a portabilidade deste tipo de armazenamento, pois todo sistema operacional possui um sistema de arquivos. Além disso, editores de texto poderiam ser a única interface necessária para alterar as informações. Porém, à medida que novas ferramentas são acrescentadas ao ambiente e este passa a abranger uma parte maior do ciclo de vida do software, os relacionamentos e dependências entre os dados ficam mais complexos e, portanto, mais difíceis de serem mantidos e controlados em arquivos. Assim, é mais conveniente utilizar uma forma de armazenamento que permita maior independência dos dados, tornando as aplicações imunes a mudanças na sua estrutura ou na estratégia de acesso a ela. Uma vez que esta característica é um dos principais

objetivos de um banco de dados, é natural pensar em utilizá-lo na construção de um ADS.

No ambiente comercial, os bancos de dados relacionais são os mais populares porque são mais flexíveis e de utilização mais simples que os modelos antecessores [SZa87]. A flexibilidade advém do fato dos relacionamentos não terem que ser pré-definidos, mas dinamicamente construídos através da operação de junção, baseada no valor dos atributos. A facilidade de utilização deve-se à noção intuitiva de tabelas e à existência das linguagens de consulta de quarta geração. Conseqüentemente, foi natural que o primeiro modelo de banco de dados aplicado em ADS's tenha sido o relacional. Na realidade, um banco de dados relacional atende a quase todos os requisitos de uma aplicação, garantindo persistência, compartilhamento e segurança dos dados, e possuindo a capacidade de modelar qualquer tipo de dado. Ele foi utilizado em diversos ADS's, dentre os quais citamos os ambientes OMEGA [Lin84] e ALMA [Lam88]. Porém, devido a complexidade e grande número de relacionamentos entre os objetos em um ADS, o modelo relacional não se mostrou conveniente.

Para dar melhor suporte a estas novas características, algumas extensões ao modelo foram propostas. Por exemplo, o *Project Master Data Base* (PMDB) [Pen86], utiliza um modelo de dados híbrido, formado pelo SGBD Ingres e o sistema de arquivos Unix para armazenamento de campos longos, como textos e programas. A experiência relatada neste projeto concluiu que o mapeamento das informações para o modelo relacional introduziu efeitos colaterais indesejáveis, tal como a redundância de dados, além de um péssimo desempenho devido a falta de localidade dos dados associados a um objeto de estrutura complexa. Desta forma, evidenciou-se a necessidade de um modelo que capte de forma mais direta entidades do mundo externo tanto a nível conceitual, quanto a nível de implementação.

Desta necessidade, em conjunto com o sucesso das linguagens orientadas a objetos, surgiram os bancos de dados orientados a objetos (BDOO). Em [Dit86], Dittrich define um BDOO como sendo um sistema que suporta um modelo de dados que permite representar uma entidade do minimundo (qualquer que seja sua complexidade ou estrutura) por exatamente um objeto do banco de dados.

2.2 Banco de Dados Orientado a Objetos

Diversas propostas de BDOO tem sido feitas [Kim87] [Mai85] [DKL85] com o propósito comum de obter um modelo de dados que capte de forma mais natural a semântica das aplicações. Porém, não há ainda um modelo de dados

orientado a objetos padrão. Nos modelos tradicionais (relacional, hierárquico e rede) podemos definir um modelo de dados como sendo um conjunto de estruturas de dados, operações sobre estas estruturas e restrições sobre seus estados [Mai89]. No entanto, em um modelo orientado a objetos não há uma coleção de tipos fixos, mas a capacidade de definir novos tipos de dados. Então, segundo Maier [Mai89], podemos dizer que o sistema de definição de tipos de um BDOO constitui o seu modelo de dados. Inicialmente, este modelo poderia ser descrito através de seus construtores, mecanismos de encapsulamento e tipos suportados pelo ambiente. O autor argumenta que estes tipos poderiam ser baseados em algumas linguagens de programação orientadas a objetos existentes, mas conclui que não há ainda experiência suficiente para padronizar o sistema de tipos. Já em [Ban88], Bancillhon aponta a falta de uma base teórica, com definições claras da semântica de tipos, programas e identidade, como sendo a causa da falta de consenso quanto ao modelo de dados orientado a objetos. Assim, existem realmente muitas atividades experimentais que "escolhem ao mesmo tempo a especificação do sistema e a tecnologia para implementá-la", ao contrário dos sistemas relacionais, onde "a especificação do sistema era comum e as pessoas estavam desenvolvendo tecnologia para dar suporte a ela" [Ban88].

Portanto, na literatura encontramos apenas uma definição operacional de um BDOO, ou seja, a caracterização através do conjunto de facilidade que ele oferece [ZMa90]. Dentre estas facilidades citamos: encapsulamento de dados e operações, identidade do objeto, suporte a objetos complexos, herança de propriedades, conceito de classe, extensibilidade da hierarquia e sistema de tipos e ligação dinâmica de métodos. Porém, muitas vezes as facilidades oferecidas dependem do tipo de aplicação ao qual se destinam, não havendo um consenso quanto a importância de cada uma delas ou quais são essenciais e caracterizam realmente um BDOO. Dittrich distingue três classes de BDOO [Dit86]:

- **estrutural:** se permite a descrição de objetos complexos que são manipulados por operações genéricas que se aplicam indistintamente aos objetos e seus componentes;
- **operacional:** não dá suporte a objetos complexos mas permite a definição de operações particulares a cada tipo de dado;
- **comportamental:** suporta objetos complexos com operações particulares associadas nos diversos níveis de abstração.

Muitos denominam de banco de dados semântico [HKi87] [ZMa90] o que Dittrich chama de BDOO estrutural, considerando como BDOO somente aqueles que enfatizam a habilidade de encapsular o comportamento, equipando cada

objeto com características autônomas [Yok89]. Na seqüência, descreveremos algumas das facilidades mais comumente citadas para um BDOO.

2.2.1 Suporte a Objetos Complexos

A habilidade de armazenar e manipular objetos complexos é uma facilidade de muitos sistemas orientados a objetos, permitindo inclusive o compartilhamento de subobjetos e objetos recursivos, sem haver a necessidade de mapeá-los para outro tipo de estrutura, como nos bancos de dados tradicionais. Em geral, o seu modelo de dados possui o relacionamento "é componente de", associado a operações que permitem a manipulação do objeto como um todo, bem como a navegação entre seus componentes. Outros oferecem ainda o conceito de generalização, implementado através da herança de propriedades e operações. Isto permite que os objetos possam compartilhar estruturas, além de favorecer a reusabilidade de software através do fatoramento dos problemas em diversos níveis de abstração, o que possibilita o compartilhamento de módulos comuns.

2.2.2 Identidade de Objetos

Identidade é um conceito que permite distinguir um objeto de todos os outros, independente de seu conteúdo, localização ou acessibilidade [KCo86]. Ele é diferente da chave de uma relação em um sistema relacional, pois independe do valor dos dados. Em um BDOO este identificador, comumentemente chamado de *surrogate key*, é fornecido pelo sistema e é imutável e único mesmo depois do objeto ser excluído. Isto permite que algumas situações como compartilhamento de objetos e objetos cíclicos possam ser modelados diretamente sem que haja necessidade de um novo nível de identificação. Além disso, assim como na linguagem Smalltalk [GRo85], é possível dispormos do teste de identidade (X == Y) e igualdade (X = Y), bem como outras operações como cópia de objetos com níveis diferenciados de cópia e compartilhamento. Esta linguagem oferece a operação de cópia rasa e cópia profunda, além da atribuição simples. Por exemplo, suponha que Y seja um objeto composto. A atribuição simples de Y para X faz com que X compartilhe os mesmos subobjetos de Y. Uma cópia rasa faria com que um novo objeto X, com sua própria identidade, fosse criada e que este compartilhasse os mesmos subobjetos de Y. Já a cópia profunda, criaria um novo objeto X, mas com novos subobjetos, criados com os mesmos valores que os subobjetos de Y, porém com identidade própria.

2.2.3 Armazenamento de Programas e Dados

Ao contrário dos sistemas tradicionais, que armazenam os dados em um banco de dados e programas em arquivos, alguns BDOO armazenam os dados e programas dentro de um mesmo sistema, possibilitando que as facilidades de recuperação dos dados possam ser também utilizados para os programas. Esta facilidade auxiliaria, por exemplo, a reusabilidade de software, no que diz respeito a sua recuperação.

2.2.4 *Overriding* e Ligação Dinâmica

Em um sistema orientado a objetos, quando um objeto é subobjeto de outro, ele herda deste suas propriedades e operações. Porém, o subobjeto pode redefini-las, particularizando-as às suas próprias características. A este processo chamamos de *overriding*. A vantagem decorrente disto é que, apesar de termos que escrever o mesmo número de programas, os demais programadores que porventura os utilizarem não terão que se preocupar em tratar os programas para cada tipo de maneira diferente. Ou seja, conceitualmente eles são um só. Assim, o código escrito fica mais simples e reutilizável, pois, se um novo tipo for incluído no sistema, os programas que utilizavam os tipos já existentes poderão ser aplicados para o novo tipo sem serem modificados. Para oferecer esta facilidade sem que haja necessidade de novas compilações, o sistema deve suportar ligação dinâmica, isto é, que os nomes das operações sejam ligados a um tipo específico em tempo de execução.

2.2.5 Extensibilidade

A extensibilidade do modelo de dados de um BDOO consiste no fato de novos tipos de dados ou classes poderem ser criados [SZa87] [Ban88] e utilizados da mesma forma que tipos pré-definidos do sistema, em contraste com o sistema relacional, que possui um único tipo parametrizável, ou seja, a relação. Estes novos tipos são definidos pela sua estrutura interna e as operações possíveis de serem aplicadas sobre ela. Isto constitui mais uma vantagem de se poder manter programas e dados dentro de um mesmo sistema. Portanto, um objeto tem duas partes, a interface e a implementação, sendo que a sua única parte visível são as operações, o que permite o encapsulamento das informações.

Em alguns projetos como o EXODUS [CDe86] e Genesis [Bat86], no entanto, a extensibilidade diz respeito à arquitetura do sistema, construído não como um sistema de banco de dados completo, mas um núcleo de banco de dados com ferramentas para facilitar a geração de um SGBD para uma aplicação específica.

Este enfoque é bastante interessante, uma vez que dificilmente um único banco de dados será capaz de atender simultaneamente aos requisitos de funcionalidade e desempenho de todas as novas aplicações ditas não convencionais.

2.3 Aplicação de BDOO em ADS

Diversos ambientes de desenvolvimento de software tem sido implementados, propondo novos bancos de dados para o armazenamento de suas informações. As características apresentadas na seção anterior são genéricas para qualquer aplicação e caracterizam de uma forma geral o conceito de BDOO. Porém, o enfoque dado a sua utilização em um ADS varia muito, visto o grande número de requisitos adicionais atualmente identificados e a dificuldade de atender a todos ao mesmo tempo. Na Figura 2.1, apresentamos uma lista de requisitos de banco de dados para ADS desenvolvida por Penedo [Row89]. Na seqüência, veremos qual a importância destas facilidades para que algumas das características desejáveis de um ADS sejam atingidas.

2.3.1 Integração

A integração de um ADS deve se dar em três níveis: interna, externa e de contexto. A integração interna diz respeito à consistência entre seus componentes internos. Para que isto ocorra, é necessário que haja coerência na metodologia suportada pelas ferramentas e compartilhamento das informações através da definição de um modelo comum de dados. Porém, esta integração não é facilmente atingida, pois as ferramentas usam muitos tipos de dados diferentes e os armazenam em formas distintas. Conseqüentemente, a transição de uma fase do desenvolvimento para outra torna-se difícil, dando a impressão que o resultado de cada fase é um produto isolado, não integrado a um ciclo de vida completo. Para que isso não ocorra, é interessante que o SGBD conheça a forma como os dados são armazenados e ser capaz de traduzi-los para diferentes representações a fim de que eles possam ser apresentados no formato conveniente para cada ferramenta. Uma idéia seria implementar conjuntos de funções de tradução para tipos atômicos padrões que pudéssemos acoplar a cada aplicativo. Isto, no entanto, não é uma tarefa simples se o BD suportar dados de diferentes tipos, como requer um ADS, tais como documentos, grafos, programas e até mesmo dados do tipo procedimento. Esta visão de comunicação através do banco de dados é generalizada por Balzer [Bal86] que afirma que a próxima geração de sistemas operacionais fará a comunicação entre processos através de um banco de dados,

Modelo Extensível de Dados
Suporte a Meta-esquemas (esquemas armazenados como dados)
Operações armazenadas com objetos e encapsulamento de dados
Relacionamentos explícitos
Suporte a dados derivados (regras)
Consultas recursivas para acesso a dados hierárquicos
Interface para diversas linguagens de programação
Otimização de consultas e indexação
Suporte a objetos complexos
Suporte a conjuntos bastante extensos de dados
Suporte a versões
Seleção automática de estruturas de armazenamento
Facilidades abrangentes para controle de acesso
Carga e descarga de dados em massa
Suporte a transações curtas e longas
Recuperação de *crash*
Facilidades de *undo*
Portabilidade para diversas plataformas
Arquitetura cliente - servidor
Suporte a bases distribuídas
Desempenho aceitável

Figura 2.1: Requisitos de banco de dados para ADS [Row89]

com a vantagem de que os dados seriam tipados e não somente uma seqüência de bytes.

A integração externa do ambiente refere-se a forma como as facilidades são apresentadas ao usuário. A importância de se ter uma interface homogênea e consistente está na redução do tempo de aprendizagem da utilização do ambiente, além de se obter um conjunto visual.

O terceiro nível de integração é quanto ao contexto no qual o ambiente é utilizado. Isto é, o sistema deve ser moldado às características organizacionais do seu usuário e não exigir que haja traduções para que pedidos e ações sejam executados pelo mesmo. Além disso, o BD deve ser capaz de fornecer dados em diversos níveis de abstração, de acordo com a visão dos diferentes usuários do sistema, como analistas, projetistas, programadores e documentadores.

2.3.2 Extensibilidade

A extensibilidade de um ADS é a sua capacidade de responder a mudanças de requisitos ou tecnologia e a portabilidade para novos equipamentos.

A forma como se dará a extensibilidade e a integração de um ambiente depende da sua arquitetura. Uma proposta interessante é a de se criar uma base comum que engloba serviços para acesso aos dados, serviços de interface do usuário e serviços do sistema, para serem utilizados na geração das ferramentas do ambiente [HBe85]. Uma arquitetura deste tipo está ilustrada na Figura 2.2. Assim, os detalhes dependentes de fatores externos à funcionalidade do ambiente ficam encapsulados nesta base. As ferramentas tornam-se, então, independentes e não se comunicam diretamente um com o outro. Todas as facilidades da interface, base de dados, do sistema ou de gerenciamento do processo são fornecidas através da interface dos serviços comuns. Além disso, sobre as ferramentas há um mecanismo de controle do processo de desenvolvimento, responsável pela definição do processo e pelo controle do acesso dos usuários às ferramentas. Este controle baseia-se em uma definição fornecida pelo próprio usuário através de uma linguagem apropriada. Uma arquitetura deste tipo permitiria que todas as ferramentas criadas pudessem ser utilizadas em qualquer ambiente suportado pela base comum.

Para que o banco de dados deste núcleo básico possa oferecer serviços comuns que possibilitem o acompanhamento da evolução natural de um ADS, moldando-o a novas metodologias, ferramentas e necessidades não previstas, é importante que possua facilidades para definição de novos tipos de dados dinamicamente e que armazene seu meta-esquema. O problema concentra-se no fato de manter a coexistência de antigas e novas definições do esquema, ou seja, programas que

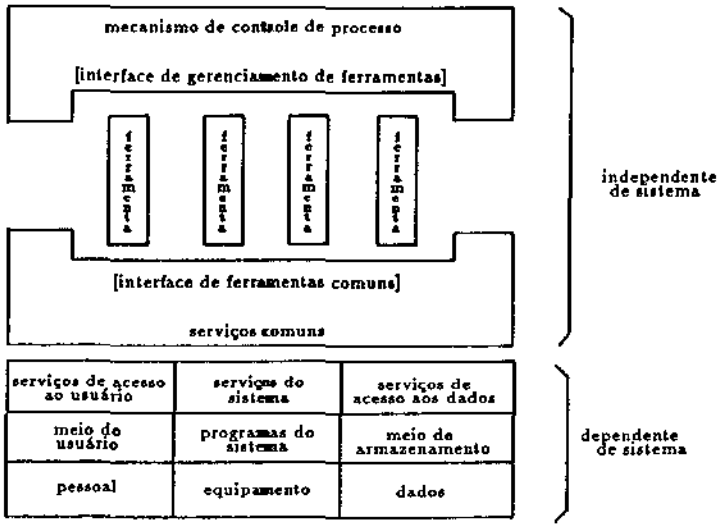

Figura 2.2: Uma proposta de arquitetura de ADS

manipulam instâncias da definição antiga podem ser incompatíveis com instâncias da nova definição e vice-versa. Neste sentido, a existência de um mecanismo de controle de versões seria interessante, pois além de versões de instâncias, teríamos versões de esquemas, como na abordagem do Sistema GERPAC [Mag89].

2.3.3 Controle de Versões e Configurações

O suporte a versões e alternativas de projeto é uma facilidade importante em um ADS porque o desenvolvimento de software é um processo iterativo e muitas vezes de tentativa e erro. Podemos pensar nas diferentes partes do projeto como uma seqüência de versões e o projeto todo como uma configuração das mesmas. Em geral, o desenvolvimento de sistemas não é uma tarefa individual, mas responsabilidade de um grupo trabalhando de forma concorrente. Para que as versões compartilhadas e corretas sejam utilizadas pelos diferentes usuários, há necessidade da noção de versões particulares, efetivas e liberadas. Versões particulares são aquelas em progresso e que só podem ser utilizadas pelo seu proprietário. A versão torna-se efetiva quando colocada para ser testada dentro de um grupo e liberada quando qualquer usuário pode utilizá-la.

As partes de um projeto podem ser compostas ou primitivas, conforme possam ou não ser decompostos em outras partes. Tendo em vista que cada parte possui

um histórico de versõs, à medida que a complexidade do sistema aumenta, mais difícil se torna combinar versões consistentes de seus módulos. Para auxiliar esta tarefa é que um ADS deve também dar suporte ao controle de configurações, que é responsável pela ligação de uma versão de uma parte composta a versões específicas de seus componentes.

Portanto, o banco de dados de um ADS deve ser capaz de armazenar diferentes versões de um mesmo objeto e manter sua seqüência, seja ela linear, em forma de árvore ou de um grafo acíclico. Além disso, ele deve manter também as definições de configurações, que constituem a representação da forma como as versões devem ser combinadas, bem como as configurações já resolvidas.

2.3.4 Distribuição

Uma questão bastante complexa neste tipo de aplicação, tipicamente distribuído e de trabalho cooperativo, é a segurança dos dados quanto a falhas, tanto do hardware quanto do software. Os esquemas de *backup* e *logging* tradicionais tem que ser reanalisados, uma vez que, ao contrário dos sistemas comerciais, as transações deste tipo de aplicação são longas, podendo estender-se durante muitos dias. Portanto, não é mais razoável que a recuperação retorne ao estado anterior ao das transações correntes. Em [Ber89], Bernstein cita o mecanismo de *savepoints* e de divisão de uma transação longa em uma série de subtransações como as linhas de investigação desta questão. Esta técnica é chamada de *nested transaction* e permite que uma subtransação seja terminada com sucesso antes que a transação-pai termine. Outras propostas citadas por Rowe em [Row89] são:

- *naming domains*: uma transação trabalha sobre uma configuração de objetos que, no caso de serem atualizados, são criadas novas versões no término da transação;
- transações participativas: a idéia é que diversos processos podem participar de uma mesma transação, onde cada processo vê as atualizações efetuadas pelos demais participantes, porém elas não são vistas pelos processos não participantes;
- *commit-serializability transactions*: uma transação é dividida em transações distintas se elas não alterarem objetos já atualizados e se os objetos lidos pelas novas transações forem distintos. Todas as transações que terminam com sucesso são serializáveis e a idéia é que as transações sejam criadas, divididas, juntadas e finalizadas à medida que o usuário examina e atualiza o banco de dados;

- *sagas*: é uma transação longa que pode ser dividida em subtransações, cuja execução pode ser feita ao mesmo tempo que outras transações. Todas as subtransações precisam terminar para que o *saga* finalize com sucesso. Além disso, as subtransações não são atômicas, pois suas atualizações podem ser desfeitas através de um *compensating transaction.*

2.3.5 Inteligência Artificial

Contribuições da área de inteligência artificial para ADS's recaem principalmente na representação do conhecimento. Um exemplo disso é a forma como são armazenadas as informações quanto ao controle do processo na arquitetura apresentada na Figura 2.2, tal que elas possam ser interpretadas automaticamente. Neste caso, o conhecimento é voluntariamente dado ao sistema. Um fator complicante seria capturar informações sem que elas sejam explicitamente apresentadas. Este aspecto é investigado pelo sistema Poise [Cro85] que, além de um especificador de tarefas, possui um reconhecedor de exceções de planos do usuário associado a um manipulador de exceções, responsável pela atualização da descrição das tarefas de acordo com as ações executadas.

A capacidade de um SGBD de armazenar conhecimento facilita também a manutenção da integridade dos dados através de restrições. Para que este objetivo seja efetivamente alcançado, além da verificação da cardinalidade dos relacionamentos, estuda-se hoje a incorporação de restrições mais poderosas, associadas a mecanismos ativos, que correspondem a ações ativadas quando determinadas operações são efetuadas sobre os objetos. Eles podem também funcionar como mecanismo de alerta, como, por exemplo, notificar um usuário sobre a atualização de um objeto que ele está editando.

2.4 Implementação de um BDOO

Existem, atualmente, duas linhas de implementação de um banco de dados orientado a objetos: construí-lo estendendo o modelo relacional ou construir um novo banco de dados, que explore as particularidades de um sistema orientado a objetos. Como exemplo do primeiro enfoque, temos o sistema POSTGRES [Sto87], que estendeu o modelo relacional com tipos abstratos de dados, incluindo operadores e procedimentos definidos pelo usuário, atributos do tipo procedimento e herança de atributos e procedimentos. Apesar desta linha ter contribuído no amadurecimento das questões relacionadas ao suporte de uma diversidade de tipos de dados, todos eles envolvem a adição de facilidades sobre um modelo existente, o que pode tornar sua utilização difícil ou ineficiente [Tha86]. Em contrapartida,

a construção de um novo banco de dados pode utilizar técnicas que implementem de forma mais apropriada algumas facilidades, como a manipulação de conjuntos de objetos complexos, ou possibilitando a modularidade, isto é, fazendo com que componentes do sistema possam ser retirados ou colocados dependendo das necessidades da aplicação.

Alguns bancos de dados tem sido implementados baseados no modelo de dados de uma linguagem de programação orientada a objetos particular. A integração destas duas tecnologias criaria o que chamamos de *Database Programming Language*. Isto é, ela possuiria a completude computacional de uma linguagem de programação, associada à persistência dos dados oferecida pelo banco de dados. O modelo de dados poderia ser, então, aquele suportado pela linguagem. Porém, não há atualmente uma linguagem de programação que possua um modelo de dados que possa ser considerado padrão para orientação a objetos. Portanto, o banco de dados poderia ficar limitado às características da linguagem e não possuir um modelo genérico, como idealmente deveria ter. As diferenças de cada tecnologia que dificultam sua integração é denominada de *impedance mismatch*.

Quanto à arquitetura do banco de dados, pode-se distinguir três classes [KTu87]: sistemas para aplicações específicas, sistemas genéricos com interface para um grande número de aplicações e arquiteturas de núcleo de banco de dados, ou seja, SGBD's extensíveis. A construção de um sistema genérico é dificultado pelos diferentes requisitos das diversas áreas de aplicação. Além disso, as facilidades de banco de dados necessárias estão intimamente relacionadas à funcionalidade desejada, como pode ser percebido na apresentação dos requisitos de ambientes de desenvolvimento de software apresentado da seção anterior. Em contrapartida, a construção de um núcleo possibilita o desenvolvimento posterior de *front-ends* específicos para cada aplicação. A questão crucial é, então, como deve ser este núcleo. Ele certamente deve possuir apenas as características essenciais. Porém, a dificuldade está na decisão do quão complexo pode ser este núcleo a fim de mantê-lo pequeno o suficiente para possibilitar futuras extensões.

Acreditamos que a experiência de utilização de um BDOO já existente no suporte a uma aplicação não convencional pode ser uma base importante na avaliação da importância das facilidades citadas neste capítulo.

Capítulo 3

Uma Metodologia de Modelagem

A identificação de todos os componentes e relacionamentos, e a escolha de mecanismos para acessar, modificar e controlar os dados são questões críticas no projeto de um ambiente integrado de engenharia de software. Neste capítulo, apresentaremos um modelo de diagramação inspirado no DODM (Design Object Data Model) do SGBD Damokles e que incorpora alguns conceitos provenientes dos modelos semânticos, além de uma metodologia de projeto semelhante ao projeto orientado a objetos proposto por Booch [Boo86].

3.1 Trabalhos Correlatos

As experiências na modelagem de ambientes de desenvolvimento de software mostram diferentes abordagens ao assunto.

Uma delas é dos ambientes que objetivam apenas a fase de programação, que tendem a modelar as construções da linguagem a que se destinam suportar em objetos a serem armazenados separadamente. Dentre estes, podemos citar o PMC (Program Module Catalog) [Mar85], que integra um compilador Modula-2 a um banco de dados relacional a fim de assegurar a consistência entre diferentes módulos de um sistema e entre os módulos de definição e implementação, que são compilados em separado na linguagem. Outro exemplo é o ambiente de programação OMEGA [Lin84], construído sobre o SGBD Ingres, um banco de dados relacional. Seu enfoque principal é obter múltiplas visões do programa a fim de facilitar a depuração do mesmo. Segundo [Lin84], o problema desta abordagem é o tempo de recuperação dos objetos, visto o grande número de

pequenas consultas necessárias para a obtenção de suas partes, distribuídas em diferentes relações. Porém, quanto ao espaço de armazenamento, observou-se que, sem contar índices e comentários, o espaço ocupado no banco de dados não é tão maior que na forma de texto [Lin84].

Um dos trabalhos pioneiros na modelagem de todo o ciclo de vida do software, é o PMDB (Project Master Data Base) [Pen86]. O modelo E-R [Che76] foi utilizado no levantamento exaustivo de todos os dados envolvidos no processo de desenvolvimento, resultando em um modelo que consiste de 31 objetos, 220 atributos e aproximadamente 170 relacionamentos.

Ainda utilizando o mesmo modelo básico (E-R), uma abordagem diferente é feita pelo ADS ALMA [Lam88]. Ao invés de tentar identificar todos os dados envolvidos, a proposta é utilizá-lo como um meta-modelo instanciável para um ambiente particular. Assim, na camada mais baixa do sistema, os únicos objetos são: "meta-entidade", "meta-relacionamento" e "meta-atributo". Através de algumas ferramentas, as entidades, relacionamentos e atributos do ambiente real são definidos, criando o "modelo instanciado". Concluída esta etapa, o usuário passa a ver somente os objetos que ele mesmo definiu. A vantagem deste tipo de abordagem é a flexibilidade obtida pelo modelo, uma vez que não é limitado a nenhuma metodologia ou padrão de desenvolvimento.

Segundo [Pen86], "o modelo E-R é apropriado, possivelmente o melhor modelo de dados disponível, porém não o suficiente para representar todos os dados envolvidos em um projeto de software, especialmente quando se trata de objetos estruturados". Assim, como exemplo de novas propostas de modelo para a área de ADS, temos o Cactis [HKi88], que incorpora construtores de tipos derivados dos modelos semânticos, cujas características serão discutidas em uma seção subsequente. Além disso, o SGBD que o implementa dá suporte a atributos derivados, que são atributos obtidos em função de outros atributos locais ou importados através de relacionamentos. Dessa forma, os objetos adquirem meios de responder automaticamente a alterações ocorridas em outro ponto no BD.

Uma nova preocupação, motivada pelo fato de que o projeto de software não é um processo linear, mas que envolve repetições de atividades devido a ocorrência de erros, alteração do projeto, ou com o intuito de aprimorar o sistema, surgiram trabalhos de modelagem do processo de desenvolvimento. Isso possibilita que o ambiente mantenha o histórico do projeto, além de monitorar atividades concorrentes e iniciar atividades em decorrência do resultado de outras. O modelo DesignNet [LHo89] utiliza operadores "e" e "ou" para descrever a estrutura dos dados e notação Petri-net para representar dependências e paralelismo entre atividades, recursos e produtos. Estes objetos são tipos básicos do modelo, juntamente com alguns relacionamentos pré-definidos, tornando-o interessante,

uma vez que o modelo básico já incorpora algumas características próprias de um ADS.

3.2 O Modelo DODM - Damokles

Uma vez que as propostas de novos modelos de dados para ADS objetivam representar mais diretamente o mundo real, é natural que os mesmos sejam utilizados no projeto conceitual dos dados.

A proposta de modelagem baseia-se no modelo *Design Object Data Model* (DODM) do SGBD Damokles. Damokles [Abr88] é um banco de dados desenvolvido na Universidade de Karlsruhe e que surgiu no âmbito do projeto alemão UNIBASE. Seu objetivo é dar suporte a ambientes de projeto e pode ser classificado como um banco de dados orientado a objetos com enfoque estrutural, segundo Dittrich em [Dit86]. Isto é, ele permite a descrição de objetos complexos que são manipulados por operações genéricas que se aplicam indistintamente aos objetos e seus componentes. Seu modelo de dados pode ser caracterizado como uma extensão do modelo E-R, já que dá suporte a:

- objetos estruturados que podem ou não ter versões;
- relacionamentos entre objetos e/ou suas versões;
- atributos associados a objetos, versões e relacionamentos.

Os objetos DODM podem ser simples, quando compostos apenas por atributos, e estruturados, quando possuem também a parte estrutural, correspondente a um conjunto de subobjetos. Estes, por sua vez, podem ser objetos simples ou estruturados. O relacionamento entre o objeto e seus subobjetos é o de composição ("is-part-of"). Como este relacionamento se aplica recursivamente, um objeto pode, direta ou indiretamente, fazer parte de sua própria estrutura. Além disso, o conjunto de subobjetos de um objeto e de outro não precisam, necessariamente, ser disjuntos. Eles podem se sobrepor, formando assim, não uma simples hierarquia, mas um grafo da estrutura de objetos.

Assim como existe o construtor de tipos "union" na linguagem C, podemos utilizar o mesmo conceito para a definição de objetos no DODM. Por exemplo, é possível expressar que uma seção de jornal é composto por partes de seção e que uma parte de seção é um artigo ou um classificado.

O conceito de versões do DODM é bastante peculiar e, em alguns pontos, confunde-se com o conceito de objeto. Tipicamente, as versões possuem as mesmas propriedades que o objeto do qual foi originado. Porém, no DODM, exceto pelo fato de uma versão pertencer a um objeto, que chamamos de objeto

genérico, ela possui sua própria definição. Assim, os atributos e estrutura das versões podem ser totalmente arbitrários em relação ao seu objeto genérico. Da mesma forma, os relacionamentos envolvendo o objeto genérico não estão automaticamente definidos em relação às suas versões. Se estas também puderem participar, isto deve ser declarado explicitamente. As versões podem até mesmo ter suas próprias versões! Na realidade, o objeto genérico representa propriedades comuns a todas as versões, sendo a versão a definição do objeto real propriamente dito. O histórico de versões pode ser de 3 tipos: linear, onde cada versão tem um único predecessor e um único sucessor; tipo árvore, no qual há um único predecessor e um número arbitrário de sucessores; e acíclico, que possui um número arbitrário de predecessores e sucessores.

Os relacionamentos são n-ários e bidirecionais, envolvendo objetos e/ou versões. Cada objeto/versão que participa do relacionamento é caracterizado por um papel, sendo possível associar restrições de cardinalidade de "no mínimo 1" e "no máximo 1" a cada papel. Além disso, os relacionamentos podem ter atributos e podem ser subobjetos de um objeto estruturado.

Quanto aos atributos, existem os tipos pré-definidos usuais como inteiro, booleano, caracter e "string", assim como construtores de tipos como "subrange", enumerado, "array", registro e "union". Existe, ainda, um tipo pré-definido para o armazenamento de dados não estruturados de comprimento longo, chamado "LONG_FIELD".

Foi procurado incorporar os conceitos DODM em um modelo de diagramação. Uma vez que o objetivo final são os experimentos com um BDOO e, ao mesmo tempo, para tornar o modelo mais expressivo, as abstrações comumente propostas pelos modelos semânticos foram acrescentadas a este modelo de diagramação.

3.3 Modelos Semânticos

Qual a relação entre modelos semânticos e modelos orientados a objetos? Modelos semânticos são modelos que suportam relacionamentos mais expressivos, permitindo a especificação de objetos complexos. Esta é uma definição bastante comum [HKi87] [HuK87] [Miy86], porém bastante vaga. Isto é, dado um determinado modelo, como podemos classificá-lo como semântico ou não? Na realidade, podemos dizer que a única característica unificadora destes modelos é que eles procuram expressar mais conteúdo semântico que o modelo relacional. Quanto ao modelo orientado a objetos, não há um consenso quanto a sua definição. A maioria delas enfatiza a habilidade de encapsular aspectos comportamentais, seguindo a linha das linguagens orientadas a objetos e, facilidades

a nível de execução de programas, como ligação dinâmica (*dynamic binding*) [Ban88].

Portanto, podemos dizer que um modelo orientado a objetos "completo" permite o encapsulamento da estrutura dos objetos complexos, definidos através das abstrações oriundas dos modelos semânticos, ao mesmo tempo que encapsula seu comportamento.

Os mecanismos de abstração dos modelos semânticos são responsáveis pela sua clareza. Eles equivalem a introdução de novos tipos de relacionamentos, ao invés de utilizar apenas uma construção para a representação de todos eles. As abstrações mais comumente incorporadas pelo modelos semânticos são: classificação, generalização e agregação.

Classificação é a forma de abstração na qual uma coleção de objetos é identificada como pertencente a uma classe, cujas propriedades são comuns a todos eles. Essencialmente, ela representa o relacionamento "is-intance-of". Por exemplo, uma classe chamada ALUNO pode ter 2 instâncias: Maria e João.

Generalização é o meio pelo qual diferenças entre objetos similares são ignoradas para criar uma classe de nível mais alto, na qual as similaridades são enfatizadas. Exemplificando, podemos dizer que ALUNO é uma generalização das classes ALUNO-DE- GRADUAÇÃO e ALUNO-DE-PÓS-GRADUAÇÃO ou, inversamente, ALUNO-DE- GRADUAÇÃO e ALUNO-DE-PÓS-GRADUAÇÃO são especializações de ALUNO. Esta abstração é comumente ligada ao relacionamento "is-a", porém este não é o único significado dado a este relacionamento na literatura [Bra83].

Agregação é o mecanismo através do qual relacionamentos entre classes de nível mais baixo são abstraídos constituindo uma classe de nível mais alto. Por exemplo, suponhamos que existam as classes ALUNO e CURSO. Para a criação da classe MATRICULA é necessário o envolvimento das duas classes citadas. Da mesma forma, podemos dizer que a classe ALUNO é uma agregação de NOME e ENDEREÇO.

Análises comparativas entre diversos modelos semânticos são apresentadas por Peckam [PMa88], Miyasato [Miy86] e Hull [HuK87].

3.4 Um Modelo de Diagramação

A experiência tem mostrado que a notação gráfica é uma ferramenta que facilita a representação dos conceitos do modelo no projeto conceitual. Nossa proposta [HMa89] é um modelo de diagramação que incorpora abstrações originárias dos modelos semânticos e conceitos propostos pelo modelo de dados do

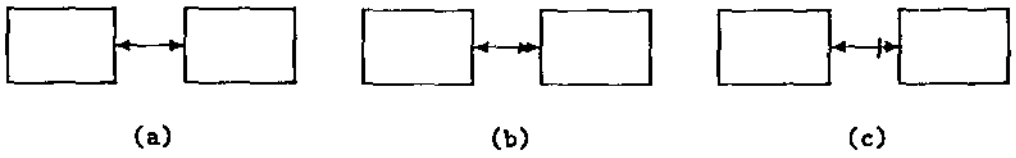

Figura 3.1: (a) notação para relacionamentos 1:1; (b) notação para relacionamentos 1:n; (c) notação para relacionamentos parciais.

SGBD Damokles (DODM).

Sendo o DODM uma extensão do modelo E-R [Che76] e, também, devido a sua popularidade, adotou-se a sua forma de diagramação como base para a construção do modelo. A notação gráfica de diversas extensões já realizadas foram incorporadas a ele, sendo outras criadas ou aproveitadas de propostas para outras áreas de conhecimento, como a inteligência artificial. Assim, uma entidade é representada por um retângulo e os relacionamentos, por questão de simplicidade, são simplesmente arcos ligando as entidades envolvidas e identificados pelos seus nomes. A representação de relacionamentos 1:1 e 1:n estão ilustrados na Figura 3.1, juntamente com a notação para relacionamentos parciais.

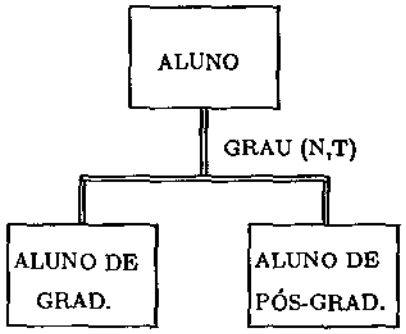

Figura 3.2: Notação para generalização/especialização

Relacionamentos do tipo n:m precisam, em alguns casos, assumir a semântica de agregação, isto é, uma combinação dos dados das duas entidades envolvidas, possivelmente para relacionar-se com uma terceira. Para não criar uma nova notação para este tipo de situação, como proposto em [Set83], ela é representada

diretamente como uma entidade, podendo, portanto, possuir atributos e manter relacionamentos 1:n com as demais entidades.

Como um dos objetivos da proposta é enfatizar a parte estrutural dos dados, foram incorporadas notações diferenciadas para os conceitos de generalização e composição. Para o primeiro, a notação adotada foi proposta por Wagner [Wag88], que utiliza linhas duplas para sua representação, como ilustrado na Figura 3.2. Estas linhas geralmente são rotuladas com o atributo que define os subtipos e a natureza do particionamento, isto é, se são subconjuntos mutuamento exclusivos (N - *non-overlapping*) ou não (O - *overlapping*) e se modelam a totalidade do mundo real (T - *total*) ou apenas parte dele (P - *partial*).

Já para o relacionamento de composição, a notação foi inspirada nos grafos AND/OR, apresentados por Nilsson [Nil80] e utilizados na representação de decomposições de problemas na área de inteligência artificial. Ele é bastante conveniente, uma vez que permite também representar as alternativas de classe existentes no modelo DODM (uma abstração similar ao construtor de tipos UNION da linguagem C). São utilizadas linhas mais espessas que as dos relacionamentos para que a estrutura dos objetos seja destacada. A Figura 3.3a ilustra a notação de composição, enquanto a Figura 3.3b apresenta a de alternativa de classe. Note que a única diferença está na presença de uma marca circular na interseção da linha ligada ao objeto de mais alto nível com a barra horizontal na notação para alternativas. Outra notação associada a esta é a de cardinalidade, idêntica a de outros relacionamentos. Assim, na Figura 3.3c, temos que o objeto A é composto por "1 a n" objetos do tipo B e, opcionalmente, um objeto do tipo C ou um objeto do tipo D.

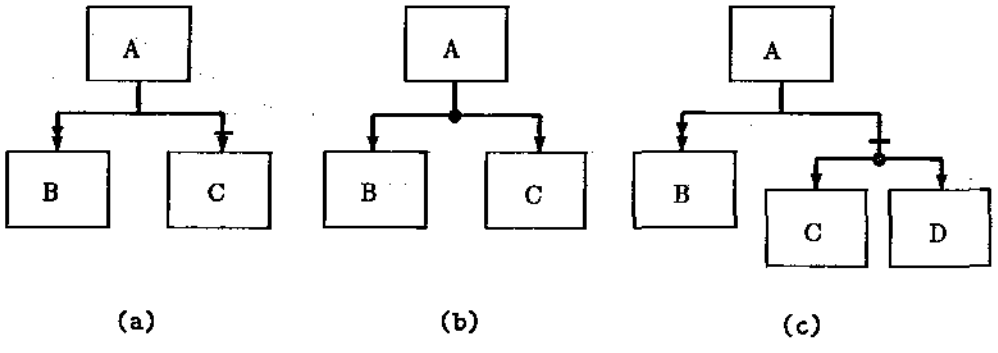

Figura 3.3: Notação para composição e alternativas de classe

O modelo também suporta o conceito de classificação, uma vez que as entidades podem ser consideradas como classes e, portanto, intanciáveis. Embora o conceito de versões se aplique às instâncias e não às classes propriamente ditas, é interessante representar o fato destas poderem possuir versões. A notação adotada, como ilustrado na Figura 3.4, é de uma dupla caixa com uma letra no canto superior direito, indicando o tipo da versão, conforme o modelo de dados Damokles: L (linear), T (tipo árvore) ou A (acíclico).

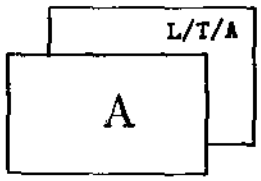

Figura 3.4: Notação para classes que possuem versões

A incorporação de uma notação para versões é uma tentativa de acrescentar ao modelo características comuns a diversos banco de dados, ditos não convencionais. Quanto às abstrações representadas, a intenção não é de se criar um novo modelo, já que diversas propostas existem na literatura, mas a de se obter uma forma de diagramação de fácil elaboração e entendimento imediato.

3.5 Metodologia de Projeto

A modelagem conceitual de um sistema consiste na especificação das propriedades estruturais das entidades, seus relacionamentos, e a associação de ações às mesmas, a fim de preservar todas as suas propriedades, em particular, as restrições de integridade [Ala89]. Podemos dividir este processo em 2 etapas: Modelagem Estrutural, responsável pela identificação das entidades, e Modelagem Operacional, que corresponde à especificação de ações associadas a cada entidade.

A metodologia de projeto que apresentaremos objetiva aproximar o processo de modelagem às características básicas usualmente encontradas nas bases de dados orientadas a objetos. Uma das técnicas utilizadas é o refinamento, no qual, inicialmente, é construído um projeto global, que, em fases subseqüentes, é decomposto e detalhado. O método utilizado se assemelha à metodologia de projeto orientado a objetos proposto por Booch [Boo86] e, basicamente, consiste

de 4 etapas:

1. construção do modelo global do sistema
2. identificação das propriedades associadas aos objetos identificados no modelo global
3. identificação das operações sobre estes objetos
4. implementação das operações

Assim como em outras metodologias, a construção do projeto global é a fase em que o projetista é introduzido ao problema, sendo o modelo utilizado a ferramenta através da qual a forma como ele vê o mundo a ser modelado é expresso. Neste momento, o foco é a identificação dos objetos existentes na aplicação real e a forma como se inter-relacionam. É para esta fase que utilizaremos o modelo de diagramação apresentado na seção anterior. Na realidade, todo o processo de modelagem ocorre paralelamente ao detalhamento informal das funções do ambiente, uma vez que é através dele que adquirimos uma visão mais rica de detalhes sobre o mesmo.

O passo seguinte corresponde ao detalhamento de cada objeto representado no projeto global. A primeira dificuldade com a qual deparamos foi a identificação dos níveis nos quais cada objeto se subdivide, isto é, cada entidade representada no modelo global, pode, na realidade, representar mais de um conceito. Por exemplo, se um módulo de programa é um objeto do modelo global e possui versões, o objeto genérico e as versões podem ser descritos por atributos diferentes. Digamos que este módulo seja utilizado dentro de dois sistemas distintos. Em cada sistema, ele pode ter uma prioridade de execução diferente ou parâmetros distintos. Estes dados não dizem respeito nem ao objeto genérico, nem à versão, mas com a execução em si. Poderíamos dizer que o módulo foi "instanciado" pelos dois sistemas, existindo, portanto, dados de um terceiro nível do mesmo objeto módulo. Isto se assemelha ao conceito de variáveis de classe e variáveis de instância existentes em linguagens de programação orientadas a objetos, como Smalltalk [GR085]. Assim, os três níveis de objetos com as quais trabalharemos daqui por diante são: objeto genérico, que chamaremos de classe, versões e instâncias.

Certamente, nem todos os objetos possuirão todos os níveis. O nível de classe estará sempre presente, uma vez que representa o conceito genérico identificado no modelo global. Ele será detalhado em termos de seus componentes, relacionamentos e atributos, podendo ser expresso da forma ilustrada na Figura 3.5. Os objetos que no projeto global foram representados como tendo versões serão também detalhados no segundo nível. A semântica associada a uma versão pode

variar de objeto a objeto. Por exemplo, ele pode representar diversas implementações de um mesmo módulo de definição, como pode ser visto como cópias de um arquivo feitas a cada final de mês, ou até mesmo diferentes unidades de um tipo de periférico. As versões são decritas em termos de componentes, relacionamentos e atributos, que podem ser totalmente diferentes do nível de classe, cujas propriedades seriam, então, compartilhadas por todas as versões. Utilizamos o terceiro nível quando o conceito de variável de instância é necessário na definição dos atributos. Se o objeto possuir os dois níveis anteriores, tanto o objeto genérico como as versões podem ser instanciados, e isso deve ser definido no modelo.

Utilizando os mesmos níveis identificados para cada objeto, passamos para a terceira etapa, na qual associamos operações aos objetos. Segundo Booch [Boo86], as operações podem ser de 3 tipos:

- Construtor: quando a operação altera o estado objeto;
- Seletor: se a operação avalia o estado do objeto;
- Iterador: se a operação permite que todas as partes do objeto sejam visitadas.

De acordo com as operações acopladas a um objeto, este pode ser classificado como um ator, agente ou servidor. Um objeto ator é aquele que não sofre nenhuma ação, mas opera sobre outros objetos. Em contrapartida, o servidor é aquele que apenas sofre ações, sendo o agente, o objeto que executa ações determinados por outro objeto e que possui capacidade de atuar sobre outros.

A metodologia de projeto que propomos trabalha apenas com objetos passivos (agentes e servidores), ou seja, nos concentramos apenas nas operações que cada objeto pode sofrer. Este tipo de enfoque se assemelha mais aos conceitos de tipos abstratos de dados que a um projeto orientado a objetos, porque, segundo Booch [Boo86], este, além dos dois tipos de objetos citados, preocupa-se com ações que não precisam ser estimuladas por outros objetos (objetos atores) e ações que um objeto requer de outros objetos. Porém, fazendo um paralelo com linguagens de programação orientadas a objetos, até o momento, o tipo de objeto passivo é o mais característico, sendo suportado por linguagens como Smalltalk e C++. Além disso, nem todos os banco de dados possuem a facilidade de determinar e executar ações automaticamente, que chamamos de mecanismos ativos, e que implementariam de forma natural objetos atores. No caso de banco de dados orientado a objetos com enfoque estrutural, considerar apenas objetos passivos não é um problema, já que nestes o encapsulamento de comportamento é feito

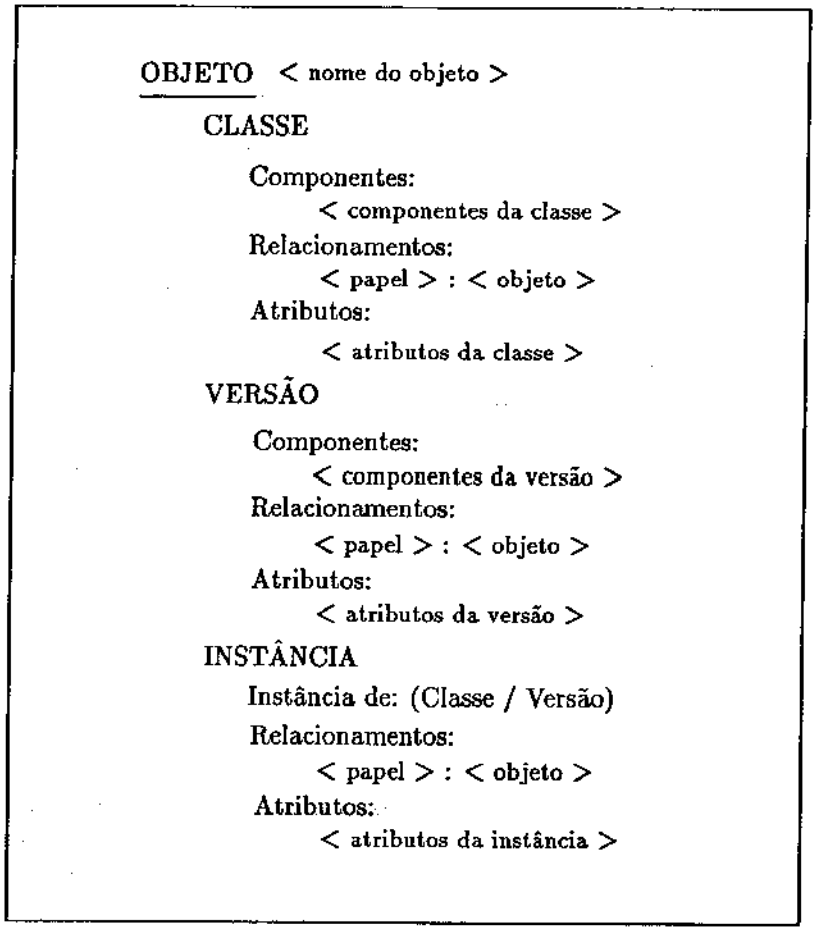

Figura 3.5: Modelo para detalhamento de objetos

criando-se uma camada sobre o modelo estrutural, que associa operações aos objetos dentro de um domínio de aplicação particular [Mai89].

Quanto ao nível de abstração das operações, podemos classificá-las em [Ala89]: transações, ações e alterações de BD. Alterações de BD são ações primitivas sobre o BD, como inserir, excluir e atualizar objetos simples. Ações são particulares a um determinado objeto e são definidos de forma a manter todas as suas restrições de integridade. Elas só manipulam objetos diretamente relacionados com o objeto da ação, podendo as ações destes serem utilizados na decomposição da ação desejada. As atualizações no BD são feitas através de ações de alteração do BD. As transações, por sua vez, são ações de alto nível correspondentes a uma operação no universo da aplicação. São compostas por ações e outras transações e não necessariamente referem-se a um objeto particular, mas podem envolver uma coleção deles.

Na etapa 3 do projeto, inicialmente associamos ações a cada objeto identificado na modelagem estrutural para, posteriormente, utilizá-las na composição das funções de alto nível da aplicação, ou seja, as transações. Dessa forma, além de validar a existência das ações, associamos a elas o contexto (restrições e seqüência) no qual devem ser executadas.

A última etapa, correspondente a implementação das operações, é feita através de um processo de composição, de forma similar a definição das transações, isto é, utilizando operações já existentes de outros objetos. Na realidade, a implementação das operações utilizadas pode ser feita posteriormente, interessando apenas a definição das mesmas. Assim, podemos criar apenas módulos "stubs" no lugar destas, para testar a funcionalidade da operação desejada, no caso da construção de protótipos.

Este tipo de enfoque de projeto, além de ser um processo mais natural em se tratando de banco de dados orientado a objetos, traz como vantagens a facilidade de manutenção, devido a localidade das operações de cada objeto, e a clareza, uma vez que os objetos tem correspondência direta com a aplicação modelada.

No capítulo seguinte apresentaremos o ambiente de desenvolvimento de software A_HAND, cuja modelagem será um exemplo de utilização da metodologia proposta.

Capítulo 4

O Ambiente A_HAND

Neste capítulo, descreveremos o Ambiente de Desenvolvimento de Software (ADS) A_HAND. Ele não será descrito em toda sua extensão e detalhes, mas de forma a tornar clara a modelagem dos seus dados, que será tratada no capítulo 5.

4.1 Introdução

O ADS A_HAND (Ambiente de Desenvolvimento de Software baseado em Hierarquias de Abstração em Níveis Diferenciados) faz parte de um projeto mais abrangente, chamado AIDSH (Ambiente Integrado de Desenvolvimento de Software e Hardware), atualmente em desenvolvimento no Departamento de Ciência da Computação da UNICAMP. Uma de suas características principais é que os objetos manipulados no ambiente podem ser compostos de objetos mais simples. Assim, cada objeto representa, na realidade, uma hierarquia de objetos, cada um provendo um nível de abstração. Daí o nome A_HAND.

Como qualquer outro ADS, seu objetivo final é dar suporte à atividade de produção de software, livrando o usuário de processos mecânicos e repetitivos do ciclo de desenvolvimento, além de facilitar o gerenciamento da produção de sistemas grandes e complexos. Idealmente, o ambiente deve também dar suporte ao desenvolvimento distribuído. Pretende-se, dessa forma, aumentar a produtividade e eficiência desta atividade e, ao mesmo tempo, diminuir seu custo. O que diferencia as diversas propostas de ADS é o direcionamento dos esforços no sentido de atingir estes objetivos. Uma das linhas-mestre do A_HAND é a manutenção da homogeneidade da interface nos diversos níveis de abstração. Dessa forma, o usuário não sente dificuldade para adaptar-se às mudanças de nível,

bem como seu tempo de aprendizado sobre todo o ambiente é reduzido. Esta interface deve, preferencialmente, ser gráfica e amigável.

Outro ponto importante é quanto a reutilização de software. De acordo com Meyer [Mey87], a viabilização desta técnica depende, principalmente, da existência de facilidades para o desenvolvimento de módulos reutilizáveis, apesar desta abordagem envolver fatores administrativos tais como o gerenciamento e recuperação de uma biblioteca de módulos. Atualmente, os programas são desenvolvidos para solucionar problemas específicos, sem a preocupação de fatorar suas semelhanças. Assim, além de continuamente estarmos "reinventando" soluções, seu custo é elevado por não aproveitarmos partes já utilizadas e testadas dentro de outros sistemas. Neste sentido, houve uma evolução nas linguagens de programação, culminando com as linguagens orientadas a objetos, que apresentam mecanismos que facilitam a construção de código reutilizável, tais como:

tipos abstratos de dados: criam uma interface de operações bem definida para cada módulo e encapsulam a sua implementação interna, ajudando o implementador a desenvolvê-los e aos demais a utilizá-los;

generalidade: é a capacidade do módulo ser definido com parâmetros genéricos que representam tipos, permitindo ao implementador escrever um único módulo para um tipo abstrato de dados, aplicáveis a vários tipos de objetos (por exemplo, um mesmo módulo pilha para pilha de inteiros, reais, strings, etc.);

herança: permite a estruturação do sistema em classes de objetos com diferentes níveis de abstração, através da herança de propriedades e operações de um objeto de nível mais alto para aqueles que foram especializados a partir dele.

Em suma, software reutilizáveis devem ter características como "chips" eletrônicos: funcionalidade encapsulada, interfaces bem definidas e alto grau de qualidade. Reunindo as facilidades citadas, o ADS A_HAND utiliza a linguagem Cm (C modular e polimórfico) [SLD88], uma derivação da linguagem C [KRi78], para seu nível de programação. O ambiente possui, ainda, duas outras linguagens: Lego Shell e CO^2 Shell. Se para a linguagem Cm, os produtos são programas ou classes Cm, para a LegoShell, os produtos são *computações*. A LegoShell é uma linguagem gráfica de configuração de programas. A semelhança da criação de *computações* com jogos de montar para crianças como o Lego, deu origem ao nome da linguagem. No caso, os objetos a serem encaixados são programas Cm, arquivos, periféricos e até mesmo computações. Sua interface padrão são portas de entrada e saída, ligadas por meio de conectores, que permitem a especificação

do fluxo de dados entre os mesmos. Uma descrição detalhada dos componentes, bem como a especificação funcional da linguagem será apresentada nas seções seguintes.

O último nível das linguagens, a CO^2Shell, corresponde a uma linguagem de comandos orientada a objetos, cuja especificação ainda não está terminada. Ela seguirá a mesma linha das linguagens de comando do Unix e servirá a dois propósitos: interface entre o usuário e o sistema operacional e linguagem de prototipagem.

Segundo a classificação proposta por Dart em [Dar87], podemos dizer que o A_HAND é um ambiente baseado na linguagem, uma vez que suas características mais marcantes são as linguagens que o compõe. Porém, há ainda um editor sensível à sintaxe [Tro90], que caracteriza ambientes baseados na estrutura, e outras ferramentas para tarefas como compilação, interpretação, depuração, catalogação e recuperação de objetos, e gerenciamento de configurações (Figura 4.1).

Na seqüência, apresentaremos detalhes das linguagens Cm e LegoShell que sejam relevantes para a apreciação da modelagem de dados do ambiente.

4.2 A Linguagem de Programação Cm

4.2.1 Características da Linguagem

A linguagem Cm - C modular e polimórfico - foi projetada para ser a linguagem básica do ambiente A_HAND. Sua descrição detalhada pode ser encontrada em [SLD88], [DSi88] e [Fur90a]. Ela é derivada da linguagem de programação C, cujos operadores e comandos foram preservados, incrementando-a com:

- verificação rigorosa de tipos;
- suporte à noção de hierarquia de tipos;
- tratamento uniforme de tipos.

Além disso, é introduzida a noção de classes, que constituem a característica básica da linguagem. As classes são construtores de tipos. Uma vez fornecidos seus parâmetros reais, ela define um tipo específico. Classes sem parâmetros são um caso particular que correspondem, por si só, a um tipo. Em essência, uma classe é um tipo abstrato de dados constituído de:

- parâmetros formais
- lista de classes importadas (cláusula **import**)

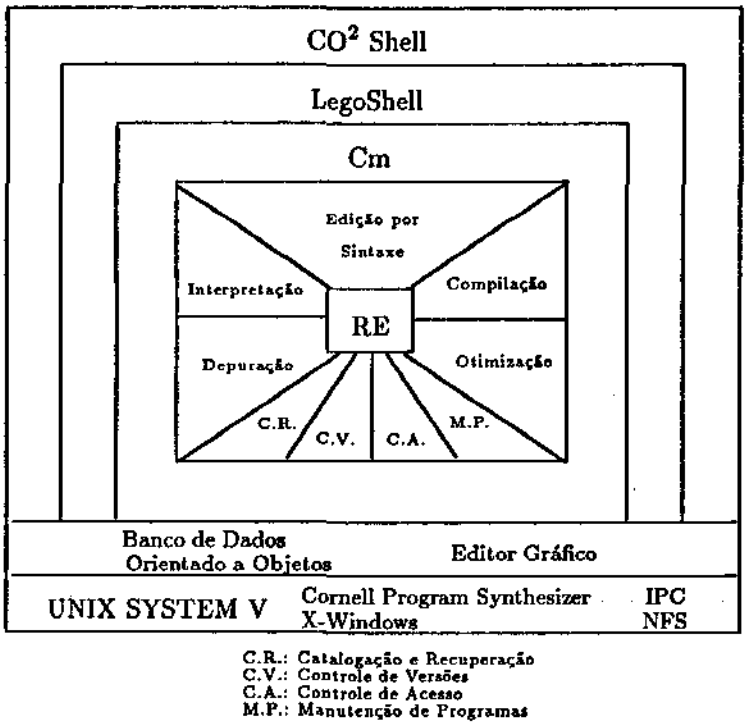

Figura 4.1: Arquitetura geral do ADS A_HAND

- lista de classes usadas (cláusula **use**)
- lista de classes herdadas (cláusula **inherit**)
- estrutura de dados primitivos
- portas
- funções privativas
- funções exportáveis

Os parâmetros podem ser de qualquer tipo, inclusive do tipo "type", o que torna a classe um construtor de tipos. No exemplo abaixo [SLD88], temos uma única definição da classe **Array**. Ela pode ser instanciada em **Array** de inteiros, "strings", ou mesmo de **Array**, bastando para isso passar o tipo dos elementos como valor do parâmetro **Tipo**. Outra característica é a possibilidade de se

definir valores *default* para os parâmetros, como é o caso de "int" para Tipo. Estes valores são utilizados caso eles não sejam especificados em uma chamada.

```
class Array (type Tipo = int; int Size)
    Tipo    Vet[Size];

    export  Tipo Index (int n){
        return (Vet[n]);
    }

    export void PutN (int n; Tipo Elem){
        Vet[n] = Elem;
    }
```

As dependências de uma classe com relação a outras são explicitamente definidas através das listas de classes importadas e usadas. Já que o Cm não permite a definição de classes aninhadas, a relação de importação é a forma de indicar as classes que serão utilizadas dentro da classe corrente. Desta forma, elas podem ser utilizadas como um construtor de tipos qualquer, bem como é possível realizar chamadas às suas funções exportáveis. A cláusula *use* se diferencia do *import* pelo fato de que, além de poder utilizar a classe, ela é compartilhada por várias classes de um mesmo programa. Isto poderia ser alcançado fazendo com que todas as classes envolvidas passassem como parâmetro o objeto declarado na classe que é "raiz" do programa. No entando, isto é tedioso e muitas vezes forçaria algumas classes que não necessitam do objeto a ser compartilhado a ter um parâmetro adicional somente para poder passá-lo para as instâncias das classes importadas que o utilizam. A cláusula *use* resolve este problema criando para cada tipo especificado, um único objeto que é compartilhado por todas as classes que "usam" este tipo [DSi88].

Outro conceito incorporado pela linguagem é a herança de tipos. Sua idéia central é que novas unidades de software possam ser criadas como extensão de unidades já existentes sem que estas sejam alteradas. Como mencionado anteriormente, esta facilidade auxilia a construção de código reutilizável. A linguagem suporta herança múltipla, isto é, uma classe pode herdar características de várias outras classes, que passam a ser suas superclasses. Dessa forma, os atributos e funções exportáveis (declaradas como visíveis para outras classes) das superclasses são incorporadas à classe corrente. Quando alguma função herdada possui o mesmo nome de uma função definida pela classe, a função da classe prevalece sobre a função da superclasse, o mesmo acontecendo com as funções dos tipos

herdados declarados em uma mesma cláusula **inherit**, onde as classes mais à esquerda redefinem as funções das classes mais à direita. Porém, se desejarmos dispor de ambas, é possível redefinir a função da superclasse com um outro nome através da cláusula **rename**, como ilustrado no exemplo abaixo [SLD88].

```
class Device()
    export Open (int Descriptor){
    ... implementacao de Open ...
    }
    ... outras funcoes de Device ...

class Terminal()
    inherit Device();
    rename GenericOpen = Open;

    open (int Descriptor) {
    ... nova implementacao de Open ...
    }
```

A estrutura de dados privativa de uma classe é composta de declarações de constantes, tipos e variáveis, de forma semelhante à linguagem C. O Cm incorpora todas as facilidades desta linguagem, além de permitir a definição de variáveis do tipo classe, desde que definidos seus parâmetros reais. Isto é, classes são construtores de tipos; classes com parâmetros definidos são tipos específicos; e variáveis de tipos específicos derivados de classes podem ser declarados da mesma forma que variáveis de tipos básicos como inteiro e caracter. Chamamos isto de instanciação da classe. Além das classes de armazenamento permitidas na linguagem C, as variáveis definidas dentro de uma classe podem ser variáveis de classe ou de instância. Para o primeiro tipo, existe apenas um exemplar deste elemento, compartilhado por todas as instâncias da classe, enquanto que, para o segundo, existe um exemplar para cada instância. As variáveis de instância descrevem o estado de uma instância.

Uma classe Cm pode ter, ainda, variáveis da classe **Port**, que é pré-definida na linguagem. É por meio de instâncias desta classe que as entradas e saídas são realizadas. Ela difere de outras classes importadas porque suas instâncias são sempre conhecidas externamente a nível de programa, já que é necessário ligá-las a outras portas. Estas podem pertencer a arquivos ou outros programas, formando, assim, um *pipe* entre os mesmos. Isto é efetuado a nível de LegoShell, que será detalhado na seção seguinte. Outro detalhe é que, como as portas de um

programa compartilham de um mesmo espaço de nomes, não é possível existir duas portas com mesmo nome em um mesmo programa. Os parâmetros desta classe são dois: o tipo de operação da porta e o tipo de dados a ser comunicado pela porta. O tipo de operação pode ser de entrada, saída ou ao mesmo tempo de entrada e saída. Quanto ao tipo de dados transmitidos, é possível declará-los como bytes (**Byte**), blocos de tamanho fixo (**Block**) ou blocos de tamanho variável (**VSBlock**). Em uma extensão futura, pretende-se criar portas com parâmetro do tipo "type", que permitirá a comunicação de dados tipados, preservando a estrutura dos seus tipos, além de possibilitar sua verificação.

As funções de uma classe podem ser exportáveis ou privativas, dependendo se podem ou não ser vistas externamente à classe, respectivamente. Ou seja, as funções exportáveis podem ser utilizadas pelas classes que a herdam, importam ou "usam", enquanto as privativas só são utilizadas para executar funções internas à classe. Assim como as variáveis, as funções podem ser também de classe ou instância, aplicando-se a mesma semântica. Com relação à linguagem C, as funções são igualmente providas de facilidades, exceto com respeito aos parâmetros. As funções Cm, assim como as classes, permitem a definição de valores "default" aos seus parâmetros.

4.2.2 Aspectos de Implementação

Um tradutor da linguagem Cm para o C padrão está atualmente sendo desenvolvido [Fur90b]. Esta estratégia de implementação foi escolhida a fim de obter maior portabilidade, uma vez que será possível executar programas escritos em Cm em todas as máquinas onde C é disponível. Alguns aspectos com relação a implementação deste tradutor são importantes para a compreensão de alguns detalhes da modelagem do ambiente, bem como de algumas decisões de projeto da LegoShell, o "configurador" de programas.

Em alguns casos, a tradução de uma classe Cm para C só é possível após a definição dos valores dos seus parâmetros formais porque, dependendo destes valores, uma mesma classe Cm pode gerar diferentes programas C. A verificação desta possibilidade nem sempre é trivial. Por exemplo, um parâmetro do tipo inteiro pode ser apenas o valor de inicialização de uma variável, como também o tamanho de um array em sua declaração. Portanto, inicialmente, teremos uma tradução para cada conjunto de parâmetros definidos. Tomemos como ilustração o parâmetro do tipo "type", que é o mais característico na geração de múltiplas traduções.

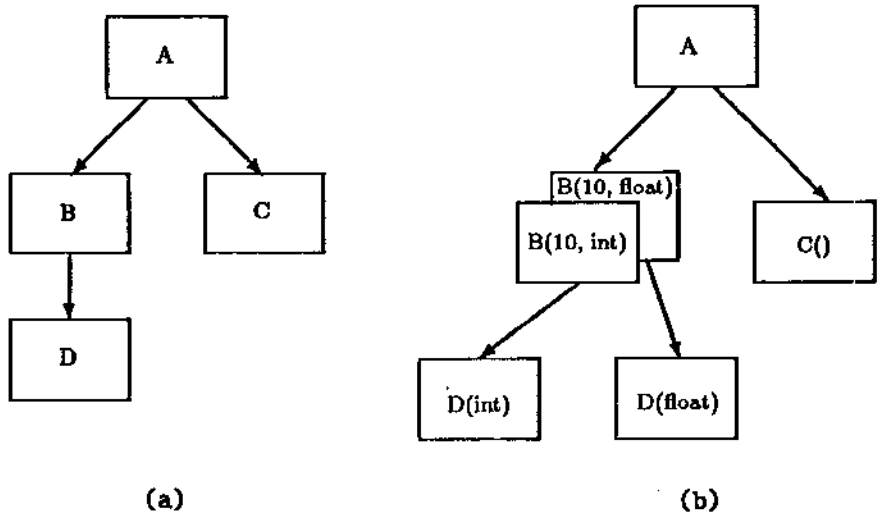

Figura 4.2: (a) Dependências da classe A. (b) Dependências vistas pelo tradutor

Consideremos a classe A do exemplo abaixo.

```
class  A ()
    import B, C;
        B(10, int)    i;
        B(10, float)  x, y;
        C()           w;
    .... funcoes de A ....
```

```
class B (int Size; type Tipo)
    import D;
        D(Tipo)    x;
    .... funcoes de B ....
```

A Figura 4.2a mostra uma representação das dependências desta classe. Porém, devido as múltiplas traduções de B e D, as dependências reais são como representadas na Figura 4.2.b. Assim, antes de realizar a tradução da classe propriamente dita, é necessário que o tradutor verifique se as classes das quais ela depende já estão compiladas para os tipos necessários. Se não existem, é preciso compilá-las antes da compilação da classe corrente. Este processo é semelhante a aquele executado pela ferramenta *make* do sistema Unix, porém com as dependências definidas internamente ao programa.

Existem muitos outros aspectos importantes na implementação do compilador Cm, mas que fogem ao escopo do nosso trabalho.

Uma classe Cm, compilada com todas as suas dependências resolvidas, é potencialmente um programa. Porém, como já vimos, as suas entradas e saídas

estão, a princípio, em aberto, sendo necessário conectá-las a outros objetos. Dessa forma, é possível combinar as classes a fim de criar um objeto com um nível de abstração maior que, por sua vez, pode ser utilizado para criar um outro objeto. Para expressarmos de forma simples estas "configurações" de programas, foi concebida a linguagem LegoShell.

4.3 A Linguagem LegoShell

A LegoShell é uma linguagem gráfica de especificação de comandos complexos chamados *computações*. A idéia é que a construção destes objetos seja tão simples quanto um jogo de montar, onde as peças básicas são:

- programas Cm
- arquivos
- dispositivos periféricos
- conectores: pipe, estrela e "mailbox"

Todos estes objetos possuem portas de comunicação associadas, que são uma generalização da entrada padrão da shell do Unix. Em cada uma delas fluirá um único tipo de dado e em um único sentido (entrada ou saída). A construção de computações consiste em interligar as portas dos objetos utilizando os conectores. Na Figura 4.3, temos um exemplo da computação MSort, constituída de 2 programas Sort e um programa Merge. Note que esta computação pode ser abstraída, tornando-se componente e peça básica para outras computações. Dessa forma, podemos construir computações com níveis ilimitados de abstrações.

Computações como MSort, que não possuem todas as suas portas conectadas, são ditas incompletas. Estas portas em aberto devem, então, ser conectadas a uma borda da abstração, onde uma porta é automaticamente criada. Um exemplo de computação completa é aquela existente olhando o exemplo da Figura 4.3 como um todo.

É possível, caso se queira desprezar os dados associados a uma porta, conectá-la a um "buraco negro". Seu propósito é "fechar" uma porta cujo acesso deva ser negligenciado, como mostrado na saída de dados duplicados do programa Sort.

4.3.1 Os Objetos da LegoShell

Os objetos da LegoShell podem ser agrupados segundo três classificações:

- *composto / primitivo:* se um objeto é construído a partir de outros objetos ou envolve outros em sua definição, ele é dito composto. Por outro lado, objetos primitivos não possuem dependências e têm um mapeamento direto para objetos físicos.
- *ativo / passivo:* objetos ativos são geradores e/ou consumidores de dados, enquanto que objetos passivos são fornecedores e/ou receptores e trabalham sob demanda.
- *objetos com conteúdo semântico / conectores:* esta classificação distingue objetos que possuem conteúdo semântico por si mesmos daqueles que só têm significado dentro de uma computação de LegoShell, interligando os demais objetos.

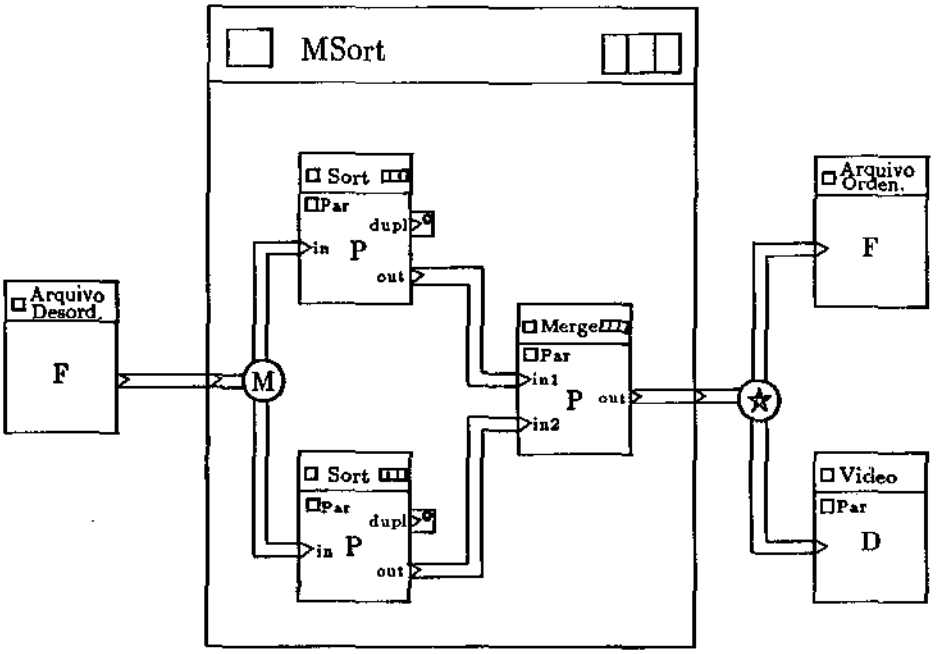

Figura 4.3: Exemplo de uma computação de LegoShell

Na Figura 4.4 apresentamos, de forma tabular, como os objetos se classificam segundo os critérios acima citados.

Na seqüência, detalharemos cada objeto básico da LegoShell.

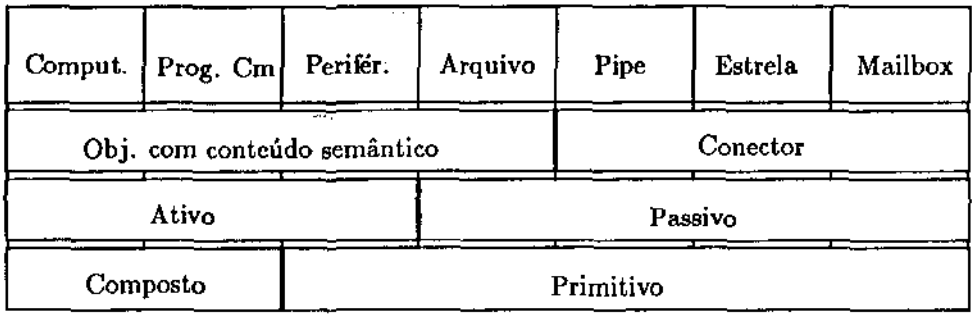

Figura 4.4: Classificação dos objetos da LegoShell

Programas Cm

Na LegoShell, as partes visíveis de um programa Cm restringem-se às suas portas e aos parâmetros da classe. É dentro de uma edição de computação da qual é componente que os valores destes parâmetros são definidos. Porém, é possível fazermos com que estes passem a ser parâmetros da computação a ser abstraída. Dessa forma, os parâmetros do componente passam a ser uma referência a um parâmetro da computação. Por exemplo, na Figura 4.3, se um dos parâmetros dos programas Sort e Merge for o tipo dos elementos a serem ordenados, é interessante que o mesmo passe a ser um parâmetro da computação MSort. Assim, além de evitar que exista um MSort para cada tipo de dado, quando este fizer parte de uma outra computação, é preciso definir apenas uma vez qual o tipo dos elementos que, automaticamente, os programas Sort e Merge serão instanciados para o tipo correto. Através deste mecanismo de abstração, as propriedades polimórficas e de encapsulamento das classes Cm são preservadas a nível de computações de LegoShell.

Um programa Cm possui, ainda, parâmetros de execução, que devem ser definidos para que este possa ser executado dentro de uma computação. Um exemplo deste tipo de parâmetro é a máquina onde ele será executado, visto que o ambiente A_HAND foi idealizado para uma configuração composta de estações de trabalho interligadas em rede. Se analisarmos a sintaxe dos comandos de uma Shell do Unix que permite execução remota, veremos que seu formato é semelhante ao seguinte:

```
UNIX3:: cat -l 100 -f /dev/tty01
```

onde UNIX3:: corresponde ao nome do hospedeiro, cat ao comando, -l -f são opções de execução e 100 /dev/tty01 são os argumentos. Assim, distinguimos no ambiente três conjuntos de parâmetros:

- *parâmetro de classe*, que são os parâmetros definidos pelas classes Cm;
- *chaves de seleção*, que correspondem às opções e
- *parâmetros de execução*, que envolvem, além dos argumentos, o nome do hospedeiro, prioridade de execução e outros.

Arquivos

Um arquivo pode ser visto como um depósito de dados. Ele possui portas de entrada e saída de acordo com a permissão de leitura e escrita que o usuário que o utilizará possui. Estas portas são virtuais, isto é, elas só passam a existir realmente quando são ligadas a um conector. Dessa forma, quando uma porta de saída é utilizada, uma nova porta virtual de saída é criada. Por outro lado, somente um processo pode escrever em um arquivo, onde podemos definir um processo como um programa em execução. Portanto, um arquivo possui no máximo uma porta de entrada. Na Figura 4.3 temos a representação gráfica de arquivos em dois exemplos: arquivo ordenado e arquivo desordenado.

Dispositivos Periféricos

Os dispositivos periféricos podem ser vistos como objetos produtores ou consumidores de dados e que possuem suas portas de entrada e saída pré-definidas de acordo com seu tipo. Assim, um dispositivo de entrada (p.ex. um teclado) será um processo que terá uma porta de saída associada, através da qual fluirão os dados que gerou. Um dispositivo de saída (p.ex. uma impressora) será um processo que terá uma porta de entrada associada, pela qual chegam os dados que deve consumir.

Assim como programas Cm, os periféricos possuem parâmetros, cujos valores são definidos em sessões da LegoShell. No seu primeiro protótipo, todos os dispositivos terão a mesma representação gráfica, como ilustrado pelo objeto "vídeo" da Figura 4.3. Futuramente, haverá a facilidade de associar ícones para cada tipo de periférico.

Conectores

Os conectores são objetos concentradores e distribuidores de dados. Cada conector terá associado um *buffer* no qual armazena os dados que consumiu e ainda

não distribuiu. Enviar um dado a uma porta de conector equivale a colocá-lo no final do *buffer* que o implementa e, receber um dado de um conector, a retirar seu primeiro elemento. O tamanho do *buffer* é um parâmetro que pode ser alterado. Se ele for igual a zero, dizemos que o conector é síncrono, caso contrário, ele é dito assíncrono. A política de consumo e distribuição dos conectores de acordo com sua semântica está descrito em [SDr88].

Na primeira versão da LegoShell, os dados transmitidos pelos conectores são apenas seqüências de bytes. Porém, assim como Balzer [Bal86] defende que a próxima geração de sistemas operacionais fará a comunicação dos processos através de um banco de dados, com a vantagem de terem comunicação de dados tipados, pretende-se, futuramente, que os dados transmitidos pelos conectores tenham tipos associados.

Os conectores previstos pela LegoShell são:

Pipe: define um canal de comunicação unidirecional. Em muitas aplicações, o conceito de canal bidirecional é necessário, o que é equivalente a dois canais unidirecionais. Neste caso, o usuário indicará estas ligações, que serão representadas graficamente como um único canal bidirecional.

Estrela: é semanticamente equivalente a uma rede de disseminação (broadcast) sem colisões. Os dados recebidos pelas portas de entrada são distribuídos para todas as portas de saída. Todas as portas de saída receberão a mesma seqüência de dados. As portas de um conector estrela são ilimitadas, virtuais e invisíveis. Elas só se tornam visíveis quando conectadas a um pipe.

Mailbox: define uma caixa postal, isto é, ao contrário da estrela, as mensagens lidas pelo mailbox são consumidas, sendo distribuídas para um único destino. Suas portas são também ilimitadas, virtuais e invisíveis.

Os conectores estrela e mailbox generalizam o conceito de pipe do Unix, sendo esta uma das principais características da linguagem LegoShell.

4.3.2 Controle de Versões e Configurações do Ambiente

O modelo de controle de versões do ambiente A_HAND foi proposto por Victorelli em [VMD89]. Ele distingue três tipos de versões:

edição: é uma versão ainda em progresso e particular a cada projetista.

efetiva: é criada por promoção de uma edição para ser testada dentro de um grupo de projeto.

liberada: é uma versão efetiva aprovada e, portanto, validada para utilização por todos os usuários do sistema.

Tanto versões liberadas como efetivas não podem ser atualizadas, isto é, estão congeladas. Quando algum projetista desejar atualizar uma destas versões, deve antes derivar uma versão particular da mesma e, então, trabalhar sobre ela.

Os objetos desenvolvidos no ambiente e que, portanto, possuem versões são os programas Cm e computações de LegoShell. Como já vimos, um programa Cm é um objeto composto, construído a partir de relacionamentos de importação, "uso" e herança. Uma vez que os programas possuem versões, é necessário a definição de um plano de configuração que ligue uma versão específica deste objeto composto a versões específicas de seus componentes, ou seja, às classes das quais depende. Portanto, quando um programa participa de uma computação, sua identificação apenas não basta. É preciso definirmos a versão deste programa e a configuração que resolve suas dependências. Da mesma forma, existem configurações de computações LegoShell que, quando utilizadas dentro de outra computação, sofrem o mesmo processo de seleção de versão e configuração.

Devemos distinguir neste ponto uma definição de configuração (dc) de uma visão de execução (ve). Uma visão de execução é composta por um conjunto S de pares do tipo (objeto de projeto, número de versão), ou (OP, v), onde "v" identifica univocamente uma versão. O número da versão é determinado a partir das entradas que compõe uma definição de configuração. Em outras palavras, a definição de configuração determina como o número das versões são encontradas e a visão de execução origina-se da resolução de uma definição de configuração para um determinado objeto composto. Neste modelo, as entradas de uma definição de configuração podem ser de três tipos:

Direta (OP,i): este tipo de entrada especifica de forma explícita a versão que será copiada em S. "i" pode ser um número de versão ou um rótulo. No primeiro caso, o par (OP, i) é diretamente inserido ao conjunto S. Para o segundo caso, os rótulos possíveis são: ULTLIB, ULTEFET e CORRENTE. Caso seja especificado ULTLIB, o número de versão correspondente à última versão liberada é procurada e o par (OP, número de versão) é inserido em S; o rótulo ULTEFET corresponde ao número de versão da última versão efetiva e o rótulo CORRENTE corresponde à última edição do usuário envolvido.

Indireta (OP, dc): a versão do objeto OP a ser inserida em S é aquela determinada utilizando a definição de configuração dc para a sua resolução.

de Inclusao (dc, p): as entradas de inclusão determinam uma prioridade p para a definição de configuração dc. São construídos conjuntos de pares S', S", ..., para cada entrada de inclusão. É feito, então, um "merge" destes conjuntos da seguinte forma: para cada objeto de projeto OP, será escolhido o número de versão correspondente à entrada de inclusão de maior prioridade. Esta versão só será inserida ao conjunto S, caso ainda não exista em S nenhuma entrada para este objeto de projeto.

Para os componentes cuja versão não foi resolvida pela definição de configuração, é utilizada uma definição de configuração "default". Ela baseia-se na propriedade dos objetos componentes e pode ser alterada segundo as necessidades do projetista. Inicialmente, o ambiente A_HAND adotará a seguinte configuração default:

- para os objetos pertencentes ao projetista, será utilizado o rótulo CORRENTE;
- para os objetos não pertencentes ao projetista, mas a algum membro de seu projeto ou ao projeto propriamente dito, será utilizado o rótulo ULTEFET;
- para todos os demais, será associado o rótulo ULTLIB.

Porém, esta definição poderá ser alterada, tanto a nível de ambiente, como de projeto e usuário. Isto é, será possível definirmos diferentes configurações default para cada projeto, bem como diferentes configurações default para cada usuário membro destes projetos. A prioridade dentre estas definições será a de mais baixo nível, ou seja, a do usuário.

Em sua primeira versão, o ambiente não dará suporte a versões alternativas, permitindo concorrência somente no último nível. Portanto, o histórico de versões típico de um objeto terá a seqüência como mostra a Figura 4.5.

Os números de versão utilizados têm a seguinte notação:

E.I + U.Ed, onde

E: é o número de versão externa ou de interface, que é acrescido de um quando a interface é alterada;

I: é o número de versão interna, que é acrescido de um quando o objeto é alterado somente internamente. Quando o campo E é acrescido de um, o campo I é automaticamente zerado;

+: indica derivação de versão. Significa que a versão corrente é derivada da versão indicada por E.I;

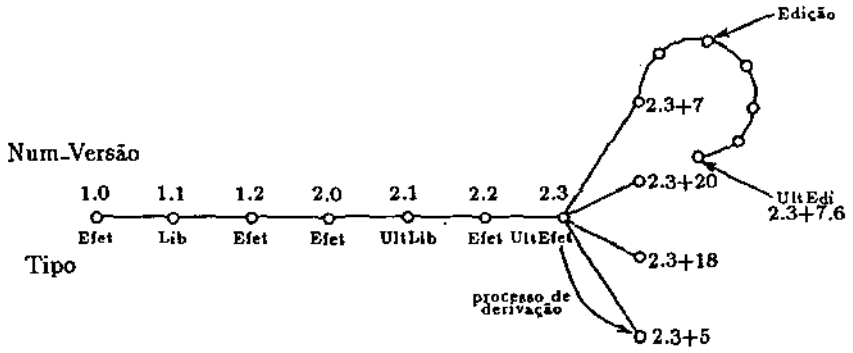

Figura 4.5: Histórico de versões do ambiente A_HAND

U: é a identificação do usuário. Cada usuário pode ter uma versão particular de um determinado objeto (seu objeto corrente). É somente nesse nível que a concorrência pode existir;

Ed: identifica o número da edição feita pelo usuário. Estas edições funcionam como aquelas feitas em um editor de textos usual. Porém, suas versões são armazenadas em um banco de dados privado de cada usuário. O ambiente permitirá que cada usuário defina o número de edições que serão mantidas, porém a responsabilidade de gerenciamento de espaço destes banco de dados particulares será responsabilidade de seu proprietário.

Quando um dos usuários consegue efetivar uma de suas versões particulares (edições), as versões particulares dos demais usuários passam automaticamente a ser derivadas desta nova versão efetiva. No esquema de Figura 4.5, se 2.3+7.6 é efetivada, ela passa a ser a última efetiva, com número 2.4 (ou 3.0). As demais versões derivadas de 2.3 passam de 2.3+20 para 2.4+20, de 2.3+18 para 2.4+18 e assim por diante. Esta notação de versões não será necessariamente mantida internamente. Ela apenas representa o esquema lógico do modelo para facilitar sua compreensão.

A criação e resolução de configurações são funções que serão desempenhadas pelo editor de LegoShell.

4.3.3 Descrição Funcional da LegoShell

O editor da LegoShell, além de permitir a manipulação dos objetos na composição de computações, executará também funções inerentes ao ambiente. Diversas

ferramentas serão colocadas à disposição do usuário através deste editor, obtendo a integração do ambiente através de uma interface padronizada. Uma descrição detalhada da interface pode ser encontrada em [Ari90].

No protótipo do editor, a interface será como ilustrado na Figura 4.6. Distinguimos nela 5 áreas:

- *área indicadora de nível:* ela é composta por uma ou mais barras de título contendo o nome do objeto sendo editado. Como é possível suspender temporariamente uma edição, iniciando outras, para depois retomá-la do ponto em que foi abandonada, as barras indicam as edições suspensas no momento. Quando da suspensão, dizemos que descemos um nível de edição, enquanto que quando a retomamos dizemos que subimos um nível de edição.
- *menu de barras:* contém as principais funções do ambiente.
- *área de trabalho:* nela será desenvolvido o trabalho de edição.
- *menu de ícones:* nesta área estão dispostos os ícones dos objetos do ambiente que podem ser selecionados e incluídos na área de trabalho durante uma edição. Em uma extensão futura, será possível associar ícones a objetos utilizados com muita freqüência no ambiente para facilitar sua inclusão.
- *área de visualização:* corresponde a uma janela de visualização total do objeto, contendo uma janela de zoom, que mostra a parte do objeto sendo representada na área de trabalho.

Na seqüência, descreveremos a semântica de cada função listada no menu de barras, juntamente com alguns detalhes que influenciaram diretamente a modelagem do ambiente. Concentraremos a descrição das funções sobre as computações de LegoShell. Elas podem ser aplicadas aos programas Cm apenas quando explicitamente citado.

Help - auxílio eletrônico

Como um dos objetivos do ambiente é a ausência de manuais, uma das formas de atingi-lo é contarmos com um procedimento de auxílio "on-line". Esta ajuda estará sempre relacionada com o contexto da edição e pode referir-se ao significado do estado atual, à próxima ação a ser executada, a condições de erro ou ao significado de parâmetros e variáveis.

No futuro, será possível ajustar o ambiente a um nível de auxílio de acordo com o conhecimento que o usuário possui sobre o mesmo. Assim, usuários mais experientes usariam simplificações de comandos ao invés de passar por vários menus para atingir a função desejada.

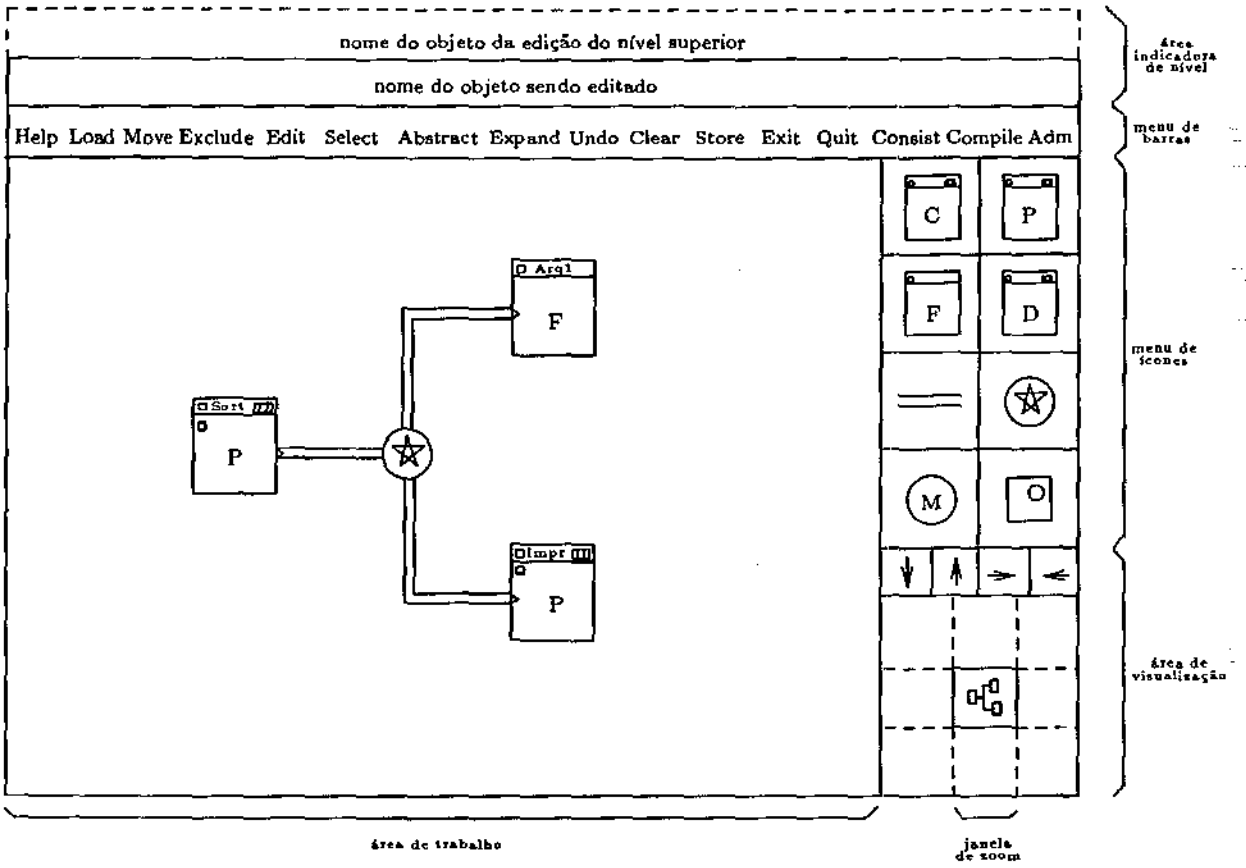

Figura 4.6: Interface do protótipo do editor da LegoShell

Load - edição de um objeto composto

A função **load** corresponde a editar um objeto, caso ele já exista, ou criá-lo, caso contrário. Este objeto pode ser um programa Cm ou uma computação. No primeiro caso, um editor sensível à sintaxe será invocado para desempenhar sua função; caso contrário, o próprio editor de LegoShell executará a edição.

Os direitos de acesso de leitura e escrita é que definem as funções disponíveis a cada usuário. Além disso, se um objeto já estiver sendo editado por um usuário, o mesmo não pode criar uma nova sessão de edição para o mesmo objeto, podendo apenas visualizá-lo.

Como já foi dito, apenas versões do tipo edição podem ser atualizadas. Portanto, se a função **load** for invocada e o usuário possuir direito de escrita sobre o objeto, automaticamente uma nova edição particular é criada. Neste caso, a configuração associada à computação é novamente resolvida e os objetos componentes que modificam sua versão são apresentados para que o usuário decida se estas serão ou não utilizadas. Por exemplo, se a entrada da configuração indica ULTLIB para o objeto componente A e uma nova versão da mesma é liberada, então o usuário terá a opção de incorporar esta nova versão à computação.

Durante o processo de edição, os componentes da computação serão incluídos e ligados através de conectores. Porém, na inclusão de objetos compostos, não basta qualificá-los apenas com o nome, apesar deste ser unívoco em todo o ambiente. Como sua representação gráfica depende do número de portas, é necessário que antes definamos a versão do componente a ser utilizada. Portanto, é preciso que a toda edição, mesmo que se trate de uma computação nova, esteja associada uma configuração. Convencionou-se que, no caso de criação de um objeto, a configuração a ser utilizada será a "default" de mais baixo nível. Se o objeto já existir, sendo a edição, portanto, uma derivação da última versão, a configuração a ser associada será a mesma utilizada pela versão da qual se originou.

Include - Inclusão de componentes A inclusão de um componente em uma computação corresponde à escolha do tipo no menu de ícones (computação, programa, periférico, arquivo ou conector), à sua identificação, no caso de objetos de conteúdo semântico e, finalmente, ao seu posicionamento dentro da computação. Quanto à identificação dos objetos de conteúdo semântico, é possível fazê-lo diretamente fornecendo o seu nome ou utilizando a facilidade do ambiente de listar os objetos segundo a classificação abaixo:

- *Uso Geral:* objetos classificados como sendo de uso geral, como por exemplo, classes Cm Data, Array e Pilha.

- Projeto (x): objetos classificados como de projeto, cujo proprietário seja o projeto x ou um membro de x. Se o usuário que efetua a edição é um membro de x, todos os objetos que satisfazem a regra acima serão listados. Caso contrário, somente aqueles que possuem versões liberadas serão apresentados.
- Particular: objetos que estão sendo editados pelo usuário corrente.

Devemos ressaltar, ainda, que apenas computações e programas Cm consistentes podem ser incluídos em uma computação. A inclusão de *pipes* na computação deve ser posterior à inclusão dos objetos interligados pelo mesmo. Para isto, é necessário apontar o ícone correspondente ao pipe e, então, posicionar sobre as suas portas de entrada e saída.

Move - Alteração geométrica de componentes

Esta função possibilita que a posição de objetos dentro de uma computação possa ser alterada. O *pipe* é uma exceção a esta facilidade, pois, a princípio, este será apenas uma reta ligando as portas relacionadas.

Exclude - Exclusão de componentes

A exclusão de componentes de uma computação é feita através da função **exclude**. Se, eventualmente, houver *pipes* ligados ao objeto selecionado, optou-se por excluí-los juntamente com o objeto.

Edit - Alteração semântica de componentes

A alteração dos componentes pode se dar nos planos de interface, conteúdo e versão. Assim, são quatro as opções desta função:

- edição de um objeto componente
- edição da configuração
- edição dos parâmetros dos componentes
- edição dos parâmetros do objeto corrente

Edição de um objeto componente A edição propriamente dita, com direito a escrita, será permitida somente se o usuário estiver utilizando a versão corrente do objeto, isto é, se a versão utilizada for uma edição. Assim, ao se retornar da edição, a versão alterada passará automaticamente a ser utilizada dentro da

computação. Esta situação é bastante comum no teste de novos objetos ou alteração dos mesmos.

Se a versão utilizada não for uma edição, o usuário terá apenas direito de leitura sobre o objeto. Caso alguma alteração sobre o mesmo for necessária, este deverá utilizar a função LOAD e, então, alterar a configuração da computação para que a nova versão seja incorporada.

Edição da configuração Para fazermos a alteração da versão de cada componente de uma computação ou das dependências de um programa Cm, utilizaremos a função de edição de configuração. Seguindo o modelo de controle de versões e configurações do ambiente já descrito, existem 4 tipos de entrada:

1. direta,
2. direta com rótulo, que pode ser ULTLIB, ULTEFET ou CORRENTE,
3. indireta, e
4. de inclusão,

onde os três primeiros são relacionados com um objeto componente e o último com a computação ou programa como um todo.

A configuração assim determinada, passará então a ser a *configuração ativa* do objeto, isto é, aquela que determina a versão dos seus componentes.

Edição dos parâmetros dos componentes Para os três tipos de parâmetros existentes (parâmetros da classe, chaves de seleção e parâmetros de execução), será permitido que se defina o seu valor ou que o mesmo seja exportado como parâmetro da computação da qual é componente. Esta exportação pode se dar de três formas:

- passando a ser parâmetro da computação com o mesmo nome que possuía no objeto componente
- passando a ser parâmetro da computação com um nome diferente do que possuía no componente, no caso de haver parâmetros com o mesmo nome e ambos serem exportados
- parâmetros de objetos distintos passando a ser um único parâmetro na computação e, portanto, com um nome único.

É importante salientar que um número será associado a cada objeto componente, o que permitirá uma referência sem ambigüidades aos diferentes componentes, mesmo que eles sejam duas cópias de um mesmo objeto.

Edição dos parâmetros do objeto corrente É através desta função que determinamos o valor "default" dos parâmetros exportados para a computação corrente.

Select - Função de seleção

Há casos em que desejamos efetuar uma mesma operação, por exemplo exclusão, sobre vários objetos. Para facilitar estas ações existe a função de seleção, implementado através de uma lista circular dos objetos selecionados. Assim, é possível andar sobre esta lista utilizando os comandos **anterior** e **próximo** e realizando a operação desejada. São previstas, ainda, opções de selecionar todos os objetos da computação e de liberação de todos os objetos.

Abstract - Abstração de computações

Esta função consiste em criar uma nova computação composta pelos objetos selecionados através da função **select**, descrita acima, substituindo-os pela mesma na computação corrente. Desta forma, portas conectadas a objetos não selecionados, bem como portas não conectadas passam a ser portas da nova computação.

Expand - Função de expansão

É a função oposta da abstração, isto é, ela consiste em substituir uma computação por seus componentes dentro da computação corrente. Isto, logicamente, implica que a computação corrente seja redesenhada.

Undo - Função de anulação

Genericamente, significa voltar ao estado de edição anterior à última operação, eliminando ou revertendo seus efeitos. Para cada estado particular, esta função possui um significado especial.

Clear - Limpa tela

Como o nome já indica, a função consiste em excluir todos os objetos componentes da computação, mantendo-se no mesmo nível de edição, ou seja, não abandonando a sessão de edição.

Store - Função de armazenamento

Esta função de armazenamento é utilizada no caso de desejarmos guardar o estado do objeto corrente e prosseguir sua edição. Assim, um novo número de versão privada é criada e nos mantemos no mesmo nível de edição.

Exit - Função de armazenamento e saída

A função implica em armazenamento e abandono de edição. O nível ao qual a edição retorna é retratado pelas opções disponíveis:

- Exit do ambiente
- Exit do nível de edição

No primeiro caso, o estado do ambiente, incluindo toda a cadeia de sessões correntes, é armazenado para que a edição possa ser reassumida no mesmo ponto onde foi suspensa.

A segunda opção consiste em armazenar o objeto, criando uma nova versão, e subir um nível de edição. Caso não haja edições suspensas, isto é, se o nível do objeto editado for o primeiro, entramos no modo criação de um novo objeto.

Quit - Abandono de edição

A função **quit** significa abortar a edição atual, sem armazená-lo, e subir um nível de edição.

Consist - Função de consistência

A verificação de consistência através desta função é feita somente no nível de edição corrente e não inclui a geração de código. Como uma das regras de inclusão de objetos complexos em uma computação é que estes estejam consistentes, mantemos, desta forma, a abstração necessária entre os diversos níveis.

A consistência de uma computação implica, entre outras coisas, na verificação da existência de objetos ou portas não conectados, compatibilidade dos dados que transitam entre duas portas e consistência da definição de parâmetros.

Compile - Função de consistência e geração de código

Esta função é equivalente à função **consist**, porém, inclui a geração de código. Esta geração não é uma operação trivial devido a alguns aspectos da execução de computações.

Execução de computação Os objetos de conteúdo semântico poderão ser monitorados de maneira interativa durante a execução. Para este propósito, eles possuem três "botões", correspondentes às funções de ligar (TURN ON), desligar (TURN OFF) e suspender (PAUSE).

Assim, no início da execução, todos os objetos estarão como seus botões TURN ON ligados, implementados através da criação de diversos processos concorrentes. Se a função PAUSE for ativada, o efeito será a suspensão da execução do processo correspondente àquele objeto, podendo ser reassumida novamente através da função TURN ON. O efeito da função TURN OFF é equivalente a abortar o processo que implementa o objeto. Por exemplo, na computação MSort da figura 4.3, se desligarmos um dos programas Sort, o processo continuará sua execução, porém apenas com um Sort. Os objetos desligados podem reiniciar sua atividade através da função TURN ON, mas isto implica em criar novamente todos os processos necessários.

A definição do ambiente prevê, ainda, a possibilidade de execução de objetos através de interpretação, caso estes não tenham sido compilados. Porém, a estratégia de implementação do módulo de execução do ambiente não está totalmente acabada.

Adm - Funções de gerenciamento

As funções que permitirão o gerenciamento de projetos são:

- ativação de uma configuração já existente para o objeto corrente
- definição da configuração default para um usuário, projeto ou para o todo o ambiente
- efetivação e liberação de versões
- funções de *tailloring*, como definição do tamanho de *buffer default* para os tipos de conectores e definição do número de edições particulares a serem guardadas
- cadastramento de usuários e projetos
- atribuição de responsabilidades
- definição de direitos de acesso

4.4 Observações Finais

A descrição funcional do ambiente pode parecer tediosa e irrelevante no contexto de modelagem de dados. Porém, o que se notou foi que sua definição, além de dar uma visão mais profunda do ambiente como um todo, teve um papel determinante em muitos pontos do seu modelo de dados, como veremos no capítulo seguinte.

Capítulo 5

Modelagem do ADS A_HAND

Este capítulo trata da modelagem do ambiente de desenvolvimento de software (ADS) A_HAND utilizando a metodologia de projeto apresentada no capítulo 3.

5.1 Introdução

Concentraremos o trabalho de modelagem do ambiente A_HAND na descrição funcional do editor de LegoShell e nas necessidades do compilador da linguagem Cm, ambos atualmente em desenvolvimento. Apenas como ilustração, apresentamos na Figura 5.1 o modelo da linguagem Cm. No restante do processo de modelagem, não trataremos as estruturas da linguagem neste nível de detalhe, pois as necessidades reais das ferramentas que as manipularão não estão totalmente definidas. Provavelmente, nem todos os objetos representados constituirão fisicamente um objeto, mas serão armazenados em uma estrutura particular que proporcione facilidades para sua manipulação.

5.2 O Modelo Global

A Figura 5.2 mostra o modelo global do ambiente, que será a base da nossa discussão. A existência de um modelo a nível de objetos reais foi importante não só como uma forma de expressar a visão que temos sobre o sistema, mas também como um meio de comunicação entre os membros responsáveis pelas suas diferentes partes e aquele cuja função é a modelagem dos dados. À medida que o ambiente se torna mais familiar, através do detalhamento de suas funções, nós o vemos de uma forma diferente. É assim que, por exemplo, no primeiro modelo da LegoShell, não havia o objeto Porta, mas sim o objeto Dado, que era

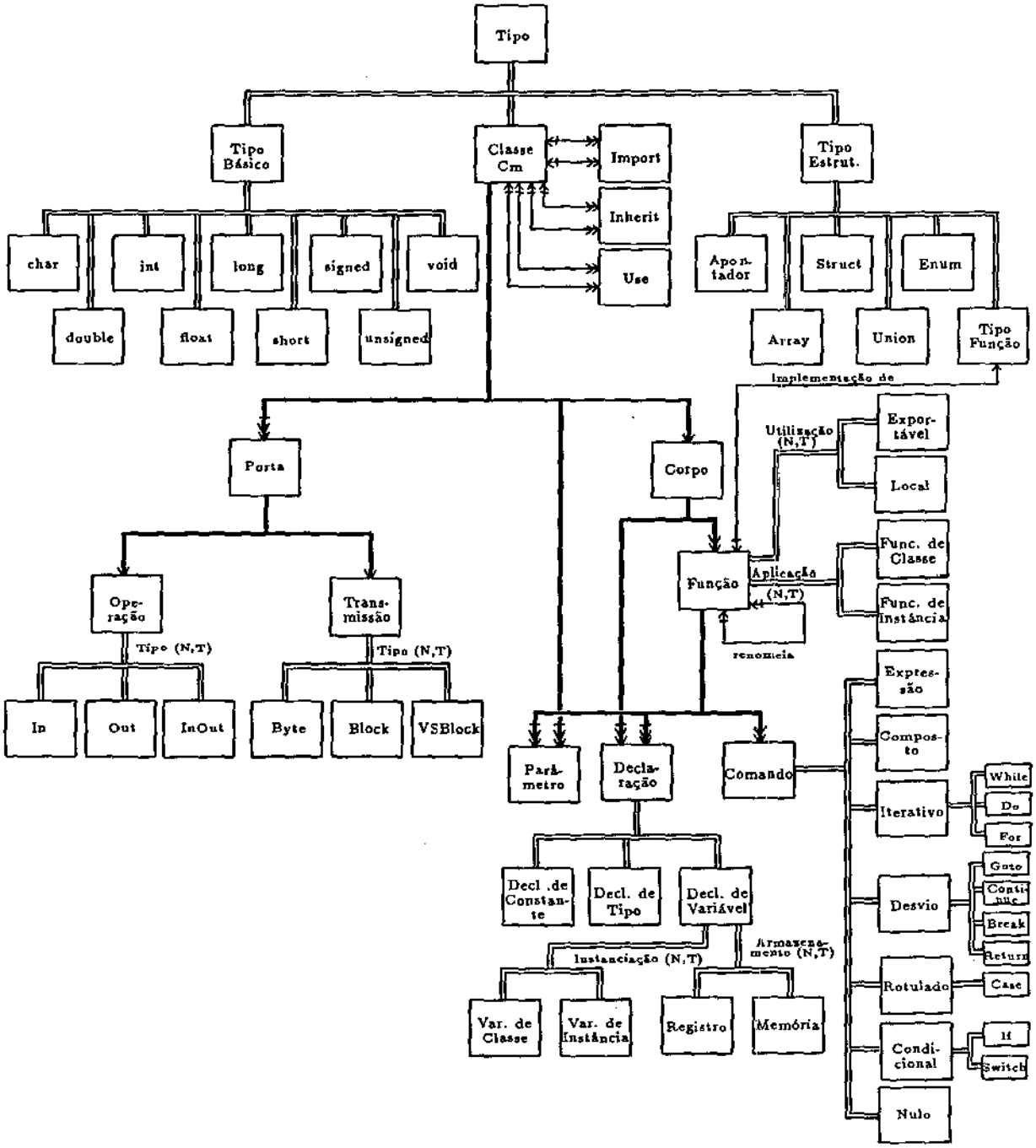

Figura 5.1: Modelo global da linguagem Cm

transmitido entre os componentes de uma computação. Porém, são as portas dos componentes que desempenham papel mais importante, uma vez que são elas os pontos de conexão dos objetos.

No processo de criação do modelo global, um ponto essencial foi a padronização da nomenclatura dos objetos do sistema. Já no detalhamento informal da funcionalidade do ambiente, é importante que identifiquemos uma estrutura básica comum aos objetos, de forma a organizar as informações recebidas, mesmo que de uma forma não definitiva. Na experiência de modelagem do ADS A_HAND, esta foi uma fase de aprimoramento de definições, onde conceitos de entidade e relacionamento confundiram-se com conceitos advindos do paradigma de orientação a objetos. A estrutura resultante é a divisão em 3 níveis (classe, versão e instância) na etapa de detalhamento das propriedades dos objetos.

5.3 Detalhamento das Propriedades dos Objetos

Apesar de, no modelo global, haver o conceito de generalização/especialização, o detalhamento das propriedades é feito somente nos objetos de nível mais baixo. Idealmente, deveríamos manter este conceito, implementado através da herança de propriedades e operações. Porém, quando uma ou mais subclasses possuem versões, estes conceitos podem se tornar incompatíveis. Por exemplo, no modelo da Figura 5.2, a classe Objeto possui como componentes as classes Porta e Parâmetro e relacionamento de "representado por" com a classe Ícone, dentre outros. Sendo a classe Computação uma subclasse de Objeto, o primeiro herdará as propriedades do último. Mas como Computação é uma classe que possui versões (e também intâncias de versões) não fica claro em que nível se dará a herança. No caso, os componentes farão parte de uma versão de computação, enquanto o relacionamento pertencerá à classe Computação. Portanto, não seria correto afirmarmos que Objeto é uma superclasse de Computação nem de versão de Computação. Para que tais incompatibilidades não ocorressem seria necessário que objetos como Computação não fossem representados apenas por um objeto no modelo global, mas como diferentes conceitos identificados no nível detalhado. Porém, isto diminuiria o seu nível de abstração, tornando-o mais confuso. Além disso, o modelo de dados Damokles, com o qual foi desenvolvida a implementação, não incorpora o conceito de generalização, obrigando-nos a fazer esta eliminação no projeto físico, isto é, no mapeamento para a sua DDL. Apenas aqueles objetos onde todas as subclasses possuem as mesmas propriedades serão detalhadas no nível mais alto, como é o caso do objeto Parâmetro.

Do modelo global, obtemos as seguintes classes: Computação, Programa

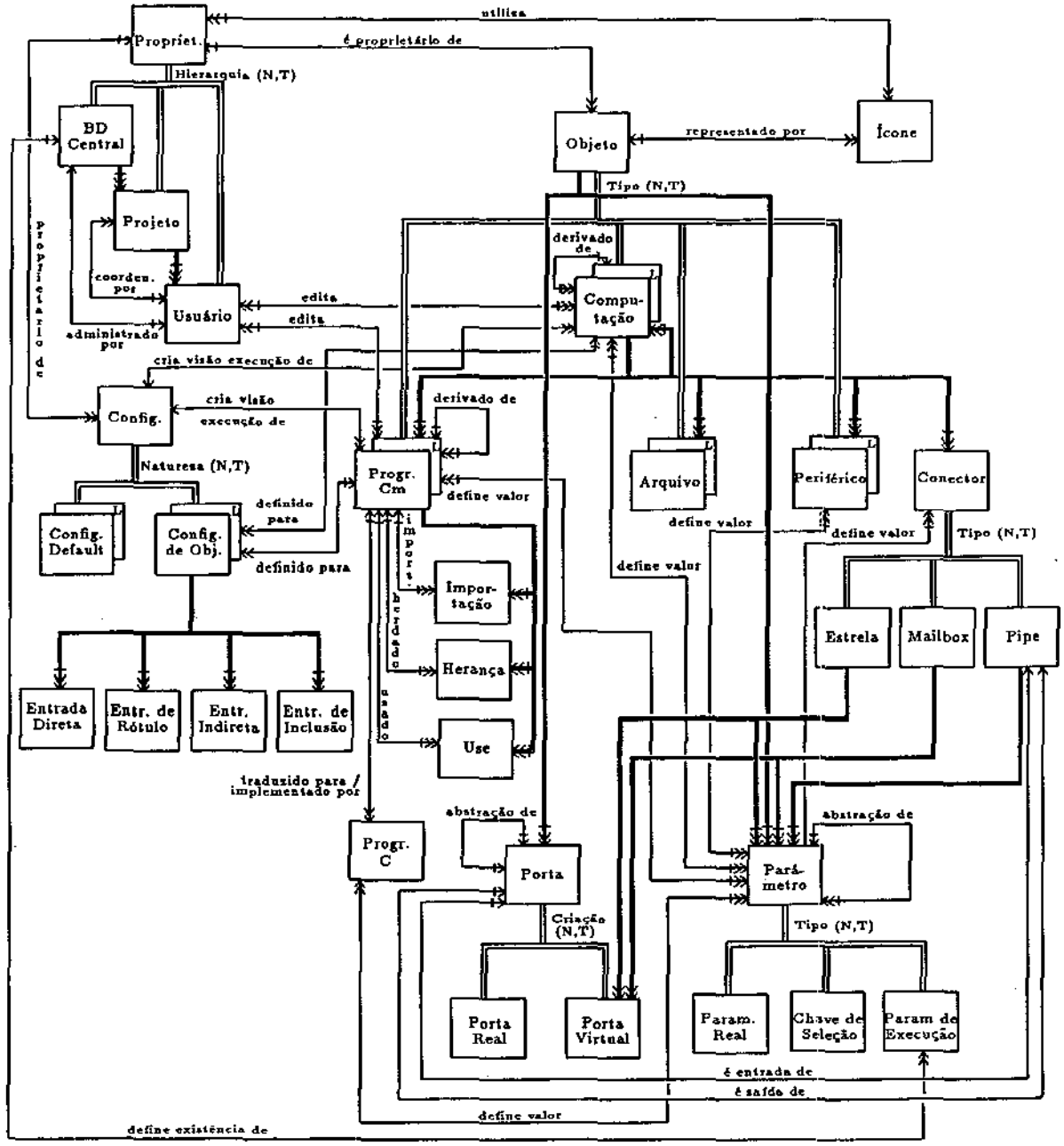

Figura 5.2: Modelo global do ambiente A_HAND

Cm, Arquivo, Periférico, Estrela, Mailbox, Pipe, Porta Real, Porta Virtual, Parâmetro, Ícone, Programa C, Configuração Default, Configuração de Objeto, Entrada Direta, Entrada de Rótulo, Entrada Indireta, Entrada de Inclusão, BD Central, Projeto e Usuário. No detalhamento dos atributos de cada uma delas é que será especificado a qual nível do objeto os componentes e relacionamentos identificados no modelo genérico está relacionado. No Apêndice A apresentamos o detalhamento dos objetos acima citados. Tomemos como exemplo o objeto Computação.

```
OBJETO  Computacao
    CLASSE
        Componentes:
        Relacionamentos:
            . objeto de: Configuracao de Objeto (CLASSE)
            . propriedade de:  Usuario / Projeto / BD Central
            . representado por: Icone
            . relacionado a: Entrada de Rotulo
            . relacionado a: Entrada Indireta
        Atributos:
            . nome
            . ultima versao liberada
            . ultima versao efetiva
            . ultima edicao
            . documentacao
            . classificacao (uso geral, de projeto, particular)

    VERSAO  /* diferentes implementacoes da classe computacao */
        Componentes:
            . Objetos Componentes:
                . Computacao (INSTANCIA)
                . Programa Cm (INSTANCIA)
                . Arquivo (INSTANCIA)
                . Periferico (INSTANCIA)
                . Estrela (INSTANCIA)
                . Mailbox (INSTANCIA)
                . Pipe (INSTANCIA)
            . Porta Virtual
            . Parametro
        Relacionamentos:
```

```
            . configurado por: Configuracao Default (VERSAO) /
                               Configuracao de Objeto (VERSAO)
            . editado por: Usuario
            . propriedade de: Usuario
            . derivado de / derivado por: Computacao (VERSAO)
            . relacionado a: Entrada Direta
        Atributos:
            . numero da versao
            . tipo de versao (edicao / efetiva / liberada)
            . data de criacao
            . data de efetivacao
            . data de liberacao
            . versao anterior
            . versao posterior
            . descricao da alteracao
            . estado da computacao
              (inconsistente / consistente / compilado)
            . codigo de execucao / interpretacao
            . numero do ultimo componente

    INSTANCIA  /* um componente de uma computacao */
        Instancia de: Computacao (VERSAO)
        Componentes:
            . Porta Virtual
        Relacionamentos:
            . define valor de: Parametro
        Atributos:
            . numero do componente dentro da computacao
            . posicao
            . ligacao das portas

FIM Computacao
```

Note que a definição da classe, versão e intância diferem totalmente uma da outra. No caso, a classe possui os atributos comuns a todas as implementações de uma computação, representadas pelas suas versões. E a instância, por sua vez, é composto por valores necessários a uma computação quando esta passa a ser componente de outra computação.

Na especificação de cada nível, um componente ou relacionamento pode pas-

sar a pertencer a apenas um deles ou repetir-se. Em geral, quando há repetição, a semântica atribuída ao mesmo em cada nível é diferente. Por exemplo, o relacionamento **propriedade** da classe de computação possui o propósito de controlar o acesso às suas versões, enquanto que, em cada versão, o proprietário é aquele que a criou. Quanto a **Porta Virtual**, componente da versão e instância, apenas aquela definida para as versões existe conceitualmente. Porém, se houver duas instâncias de uma mesma computação como componentes de uma outra computação, e se tivermos que ligar suas portas através de pipes, não basta apenas sabermos de que versão as intâncias foram criadas e as portas existentes nesta versão, mas é necessário associar as portas às suas instâncias. Assim, a Porta Virtual de uma instância é, na realidade, uma "representação" da Porta Virtual da versão de computação da qual foi instanciada.

Um outro passo do detalhamento é a especificação dos níveis dos objetos referenciados através dos relacionamentos e componentes, isto é, se ele envolve uma instância, versão ou classe do objeto, representados entre parêntesis no exemplo acima. Porém, quando iniciamos esta etapa, não sabemos exatamente que semântica adotaremos para cada objeto. Portanto, esta é uma tarefa que devemos executar após ou paralelamente a definição dos demais objetos. Assim, inicialmente, basta sabermos que os componentes de uma versão de computação são computações, programas Cm e outros, sem nos importarmos com a especificação do seu nível.

5.4 Detalhamento das Funções de um Objeto

A etapa seguinte do detalhamento de um objeto corresponde a associação de funções aos mesmos, nos seus diversos níveis. A elaboração destas funções baseia-se totalmente no conhecimento de seus atributos e da funcionalidade do ambiente. O Apêndice B contém a definição de todos os objetos. Continuemos com o exemplo do objeto Computação.

OBJETO Computação

- **Classe**
  - cria / remove classe de computação
  - obtém configuração de objeto associado à computação
  - obtém / altera proprietário da computação
  - obtém ícone associado à computação utilizado por um usuário
  - associa / desassocia ícone à computação

- obtém computação pelo nome
- seleciona as computações particulares, de projeto e de uso geral de acordo com a classificação
- obtém direitos de um usuário sobre uma computação (Derivação, Leitura, Execução)
- verifica se é relacionado a uma entrada de rótulo de uma determinada versão de configuração
- verifica se é relacionado a uma entrada indireta de uma determinada versão de configuração
- obtém quantidade de versões
- obtém a última versão

• **Versão**

- cria / remove versão de computação
- obtém os objetos componentes de uma versão de computação
- inclui / exclui um objeto componente de uma versão de computação
- obtém as portas de uma versão de computação
- inclui / exclui uma porta de uma versão de computação
- compara portas de duas versões de computação
- obtém os parâmetros de uma versão de computação
- inclui / exclui um parâmetro de uma versão de computação
- compara parâmetros de duas versões de computação
- obtém a configuração que determina as versões dos objetos componentes
- associa / desassocia a uma configuração que determina as versões dos objetos componentes
- obtém proprietário da versão de computação
- associa / desassocia proprietário de uma versão de computação
- copia componentes de uma versão de computação para outra
- copia versão de computação
- inclui / exclui versão de computação da lista de sessões de um usuário com um determinado direito de acesso

- verifica se uma versão de computação está sendo editada por um usuário
- verifica se pode ser removida
- verifica se existem instâncias de uma versão de computação
- verifica se é relacionado a uma entrada direta

• **Instância**

- cria / remove instância de computação
- obtém versão de computação da qual foi instanciada
- obtém nome da classe da qual foi gerada
- obtém portas de uma instância de computação
- inclui / exclui portas a uma instância de computação
- define / altera valor de parâmetro da versão de computação da qual foi instanciada
- copia / transfere uma instância de uma computação para outra
- verifica se todos os parâmetros possuem valores definidos
- verifica se todas as portas estão conectadas
- obtém o tipo de entrada da configuração que determina a versão de uma instância de computação (Direta, UltLib, UltEfet, Corrente, Indireta, Inclusão)
- associa / desassocia uma versão como derivação de outra
- verifica se uma versão é derivada de outra

Uma característica da definição das funções e também das propriedades é que não nos preocupamos em obter uma identificação única para cada objeto através de atributos, como ocorre no projeto para um BD relacional. Pressupomos que esta identificação exista automaticamente, uma vez que a maior parte dos BD's orientados a objetos proporciona esta facilidade, o que denominamos de *surrogate key*. Além disso, as funções não incluem aquelas destinadas à obtenção e alteração dos atributos dos objetos. Isto é apenas uma simplificação, pois estas operações devem ser implementadas como tal para que haja realmente encapsulamento de sua representação.

Como próximo passo, temos o detalhamento das funções de alto nível da interface. Este é um processo de refinamentos sucessivos até atingirmos o nível das funções especificadas para cada objeto. Por exemplo, em uma primeira versão, a operação "Load de Computação" poderia ser descrita da seguinte forma:

LOAD de Computação
    SE for criação
        cria nova computação
        cria nova edição em branco para o usuário
        associa edição à configuração default de mais baixo nível
        coloca edição na lista de sessões do usuário
    SE a computação já existe
        verifica se o usuário tem permissão para editar
        cria uma edição para o usuário
        copia os componentes da última versão para a nova edição
        associa a edição à configuração igual a versão anterior
        resolve a configuração
        avisa o usuário se a configuração muda a versão de algum
            componente
        coloca a edição na lista de sessões do usuário

Nesta definição, vários detalhes foram omitidos, como, por exemplo, verificar se o limite de edições para o usuário foi ultrapassado ou se já existe uma edição da computação na lista de sessões do usuário. É através da incorporação de novas condições e detalhamento daquelas já existentes que alcançamos o nível das funções dos objetos. Além disso, esta tarefa é uma forma de validá-las, bem como de identificar novas necessidades.

5.5 Implementação do Modelo

Até o momento, nada foi discutido com relação à distribuição dos dados. Porém, esta é uma questão muito importante, principalmente em aplicações como o ADS A_HAND, projetado para uma arquitetura de rede de estações de trabalho. Nesta aplicação, onde as transações geralmente são longas e envolvem um grande volume de dados, é interessante que os mesmos fiquem disponíveis na estação na qual a transação se desenvolve. Em uma primeira instância, a distribuição dos dados será implementada da forma ilustrada na Figura 5.3. Isto é, haverá um banco de dados particular para cada usuário e um banco de dados central que concentrará todas as informações compartilháveis.

Definida a forma de distribuição, o detalhamento das funções dos objetos, bem como a especificação das operações de alto nível da interface, devem ser revisados a fim de verificar se é necessária a inclusão de novas funções para a manipulação de múltiplos bancos de dados. Por exemplo, no caso do objeto

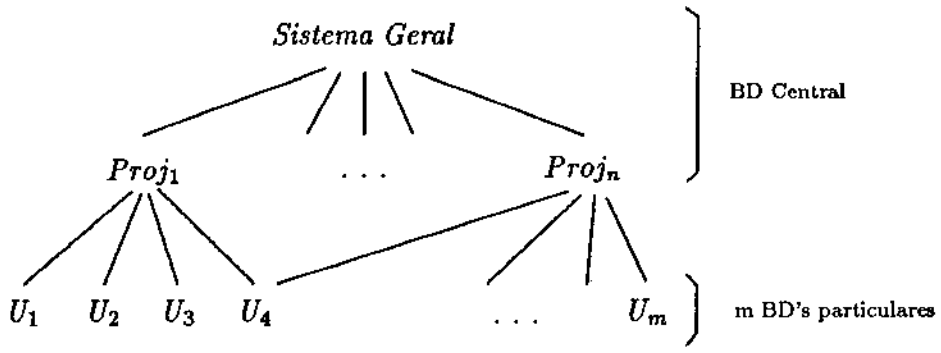

Figura 5.3: Esquema de distribuição de dados do ADS A_HAND

Computação, o relacionamento de derivação será sempre entre o BD Central e um BD particular. A definição final detalhada de algumas operações do editor da LegoShell está descrita no Apêndice C.

Passamos, então, à última etapa, que corresponde à implementação do modelo. Seu mapeamento para a DDL Damokles e a utilização das funções DML na implementação das funções será tratada no capítulo seguinte, uma vez que muitas decisões de projeto são conseqüências de características ou limitações do SGBD, o que nos dará subsídios na argumentação quanto às necessidades de um banco de dados para aplicações como ambientes de projeto.

Capítulo 6

Utilização de um BDOO

Neste capítulo, apresentaremos a implementação do modelo do Ambiente de Desenvolvimento de Software (ADS) A_HAND, em conjunto com algumas características do SGBD Damokles que determinaram as decisões de projeto. Conclusões obtidas através desta experiência quanto às facilidades oferecidas por este SGBD para suportar Ambientes de Desenvolvimento de Software (ADS) serão também relatadas.

6.1 Introdução

A criação de um banco de dados orientado a objetos surgiu da necessidade de um meio de armazenamento que suportasse objetos de estrutura complexa de forma mais natural, isto é, representando-o como uma imagem do mundo real. Portanto, idealmente, em se tratando de BDOO, terminado o projeto conceitual, o seu mapeamento para estruturas de armazenamento seria praticamente automática. Ou ainda, que não haveria necessidade do mapeamento, desde que a modelagem conceitual utilizasse os mesmos conceitos suportados pelo SGBD.

Este nível de abstração é obtido se for possível tornar todos os detalhes de implementação transparentes para a linguagem de definição de dados do banco de dados. Porém, se isto não for possível, a facilidade com que estes dados serão manipulados pode depender da forma com que foram declarados. Na seqüência, relataremos este tipo de interferência na utilização do SGBD Damokles para a implementação do modelo do ADS A_HAND, apresentado no capítulo anterior.

6.2 Distribuição dos Dados

O SGBD Damokles é capaz de manipular múltiplos banco de dados em um mesmo equipamento ou distribuído em uma rede. A cada banco de dados existente é relacionado um proprietário, que pode ser um grupo de usuários ou um usuário. Dizemos que um banco de dados é público caso pertença aos primeiros, e privado, caso contrário. Além disso, é permitido que se crie uma hierarquia de grupos de usuários, ou seja, grupos podem fazer parte de outros grupos, sendo que os grupos de mais baixo nível são formados por usuários. É esta hierarquia que determina, estaticamente, os direitos de acesso de cada usuário aos bancos de dados.

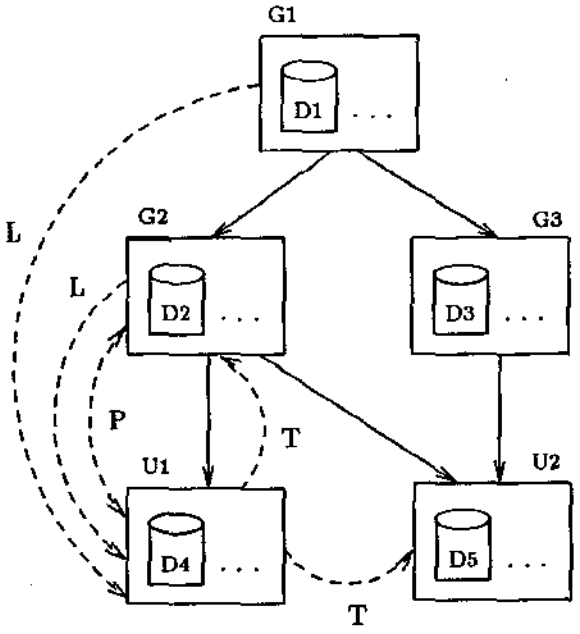

Figura 6.1: Direitos de acesso do SGBD Damokles [Abr88]

Os tipos de direitos de acesso exitentes são:

escrita: é o direito unicamente do proprietário do banco de dados e que pode realizar qualquer operação sobre o mesmo.

leitura: o direito de leitura é garantido a todos os membros diretos e indiretos de um grupo de usuários com relação a todos os bancos de dados deste

grupo. Por exemplo, na Figura 6.1, ao usuário U1 é permitido ler dos bancos de dados D1, D2 e D4.

transferência: é o direito assegurado aos membros diretos de um grupo de usuários com relação aos banco de dados pertencentes a este grupo e àqueles pertencentes a outros membros do mesmo grupo. Assim, o usuário U1 pode transferir ou copiar dados de seu próprio banco de dados para o banco de dados D2, bem como para D5.

projeto: corresponde ao direito de executar *checkout* de objetos, isto é, copiar objetos de um banco de dados público para um particular a fim de fazer modificações sobre o mesmo para, ao final do trabalho, devolvê-lo ao banco de dados original. Ele só é permitido aos membros diretos do grupo de usuários ao qual pertence o banco de dados público. Na Figura 6.1, o usuário U1 pode fazer *checkout* somente do banco de dados D2, enquanto o usuário U2 pode fazê-lo dos bancos de dados D2 e D3.

Este tipo de controle de acesso é muito restritivo pois o nível de proteção é somente a nível de banco de dados e não a nível de objetos. Além disso, não há nenhuma facilidade para alterar dinamicamente os tipos de acesso permitidos. Portanto, ou ele se adapta totalmente ao sistema a ser desenvolvido, ou nos vemos obrigados a não utilizá-lo, criando um esquema de segurança particular. Uma opção para contornar este problema seria dividir os objetos em bancos de dados segundo o acesso aos objetos neles contidos. Porém, esta abordagem não é interessante, uma vez que existem as seguintes restrições quanto ao particionamento dos dados [Abr88]:

- um objeto ou relacionamento está localizado somente em um banco de dados, isto é, um objeto estruturado não pode estar distribuído entre diferentes bancos de dados;
- um relacionamento pode envolver objetos de diferentes bancos de dados, porém a instância do relacionamento existe somente em um deles;
- versões só poderão existir dentro do banco de dados do seu objeto genérico, ou seja, o grafo de versões de um dado objeto deve estar totalmente armazenado em um banco de dados.

À primeira vista, estas restrições nos parecem bastante razoáveis. Porém, a maior parte dos problemas encontrados na utilização do SGBD Damokles referem-se a estas limitações na manipulação de múltiplos bancos de dados.

Como em qualquer aplicação na qual os dados estão intimamente ligados, o A_HAND possui um grande número de relacionamentos. Segundo uma das

restrições acima citadas, quando um relacionamento envolve objetos de bancos de dados distintos, a instância do relacionamento fica armazenado em somente um deles. Assim, se desejarmos verificar sua existência ou obter o segundo objeto, a pesquisa teria que percorrer todos os bancos de dados possíveis. Além disso, o Damokles suporta apenas distribuição dos dados como uma rede do tipo estrela, onde um nó, que chamamos de servidor, concentra todos os pedidos e acessos a outros bancos de dados, sugerindo uma arquitetura do tipo servidor - cliente. Por estas razões, a arquitetura adotada para o A_HAND é bastante simples, mas que proporciona ao menos a localidade dos dados de um usuário. Assim, temos dois níveis, onde no nível mais alto temos um banco de dados central, concentrando todos os dados compartilhados e, no nível mais baixo, um banco de dados privado para cada usuário, sendo o primeiro armazenado no servidor e os demais nas estações de trabalho particulares.

Nesta implementação, o esquema de segurança dos dados também é bastante simples, não havendo a preocupação de sobreposição, uma vez que todos os dados compartilhados estarão centralizados e nunca poderão ser alterados sobre as versões já liberadas ou efetivadas. Isto é, se algum usuário desejar modificar algum objeto do banco de dados central, será necessário derivar uma versão do mesmo para o seu banco de dados particular, criando uma edição de trabalho. Para que esta edição passe a ser compartilhada, passando então para o banco de dados central, é necessário que ela seja efetivada, função atribuída a um coordenador de projeto, cujos membros utilizá-la-ão para testes. Posteriormente, esta versão pode ser liberada, tornando-se disponível para ser utilizada por outros usuários. Assim, o compartilhamento de uma versão depende sempre da aprovação de um superior. Os direitos de derivação, leitura e execução são controlados através dos atributos classificação e propriedade de cada objeto. Note que o proprietário de uma computação, por exemplo, tem conotações diferentes para o objeto genérico e suas versões. Para o primeiro, o proprietário é uma variável para controle de acesso, enquanto que, para os segundos, o proprietário é o criador da versão e é imutável.

O direito de derivação é definido da seguinte forma: se o proprietário do objeto for o banco de dados central, todos os usuários poderão derivá-lo; se pertencer a um projeto, somente os membros do projeto e seu coordenador terão direito a esta operação, enquanto que objetos que são propriedade de um usuário só poderão ser derivados por ele mesmo. Quanto ao direito de leitura, ela é estendida a todos, independentemente do proprietário, se o objeto possuir versões liberadas. Além disso, se o proprietário for um usuário, todos os membros dos projetos dos quais ele participa também possuem acesso de leitura a seus objetos. O direito de execução corresponde à execução de objetos completos, ou seja, programas

e computações com todas as dependências resolvidas, e também à utilização de objetos incompletos na composição de uma nova computação ou programa Cm. Este direito é determinado principalmente pelo atributo classificação de cada objeto. Assim, dado um usuário, ele tem direito de execução sobre os objetos cuja classificação seja `particular` e pertença a ele próprio, a todos aqueles com classificação `projeto`, sendo o proprietário ou um projeto ao qual o usuário pertença ou um membro deste. Além disso, os demais objetos com classificação `projeto` que possuam versões liberadas e todos aqueles com classificação uso `geral` também poderão ser utilizados.

6.3 Mapeamento para a DDL

A Data Definition Language (DDL) Damokles suporta os conceitos de objeto, relacionamento, atributo e versão. Portanto, o mapeamento de um modelo conceitual, como aquele apresentado no capítulo 3, para a DDL é praticamente direta, apenas utilizando algumas regras para moldá-lo a certas limitações não consideradas até o momento. Este é o caso de atributos que são identificadores de outros objetos. Em geral, ele será transformado em um relacionamento com o objeto participante. Por exemplo, os atributos última versão liberada, última versão efetiva e última edição dos objetos Computação e Programa Cm são transformados nos relacionamentos Ult_Lib, Ult_Efet e Ult_Edi entre os respectivos objetos genéricos e suas versões, como mostrado no resultado do mapeamento do modelo A_HAND, contido no Apêndice D. Outra transformação necessária é quanto aos atributos multivalorados. A maneira mais natural de implementá- los seria através da cláusula de construção de objetos compostos STRUCTURE. Porém, se houver necessidade de qualquer tipo de ordenação, o mais apropriado é a utilização de relacionamentos, uma vez que estes oferecem a facilidade ao usuário de controlar sua seqüência, ao contrário dos objetos componentes, controlados pelo próprio gerenciador.

Como a DDL Damokles não possui um conceito semelhante ao conceito de variáveis de classe e variáveis de instância das linguagens de programação, adotamos três níveis no projeto conceitual, sendo o terceiro correspondente às propriedades das instâncias. Também neste caso, há duas possibilidades de implementação: declarando os objetos que representam as instâncias como componentes da classe ou como um relacionamento. Neste caso, um detalhe externo às características da DDL determinou a opção pelo relacionamento, isto é, uma vez que o ambiente compreende múltiplos bancos de dados, possivelmente a classe estará armazenada em um banco de dados e a instância seja necessária em outro,

o que contraria uma das restrições se as instâncias forem componentes da classe.

Como já foi dito anteriormente, este tipo de problema surge a partir do momento que passamos a pensar no ambiente distribuído. Idealmente, a distribuição física dos dados deveria ser totalmente independente do seu esquema. Isto é, sem restrições quanto ao tipo de dado armazenado em cada nó na rede. Mas, devido às restrições citadas na seção anterior, é necessário que no momento da criação do esquema já haja a preocupação quanto a distribuição física dos dados. Assim, tomando como exemplo o objeto Programa Cm, teríamos como componentes os programas por ele "usados", "importados" ou herdados". Uma vez que estes são tipicamente objetos que podem estar localizados em outro banco de dados, eles terão que ser mapeados para relacionamentos no esquema DDL. Neste caso em particular, estes relacionamentos existiam previamente porque, do ponto de vista da linguagem Cm, eles são componentes da definição de uma classe Cm, mas para a linguagem LegoShell, eles são relacionamentos de dependência de um programa e determinam os demais programas envolvidos na sua configuração. Porém, as intâncias dos relacionamentos podem continuar fazendo parte do objeto como um componente, já que é interessante que estejam localizados no mesmo banco de dados que o objeto que os contém. A decisão de tornar um relacionamento componente de um objeto, além de se levar em conta que ambos deverão estar contidos em um mesmo banco de dados, depende de se ele deve continuar fazendo parte do objeto em uma eventual cópia utilizando a função **db_cpo** da DML Damokles. Por exemplo, se uma instância de computação **I_Computação** for copiada, o relacionamento de instanciação, que a liga à classe geradora, deveria ser mantido. Portanto, ele deveria ser um componente da instância. Da mesma forma, os relacionamentos Val_Param, Abstr_Param e Bur_Negro fariam parte da instância. Em contrapartida, este tipo de raciocínio pode levar a uma distribuição maior dos relacionamentos. No caso do relacionamento de instanciação, se concentrássemos todos eles junto à classe geradora, no momento de verificarmos a existência de instâncias, bastaria uma pesquisa no banco de dados que a contém, enquanto que, se o relacionamento estiver junto às instâncias e, havendo a possibilidade destas existirem em vários bancos de dados, a pesquisa teria que percorrer quantos bancos de dados fossem necessários. Como a função **db_cpo** não será utilizada nesta implementação, por motivos que serão discutidos na seção seguinte, optou-se, sempre que possível, concentrar os relacionamentos junto ao objeto único quando a cardinalidade for de 1:n.

Outra alteração com relação ao projeto conceitual foi quanto ao relacionamento de derivação. À princípio, o relacionamento envolveria duas versões de objetos compostos, isto é, versões de computação ou programa Cm. Como os objetos em desenvolvimento por um usuário estarão armazenados em seu banco

de dados particular e outras versões liberadas ou efetivadas do mesmo estarão localizadas no banco de dados central, não é possível, segundo as restrições, que apenas as versões particulares, ou seja, edições de um usuário, estejam armazenadas em outro banco de dados. Assim, a opção adotada foi a de se criar um outro objeto genérico do mesmo objeto no banco de dados particular do usuário, ligando ao mesmo suas edições. Portanto, o relacionamento de derivação passa a ser entre o objeto genérico do banco de dados central e o objeto genérico do banco de dados particular. Esta forma de implementação traz a vantagem de que, quando uma nova versão do objeto for efetivada, não é preciso alterar os relacionamentos para que as edições passem a ser derivações da nova versão, pois isto fica implícito, já que a derivação é sempre da última versão do objeto genérico relacionado.

Na realidade, esta forma de manipulação de objetos de projeto não utiliza a facilidade das transações longas proporcionada pelo SGBD Damokles, dentro das quais é possível fazer *checkout* e *checkin* de objetos de um banco de dados público para um particular a fim de fazer modificações sobre o mesmo. É que, mesmo durante este tipo de transações, as restrições de distribuição devem ser respeitadas. Portanto, no caso da derivação de objetos, se fizermos um *checkout* da versão da qual queremos derivar, só será possível fazer alterações sobre a própria versão, uma vez que não é permitido que se crie novas versões de um objeto genérico cujo grafo de versões não esteja totalmente localizado no próprio banco de dados. Além disso, isto incompatibilizaria a existência de edições de diferentes usuários, pois o objeto deve ser protegido contra escrita enquanto outro o modifica. Uma opção seria criar uma edição no BD Central e então fazer um *checkout*. Neste caso, se optarmos por manter mais de uma versão do tipo edição para cada usuário, seria necessário que a cada nova edição criada executássemos um *checkin* da edição terminada. A desvantagem é que não mantemos as edições, que são versões particulares, no banco de dados do usuário e sim no banco de dados central. Portanto, a fim de mantermos a proposta inicial de armazenar objetos particulares em um banco de dados próprio a cada usuário, optamos por criar um objeto genérico particular que possui relacionamento de derivação com o banco de dados central.

Características de outras operações de manipulação de dados também podem influenciar na criação do esquema, tornando-o mais eficiente. Um exemplo é o caso de versões que possuem relacionamentos com o seu objeto genérico como em Ult_Lib, Ult_Efet e Ult_Edi. Já dissemos anteriormente que eles podem ser implementados como relacionamentos da forma mostrada no Apêndice D. Porém, como a operação de inclusão de versões devolve um número de versão que o identifica e existe também uma operação que faz uma busca direta a partir do objeto

genérico com base neste número, é possível implementar este tipo de situação como mostrado no objeto Config_Default. Neste caso, o número de versão da configuração default ativa é armazenada como um atributo do objeto genérico e a obtenção da versão, ao invés de navegar através de um relacionamento, passa a ser uma busca direta.

De uma forma geral, se não considerarmos as restrições quanto à distribuição, mas se virmos a DDL Damokles somente como um modelo de dados, acreditamos que baseá-lo em objeto estruturados foi uma decisão bastante acertada, sendo esta facilidade essencial em se tratando de ambientes de projeto. Além disso, existe a cláusula UNION, similar a da linguagem C, que proporciona várias facilidades. Ela é utilizada na construção de alternativas de classe e também participa na implementação do conceito de generalização, apesar de não haver herança de propriedades. Por exemplo, no ambiente A_HAND, uma definição de configuração pode ser uma configuração default ou uma configuração de objeto, sendo que tanto uma quanto a outra podem definir as versões dos componentes de um objeto. Portanto, define-se um objeto Configuração como UNION dos objetos Configuração Default e Configuração de Objeto e um relacionamento entre o objeto Configuração e um objeto composto. Quando, a partir do objeto composto, quisermos encontrar sua configuração, a navegação independe do tipo de configuração que será atingida. Essa é uma das facilidades de recuperação de dados obtida através da utilização desta cláusula.

Um outro ponto a ser destacado é a generalidade do conceito de versões. Apesar do nome, elas podem representar uma seqüência de objetos com estruturas idênticas que podem ser mascarados da forma desejada, como se houvesse uma forma de declarar a existência de um grafo dirigido e operações para manipulá-lo. Apesar de não ser utilizado na implementação do A_HAND, permite-se, também, que versões possuam versões.

6.4 Implementação das Funções

A linguagem de manipulação de dados (DML) do SGBD Damokles possui uma interface funcional embutida na linguagem de programação C. Ela pode ser caracterizada como uma interface de "um objeto/relacionamento por vez". Ao contrário de interfaces orientadas a conjuntos como o SQL, todas as funções DML retornam apenas um objeto ou relacionamento a cada chamada. Além disso, ela incorpora o conceito de identidade forte, chamada de *surrogate key*.

A implementação de todas as funções definidas para o ambiente A_HAND é uma tarefa bastante extensa. Portanto, optou-se pela implementação da função

LOAD de computação, pois ela envolve praticamente todos os objetos do ambiente e também incorpora características importantes para uma experiência com o SGBD Damokles. Além das funções de criação e remoção de edições, atualização da lista de sessões do usuário e gerenciamento do número de edições, a função LOAD é composta por dois módulos principais:

- cópia de uma versão de computação: utilizado no caso do usuário desejar modificar um objeto já existente, sendo necessário, portanto, criar uma nova edição, inicialmente idêntica a versão da qual é derivada;
- resolução de uma configuração: na criação de uma edição de objeto já existente, a versão dos objetos componentes são novamente determinados segundo a definição de configuração associada, para que o usuário possa optar pela utilização de uma nova versão, caso ela exista.

O código fonte C resultante para esta operação é de aproximadamente 3200 linhas. Além disso, foi implementado um módulo de carga de dados e outro para auxiliar na depuração dos programas, cuja função é listar todos os objetos e relacionamentos existentes em um banco de dados.

6.4.1 Cópia de uma Versão de Computação

Similarmente à edição de textos, onde iniciamos uma nova sessão do estado em que a deixamos, uma nova versão deve partir do estado de sua última versão. Portanto, a sua criação, seja na operação de derivação do banco de dados público para um particular, como na criação de uma outra edição, deve iniciar com a cópia da versão anterior.

A DML Damokles possui a operação de cópia de objetos. Para definir a semântica das operações sobre objetos estruturados ou com versões, o Damokles possui os conceitos de *extent*, *cut* e *dispersion*. Eles estabelecem quais os objetos que são envolvidos juntamente com o objeto composto neste tipo de operação.

O *extent* **ext(o)** de um objeto ou versão é recursivamente definido como o conjunto [Abr88]:

1. o
   se *o* não possui componentes nem versões
2. o U ext(c1) U ... U ext(cm) U ext(v1) U ... U ext(vn)
   se *o* possui componentes c1..cm e versões v1..vn.

Expressando em palavras, o *extent* de um objeto ou versão *o* inclui todos os objetos, relacionamentos e versões que podem ser atingidos a partir do objeto

o, navegando uma seqüência arbitrária de relacionamentos de componente ou versão.

Como o modelo de dados Damokles permite que objetos estruturados possuam componentes que se sobreponham, ou seja, permite o compartilhamento de objetos, criou-se o conceito de *cut*, **cut(o)**, definido como o conjunto de objetos e relacionamentos pertencentes ao *extent* de *o* mas que também pertencem ao *extent* de um objeto *o'*, tal que *o'* não pertence a ext(o). Ou seja, o *cut* envolve os elementos do *extent* de um objeto estruturado que são compartilhados por outro objeto estruturado.

As operações de cópia e tranferência de objetos de um banco de dados para outro afetam todo o seu *extent*. Mas como não é permitido que uma versão exista em um banco de dados sem o seu objeto genérico, os objetos genéricos (e possivelmente objeto genérico de objeto genérico) das versões pertencentes a este *extent* são também envolvidos na operação. Em conjunto com os objetos genéricos, os seus *extent*'s são também copiados. Novamente, a cadeia de objetos genéricos das versões atingidas são determinadas e assim sucessivamente. O conjunto de objetos e relacionamentos que são indiretamente afetados desta maneira é chamado de *dispersion* do objeto original da operação.

Apesar da operação de cópia envolver automaticamente todo o *extent* e *dispersion* de um objeto, ele não foi utilizada na implementação de cópia de uma versão de computação. Como já foi mencionado anteriormente, a operação de derivação de uma versão foi implementada criando-se, no banco de dados particular, um novo objeto genérico. Se utilizássemos a operação de cópia oferecida pelo Damokles, não somente a última versão seria copiada para o banco de dados particular, mas todo o grafo de versões do objeto genérico, uma vez que ele faz parte do *dispersion* da versão. Por outro lado, poderíamos optar por manter todas as edições no banco de dados central, trazendo a última versão para o banco de dados particular através da operação de *checkout*. Mesmo assim, a operação **db_cpo** não poderia ser utilizada na criação de novas edições, já que ela não adicionaria uma nova versão ao objeto, mas criaria um novo objeto idêntico ao primeiro. É que não é possível indicarmos que a versão copiada será acrescentada ao grafo de um objeto já existente. Ela sempre copia o objeto genérico original e todo o seu grafo de versões. Mesmo que isto fosse possível, a opção de manter as edições no banco de dados central criaria versões alternativas no último nível de versões. Na efetivação de uma destas versões, as outras passariam automaticamente a ser sucessoras daquela efetivada, sendo necessário, portanto, atualizar o grafo de versões. Entretanto, o Damokles não dispõe de operações de manipulação direta do grafo, mas apenas de inserção e remoção de versões no final do mesmo. Não é possível, então, manter a existência de uma versão e mudar

somente a sua lista de antecessores ou sucessores.

Portanto, a cópia de uma versão de computação foi implementada criando-se individualmente cada uma de suas partes. Isto é uma pena pois a operação **db_cpo** é muito poderosa. Por maior e mais intrincado que seja o objeto copiado, a operação mantém a correspondência entre os objetos originais e aqueles criados. Assim, se um relacionamento fizer parte do *extent* do objeto copiado juntamente com o outro objeto a ele relacionado, um novo relacionamento é criado entre as cópias dos objetos originais. Se o objeto envolvido não for copiado, a cópia do relacionamento se mantém com o objeto original. A manutenção desta correspondência é que torna a implementação da cópia trabalhosa. Tomemos como exemplo a derivação da computação ilustrada na Figura 6.2. As instâncias componentes de uma versão de computação não são compartilhadas, pois elas concentram apenas os dados e relacionamentos que podem ser alterados dentro de cada uma delas. Portanto, para que a instância de pipe que liga as portas de PROG1 a PROG2 seja copiada corretamente, à medida que as instâncias destes programas são criados, a correspondência entre as portas das instâncias de programa da computação original e das novas instâncias precisam ser mantidas. Isto também é necessário para a criação das abstrações das portas da nova computação, ou seja, o relacionamento que indica que a porta de um componente é ligada à borda da computação, como é o caso de PV1 de PROG1 e PV5 de PROG2.

Outra correspondência que requer a existência de uma estrutura auxiliar é a abstração de parâmetros. Este relacionamento determina que parâmetros das instâncias componentes passem a ser parâmetros da computação, sendo que, quando os valores dos mesmos são definidos, automaticamente eles são transmitidos àqueles dos quais foi abstraído. Isto está ilustrado na Figura 6.2 com o parâmetro *tipo*, que era, originalmente, de PROG1 e passou a ser um parâmetro de COMP1. Ao contrário das portas (virtuais), que pertencem às instâncias, os parâmetros são componentes das classes ou versões das quais elas foram instanciadas. Portanto, o relacionamento envolve, além das instâncias componentes e da computação, a classe instanciada, que pode ou não estar armazenada no mesmo banco de dados do componente. Portanto, a cópia dos parâmetros da computação antecedeu a cópia das instâncias, criando-se paralelamente uma lista ligada contendo os identificadores do parâmetro na computação origem, do parâmetro da nova computação, do componente cujo parâmetro foi abstraído e do banco de dados que contém o parâmetro abstraído. Assim, na criação dos componentes da nova computação, bastou verificar se o componente correspondente da computação origem pertencia a esta lista. Em caso afirmativo, todas as informações necessárias para inserir o relacionamento de abstração de parâmetros para a nova

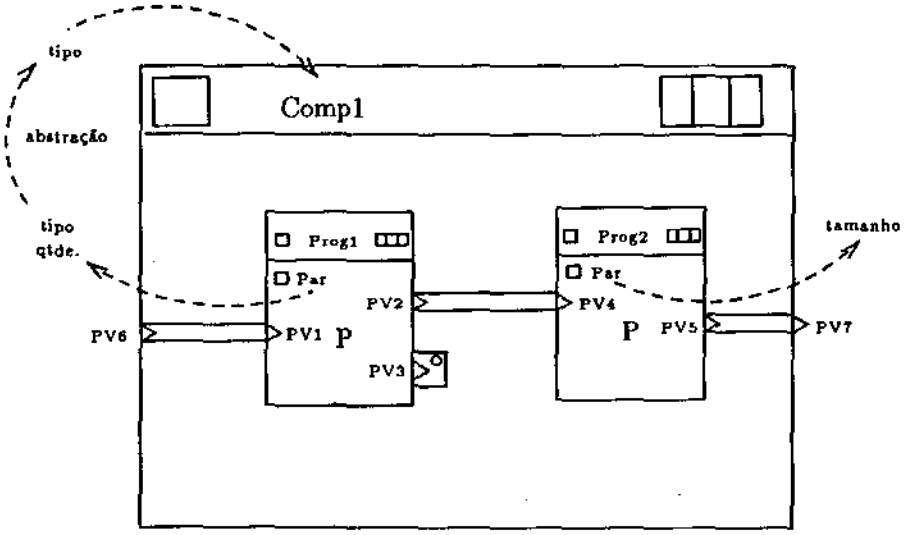

Figura 6.2: Exemplo de uma Computação

computação estavam disponíveis na lista.

6.4.2 Resolução de uma Configuração

A resolução de configurações default é bastante simples, já que se limita a obtenção das versões correspondentes aos rótulos de última liberada **Ult_Lib**, última efetiva **Ult_Efet** e corrente **Ult_Edi**, que podem estar relacionados aos objetos de uso geral, de projeto ou particulares. Como estes relacionamentos existem diretamente no esquema, basta navegar através do mesmo para obter a versão. Portanto, nos concentraremos na resolução de configurações de objetos. Na seqüência, adotaremos a seguinte nomenclatura [VMD89]:

definição de configuração: conjunto de entradas diretas, indiretas e de inclusão, apresentadas no capítulo 4 e através das quais determinamos as versões dos componentes dos objetos compostos.

visão de execução: é o resultado da resolução de uma definição de configuração para um objeto, isto é, é o conjunto das versões dos componentes determinadas através de uma definição de configuração.

No esquema apresentado no Apêndice D, a definição de configuração está representado pelo objeto **Configuração**, sendo que as visões de execução são as

próprias versões dos objetos compostos, já que uma vez congeladas, as versões dos componentes estão estaticamente ligadas. No caso das computações, os componentes são representados pelas instâncias, que, por sua vez, são geradas de versões específicas dos objetos componentes. Quanto aos programas Cm, as definições de configuração resolvem as versões dos objetos importados, "usados" e herdados por uma determinada classe Cm e que são representadas pelos relacionamentos **Import, Use** e **Inherit.**

O objeto **Configuração** possui quatro componentes: **Entr_Direta, Entr_Rotulo, Entr_Indir** e **Entr_Inclusao.** Para resolver a configuração, inicialmente é criada uma lista ligada contendo os identificadores do objeto genérico do componente, do banco de dados no qual ele está armazenado e campos inicializados com valores pré-definidos para os identificadores da nova versão, do seu objeto genérico e do banco de dados que os contém.

Entradas Diretas

Em primeiro lugar, são resolvidas as entradas diretas **Entr_Direta**, que são diretamente ligadas à versão do objeto que deve ser incluída, caso ele seja um componente do objeto composto. Portanto, para cada entrada direta, obtém-se o objeto genérico da versão, é verificado se ele pertence a lista de componentes e, em caso afirmativo, os campos referentes a nova versão são atualizados.

Entradas de Rótulo

Em seguida, resolvemos as entradas de rótulo **Entr_Rotulo.** Na realidade, elas são entradas diretas que, ao invés do identificar diretamente o número da versão, relacionam o objeto genérico a um rótulo. Identicamente às configurações default, estes rótulos podem ser última liberada, última efetiva e corrente. Então, se este objeto estiver contido na lista e não tiver sido resolvido ainda, a versão correspondente ao rótulo deve ser obtido e a lista atualizada.

Entradas Indiretas

Se ainda houver objetos cujas versões não foram resolvidas, passamos a tratar das entradas indiretas. Elas relacionam um determinado objeto genérico a uma configuração. A versão deste objeto que será utilizada é aquela resultante da configuração ao qual está relacionada. Portanto, se o objeto estiver presente na lista de componentes, criamos uma nova lista contendo apenas este objeto e chamamos recursivamente a função para resolução da configuração associada.

Entradas de Inclusão

A resolução das entradas de inclusão é basicamente constituída de operações de conjuntos. Ela é composta por uma configuração e um peso associado. Quanto maior o seu peso, maior a sua prioridade. Assim, em primeiro lugar, é necessário obter todas as entradas de inclusão e classificá-las em ordem descendente de peso. Para isso, existe uma função da DML Damokles chamada **db_ret** que possui uma linguagem de consulta embutida, tendo a facilidade de ordenar os objetos obtidos em ordem ascendente ou descendente de algum atributo. O resultado é um conjunto de objetos contidos em uma entidade chamada *cursor*.

O conceito de *cursor* do Damokles representa um conjunto de objetos e relacionamentos distintos e de tipos arbitrários. Uma de suas restrições é que todos os seus elementos devem pertencer a um mesmo banco de dados. Para todo cursor, o SGBD mantém um apontador para o seu elemento corrente. Isso possibilita que os programas tenham acesso seqüencial a cada elemento. Além destas operações sobre elementos, que incluem remoção, inclusão e recuperação de um elemento, existem as operações de conjunto como união, interseção e diferença. Devido a existência destas operações, considerou-se a sua utilização na implementação da resolução de configurações. Neste caso, os objetos seriam incluídos em um cursor à medida que suas versões fossem encontradas. Então, para resolver as entradas de inclusão, as configurações associadas seriam resolvidas em ordem descendente de peso, criando para cada uma um cursor. Na seqüência, chamaremos este cursor de cursor inclusão e aquele que contém os objetos cujas versões já foram identificadas de cursor atual. Assim, no final de cada configuração, faríamos a diferença do cursor inclusão com o cursor atual, criando um terceiro cursor. Este último conteria aqueles objetos que foram resolvidos pela configuração da entrada de inclusão e que ainda não haviam sido resolvidos. Portanto, bastaria fazer a união deste com o cursor atual para obtermos o novo cursor atual. Note que os cursores não contêm as versões, mas os seus objetos genéricos. Do contrário, se alguma configuração associada a uma entrada de inclusão identificasse uma versão diferente de um objeto já existente no cursor atual, ela também passaria a fazer parte deste cursor, o que seria um erro. Portanto, as versões teriam que ser armazenadas em uma outra estrutura, que poderia ser uma lista ligada ou um outro cursor, se isto fosse possível.

Como já foi dito anteriormente, um cursor comporta objetos de apenas um banco de dados. Assim, ele não poderia manter as versões, pois as versões liberadas e efetivas de um objeto ficam armazenadas no banco de dados central, enquanto as edições particulares ficam nos bancos de dados privados de cada usuário. Pela mesma razão, os cursores não foram utilizados na resolução de

configuração. Um detalhe não mencionado até o momento é que é possível um usuário criar um objeto em seu banco de dados particular sem que este exista previamente no banco de dados central. Um exemplo típico é um programa teste. Não faria sentido que criássemos qualquer referência a ele no banco de dados central até que fosse indicado que ele não é um programa temporário através de uma efetivação ou liberação. Assim, há a possibilidade de existirem objetos somente no banco de dados central, só no banco de dados particular e em ambos, no caso do objeto ter sido derivado. Portanto, neste caso também os cursores não poderiam ser utilizados, fazendo-nos retornar à utilização da lista ligada.

As entradas de inclusão são as últimas a serem resolvidas, após as entradas indiretas. Similarmente a elas, chamamos recursivamente, em ordem descendente de peso, a função de resolução de configurações, porém com a lista de todos os componentes. Já que antes de sua atualização é verificado se a versão do elemento já foi identificado e somente em caso negativo ele é atualizado, apenas as versões daqueles não identificados serão modificados, garantindo que não sejam sobrepostas. Além da lista de componentes é mantida também uma outra lista contendo a identificação de todas as configurações utilizadas pelas entradas de inclusão desde o início da cadeia. Na entrada da função, verifica-se se a configuração pertence a esta lista e, em caso afirmativo, a função é abandonada. Assim, evitamos a repetição da resolução de uma configuração, bem como que o programa entre numa repetição infinita. Isto ocorre se uma configuração fizer parte de uma entrada de inclusão de outra configuração e vice-versa.

Se depois de todas as entradas serem analisadas ainda restarem objetos com versões não resolvidas, passamos a utilizar a configuração default de mais baixo nível para identificá-las. De mais baixo nível chamamos a configuração pertencente ao usuário, projeto ou BD Central, nesta ordem de prioridade.

6.5 Considerações Gerais sobre o SGBD Damokles

Em linhas gerais, podemos dizer que o modelo de dados Damokles é bastante adequado, dando suporte a objetos compostos, o que constitui o principal requisito em se tratando de ambientes de projeto. Embora não implemente o conceito de generalização através da herança de propriedades e conceitualmente isto possa ser facilmente contornado, ele seria uma abstração poderosa a nível de projeto. Além disso, a maior parte das situações com as quais nos deparamos e que, como conseqüência, levou à revisão das opções de implementação foram causadas pela forma como se dá a distribuição dos dados no ambiente A_HAND. Se centralizássemos todas as informações no BD Central e utilizássemos nos bancos

de dados particulares apenas dados trazidos das operações de *checkout*, provavelmente muitas das dificuldades não existiriam. Porém, como o ambiente foi projetado para uma rede de estações de trabalho e, também, com o intuito de testar como o SGBD se adapta às necessidades do mesmo, a forma de distribuição foi mantida. Nesta seção, colocaremos de forma resumida as observações feitas ao longo das seções anteriores.

Em primeiro lugar, devemos ressaltar a dependência entre a criação do esquema e a facilidade de manipulação dos seus dados. Neste caso incluímos as restrições quanto a distribuição e as conseqüências sobre operações de cópia e transferência de objetos estruturados. Isso faz com que na criação do esquema o objetivo não seja apenas manter a correspondência com o mundo a ser modelado, mas onde algumas particularidades do SGBD devem ser pesadas como, por exemplo, as vantagens ou a possibilidade de se declarar um objeto ou relacionamento como componente de um outro objeto. Na realidade, por falta de uma documentação mais rica, estes detalhes aparecem somente na fase em que já estamos trabalhando com as funções, causando a reelaboração do esquema em algumas situações.

Dentro da funcionalidade oferecida pelo sistema, algumas extensões que facilitariam a implementação da aplicação são:

- o conceito de *cursor* poderia ser estendido para objetos pertencentes a diferentes bancos de dados;
- a restrição de que todo o grafo de versões deve estar contido em apenas um banco de dados poderia ser relaxado ao menos durante as transações longas. Assim, se for efetuado um *checkout* de uma versão, poderíamos criar novas versões do mesmo no banco de dados particular sem que este exista previamente no banco de dados original;
- poderia haver funções para manipulação do grafo de versões. Idealmente, elas deveriam existir somente dentro de transações para que no final possa ser verificado se suas restrições não foram violadas. Infelizmente, o conceito de transação ainda não é disponível no SGBD Damokles;
- funções como cópia de objetos, que envolvem objetos compostos e versões, deveriam possuir opções quanto à profundidade da operação. Isto é, a determinação se todos os componentes, objetos genéricos e versões devem ser copiados poderia ser uma decisão do usuário.

Além disso, o sistema deveria oferecer maior transparência quanto a distribuição dos relacionamentos. Isto é, se um objeto possui relacionamentos com objetos de outros bancos de dados, estes relacionamentos deveriam ser diretamente

visíveis a partir do banco de dados deste objeto, mesmo que o relacionamento propriamente dito esteja armazenado em outro. Certamente isto acarretaria uma sobrecarga na atualização destes relacionamentos. Há possibilidade de isto ser implementado pelo programa aplicativo. Porém, para que a implementação seja mais eficiente, utilizando identificadores de relacionamentos e bancos de dados já existentes internamente ao SGBD, seria necessário entrarmos em particularidades da sua própria implementação. Portanto, poderia haver a opção de se manter tal transparência, estando o usuário ciente de que isto diminuiria a eficiência das atualizações.

A versão 2.3 do SGBD Damokles que foi utilizada não possui linguagem de consulta e é monousuária. O código executável de um programa é obtido ligando o seu código objeto a biblioteca de funções do gerenciador. Conseqüentemente, todos os programas executáveis são muito grandes. Por exemplo, a implementação da função *load* de computação, descrita nas seções anteriores, resultou em um programa executável de aproximadamente 2 MBytes. Uma crítica a se fazer é que não podemos encarar o Damokles como um SGBD completo, já que não atende a requisitos como segurança e sincronização, e também não podemos vê-lo como uma ferramenta para construção de outras ferramentas, pois suas funções são de muito alto nível para tanto. Tendemos a considerá-lo como uma ferramenta, mas para que isso fosse totalmente verdadeiro, ele não deveria ter modelos pré-definidos para controle de acesso, mas funções de mais baixo nível, complementando as já existentes, para implementar mais facilmente, por exemplo, um *browser* para o SGBD.

De qualquer maneira, é possível imaginar o quanto um banco de dados com um modelo mais rico como o Damokles facilita a implementação de um sistema com dados tão inter-relacionados e complexos como um ambiente de desenvolvimento de software. Em um modelo relacional, por exemplo, o número de relações certamente seria tão grande, com os objetos tão fragmentados, que o código ficaria muito confuso, mesmo para aqueles que dominassem o problema, tornando os programas mais propensos ao erro.

Capítulo 7

Conclusão

A experiência em utilizar um banco de dados orientado a objetos para suporte a ambientes de desenvolvimento de software mostrou-se útil em vários aspectos.

7.1 Modelagem

Com relação ao modelo de dados Damokles, a decisão de baseá-lo no conceito de composição de objetos, essencial para os ambientes ao qual se destina, foi bastante acertada. Combinando-o à facilidade de se declarar alternativas de classe através da cláusula UNION, podemos indiretamente implementar o conceito de generalização. Por exemplo, o objeto ALUNO pode ser criado tendo como componente o objeto TIPO DE ALUNO, sendo este uma alternativa entre ALUNO DE GRADUAÇÃO e ALUNO DE PÓS GRADUAÇÃO. Porém, se analisarmos a questão do ponto de vista da integração de banco de dados com linguagens de programação, a inexistência deste conceito pode ser mais um fator na dificuldade de integração das duas tecnologias, o que é chamado de *impedance mismatch*. Além disso, as linguagens de programação orientadas a objetos implementam objetos compostos através da definição de atributos com o tipo de seus subobjetos. Isto é, a definição de uma classe efetivamente define um tipo que pode ser utilizado como um tipo primitivo da linguagem. O mesmo não ocorre com o Damokles, onde o tratamento entre tipos primitivos e declarados, bem como os relacionamentos existentes entre os objetos não é uniforme.

Na modelagem do ambiente A_HAND, a primeira versão do detalhamento das propriedades dos objetos seguia a linha das linguagens de programação e continha apenas atributos em sua definição, não importando se seu tipo era primitivo no banco de dados ou seria um objeto definido no esquema. O que pode ser

concluído é que, embora a composição de objetos e relacionamentos possam ser definidos desta forma, a versão final, que separa atributos de relacionamentos, representa melhor a semântica associada aos dados. Um modelo de dados que possibilite a integração do potencial semântico de BDOO com a funcionalidade de linguagens de programação orientadas a objetos, dando-lhes persistência de dados, deve ser um requisito para a nova geração de bancos de dados. Além disso, este modelo deve incorporar os conceitos de variáveis de classe e de instância, implementando de forma direta os níveis de objetos utilizados no detalhamento dos objetos na modelagem e tornando o relacionamento de instanciação uma primitiva do modelo.

7.2 Funcionalidade

O principal fator limitante do SGBD Damokles quanto a sua funcionalidade é que os esquemas só podem ser definidos estaticamente. Isto é, não há facilidades para definir novos tipos de dados ou que permitam a alteração do esquema original em tempo de execução. Na experiência com o ambiente A_HAND, limitamos a modelagem à funcionalidade exigida pelo editor de LegoShell e compilador Cm, pois eles comportam duas linguagens básicas no ambiente e eram as ferramentas melhor definidas na época do levantamento dos dados. Uma primeira versão do ambiente poderia ser criada contendo estas facilidades. Porém, muitas outras estão atualmente em desenvolvimento para serem posteriormente integradas. Para que o banco de dados acompanhe a evolução natural do ADS, sem que isto implique na incompatibilidade dos dados e programas em suas diferentes etapas de desenvolvimento, ele deve permitir a alteração do esquema dinamicamente. Além disso, esta facilidade seria essencial na criação de ambientes que geram ambientes, ou seja, ambientes configuráveis. Neste tipo de sistema, ambientes para suportar uma determinada metodologia de desenvolvimento e uma estrutura organizacional particular seriam criados especializando objetos de um modelo de dados genérico. Esta facilidade poderia ser implementada através da criação e alteração de tipos à medida que o usuário adapta o ambiente às suas necessidades.

Por outro lado, o Damokles incorpora conceitos importantes, tais como identidade forte de objetos e suporte a transações longas. A identidade, além de livrar o usuário da tarefa de criar ou apontar um ou mais atributos como a chave de um objeto, permite a existência de operações que não seriam diretamente implementadas em um modelo relacional, tais como cópias de objetos com níveis diferenciados de compartilhamento. A função de cópia do Damokles implementa

o conceito de cópia profunda, isto é, um novo objeto idêntico ao original é criado, bem como são geradas cópias de todos seus subobjetos. Se quiséssemos, no A_HAND, que as versões de computações compartilhassem os subobjetos criando novas instâncias somente quando alguma propriedade fosse alterada, seria interessante que houvesse também a disponibilidade de cópia rasa. Deste modo, na criação de uma nova versão, somente um objeto para a nova versão seria criado, compartilhando todos os subobjetos da versão original. Assim como no caso de subobjetos, a cópia de versões deve ter associado também mais um nível de especificação: se ela deve constituir um novo objeto independente das versões existentes, como é implementado no SGBD Damokles, ou se a versão deve ser "derivada", isto é, se a nova versão do objeto deve ser criada sucedendo a versão original.

Apesar deste banco de dados não implementar o conceito de transações convencionais, suas transações longas associadas às funções de *check out* e *check in* de objetos é muito interessante, pois podemos "emprestar" os objetos do banco de dados público para o particular à medida que eles são necessários e devolvê-los depois de atualizados dentro de uma mesma transação. É também possível iniciarmos uma transação dentro de outra transação longa. Esta facilidade só não foi utilizada na implementação do ambiente A_HAND pelas restrições quanto à distribuição dos dados do SGBD Damokles. Portanto, um outro requisito para estes sistemas é que haja suporte a bases distribuídas, porém, não devem existir restrições quanto aos tipos de dados armazenados em cada base. Por exemplo, o SGBD deve permitir que um objeto esteja em um banco de dados e seus subobjetos espalhados por outros, dando ao usuário maior transparência e flexibilidade quanto a sua distribuição.

Finalizando, o banco de dados deve implementar estas facilidades utilizando técnicas que resultem em um desempenho aceitável para ser utilizado em ambientes de desenvolvimento de software reais.

7.3 Aplicação

A literatura apresenta poucas experiências que fazem um levantamento exaustivo dos dados de um ADS. O trabalho mais significativo nesta área é o de Penedo [Pen86], que modela um ambiente fictício utilizando o modelo entidade-relacionamento e o implementa em um banco de dados relacional. Ao contrário, nossa proposta foi a modelagem de um ambiente real e sua implementação utilizando um banco de dados orientado a objetos. A modelagem destes dados foram importantes na comprovação de algumas características quanto a estru-

tura dos dados envolvidos em tal aplicação, como a sua complexidade e grande inter-dependência. Mesmo não realizando o mapeamento do modelo conceitual obtido para um modelo relacional, é possível imaginar que o número de relações ficaria tão grande que tornaria tanto a visão geral dos dados quanto os programas que os manipulam bastante confusos.

Além disso, o levantamento detalhado dos objetos e funções do ambiente contribui na formação de uma base sólida para críticas e melhoramentos de propostas de bancos de dados orientados a objetos para ambientes de projeto, muito falado na literatura, mas sem comprovação em situações reais.

Para o ambiente A_HAND, a contribuição maior consiste no levantamento dos dados necessários para atender parte de sua funcionalidade, juntamente com as funções necessárias sobre cada objeto. Ele já está sendo utilizado na implementação de uma estrutura em memória que será manipulada pelo editor de LegoShell. Como o modelo é independente do meio de armazenamento sobre o qual será implementado, as demais ferramentas podem ser construídas utilizando as operações definidas, abstraindo-se de detalhes quanto a persistência dos dados.

7.4 Trabalhos Futuros

A experiência de modelagem de um ADS real é importante para o amadurecimento das idéias quanto às dificuldades e necessidades envolvidas na manipulação dos seus dados. Porém, o modelo resultante está intimamente relacionado às particularidades, tais como a metodologia, as ferramentas e as linguagens utilizadas. Para atender um ambiente configurável, independente destas características, é necessário um modelo de dados genérico para ADS's. Este modelo criaria um novo nível intermediário entre um modelo como o DODM do Damokles, por exemplo, e o modelo de um ambiente específico como o que geramos para o A_HAND. Sobre este modelo intermediário, então, é que todas as informações do ambiente seriam definidas. Ele faria parte da base de uma arquitetura genérica de ambientes, podendo comportar dados sobre o processo de desenvolvimento, ferramentas, pessoal, produtos e a própria arquitetura do ambiente através de informações fornecidas pelo próprio usuário. Isto faria com que alterações sobre estas definições tornem-se mais fáceis, gerando um ambiente realmente extensível e adaptado às necessidades de cada organização. Uma idéia interessante seria que esta extensibilidade (claramente com conseqüências sobre o desempenho do ambiente) pudesse existir durante um período definido pelo próprio usuário. Por exemplo, se uma definição mostrou-se satisfatória durante um tempo razoável e o usuário concluir que não deseja mais fazer alterações sobre o mesmo, poderia

haver facilidades para mapear o modelo intermediário para aquele mais próximo à implementação, a fim de melhorar o desempenho geral do ambiente. A obtenção deste modelo genérico é um importante passo na área de modelagem de ADS.

Apêndice A

Detalhamento das Propriedades dos Objetos

```
OBJETO Computacao
    CLASSE
        Componentes:
        Relacionamentos:
            . objeto de: Configuracao de Objeto (CLASSE)
            . propriedade de: Usuario / Projeto / BD Central
            . representado por: Icone
            . relacionado a: Entrada de Rotulo
            . relacionado a: Entrada Indireta
        Atributos:
            . nome
            . ultima versao liberada
            . ultima versao efetiva
            . ultima edicao
            . documentacao
            . classificacao (uso geral, de projeto, particular)

    VERSAO  /* diferentes implementacoes da classe computacao */
        Componentes:
            . Objetos Componentes:
                . Computacao (INSTANCIA)
                . Programa Cm (INSTANCIA)
                . Arquivo (INSTANCIA)
```

```
                . Periferico (INSTANCIA)
                . Estrela (INSTANCIA)
                . Mailbox (INSTANCIA)
                . Pipe (INSTANCIA)
            . Porta Virtual
            . Parametro
        Relacionamentos:
            . configurado por: Configuracao Default (VERSAO) /
                               Configuracao de Objeto (VERSAO)
            . editado por: Usuario
            . propriedade de: Usuario
            . derivado de / derivado por: Computacao (VERSAO)
            . relacionado a: Entrada Direta
        Atributos:
            . numero da versao
            . tipo de versao (edicao / efetiva / liberada)
            . data de criacao
            . data de efetivacao
            . data de liberacao
            . versao anterior
            . versao posterior
            . descricao da alteracao
            . estado da computacao
              (inconsistente / consistente / compilado)
            . codigo de execucao / interpretacao
            . numero do ultimo componente

    INSTANCIA  /* um componente de uma computacao */
        Instancia de: Computacao (VERSAO)
        Componentes:
            . Porta Virtual
        Relacionamentos:
            . define valor de: Parametro
        Atributos:
            . numero do componente dentro da computacao
            . posicao
            . ligacao das portas
FIM Computacao
```

```
OBJETO Programa Cm
    CLASSE
        Componentes:
        Relacionamentos:
            . objeto de: Configuracao de Objeto (CLASSE)
            . propriedade de: Usuario / Projeto / BD Central
            . representado por: Icone
            . relacionado a: Entrada Rotulo
            . relacionado a: Entrada Indireta
        Atributos:
            . nome
            . ultima versao liberada
            . ultima versao efetiva
            . ultima edicao
            . documentacao
            . classificacao (uso geral, projeto, particular)

    VERSAO /* diferentes implementacoes de programa Cm */
        Componentes:
            . Porta Real
            . Parametro
            . Programas Cm "usados" (VERSAO)
            . Programas Cm "importados" (VERSAO)
            . Programas Cm "herdados" (VERSAO)
            . "rename" de funcao
            . declaracao de constantes
            . declaracao de tipos
            . declaracao de variaveis de classe
            . declaracao de variaveis de instancia
            . cabecalho das funcoes exportaveis
            . cabecalho das funcoes privadas
            . fonte
        Relacionamentos:
            . configurado por: Configuracao Default (VERSAO) /
                               Configuracao de Objeto (VERSAO)
            . editado por: Usuario
            . propriedade de: Usuario
            . associado como importador a: importacao
```

```
            . associado como exportador a: importacao
            . associado como herdeiro a: heranca
            . associado como herdado a: heranca
            . associado como o que "usa": "uso"
            . associado como "e usado" a: "uso"
            . derivado de: Programa Cm (VERSAO)
            . derivado por: Programa Cm (VERSAO)
            . traduzido para: Programa C
            . relacionado a: Entrada Direta
        Atributos:
            . numero da versao
            . tipo de versao (edicao / efetiva / liberada)
            . data de criacao
            . data de efetivacao
            . data de liberacao
            . versao anterior
            . versao posterior
            . descricao da alteracao
            . estado do programa (consistente / inconsistente)

    INSTANCIA  /* componente de computacao */
        Instancia de: Programa Cm (VERSAO)
        Componentes:
            . Porta Virtual
        Relacionamentos:
            . implementado por: Programa C
            . define valor de: Parametro
        Atributos:
            . numero do componente dentro da computacao
            . posicao
            . ligacao das portas
FIM Programa Cm


OBJETO Programa C
    CLASSE
        Componentes:
        Relacionamentos:
            . traducao de: Programa Cm (VERSAO)
```

```
            . implementacao de: Programa Cm (INSTANCIA)
            . define valor de: Parametro
        Atributos:
            . data da traducao
            . fonte C
            . codigo objeto
            . codigo executavel
FIM Programa C


OBJETO Arquivo
    CLASSE
        Componentes:
        Relacionamentos:
            . propriedade de: Usuario / Projeto / BD Central
            . representado por: Icone
        Atributos:
            . nome
            . diretorio
            . classificacao
            . acesso de leitura/escrita para proprietario, projeto e outros

    VERSAO  /* backup de arquivo */
        Componentes:
        Relacionamentos:
        Atributos:
            . data do backup
            . estado (arquivado / on line)
            . localizacao - se arquivado: meio fisico / identificador
                            se on line: path do diretorio

    INSTANCIA  /* componente de computacao */
        Instancia de: Arquivo (CLASSE)
        Componentes:
            . Porta Virtual
        Relacionamentos:
        Atributos:
            . numero do componente dentro da computacao
            . posicao
```

```
            . ligacao das portas
FIM Arquivo


OBJETO Periferico
    CLASSE
        Componentes:
            . Parametro
        Relacionamentos:
            . representado por: Icone
        Atributos:
            . tipo de periferico (impressora, terminal, ...)
            . tipo de operacao (E / S)

    VERSAO  /* unidades do tipo de periferico da classe */
        Componentes:
        Relacionamentos:
            . propriedade de: Usuario / Projeto / BD Central
        Atributos:
            . identificacao da unidade
            . localizacao (local / remoto / rede)
            . endereco
            . especificacao (modelo, velocidade, processador, ...)

    INSTANCIA  /* componente de uma computacao */
        Instancia de: Periferico (VERSAO)
        Componentes:
            . Portas Virtuais
        Relacionamentos:
            . define valor de: Parametro
        Atributos:
            . numero do componente dentro da computacao
            . posicao
            . ligacao das portas
FIM Periferico


OBJETO Estrela
    CLASSE
```

```
        Componentes:
            . Parametro
        Relacionamentos:
        Atributos:

    INSTANCIA  /* componente de uma computacao */
        Instancia de: Estrela (CLASSE)
        Componentes:
            . Portas Virtuais
        Relacionamentos:
            . define valor de: Parametro
        Atributos:
            . numero do componente dentro da computacao
            . posicao
            . ligacao das portas
FIM Estrela


OBJETO Mailbox
    CLASSE
        Componentes:
            . Parametro
        Relacionamentos:
        Atributos:

    INSTANCIA  /* componente de uma computacao */
        Instancia de: Mailbox (CLASSE)
        Componentes:
            . Portas Virtuais
        Relacionamentos:
            . define valor de: Parametro
        Atributos:
            . numero do componente dentro da computacao
            . posicao
            . ligacao das portas
FIM Mailbox


OBJETO Pipe
```

```
    CLASSE
        Componentes:
            . Parametro
        Relacionamentos:
        Atributos:

    INSTANCIA  /* componente de uma computacao */
        Instancia de: Pipe (CLASSE)
        Componentes:
        Relacionamentos:
            . define valor de: Parametro
            . tem como entrada: Porta Virtual
            . tem como saida: Porta Virtual
        Atributos:
            . numero do componente dentro da computacao
FIM Pipe


OBJETO Porta Real
    CLASSE
        Componentes:
        Relacionamentos:
            . e abstraido por: Porta Virtual
            . e representado por: Porta Virtual
        Atributos:
            . nome
            . tipo de operacao (entrada / saida / entrada_saida)
            . tipo de transmissao (byte / block / VSblock)
            . tamanho de bloco
FIM Porta Real


OBJETO Porta Virtual
    CLASSE
        Componentes:
        Relacionamentos:
            . e abstraido por: Porta Virtual
            . e abstracao de: Porta Virtual / Porta Real
            . representa: Porta Real / Porta Virtual
```

```
            . e representado por: Porta Virtual
            . e entrada de / e saida de: Pipe
        Atributos:
            . tipo de operacao (entrada / saida / entrada_saida)
FIM Porta Virtual


OBJETO Parametro
    CLASSE
        Componentes:
        Relacionamentos:
            . e abstracao de: Parametro
            . e abstraido por: Parametro
            . valor definido por: Computacao (INSTANCIA) /
                                  Programa Cm (INSTANCIA) /
                                  Programa C (CLASSE) /
                                  Periferico (INSTANCIA) /
                                  Estrela (INSTANCIA) /
                                  Mailbox (INSTANCIA) /
                                  Pipe (INSTANCIA)
        Atributos:
            . nome
            . tipo (real / chave de selecao / param. de execucao)
            . obrigatoriedade (sim / nao)
            . tipo de dado
            . valor default
            . valor real
FIM Parametro


OBJETO Configuracao de Objeto
    CLASSE
        Componentes:
        Relacionamentos:
            . definido para: Computacao (CLASSE) /
                             Programa Cm (CLASSE)
        Atributos:
            . numero da ultima configuracao
```

```
    VERSAO  /* uma definicao de configuracao */
        Componentes:
            . Entrada Direta
            . Entrada de Rotulo
            . Entrada Indireta
            . Entrada de Inclusao
        Relacionamentos:
            . propriedade de: Usuario / Projeto / BD Central
            . determina versoes de: Computacao (VERSAO) /
                                    Programa Cm (VERSAO)
            . incluido por: Entrada de Inclusao
            . relacionado a: Entrada Indireta
        Atributos:
            . numero da configuracao
            . data de criacao
FIM Configuracao de Objeto


OBJETO Entrada Direta
    CLASSE
        Componentes:
        Relacionamentos:
            . relaciona: Computacao (VERSAO) /
                         Programa Cm (VERSAO)
        Atributos:
FIM Entrada Direta


OBJETO Entrada de Rotulo
    CLASSE
        Componentes:
        Relacionamentos:
            . relaciona: Computacao (CLASSE) /
                         Programa Cm (CLASSE)
        Atributos:
            . rotulo (UltLib / UltEfet / Corrente)
FIM Entrada de Rotulo
```

```
OBJETO Entrada Indireta
    CLASSE
        Componentes:
        Relacionamentos:
            . relaciona:  Computacao (CLASSE) /
                          Programa Cm (CLASSE)
            . definido atraves de: Configuracao de Objeto (VERSAO)
        Atributos:
FIM Entrada Indireta


OBJETO Entrada de Inclusao
    CLASSE
        Componentes:
        Relacionamentos:
            . inclui: Configuracao de Objeto (VERSAO)
        Atributos:
            . peso
FIM Entrada de Inclusao


OBJETO Configuracao Default
    CLASSE
        Componentes:
        Relacionamentos:
            . propriedade de: Usuario / Projeto / BD Central
        Atributos:
            . numero da configuracao default ativa

    VERSAO  /* definicao de uma configuracao default */
        Componentes:
        Relacionamentos:
            . determina versoes de: Computacao (VERSAO) /
                                    Programa Cm (VERSAO)
        Atributos:
            . numero da configuracao
            . data de criacao
            . rotulo para objetos particulares (UltLib, UltEfet, Corrente)
            . rotulo para objetos de projeto (UltLib, UltEfet, Corrente)
```

```
            . rotulo para outros objetos (UltLib, UltEfet, Corrente)
FIM Configuracao Default


OBJETO BD Central
    CLASSE
        Componentes:
            . Projeto
        Relacionamentos:
            . proprietario de: Computacao (CLASSE) /
                               Programa Cm (CLASSE) /
                               Arquivo (CLASSE) /
                               Periferico (VERSAO) /
                               Configuracao de Objeto (VERSAO) /
                               Configuracao Default (CLASSE)
            . coordenado por: Usuario
            . define: Parametros (de execucao)
            . utiliza: Icone
        Atributos:
            . numero de edicoes guardadas
            . tamanho de buffer default para: pipe
                                              estrela
                                              mailbox
FIM BD Central


OBJETO Projeto
    CLASSE
        Componentes:
            . Usuario
        Relacionamentos:
            . proprietario de: Computacao (CLASSE) /
                               Programa Cm (CLASSE) /
                               Arquivo (CLASSE) /
                               Periferico (VERSAO) /
                               Configuracao de Objeto (VERSAO) /
                               Configuracao Default (CLASSE)
            . coordenado por: Usuario
            . utiliza: Icone
```

```
        Atributos:
            . nome
            . data de inicio
            . data final
            . numero de edicoes guardadas
            . tamanho de buffer default para: pipe
                                              estrela
                                              mailbox
FIM Projeto


OBJETO Usuario
    CLASSE
        Componentes:
        Relacionamentos:
            . proprietario de: Computacao (CLASSE) /
                               Computacao (VERSAO) /
                               Programa Cm (CLASSE) /
                               Programa Cm (VERSAO) /
                               Arquivo (CLASSE) /
                               Periferico (VERSAO) /
                               Configuracao de Objeto (VERSAO) /
                               Configuracao Default (CLASSE)
            . coordena: Projeto / BD Central
            . edita: Computacao (VERSAO) / Programa Cm (VERSAO)
            . utiliza: Icone
        Atributos:
            . nome
            . data de inicio
            . data final
            . numero de edicoes guardadas
            . tamanho de buffer default para: pipe /
                                              estrela /
                                              mailbox
FIM Usuario


OBJETO Icone
    CLASSE
```

```
        Componentes:
        Relacionamentos:
            . representa: Computacao (CLASSE) /
                          Programa Cm (CLASSE) /
                          Arquivo (CLASSE) /
                          Periferico (CLASSE)
            . utilizado por: BD Central / Projeto / Usuario
        Atributos:
            . representacao grafica
FIM Icone
```

Apêndice B

Detalhamento das Funções dos Objetos

OBJETO Computação

- **Classe**
  - cria / remove classe de computação
  - obtém configuração de objeto associado à computação
  - obtém / altera proprietário da computação
  - obtém ícone associado à computação utilizado por um usuário
  - associa / desassocia ícone à computação
  - obtém computação pelo nome
  - seleciona as computações particulares, de projeto e de uso geral de acordo com a classificação
  - obtém direitos de um usuário sobre uma computação (Derivação, Leitura, Execução)
  - verifica se é relacionado a uma entrada de rótulo
  - verifica se é relacionado a uma entrada indireta
  - obtém quantidade de versões
  - obtém a última versão
  - associa / desassocia uma computação do BD particular como derivado de uma computação do BD Central
  - verifica se uma computação é derivação de outra no BD Central

- **Versão**
  - cria / remove versão de computação
  - obtém os objetos componentes de uma versão de computação
  - inclui / exclui um objeto componente de uma versão de computação
  - obtém as portas de uma versão de computação
  - inclui / exclui uma porta de uma versão de computação
  - compara portas de duas versões de computação
  - obtém os parâmetros de uma versão de computação
  - inclui / exclui um parâmetro de uma versão de computação
  - compara parâmetros de duas versões de computação
  - obtém a configuração que determina as versões dos objetos componentes
  - associa / desassocia a uma configuração que determina as versões dos objetos componentes
  - obtém proprietário da versão de computação
  - associa / desassocia proprietário de uma versão de computação
  - copia componentes de uma versão de computação para outra
  - copia versão de computação
  - inclui / exclui versão de computação da lista de sessões de um usuário com um determinado direito de acesso
  - verifica se uma versão de computação está sendo editada por um usuário
  - verifica se pode ser removida
  - verifica se existem instâncias de uma versão de computação
  - verifica se é relacionado a uma entrada direta
- **Instância**
  - cria / remove instância de computação
  - obtém versão de computação da qual foi instanciada
  - obtém nome da classe da qual foi gerada
  - obtém portas de uma instância de computação
  - inclui / exclui portas a uma instância de computação

- define / altera valor de parâmetro da versão de computação da qual foi instanciada
- copia / transfere uma instância de uma computação para outra
- verifica se todos os parâmetros possuem valores definidos
- verifica se todas as portas estão conectadas
- obtém o tipo de entrada da configuração que determina a versão de uma instância de computação (Direta, UltLib, UltEfet, Corrente, Indireta, Inclusão)

OBJETO Programa Cm

- **Classe**
  - cria / remove Programa Cm
  - obtém Configuração de Objeto associado ao Programa Cm
  - obtém / altera proprietário do Programa Cm
  - obtém Ícone associado utilizado por um determinado Usuário
  - associa / desassocia Ícone ao Programa Cm
  - obtém Programa Cm pelo nome
  - seleciona os programas particulares, de projeto e de uso geral de acordo com a classificação
  - verifica se é relacionado a uma entrada de rótulo
  - verifica se é relacionado a uma entrada indireta
  - obtém quantidade de versões
  - associa / desassocia um Programa Cm do BD particular como derivado de um Programa Cm do BD Central
  - verifica se um Progrma Cm é derivação de outro no BD Central
  - obtém direitos de acesso de um usuário sobre um Programa Cm (derivação, leitura, execução)
- **Versão**
  - cria / remove versão de Programa Cm
  - obtém a última versão de Programa Cm
  - inclui / exclui uma porta do Programa Cm

- compara as portas entre duas versões de Progrma Cm
- obtém os parâmetros de uma versão de Programa Cm
- inclui / exclui um parâmetro de uma versão de Programa Cm
- compara os parâmetros de duas versões de Programas Cm
- obtém os Programas Cm "usados"
- inclui / exclui um Programa Cm como "usado"
- obtém os Programas Cm "importados"
- inclui / exclui um Programa Cm como "importado"
- obtém os Programas Cm "herdados"
- inclui / exclui um Programa Cm como "herdado"
- obtém / altera outros componentes (rename, constantes, tipos, variáveis de classe, variáveis de instância, cabeçalho das funções exportáveis, cabeçalho das funções privadas, fonte)
- obtém / associa / desassocia à configuração que determina as versões dos programas "usados", "importados" e "herdados"
- verifica se é relacionado a uma entrada direta
- inclui / exclui uma versão de Programa Cm da lista de sessões de um usuário
- verifica se uma versão de Programa Cm está sendo editada por um usuário
- obtém / associa proprietário à uma versão de Programa Cm
- copia os componentes de outra versão de Programa Cm
- associa um Programa C como tradução de uma versão de Programa Cm
- verifica se existe tradução de uma versão de Programa Cm para um determinado conjunto de parâmetros
- verifica se existem instâncias de uma versão de Programa Cm
- copia versão de Programa Cm

• **Instância**

- cria / remove instância de Programa Cm
- obtém nome da classe da qual foi gerada
- obtém versão da qual é instância

- inclui / exclui portas virtuais
- define / altera valor dos parâmetros da versão da qual foi instanciada
- obtém Programa C que a implementa
- associa / desassocia a um Programa C que a implementa
- copia instância de Programa Cm
- transfere instância de Programa Cm de uma computação para outra
- verifica se todos os parâmetros estão definidos
- verifica se todas as portas estão conectadas
- obtém o tipo de entrada da configuração que determina a versão da instância (Direta, UltLib, UltEfet, Corrente, Indireta, Inclusão)

OBJETO Programa C

- **Classe**
  - cria / remove Programa C
  - associa / desassocia Programa C como tradução de uma determinada versão de Programa Cm
  - associa / desassocia Programa C como implementação de uma instância de Programa Cm
  - define valor dos parâmetros da versão de Programa Cm para os quais foi traduzido
  - verifica se existe Programa C para uma determinada versão e conjunto de valores de parâmetros de Programa Cm
  - exclui todos os Programas C que são traduções de uma determinada versão de Programa Cm

OBJETO Arquivo

- **Classe**
  - cria / remove arquivo
  - obtém / altera proprietário de um arquivo
  - obtém ícone associado ao arquivo utilizado por um usuário
  - associa / desassocia um ícone a um arquivo

- obtém um arquivo pelo nome
- selecionar os arquivos particulares, de projeto, e de uso geral de acordo com a classificação
- verifica se exitem instâncias da classe

• **Versão**

- cria / remove backup de um arquivo
- gerar um arquivo (classe) copiado de um backup

• **Instância**

- cria / remove instância de arquivo
- obtém portas virtuais de uma instância de arquivo
- inclui / exclui uma porta virtual de uma instância de arquivo
- copia uma instância de arquivo
- transfere uma instância de arquivo de uma computação para outra
- obtém arquivo do qual foi instanciado

OBJETO Periférico

• **Classe**

- cria / remove periférico
- obtém parâmetros de um periférico
- inclui / exclui um parâmetro de um periférico
- obtém o ícone associado a um periférico utilizado por um usuário
- associa / desassocia um ícone a um periférico
- obtém todas as unidades de um determinado tipo de periférico

• **Versão**

- cria / remove uma unidade de periférico
- obtém / altera proprietário de uma unidade de periférico
- obtém uma unidade de periférico pela identificação externa
- seleciona todas as unidades locais, remotas ou em rede
- verifica se existem instâncias de uma unidade de periférico

- **Instância**
  - cria / remove uma instância de periférico
  - obtém as portas virtuais de uma instância de periférico
  - inclui / exclui uma porta virtual de uma instância de periférico
  - copia uma instância de periférico
  - transfere uma instância de periférico de uma computação para outra
  - define / altera valor de um parâmetro do tipo de periférico do qual foi instanciado
  - verifica se todos os parâmetros estão definidos
  - obtém versão da qual foi instanciada

OBJETO Estrela

- **Classe**
  - obtém parâmetros do conector estrela
  - inclui / exclui um parâmetro do conector estrela
- **Instância**
  - cria / remove uma instância de estrela
  - obtém as portas virtuais de uma instância de estrela
  - inclui / exclui uma porta virtual de uma instância de estrela
  - define / altera valor de um parâmetro do conector estrela
  - verifica se todos os parâmetros estão definidos
  - copia uma instância de estrela

OBJETO Mailbox

- **Classe**
  - obtém parâmetros do conector mailbox
  - inclui / exclui um parâmetro do conector mailbox
- **Instância**
  - cria / remove uma instância de mailbox

- obtém portas virtuais da instância de mailbox
- inclui / exclui uma porta virtual de uma instância de mailbox
- define / altera o valor de um parâmetro do conector mailbox
- verifica se todos os parâmetros estão definidos
- copia uma instância de mailbox

OBJETO Pipe

- **Classe**
  - obtém parâmetros do conector pipe
  - inclui / exclui um parâmetro do conector pipe
- **Instância**
  - cria / remove uma instância de pipe
  - define / altera valor de um parâmetro do conector pipe
  - obtém / altera porta de entrada associada ao pipe
  - obtém / altera porta de saída associada ao pipe
  - obtém todos os pipes ligados a uma determinada instância
  - copia uma instância de pipe

OBJETO Porta Real

- **Classe**
  - cria / remove uma porta real
  - cria / remove abstração de porta
  - cria / remove representação de porta
  - remove todas as portas virtuais que representam uma porta

OBJETO Porta Virtual

- **Classe**
  - cria / remove porta virtual
  - cria / remove abstração de porta

- cria / remove representação de porta
- obtém a porta real representada por uma porta virtual
- remove todas as portas virtuais que representam uma porta
- associa como entrada de um pipe
- associa como saída de um pipe
- desassocia de um pipe
- obtém a instância que possui a porta
- associa / desassocia buraco negro à porta
- exclui todas as portas virtuais de uma instância

OBJETO Parâmetro

- **Classe**
  - cria / remove um parâmetro
  - cria / remove uma abstração de parâmetro
  - obtém parâmetros de execução
  - define parâmetros de execução
  - obtém / altera valor do parâmetro e propaga para as suas abstrações
  - obtém objeto que possui o parâmetro

OBJETO Configuração de Objeto

- **Classe**
  - cria / remove uma configuração de objeto
  - associa / desassocia uma configuração a um Programa Cm ou computação
  - verifica se existe alguma definição de configuração
- **Versão**
  - cria / remove uma definição de configuração
  - exclui todas as definições de uma determinada classe
  - obtém uma definição de configuração pelo seu número
  - obtém o proprietário de uma definição de configuração

- seleciona as versões de programa Cm e computação de cujos componentes define a versão
- verifica se é incluído por uma entrada de inclusão
- verifica se é relacionado por alguma entrada indireta
- obtém as entradas da configuração
- inclui / exclui uma entrada de configuração
- dado um objeto, obter sua versão segundo uma definição de configuração
- resolve uma configuração para um objeto composto
- copia uma definição de configuração
- associa / desassocia a um objeto cujas versões dos componentes ela determina

OBJETO Entrada Direta

- **Classe**
  - cria / remove uma entrada direta
  - associa / desassocia a uma versão de computação ou programa Cm

OBJETO Entrada de Rótulo

- **Classe**
  - cria / remove uma entrada de rótulo
  - associa / desassocia a uma classe de computação ou programa Cm

OBJETO Entrada Indireta

- **Classe**
  - cria / remove uma entrada indireta
  - associa / desassocia a uma classe de computação ou programa Cm
  - associa / desassocia a uma definição de configuração

OBJETO Entrada de Inclusão

- **Classe**
  - cria / remove uma entrada de inclusão
  - associa / desassocia a uma definição de configuração
  - obtém todas as entradas de inclusão de uma definição de configuração

OBJETO Configuração Default

- **Classe**
  - cria uma configuração default relacionado a um usuário, projeto ou BD Central
  - remove uma configuração default
  - obtém definição de configuração default ativa
- **Versão**
  - cria / remove definição de configuração default
  - verifica se existe objetos cujas versões dos componentes é determinado pela definição de configuração
  - dado um objeto, obter sua versão segundo uma definição de configuração default
  - resolve uma definição de configuração default para um objeto composto
  - associa / desassocia a um objeto cujas versões dos componentes determina

OBJETO BD Central

- **Classe**
  - obtém coordenador central
  - associa / altera coordenador central
  - obtém ícones que utiliza
  - associa / desassocia ícones
  - obtém / altera definição de configuração default ativa para o BD Central

OBJETO Projeto

- **Classe**
  - cria / remove projeto
  - obtém usuários componentes do projeto
  - associa / desassocia um usuário a um projeto
  - obtém projeto pelo nome
  - obtém coordenador de um projeto
  - associa / altera coordenador de um projeto
  - obtém / altera definição de configuração default ativa para um projeto
  - obtém ícones utilizados pelo projeto
  - associa / desassocia ícones a um projeto

OBJETO Usuário

- **Classe**
  - cria / remove um usuário
  - obtém projetos dos quais participa
  - associa / desassocia a um projeto
  - obtém projetos que coordena
  - associa / desassocia a um projeto como coordenador
  - obtém / altera definição da configuração default ativa para um usuário
  - obtém objetos sendo editados
  - verifica se um determinado objeto está sendo editado
  - inclui / exclui um objeto da lista de edições de um usuário
  - obtém o último objeto da lista de sessões de um usuário
  - obtém ícones utilizados por um usuário
  - associa / desassocia ícones a um usuário
  - obtém o número de edições a serem guardadas por um usuário

OBJETO Ícone

- **Classe**

- cria / remove um ícone
- obtém o objeto que um ícone representa
- associa / desassocia ao objeto que representa
- obtém os usuários que o utilizam
- obtém os ícones utilizados por um usuário
- associa / desassocia um ícone a um usuário, projeto ou BD Central

Apêndice C

Detalhamento das Funções de Interface

C.1 Help

Como a definição exata da interface desta função não está totalmente terminada, optou-se por não considerá-la na modelagem, sendo as mensagens armazenadas em arquivos comuns, manipulados diretamente pelo editor de LegoShell.

C.2 Edição de Computação ou Programa Cm

Load (usuário, nome do objeto)

```
obtém objeto pelo nome no BD particular do usuário
SE achou
    SE objeto.última edição > 0
        SE objeto.última edição está sendo editado pelo usuário
            inclui a edição na lista de sessões do usuário com direito de
                acesso READ ONLY
            obtém os objetos componentes da edição (instâncias)
        SENÃO /* não está sendo editado */
            obtém número de edições do objeto
            obtém o número de edições a serem guardadas pelo usuário
            SE número de edições > limite estabelecido
                SE a primeira edição pode ser removida
                    remove a primeira edição
```

```
            objeto.última edição = objeto.última edição + 1
            cria nova versão do objeto do tipo edição
            cria componentes iguais à edição anterior
            associa à configuração igual a edição anterior
            resolve a configuração para o objeto para avisar se algum
                componente mudou de versão
            associa o usuário como proprietário da nova edição
            inclui a edição na lista de sessões do usuário com direito
                de acesso READ_WRITE
        SENÃO /* se ainda não existe edição */
            SE o objeto é derivação de um outro no BD Central
                obtém quantidade de versões do objeto no BD Central
                SE quantidade > 0
                    copia última versão do BD Central
                SENÃO /* objeto do BD Central não possui versão */
                    cria a primeira edição no BD particular
            SENÃO /* se o objeto no BD particular não é derivação */
                cria a primeira edição no BD particular
    SENÃO /* se o objeto não existe no BD particular */
        obtém objeto pelo nome no BD Central
        SE achou
            obtém direitos de acesso do usuário sobre o objeto
            SE direito = DERIVAÇÃO
                cria objeto (classe) no BD particular
                associa o novo objeto (classe) como derivação do objeto no BD
                    Central
                obtém quantidade de versões do objeto no BD Central
                SE quantidade > 0
                    copia versão do BD Central
                SENÃO
                    cria a primeira edição no BD particular
            SENÃO /* se direito = LEITURA */
                obtém a última versão do objeto no BD Central
                inclui a versão na lista de sessões do usuário com direito de
                    acesso READ_ONLY
                obtém os objetos componentes da versão
        SENÃO /* se não existe nem no BD particular nem no Central */
```

cria objeto (classe) no BD particular
cria a primeira edição no BD particular

Copia versão do BD Central
objeto.última edição = 1
obtém a última versão do objeto no BD Central
cria nova versão do objeto do tipo edição no BD particular
cria componentes iguais à última versão do BD Central
associa à configuração igual à última versão do BD Central
resolve a configuração para o objeto para avisar o usuário se mudou alguma versão de componente
associa o usuário como proprietário da edição
inclui a edição na lista de sessões do usuário com direito READ_WRITE

Cria a primeira edição no BD particular
objeto.última edição = 1
cria versão do tipo edição no BD particular
obtém configuração default para o usuário
associa esta configuração à edição
associa o usuário como proprietário da nova edição
coloca a edição na lista de sessões do usuário com direito READ_WRITE

C.3 Inclusão de componentes na computação

Computação ou Programa Cm (versão da computação, id. do objeto componente, posição)
número do componente = computação.último componente + 1
obtem configuração da computação
dado o componente, obter sua versão segundo a configuração da computação
cria uma instância da versão do objeto componente
inclui a instância como componente da versão de computação

Arquivo (usuário, versão da computação, id. do arquivo, posição)
número do componente = computação.último componente + 1
cria instância do arquivo
obtém direitos de acesso do usuário
/* para determinar o tipo das portas virtuais a serem criadas */
inclui a instância como componente da versão de computação

Periférico (versão da computação, id. do periférico, posição)
número do componente = computação.último componente + 1
cria instância do periférico
/* o tipo de operação do periférico define o tipo das portas */
inclui a instância como componente da versão de computação

Estrela ou Mailbox (versão da computação, posição)
número do componente = computação.último componente + 1
cria instância do conector
inclui a instância como componente da versão de computação

Pipe (versão da computação, porta de entrada, porta de saída)
número do componente = computação.último componente + 1
cria instância de pipe com as portas de entrada e saída
inclui a instância como componente da versão de computação

Buraco Negro (versão da computação, id. da porta)
associa um buraco negro à porta

C.4 Alteração geométrica de componentes

Move (versão de computação, instância componente, nova posição)

componente.posição = nova posição

C.5 Exclusão de componentes

Exclude (versão de computação, id. da instância componente)

Computação, Programa Cm, Arquivo, Periférico, Estrela, Mailbox
exclui a instância como componente da computação
exclui todas os pipes associados à instância
exclui a instância

Pipe, Buraco Negro
exclui a instância como componente da computação
exclui a instância

C.6 Edição de objeto componente

Edit de componente (usuário, versão de computação, instância componente)

SE tipo de entrada que determina versão da instância componenete = CORRENTE
    obtém nome da classe da qual a instância foi gerada
    LOAD (usuário, nome da classe)
    exclui a instância componente
    cria uma instância da nova edição gerada
    inclui a instância como componente da computação
SENÃO /* só é permitida leitura */
    obtém a versão da qual a instância foi gerada
    inclui a versão na lista de sessões do usuário com direito de acesso
        READ_ONLY
    obtém os objetos componentes da versão

C.7 Edição de configuração

Edit de configuração (versão de computação)

obtém a configuração que determina as versões dos componentes da computação
PARA CADA componente da versão de computação do tipo computação ou programa Cm
    obtém número da versão da qual a instância foi gerada
    obtém o tipo de entrada da configuração que determina sua versão
SE tipo da configuração = configuração de objeto
    obtém todas as entradas de inclusão da configuração
cria uma nova definição de configuração para a computação
/* durante a edição */
cria o tipo de entrada definido
dado o objeto, obtém sua versão segundo a entrada definida
/* terminada a edição */
PARA CADA componente do tipo computação ou programa Cm que mudou de versão
    cria uma instância da nova versão
    inclui a nova instância como componente da computação
    exclui o componente original
associa a nova configuração como a que determina as versões dos componentes
    da computação

C.8 Edição dos parâmetros dos componentes

Edit dos parâmetros (versão de computação)

PARA CADA componente da versão de computação
    obtém o objeto do qual foi instanciado
    obtém os parâmetros deste objeto
    SE componente for computação ou programa Cm
        obtém os parâmetros de execução
    obtém valor dos parâmetros definidos pelo componente
/* após a edição */
SE foi definido um nova valor
    altera o valor do parâmetro e propaga para os parâmetros dos quais é
        abstraído
SE passa o parâmetro para a computação
    cria novo parâmetro
    inclui o parâmetro na computação
    cria relacionamento de abstração com o(s) parâmetro(s) do(s) componente(s)

C.9 Edição dos parâmetros do objeto corrente

Edit de parâmetros (versão de computação)

obtém os parâmetros da computação
PARA CADA valor default de parâmetro
    altera o valor default do parâmetro

C.10 Seleção de objetos

Consiste em criar uma lista circular com a identificação dos objeto selecionados.

C.11 Abstração de uma computação

Abstract (versão de computação, objetos selecionados)

cria nova computação (classe) no BD particular
nova computação.última edição = 1
cria versão da nova computação do tipo edição
PARA CADA objeto selecionado

obtém todos os pipes ligados a ele sem estar conectados a buraco negro
fazer a interseção de todos estes pipes de todos os objetos
transfere os objetos selecionados da computação original para a nova computação
e os pipes que pertencem à interseção
portas dos objetos selecionados ligados aos pipes não pertencentes à interseção
são abstraídas para novas portas, criadas para a edição da nova computação
cria instância da edição da nova computação
inclui a instância como componente da computação original
as novas portas criadas são ligadas aos pipes da computação original

C.12 Expansão de uma computação

Expand (versão de computação, instância de computação componente)

obtém a versão de computação da qual a instância componente foi gerada
exclui a instância como componente da versão de computação original
exclui a instância
PARA CADA componente da computação da qual a instância componente foi gerada
cria uma instância idêntica
número do componente = computação original.último componente + 1
inclui como componente da versão de computação original

C.13 Limpa tela

Clear (versão de computação)

exclui todos os objetos componentes

C.14 Armazenamento de Computação ou Programa Cm

Store (usuário, edição de objeto [computação / programa Cm])

retira o objeto da lista de sessões do usuário
cria nova edição do objeto
número de edição = objeto.última edição + 1
cria componentes iguais à edição anterior
associa à configuração igual à edição anterior
associa usuário como proprietário da edição
inclui a nova edição na lista de sessões do usuário com acesso READ_WRITE

C.15 Armazenamento e saída do ambiente

Exit (usuário, edição de objeto [computação / programa Cm])

retira o objeto da lista de sessões do usuário
obtém o último objeto da lista de sessões do usuário
obtém os componentes deste objeto e reassume a edição

C.16 Abandono de edição

Quit (usuário, edição de objeto [computação / programa Cm])

retira o objeto da lista de sessões do usuário
exclui a edição do objeto
obtém o último objeto da lista de sessões do usuário
obtém os componentes deste objeto e reassume a edição

C.17 Ativação de configuração

Ativação (edição de computação, configuração)

resolve a configuração para a computação
PARA CADA componente que muda de versão
    exclui a instância componente original da computação
    exclui a instância
    cria uma instância da versão do objeto definida pela configuração
    inclui a instância como componente da computação

C.18 Liberação e Efetivação de versões

Liberação / Efetivação (usuário, versão de computação)

PARA CADA componente da computação do tipo computação ou programa Cm
    SE a versão do objeto da qual foi instanciada não é efetiva ou
        liberada
        ERRO
SE a edição de computação é derivação de um objeto no BD Central
    cria nova versão do objeto do qual foi derivado no BD Central
    copia a edição da computação para a nova versão do BD Central
    compara os parâmetros da nova versão com a versão anterior

```
    compara as portas da nova versão com a versão anterior
    SE parâmetros e portas são iguais
        número da versão: versão anterior.número externo igual
                          versão anterior.número interno + 1
    SENÃO
        número da versão: versão anterior.número externo + 1
                          número interno = 0
SENÃO /* é computação nova */
    cria nova computação (classe) no BD Central
    cria a primeira versão
    copia a edição da computação para a nova versão
    número da versão: número externo = 1
                      número interno = 0
```

Apêndice D

Mapeamento para DDL

```
SCHEMA ahand

    CONST
        TAM_NOME = 25;

    VALUE_SET
        Nome : STRING [25];
        Vers_id : STRUCT
                      externo : INT;
                      interno : INT
                  END;
        Versao : UNION
                     vers : Vers_id;
                     edic : INT
                 END;
        Tipo_obj : ENUM {GERAL, DE_PROJETO, PARTICULAR};
        Tipo_vers : ENUM {LIBERADA, EFETIVA, EDICAO};
        Data : STRUCT
                   dia : INT SUBR [1..31];
                   mes : INT SUBR [1..12];
                   ano : INT SUBR [1989..2050]
               END;
        Est_Consist : ENUM {INCONSISTENTE, CONSISTENTE, TRADUZIDO,
                            COMPILADO};
        Tipo_Entr_Conf : ENUM {DIRETA, INDIRETA, INCLUSAO};
        Esp_Versao : ENUM {ULTLIB, ULTEFET, CORRENTE};
```

```
        Esp_Entrada : UNION
                          tipo_vers : Esp_Versao;
                          peso      : INT
                      END;
        Tipo_Porta : ENUM {ENTRADA, SAIDA, E_S};
        Loc_Perif : ENUM {LOCAL, REDE, REMOTO};
        Direito_Acesso : ENUM {READ, WRITE, EXECUTE, RWE, RW, RE, WE};
        Acesso : STRUCT
                     owner : Direito_Acesso;
                     projeto : Direito_Acesso;
                     outros : Direito_Acesso
                 END;
        Estado_Arq : ENUM {ARQUIVADO, NAO_ARQUIVADO};
        Tipo_Param : ENUM {FORMAL, EXECUCAO, OPCAO, ARGUMENTO};
        Bloco_Porta : ENUM {BYTE, BLOCK, VSBLOCK};
        Tam_Buffer : STRUCT
                         pipe : INT;
                         estrela : INT;
                         mail : INT
                     END;
        Local_BD : ENUM {SERVER, WORKSTATION};
        Partes_Edicao : ENUM {PRINCIPAL, ED_PARAM, ED_CONFIG, ED_USE,
                              ED_IMPORT, ED_INHERIT, ED_PORT,
                              ED_RENAME, ED_CONST, ED_TYPE, ED_VAR,
                              ED_FINT, ED_FEXT, ED_IMPLEM, NO_ED};

OBJECT TYPE Computacao
    ATTRIBUTES
        nome : Nome;
        classificacao : Tipo_obj;
        documentacao : LONG_FIELD
        UNIQUE (nome)
    VERSIONS LINEAR
    (
        ATTRIBUTES
            num_vers : Versao;
            tipo : Tipo_vers;
            data_criacao : Data;
            data_efet : Data;
```

```
            data_lib : Data;
            num_ult_comp : INT;
            estado : Est_Consist;
            documentacao : LONG_FIELD;
            saida_compila : LONG_FIELD
        STRUCTURE IS
            Instancia, Parametro, Port_Virtual
    )
    STRUCTURE IS
        Ult_Lib, Ult_Efet, Ult_Edi
END Computacao;

OBJECT TYPE I_Computacao
    ATTRIBUTES
        num_componente : INT;
        posicao : INT
    STRUCTURE IS
        Port_Virtual, Bur_Negro, Val_Param, Abstr_Param
END I_Computacao;

OBJECT TYPE Programa
    ATTRIBUTES
        nome : Nome;
        classificacao : Tipo_obj;
        documentacao : LONG_FIELD
        UNIQUE (nome)
    VERSIONS LINEAR
    (
        ATTRIBUTES
            num_vers : Versao;
            tipo : Tipo_vers;
            data_criacao : Data;
            data_efet : Data;
            data_lib  : Data;
            estado : Est_Consist;
            documentacao : LONG_FIELD;
            fonte : LONG_FIELD;
            rename : LONG_FIELD;
            var_classe : LONG_FIELD;
```

```
            var_instancia : LONG_FIELD;
            constantes: LONG_FIELD;
            decl_tipo: LONG_FIELD;
            cab_fun_int : LONG_FIELD;
            cab_fun_ext : LONG_FIELD
        STRUCTURE IS
            Parametro, Port_Real, Use, Import, Inherit
    )
    STRUCTURE IS
        Ult_Lib, Ult_Efet, Ult_Edi
END Programa;

OBJECT TYPE I_Programa
    ATTRIBUTES
        num_componente : INT;
        posicao : INT;
        estado : Est_Consist
    STRUCTURE IS
        Port_Virtual, Bur_Negro, Val_Param, Abstr_Param
END I_Programa;


OBJECT TYPE Prog_C
    ATTRIBUTES
        fonte : LONG_FIELD;
        objeto : LONG_FIELD;
        executavel : LONG_FIELD
END Prog_C;

RELSHIP TYPE Traducao
    RELATES
        traduzido_de : Programa.VERSION,
        traduzido_para : Prog_C
    ATTRIBUTES
        data : Data
END Traducao;

RELSHIP TYPE Implementacao
    RELATES
```

```
        implem_por: I_Programa,
        implementa: Prog_C
END Implementacao;

RELSHIP TYPE Use
    RELATES
        usa : Programa.VERSION,
        usado : Programa.VERSION
END Use;

RELSHIP TYPE Import
    RELATES
        importa : Programa.VERSION,
        importado : Programa.VERSION
END Import;

RELSHIP TYPE Inherit
    RELATES
        herda : Programa.VERSION,
        herdado : Programa.VERSION
END Inherit;

OBJECT TYPE Def_Configuracao
    UNION
        Configuracao.VERSION, Config_Default.VERSION
END Def_Configuracao;

OBJECT TYPE Config_Default
    ATTRIBUTES
        def_ativa: INT
    VERSIONS LINEAR
    (
        ATTRIBUTES
            geral : Esp_Versao;
            projeto : Esp_Versao;
            partic : Esp_Versao;
            data_espec : Data
    )
END Config_Default;
```

```
RELSHIP TYPE Config_Obj
    RELATES
        config_de : Configuracao,
        config_por : Obj_Composto
END Config_Obj;

OBJECT TYPE Configuracao
    ATTRIBUTES
        num_ult_conf : INT
    VERSIONS LINEAR
    (
        ATTRIBUTES
            num_conf : INT;
            data_criacao : Data
            UNIQUE (num_conf)
        STRUCTURE IS
            Entr_Direta, Entr_indir, Entr_Inclusao, Entr_Rotulo
    )
END Configuracao;

RELSHIP TYPE Entr_Direta
    RELATES
        config : Configuracao.VERSION,
        obj : Vers_Obj_Composto
END Entr_Direta;

RELSHIP TYPE Entr_Rotulo
    RELATES
        config : Configuracao.VERSION,
        obj : Obj_Composto
    ATTRIBUTES
        rotulo : Esp_Versao
END Entr_Rotulo;

RELSHIP TYPE Entr_Indir
    RELATES
        quem_referencia : Configuracao.VERSION,
        referenciado : Configuracao.VERSION,
```

```
        obj : Obj_Composto
END Entr_Indir;

RELSHIP TYPE Entr_Inclusao
    RELATES
        quem_inclui : Configuracao.VERSION,
        incluido : Configuracao.VERSION
    ATTRIBUTES
        peso : INT
END Entr_Inclusao;

RELSHIP TYPE Visao_Exec
    RELATES
        obj_config : Vers_Obj_Composto,
        visao_def : Def_Configuracao
END Visao_Exec;

OBJECT TYPE Obj_Composto
    UNION
        Computacao, Programa
END Obj_Composto;

OBJECT TYPE Vers_Obj_Composto
    UNION
        Computacao.VERSION, Programa.VERSION
    AT LEAST ONCE (Visao_Exec.obj_config)
    AT MOST ONCE  (Visao_Exec.obj_config)
END Vers_Obj_Composto;

OBJECT TYPE Inst_Obj_Composto
    UNION
        I_Computacao, I_Programa
END Inst_Obj_Composto;

RELSHIP TYPE Ult_Lib
    RELATES
        obj_gen : Obj_Composto,
        versao : Vers_Obj_Composto
END Ult_Lib;
```

```
RELSHIP TYPE Ult_Efet
    RELATES
        obj_gen : Obj_Composto,
        versao : Vers_Obj_Composto
END Ult_Efet;

RELSHIP TYPE Ult_Edi
    RELATES
        obj_gen : Obj_Composto,
        edicao : Vers_Obj_Composto
END Ult_Edi;

OBJECT TYPE Estrela
    STRUCTURE IS
        Parametro
END Estrela;

OBJECT TYPE I_Estrela
    ATTRIBUTES
        num_componente : INT;
        posicao : INT
    STRUCTURE IS
        Port_Virtual, Val_Param, Abstr_Param
END I_Estrela;

OBJECT TYPE Mailbox
    STRUCTURE IS
        Parametro
END Mailbox;

OBJECT TYPE I_Mail
    ATTRIBUTES
        num_componente : INT;
        posicao : INT
    STRUCTURE IS
        Port_Virtual, Val_Param, Abstr_Param
END I_Mail;
```

```
OBJECT TYPE Pipe
    STRUCTURE IS
        Parametro
END Pipe;

OBJECT TYPE I_Pipe
    ATTRIBUTES
        num_componente : INT
    STRUCTURE IS
        Port_Entr_Sai, Val_Param, Abstr_Param
END I_Pipe;

RELSHIP TYPE Port_Entr_Sai
    RELATES
        port_entr : Port_Virtual,
        port_sai : Port_Virtual
END Port_Entr_Sai;

OBJECT TYPE Periferico
    ATTRIBUTES
        nome : Nome;
        tipo : Tipo_Porta
        UNIQUE (nome)
    VERSIONS LINEAR
    (
        ATTRIBUTES
            id_unidade : Nome;
            local : Loc_Perif;
            endereco : Nome;
            especificacao : LONG_FIELD
            UNIQUE (id_unidade)
    )
    STRUCTURE IS
        Parametro
END Periferico;

OBJECT TYPE I_Periferico
    ATTRIBUTES
        num_componente : INT;
```

```
        posicao : INT
    STRUCTURE IS
        Port_Virtual, Val_Param
END I_Periferico;

OBJECT TYPE Arquivo
    ATTRIBUTES
        nome : STRING [60];
        classificacao : Tipo_Obj;
        acesso : Acesso;
        data_criacao : Data
        UNIQUE (nome)
    VERSIONS LINEAR
    (
        ATTRIBUTES
            data_criacao : Data;
            estado : Estado_Arq;
            local : STRING [50]
    )
    STRUCTURE IS
        Parametro
END Arquivo;

OBJECT TYPE I_Arquivo
    ATTRIBUTES
        num_componente : INT;
        posicao : INT
    STRUCTURE IS
        Port_Virtual, Val_Param
END I_Arquivo;

OBJECT TYPE Porta
    UNION
        Port_Virtual, Port_Real
END Porta;

OBJECT TYPE Port_Virtual
    ATTRIBUTES
        tipo : Tipo_Porta
```

```
    STRUCTURE IS
        Abstr_Porta
END Port_Virtual;

OBJECT TYPE Port_Real
    ATTRIBUTES
        tipo : Tipo_Porta;
        nome : Nome;
        blocagem : Bloco_Porta;
        tam_bloco : INT
END Port_Real;

RELSHIP TYPE Abstr_Porta
    RELATES
        abstraido_de : Port_Virtual,
        abstraido_para : Port_Virtual
END Abstr_Porta;

RELSHIP TYPE Bur_Negro
    RELATES
        fechado : Port_Virtual,
        de : Inst_Obj_Composto
END Bur_Negro;

OBJECT TYPE Parametro
    ATTRIBUTES
        nome : Nome;
        tipo : Tipo_Param;
        tipo_dado : Nome;
        val_default : LONG_FIELD
END Parametro;

OBJECT TYPE Obj_Param
    UNION
        Instancia, Prog_C
END Obj_Param;

RELSHIP TYPE Val_Param
    RELATES
```

```
        define_param : Obj_Param,
        definido_por : Parametro
    ATTRIBUTES
        valor : LONG_FIELD
END Val_Param;

RELSHIP TYPE Abstr_Param
    RELATES
        param_de : Instancia,
        abstraido_de : Parametro,
        abstraido_para : Parametro
END Abstr_Param;

RELSHIP TYPE Coordenacao
    RELATES
        chefiado : Grupo,
        chefe : Usuario
    ATTRIBUTES
        data_inicio : Data;
        data_fim : Data
END Coordenacao;

OBJECT TYPE BD_Central
    ATTRIBUTES
        num_edicao : INT;
        buffer : Tam_Buffer;
        login_adm : Nome;
        passwd : Nome;
        tmp_arq : INT;
        num_proj : INT;
        num_usuario : INT
    STRUCTURE IS
        Parametro
END BD_Central;

OBJECT TYPE Projeto
    ATTRIBUTES
        numero : INT;
        nome : Nome;
```

```
            login : Nome;
            passwd : Nome;
            num_edicao : INT;
            buffer : Tam_buffer;
            data_inicio : Data;
            data_fim : Data
            UNIQUE (numero, login)
    END Projeto;

    OBJECT TYPE Usuario
        ATTRIBUTES
            numero : INT;
            nome : Nome;
            login : Nome;
            passwd : Nome;
            nome_BD : Nome;
            local : Local_BD;
            num_edicao : INT;
            buffer : Tam_Buffer;
            diretorio : STRING [50];
            data_inicio : Data;
            data_fim : Data
            UNIQUE (numero, login)
    END Usuario;

    RELSHIP TYPE Membro
        RELATES
            agrupa : Projeto,
            membro_de : Usuario
        ATTRIBUTES
            inicio: Data;
            fim: Data
    END Membro;

    OBJECT TYPE Grupo
        UNION
            BD_Central, Projeto
        AT LEAST ONCE (Coordenacao.chefiado)
    END Grupo;
```

```
RELSHIP TYPE Sessao
    RELATES
        edicao_de : Vers_Obj_Composto,
        editado_por : Usuario
    ATTRIBUTES
        acesso : Direito_Acesso;
        parte_ed : Partes_Edicao;
        posicao : INT
END Sessao;

RELSHIP TYPE Icone
    RELATES
        repres_de : Classe,
        utilizado_por : Proprietario
    ATTRIBUTES
        representacao : LONG_FIELD
END Icone;

OBJECT TYPE Proprietario
    UNION
        BD_Central, Projeto, Usuario
END Proprietario;

OBJECT TYPE Objeto
    UNION
        Computacao, Computacao.VERSION, Programa, Programa.VERSION,
        Config_Default, Configuracao.VERSION, Periferico.VERSION,
        Arquivo
    AT MOST ONCE  (Propriedade.posse)
END Objeto;

RELSHIP TYPE Propriedade
    RELATES
        posse : Objeto,
        owner : Proprietario
END Propriedade;

OBJECT TYPE Instancia
```

```
    UNION
        I_Computacao, I_Programa, I_Periferico, I_Arquivo,
        I_Mail, I_Estrela, I_Pipe
    AT LEAST ONCE  (Rel_Instanciacao.instancia_de)
END Instancia;

OBJECT TYPE Classe
    UNION
        Computacao, Programa, Periferico, Arquivo,
        Mailbox, Estrela, Pipe
END Classe;

OBJECT TYPE Obj_Instanciaveis
    UNION
        Computacao.VERSION, Programa.VERSION, Periferico.VERSION,
        Arquivo
END Obj_Instanciaveis;

RELSHIP TYPE Rel_Instanciacao
    RELATES
        instancia_de : Instancia,
        gerado_de : Obj_Instanciaveis
END Rel_Instanciacao;

RELSHIP TYPE Derivacao
    RELATES
        original : Obj_Composto,
        derivado : Obj_Composto
END Derivacao;

END ahand
```

Bibliografia

[Abr88] Abramowicz, K., et al, *DAMOKLES - Database Management System for Design Applications - Reference Manual Release 2.0*, Forschungszentrum Informatik an der Universität Karlsruhe, Mar 1988

[Ala89] Alagić, S., *Object-Oriented Database Programming*, Springer-Verlag, 1989

[Ari90] Arias, H.P., *Editor para a LegoShell*, versão preliminar de tese em andamento, Departamento de Ciência da Computação, UNICAMP

[Bal86] Balzer, R.M., "Living in the Next Generation Operating System", *Proc. of the IFIP 10th World Computer Congress*, Set 1986, pp. 283-291

[Ban88] Bancilhon, F., "Object-Oriented Database Systems", *Proc. of the 7th ACM Sigart-Sigmod-Sigact Symposium on Principles of Database Systems*, Mar 1988

[Bat86] Batory, D.S, "GENESIS: A Project to Develop an Extensible Database Management System", *Proc. of the 1986 International Workshop on Object-Oriented Database Systems*, Set 1986, pp. 207-207

[Ber89] Bernstein, P.A., "Database system Support for Software Engineering - An Extended Abstract", *Proc. of the 9th IEEE International Conference on Software Engineering*, 1989, pp. 166-178

[Boo86] Booch, G., "Object-Oriented Development", *IEEE Transactions on Software Engineering*, vol. 12, no. 2, (Fev 1986), pp. 211-221

[Bou85] Bourguignon, J.P., "PCTE: A Basis for a Portable Common Tool Environment", *ESPRIT'84 Status Report on Ongoing Work*, 1985

[Bra83] Brachman, R.J., "What IS-A is and isn't: An Analysis of Taxonomic Links in Semantic Networks", *Computer*, vol. 16, no. 10, (Out 1983), pp. 67-73

[CDe86] Carey, M.J. e DeWitt, D.J., "The Architecture of the EXODUS Extensible DBMS", *Proc. of the 1986 International Workshop on Object-Oriented Database Systems*, Set 1986, pp. 52- 65

[Che76] Chen, P.P, "The Entity-Relationship Model - Toward a Unified View of Data", ACM Transactions on Database Systems, vol. 1, no. 1, (Mar 1976), pp. 9-36

[Cro85] Croft, W.B., "Task Management for an Intelligent Interface", *IEEE Database Engineering*, vol. 8, no. 4, (Dez 1985), pp. 8-13

[Dar87] Dart, S.A., et al, "Software Development Environments", *Computer*, (Nov 1987), pp. 18-28

[DGL87] Dittrich, K., Gotthard, W. e Lockemann, P.C., "DAMOKLES - the Database System for the UNIBASE Software Engineering Environment", *IEEE Data Engineering*, vol. 10, no. 1, (Mar 1987), pp. 37-47

[Dit86] Dittrich, K.R., "Object-Oriented Database Systems - A Workshop Report",*Proc. 5th ER Conference*, 1986, pp. 1- 16

[DKL85] Derret, N., Kent, W. e Lyngback, P., "Some Aspects of Operations in an Object-Oriented Database", *IEEE Database Engineering*, vol. 8, no. 4, (Dez 1985), pp. 66-74

[DLi87] Drummond, R. e Liesenberg, H., "A_HAND: Ambiente de Desenvolvimento de Software Baseado em Hierarquias de Abstração em Níveis Diferenciados", *IV Encontro do Projeto ETHOS*, Abr 1987, pp. 313-322

[DLi87] Drummond, R. e Liesenberg, H., "Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS", *I Encontro IBM de Ciência e Tecnologia em Informática*, Nov 1987

[Don85] Donahue, J., "Integration Mechanisms in Cedar", *Sigplan Notices (Proc. ACM Sigplan Symp. on Language Issues in Programming Environments)*, Jul 1985, pp. 245-251

[Dow87] Dowson, M., "Integrated Project Support with IStar", *IEEE Software*, (Nov 1987), pp. 6-15

[Dru89] Drummond, R., "LegoShell: Linguagem de Computações", *Anais do III Simp. Bras. de Engenharia de Sortware*, 1989

[DSi88] Drummond, R. e da Silva, F.Q.B., "Manual de Referência - Linguagem Cm", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, Mai 1988

[Est86] Estrin, G., et al, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", *IEEE Transactions on Software Engineering*, vol. 12, no. 2, (Fev 1986), pp. 293-311

[FLa89] Fernandes, A.C. e Laender, A.H.F., "MER+: Uma Extensão do Modelo de Entidades e Relacionamentos para Projeto Conceitual de Bancos de Dados", Rel. Téc. RT001/89, Dep. de Ciência da Computação, UFMG, 1989

[Fur90a] Furuti, C.A., "Cm, Linguagem e Implicações", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, 1990

[Fur90b] Furuti, C.A., "Implementação do Tradutor Cm $->$ C", tese de mestrado em andamento, Departamento de Ciência da Computação, UNICAMP

[GRo85] Goldberg, A. e Robson, D., *Smalltalk-80 - The Language and its Implementation*, Addison-Wesley Publishing Company, 1985

[HBe85] Hoffnagle, G.F. e Beregi, W.E., "Automating the Software Development Process", *IBM Systems Journal*, vol. 24, no. 2, (1985), pp. 102-120

[HKi87] Hudson, S.E. e King, R., "Object-Oriented Database Support for Software Environments", *Sigmod Record*, vol. 16, no. 3, (Dez 1987), pp. 491-503

[HKi88] Hudson, S.E. e King, R., "The Cactis Project: Database Support for Software Environments", *IEEE Transactions on Software Engineering*, vol. 14, no. 6, (Jun 1988), pp. 709-719

[HMa89] Hara, C.S e Magalhães, G.C., "Uma Experiência em Modelar Banco de Dados Orientado a Objetos", *Anais do 4º Simp. Bras. de Banco de Dados*, 1989, pp. 119-127

[HNo86] Habermann, A.N. e Notkin, D., "GANDALF: Software Development Environments", *IEEE Transactions on Software Engineering*, vol. 12, no. 12, (Dez 1986), pp. 1117-1127

[HuK87] Hull, R. e King, R., "Semantic Database Modeling: Survey, Applications, and Research Issues", *ACM Computing Surveys*, vol. 19, no. 3, (Set 1987), pp. 201-260

[KCo86] Khoshafian, S.N. e Copeland, G.P., "Object Identity", *OOPSLA'86 Proceedings*, Set 1986, pp. 406-416

[Kim87] Kim, W., et al, "Composite Object Support in an Object-Oriented Database System", *OOPSLA'87 Proceedings*, Out 1987, pp. 118-125

[KRi78] Kernighan, B.W. e Ritchie, D.M, *The C Programming Language*, Prentice-Hall, 1978

[KTu87] Kuo, J.H. e Tu, H., "Prototyping a Software Information Base for Software Engineering Environments", *IEEE Proceedings of Compsac'87*, 1987, pp. 38-44

[Lam88] Lamsweerde, A.V, et al, "Generic Lifecycle Support in the ALMA Environment", *IEEE Transactions on Software Engineering*, vol. 14, no. 6, (Jun 1988), pp. 720-741

[LHo89] Liu, L. e Horowitz, E., "Object Database Support for a Software Project Management Environment", *Sigplan Notices (Proc. of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments)*, vol. 24, no. 2, (Fev 1989), pp. 85-96

[Lin84] Linton, M.A., "Implementing Relational Views of Programs", *Proc. of the ACM Sigsoft/Sigplan Software Engineering Symp. on Practical Development Environments*, Mai 1984, pp. 132-140

[Mag89] Magalhães, L.P., et al, "Implementação de um Banco de Dados não Convencional", *Anais do 4º Simp. Bras. de Banco de Dados*, 1989, pp. 77-89

[Mai85] Maier, D., "Object-Oriented Database Development at Servio Logic", *IEEE Database Engineering*, vol. 8, no. 4, (Dez 1985), pp. 58-65

[Mai89] Maier, D., "Why isn't there an Object-Oriented Data Model?", *Proc. of the IFIP 11th World Commputer Congress*, Set 1989, pp. 793-798

[Mar85] Marti, R.W., "Applying Database Techniques to the Management of Program Module Descriptions", *Proc. of the Second Conference on Software Development Tools, Techniques, and Alternatives*, Dec 1985, pp. 132-140

[Mey87] Meyer, B., "Reusability: The Case for Object- Oriented Design", *IEEE Software*, (Mar 1987), pp. 50-64

[Miy86] Miyasato, B.Z., "Modelos Semânticos de Dados: Comparações e Aplicações", *Coleção PRODESP*, Série Software 4, 1986

[Nil80] Nilsson, N.J, *Principles of Artificial Inteligence*, Springer-Verlag, 1980

[Pen86] Penedo, M.H., "Prototyping a Project Master Data Base for Software Engineering Environments", *Sigplan Notices*, vol. 1, no. 22, (1986), pp. 1-10

[Pet87] Peterson, G.E., "Object-Oriented Design", em *Tutorial: Object-Oriented Computing - vol. 2: Implementations*, Peterson, G.E., Eds., (IEEE), 1987, pp. 1-3

[PMa88] Peckham, J. e Maryanski, F., "Semantic Data Models", *ACM Computing Surveys*, vol. 20, no. 3, (Set 1988), pp. 153-189

[Row89] Rowe, L.A., "Report on the 1989 Software CAD Databases Workshop", *Proc. of the IFIP 11th World Computer Congress*, Set 1989, pp. 719-725

[RWi86] Riddle, W.E. e Williams, L.G., "Software Environments Workshop Report", *ACM Sigsoft Software Engineering Notes*, vol. 11, no. 1, (Jan 1986), pp. 73-102

[SDr88] de Sarno, A. e Drummond, R., "LegoShell: Linguagem de Configuração de Programas", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, (Dez 1988)

[Set86] Setzer, V.W., *Projeto Lógico e Projeto Físico de Bancos de Dados*, V Escola de Computação, 1986

[SLD88] da Silva, F.Q.B, Liesenberg, H. e Drummond, R., "Programação em Cm", Rel. Interno, Departamento de Ciência da Computação, UNICAMP, Mar 1988

[Sto87] Stonebraker, M., "The Design of the POSTGRES Storage System", *Proc. of the 13th VLDB Conference*, 1987, pp. 289-300

[SZa87] Smith, K.E. e Zdonik, S.B., "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems", *OOPSLA'87 Proceedings*, Out 1987, pp. 452-465

[Tei81] Teitelbaum, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, vol. 24, no. 9, (Set 1981), pp. 563-573

[Tha86] Thatte, S.M., "Persistent Memory: A Storage Architecture for Object-Oriented Database Systems", *Proc. of the 1986 International Workshop on Object-Oriented Database Systems*, Set 1986, pp. 148-159

[Tro90] Trombetta, O., "Editor Sensível a Sintaxe para a Linguagem Cm", tese de mestrado em andamento, Departamento de Ciência da Computação, UNICAMP

[TYF86] Teorey, T.J., Yang, D. e Fry, J.P, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", *Computing Surveys*, vol. 18, no. 2, (Jun 1986), pp. 197-222

[VMD89] Victorelli, E.Z., Magalhães, G.C. e Drummond, R., "Mecanismo de Gerenciamento de Versões e Configurações do A_HAND", *Revista Brasileira de Computação*, vol. 5, no. 2, (Dez 1989), pp. 3-9

[Wag88] Wagner, C.F, "Implementing Abstraction Hierarchies", *Proc. ER Conference*, 1988

[Yok89] Yokota, K., "What is Expected of an Object-Oriented Data Model?", *Proc. of the IFIP 11th World Computer Congress*, Set 1989, pp. 799-800

[ZMa90] Zdonik, S.B. e Maier, D., "Fundamentals of Object- Oriented Databases", em *Readings in Object-Oriented Database Systems*, Zdonik, S.B. e Maier, D., Eds, Morgan Kaufmann Publishers, Inc., California, 1990