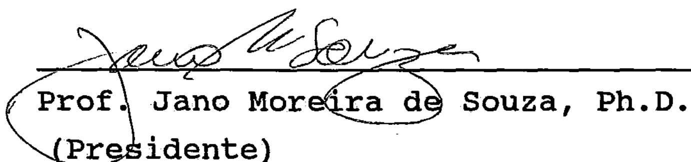
MANO: LINGUAGEM DE **MANIPULAÇÃO** DE OBJETOS DO PROTOGEO
E SEU PROCESSADOR

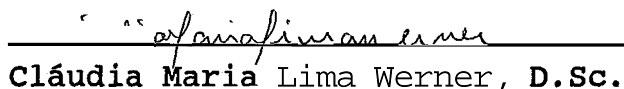
Jugurta Lisboa Filho

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

Aprovada por:

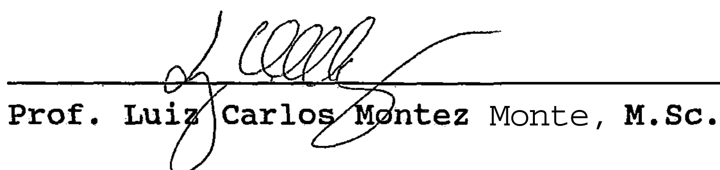


Prof. Jano Moreira de Souza, Ph.D.
(Presidente)



Cláudia Maria Lima Werner, **D.Sc.**

Profª. Regina **Célia** de **Souza Pereira,** D.Sc.



Prof. Luiz Carlos Montez Monte, **M.Sc.**

RIO DE JANEIRO, RJ - BRASIL
OUTUBRO DE 1992

LISBOA FILHO, JUGURTA

MANO: Linguagem de Manipulação de Objetos do ProtoGEO e seu Processador [Rio de Janeiro] 1992.
XIII, 159 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1992)
Tese - Universidade Federal do Rio de janeiro, COPPE
1. Banco de Dados
I. COPPEIUFRJ II. Título (série)

À minha esposa *e*
amiga **Janaina**

Aos meus filhos
Pedro, Tomas e Maria

Aos meus pais
Jugurta e Diva

Agradecimentos,

Ao Professor Jano Moreira de Souza, pela orientação e incentivo pelo trabalho, e aos professores da COPPE em geral, pela dedicação e empenho em manter um curso de alto nível, apesar de todas as dificuldades enfrentadas pela instituição.

Aos pesquisadores da linha de Banco de Dados da COPPE, Marta Mattoso, Cláudia Werner, Cláudio Trotta e principalmente ao Prof. Luiz Carlos Monte, cuja participação neste trabalho foi de fundamental importância.

A todos os colegas da COPPE, pelo companheirismo nos momentos alegres e também nas horas mais difíceis, em especial a Cláudia Susie, Milton Ramires e Eduardo Chaves.

Aos colegas do Departamento de Informática da Universidade Federal de Viçosa, pela cooperação empenhada principalmente nos momentos finais deste trabalho.

Aos meus pais Jugurta e Diva e aos meus irmãos Magda, Gizelda, Mirna e Antônio pelo carinho e às minhas tias Anita e Celeste pela imensa ajuda no decorrer destes dois anos e meio de convivência.

Ao Sr. Jayr Miranda, por ter feito uma revisão tão minuciosa e enriquecedora do texto final.

À minha querida esposa Janaina, que a todo instante esteve ao meu lado com seu apoio e compreensão, e aos meus filhos Pedro, Tomas e Maria que tanto me ensinaram com seus gestos, seus questionamentos e suas colocações.

A todos aqueles que não foram citados mas que participaram direta ou indiretamente desta empreitada.

E, finalmente à Universidade Federal de Viçosa e à CAPES pela oportunidade do treinamento e pelo suporte financeiro.

Resumo da Tese apresentada à COPPEIUFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M. Sc.).

MANO: LINGUAGEM DE MANIPULAÇÃO DE OBJETOS DO PROTOGEO
E SEU PROCESSADOR

Jugurta Lisboa Filho
OUTUBRO, 1992

Orientador: Prof. Jano Moreira de Souza
Programa : Engenharia de Sistemas e Computação

Encontra-se em desenvolvimento na COPPE/UFRJ o sistema GEOTABA, que é um sistema de gerência de bases de dados orientado a objetos (SGBDOO). O GEOTABA baseia-se em um modelo de objetos cuja preocupação central está em resolver os problemas de incompatibilidade entre as linguagens de definição e manipulação de dados, as de programação de métodos e as de especificação de interface com usuário. Com o objetivo de validar o modelo de objetos do GEOTABA, está sendo construído um protótipo, o ProtoGEO.

Este trabalho descreve a Linguagem de Manipulação de Objetos do ProtoGEO (MANO) e seu Processador. A linguagem MANO foi projetada para ser a ferramenta de programação do nível de implementação do modelo de objetos do GEOTABA, fornecendo uma sintaxe para a definição de métodos e envio de mensagens. Foi definida segundo os conceitos do paradigma da orientação a objetos e fornece recursos de interface com usuário.

O Processador, escrito em C++, corresponde ao núcleo do ProtoGEO e é o responsável pelo processamento de mensagens/métodos. Funciona como uma "máquina de pilha" que executa as instruções do código intermediário, gerado pelo Compilador da linguagem MANO. Para permitir a criação e a manipulação de objetos, foi implementado também o Gerente do "Buffer" de Objetos que gerencia os objetos residentes em memória.

Abstract of Thesis presented to COPPEIUFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.).

MANO: OBJECT MANIPULATION LANGUAGE OF THE PROTOGEO

AND ITS PROCESSOR

Jugurta Lisboa Filho
OCTOBER, 1992

Thesis Supervisor: Prof. Jano Moreira de Souza
Department: Computation and Systems Engineering

The GEOTABA system, an object-oriented database management system (OODBMS), has been developed in COPPE/UFRJ. GEOTABA is based on an object model whose central preoccupation is to solve the problems of impedance mismatch between the data definition and manipulation languages, methods programming languages, and user interface specification languages. With the objective of validating the GEOTABA object model, a prototype called ProtoGEO is being built.

This work describes the ProtoGEO Object Manipulation Language (MANO) and its Processor. The MANO language was projected to be the programming tool for the implementation level of the GEOTABA object model, providing a syntax for method definition and message sending. It was also defined according to the object-oriented paradigm concepts and provides a user interface facility.

The Processor, written in C++, corresponds to the ProtoGEO nucleus and is responsible for message/method processing. It works like a "stack machine" that executes the intermediate code instructions produced by the MANO language compiler. To permit object creation and manipulation, the Object Buffer Manager was implemented whith the goal of controlling memory resident objects.

Índice

Capítulo I - INTRODUÇÃO

I.1 Motivação.....................................1

1.2 Justificativa.................................4

1.3 Estrutura da Tese.............................6

Capítulo II - BANCO DE DADOS ORIENTADO A OBJETOS: Algumas Experiências

11.1 Introdução...................................8

11.2 Sistema O_2...................................10

Modelo de Objetos do O_2.........................12

Outras Características do O_2....................15

Interfaces para Programação de Aplicações.........16

Arquitetura do O_2Engine.........................17

O Compilador CO_2................................18

Comentários Finais................................20

11.3 Sistema ORION...............................21

Modelo de Objetos Orion...........................22

Arquitetura Funcional.............................23

Evolução Dinâmica de Esquema......................24

Versões de Objetos................................25

Objetos Compostos.................................26

Gerenciamento de Objetos..........................27

Considerações Finais..............................29

11.4 Sistema GemStone 30

Modelo de Dados GemStone..........................30

A Linguagem OPAL..................................31

Arquitetura do Sistema GemStone...................32

Interfaces para Aplicações........................34

Interface GemStone - C............................34
Interface GemStone - C++..........................35
Interface GemStone - Smalltalk....................36
Alteração de Esquema no GemStone..................37
Considerações Finais..............................38

Capítulo III - PROTOGEO: PROTóTIPO DO SGBDOO GEOTABA

1.1 Introdução......................................39
111.2 Modelo de Objetos do GEOTABA...................43
III.2.1 principais Características do Modelo......44
Um Modelo de Dados Orientado a Objetos.........44
Níveis de Abstração do Modelo..................45
O Nível de Implementação.......................46
O Nível Conceitual.............................47
O Nível Externo................................48
O Nível de Representação.......................48
III.2.2 O Modelo de objetos e o ProtoGEO..........49
111.3 MANO - Linguagem de Manipulação de Objetos.....50
III.3.1 Características da Linguagem MANO.........51
Características de Interface c/ Usuário........51
Estrutura de um Método MANO....................52
Mensagens em MANO..............................54
Definição de Classes...........................59
Criação de Objetos.............................61
Variáveis Globais..............................61
Variáveis Especiais $mim e $super..............62
Outras Características da Linguagem MANO.......63
111.4 Arquitetura do ProtoGEO.........................67
III.4.1 Gerente de Objetos........................68
III.4.2 Gerente de Execução.......................68
III.4.3 Gerente de Armazenamento..................70

Capítulo IV - PROCESSADOR MANO

IV.1 Introdução.....72
IV.2 Estruturas de Dados.....73
Métodos.....73
Contexto.....78
Contexto de Bloco.....80
Classes.....81
IV.3 Instruções MANO.....83
IV.4 O Processador.....88
Instruções de Pilha.....90
Instruções de Desvio.....91
Instruções de Envio de Mensagens.....91
Instruções de Retorno.....93
IV.5 Rotinas Primitivas.....93
Primitivas Aritméticas.....94
primitivas de Manipulação de Cadeias.....96
primitivas de Gerenciamento de Memória.....96
primitivas de Controle.....97
primitivas do Sistema.....97

Capítulo V - COMPILADOR MANO

V.1 Introdução.....98
V.2 O Compilador.....100
V.3 Analisador Léxico.....103
V.4 Analisador Sintático.....106
A árvore sintática.....107
O segundo passo.....115
V.5 Gerador de Código.....119
Identificador de Variável.....120
Número.....122
Literal Constante.....122
Seqüência de Identificadores.....123

Operadores..125
Mensagem com Parâmetros (Chave)...................127
Atribuição.......................................129
Bloco de Mensagens...............................130
Mensagens Especiais..............................131
Finalização do Método............................134

Capítulo VI - GERENTE DE ARMAZENAMENTO

VI.1 Introdução...................................135
VI.2 Gerente do "Buffer" de Objetos...............138
"Buffer" de Objetos..............................138
Tabela de Objetos................................140
Identidade de Objetos (idO)......................141
Entrada de Objetos na Tabela.....................143
Interface do Gerente do "Buffer" de Objetos......145

Capítulo VII - CONCLUSÕES

....150

BIBLIOGRAFIA

....153

Índice de Figuras

Capítulo II

2.1 Arquitetura Funcional do O_2....................11
2.2 Definição de classes em O_2C....................13
2.3 Corpo de método em O_2C.........................14
2.4 Arquitetura do Compilador C02.....................19
2.5 Arquitetura Funcional do ORION-1..................23
2.6 Subsistema de Armazenamento do ORION-1............27
2.7 Sistema GemStone Versão 1.0 33

Capítulo III

3.1 Níveis de Abstração do Modelo de Objetos.......45
3.2 Método MANO usando atributos de caracteres.....51
3.3 Exemplo de Método em MANO......................52
3.4 Exemplo de uso de variável temporária..........53
3.5 Mensagens onde os seletores são operadores.....55
3.6 Exemplo de mensagens em cascata................57
3.7 Método MANO usando blocos de mensagens.........58
3.8 Definição da classe Janela.....................60
3.9 Declaração de Variáveis Globais................62
3.10 Exemplo de uso da variável $mim...............63
3.11 Acesso e alteração de variável de instância...64
3.12 Definição da classe Janela_de_Edição..........65
3.13 Arquitetura do ProtoGEO.......................67
3.14 Núcleo do ProtoGEO............................ 69

Capítulo IV

4.1 Tabela de Seletores Especiais..................74
4.2 Cabeçalho de Método............................75
4.3 Valores possiveis do campo código..............76
4.4 Estrutura de um método.........................77
4.5 Contexto de Execução...........................79
4.6 Contexto de Bloco..............................80
4.7 Estrutura de uma instância de Classe...........82
4.8 Tabela de Instruções...........................84

Capítulo V

5.1 Interligação de Arquivos Fonte.................99
5.2 Estrutura do Compilador MANO..................101
5.3 Gramática da Linguagem MANO...................104
5.4 Conjunto de Símbolos da Linguagem.............106
5.5 Lista de comandos da árvore sintática.........108
5.6 Representação do nó da árvore sintática.......109
5.7 Exemplo de árvore sintática...................110
5.8 Mensagem nas formas pré e pós-fixa............111
5.9 Exemplo de árvore sintática com MSG_CHAVE.....112
5.10 Exemplo de árvore sintática com MSG_BLOCO....114
5.11 Transformação de atribuição em mensagem......116
5.12 Transformação de MSG em MSG_CHAVE............118
5.13 Gere Identificador de Variável...............121
5.14 Gere Número..................................122
5.15 Gere Literal Constante.......................123
5.16 Gere Mensagem Unária.........................123
5.17 Lista de Seletores Especiais..................124
5.18 Gere Mensagem Seletor Especial...............124
5.19 Gere Operadores..............................125
5.20 Lista de Operadores Especiais................126

5.21 Gere Mensagem Binária........................126
5.22 Gere Mensagem Chave...........................127
5.23 Gere Mensagem Ternária........................128
5.24 Gere Mensagem com N Argumentos................128
5.25 Gere Atribuição...............................129
5.26 Gere Bloco de Mensagens.......................131
5.27 Gere Mensagem #seVero:........................132
5.28 Gere Comando Composto.........................132
5.29 Gere Mensagem #seVero:senão:..................133
5.30 Gere Mensagem #enquanto:......................133

Capítulo VI

6.1 Gerente de Armazenamento......................136
6.2 Formato de um Objeto..........................139
6.3 Esquema de Dupla Indireção....................141
6.4 Identidade de Objetos.........................142
6.5 Entradas na Tabela de Objetos.................143
6.6 Lista de Entradas Disponíveis.................144

Capítulo I

Introdução

1.1 Motivação

A partir do início da década de **70,** a tecnologia de banco de dados vem sendo utilizada por empresas usuárias de sistemas de computação. Sistemas gerenciadores de banco de dados (SGBDs) relacionais, chamados assim por utilizarem o modelo de dados relacional, têm sido amplamente utilizados por atenderem, de forma satisfatória, às necessidades dos sistemas voltados às áreas comerciais. Estes sistemas caracterizam-se por manipularem grandes volumes de dados, mas que apresentam estruturas simples, podendo ser naturalmente modelados na forma de tabelas.

Com o desenvolvimento de novas tecnologias e o crescimento da utilização de sistemas de computadores, surgiram novas áreas de aplicação, porém, a tecnologia de banco de dados até o momento existente, não atende de forma satisfatória aos requisitos impostos por essas aplicações.

Entre as novas áreas de aplicação destacam-se a engenharia de software assistida por computador (CASE), sistemas de apoio a projeto e manufatura **(CAD/CAM),** automação de escritório (OIS), aplicações médicas e científicas, sistemas de informações geográficas (GIS), representação do conhecimento para inteligência artificial e aplicações comerciais onde os SGBDs tradicionais mostraram-se inadequados.

Estas aplicações diferem das convencionais por diversos motivos, entre eles a necessidade de processar entidades com estruturas complexas como imagem, som, textos, etc.

Segundo **Melo [MELO88],** os principais requisitos dessas aplicações, chamadas não convencionais, são:

a) definição e manipulação de objetos complexos

b) controle de transações de longa duração

c) consultas complexas e recursivas

d) controle de versões de objetos

Objetos complexos são representações de entidades do mundo real cuja estrutura contém dados não usuais (campos longos por exemplo). Os novos SGBDs devem possuir uma semântica que possibilite que um objeto complexo seja manipulado ou acessado como uma unidade de processamento.

As transações nos sistemas não convencionais podem ser muito demoradas, pois podem envolver grande volume de dados e ter um processamento muito complexo. As consultas nos sistemas de informações geográficas por exemplo, podem ser bastante complexas onde o processador de consultas deve ser capaz de resolver consultas do tipo "ao longo de", "ao norte de", "na fronteira de", etc. Já nos sistemas de CAD, consultas à objetos compostos podem necessitar de consultas recursivas.

Em algumas áreas de aplicação (CASE, CAD e OIS), é fundamental a existência de mecanismos de controle de versões. As versões de um objeto podem representar, por exemplo, o histórico do desenvolvimento do objeto, diferentes alternativas dentro de um projeto, etc.

A solução que tem sido adotada para que os sistemas de banco de dados possam atender a esses novos requisitos é a de trazer os conceitos utilizados no paradigma da orientação a objetos para o âmbito dos sistemas de banco de dados, levando ao desenvolvimento dos chamados Sistemas de Gerência de Objetos (SGO) ou SGBDs da próxima geração **[CATT91].**

Duas abordagens diferentes surgiram no desenvolvimento destes sistemas. A primeira é a que procurou adicionar aos SGBDs relacionais características do paradigma da orientação a objetos, tentando assim aproveitar toda uma cultura formada pelos sistemas já instalados. Esta abordagem tem gerado sistemas que estão sendo classificados como SGBDs relacionais estendidos. Como exemplos pode-se citar o POSTGRES [STON87] e o Starburst [SCHW86].

A segunda abordagem buscou o desenvolvimento de um modelo próprio, que fosse expressivo e bem fundamentado. Nesta categoria de sistemas, chamados de SGBDs orientado a obietos (SGBDOO), estão os projetados como uma extensão das linguagens de programação orientadas a objetos, para prover persistência, controle de concorrência, linguagem de consulta e outras características de banco de dados. Entre os sistemas que seguiram essa abordagem, estão o O_2 [BANC88], ORION [BANE87a], GemStone [MAIE85], ENCORE [HORN87] e ObjectStore [LAMB91].

Os Sistemas de Gerência de Objetos vêm recebendo muita atenção, porém, ainda hoje não existe uma definição padrão de um modelo de objetos para os SGBDOOs, como existe para os SGBDs relacionais, apesar de existirem vários trabalhos direcionados neste sentido [DITT86a] [ATKI90] [STON90].

O que se pode notar, no entanto, é que os diversos protótipos de SGOs (alguns já em fase de comercialização) estão seguindo uma linha básica comum, que é a de prover, além das características existentes nos SBGDs relacionais, como gerenciamento de memória secundária, persistência, concorrência, reconstrução e facilidade de consultas "ad hoc", características existentes nas linguagens de programação orientadas a objetos tais como: suporte a objetos complexos, identidade de objeto, encapsulamento, hierarquia de classes, herança de propriedades, etc.

1.2 Justificativa

Uma das características mais importantes de um SGBD é a linguagem que ele fornece para operação do banco de dados. Os SGBDs relacionais, normalmente são providos de uma linguagem de consulta declarativa (padrão SQL) englobando duas sublinguagens, uma para definição do esquema de dados (LDD) e outra para manipulação dos dados (LMD). Além disso, as aplicações podem ser programadas em algumas linguagens de programação procedimentais como FORTRAN, COBOL, PL/I, etc, com o acesso ao banco de dados sendo feito através de chamadas de procedimentos que recuperam ou gravam informações no banco de dados [DATE86].

Como os sistemas de tipos, do ambiente de banco de dados e das linguagens de programação são diferentes, a transferência de dados entre os dois ambientes requer um procedimento de conversão de tipos.

Os Sistemas de Gerência de Objetos (SGOs) devem fornecer, além da linguagem de operação do banco de dados, uma linguagem para a programação dos métodos (procedimentos que manipulam o estado dos objetos).

Como nos SGBDs relacionais, nos SGOs existe o problema de incompatibilidade entre as linguagens de operação e de programação. Este problema tem sido tratado como ***problema de "impedance mismatch"*** [SHYY91] e tem motivado as pesquisas na área de Linguagens de Programação de Banco de Dados (LPBD) [BLOO87], que tentam buscar uma solução a partir da utilização de um único ambiente de execução com uma linguagem procedimental e um sistema de tipos. A linguagem de programação é geralmente estendida com uma sintaxe de linguagem de consulta [CATT91].

Segundo G. Copeland [COPE84], existem dois tipos de incompatibilidade, o primeiro é a incompatibilidade conceitual, que ocorre quando as linguagens de operação do banco de dados e as de programação empregam diferentes paradigmas de programação. Por exemplo, a linguagem de operação pode ser uma linguagem declarativa e a linguagem de programação ser estritamente procedimental.

O outro tipo é a incomvatibilidade estrutural, que acontece quando as linguagens envolvidas suportam diferentes sistemas de tipos. Neste caso, quando um objeto é transferido de um ambiente para outro, uma conversão de tipos é necessária. Além disso, em algumas situações um objeto pode perder sua identidade se ele for transformado em uma cadeia de caracteres, por exemplo, e for transferido para um ambiente que não seja orientado a objetos, como ocorre com as linguagens C, Pascal, etc.

No Programa de Engenharia de Sistemas da COPPE/UFRJ vem sendo desenvolvido o Sistema de Gerência de Objetos GEOTABA [MATT89]. Seu objetivo é gerenciar os objetos contidos nos ambientes de desenvolvimento de software das estações de trabalho para engenharia de software TABA [ROCH90].

Um dos principais objetivos do GEOTABA é fornecer um ambiente onde as linguagens de definição de objetos, de manipulação de objetos e de programação de métodos sejam mais próximas umas das outras [MONT92]. Para isto, o modelo de objetos do GEOTABA foi projetado para, além de conter características de modelos de dados de SGBDs, ter o poder de expressão das linguagens de programação, para que ele possa ser implementado por uma verdadeira LPBD, no caso a linguagem DEMO - Definição e Manipulação de Objetos [MONT90].

Para validar e aperfeiçoar o modelo de objetos do GEOTABA, assim como experimentar idéias sobre a implementação de sua arquitetura, está sendo construido um protótipo, o ProtoGEO. O ProtoGEO concentra-se na implementação da gerência de armazenamento de objetos e na implementação das principais características do modelo de objetos do GEOTABA. Questões do tipo compartilhamento, concorrência, distribuição, segurança e reconstrução, não são contempladas no ProtoGEO.

Neste trabalho, pretendeu-se, além da realização de um estudo das diversas linguagens envolvidas na implementação de alguns SGBDOOs, desenvolver e implementar uma linguagem de programação orientada a objetos, que é a linguagem MANO - Linguagem de Manipulação de Objetos. A linguagem MANO será a base do modelo de objetos do GEOTABA. Por ser uma linguagem de programação com força expressiva, as demais partes do protótipo, como a linguagem de definição de dados, linguagem de consulta, etc, serão implementadas em MANO. Além disso, as aplicações

dos usuários também poderão ser implementadas em MANO, podendo dessa forma ser resolvido o problema de "*impedance mismatch*" dentro do sistema GEOTABA.

1.3 Estrutura da Tese

No caoítulo I - Introducão, são descritos os requisitos impostos pelas novas áreas de aplicação e os dois tipos de abordagens que estão sendo seguidas no desenvolvimento dos SGBDOOs. Em seguida é justificada a importância do trabalho realizado e os objetivos a que se pretendeu alcançar.

O caoítulo II - Banco de Dados Orientado a Objetos: Algumas Exueriências, apresenta o estudo realizado de alguns protótipos de SGBDOOs, todos já em fase de comercialização, onde procurou-se descrever as funções disponíveis mais relevantes, com ênfase nas linguagens de operação usadas por esses sistemas.

O capítulo 111 - ProtoGEO: Protótipo do SGBDOO GEOTABA descreve brevemente o modelo de objetos do GEOTABA [MONT90] e apresenta a Linguagem MANO, mostrando através de exemplos suas principais características. Por último, é mostrada a arquitetura adotada na implementação do ProtoGEO.

O capítulo IV - Processador MANO descreve o Processador da Linguagem MANO, mostrando as estruturas de dados que são usadas por ele e os tipos de instruções existentes. Ao final, é descrito o conjunto de rotinas primitivas que foram implementadas para que o sistema funcionasse com maior rapidez.

No capítulo V - Compilador MANO são descritos, detalhadamente, os diversos componentes do Compilador MANO e, também, é apresentada a sintaxe completa da linguagem MANO.

O capítulo VI - Gerente de Armazenamento apresenta uma proposta para a construção do módulo de armazenamento de objetos do ProtoGEO e descreve a versão

provisória do Gerente do "Buffer" de Objetos, que foi implementada para que a linguagem MANO pudesse ser utilizada.

O cavítulo VII - Conclusões faz uma análise dos objetivos traçados e dos resultados obtidos, propondo alguns tópicos que poderão gerar futuras pesquisas envolvendo *o* protótipo ProtoGEO.

Capítulo II

Banco de Dados Orientado a Objetos: Algumas Experiências

II.1 Introdução

Os Sistemas de Gerência de Objetos **(SGOs)**, chamados assim por englobar tanto os SGBDs orientados a objetos como os SGBDs relacionais estendidos, surgiram da necessidade de atender aos requisitos impostos pelas aplicações das áreas de CAD, CASE, OIS entre outras.

Os SGBDs orientados a objetos e os SGBDs relacionais estendidos têm como objetivo comum o gerenciamento de objetos através da aplicação de conceitos trazidos do paradigma da orientação a objetos. Apesar de existirem algumas variações na forma em que esses conceitos foram empregados, um conjunto básico de características podem ser encontradas na maioria dos sistemas. Essas características são: o suporte a objetos complexos, identidade de objetos, encapsulamento de dados e procedimentos, objetos compostos, relacionamentos entre objetos, suporte para dados multimídia, herança de propriedades dentro de uma hierarquia de classes, versões, etc, além da manutenção das características existentes nos SGBDs relacionais, como armazenamento de grandes volumes de dados persistentes, linguagem de consulta, concorrência, transações, reconstrução, independência de dados, etc.

A diferença básica entre estes dois tipos de sistemas é que nos SGBDs relacionais estendidos, procurou-se estender as linguagens de consulta dos já bem conhecidos sistemas relacionais para incorporarem procedimentos e outras características de gerenciamento de objetos. Neste caso, dois ambientes de programação separados são fornecidos pelo sistema, o ambiente da linguagem de consulta estendida e o ambiente da linguagem de programação onde são desenvolvidas as aplicações. Apesar de as duas linguagens serem capazes de chamar uma à outra, sempre existem dois ambientes de execução com sistemas de tipos separados, o que resulta na necessidade de um procedimento de conversão de tipos quando na transferência de dados entre os ambientes.

Já os SGBDs orientados a objetos tiveram sua origem a partir da extensão de alguma linguagem de programação orientada a objetos existente, como Smalltalk [GOLD83], C++ [STRO86] ou CLOS [MOON89], para suportar características de sistemas de gerência de banco de dados. Neste caso, as linguagens de operação do banco de dados e de programação são executadas no ambiente da aplicação, compartilhando o mesmo sistema de tipos [CATT91].

Existem outras diferenças entre os SGBDs orientados a objetos e os SGBDs relacionais estendidos, entre elas o modelo de objetos adotado, a sintaxe da linguagem de consulta, o tipo de objeto resultante de uma consulta, etc.

Este trabalho concentrou-se principalmente, no estudo de SGBDs orientados a objetos, com ênfase nas linguagens de operação de banco de dados. Esta escolha deve-se ao fato do Sistema GEOTABA ser um SGBD com as características de um SGBDOO, e *o* seu modelo de objetos [MONT90] apresentar diversas características do paradigma da orientação a objetos.

Nas seções seguintes, apresentam-se as principais características encontradas nos SGBDOOs O_2, ORION e GemStone. Estes três sistemas foram escolhidos inicialmente, pela grande quantidade de publicações técnicas disponíveis. Além disso, da mesma forma que o GEOTABA, tais sistemas seguiram a abordagem de estender uma linguagem de programação orientada a objetos com as características existentes nos bancos de dados tradicionais. No sistema O_2, foi desenvolvida uma versão própria da linguagem C, o

sistema ORION utilizou como ponto de partida a linguagem LISP e o sistema GemStone foi implementado como uma extensão da linguagem Smalltalk.

11.2 Sistema O_2

Foi projetado e desenvolvido pelo consórcio de pesquisa francês chamado Altaïr, que é formado pelo INRIA (Instituto Nacional de Pesquisas em Informática e Automação), Siemens-Nixdorf, Bull, CNRS e Universidade de Paris XI.

O Projeto Altaïr foi iniciado em setembro de 1986 com o objetivo de desenvolver um SGBD da próxima geração. Primeiro construiu-se um protótipo descartável, concluído em dezembro de 1987 [BANC88]. A segunda versão do O_2 ficou pronta em março de 1989, e está descrita em [DEUX90]. O Projeto Altaïr foi encerrado em setembro de 1990 *e* agora *o* Sistema O_2 está sendo comercializado pela empresa O_2 Technology.

O Sistema O_2 foi projetado para atender a diversas áreas de aplicação não convencionais. Segundo O. Deux [DEUX91], os principais objetivos, que nortearam o desenvolvimento do sistema, foram:

- Aumentar a produtividade no desenvolvimento de aplicações convencionais e não convencionais;
- Fornecer melhores ferramentas para o desenvolvimento de aplicações;
- Melhorar a qualidade final dessas aplicações.

Para alcançar esses objetivos, o Sistema O_2 misturou as tecnologias de banco de dados, linguagens de programação e interface com usuário, empregando conceitos do paradigma da orientação a objetos.

A versão mais recente da arquitetura funcional do sistema, apresentada em [DEUX91], está esboçada na figura 2.1. A parte principal do sistema (O_2Engine) é composta de uma máquina de banco de dados de objetos. O O_2Engine tem a função de

gerenciar o armazenamento de objetos estruturados e multimídia. É de sua responsabilidade o gerenciamento de disco (utilizando "buffers", índices, agrupamento e I/O), transações, concorrência, reconstrução, distribuição, segurança e administração de dados.

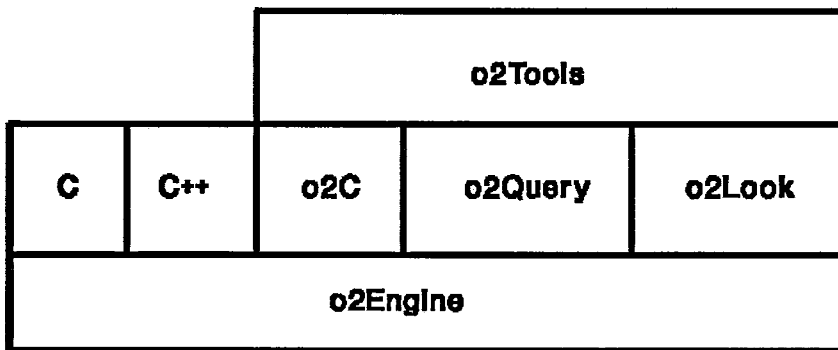


Figura 2.1: Arquitetura Funcional do O_2

O Sistema O_2 fornece dois tipos de interfaces: o ambiente O_2 e interfaces para linguagens de programação.

O ambiente O_2 é formado por um conjunto de ferramentas, entre elas, uma linguagem de consulta (O_2Query), um gerador de interface com usuário (O_2Look), uma linguagem de quarta geração (O_2C) e um ambiente gráfico de programação (O_2Tools) que inclui um depurador, um navegador de esquema e um navegador de banco de dados.

O conjunto de linguagens de programação, previstas para servirem de interface com o O_2, sofreu algumas modificações. Inicialmente previa-se que os métodos seriam escritos em C, Basic ou Lisp [BANC88]. Posteriormente, este conjunto reduziu-se para C e Basic [LÉCL89] e atualmente o conjunto é formado pelas linguagens C e C++ [DEUX91].

O Sistema O_2 segue uma abordagem multi-linguagem, ou seja, além de fornecer as linguagens C e C++ para a programação de aplicações, o sistema pode ser programado a partir da linguagem O_2C. O. Deux [DEUX91] classifica a linguagem O_2C como sendo uma linguagem de quarta geração, por permitir que o usuário realize manipulação do banco de dados, programação de métodos e geração de interface com usuário.

Segundo Lécluse [LÉCL89], inicialmente o sistema era programado da seguinte forma: primeiro o usuário definia as classes, usando comandos O_2 (semelhante a uma linguagem de definição de dados), depois os métodos eram programados nas linguagens $BasicO_2$ ou CO_2 (que mais tarde transformou-se na linguagem O_2C). CO_2 é uma derivação da linguagem C que permite a manipulação de objetos O_2, enquanto $BasicO_2$ é uma derivação da linguagem Basic.

O casamento da linguagem de programação CO_2 com a linguagem de definição de classes (comandos O_2) e também recursos para geração de interface com usuário deram origem a linguagem O_2C, que pode ser também classificada como uma linguagem de programação de banco de dados [ATKI87].

A seguir são descritas algumas características do modelo de objetos do sistema O_2 que influenciaram no projeto dessas linguagens.

Modelo de Objetos do O_2

Em [LÉCL88] apresenta-se o modelo de dados orientado a objetos utilizado no sistema O_2. Diferente do modelo usado pela linguagem Smalltalk, onde mesmo os

valores atômicos são objetos, o sistema O_2 fornece aos usuários dois conceitos distintos, objetos e valores.

Um objeto O_2 possui identidade, encapsula valores e métodos e é instância de uma classe. Os valores são descritos por seus tipos, existem os tipos atômicos que são: boolean, integer, real, character, string e bits. Além disso, podem ser definidos tipos complexos, usando recursivamente os construtores tupla (tuple), lista (list) e conjunto (set). Cada valor é instância de um tipo.

Os objetos são manipulados através de métodos, que podem ser públicos (visíveis fora da classe) ou privados (visíveis somente por outros métodos da classe). Já os valores não são encapsulados e podem ser manipulados por operadores primitivos.

As classes são organizadas em uma hierarquia, enquanto os tipos somente aparecem como componentes das classes. A parte estrutural de uma classe é definida através de um tipo, e um conjunto de métodos define a parte comportamental. A figura 2.2 exemplifica a definição de classes em O_2C.

```
class City
   type tuple(name:string,
           map:Bitmap,
           hotels:set(Hotel))

   method how_many vacancies(star:integer):integer,
        built_new_hotel(h:Hotel)
end;

class Hotel
   type tuple(name:string,
           read stars:integer,
           read free_rooms:integer)
   method reserve_room:boolean,
        check_out
end
```

Figura 2.2: Definição de classes em O_2C

(fonte [DEUX91])

A figura mostra as declarações das classes City e Hotel. A classe City possui três variáveis de instância: ***name*** - cadeia de caracteres correspondente ao nome da cidade, map - armazena uma figura ("bitmap") contendo o mapa da cidade e hotel - contém o conjunto de hotéis existentes na cidade. A classe City também inclui a declaração dos métodos ***how_many_vacancies***, que retorna o número de vagas existentes em hotéis de ***n*** estrelas, onde ***n*** é um valor passado como parâmetro e built_***new_hotel***, que permite a inclusão de um novo hotel no conjunto de hotéis da cidade.

A classe Hotel também contém três variáveis de instância: ***name*** - nome do hotel, ***stars*** - número de estrelas e ***free_rooms*** - número de quartos vagos, além dos métodos reserve_room, que devolve um valor booleano indicando se o quarto está vago e ***check_out***, que permite que um quarto seja desocupado.

A implementação do método é feita separadamente da sua especificação, que aparece junto da declaração da classe. A figura 2.3 ilustra o corpo do método ***how_many_vacancies*** pertencente à classe City, escrito em O_2C.

```
method body how_many_vacancies(star:integer):integer
in class City {
   int number = 0;
   02 Hotel h;
   for (h in self->hotels where h->free_rooms > 0 &&
      h->stars == star)
     number += h->free_rooms;
   return number;
}
```

Figura 2.3: Corpo de método em O_2C

(fonte [DEUX91])

O_2 fornece um mecanismo de herança, onde métodos e atributos de uma classe podem ser herdados de suas superclasses. O mecanismo de herança usado no O_2 é baseado em subtipos. Segundo [LÉCL89], um tipo *6* subtipo de outro se e somente se toda instância desse tipo é também uma instância do seu supertipo. Na definição de uma subclasse o usuário precisa fornecer apenas os novos atributos e os atributos redefinidos.

O sistema também permite múltipla herança. A resolução do problema de conflitos de nomes é feita pelo usuário, que tem de **redefinir** explicitamente o nome de atributos e métodos quando necessário.

Outras Características do O_2

Em O_2, tanto os objetos como os valores podem tomar-se persistentes. A persistência é conseguida através da atribuição de nomes a objetos ou valores. Um nome pode ser visto como uma variável global que faz referência ao objeto para tomá-lo persistente. As regras de persistência empregadas são as seguintes:

1. todo objeto ou valor nomeado é persistente,
2. todo objeto ou valor que é parte de outro objeto ou valor persistente também é persistente.

Uma característica existente no O_2, e pelo que foi estudado até o momento só existe no O_2, é a possibilidade de definição de atributos e métodos excepcionais.

Um atributo extra pode ser adicionado a um objeto ou valor do tipo tupla. Este atributo não pode ser manipulado por métodos associados à classe do objeto. Ao invés disto, ele é manipulado por operadores disponíveis em tuplas. Atributos extras são possíveis para objetos e valores do tipo tupla, independentemente de terem sido nomeados.

Aos objetos nomeados, pode-se adicionar métodos específicos. Porém esses métodos ficam associados ao nome e não ao objeto. Se um outro objeto for associado ao nome, então o método extra poderá ser aplicado ao novo objeto. Segundo [LÉCL89], esses métodos são usados para caracterizar comportamento excepcional de um objeto.

O_2C é uma linguagem fortemente tipada, onde todo objeto é uma instância de um tipo, além disso todo atributo é declarado como sendo de algum tipo. O Sistema O_2 utiliza um verificador de tipos em tempo de compilação. Em uma variável só podem ser

atribuídos valores (ou objetos) do tipo (ou classe) que foi declarado ou de algum de seus subtipos (ou subclasses).

Como o sistema permite que sejam feitas dinamicamente modificações pelo usuário, algumas verificações de tipos são realizadas em tempo de execução.

Como na maioria dos sistemas orientados a objetos, a seleção do método a ser executado depende do tipo do objeto receptor da mensagem. Para isto, o sistema O_2 pode fazer o acoplamento dos métodos de forma dinâmica (late *binding*), quando o método não pode ser determinado em tempo de compilação.

Interfaces para Programação de Aplicações

Uma das metas do Sistema O_2 é ser aberto para o uso de diferentes linguagens, **utilizando** para isto, um modelo de dados que é independente de linguagem.

Na versão atual do O_2 as aplicações podem ser programadas usando o ambiente O_2, através de sua linguagem de programação de banco de dados O_2C, ou através das linguagens de programação C e C++.

Utilizando-se uma linguagem de programação, C++, por exemplo, o sistema O_2 pode ser visto de duas maneiras: classes podem ser exportadas do sistema O_2 para o ambiente C++, ou uma aplicação C++ pode tornar objetos persistentes, fazendo com que suas classes sejam importadas pelo sistema O_2.

Quando o esquema está definido no O_2, o comando ***export*** to C++ gera classes C++ e, então, objetos O_2 podem ser manipulados dentro do ambiente C++. Neste caso o sistema gera dois métodos escritos em C++, um para ler e outro para gravar objetos de ***e*** para o banco de dados. Depois que uma classe é gerada no ambiente C++, novos atributos **e/ou** métodos podem ser adicionados a ela, porém esses novos atributos não são persistentes.

Se as classes estiverem definidas no ambiente C++, o sistema O_2 pode ser usado **para** que objetos dessas classes tornem-se persistentes. Isto é feito utilizando-se o

utilitário ***import from*** C++ que, a partir de análise da classe C++, gera dois métodos 02_*read* e 02_*write*, que devem ser usados na aplicação C++, para ler e gravar objetos no sistema O_2.

Arquitetura do O_2Engine

Como foi dito inicialmente, o O_2Engine é a parte do sistema responsável pelo gerenciamento de objetos. O_2Engine é composto de três camadas principais: o Gerente de Esquema, o Gerente de Objetos e o Gerente de Disco.

O Gerente de Esquema é responsável pela manipulação de classes, métodos e atribuição de nomes a objetos e valores. Além disso, ele gerencia a interface com as linguagens de programação C e C++, através do controle da importação e exportação de esquemas.

O Gerente de Objetos é responsável por fornecer identidades a objetos, por despachar mensagens e gerenciar valores e seus operadores. Adicionalmente, ele controla a persistência de objetos, a coleta de lixo (***garbage collect***) e implementa estratégias de índices e agrupamentos, baseado em objetos complexos e herança.

Os objetos no sistema O_2 são acessados através de identificadores, que lhes são atribuídos no momento de sua criação e permanecem constantes durante toda a sua existência. Este sistema utiliza identificadores físicos, ao contrário dos sistemas ORION e GemStone, onde os identificadores são lógicos, isto é, não possuem nenhuma informação sobre a sua localização na memória secundária.

Segundo F. Velez [VELE89], identificadores físicos são usados para evitar a necessidade de uma tabela de correspondência entre o identificador lógico e o endereço físico. Um problema que surge neste esquema é quando existe a necessidade de se mover o objeto dentro do disco. Como o identificador do objeto não pode ser alterado, o sistema utiliza a técnica de apontar para frente, ou seja, o novo endereço do objeto é armazenado na antiga posição, permitindo assim a localização do objeto no novo endereço.

Como no sistema ORION, o sistema O_2 utiliza um esquema de gerenciamento de "buffer" duplo, ou seja, a memória principal é dividida em um "buffer" de páginas e um "buffer" de objetos. Os objetos no "buffer" de páginas são armazenados no formato de disco, enquanto os objetos no "buffer" de objetos estão armazenados no formato de memória. Quando o objeto requerido não está na memória, a página que contém o objeto é trazida do disco para o "buffer" de páginas e todos os objetos válidos desta página são copiados para o "buffer" de objetos. Esta estratégia de leitura à frente é adotada para aproveitar o fato de que os objetos estão agrupados.

O Compilador CO_2

Por falta de material publicado, nesta seção é feita a descrição do compilador CO_2 utilizado na versão protótipo do sistema O_2 e não do compilador O_2C. Como este assunto está diretamente ligado ao escopo deste trabalho, optou-se por descrevê-lo para enriquecer *o* levantamento bibliográfico.

O compilador CO_2 é usado pelo Gerente de Esquema para analisar as definições de classes (tipos) e métodos.

No sistema O_2, a unidade computacional é o método. Cada método é extraído de um programa fonte CO_2 e é convertido em uma função C. O compilador CO_2 transforma código CO_2 em código C e utiliza um compilador C para analisar cada método em separado. **A** figura 2.4 representa a arquitetura do compilador CO_2.

Para otimizar o processo de compilação, o sistema O_2 possui o Interpretador O_2, que é sempre ativado no início da compilação. *O* Interpretador é capaz de reconhecer um pequeno subconjunto da linguagem CO_2, como por exemplo o envio de mensagens. Se um método pode ser resolvido pelo interpretador, então ele não passa pelas etapas de compilação, sendo interpretado cada vez que for utilizado.

A precompilação é a primeira fase da compilação, propriamente dita. Nesta fase, uma análise sintática e semântica é realizada na parte de definições de tipos e uma função

C é gerada para cada método. O **precompilador** também verifica se o esquema do banco de dados está sendo preservado, quando da inclusão de novas classes ou métodos.

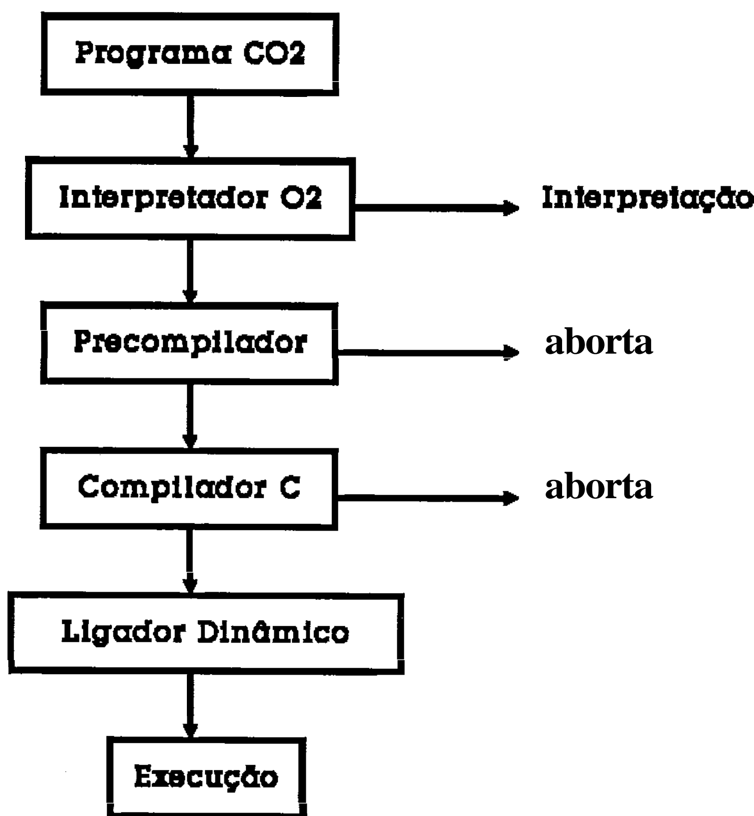


Figura 2.4 Arquitetura do Compilador C02

Se a fase de precompilação não gerar nenhum erro, o compilador C é ativado para gerar código objeto de cada função, correspondente aos métodos encontrados. Finalmente, um **ligador** dinâmico (UNIX BSD **4.3**) é usado, para ligar os códigos objetos gerados na fase de compilação.

O sistema O_2 armazena os métodos compilados no banco de dados, junto com os demais objetos.

Comentários Finais

O sistema O_2 teve, como um de seus objetivos, resolver *o* problema de incompatibilidade entre as linguagens de operação do sistema. Para isto, foi desenvolvida uma linguagem independente (O_2C) que permite ao usuário definir as classes, escrever os métodos e ainda gerar a interface com usuário. Uma aplicação pode ser totalmente implementada em O_2C, uma vez que a linguagem O_2C é uma linguagem com força expressiva *e* o usuário tem ainda a vantagem de estar utilizando um único sistema de tipos.

Caso tenha feito a escolha de desenvolver sua aplicação em uma das linguagens C ou C++, apesar da necessidade de conversão de tipos entre os ambientes, o usuário tem a vantagem de estar utilizando linguagens onde as sintaxes são bastante semelhantes, visto que a linguagem O_2C é uma derivação da linguagem C.

O sistema O_2 foi implementado em C e roda em um ambiente de estações SUN interligadas em rede.

11.3 Sistema ORION

O sistema ORION é o resultado do desenvolvimento de um projeto que teve início no final do ano de 1985, no Advanced Computer Architecture Program do MCC em Austin/Texas. Atualmente, o sistema está sendo comercializado com o nome de ITASCA pela empresa Itasca Systems de Minneapolis, Minnesota.

O sistema utiliza o paradigma da orientação a objetos, apresentando características de banco de dados como persistência, suporte a transações e outras funções avançadas que são requisitos de aplicações de domínios como CAD/CAM, Inteligência Artificial e Sistemas de Automação de Escritório.

Entre as principais funções encontradas no ORION tem-se: controle de versões, suporte para objetos compostos, consultas, evolução dinâmica do esquema e gerenciamento de dados multimídia.

O Sistema ORION foi implementado em Common LISP. Seu modelo mais simples é um sistema monousuário e multitarefa, projetado para ser usado em uma máquina Symbolics 3600 LISP sob o sistema operacional Genera 70 ou em uma estação SUN-3 sob o sistema operacional UNIX.

Segundo W.Kim [KIM89c], a decisão de usar a linguagem Common LISP na implementação do ORION, se deu pela necessidade de obter compatibilidade com outros projetos, que estavam em desenvolvimento no MCC.

No ORION, a linguagem LISP é estendida para conter capacidades orientadas a objetos, fazendo chamadas ao banco de dados para acesso navegacional ou para consultas, além de ser usada na definição de tipos de dados.

Segundo J. Banerjee [BANE87a], existe uma integração entre o protocolo de envio de mensagens no ORION, com chamadas de funções LISP. Com isto, as

aplicações ORION podem acessar tanto os objetos ORION como as estruturas LISP, sem que haja a necessidade de transferir objetos entre os ambientes de programação. O sistema ORION não fornece suporte para múltiplas linguagens de programação no desenvolvimento de aplicações.

A seguir é feita uma breve descrição das principais características encontradas no sistema ORION.

Modelo de Objetos ORION

O modelo de objetos utilizado no sistema ORION [BANE87a] é muito semelhante ao modelo empregado na linguagem Smalltalk. No sistema ORION todas as entidades são modeladas como objetos. Mesmo as entidades mais simples, que só possuem um valor (ex. inteiro, real, char, etc) são modeladas em objetos, chamados de primitivos.

Cada objeto possui sua identidade própria e encapsula dados e comportamento através de métodos. Todo objeto é instância de uma classe. As classes estão organizadas em uma hierarquia, onde uma classe herda propriedades de suas superclasses.

Uma classe pode ter mais de uma superclasse direta, ou seja, o modelo permite herança múltipla. A resolução de conflitos de nomes, no sistema ORION, é feita dando prioridade à defmição dentro da classe sobre sua superclasse, e ordenando suas superclasses segundo uma ordem de precedência. Por exemplo, se um atributo ou método com o mesmo nome aparece em mais de uma superclasse direta, então por definição, o atributo ou método escolhido será aquele que constar da superclasse que estiver com maior prioridade na lista de precedência.

Arquitetura Funcional

Em [KIM90a] são apresentadas três arquiteturas para o sistema, são elas: ORION-1, ORION-1SX e ORION-2. O sistema ORION-1SX utiliza uma arquitetura cliente/servidor, enquanto o sistema ORION-2 é um sistema de gerenciamento de objetos distribuído. O sistema ORION-1 é um sistema monousuário e os outros dois modelos rodam em um ambiente de estações interligadas em rede. A arquitetura do ORION-1, especificada na figura 2.5, é composta de quatro módulos: Manipulador de Mensagens, Subsistema de Objetos, Subsistema de Transações e Subsistema de Armazenamento.

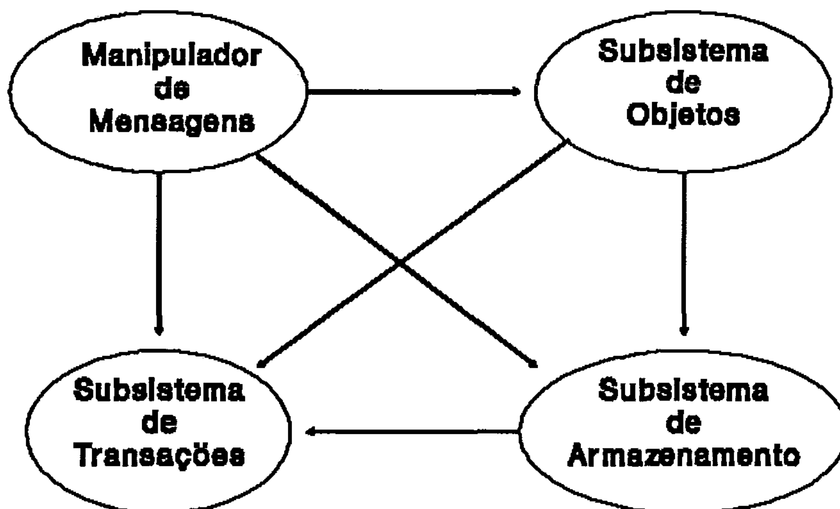


Figura 2.5: Arquitetura Funcional do ORION-1

Todas as mensagens enviadas ao sistema ORION são recebidas e despachadas pelo Manipulador de Mensagens. Existem três categorias de mensagens: mensagens definidas pelo usuário, que ativam métodos já armazenados no banco de dados;

mensagens de acesso, que possibilitam acesso e alteração dos valores dos atributos de um objeto; funções definidas pelo sistema, como por exemplo, mensagens para definição do esquema, criação e deleção de instâncias, gerenciamento de transações, etc.

O Subsistema de Objetos é responsável por gerenciar funções como: evolução de esquema, controle de versões, otimização de consultas, objetos compostos e gerenciamento de informações multimídia (campos longos).

O Subsistema de Transações é responsável pelos mecanismos de controle de concorrência e reconstrução.

O Subsistema de Armazenamento é responsável pela criação de objetos e pelo acesso e gravação de objetos no disco. Além disso, gerencia índices criados sobre atributos de uma classe, com o objetivo de melhorar o desempenho na avaliação de consultas.

Evolução Dinâmica de Esquema

Em domínios de aplicações como CAD/CAM, CASE e Sistemas de Automação de Escritório, são muito frequentes as alterações na definição das classes, ou seja, no esquema do banco de dados. O sistema ORION fornece um mecanismo que permite que o esquema seja alterado dinamicamente.

Segundo W. Kim [KIM90a], existem duas técnicas que podem ser empregadas: evolução de esquema e versões de esquema. A técnica de evolução de esquema, descrita em [BANE87b], consiste em permitir aos usuários realizarem todas as alterações sobre uma única cópia lógica do esquema. Na técnica de versões de esquema, descrita em [KIM88a], a alteração do esquema é feita através da criação de uma nova versão do esquema lógico. Neste caso, o usuário pode trabalhar com diferentes visões do esquema. No sistema ORION-1, a técnica implementada é a de evolução de esquema.

Dois tipos de alterações podem ser realizadas no esquema de um SGBDOO. O primeiro tipo é a alteração na definição da classe, ou seja, adição ou exclusão de atributos *e* métodos. O segundo tipo envolve alterações na hierarquia de classes, que

podem ser: a inclusão de uma nova classe, a exclusão de uma classe existente e a alteração no relacionamento classe/subclasse.

Algumas dessas alterações, como por exemplo, a inclusão ou exclusão de um atributo, afetam as instâncias existentes. Neste caso, duas abordagens para a correção dessas instâncias podem ser empregadas: alteração imediata ou alteração adiada. Na abordagem de alteração adiada, que é utilizada no sistema ORION, as instâncias só serão alteradas quando forem acessadas pela primeira vez, após a alteração no esquema. O banco de dados pode armazenar informações desnecessárias enquanto todas as instâncias não tiverem sido acessadas.

Na abordagem de alteração imediata todas as instâncias são corrigidas imediatamente, após a alteração do esquema. Neste caso, uma alteração de esquema pode consumir muito tempo, mas o banco de dados não manterá informações obsoletas.

Versões de Objetos

O sistema ORION possui um mecanismo que permite ao usuário criar diferentes versões de um objeto. Neste esquema, descrito em [CHOU86], existem dois tipos de versões: versões temporárias e versões de trabalho. Diversas versões temporárias podem ser criadas a partir de uma versão existente. Uma versão temporária pode ser alterada, apagada ou promovida a versão de trabalho, enquanto uma versão de trabalho só pode ser apagada.

Para que possam ser criadas versões de um objeto, sua classe deve ser declarada como sendo ***versonificada.*** Neste caso cada objeto dessa classe é um objeto dito genérico. Um objeto genérico mantém a história de derivação de suas versões.

A referência a um objeto ***versonificado*** pode ser feita a uma versão específica do objeto ou ao objeto genérico. Todo objeto genérico possui sua versão corrente ('default'), que é utilizada quando ele é referenciado.

Um objeto normalmente faz referência a outros objetos. Além disso, um objeto pode estar sendo referenciado por diversos outros objetos. Quando um objeto é alterado

ou apagado, ou uma nova versão do objeto é criada, então, os objetos que fazem referência a ele podem ficar inconsistentes e por isso precisam ser "avisados" da alteração. Junto com o mecanismo de versões, o sistema ORION implementa um mecanismo de notificação de alteração, que é baseado em "time stamps".

Um levantamento dos mecanismos de controle de versões em SGBDOOs é apresentado em [LISB90].

Objetos Compostos

O sistema ORION fornece suporte para tratamento de objetos compostos [KIM89b]. Um objeto composto é formado por uma hierarquia de objetos componentes. Um objeto componente não pode ser compartilhado entre objetos compostos, ou seja, um objeto componente pertence a no máximo um objeto composto, além disso, sua existência depende da existência do objeto composto.

O sistema trata um objeto composto como uma unidade para integridade semântica, agrupamento físico e bloqueio. Se um objeto composto for eliminado, todos os seus objetos componentes também serão eliminados.

O modelo de dados orientado a objetos não captura o relacionamento É_PARTE_DE entre objetos. Um objeto pode referenciar outros objetos, porém, essa referência não implica que ele "contém" os outros objetos. Uma hierarquia de objetos permite capturar o relacionamento É_PARTE–DE entre uma classe e suas classes componentes, enquanto uma hierarquia de classes representa o relacionamento É_UM entre uma superclasse e suas subclasses.

O sistema ORION ainda suporta evolução de esquema em uma hierarquia de objetos compostos.

Gerenciamento de Objetos

O gerenciamento de objetos é feito pelo Subsistema de Armazenamento, que fornece acesso aos objetos do banco de dados. O Subsistema de Armazenamento, como mostra a **figura** 2.6, é composto de quatro módulos: Gerente de Acesso, Gerente de Armazenamento, Gerente do "Buffer" de Objetos e Gerente do "Buffer" de Páginas.

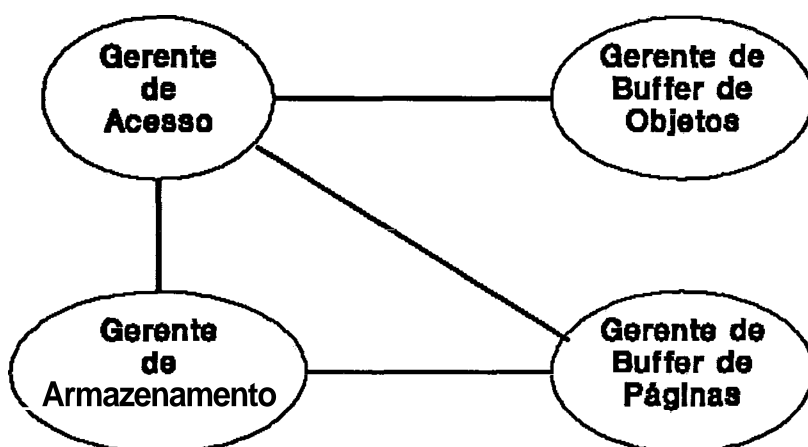


Figura 2.6: Subsistema de Armazenamento do ORION-1

O gerenciamento dos objetos na memória principal é feito, baseado no sistema LOOM [KAEH83], dividindo-se o espaço de memória em um "buffer" de objetos e um "buffer" de páginas. Segundo W. **Kim [KIM90a],** a justificativa para adotar esse esquema, é que aplicações não convencionais tendem a carregar um grande número de objetos na memória para manipulá-los. Em um esquema simples contendo apenas o "buffer" de páginas, um volume grande de objetos ficaria carregado na memória

desnecessariamente. Com o esquema de "buffer" duplo a memória é melhor aproveitada. Neste esquema, quando um objeto precisa ser acessado, primeiro a página que contém o objeto é copiada do disco para o "buffer" de páginas, depois o objeto é localizado na página, recuperado e copiado para o "buffer" de objetos.

O Gerente de "Buffer" de Objetos mantém uma tabela de objetos residentes (ROT), que permite a localização de objetos dentro do "buffer" de objetos. Cada objeto presente na memória possui uma estrutura chamada Descritor de Objeto Residente (ROD) onde estão armazenadas informações como: identificador lógico, endereço da sua localização física no disco, um ponteiro para o ROD da sua classe, um ponteiro para o objeto na memória, etc.

Os objetos são manipulados em dois formatos distintos: no formato de disco e no formato de memória. Os objetos são armazenados no formato de disco por motivos de eficiência no armazenamento e recuperação, e são transformados para o formato de memória, para permitir que as aplicações manipulem os objetos de acordo com as necessidades das linguagens de programação.

O Gerente de Acesso é o responsável pela manipulação de objetos. É ele que verifica se um objeto está em disco, no "buffer" de páginas ou no "buffer" de objetos. Além disso, ele transforma os objetos do formato de disco para o formato de memória e vice-versa, e controla a transferência de objetos entre o "buffer" de páginas e o "buffer" de objetos.

O Gerente de Armazenamento controla o espaço em disco. O disco é dividido em segmentos, que por sua vez são divididos em páginas. O sistema ORION suporta apenas um esquema simples de agrupamento, no qual as instâncias de uma mesma classe são armazenadas no mesmo segmento físico [BANE88b]. Uma variação desse esquema é utilizada no armazenamento de objetos compostos, onde os componentes de um objeto composto são armazenados juntos no mesmo segmento.

Outra característica importante de banco de dados, encontrada no sistema ORION, é a manutenção de índices. Os índices podem ser defmidos em uma classe ou em subclasses dessa classe e são usados pelo processador de consultas do ORION [KIM89d].

Considerações Finais

O sistema ORION foi implementado em Common LISP, uma extensão orientada a objetos da linguagem LISP. Através de chamadas de funções LISP, as aplicações podem acessar os objetos ORION sem a necessidade de transferência dos objetos ORION entre os ambientes de programação e do banco de dados.

Segundo [CATT91], a versão comercial do ORION chamada ITASCA, inclui também uma interface C, além de ferramentas para o usuário final. Visando simplificar esta apresentação e por falta de material publicado, esta interface com a linguagem C não foi coberta neste trabalho.

11.4 Sistema GemStone

GemStone é um SGBDOO desenvolvido pela Servio Logic Corporation. O projeto iniciou em 1983 com o objetivo de transformar a linguagem Smalltalk em um SGBDOO [COPE84]. Foi um dos primeiros SGBDOOs a se tornar um produto comercializável.

O sistema segue uma abordagem orientada a objetos e foi desenvolvido para satisfazer as necessidades de aplicações tanto da área comercial como da área de engenharia.

GemStone suporta múltiplas linguagens, fornecendo interfaces para a programação de aplicações nas linguagens C, C++ e Smalltalk, que estão descritas no final desta seção.

O sistema utiliza a linguagem OPAL para operação do banco de dados. OPAL é usada para definição de esquema, manipulação de objetos e programação em geral. A linguagem OPAL foi desenvolvida para o sistema GemStone com o objetivo de eliminar algumas limitações impostas pela linguagem Smalltalk.

A primeira versão do sistema foi projetada para rodar em um ambiente composto de uma máquina VAX da DEC interligada por uma rede local com vários PCs IBM. Com a liberação de novas versões algumas mudanças foram realizadas na arquitetura do sistema. A evolução das versões da arquitetura do sistema está descrita mais adiante.

Modelo de Dados GemStone

O modelo de dados usado no sistema GemStone é totalmente baseado no modelo Smalltalk com adição de algumas características. O modelo Smalltalk é baseado em três conceitos: objeto, mensagem e classe [GOLD83]. Cada objeto possui sua identidade e uma memória privada, onde são armazenados os seus dados. Esses dados (variáveis de instância) só podem ser acessados através de métodos ativados por mensagens enviadas

ao objeto. Todas as entidades, inclusive valores primitivos (inteiros, caracteres, etc) são modeladas como objetos.

Os objetos são descritos por suas classes, as **quais** estão organizadas em uma estrutura hierárquica que permite que uma subclasse herde propriedades de sua superclasse. O sistema GemStone fornece apenas mecanismos de herança simples, ou seja, cada classe possui no máximo uma superclasse. As classes também são objetos e por sua vez, são instâncias de classes chamadas metaclasses.

No modelo Smalitalk não existe o conceito de tipos. Cada variável de instância de um objeto armazena a identidade de outro objeto e um objeto pode não ter variáveis de instância.

A linguagem OPAL suporta tipos para as variáveis de instância de subclasses das classes ***Bag*** e Set. Na declaração dessas subclasses, uma variável de instância pode ser declarada como sendo de uma Classe-Tipo, isto significa que essas variáveis só podem conter o **valor** nulo ou uma instância da classe ou subclasse da classe especificada (Classe_***Tipo).*** Estas restrições de Classe-Tipo são herdadas através da hierarquia de classes ***e*** podem ser redefinidas em uma subclasse.

Além de tipos, a linguagem OPAL permite o uso de expressão caminho, que é formada por um nome de uma variável seguido de zero ou mais nomes de variáveis de instância, chamados de **ligações.** Uma expressão caminho pode ser usada em qualquer lugar onde é permitida uma expressão, mas, normalmente, aparecem como seleções em consultas associativas **[BRET89].** Expressão caminho é uma forma de acesso não encapsulado do estado privado dos objetos e é usada em consultas por questões de eficiência.

A Linguagem OPAL

Apesar do modelo Smalltalk ser totalmente utilizado no sistema GemStone, a linguagem **Smalltalk** por sua vez, possui algumas deficiências que levaram ao desenvolvimento de uma nova linguagem para ser usada no sistema. Duas dessas

deficiências são: a quantidade e o tamanho dos objetos são limitados pelo tamanho da memória principal ou virtual. O sistema LOOM [KAEH83] permite um maior número de objetos, mas a limitação de tamanho permanece. Smalltalk é um sistema monousuário e não satisfaz alguns requisitos de sistemas de banco de dados como: acesso concorrente por múltiplos usuários, suporte a mecanismos de reconstrução, autorização e estruturas auxiliares.

A linguagem OPAL foi desenvolvida para suprir essas necessidades. O sistema GemStone gerencia múltiplos espaços de nomes. Cada usuário possui seu dicionário de variáveis globais, que é a raiz do mecanismo de persistência de objetos, além de poder acessar dicionários globais que são compartilhados entre os usuários.

Cada usuário conecta-se com o sistema através de uma sessão GemStone e possui seu espaço de trabalho individual. Um objeto alterado só é visível às outras sessões, quando for salvo ("commited") no banco de dados.

Arquitetura do Sistema GemStone

A arquitetura do sistema GemStone é composta de dois componentes básicos: o servidor de processos Gem e o monitor Stone. O servidor Gem é responsável pela execução dos métodos, que são ativados a partir de mensagens OPAL. Ele controla os objetos presentes na memória utilizando um esquema de "buffer" duplo, ou seja, um "buffer" de objetos e um "buffer" de páginas, como nos sistemas O_2 e ORION. O monitor Stone é responsável pelo gerenciamento dos objetos armazenados em disco. Ele também controla operações como: concorrência, controle de transações, autorização, reconstrução, etc. O monitor Stone coordena as atividades de diversos servidores Gem, um para cada usuário.

A primeira versão (1.0) do sistema roda em um ambiente formado por uma rede de máquinas VAX, da DEC, sobre o sistema VMS. Além disso, as aplicações podem rodar em PCs IBM interligados ao VAX através de uma rede local. A figura 2.7 mostra uma configuração possível do sistema GemStone, em sua primeira versão.

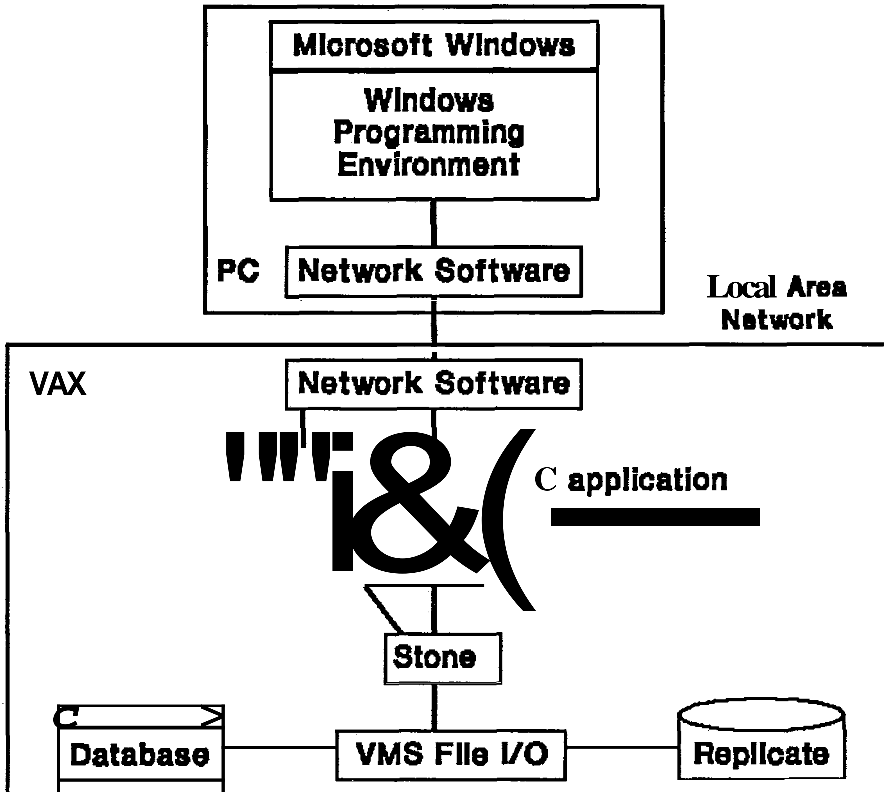


Figura 2.7: Sistema GemStone Versão 1.0

Nesta versão, o sistema apresenta caracteristicas como índices, agrupamentos, facilidades de consultas "ad hoc" e gerenciamento de campos longos.

Na versão 1.0 do sistema, além da linguagem OPAL, as aplicações também podem ser escritas na linguagem C, como será visto mais adiante.

As definições das classes, a hierarquia de classes e os métodos são armazenados dentro do banco de dados junto com os demais objetos gerenciados pelo sistema. Essas informações ficam disponíveis em tempo de execução, o que, segundo P. Butterworth **[BUTT91],** é referenciado como "dicionário de dados ativo".

A segunda versão (2.0) do sistema pode rodar em diferentes estações de trabalho, entre elas: **Sun3, Sun4,** Sony NEWS, **DECstation** e RS 6000. Além disso foram

incorporadas interfaces para aplicações escritas em C++ e Smalltalk (Smalltalk-80 e Smalltalk/V), que podem rodar também em máquinas Macintosh.

Nesta versão foram introduzidas melhorias para aumentar o desempenho do sistema, como por exemplo, alterações no algoritmo de coleta de lixo e a introdução de um protocolo de bloqueio de objetos.

A terceira versão (2.5) tem como principal característica o suporte para armazenamento multivolume. Os objetos são armazenados em mais de um volume de disco e estes discos podem ser controlados por diferentes máquinas.

Nesta versão o gerenciamento de "buffer" foi alterado para aumentar o desempenho do sistema. Diferentemente das versões anteriores, onde um objeto recuperado em disco era carregado no "buffer" de páginas e no "buffer" de objetos, agora um objeto recuperado é carregado apenas no "buffer" de páginas. O "buffer" de objetos não foi eliminado sendo usado, apenas, para os objetos criados pelas transações. Uma coleta de lixo preventiva é realizada para evitar que objetos temporários sejam transferidos para disco. Outra novidade na versão 2.5 é a introdução de um mecanismo de controle de versões de objetos.

Interfaces para Aplicações

Como foi dito anteriormente, o sistema GemStone suporta múltiplas linguagens. A seguir são descritas as interfaces para programação de aplicações nas linguagens C, C++ e Smalltalk.

Interface GemStone - C

A interface do sistema GemStone com a linguagem C é composta de uma biblioteca de funções C, que permitem uma ligação entre um programa de aplicação

escrito em C e o banco de dados. O acesso aos objetos pode ser feito estruturalmente via comandos C ou através do envio de mensagens OPAL.

Uma vez que a linguagem C não possui o conceito de objeto, todos os objetos importados do banco de dados GemStone precisam ser transformados em elementos que a linguagem C reconhece, como apontadores, cadeia de caracteres, inteiros, etc.

Algumas das funções disponíveis na interface GemStone - C são: (1) criar novos objetos no banco de dados; (2) acessar e alterar os campos dos objetos armazenados no banco de dados; (3) fazer a transformação da representação de valores primitivos, quando os objetos mudam de ambientes; (4) transferir objetos chamados "objeto relatório", que são estruturas de dados que contém informações sobre a identidade do objeto, tamanho da classe e os valores das variáveis de instância do objeto transferido.

Outro modo de acessar o banco de dados a partir de uma aplicação C é através do uso de comandos da linguagem OPAL. Desta forma, uma aplicação pode por exemplo, criar uma nova classe, definir um novo método, executar expressões e enviar mensagens para objetos no sistema GemStone.

A linguagem C também pode ser usada para escrever rotinas primitivas que são usadas pelo sistema GemStone.

Interface GemStone - C++

A interface para a linguagem C++ pode ser usada para possibilitar o armazenamento persistente de objetos C++ e para permitir o acesso a objetos armazenados no banco de dados por aplicações C++. Um objeto C++, armazenado no sistema GemStone, possui identidade própria e sua existência independe do programa que o criou, além disso, pode ser acessado por outras aplicações que tenham acesso ao sistema.

A interface C++, que é um preprocessador baseado na sintaxe C++ padrão, é usada através de chamadas de procedimentos. Ela também inclui uma biblioteca de classes contendo algumas classes mais usadas, como por exemplo conjuntos, arranjos e

listas. A interface também fornece funções para manipular objetos GemStone com código C++.

As classes C++ são incluídas no sistema GemStone através da edição de um arquivo que contém a declaração da classe, escrita em C++. Este arquivo é submetido a um utilitário chamado "Registrar", que, além de armazenar a definição da classe no banco de dados, gera código que permite o **mapeamento** de uma classe C++ em sua classe GemStone correspondente.

Um programa de aplicação escrito em C++ pode acessar os objetos GemStone de duas maneiras. A primeira é através de um ***GPointer***, que é um identificador de objeto que permite a localização do objeto na memória via tabela de "hash". A segunda maneira é através de um ***DPointer***, que é um ponteiro que aponta diretamente para a posição do objeto dentro da memória virtual. Aplicações C e C++ podem facilmente violar o mecanismo de encapsulamento de dados do sistema GemStone.

Interface GemStone - Smalltalk

Pelo fato da linguagem OPAL ser derivada da linguagem Smalltalk, a interface **GemStone-Smalltalk** é mais integrada do que as interfaces C e C++. Isto ocorre porque, neste caso, não há a necessidade de transformação de objetos quando estes migram de um ambiente para outro.

Esta interface é formada por um conjunto de classes instaladas no ambiente Smalltalk, que permite **às** aplicações acessarem objetos do banco de dados. Estas classes são: ***GSSession***, GSObject, ***GSObjectTraversal*** e ***GSObjectReport***.

Uma instância da classe ***GSSession*** representa uma sessão no banco de dados GemStone. Esta classe possui no seu protocolo, funções (métodos) para controle sobre a conexão com o sistema GemStone, gerenciamento de transações e gerenciamento de sessões cooperativas. Os objetos importados do banco de dados são mantidos no ambiente Smalltalk como instâncias da classe GSObject. Estes objetos na verdade, são cópias criadas automaticamente quando o objeto é recuperado no banco de dados.

GSObjectTraversal é uma classe cujas instâncias contém a descrição dos objetos importados. E finalmente, as instâncias da classe *GSObjectReport* armazenam informações sobre objetos GemStone e seus valores.

Um objeto transferido do sistema GemStone para o ambiente Smailtalk, depois de ter sido alterado pode ser salvo no banco de dados. Este procedimento pode ser solicitado explicitamente pela aplicação ou pode ser executado automaticamente pelo sistema.

Segundo R.G.G. Cattel [CATT91], a linguagem OPAL é similar à linguagem Smalltalk o suficiente para que programas Smalltalk possam rodar em um ambiente OPAL com poucas alterações.

Alteração de Esquema no GemStone

O sistema GemStone utiliza uma abordagem de modificação de esquema que é diferente do sistema ORION [BANE87b]. No sistema ORION a correção das instâncias de uma classe alterada é feita somente no momento em que o objeto for acessado. O sistema GemStone segue a abordagem de correção imediata, na qual todas as instâncias da classe alterada são corrigidas imediatamente após a alteração ter sido realizada.

Segundo Penney [PENN87], as duas abordagens possuem suas vantagens e desvantagens. Enquanto a abordagem de alteração adiada acrescenta um "overhead" para decidir se uma instância precisa ser alterada e alterá-la, a abordagem de alteração imediata tem o custo de corrigir todas as instâncias no momento da modificação do esquema.

Segundo R. Bretl [BRET89], uma das metas do projeto é desenvolver um mecanismo no qual as duas abordagens possam ser usadas no momento apropriado, ou seja, possibilitar a utilização de um mecanismo híbrido.

As operações que podem ser realizadas no esquema do banco de dados GemStone são:

. Adicionar uma variável de instância

. Remover uma variável de instância

. Renomear uma variável de instância

. Tornar uma classe indexável

. Tornar uma classe não indexável

. Modificar a restrição (Classe-Tipo) de uma variável de instância

. Adicionar uma nova classe

. Remover uma classe

Considerações Finais

No sistema GemStone o problema de incompatibilidade entre linguagens de operação do banco de dados ("impedance *mismatch*") é resolvido pela semelhança entre as linguagens.

As interfaces C e C++ permitem o acesso aos dados de um objeto de forma estrutural, ou seja, sem a necessidade de utilizar métodos, violando dessa forma o mecanismo de encapsulamento.

O acesso de objetos através de expressões caminho também permite que uma aplicação viole *o* mecanismo de encapsulamento do sistema.

Por não utilizar uma linguagem fortemente tipada, o sistema GemStone apresenta facilidades na prototipação de sistemas complexos, uma vez que nestes sistemas é comum não se saber de início o tipo (classe) dos objetos que uma variável de instância irá conter.

Capítulo III

ProtoGEO: Protótipo do SGBDOO GEOTABA

III.1 Introdução

Os SGBDs possuem em seu núcleo, uma máquina virtual que tem como um de seus objetivos evitar que o sistema fique dependente de um único equipamento ("hardware"), sendo capaz de ser portado para outros equipamentos.

A máquina virtual é responsável pela execução de operações com os valores armazenados no banco de dados. Por exemplo, em um SGBD relacional a máquina virtual deve ser capaz de realizar operações envolvendo os atributos das relações. Os atributos de uma relação pertencem a domínios onde os valores são sempre de um dos tipos primitivos (inteiros, reais, caracteres, cadeias, etc). Como conseqüência, as operações executadas pela máquina virtual são bastante simples. Essas operações constituem-se de operações lógicas e aritméticas envolvendo valores primitivos, e operações para manipulação de cadeia de caracteres.

Outra característica existente na máquina virtual de um SGBD relacional, é que estes sistemas utilizam um sistema de tipos onde somente relações são permitidas, ou

seja, todas as entidades são modeladas em tabelas (relações) sendo que cada linha da tabela (tupla) contém um número fixo de colunas (atributos).

No caso dos SGBDOOs, o problema é bem mais complexo. Como foi dito anteriormente, em um SGBD orientado a objetos as informações estão distribuídas em objetos. Os objetos, além de armazenarem os seus dados, também possuem procedimentos que são usados para manipular esses dados. A manipulação dos dados é feita de uma forma encapsulada, de modo que somente através desses procedimentos é possível se obter acesso aos dados do objeto.

A definição dos objetos, que corresponde à definição do esquema do banco de dados, é feita através da especificação de metaobjetos, que são as classes e as meta-classes. Estes metaobjetos estão organizados em uma estrutura hierárquica que permite o emprego de herança de propriedades entre classes e subclasses.

A comunicação entre objetos é feita através da troca de mensagens. Isto ocorre de tal forma que, quando um objeto recebe uma mensagem, um de seus métodos é ativado como conseqüência da mensagem.

Estas características apresentadas pelos SGBDOOs implicam em uma maior complexidade na construção de suas máquinas virtuais. Além das operações normalmente presentes na máquina virtual de um SGBD relacional, a máquina virtual de um SGBDOO precisa executar outras operações, como por exemplo, as operações que são definidas pelo usuário. Além disso, uma operação pode ativar a execução de outra operação e a busca para a localização dos procedimentos (métodos) a serem executados tem que ser feita dentro da hierarquia de classes, o que normalmente é feito em tempo de execução, através de um mecanismo de acoplamento dinâmico.

Na verdade, a máquina virtual de um SGBDOO é equivalente a um processador de uma linguagem orientada a objetos, que também possui estas mesmas características. O primeiro problema que surgiu, no início da definição do protótipo, foi definir que linguagem de programação orientada a objetos (LPOO) seria usada na construção da máquina virtual do SGBDOO GEOTABA.

As duas alternativas existentes para a solução deste problema eram: usar uma LPOO disponível no mercado ou implementar uma nova LPOO.

A primeira opção seria usar uma das linguagens já consagradas como C++ [STRO86] ou Smalltalk [GOLD83], entre outras. Porém, após estudo dessas linguagens e uma análise das vantagens e desvantagens de se usar uma delas, chegou-se a conclusão de que seria melhor implementarmos uma nova linguagem. Esta nova linguagem recebeu o nome de MANO (MANipulação de objetos).

Escolhendo a linguagem Smalltalk, por exemplo, haveria a vantagem de estar utilizando uma linguagem já conhecida, que possui um conjunto razoável de classes disponíveis e que está muito próxima do modelo de objetos definido para o GEOTABA [MONT90]. Por outro lado, Smalltalk apresenta diversas restrições e com isto não atende aos requisitos de um sistema de banco de dados. Como exemplos de restrições, podemos citar o fato de Smalltalk ser uma linguagem monousuária, impor limitações com relação ao tamanho dos objetos, quantidade de variáveis permitidas em um método, espaço limitado de endereçamento de objetos, o que não atende a sistemas que necessitam armazenar grandes volumes de dados, etc. Outra desvantagem é que, por ser uma linguagem fechada, haveria dificuldade para incluir características pertinentes a sistemas de banco de dados como gerenciamento de transações, mecanismos de reconstrução, concorrência, etc.

A linguagem C++ está tendo grande aceitação por parte da comunidade de processamento de dados em geral. Esta aceitação se deve principalmente pelo sucesso conseguido pela sua predecessora, a linguagem C. A utilização da linguagem C++ na construção do GEOTABA, seria muito conveniente por dois motivos principais: primeiro por ser uma linguagem de grande aceitação pública e segundo pelo seu ótimo desempenho. Mas, da mesma forma que a linguagem Smalltalk, a linguagem C++ possui um conjunto de desvantagens que foram fundamentais na sua recusa. Entre estas desvantagens pode-se citar o fato de que C++ é uma linguagem fortemente tipada e a "linkedição" do sistema seria necessária a cada inclusão ou alteração de uma classe ou método. Estas características são muito negativas quando presentes em um sistema que servirá de base para um projeto como o TABA, uma vez que por ser este um "ambiente de desenvolvimento de software", alterações no esquema do banco de dados são realizadas muito freqüentemente.

A linguagem de MANO foi então projetada para ser o "processador" da máquina virtual do sistema GEOTABA. Nos SGBDOOs, as operações que podem ser executadas

com um objeto são definidas através da especificação do protocolo do objeto, ou seja, do conjunto de mensagens que podem ser recebidas pelo objeto. Este protocolo engloba tanto as mensagens definidas na classe do objeto como as mensagens herdadas de suas superclasses. Cada mensagem está associada a um método, correspondente ao procedimento que deve ser executado quando a mensagem é recebida por um objeto desta classe.

A linguagem MANO é a linguagem através da qual são escritos os métodos e também, a linguagem usada para fazer o envio de mensagens aos objetos do banco de dados.

MANO foi definida, como será visto mais adiante, para ser uma representação sintática para o nível de implementação do modelo de objetos do GEOTABA. Sua sintaxe foi elaborada em um trabalho conjunto com o Prof. Luiz Carlos M. Monte.

O protótipo ProtoGEO está sendo construído com o objetivo de validar e aperfeiçoar o modelo de objetos do GEOTABA, e também para experimentar idéias sobre a implementação da arquitetura do sistema GEOTABA. Questões do tipo compartilhamento, concorrência, distribuição, segurança e reconstrução não são contempladas no ProtoGEO.

No restante deste capítulo detalha-se o ProtoGEO, começando com uma descrição sucinta do seu modelo de objetos. Uma descrição das principais características da linguagem MANO é apresentada na Seção 3 e, por último, é feito o detalhamento da arquitetura do ProtoGEO e de cada um de seus módulos componentes.

111.2 Modelo de Objetos do GEOTABA

Esta seção tem como objetivo mostrar as principais características existentes no modelo de objetos do GEOTABA [MONT92], além de servir para situar o presente trabalho no âmbito do projeto GEOTABA.

O modelo de objetos do GEOTABA, descrito em [MONT90], vai ao encontro dos princípios fundamentais do projeto GEOTABA, que são: gerenciamento de dados e procedimentos, com ênfase na manutenibilidade e na reusabilidade; possibilidade de modelagem conceitual da informação; fornecimento de um sistema de gerência de interfaces com usuário incorporado à definição e manipulação de objetos; e ser extensível.

Segundo L.C.M.Monte et al. [MONT92], tais princípios encontram sua vertente tecnológica em: linguagens de programação de bases de dados; paradigma da orientação a objetos; modelos semânticos de dados; sistemas de gerência de interface com usuário (SGIU); e bancos de dados extensíveis.

Linguagens de programação de banco de dados (LPBDs) têm sido projetadas para integrar a programação de aplicações e o acesso à base de dados persistente, normalmente através da utilização de um sistema de tipos único. Com o objetivo de resolver o problema de incompatibilidade entre linguagens de programação e linguagens de banco de dados, o modelo do GEOTABA foi projetado para ser implementado por uma LPBD chamada DEMO (DEfinição e Manipulação de Objetos), que será construída como uma sublinguagem da linguagem MANO, para dar suporte aos diversos níveis do modelo, como será visto mais a frente.

O SGBDOO GEOTABA, assim como a maioria dos sistemas da próxima geração, utiliza conceitos provenientes do paradigma da orientação a objetos como: identidade de objetos, classificação, herança e polimorfismo. A adoção desses conceitos tem como objetivo alcançar um maior grau de manutenibilidade e reusabilidade dos objetos gerenciados pelo GEOTABA.

Um dos pontos fortes do modelo é a integração de SGIUs e SGBDs. *O* modelo inclui mecanismos de modelagem de objetos de interface com usuário e define o relacionamento destes com os demais objetos das aplicações. A idéia é fornecer ferramentas que permitam ao usuário manipular diretamente os objetos através de suas diversas representações gráficas.

Um modelo de objetos para um sistema que é desenvolvido para suportar aplicações não convencionais, como é o caso do GEOTABA, não pode ser restrito a um número predeterminado de tipos padrão de objetos. O modelo do GEOTABA permite que novas classes sejam adicionadas à hierarquia, a medida que novas aplicações vão sendo desenvolvidas.

III.2.1 Principais Características do Modelo

Um Modelo de Dados Orientado a Objetos

No modelo de objetos do GEOTABA toda informação é representada por objetos, ao contrário de alguns modelos de objetos, onde o universo é particionado entre objetos e valores, como é o caso do sistema O_2 [BANC88]. Isto significa que inteiros, reais, caracteres e cadeias de caracteres também são objetos.

Cada objeto tem uma identidade própria, Única no sistema, que o acompanha durante toda a sua existência. Objetos se comunicam através de mensagens. A mesma mensagem enviada a objetos distintos pode ativar procedimentos distintos. Isto porque um objeto tem comportamento próprio, isto é, está associado a um conjunto de procedimentos, denominados métodos do objeto. Um objeto também tem estado, que varia sob a ação de seus métodos. O estado do objeto é o conteúdo de sua estrutura de dados, preenchida por referências a outros objetos, através de suas identidades. Alguns objetos são constantes, isto é, nunca mudam de estado. Objetos triviais, como os inteiros e os valores booleanos, nem sequer possuem estrutura de dados; suas identidades representam inteiramente o objeto.

Níveis de Abstração do Modelo

O modelo deve poder representar desde estruturas internas da implementação dos objetos até representações de interface com usuário destes. Para isto o modelo de objetos do GEOTABA foi projetado contendo quatro níveis de abstrações, como mostra a figura **3.1.**

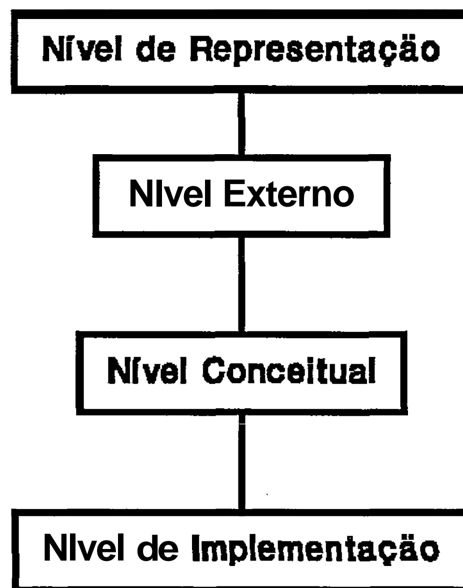


Figura 3.1: Níveis de Abstração do Modelo de Objetos

Um objeto pode ser visto de modo diferente em cada um destes níveis. O nível de implementação descreve estruturas de dados e métodos dos objetos conforme implementados na base de dados. O nível conceitual trata da estrutura *e* comportamento dos objetos conforme percebidos pela comunidade de usuários e pelos outros objetos. O nível externo particulariza visões específicas, para grupos de usuários, criando objetos

virtuais. O nível de representação refere-se a como os objetos são apresentados e manipulados nos meios de interface com usuário. A seguir é feita uma breve descrição das principais características de cada um dos níveis acima. Uma descrição ampliada pode ser encontrada em **[MONT90].**

O Nível de Implementação

O nível de implementação de um objeto define sua estrutura e seu comportamento, que não são visíveis para os demais objetos. **A** estrutura de um objeto é composta de campos ou variáveis de instâncias e o comportamento é representado por uma **coleção** de métodos. Os métodos de um objeto não acessam diretamente os campos de outros objetos, toda comunicação entre os objetos é feita através da troca de mensagens entre eles.

Classes são objetos que armazenam propriedades comuns a grupos de objetos. **A** classe define a estrutura de dados e o conjunto de métodos para esses objetos. É através das classes que novas instâncias de objetos são criadas. Todo objeto é descrito por um **metaobjeto.** Classes e Metaclasses são tipos específicos de metaobjetos.

Ao contrário de muitos modelos de dados orientados a objetos, no GEOTABA, classes e **coleções** são dois conceitos distintos. Uma **coleção** reúne objetos e uma classe reúne descrições de propriedades de objetos. Uma **coleção homogênea** é aquela em que todos os seus objetos são instâncias de uma mesma classe. Normalmente as consultas são realizadas sobre as **coleções** de objetos e não sobre as classes de objetos. Há vários tipos predefinidos de coleções - conjunto, lista, etc - podendo-se definir novos modelos de **coleções,** pois como coleções são objetos, são descritas por classes.

Uma das **coleções** do sistema é a que define os objetos globais, isto é, acessíveis a partir de qualquer classe. Esta coleção é a raiz da árvore de persistência. Um objeto (que pode ser uma **coleção)** é persistente se é global, ou então se existe alguma referência a ele em algum objeto persistente. **A** persistência é garantida para os objetos residentes tanto na memória, quanto em disco.

O Nível Conceitual

O nível conceitual de um objeto diz respeito a como ele é percebido pelos demais objetos, ou seja, corresponde ao protocolo do objeto - conjunto de mensagens que ele pode receber. Parte deste protocolo representa a estrutura percebida do objeto, o restante é formado pelos **serviços** fornecidos pelo objeto.

A estrutura percebida de um objeto compõe-se de atributos. Um atributo, definido no nível conceitual, pode ser mapeado automaticamente em um campo no nível de implementação, acrescido de dois métodos, um para atribuição e outro para recuperação do valor atual do campo. Um atributo constante não possui método de atribuição associado. **A** implementação do atributo pode ser feita manualmente, com seu valor sendo calculado a partir de outros campos.

Existem três tipos de atributos no modelo: mapeamentos, componentes e atributos propriamente ditos. Estes últimos informam valores de características do objeto, como cor, tamanho, data de nascimento, etc. Componentes são campos que descrevem um objeto em termos de sua composição percebida, isto é, indicam que seus valores são partes ou componentes do objeto. Um mesmo objeto não pode ser componente de dois objetos. Os mapeamentos são atributos que caracterizam o objeto em função de relacionamentos com outros objetos.

Serviços correspondem aos métodos que primordialmente produzem processamento e não representam a estrutura do objeto. São usados para descrever cada ação possível de ser executada pelo objeto.

No nível conceitual, a cada atributo há uma **restrição de tipo** associada, indicando que apenas os objetos que atendam esta restrição podem ser atribuídos ao atributo. O tipo de um atributo define o protocolo mínimo desses objetos. **A** maior parte da verificação de compatibilidade de tipos é realizada estaticamente.

Os tipos conceituais são definidos a partir das classes de implementação. Uma classe **pode** definir, por "default", um tipo correspondente ao seu protocolo. Um tipo também pode ser derivado de outros tipos, por herança de protocolo. **A** hierarquia de tipos é distinta da hierarquia de classes. Somente atributos, parâmetros de mensagens e

variáveis possuem tipo. Um objeto comumente pode ser atribuído a diversos tipos diferentes.

O Nível Externo

No nível externo, os objetos são descritos em termos das perspectivas pelas quais podem ser vistos. Na maioria dos modelos de objetos existentes, como no caso do Smalltalk, o protocolo definido por uma classe é totalmente visível pelos usuários desta classe. Alguns modelos limitam a visibilidade a dois níveis, definindo que certos métodos são privativos. Um método privativo significa que ele não é acessível por outras classes. Esta restrição é pouco adequada, pois comumente precisa-se definir métodos usáveis por um conjunto de classes, mas de uso restrito a indivíduos qualificados. A solução sugerida no modelo é a possibilidade de se definir vistas de um objeto. Vistas, como em banco de dados relacionais, são janelas que permitem ver partes do objeto.

Uma vista é definida sobre um tipo (nível conceitual) de objeto ou sobre outra vista. Uma vista fornece um novo protocolo (atributos e serviços), definindo um novo tipo, construído a partir do protocolo do tipo (ou vista) base, excluindo atributos *e* serviços ou acrescentando novos serviços e atributos derivados do protocolo base. Uma vista não é um objeto em si, mas uma outra maneira de se ver um mesmo objeto. Outra propriedade do nível externo do modelo é a possibilidade de se definir coleções virtuais. Uma coleção virtual é uma vista, calculada a partir de uma coleção base, selecionando membros desta.

O Nível de Representação

O nível de representação inclui as possíveis formas nas quais objetos são apresentados ao usuário e por ele manipulado. De uma maneira geral um objeto pode ser apresentado no formato de ícones, formulários, tabelas, etc. A implementação de diferentes representações para objetos envolve a utilização de outros tipos de objetos que para tal devem ser especialmente definidos.

A descrição da representação de um objeto é mapeada sobre sua descrição conceitual ou externa. Assim, as propriedades estruturais (atributos, componentes e relacionamentos) passam a ter uma forma no meio de apresentação (vídeo, impressora), enquanto algumas propriedades comportamentais (os serviços) passam a ser vistas como possíveis operações que o usuário pode executar sobre o objeto.

Interfaces com manipulação direta permitem que o usuário veja a informação desejada na tela e altere essa informação através de manipulação de sua representação. Desse modo, cada objeto pode ter formas próprias de representação nos meios de interação com o usuário (tela, impressora, etc.). Estas representações devem simular os objetos base no seu comportamento típico.

Todavia, as representações são objetos distintos dos objetos base, com características próprias e implementados por classes totalmente diferentes. Por exemplo, um estudante pode estar sendo visto sobre a forma de um quadro contendo um gráfico de barras comparando as suas notas em diversas matérias. Evidentemente este quadro não é o estudante, porém, para o usuário, em uma dada aplicação, ele pode ser o real representante do estudante. Alterar a altura de uma barra do gráfico pode significar, nesta aplicação, alterar realmente a nota do estudante.

III.2.2 O Modelo de Objetos e o ProtoGEO

O ProtoGEO está sendo construído com o objetivo de validar e aperfeiçoar o modelo proposto, entre outras coisas. Numa primeira etapa, somente os níveis de implementação e conceitual serão contemplados. A linguagem MANO é utilizada para a implementação dos métodos que descrevem o comportamento dos objetos no nível de implementação do modelo e também fornece uma sintaxe para o envio de mensagens aos objetos.

111.3 MANO - Linguagem de Manipulação de Objetos

Neste item descreve-se a linguagem de Manipulação de Objetos - MANO. Como foi dito anteriormente, MANO é utilizada para a implementação dos métodos que descrevem o comportamento dos objetos no nível de implementação do modelo de objetos do GEOTABA.

Além da definição de métodos, MANO fornece uma sintaxe para envio de mensagens a objetos, ou seja, uma sintaxe para que possa ser feita a comunicação entre os objetos.

MANO pode ser classificada como uma linguagem de programação orientada a objetos, pois emprega os conceitos que caracterizam este paradigma, como por exemplo polimorfismo, encapsulamento de dados e procedimentos, casamento dinâmico de métodos-mensagens, herança de propriedades através de uma hierarquia de classes, etc.

O sistema GEOTABA será operado pela linguagem de programação de banco de dados DEMO, que possibilita o acesso ao banco de dados e possui o poder de expressão das linguagens de programação. O emprego da linguagem DEMO tem como objetivo resolver o problema de incompatibilidade entre as linguagens de operação e programação do banco de dados no âmbito do sistema GEOTABA. DEMO será implementada como uma sublinguagem de MANO, onde todo programa DEMO será na realidade um programa MANO, porém fazendo acesso aos objetos do metaesquema preprogramados em MANO.

A linguagem MANO foi desenvolvida com base na linguagem Smalltalk [GOLD83], e foi projetada para ser uma linguagem aberta, na qual a adição de características de banco de dados fosse facilitada. O projeto de MANO objetivou resolver, também, alguns aspectos limitantes existentes na linguagem Smalltalk, como por exemplo o fato desta ser uma linguagem onde todos os objetos ficam residentes em memória durante uma sessão, a quantidade de objetos endereçáveis, etc.

III.3.1 Características da Linguagem MANO

Características de Interface com Usuário

A linguagem MANO foi projetada para ser utilizada através de um editor que use atributos de caracteres, para realçar a visualização dos programas. O exemplo mostrado na figura 3.2 apresenta uma alternativa para a construção deste editor. Neste exemplo os comentários estão colocados dentro de caixas, os nomes de objetos aparecem em negrito e a declaração de variáveis temporárias é feita sublinhando o nome dessas variáveis.

Como ainda não foi definido o editor de métodos, na versão atual do protótipo, os métodos foram escritos usando a representação interna da linguagem. Nesta representação interna, são usados símbolos especiais, para diferenciar os diversos elementos da linguagem. Por exemplo, nomes de objetos são precedidos do caracter cifrão ($), preposições são precedidas do caracter sublinhado (_), etc.

```
|Janela|

inverte_janela;

        +------------------------------------+
        |      Este  método  troca  os  valores |
        | base e altura do objeto receptor     |
        +------------------------------------+

   temv;

   temp := base;
   base := altura;
   altura := temp;
```

Figura 3.2: Método MANO usando atributos de caracteres

Outra característica interessante de interface com usuário, é que MANO reconhece o conjunto de caracteres ascii estendidos, com isto são permitidas palavras acentuadas no código fonte de um método MANO, como poderá ser visto nos exemplos. Nas **seções** seguintes estão descritos os diversos elementos da linguagem MANO e por isso o método que aparece na figura 3.2 não será explicado aqui.

Estrutura de um Método MANO

Todo método armazenado no sistema está associado a uma classe através de seu seletor. O seletor do método é formado por uma cadeia de caracteres que o identifica dentro de sua classe. O seletor é extraído do cabeçalho do método no momento da compilação e é armazenado em uma tabela que contém a relação dos métodos da classe.

Métodos podem receber parâmetros e retornar um valor. O valor retomado por um método corresponde à identidade do objeto resultante. Os argumentos do método são declarados no seu cabeçalho junto com o seletor. O símbolo de circunflexo (^) indica que o objeto resultante da mensagem que segue, é o valor a ser devolvido pelo método.

```
|Janela|

altere janela corrente: $num_janela;

/* Este método altera o número da janela ativa */

   $janela_corrente := $num_janela;
   ^$janela_corrente;
```

Figura 3.3: Exemplo de Método em MANO

A figura 3.3 exemplifica um método MANO. Em um arquivo fonte, o método é precedido do nome da classe sob a qual será associado. A figura exemplifica o método de seletor *#alterejanelacorrente:* pertencente à classe ***Janela*** e que possui um único

argumento *($numjanela)*. Neste método, o valor do argumento ***$num_janela*** é atribuído à variável de instância ***$janela-corrente*** e ao final de sua execução, o valor da variável ***$janela_corrente*** é devolvido como "resposta" à mensagem que o ativou.

Variáveis temporárias podem ser declaradas dentro de um método. Quando for o caso, elas deverão ser declaradas imediatamente após o cabeçalho do método. A declaração de variáveis temporárias é feita, de acordo com a representação interna de **MANO,** acrescentando-se o símbolo de porcentagem (%) na frente de cada variável declarada.

O nível de implementação do modelo não inclui tipos, que é uma restrição que aparecerá na linguagem DEMO, quando se tratar dos demais níveis do modelo. Em **MANO** as variáveis de instância de um objeto, argumentos de métodos e demais variáveis armazenam valores que correspondem à identidades de objetos. Por isto, a declaração de variáveis temporárias é feita simplesmente citando seus nomes.

A figura **3.4** mostra um método contendo a declaração da variável temporária ***$temp***.

```
|Janela|

inverte_janela;

   /*   Este método troca os valores base e
      altura do objeto receptor          */

   % $temp;

   $temp := $base;
   $base:= $altura;
   $altura:= $temp;
```

Figura 3.4: Exemplo de uso de variável temporária

Mensagens em MANO

Uma mensagem é equivalente a uma chamada de subrotina nas linguagens procedimentais. Nas linguagens de programação orientadas a objetos, quando um objeto recebe uma mensagem, o método que estiver associado à mensagem recebida é executado. **A** busca do método a ser executado é realizada na hierarquia de classes, a partir da classe do objeto receptor da mensagem. Este procedimento está detalhado no capítulo IV (Processador MANO).

Uma mensagem pode ser simples, isto é, não possuir parâmetros, como no exemplo abaixo:

```
30.raiz_quadrada
```

Neste caso a mensagem de seletor #raiz_quadrada está sendo enviada ao objeto ***30.*** Como este objeto é uma instância da classe Inteiros, o método associado a este seletor será localizado na tabela de mensagens da classe Inteiros (ou de uma de suas superclasses) e será executado.

Mensagens com parâmetros também são permitidas. Observe os exemplos abaixo:

```
$aplicação altere janela corrente: 2
```

```
$janela_corrente exiba em: $x e: $y
```

No primeiro exemplo, a mensagem de seletor ***#alterejanelacorrente:*** e argumento 2 está sendo enviada ao objeto cuja identidade está armazenada na variável $aplicação. No segundo exemplo temos uma mensagem de dois argumentos *($x)* e *($y)* cujo seletor é ***#exibaem:e:*** sendo enviada ao objeto cuja identidade está armazenada na variável $janela_corrente. Como pode ser observado no primeiro exemplo, as palavras altere, janela e corrente, que formam o seletor da mensagem estão separadas por caracteres

branco (espaço). Quando o compilador MANO encontra um seletor assim, ele faz a concatenação das palavras para formar o seletor da mensagem.

Um tipo comum de mensagens com parâmetros são aquelas cujos seletores são operadores aritméticos, lógicos e relacionais. Nas linguagens convencionais, normalmente o conjunto dessas operações é preestabelecido na sintaxe da linguagem e essas operações são resolvidas por rotinas primitivas. Na linguagem MANO existe também um conjunto de rotinas primitivas para a execução eficiente das operações mais comuns. Porém, o conjunto de operações é aberto permitindo assim que o usuário defina novas operações a medida que se façam necessárias. Por exemplo, uma classe ***Matriz*** pode ser criada e operações de adição, subtração, multiplicação e transposta, podem ser **definidas** para serem aplicadas em suas instâncias.

Por isto, dá-se um tratamento uniforme tanto para mensagens cujos seletores são operadores aritméticos, lógicos e relacionais, como para as demais mensagens. Por exemplo, na figura 3.5 existem duas mensagens na expressão ***($base * $altura) /*** 2, uma com seletor / (divisão) e argumento 2 sendo enviada para o objeto resultante da outra mensagem, de seletor * (multiplicação) com argumento ***$altura*** que está sendo enviada para ***o*** objeto ***$base.***

```
|Janela|
área;
   /* Calcula a área do objeto receptor */
   ^ ( $base * $altura ) / 2;
```

Figura 3.5: Mensagens onde os seletores são operadores

Ao contrário da linguagem **Smalltalk,** MANO estabelece uma <u>ordem de precedência</u> para a execução de mensagens cujos seletores são operadores aritméticos, lógicos e relacionais.

Em MANO, uma expressão contendo diferentes operadores é executada de acordo com a seguinte ordem de precedência: primeiro são executadas as operações multiplicativas (* e /), depois são executadas as operações adicionais (+ e -), em seguida as operações relacionais (< , > , <= , >= , = , ~=) e por último as operações lógicas (& , | , ~). MANO também permite a utilização de parênteses, através dos **quais** é possível alterar a ordem de execução em uma expressão.

Outra característica inovadora desta linguagem é a possibilidade de se escrever as mensagens nas formas **pré-fixa e pós-fixa**. Por exemplo, para obter o nome do chefe do departamento de um funcionário podemos escrever:

```
$func.depto.chefe.nome
```

Nesta expressão a mensagem ***#depto*** está sendo enviada ao objeto designado pela **variável** *$func*. **Ao objeto** resultante desta mensagem, está sendo enviada a mensagem #chefe, e da mesma forma a mensagem #nome é enviada ao objeto resultante da mensagem #chefe.

Esta mesma expressão poderia ter sido escrita usando-se preposições, obtendo-se uma forma mais próxima da língua portuguesa, como mostra o exemplo abaixo.

```
nome _do chefe _do depto _de $func
```

Na representação interna da linguagem MANO, preposições são precedidas do símbolo de sublinhado (_). As preposições são descartadas pelo compilador da mesma forma que o ponto (.) que aparece nas mensagens escritas na forma pós-fixa.

Toda mensagem possui um objeto receptor, o seletor da mensagem e se for o caso, um ou mais parâmetros. Quando diversas mensagens precisam ser enviadas ao mesmo objeto, o usuário pode encadeá-las escrevendo o objeto receptor uma única vez e separando as diversas mensagens com o caractere vírgula (,). **A** figura **3.6** exemplifica as **mensagens** em cascata.

```
|Janela|

inicialize com base: $base altura: $altura cor: $cor;

    /* Cria e Inicializa uma janela  */

         % $janela;

    $janela := $Janela.novo;
    $janela base: $base,
              altura: $altura,
              cor: $cor;

    ^$janela;
```

Figura 3.6: Exemplo de mensagens em cascata

Esta figura mostra o método de seletor *#inicializecombase:altura:cor:*, que possui três argumentos ($base, $altura e $cor). Uma variável temporária *($janela)* é declarada para armazenar a identidade da nova instância, que é criada quando a mensagem #novo é enviada à classe Janela. A criação de objetos está descrita em uma seção mais a frente. Após criada, a nova instância tem seus valores **inicializados** através de uma seqüência de mensagens que estão sendo enviadas (em cascata) à **variávelD** ***$janela.*** Ao final, o método retoma a identidade do objeto criado.

Um conjunto de mensagens podem ser agrupadas para formar um bloco de mensagens. Como na maioria das linguagens, blocos de mensagens são utilizados principalmente em comandos de decisão e de repetição. Um bloco de mensagens é delimitado pelos caracteres abre-chaves ({) e fecha-chaves (}).

Blocos de mensagens possuem mais semântica do que blocos de comandos das linguagens convencionais. Um bloco de mensagens tem um comportamento semelhante a um objeto, na medida em que, para ser executado, ele precisa receber a mensagem #avalie, que é uma mensagem especial da linguagem. Porém, um bloco não é um objeto, uma vez que não é instância de nenhuma classe.

A vantagem de se ativar um bloco a partir de uma mensagem, é que ele pode receber parâmetros no momento de sua ativação. Isto é útil por exemplo nas mensagens que implementam um laço repetitivo onde a cada iteração o valor da variável de controle é incrementado, semelhante ao comando *for* de linguagens como Pascal e C++. Porém, como a passagem de parâmetro para um bloco é feita via programação, a estrutura usada torna-se muito mais flexível, podendo inclusive ser passado mais de um parâmetro.

```
!Objeto!

duplique;

/* Este método retoma um objeto idêntico ao receptor.
   A cópia é realizada em profundidade                    */

                 % $cópia; % $classe; % $vars;

$classe := classe;
( $classe. é Variável )
    severo:{    $vars := tamBásico;
                    $cópia := $classe novo: $vars;
                }
    senão:{     $vars := 0;
                    $cópia := $classe.novo;
                }
( $classe.é Apontador )
    severo:{    1 até: ($vars + $classe.tamInst) faça:
                            { % $i;$cópia[$i]:=$mim[$i].duplique; }
                }
    senão:{     1 até: $vars faça:
                            { % $i;$cópia[$i]:=$mim[$i]; }
                }
^$cópia;
```

Figura 3.7: Método MANO usando blocos de mensagens

A figura 3.7 mostra o método de seletor #duplique pertencente à classe ***Objeto.*** Este método retorna uma cópia em profundidade do objeto receptor da mensagem. Neste exemplo, blocos de mensagens são usados tanto em mensagens que são executadas condicionalmente como em mensagens que são executadas de forma repetitiva. A

mensagem ***#até:faça:*** faz com que o bloco de mensagens correspondente ao segundo argumento seja executado diversas vezes. A cada repetição, a variável temporária $i, que é o argumento do bloco, é incrementada.

Este método faz uso da variável especial ($mim), que está descrita mais a frente e que **contém** a identidade do objeto receptor da mensagem que ativou o método. Em alguns casos, quando outras mensagens estão sendo enviadas ao objeto receptor, como nas expressões $classe := classe e ***$vars*** := ***tamBásico*,** pode-se omitir o receptor da mensagem, que neste caso é a variável especial ($mim). Estas mensagens poderiam ter sido escritas da seguinte forma: $classe := ***$mim.***classe e ***$vars*** := $mim.***tamBásico*** respectivamente.

Definição de Classes

A definição de classes, que em **SGBDOOs** corresponde à definição do esquema do banco de dados, será feita no GEOTABA através de um utilitário que utilizará técnicas de interface gráfica. Outra maneira possível de se definir uma classe, é através do uso de métodos existentes na classe Classe, que é predefinida e que pertence ao metaesquema do GEOTABA.

O projeto do metaesquema do GEOTABA está sendo desenvolvido paralelamente ao projeto do protótipo ProtoGEO. Como a linguagem MANO necessita usar classes do metaesquema que ainda não estão disponíveis, uma solução alternativa foi desenvolvida para ser usada temporariamente. Esta solução consistiu na implementação de rotinas escritas em C++, (a linguagem na qual está sendo implementado o núcleo do ProtoGEO e do qual fazem parte o Compilador e o Processador MANO) para permitir a construção da hierarquia de classes, o que é feito através da especificação de novas classes.

No topo da hierarquia de classes está a classe predefinida Objeto. A classe Objeto descreve, através de seus métodos, o comportamento que é comum a todos os objetos existentes no banco de dados. Novas classes são sempre criadas como sendo subclasses de classes já existentes na hierarquia. **A** figura **3.8** exemplifica a criação da classe Janela utilizando métodos da biblioteca de classes C++.

```
C1 janela;  // Define variável temporária do tipo C1

janela := Classe.novo();

janela.inicialize( Objeto,"Janela",3,10 );

janela.acrVarInst("$base");
janela.acrVarInst("$altura");
janela.acrVarInst("$cor");
```

Figura 3.8 Definição da classe Janela

Como mostra a figura, a definição de uma classe **MANO** é feita através de chamadas à **subrotinas** de uma biblioteca C++. Uma variável temporária é declarada para armazenar a identidade da nova instância da classe *(Cl)* que é criada através da mensagem ***Classe.novo().*** ***Cl*** é a classe C++ cuja definição especifica um objeto MANO que é instância da classe predefinida Classe. Esta classe **(*Cl*)** é um dos elos entre um objeto especificado na linguagem de implementação (C++) e o mesmo objeto que será manipulado na linguagem que está sendo implementada.

Após a criação da nova instância de Classe, o método inicialize é executado. Este método faz a introdução da nova classe (conteúdo da variável janela) na hierarquia de classes, criando um elo entre a que está sendo definida e sua superclasse, no caso Objeto (primeiro argumento da mensagem inicialize). Isto quer dizer que a classe Janela será uma subclasse direta da classe Objeto. **O** segundo argumento especifica um nome para a nova classe que está sendo criada (Janela). **O** terceiro e quarto argumentos correspondem respectivamente, ao número de variáveis de instância e à quantidade máxima de métodos que serão definidos para a classe. Estes valores poderão ser alterados através de rotinas de manutenção do esquema.

Para completar a definição da classe, a declaração das variáveis de instâncias é feita através da ativação do método ***acrVarInst*** (acrescenta variáveis de instância), que tem como parâmetro o nome da variável a ser incluída. **A** definição dos métodos **MANO**

é feita através da compilação de arquivos fonte. Este procedimento está descrito no capítulo V (Compilador MANO).

Criação de Objetos

A criação de objetos é feita através do envio da mensagem de seletor #novo para a classe do objeto que se deseja criar.

Quando uma classe recebe a mensagem #novo, um objeto com as características desta classe é criado, ou seja, é alocado um pedaço de memória de acordo com as características descritas em sua classe, e uma identidade é atribuída a esta nova instância. A criação de objetos é responsabilidade do Gerente de Armazenamento, que está descrito no capítulo VI (Gerente de Armazenamento).

A figura *3.6* mostrou um método onde uma instância da classe Janela foi criada e sua identidade foi armazenada na variável temporária ($janela).

Algumas classes possuem instâncias com tamanho variável, como é o caso da classe ***Arranjo***. A criação de instâncias destas classes é feita através da mensagem de seletor ***#novo:*** que possui como argumento o tamanho da instância desejada. Por exemplo, para criar uma instância da classe ***Arranjo*** com tamanho igual a ***30,*** deve-se escrever a seguinte mensagem:

```
Janela novo: 30
```

Variáveis Globais

MANO fornece suporte para variáveis globais. As variáveis globais, como as demais variáveis (temporárias, de instância e argumentos), armazenam identidades de objetos. Uma variável global pode ser acessada a partir de qualquer classe, isto é, qualquer método pode referenciar uma variável global.

As variáveis globais estão armazenadas em um objeto coleção cuja identidade é armazenada na variável global *$MANO*. Esta coleção é usada na definição de quais objetos são persistentes. Um objeto é persistente se é global, ou então se existe alguma referência a ele em algum objeto persistente. Objetos persistentes são aqueles cuja existência independe da execução de um programa, ou seja, ao final de uma sessão os objetos persistentes não são perdidos.

A declaração de variáveis globais será feita através de um utilitário que será desenvolvido para o GEOTABA. Nesta primeira fase do protótipo, as variáveis globais são declaradas por intermédio de um procedimento C++ chamado acrVarGlobal, cujo argumento corresponde ao nome da variável a ser declarada. A figura **3.9** mostra exemplos de declaração de variáveis globais.

```
acrVarGlobal( "$Cadastro_de_Aplicações" );
acrVarGlobal( "$Janela_Menu" );
acrVarGlobal( "$Cursor" );
```

Figura 3.9: Declaração de Variáveis Globais

A declaração das variáveis globais é feita via rotina C++, porém, o acesso e a manipulação dessas variáveis só é possível através de mensagens MANO.

Variáveis Especiais $mim e $super

Além das variáveis de instância, temporárias, argumentos de métodos e globais, existem duas variáveis especiais: $mim e $super que também podem ser acessadas pelos métodos. A variável *$mim* contém a identidade do objeto receptor da mensagem que ativou o método. Esta variável é útil quando o método que está sendo executado precisa enviar outras mensagens para o objeto receptor. **A** figura 3.10 exemplifica esta situação.

```
|Janela|

exibe na horizontal;

   /* Exibe janela horizontalmente na
      posição atual do cursor      */

( $altura > $base )
  severo: {
        $mim.inverte_janela.exiba em: $Cursor.x
                         e: $Cursor.y;
     }
  senão: {
        exiba em: $Cursor.x e: $Cursor.y;
     };
```

Figura 3.10: Exemplo de uso da variável $mim

Neste exemplo, a mensagem de seletor #invertejanela será enviada ao mesmo objeto que receber a mensagem de seletor ***#exibenahorizontal.*** **A** figura também mostra exemplos de uso da variável global ***($Cursor).***

Um método herdado pode ser redefinido em uma subclasse. Há situações no entanto, em que desejamos executar não o método redefinido, mas sim aquele que estava sendo herdado. Para isto, pode-se usar a variável ***$super,*** que faz referência à superclasse da classe do objeto receptor. Quando uma mensagem é enviada à variável ***$super,*** o Processador MANO começa a busca do método que será executado a partir da tabela de mensagens da superclasse da classe do objeto receptor.

Outras Características da Linguagem MANO

Os dados de um objeto são armazenados em variáveis de instância. Porém, MANO emprega o conceito de encapsulamento, que faz com que esses dados (variáveis de instância) não sejam visíveis à outros objetos. A única forma de se ter acesso aos dados de um objeto é através da execução de seus métodos. Normalmente, para cada variável de instância de um objeto existem dois métodos associados, um para alterar o

seu valor e outro que fornece o valor armazenado na variável. Por exemplo, na classe Janela que foi definida anteriormente, existem dois métodos associados com a variável de instância $cor, o método de seletor #cor, que devolve o valor armazenado nesta variável e o método de seletor #cor: e argumento ***$umaCor***, que permite alterar o valor da variável. A figura 3.11 mostra o código destes métodos.

```
|Janela|

cor;
    /* Retorna o valor da cor da janela */
  *$cor;


cor: $umaCor;
    /* Altera o valor da cor da janela */
  $cor := $umaCor;
```

Figura 3.11: Acesso e alteração de variável de instância

O sistema GEOTABA fornecerá um mecanismo no qual o mapeamento do esquema do banco de dados do nível conceitual para o nível de implementação gerará automaticamente, para cada variável de instância de uma classe, seus dois métodos associados.

O conceito de encapsulamento de dados permite o desenvolvimento de aplicações com alto grau de manutenibilidade, uma vez que é um conceito que teve suas origens nos Tipos Abstratos de Dados. Além disso, permite que os dados sejam independentes dos programas, o que é uma característica muito importante nos sistemas de gerenciamento de banco de dados. Por exemplo, um programa de aplicação que precisa **acessar** um objeto em uma **coleção,** provavelmente o fará através do envio de uma mensagem ao objeto coleção, passando como parâmetro a identificação do objeto desejado. Suponhamos que a classe Coleção possua duas subclasses, uma onde os componentes da **coleção** são recuperados através de um índice do tipo árvore-B e outra onde os componentes são recuperados por meio de uma tabela de "hash". Como estas

características são internas aos objetos, o programa de aplicação não tem como saber o tipo de **índice** que foi usado na recuperação do objeto.

Outra característica importante presente nesta linguagem é o conceito de **herança**. Herança reduz a necessidade de especificação de informações redundantes possibilitando a obtenção de um alto grau de reusabilidade.

As classes estão organizadas em uma estrutura hierárquica, onde uma subclasse herda as propriedades (variáveis de instância e métodos) de sua superclasse, esta por sua vez também herda as propriedades de sua superclasse e assim sucessivamente até chegar no topo da hierarquia, onde está localizada a classe Objeto. Com isto, uma subclasse herda as propriedades de todas as suas classes ancestrais. Por exemplo, na definição da classe Janela–de–Edição (figura 3.12), que é uma subclasse da classe Janela, só é necessário definir os novos atributos e métodos que caracterizam a especialização da classe, já que as variáveis $base, $altura e *$cor* e todos os métodos da classe Janela são herdados pela classe Janela–de–Edição.

Apesar da definição da classe Janela_de_Edição **só** incluir três variáveis de instância, cada instância de Janela_de–Edição conterá seis variáveis de instância, que são: $base, $altura, $cor, $barra–na–horizontal, $barra_na_vertical e *$tipo_cursor* e responderão a todos os métodos definidos nas classes Janela–de–Edição, Janela e Objeto.

```
Cl janelaEdita;

janelaEdita := Classe.novo();

janelaEdita.inicialize( Janela,"Janela_de_Edição",3,12 );

janelaEdita.acrVarInst( "$barra_na_horizontal" );
janelaEdita.acrVarInst( "$barra_na_vertical" );
janelaEdita.acrVarInst( "$tipo_cursor" );
```

Figura 3.12 Definição da classe Janela_de_Edição

Apesar do modelo de objetos do GEOTABA definir que uma classe poderá ter mais de uma superclasse direta, isto é, que o sistema GEOTABA **suportará** herança múltipla, a primeira versão da linguagem MANO restringe a hierarquia ao mecanismo de herança simples, ou seja, a hierarquia de classes equivale a uma estrutura em árvore. Herança múltipla *será* introduzida na linguagem em trabalhos futuros.

Outro conceito importante utilizado é o conceito de polimorfismo. **Polimorfismo** em geral, significa "ter ou assumir diferentes formas" [STEF86]. Esta característica ocorre quando diferentes objetos podem responder a uma mesma mensagem, porém com comportamento próprio. Por exemplo, um objeto da classe ***Data*** responde a mensagem ***imprima*** exibindo o seu estado no formato *dd/mm/aa*. Já um objeto da classe ***Dinheiro*** responde a mesma mensagem exibindo o seu estado no formato *Cr$x.xxx,xx*.

Para isto, o acoplamento entre a mensagem enviada a um objeto e o método que será executado é feito dinamicamente, ou seja, em tempo de execução. Este atraso na ligação mensagem-método é conhecido como acoplamento dinâmico ou "late binding".

111.4 Arquitetura do ProtoGEO

Um dos objetivos da construção do ProtoGEO, foi o de desenvolver um protótipo para experimentar idéias sobre a implementação da arquitetura do GEOTABA. A arquitetura definida inicialmente para o ProtoGEO (figura 3.13) é uma arquitetura simplificada, uma vez que o protótipo não fornecerá suporte à características como compartilhamento, concorrência, distribuição, segurança e reconstrução.

A arquitetura do ProtoGEO é composta de três componentes principais: o gerente de Objetos, o Gerente de Execução e o Gerente de Armazenamento.

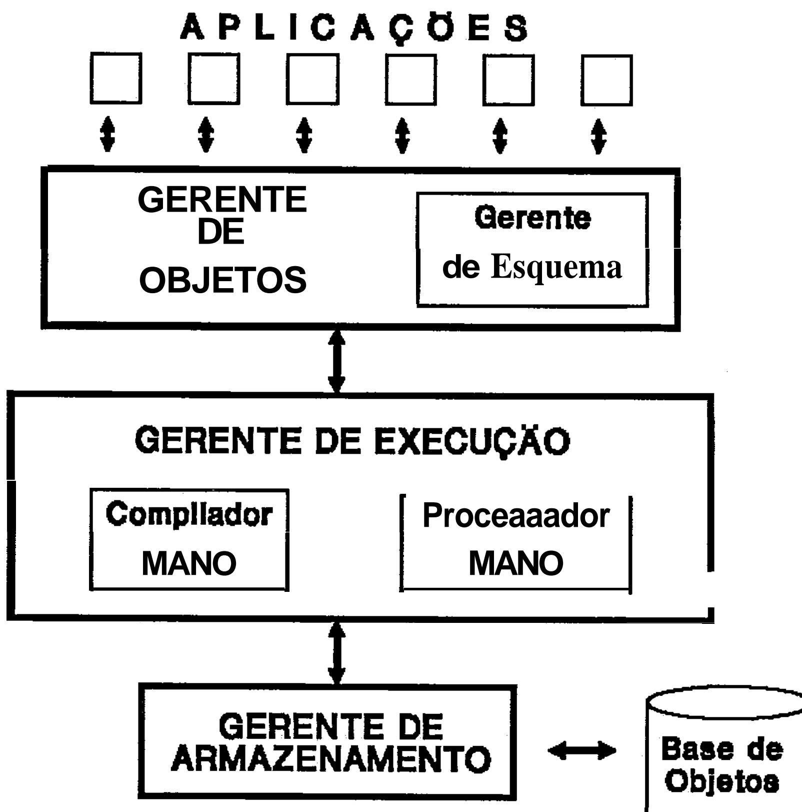


Figura 3.13: Arquitetura do ProtoGEO

III.4.1 Gerente de Objetos

O Gerente de Objetos (GO) gerenciará os objetos nos diversos níveis de abstração do modelo de objetos proposto, implementando os mapeamentos entre esses níveis. Na primeira etapa do projeto, apenas os níveis de implementação e conceitual serão contemplados. O GO possuirá um conjunto de classes predefinidas que serão gerenciadas pelo Gerente de **Esquema**, que será responsável pelas classes que formam o metaesquema do banco de dados. Como exemplo dessas classes predefinidas pode-se citar as classes Classe, Metaclasse, Atributo, Método e Coleção. O projeto do metaesquema do GEOTABA está sendo desenvolvido paralelamente à implementação do ProtoGEO, e por isso, a descrição destas classes está fora do escopo deste trabalho.

Os gerentes de Execução e de Armazenamento formam o núcleo do ProtoGEO. O núcleo do ProtoGEO é responsável pelo processamento de mensagens e métodos escritos na linguagem de Manipulação de Objetos MANO. A figura **3.14** mostra de forma mais detalhada, os componentes presentes no núcleo do ProtoGEO.

III.4.2 Gerente de Execução

O Gerente de Execução (GE) é responsável pelo processamento das mensagens enviadas aos objetos do sistema. Quando um objeto recebe uma mensagem, um método especifico deve ser executado como "resposta" à mensagem recebida. **A** identificação do método a ser executado, bem como sua execução são tarefas de responsabilidade do GE. O GE utiliza o Compilador MANO para traduzir uma mensagem MANO, em um método interpretável. Após esta tradução o Processador MANO é então ativado para executar o método. Durante a execução de um método, diversas mensagens são enviadas à objetos, o que resulta na ativação de outros métodos. O Processador realiza a busca dos métodos que serão executados na hierarquia de classes, o que é feito em tempo de execução.

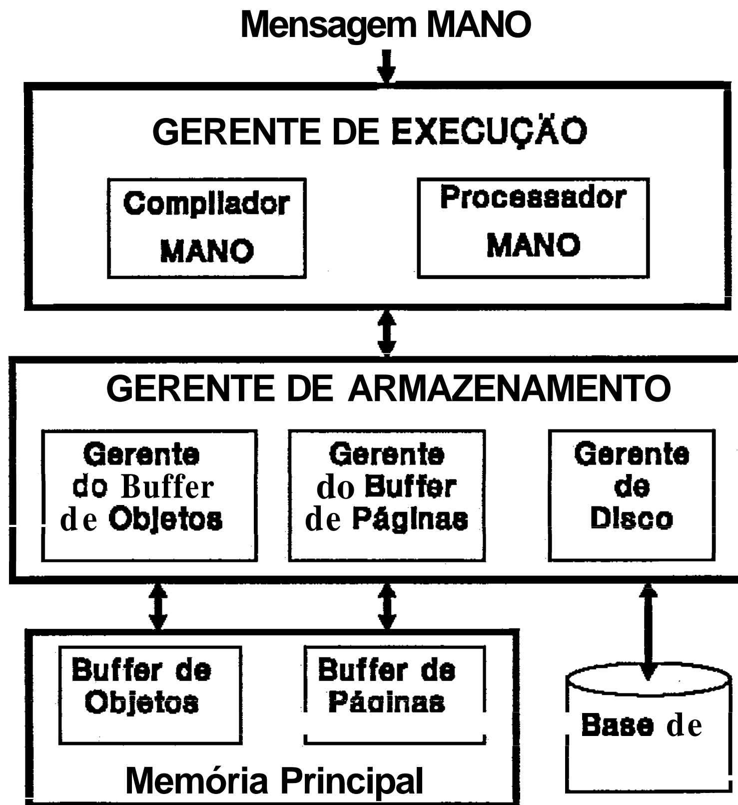


Figura 3.14: Núcleo do **ProtoGEO**

O Compilador tem a função de transformar um método ou mensagem escritos em MANO, em um método escrito em um código intermediário. Os métodos traduzidos pelo Compilador são armazenados em instâncias da classe ***Método,*** o que significa que os métodos, em sua forma traduzida, são também objetos e ficam armazenados juntamente com os demais objetos no banco de dados.

O Processador tem a função de identificar e executar o método que estiver associado à mensagem enviada a um objeto do sistema. Para que o Processador execute um método, este já deverá ter sido traduzido pelo Compilador, ou seja, o Processador executa métodos escritos em código intermediário.

Este código intermediário é formado por um conjunto de bytes aos **quais** chamamos de instruções MANO. Estas instruções são semelhantes aos "bytecodes" usados na linguagem Smalltalk.

O Processador, nesta primeira fase do projeto, é equivalente ao interpretador Smalltalk, conforme especificado em **[GOLD83]**, já o Compilador é totalmente original, apesar da sintaxe da linguagem MANO possuir alguma semelhança com a sintaxe Smalltalk.

O Processador e o Compilador criam novos objetos a medida que são executados. Eles também fazem acesso a objetos já armazenados no banco de dados. No entanto, tanto o Processador como o Compilador só acessam objetos presentes na memória principal. A recuperação de objetos armazenados nos meios secundários e sua transferência para a memória são tarefas de responsabilidade do Gerente de Armazenamento. Como os objetos manipulados pelo Processador e pelo Compilador sempre estão na memória, ou melhor, no "Buffer" de Objetos, estes dois módulos interagem diretamente com o Gerente do "Buffer" de Objetos, que é o componente do Gerente de Armazenamento responsável pelo gerenciamento dos objetos residentes no "Buffer" de Objetos.

O Processador MANO e o Compilador MANO estão descritos em detalhes nos capítulos IV e V respectivamente.

III.4.3 Gerente de Armazenamento

O Gerente de Armazenamento (GA) tem como funções primordiais a criação, **armazenamento** e recuperação de objetos. O GA é formado por três submódulos, que são: o Gerente do "Buffer" de Objetos, que gerencia os objetos residentes na memória principal; o Gerente do "Buffer" de **Páginas**, que gerencia as páginas carregadas na memória principal; e o Gerente de Disco é o responsável pelo gerenciamento dos objetos armazenados no disco.

Para executar um método, o Processador necessita manipular diversos objetos. Estes objetos são obtidos através do Gerente de Armazenamento, que os coloca disponíveis na memória. Durante a compilação de um método, o Compilador também faz acesso à objetos armazenados no banco de dados, como por exemplo, a classe (e suas superclasses) sob a qual o método está sendo compilado e a coleção de variáveis globais do sistema.

Os objetos existentes no banco de dados estão armazenados em disco ou em outros meios secundários. O Compilador e o Processador, manipulam os objetos somente em memória principal. Para fazer a transferência de objetos do disco para a memória principal, e vice-versa, o **ProtoGEO** gerencia uma memória virtual.

O mecanismo de memória virtual que está sendo empregado, utiliza um esquema de "buffer" duplo, isto é, a memória principal contém duas áreas de "buffer", um "buffer" de páginas e um "buffer" de objetos. Este esquema tem sido utilizado na construção de vários sistemas de banco de dados orientados a objetos, como o ORION, **O_2** e **GemStone,** e tem sua origem no sistema LOOM [KAEL83]. O GA está descrito em detalhes no capítulo VI.

Capítulo IV

Processador MANO

IV.1 Introdução

O Processador MANO é o componente do Gerente de Execução que tem a função de executar métodos. As instruções executadas são obtidas em instâncias da classe Método e correspondem a bytes, ou seja, valores de oito bits que possuem informações embutidas em seus bits.

Para executar um método, o Processador utiliza como estrutura de dados, um objeto que é instância da classe Contexto. Contextos são objetos que armazenam em suas variáveis de instância, o estado do Processador durante a execução de um método. Cada contexto possui uma pilha que é usada na execução do método. Para cada método a ser executado é criado um novo contexto que é descartado logo após o término da execução do método.

O terceiro tipo de objetos manipulados, são as instâncias da classe Classe. Estes objetos são usados para que o Processador possa pesquisar a tabela de mensagens da classe do objeto que recebeu a mensagem. Outra informação obtida nestes objetos é a superclasse da classe que está sendo pesquisada. O Processador utiliza a superclasse para continuar a busca, caso a mensagem procurada não seja encontrada na tabela de mensagens pesquisada.

O Processador possui também um conjunto de rotinas primitivas (escritas em C++), que foram desenvolvidas para otimizar a execução das operações que são usadas com maior frequência, como por exemplo, as operações aritméticas, lógicas, de manipulação de cadeias, de entrada/saída, etc. Estas operações são executadas sem que haja a necessidade da troca de contexto, ou seja, não são criados contextos para a execução dos métodos que estiverem associados à rotinas primitivas.

IV.2 Estruturas de Dados

Nesta seção descreve-se detalhadamente, os objetos que correspondem às estruturas de dados utilizadas pelo Processador.

Métodos

A classe ***Método*** descreve os objetos cujo conteúdo representa um método já traduzido, ou seja, um conjunto de instruções MANO. Instâncias da classe ***Método*** são criadas única e exclusivamente pelo Compilador.

Além das instruções, o método possui um conjunto de apontadores, ou melhor, as identidades dos objetos que aparecem no método, mas que não podem ser referenciados diretamente por uma instrução MANO. Estes objetos são chamados de literais e podem ser, por exemplo, seletores de mensagens, nomes de variáveis globais, alguns objetos constantes como símbolos, cadeias e etc.

As instruções só podem referenciar diretamente algumas categorias de objetos, que são: o receptor e os argumentos da mensagem, os valores das variáveis de instância do objeto receptor, os valores das variáveis temporárias declaradas no método, os objetos especiais (verdadeiro, falso, nulo, -1, 0, 1 e 2) e mais um conjunto de seletores especiais, que são mostrados na figura 4.1.

Os objetos que não podem ser referenciados diretamente por uma instrução, aparecem sempre como literais do método. Mesmo que um objeto apareça mais vezes no método, ele será gravado uma única vez como literal.

As variáveis temporárias, quando declaradas, só podem ser acessadas dentro do método. O Processador sabe, a partir de informações gravadas no método, o número de variáveis temporárias declaradas no método. Os valores das variáveis temporárias, bem como dos argumentos do método, são armazenados no contexto criado para sua execução, como será visto mais adiante. Como, ao final da execução de um método, o contexto é descartado, então os valores das variáveis temporárias também são perdidos.

```
+     *     em:          copieBloco:
-     /     em:mud:      avalie
<     \     tam          avalie:
>     @     prox         faça:
<=    <<    proxmud:     novo
>=    \\    fim          novo:
=     &     --           x
~=    |     classe       Y
```

Figura 4.1: Tabela de Seletores Especiais

Todas as variáveis globais existentes no sistema estão armazenadas em instâncias da classe **predefinida** chamada Associação. Uma associação é um objeto que associa uma cadeia de caracteres, correspondente ao nome da variável global, com seu valor, que na verdade é a identidade do objeto referenciado pela variável global.

Quando um método utiliza uma variável global, a identidade da instância de Associação cuja cadeia de caracteres equivale ao nome da variável global encontrada, é armazenada no método como um literal.

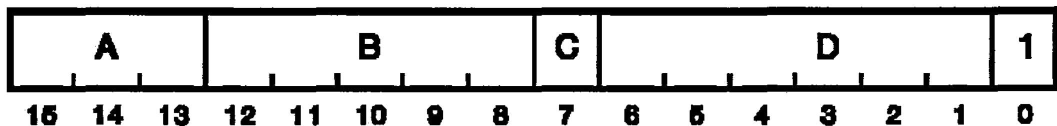
Além das instruções e dos literais, todo método possui um cabeçalho. Este **cabeçalho** é uma instância da classe Inteiros que contém quatro campos de informações codificados em seus bits, como mostra a figura **4.2.**

O campo ***número de temporárias*** informa ao Processador o número de variáveis temporárias usadas no método, incluindo o número de argumentos.

O campo ***tipo de contexto*** indica o tamanho do contexto que deverá ser criado. Um contexto pode ser criado com dois tamanhos diferentes, como será visto mais a frente.

O campo ***número de literais*** além de informar, como o próprio nome já diz, o número de literais do método, indica também a posição da primeira instrução do método.

O campo ***código*** serve para identificar alguns tipos especiais de métodos. Normalmente, indica simplesmente o número de argumentos do método. Quando o método está associado a alguma rotina primitiva, então ele passa a ter outros significados, como mostra a tabela da figura **4.3.**



A - código

B - número de temporárias

C - tipo de contexto

D - número de literais

1 - significa que é um inteiro

Figura 4.2: Cabeçalho de Método

Valor do código	Significado
de 0 a 4	Sem primitivas, contendo de 0 a 4 argumentos
5	Primitiva especial retorno de $mim (sem argumentos)
6	Primitiva retorno de variável de instância (sem argumentos)
7	O método possui uma extensão do cabeçalho

Figura 4.3: Valores possíveis do campo código

Da mesma forma que o Interpretador Smalltalk, o Processador MANO dá um tratamento especial a dois tipos de métodos que ocorrem com muita frequência. No primeiro tipo estão aqueles métodos que simplesmente retomam o objeto receptor da mensagem, ou seja, retomam o valor da variável especial $mim. O segundo tipo engloba os métodos que simplesmente retomam uma das variáveis de instância do objeto receptor. Estes dois tipos de métodos são otimizados pelo Compilador. Nestes casos, os métodos gerados não possuem nenhum literal ou instrução, isto é, são formados apenas pelo seu cabeçalho, que possui no campo código os valores 5 ou 6 conforme o caso.

Quando código é igual a seis, o índice da variável de instância a retomar é encontrado no próprio cabeçalho, na posição correspondente ao campo número de temporárias.

Quando o método está associado a uma rotina primitiva, ou seja, o campo código possui o valor sete, após o último literal do método é gravado uma outra instância de Inteiros, que é a extensão do cabeçalho. Nesta extensão estão codificados dois campos extras: o número de argumentos do método e o número da rotina primitiva associada.

cabeçalho	
literal 1	
literal 2	
: : :	
literal N	
Instr. 1	Instr. 2
instr. 3	instr. 4
Instr. 5	Instr. 6
: : :	Instr. N

Figura 4.4: Estrutura de um método

Por último, todo método que contém uma extensão de cabeçalho ou que envia uma mensagem do tipo superclasse, isto é, uma mensagem para a variável especial *$super*, possui como último literal uma instância da classe Associação. Esta associação mantém no seu campo de valor, a classe que contém a tabela de mensagens onde o método poderá ser encontrado. A figura 4.4 mostra a estrutura completa de um método já compilado.

Contexto

O segundo tipo de objeto usado pelo Processador MANO, são as instâncias da classe ***Contexto.*** Um contexto é um objeto que armazena o estado do Processador durante a execução de um método.

Quando o Compilador encontra uma mensagem sendo enviada a um objeto dentro do método, normalmente traduz esta mensagem em uma instrução de envio de mensagens. Para a execução dessa instrução, é possível que um outro método tenha que ser executado. Nestes casos, o estado atual do Processador, ou seja, o contexto que está sendo usado (contexto ativo), tem que ser salvo para que o novo método possa ser executado. Isto é feito colocando-se o contexto ativo no estado suspenso e fazendo com que o contexto criado para a execução do novo método fique ativo.

Ao final da execução deste novo método, o contexto que foi suspenso é reativado para que o método original possa ser concluído. A figura 4.5 mostra a estrutura de um contexto e seu método associado.

O campo ***remetente*** armazena a identidade do último contexto suspenso e é usado pelo Processador para reativá-lo ao término da execução do método corrente. O campo ***apontador de instrução*** indica a próxima instrução a ser executada pelo Processador. ***Apontador*** de ***pilha*** contém o índice onde está localizado o objeto que está no topo da pilha de execução. O campo ***método*** indica o método ao qual o contexto está associado e ***Receptor*** contém a identidade do objeto que recebeu a mensagem que provocou a ativação do ***método.***

Estes cinco primeiros campos são fixos e estão presentes em todos os contextos. Os dois campos seguintes são dimensionados de acordo com a especificação contida no cabeçalho do método.

Como foi dito anteriormente, os valores relativos aos argumentos do método e das variáveis temporárias são armazenados no contexto. Isto faz com que estes valores sejam perdidos ao término da execução do método. Se o método não tiver argumentos ou variáveis temporárias, então esses campos não existirão.

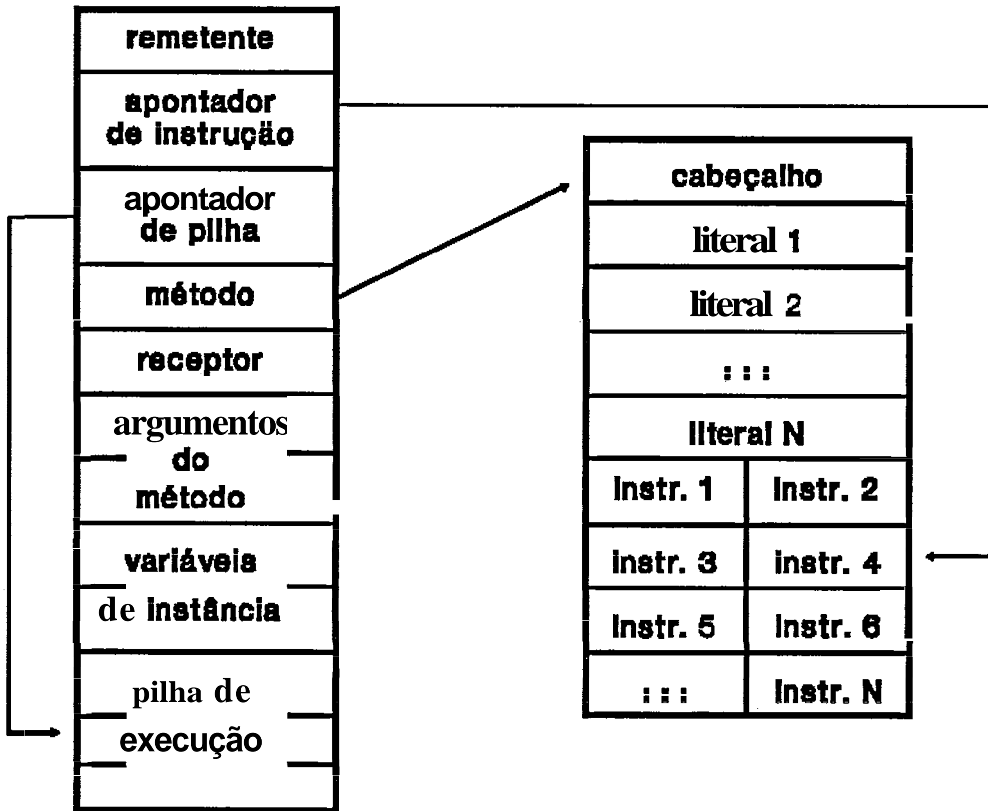


Figura 4.5: Contexto de Execução

Os últimos campos de um contexto são usados para formar uma pilha de execução que é usada pelo Processador.

Existem dois tamanhos predeterminados de contextos, um com espaço para doze campos variáveis e outro com espaço para trinta e dois campos variáveis. O número de campos variáveis é formado pela soma do número de variáveis temporárias e a profundidade da pilha de execução. Estas limitações poderão ser facilmente relaxadas em versões futuras da linguagem MANO.

Contexto de Bloco

O Processador também utiliza um segundo tipo de contexto, que são os contextos de bloco. Contextos de bloco são instâncias da classe predefinida chamada *ContextoDeBloco*. Estes objetos são usados pelo Processador para a execução de um bloco de mensagens. Um contexto de bloco está diretamente relacionado com o contexto do método que contém o bloco que está sendo executado, inclusive compartilhando diversas informações com este.

A figura 4.6 mostra um contexto de bloco junto com o contexto principal e seu método associado.

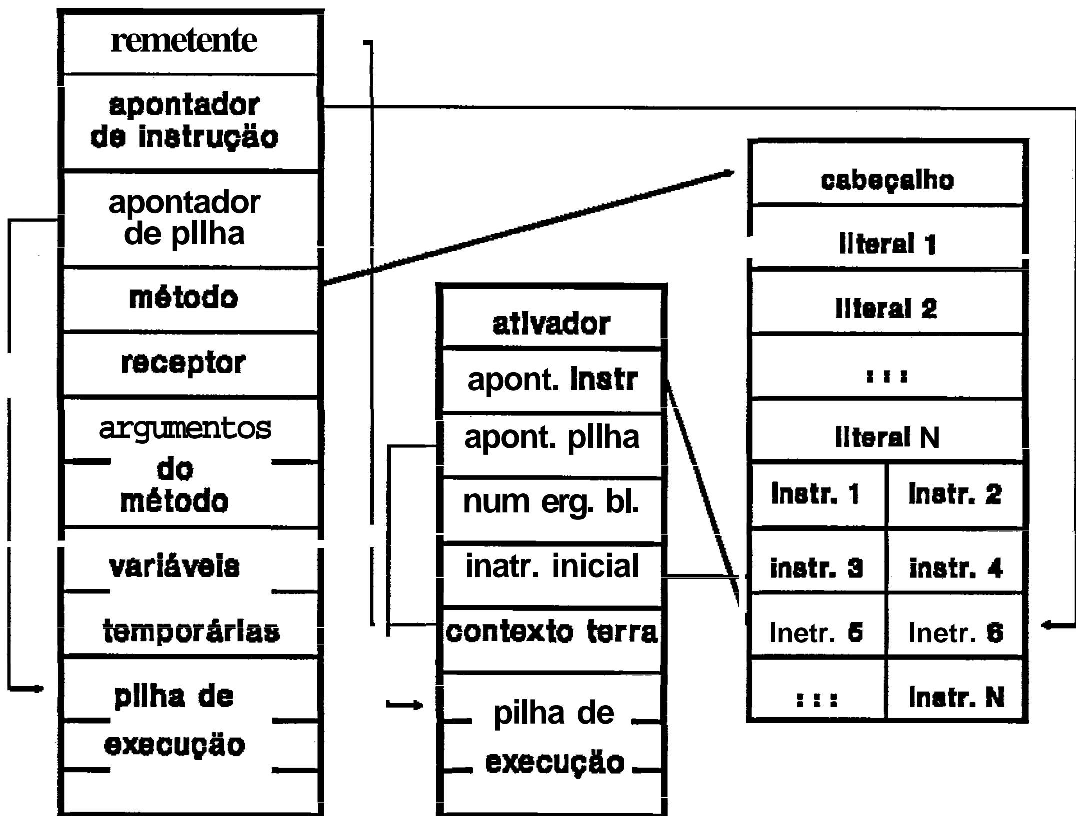


Figura 4.6: Contexto de Bloco

O campo ***ativador*** tem a mesma função do campo ***remetente*** contido em um contexto. **O ativador** é o contexto reativado quando termina a execução de um bloco de mensagens, enquanto que o remetente é o contexto reativado ao término da execução de um método. Um método termina quando sua última instrução for executada, ou após a execução de uma instrução de retomo, gerada quando o símbolo de circunflexo (^) precede uma expressão **MANO.**

Um contexto de bloco possui seus próprios ***apontadores de instrução e de pilha,*** porém os argumentos do método e as variáveis temporárias, inclusive os argumentos do bloco, são recuperados no ***contexto terra.***

O campo ***instrução inicial*** contém a posição no método onde está armazenada a primeira instrução referente ao bloco de mensagens. O ***contexto terra*** é o campo que contém a identidade do contexto que está associado ao método. Por último, cada contexto de bloco também possui sua própria ***pilha de execução.***

Um **contexto** de bloco C **criado pelo** Processador quando a mensagem de seletor especial ***#copieBloco:*** é enviada ao contexto ativo. Esta mensagem tem como parâmetro o número de argumentos do bloco.

Após a criação de um contexto de bloco, sua identidade é acrescentada na pilha de execução do contexto ativo. **A** execução de um bloco é ativada quando este recebe a mensagem de seletor especial ***#avalie.*** Em métodos da classe ***Coleção,*** por exemplo, o bloco de mensagens que estiver associado à mensagem enviada a uma coleção, será ativado através da mensagem de seletor ***#avalie:*** tendo como parâmetro os elementos da coleção.

Classes

As classes são objetos **MANO** cuja função é semelhante às declarações de tipos, existentes nas linguagens de programação convencionais. Nestas linguagens, normalmente os tipos são usados pelo compilador para que este possa fazer as verificações de tipos que são necessárias em linguagens tipadas. Na linguagem MANO, o

Compilador utiliza estes objetos para recuperar os nomes das variáveis de instância que podem aparecer no código fonte de um método.

Como na linguagem MANO o acoplamento de mensagens/métodos é feito dinamicamente, ou seja durante a execução de um método, então o Processador necessita recuperar informações contidas nas classes. O Processador especificamente, utiliza três campos de uma classe, que são: a tabela de mensagens, sua superclasse e também o campo que contém a especificação dos requisitos de memória necessários para a criação de suas instâncias. A figura 4.7 mostra a estrutura de uma classe. Os campos que aparecem nesta figura são os campos necessários para a implementação do núcleo do ProtoGEO. Outros campos ainda poderão ser adicionados, de acordo com as necessidades do projeto do metaesquema do ProtoGEO.

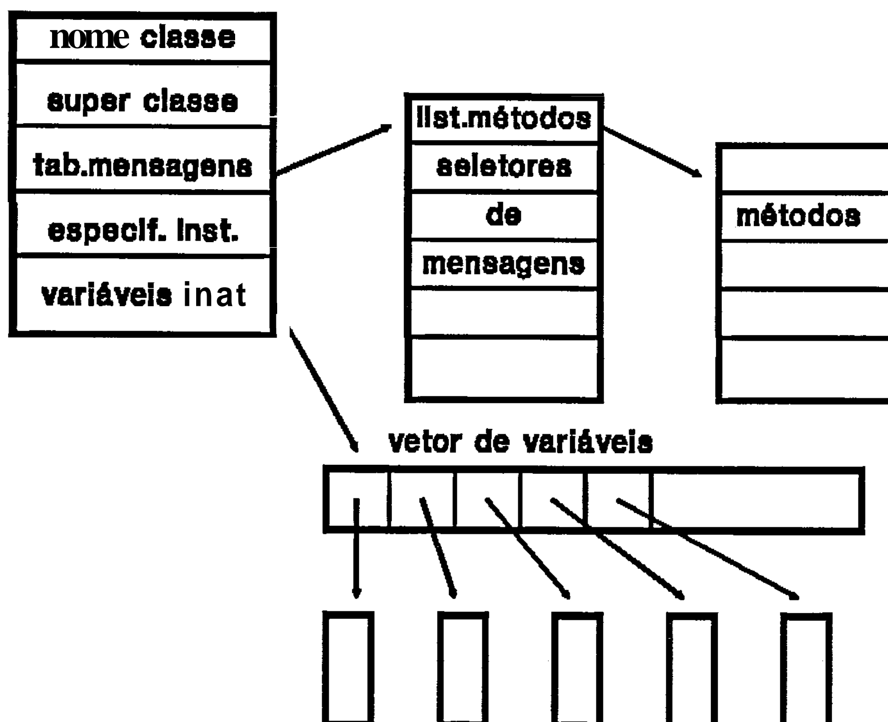


Figura 4.7: Estrutura de uma instância de Classe

O nome da classe é utilizado apenas pelo Compilador. Este nome é informado no momento da definição da classe e posteriormente, aparece junto ao código fonte dos métodos pertencentes à classe.

O campo ***superclasse*** é utilizado pelo Processador para percorrer a hierarquia de classes. Quando for introduzido o mecanismo de múltipla herança no sistema, este campo deverá ser alterado para conter, em vez da identidade da superclasse, a identidade da coleção que armazenará as identidades de suas superclasses.

O campo tabela de mensagens contém a identidade de uma instância da classe Array, onde são armazenados os seletores das mensagens. O primeiro campo deste "array" contém a identidade de outra instância de Array, onde podem ser recuperados os métodos associados a cada seletor de mensagem.

O campo ***especificação*** de instância é utilizado para definir o tipo das instâncias da classe. Ele possui uma instância da classe Inteiros que contém quatro informações embutidas, são elas: (a) se os campos das instâncias contém apontadores de objetos ou valores numéricos; (b) se os campos das instâncias são endereçados em quantidades de "byte" ou "word"; (c) se as instâncias possuem campos indexáveis; (d) o número de campos fixos da instância.

O último campo de uma classe contém a identidade de uma instância da classe Array onde estão armazenados os nomes das variáveis de instância da classe. Este campo só é usado pelo Compilador MANO.

IV.3 Instruções MANO

As instruções que são executadas pelo Processador MANO correspondem aos "bytecodes" usados pelo interpretador Smalltalk. Existem 256 instruções que podem ser classificadas em cinco categorias: instruções de pilha, instruções de armazenamento, instruções de envio de mensagens, instruções de desvio e instruções de retorno. A figura 4.8 mostra a tabela de instruções conforme especificada em [GOLD83].

INTERVALO	BITS	**FUNÇÃO**
0-15	**0000iiii**	Acrescente variável de instância **#iiii**
16-31	**0001iiii**	Acrescente posição temporária **#iiii**
32-63	**001iiiii**	Acrescente Literal Constante **#iiiii**
64-95	**010iiiii**	Acrescente Literal Variável **#iiiii**
96-103	**01100iii**	Tire e armazene na variável de instância **#iii**
104-111	**01101iii**	Tire e armazene na posição temporária **#iii**
112-119	**01110iii**	**Acrescente(receptor,vero,falso,nulo,-1,0,1,2)**
120-123	**011101ii**	Retorne(receptor,vero,falso,nulo) da mensagem
124-125	**0111110i**	Retorne topo da pilha da **(mensagem/bloco)** #i
126-127	0111111i	Não usado
128	10000000 jjkkkkkk	**Acrescente(Variável Inst.,posição** temporária, literal **const.,literal** Var.) **[jj] #kkkkk**
129	10000001 jjkkkkkk	Armazene (Variável **Inst.,posição** temporária, **ilegal,literal** variável) **[jj] #kkkkk**
130	10000010 jjkkkkkk	Tire e armazene (Var. **Inst.,posição** literal, **ilegal,literal** variável) **[jj] #kkkkk**
131	10000011 jjjkkkkk	Envie seletor literal **#kkkkk** com **jjj** argumentos
132	10000100 **jjjjjjjj** kkkkkkkk	Envie seletor literal **#kkkkkkkk** com **jjjjjjjj** argumentos
133	10000101 jjjkkkkk	Envie seletor literal **#kkkkk** para superclasse com **jjj** argumentos
134	10000110 **jjjjjjjj**	Envie seletor literal **#kkkkkkkk** para superclasse com **jjjjjjjj** argumentos
135	10000111	Tire topo da pilha
136	10001000	Duplique topo da pilha
137	10001001	Acrescente contexto ativo
138-143		Não usado
144-151	**10010iii**	Desvie **iii+1** (isto é, de 1 até 8)
152-159	**10011iii**	Tire e desvie se falso **iii+1**
160-167	**10100iii** **jjjjjjjj**	Desvie **(iii-4)*256+jjjjjjjj**
168-171	**101010ii** **jjjjjjjj**	Tire e desvie se verdadeiro **ii*256+jjjjjjjj**
172-175	**101011ii** **jjjjjjjj**	Tire e desvie se falso **ii*256+jjjjjjjj**
176-191	**1011iiii**	Envie mensagem aritmética **#iiii**
192-207	**1100iiii**	Envie mensagem seletor especial **#iiii**
208-223	**1101iiii**	Envie seletor literal **#iiii** sem argumentos
224-239	**1110iiii**	Envie seletor literal **#iiii** com 1 argumento
240-255	1111iiii	Envie seletor literal **#iiii** com 2 argumentos

Figura 4.8: Tabela de Instruções

Como as informações estão embutidas nos bits da instrução, isto faz com que sejam impostos limites muitas vezes rigorosos ao conjunto de instruções. Por exemplo, o primeiro grupo de instruções (de 0 a 15) é usado para acrescentar na pilha, variáveis de instância do objeto receptor. Porém, estes objetos só podem ter no máximo dezesseis variáveis de instância.

A maioria dos objetos não possui mais do que dezesseis variáveis de instância, caso isto ocorra, podem ser usadas as instruções estendidas. Uma instrução estendida possui um ou dois bytes a mais, nos quais estão embutidas informações com limites mais flexíveis. Por exemplo, a instrução 128 também permite colocar na pilha as variáveis de instância de um objeto, porém, este objeto poderá ter até 64 variáveis de instância.

As instruções de pilha indicam ao Processador, objetos a serem acrescentados no topo da pilha de execução. Estes objetos podem ser: o receptor da mensagem, as variáveis de instância do objeto receptor, os argumentos do método, as variáveis temporárias, os literais do método, o conjunto de objetos especiais (nulo, verdadeiro, falso, -1, 0, 1 *e* 2) e o objeto localizado no topo da pilha, isto é, permite a duplicação do topo da pilha.

Os argumentos do método e as variáveis temporárias são armazenados juntos no contexto de execução e são tratados indistintamente pelo Processador. Com isto, as instruções usadas para manipular os argumentos são as mesmas usadas para manipular as variáveis temporárias.

Já os literais do método são manipulados por dois conjuntos diferentes de instruções. *O* primeiro é usado para manipular os literais constantes, ou seja, aqueles que armazenam diretamente seu objeto (cadeias, números, seletores de mensagens, etc). O segundo permite a manipulação de literais variáveis, ou seja, os literais que armazenam instâncias da classe Associação, que são usadas para armazenar as variáveis globais.

As instruções de armazenamento indicam que o objeto localizado no topo da pilha deve ser armazenado em uma variável, que pode ser uma das variáveis de instância do objeto receptor, uma variável temporária ou uma variável global. Sempre que o Compilador encontra uma expressão contendo uma atribuição, a última instrução gerada é sempre uma instrução de armazenamento.

Existem dois tipos de instruções de armazenamento, as que removem o objeto da pilha para ser armazenado em uma variável, e as que deixam o objeto no topo da pilha após armazená-lo.

As instrucões de envio de mensagens indicam que uma mensagem está sendo enviada. Estas instruções contém informações sobre o seletor da mensagem a ser enviada e o número de argumentos que ela possui. Estas instruções são sempre precedidas no método por um conjunto de instruções responsáveis por colocar na pilha de execução o objeto receptor e os argumentos da mensagem.

Após o Processador executar uma instrução de envio, o que na maioria das vezes implica na execução de um novo método, tanto o objeto receptor como os argumentos da mensagem são removidos da pilha e o objeto resultante da mensagem executada é então colocado no topo da pilha de execução.

Instrucões de desvio são usadas pelo Compilador para otimizar a execução de estruturas de controle. Uma instrução de desvio indica ao Processador a próxima instrução a ser executada, que não necessariamente será a instrução que segue a instrução de desvio.

Existem dois tipos de instruções de desvio: condicional e incondicional. As instruções de desvio incondicional transferem o controle de execução, isto é, incrementam ou decrementam o apontador de instruções, sempre que são executadas. As instruções de desvio condicional são executadas dependendo do valor do objeto que está no topo da pilha. Algumas dessas instruções, transferem o controle de execução se o objeto no topo da pilha tiver valor verdadeiro e outras transferem se o objeto tiver valor falso.

As estruturas de controle otimizadas pelo Compilador são as expressões que utilizam as mensagens cujo seletores são: ***#seVero:***, ***#seVero:senão:*** e ***#enquanto:***.

Estas mensagens não são executadas pelo Processador da mesma forma que as outras. Nestes casos, o Processador não realiza a busca de métodos na hierarquia de classes e também não cria um contexto para sua execução.

A mensagem de seletor ***#seVero:*** possui como argumento um bloco de mensagens que só será executado caso o objeto receptor da mensagem seja o objeto verdadeiro. (Equivalente ao comando **IF-THEN** do Pascal)

A mensagem de seletor ***#seVero:senão:*** possui dois blocos de mensagens como argumentos, um dos **quais** será executado conforme o valor do objeto receptor. (Equivalente ao comando **IF-THEN-ELSE** do Pascal)

A mensagem de seletor ***#enquanto:*** possui como argumento um bloco de mensagens que será executado repetitivamente enquanto o objeto receptor for o objeto verdadeiro. (Equivalente ao comando WHILE-DO do Pascal)

As <u>**instruções** de retomo</u> indicam o término da execução do método e podem ser geradas em duas situações distintas. Uma instrução de retorno pode ser gerada após o Compilador ter analisado todo o método, ou seja, ao **final** do método. A outra situação é quando um símbolo de circunflexo ("^") é usado em uma expressão MANO.

Quando uma instrução de retorno é executada, o último contexto suspenso é reativado e o objeto resultante da mensagem executada é então colocado no topo da pilha de execução deste contexto.

Existem seis instruções de retomo, as cinco primeiras **reativam** o contexto cuja identidade encontra-se no campo ***remetente*** do contexto ativo e retomam um dos seguintes valores: o objeto receptor da mensagem ($mim), verdadeiro, falso, nulo ou o objeto que estiver no topo da pilha de execução. A sexta instrução de retomo é gerada quando o Compilador identifica o final de um bloco de mensagens. Neste caso, o contexto reativado é o contexto cuja identidade encontra-se armazenada no campo ***ativador*** do contexto corrente e o valor retomado é sempre o objeto que estiver no topo da pilha de execução.

IV.4 O Processador

O Processador MANO é uma máquina de pilha que tem a função de executar os métodos. Ele utiliza a pilha existente no contexto para executar a maioria das instruções. Por exemplo, as instruções de pilha indicam objetos para serem acrescentados na pilha, instruções de **armazenamento** indicam onde colocar os objetos que são retirados da pilha, instruções de envio de mensagens removem da pilha o objeto receptor e os argumentos da mensagem e as instruções de retomo acrescentam o resultado de uma mensagem na pilha do contexto reativado.

O Processador executa as instruções armazenadas em um método e utiliza um contexto para armazenar o seu estado. Por questões de eficiência, algumas informações do contexto são armazenadas em registradores e são manipuladas diretamente. Os registradores usados são os seguintes:

. contexto corrente - contém a identidade do contexto ativo.

. contexto terra - se o contexto corrente é um contexto de bloco, este registrador contém a identidade do contexto que está associado ao contexto de bloco, senão, contém o próprio contexto corrente.

. método corrente - contém a identidade do método que está sendo executado.

. receptor - contém a identidade do objeto que recebeu a mensagem responsável pela ativação do método corrente.

. apontador de instrução - contém o índice no método corrente, onde está armazenada a próxima instrução a ser executada.

. apontador de pilha - contém o índice do campo do contexto corrente referente ao topo da pilha.

Estes seis registradores correspondem às informações que são salvas no contexto corrente, quando este toma-se suspenso. *O* Processador possui mais quatro registradores que são usados para a execução das instruções de envio de mensagens. São eles:

- seletor da mensagem - contém a identidade do seletor da mensagem que está sendo enviada.

- número de argumentos - contém o número de argumentos da mensagem que está sendo enviada. É usado também para informar a posição na pilha onde está armazenado o receptor da nova mensagem, uma vez que este foi acrescentado na pilha antes dos argumentos.

- novo método - quando o Processador localiza a mensagem na hierarquia de classes, a identidade do método que estiver associado ao seletor da mensagem é armazenada neste registrador.

- índice da primitiva - se o novo método está associado a alguma rotina primitiva, seu índice é armazenado neste registrador.

O Processador é um dos componentes do Gerente de Execução (GE). Para executar uma mensagem, o GE segue os seguintes passos: primeiro o Compilador é ativado para traduzir a mensagem em um método contendo instruções MANO. Após a compilação, o GE cria o novo contexto, que será utilizado na execução do método. Em seguida, os campos do contexto são inicializados e uma rotina é ativada para carregar os registradores. Finalmente o Processador é ativado.

O Processador, quando ativado, executa repetidamente o seguinte ciclo de três passos:

1) Busca no método corrente a instrução **indicada** pelo apontador de instruções.

2) Incrementa o apontador de instruções.

3) Executa a rotina especificada pela instrução.

Este ciclo é repetido até que uma instrução de retorno seja executada ou quando uma situação de erro for detectada. Um erro pode ocorrer, por exemplo, quando um seletor de mensagem não é encontrado.

Se o método terminou normalmente, o Processador devolve o controle ao **GE** que tem a função de fornecer o resultado da mensagem. **O** objeto resultante da mensagem fica armazenado no topo da pilha do contexto criado para a execução do método.

As rotinas que executam cada uma das instruções foram implementadas conforme especificado em **[GOLD83].** A seguir são descritas as principais caracterlsticas de cada uma das categorias de instruções executadas pelo Processador.

Instruções de Pilha

As instruções de pilha utilizam sempre o contexto corrente, colocando ou retirando objetos no topo da pilha. As instruções que acrescentam objetos na pilha, especificam diretamente os seguintes objetos:

- O receptor da mensagem.
- As variáveis de instância do objeto receptor.
- As posições temporárias do contexto (variáveis temporárias e argumentos do método).
- Os literais presentes no método corrente.
- Os objetos de identidades constantes (verdadeiro, falso, nulo, -1, 0, 1 e 2).

Além dessas instruções, existe uma instrução que permite acrescentar na pilha o próprio contexto corrente (usada pelo Compilador) e uma instrução que duplica o topo da pilha (usada em situações onde as mensagens estão em cascata).

As instruções de armazenamento, utilizam a pilha transferindo objetos no sentido oposto ao descrito anteriormente, ou seja, são instruções que informam ao Processador que o objeto no topo da pilha deve (ou não) ser retirado da pilha e armazenado em uma das seguintes variáveis:

- variáveis de instância do objeto receptor

- variáveis temporárias (armazenadas no contexto corrente)
- variáveis globais (instâncias da classe ***Associação*** armazenadas como literais do método corrente)

Instruções de Desvio

As instruções de desvio alteram o campo ***apontador de instruções*** do contexto corrente. Estas instruções somam ou subtraem um determinado valor que corresponde ao deslocamento indicado na instrução.

As instruções de desvio incondicionais alteram o ***apontador de instruções*** sempre que são executadas. As instruções de desvio condicional, primeiro removem o objeto do topo da pilha. Algumas instruções fazem o desvio se esse objeto for verdadeiro, outras se o objeto for **falso.** Se **o objeto** removido não for o objeto verdadeiro ou o objeto **falso,** uma situação de erro é detectada e o processamento é interrompido.

Instruções de Envio de Mensagens

As instruções de envio de mensagens causam a execução de um novo método ou de uma rotina primitiva, e determinam o seletor da mensagem e o número de argumentos que esta mensagem possui. O receptor da mensagem e os argumentos são previamente colocados na pilha de execução.

Os seletores indicados pelas instruções são encontrados nos literais do método corrente ou pertencem ao conjunto de seletores especiais utilizados pelo sistema (figura **4.1).**

Quando uma mensagem é enviada, o Processador realiza uma busca na hierarquia de classes para localizar o método que deverá ser executado. Esta busca é feita seguindo-se os seguintes passos:

1. Obter o receptor da mensagem, que está armazenado na pilha de execução logo abaixo dos argumentos da mensagem.

2. Identificar a classe do objeto receptor.

3. Verificar se a tabela de mensagens da classe do objeto receptor contém o seletor procurado.

4. Se encontrou, carrega no registrador ***novo método*** a identidade do método correspondente.

5. Se o seletor não foi encontrado, verifica na tabela de mensagens da superclasse da classe pesquisada (volta ao item 4).

A busca é realizada subindo-se a hierarquia de classes até a classe ***Objeto*** (raiz da hierarquia de classes). Se o seletor não foi encontrado, uma situação de erro foi detectada e o processamento é interrompido.

Se a busca termina com sucesso, o Processador executa o novo método. Antes porém de ativar a execução do novo método, o Processador verifica se o método está associado a alguma rotina primitiva. Caso esteja, a rotina primitiva é executada. Se ocorrer uma falha da primitiva (ver seção IV.5 - Rotinas Primitivas) ou se o método não estiver associado à rotina primitiva, então o Processador executa o novo método.

Para ativar o novo método, os seguintes passos são necessários:

1. Criar e inicializar um novo contexto.

   1.1 campo ***remetente*** = contexto corrente

   1.2 campo ***apontador de instruções*** = posição da primeira instrução do novo método

   1.3 campo ***apontador de pilha*** = primeira posição após as posições temporárias do contexto

   1.4 campo ***método*** = novo método

1.5 Transferir o receptor e os argumentos da mensagem, da pilha do contexto corrente para suas respectivas posições no novo contexto.

2. Alterar o registrador contexto corrente para apontar para o novo contexto

Após realizar o passo 2, o Processador automaticamente estará executando o novo método.

Uma variação que ocorre nestes procedimentos descritos anteriormente, é quando um mensagem é enviada à variável ***$super.*** Neste caso, quando o Processador vai realizar a busca na hierarquia de classes, a primeira classe a ser pesquisada é a superclasse da classe do objeto receptor.

Instruções de Retorno

As instruções de retorno indicam o término da execução do método corrente. Quando o Processador executa uma instrução de retorno, duas funções principais são executadas: a reativação do último contexto suspenso e o retorno do objeto resultante da mensagem.

A reativação do último contexto suspenso, é feita alterando o registrador contexto corrente para receber o campo remetente ou o campo ativador, conforme especificado pela instrução. Após a reativação do contexto, o objeto resultante (também especificado pela instrução) é acrescentado no topo do pilha do contexto reativado.

IV.5 Rotinas Primitivas

Rotinas primitivas são procedimentos (métodos) escritos em C++ que tem como objetivo executar, de uma forma mais eficiente, algumas operações que são usadas com maior frequência.

Quando uma instrução de envio de mensagens é executada, o Processador normalmente ativa um novo método. Para executar o novo método, o Processador precisa realizar diversas operações extras, como por exemplo, criar um novo contexto, carregar os campos deste novo contexto, suspender o contexto corrente e ativar o novo contexto. Além disso, ao término da execução do novo método, o contexto que foi suspenso tem que ser reativado, o que implica em nova carga dos registradores.

Uma rotina primitiva é executada pelo Processador sem que seja criado um novo contexto e sem executar mais nenhuma instrução. A rotina primitiva remove da pilha os argumentos e o receptor da mensagem, executa a operação específica e coloca na pilha o objeto resultante da mensagem. Após a execução de uma rotina primitiva, o Processador continua normalmente a execução das instruções do método.

Eventualmente, uma rotina primitiva pode falhar, isto é, a rotina não pode fornecer o resultado esperado. Nestes casos, o Processador irá executar o método ao qual a rotina está associado, como se não existisse a rotina primitiva.

Uma rotina primitiva pode falhar por exemplo, quando os argumentos encontrados na pilha não estão de acordo com o esperado. Suponha que a rotina primitiva que executa a operação de divisão inteira não recebeu valores inteiros como argumentos, ou se o valor calculado não for um valor inteiro, então a rotina irá falhar e o método ao qual a rotina está associado será ativado.

O interpretador **Smalltalk** possui uma grande quantidade de rotinas primitivas. No protótipo ProtoGEO, apenas uma parte destas rotinas foram implementadas. **A** seguir são descritas as rotinas primitivas implementadas no ProtoGEO.

Primitivas Aritméticas

Um conjunto de rotinas primitivas foram implementadas para executar as operações aritméticas envolvendo os objetos da classe Inteiros. São elas:

. Soma

. Subtrai

. Multiplica

. Divisão Exata

. resto da Divisão

. Divisão com Arredondamento

. Deslocamento de Bits (shift)

. BitAnd

. BitOr

Todas essas rotinas retiram da pilha de execução, o argumento e o objeto receptor, calculam a operação específica e acrescentam na pilha o objeto resultante. Se a primitiva falhar, o receptor e o argumento são colocados de volta na pilha para que o método associado possa ser executado.

Também foram implementadas rotinas primitivas para executar operações cujos operadores são operadores relacionais e os operandos são objetos da classe Inteiros. São elas:

. Menor (<)

. Maior (>)

. Menor ou Igual (<=)

. Maior ou Igual (>=)

. Igual (=)

. Diferente (~=)

Da mesma forma que as rotinas aritméticas, estas rotinas retiram da pilha o argumento e o objeto receptor e colocam na pilha o objeto resultante da mensagem, que nestes casos é o objeto verdadeiro ou o objeto falso.

Primitivas de Manipulação de Cadeias

Neste grupo estão as rotinas primitivas implementadas para fazer a manipulação dos objetos cujos campos são indexáveis (array, stream, etc). São elas:

. Em - retorna o valor do campo do objeto receptor cujo índice é o argumento retirado da pilha

. EmMud - altera o valor do campo do objeto receptor cujo índice é o primeiro argumento, para conter o valor do segundo argumento

. Tamanho - retorna o número de campos indexáveis do objeto receptor

Primitivas de Gerenciamento de Memória

Neste grupo estão rotinas que permitem manipular a identidade dos objetos e criar novas instâncias. São elas:

. Novo - permite criar uma nova instância de classes que não possuem campos variáveis

. Novo com Argumentos - permite a criação de uma nova instância com o número de campos indexáveis especificado pelo argumento

. Vira - possibilita a troca da identidade entre os objetos receptor e argumento

. Identidade - fornece um inteiro correspondente à identidade do objeto receptor

Primitivas de Controle

Neste grupo estão as duas rotinas que são usadas pelo Processador quando o método que está sendo executado possui blocos de mensagens, ou seja, quando um contexto de bloco precisa ser criado. A primeira delas é a rotina Primitiva Copie Bloco, que cria uma nova instância da classe ***ContextoDeBloco***.

Esta rotina está associada à mensagem de seletor especial ***#copieBloco:*** que pode ser enviada a um contexto ou um contexto de bloco. O número de argumentos que o bloco possui é passado como argumento. Se o receptor é uma instância de Contexto, então ele será o contexto terra do novo contexto de bloco. Se o receptor é uma instância de ***ContextoDeBloco***, então o seu contexto terra será também o contexto terra do novo contexto de bloco.

A outra rotina primitiva deste grupo é a Primitiva Avalie, que está associada à mensagem de seletor #avalie. **A** mensagem de seletor #avalie quando enviada a um bloco de mensagens, faz com que este bloco seja executado.

Esta rotina transfere os argumentos, da pilha do contexto corrente para a pilha do contexto receptor, armazena no campo ***ativador*** pertencente ao contexto receptor, a identidade do contexto corrente, inicializa os campos apontador de instruções e apontador de pilha do contexto receptor e faz com que o contexto receptor fique ativo, o que é feito atribuindo a identidade do contexto receptor ao registrador contexto corrente.

Primitivas do Sistema

Neste grupo, foram implementadas duas rotinas primitivas que estão associadas a métodos da classe Objeto. **A** primeira delas é a Primitiva Equivalente, **que** está associada à mensagem == (equivalente), pertencente à classe Objeto. Esta rotina retoma verdadeiro se o receptor e o argumento são o mesmo objeto, ou seja, possuem a mesma identidade. Caso contrário retorna falso.

A outra é a rotina Primitiva Classe, que está associada à mensagem #classe na classe Objeto. Esta rotina retorna a identidade da classe do objeto receptor.

Capítulo V

Compilador MANO

V.1 Introdução

O Compilador é o componente do Gerente de Execução, responsável pela tradução de métodos fontes (escritos em MANO) em métodos contendo código intermediário (instruções MANO). Além da compilação dos métodos, este Compilador também é usado para traduzir as mensagens que são enviadas aos objetos do sistema. Estas mensagens normalmente ativam a execução de métodos já compilados previamente e que estão armazenados no banco de dados.

No sistema ProtoGEO, os métodos fontes são armazenados em arquivos do tipo texto (seqüência de caracteres ascii) e os métodos traduzidos são gravados em instâncias da classe ***Método*** (conforme especificado na seção IV.1 Estruturas de Dados). Os métodos traduzidos também são objetos e por isso são armazenados e gerenciados da mesma forma que os demais objetos do banco de dados.

A linguagem MANO possui um precompilador que diferencia um método fonte de uma mensagem. Os métodos possuem cabeçalho e são precedidos do nome da classe, cuja tabela de mensagens contém o seletor da mensagem que está associada ao método. Já as mensagens não possuem cabeçalho e nem são precedidas por nomes de classes.

Quando o Precompilador identifica um método, após sua compilação, o par (seletor do método, método traduzido) é inserido na tabela de mensagens da classe sob a qual o método foi compilado. Quando identifica uma mensagem, o método traduzido tem sua execução ativada imediatamente após ter sido compilado.

Os arquivos contendo os métodos fontes podem ser interligados através do comando *de* inclusão, que é um comando interpretado pelo Precompilador. *O* comando de inclusão é constituído do símbolo '~' seguido do nome do arquivo a ser incluído, ou seja, a ser precompilado.

A figura 5.1 mostra o esquema de interligação de arquivos fonte, usado pelo Precompilador MANO.

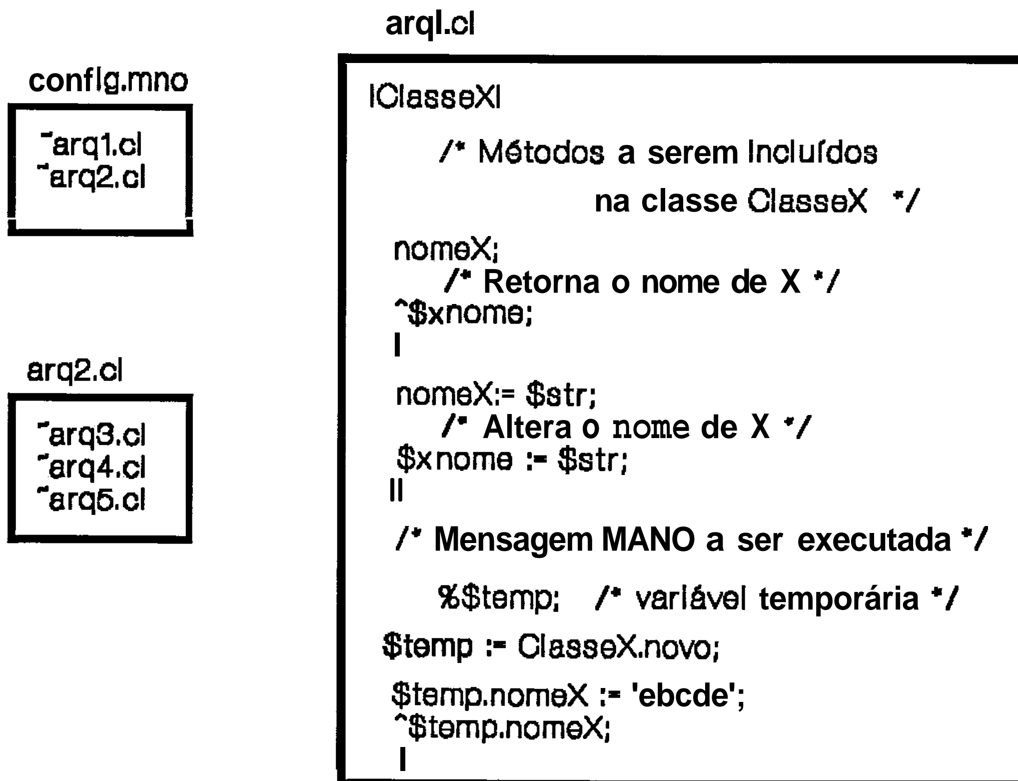


Figura 5.1: **Interligação** de Arquivos Fonte

O símbolo de exclamação ('!') indica ao Precompilador o final de um método ou mensagem. Duas exclamações ('!!') indicam o final de uma seqüência de métodos a serem compilados sob uma determinada classe.

Nesta primeira fase do projeto, o sistema ProtoGEO será carregado a partir da precompilação do arquivo **CONFIG.MNO.** Este arquivo conterá comandos de inclusão dos arquivos que formarão o metaesquema do ProtoGEO (em desenvolvimento na COPPE). Após a pré-compilação do arquivo **CONFIG.MNO,** uma janela de trabalho é aberta ao usuário para que ele possa interagir com o sistema. Nesta janela, o usuário pode inserir novos métodos no banco de dados, pode enviar mensagens aos objetos existentes no banco de dados e também pode solicitar a inclusão de novos arquivos previamente digitados.

O Compilador MANO foi implementado em C++, e seus principais componentes são o Analisador Léxico, Analisador Sintático, Gerador de Código e o Compilador propriamente dito. As seções que seguem descrevem detalhadamente cada um desses módulos

V.2 *O* Compilador

O Compilador foi construído como um conjunto de classes C++. A classe ***Compilador*** tem como principal método público, o método ***compile().*** Este método é responsável pela ativação dos demais módulos do Compilador e pelo armazenamento do código gerado durante a compilação do método.

O funcionamento do Compilador (figura 5.2) é o seguinte: primeiro o Analisador Sintático é ativado, se nenhum erro de sintaxe for detectado uma árvore sintática é gerada e a análise sintática termina com sucesso. Nesta fase também são identificadas as variáveis temporárias, as variáveis globais *e* os literais usados no método. O Analisador Sintático obtém do Analisador Léxico os símbolos identificados a partir da leitura do arquivo fonte. A seguir, o módulo Gerador de Código é ativado. A geração de código é

feita a partir do **percorrimento** da árvore sintática e o resultado desta fase é uma lista encadeada contendo as instruções geradas.

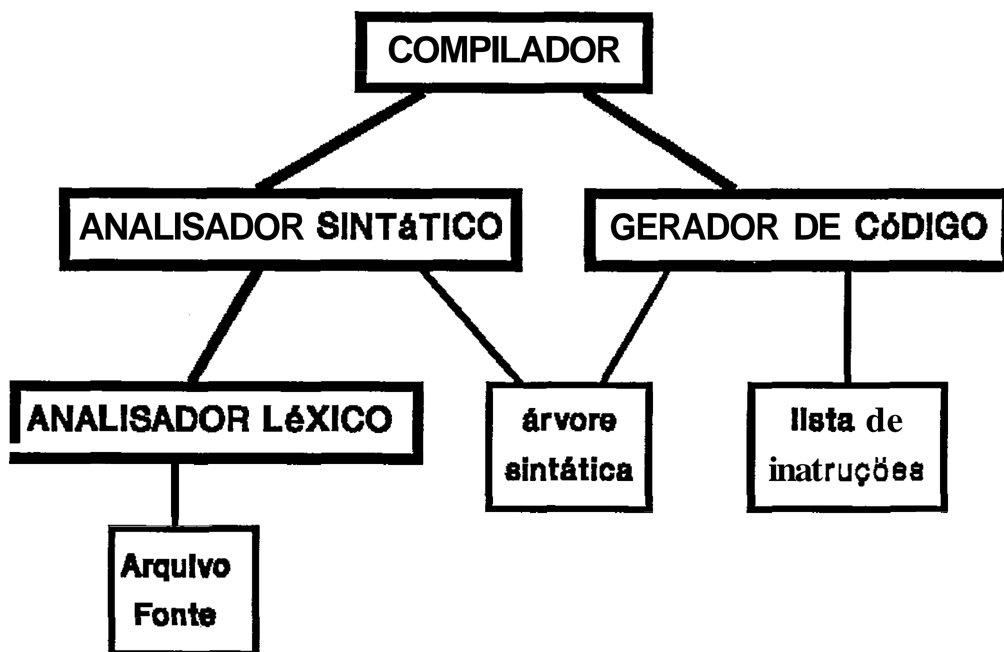


Figura 5.2: Estrutura do Compilador MANO

Após a geração de código, o Compilador armazena as informações geradas durante as fases de compilação, criando para isto uma nova instância da classe ***Método.*** Como foi visto na seção IV.2, a estrutura de armazenamento de um método é composta de um cabeçalho, seguido ou não de uma lista de identidades de objetos que formam os literais e por último são gravadas as instruções.

Após a gravação do método todo espaço de memória ocupado durante a fase de compilação é liberado. Isto inclui os espaços de memória ocupados pela árvore sintática, lista de instruções e outras estruturas temporárias.

Os módulos Analisador Sintático, Analisador **Léxico** e Gerador de Código utilizam algumas estruturas de dados que são definidas juntamente com a classe Compilador. Estas estruturas de dados são:

. Pilha Sintática - estrutura de dados do tipo pilha cujos elementos correspondem aos **nós** da árvore sintática.

. Pilha de Símbolos - estrutura de dados do tipo pilha cujos elementos correspondem a cadeias de caracteres.

. Fila de Comandos - estrutura de dados do tipo ***fila***, usada para encadear os diversos "comandos" na árvore sintática.

. Lista de Instrucões - estrutura de dados do tipo lista ***encadeada***, cujos elementos correspondem a instruções MANO.

. Lista de Símbolos - estrutura de dados do tipo lista ***encadeada***, cujos elementos correspondem à cadeias de caracteres.

As variáveis de instância da classe Compilador armazenam os dados necessários durante a compilação de um métodolmensagem, são elas:

. classe - identidade da classe sob a qual está sendo compilado o método

. nome - nome extraído do cabeçalho do método, que é usado como seletor da mensagem que fica associada ao método

. **numArgumentos** - número de argumentos do método

. **numInstrucões** - número de instruções geradas

. **numPrimitiva** - número da rotina primitiva associada ao método

. linha e coluna - posição do caractere lido do arquivo fonte

. **listComandos** - lista de comandos gerados na árvore sintática

. **listVarTemp** - lista de variáveis temporárias

. listLiterais - lista de literais encontrados no método

. listVarInst - lista dos nomes das variáveis de instância da classe

. listInstruções - lista que contém as instruções geradas pelo gerador de Código

. método - identidade do método traduzido

A classe Compilador também possui alguns métodos que são responsáveis pelo tratamento dos erros que por ventura apareçam. Quando uma situação de erro ocorre, a compilação do método/mensagem é cancelada e uma mensagem, contendo a descrição do erro e a posição no texto (linha e coluna) onde o erro foi detectado, é enviada ao usuário.

V.3 Analisador Léxico

O Analisador Léxico é o componente do Compilador MANO, responsável pela identificação dos símbolos pertencentes a sintaxe da linguagem.

A linguagem possui uma sintaxe bastante simples. Não existem palavras reservadas e todos os "comandos" possuem a forma de envio de mensagens, isto é, possuem um objeto receptor, um seletor de mensagem e se for o caso o(s) argumento(s) da mensagem.

A figura 5.3 mostra a gramática da linguagem, escrita na forma BNF ("Backus-Naur-Form"). A convenção utilizada é a seguinte:

1) Os símbolos entre colchetes ([ ]) são opcionais;
2) Símbolos entre chaves ({ }) podem ocorrer de forma repetitiva;
3) Símbolos entre apóstrofo ('') definem símbolos terminais;
4) Símbolos separados por (|) são símbolos alternativos;
5) O símbolo de igualdade (=) permite a definição de símbolos não terminais.

```
método     = cab ';' [primit] {temp ';'} comando ';' {comando ';'}
cab        = ( seqid [cabAtrib | cabChaves]
           | '[' identVar ']' [cabAtrib]
           | operador identVar
           | cabChaves )
cabAtrib   = ':=' identVar
cabChaves  = {chave identVar}
primit     = '«' número '»'
temp       = '%' identVar
comando    = ['^'] expr
expr       = msg {',' msgCascat}
msg        = exprChv {':=' exprChv}
msgCascat  = ( [seqid] {chave exprlog)
             | seqid )
exprChv    = exprlog [ [seqid] {chave exprlog) ]
exprlog    = relação {opLog relação)
relação    = exprArit {opRel exprArit}
exprArit   = parcela {opSoma parcela)
parcela    = fator {opMul fator)
fator      = {seqId prep} secundário
secundário  = primário {( '.' seqid | índice )}
índice     = '[' expr ']'
primário   = ( identVar | número | hexadec | nome | carac
             | cadeia | bloco | '(' expr ')' )
bloco      = '{' {paramBl ';'} comando ';' {comando ';'} '}'
paramBl    = '%' identVar
operador   = ( opLog | opRel | opAd | opMul )
opLog      = carLog {carOp}
opRel      = carRel {carOp}
opAd       = carAd {carOp}
```

Figura 5.3: **Gramática** da Linguagem MANO (parte 1)

```
opMul       = carMul {carOp}
carOp       = ( carLog | carRel | carAd | carMul | ':' )
carLog      = ( '&' | '|' | '~' )
carRel      = ( '=' | '<' | '>' )
carAd       = ( '+' | '-' | '±' )
carMul      = ( '*' | '/' | '÷' )
seqid       = ident {ident)
identVar    = '$'ident
chave       = ident ':'
prep        = '_' ident
nome        = ( '#' ident {chave) | '#' operador )
ident       = let {( '_'| let | dig ))
número      = dig {dig)
carac       = '\' carAscii
hexadec     = '@' carHex {carHex}
cadeia      = ( ''' {carCad1} ''' | '"' {carCad2} '"' )
carCad1     = carAscii exceto '''
carCad2     = carAscii exceto '"'
dig         = ( 0 .. 9 )
let         = ( a .. z | A .. Z | letras acentuadas )
carHex      = ( dig | A .. F | a .. f )
carAscii    = qualquer caractere ascii
```

Figura 5.3: Gramática da Linguagem MANO (parte 2)

Os símbolos identificados pelo Analisador Léxico e que são passados ao Analisador Sintático estão mostrados na figura 5.4. Trechos de comentários são descartados diretamente pelo Analisador Léxico.

```
FimDeArq    - caractere final de arquivo
ident       - identificador
chave       - chave ( xxx: )
identVar    - identificador de variável ( $xxx )
prep        - preposição ( _xxx )
nome        - nome ( #xxx )
número      - seqüência de dígitos
caracter    - caractere ascii ( \x )
cadeia      - cadeia de caracteres ( 'xxx' ou "xxx" )
bloco       - bloco de mensagens ( {} )
índice      - mensagem tipo índice ( [] )
opLog       - operador lógico
opRel       - operador relacional
opAd        - operador adicional
opMul       - operador multiplicativo
hexdec      - número em hexadecimal ( @xxx )
delimit     - delimitador (; [ ] « » ^ , . ( ) { } % ! := %%)
nDef        - símbolo não definido
```

Figura 5.4: Conjunto de Símbolos da Linguagem

V.4 Analisador Sintático

A função do Analisador Sintático é analisar sintaticamente os **métodos/mensagens**, isto é, verificar se as sentenças foram construídas obedecendo as regras gramaticais **definidas** para a linguagem (figura 5.3).

O Analisador Sintático gera, a partir dos símbolos fornecidos pelo Analisador Léxico, uma árvore sintática que posteriormente é utilizada pelo Gerador de Código.

Após a criação da árvore sintática, é feita uma verificação para conferir se são válidos, os símbolos que aparecem ao lado esquerdo de um "comando" de atribuição (: =), como está descrito mais adiante.

O Analisador Sintático recebe do Precompilador a informação que diz se o trecho a ser analisado corresponde a um método ou a uma mensagem MANO. Se for um

método, o Analisador Sintático identifica inicialmente o cabeçalho do método, extraindo dele o seletor da mensagem ao qual o método estará associado.

O seletor de um método é formado pela concatenação dos símbolos que constituem o cabeçalho do método, excluindo-se os identificadores dos argumentos do método.

Quando um método está sendo analisado, após a formação do cabeçalho, o Analisador Sintático ainda verifica se este método está associado a alguma rotina primitiva. A partir deste ponto, métodos e mensagens são analisados indistintamente.

A árvore sintática

A estrutura da árvore sintática gerada pelo Analisador Sintático é na verdade uma lista encadeada de sub-árvores, como será visto a seguir.

Dentro de um método, cada mensagem é finalizada com o símbolo ponto-e-vírgula (;). Por questões de clareza, trata-se estas mensagens por ***comandos.*** Com isto, pode-se dizer que os métodos ou mensagens MANO podem possuir vários ***comandos.***

Um ***comando*** pode ser precedido do símbolo de circunflexo (^), o que significa que o objeto resultante deste ***comando*** deverá ser retomado como resposta do método quando este for executado.

Cada ***comando*** analisado é acrescentado na árvore sintática, a partir da inserção de um novo elemento na lista de comandos da árvore sintática. O apontador da cabeça da lista de comandos é uma variável de instância da classe ***Compilador.*** A figura ***5.5*** exemplifica a lista de comandos da árvore sintática.

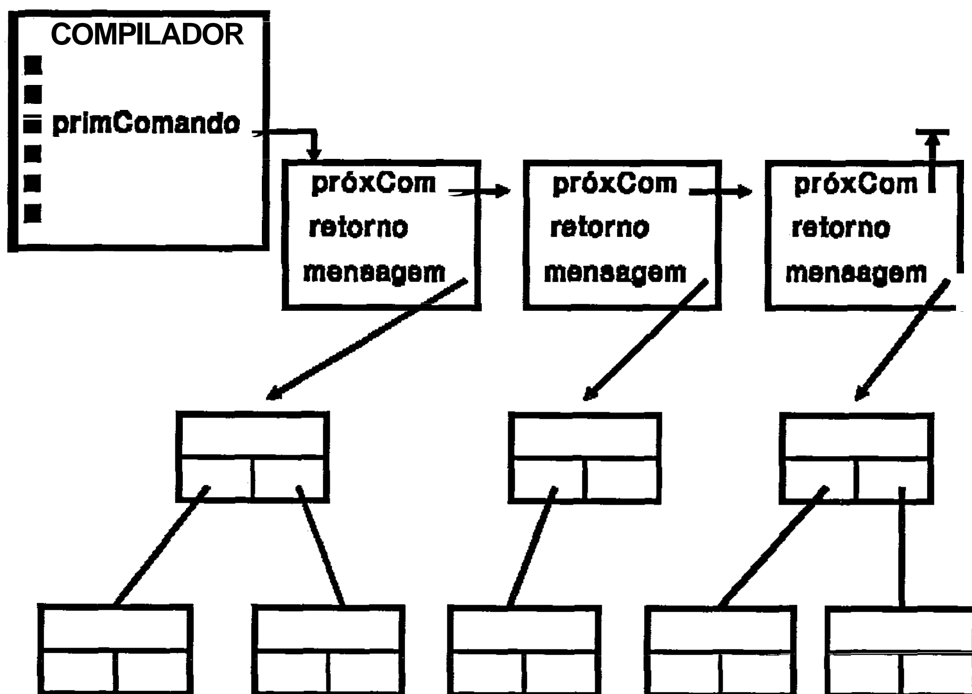


Figura 5.5: Lista de comandos da árvore sintática

Cada elemento da lista de comandos possui: um apontador para o próximo elemento da lista, um valor booleano indicando se o comando é um comando de retorno e um apontador para o nó cabeça da sub-árvore sintática gerada para o comando.

Cada nó da sub-árvore sintática é uma instância da classe C++ chamada MSG (mensagem). Uma MSG possui as seguintes variáveis de instância:

. receptor - apontador para a MSG que define o receptor da mensagem

. tipoSeletor - tipo do seletor da mensagem (figura 5.4)

. seletor - cadeia de caracteres correspondente ao seletor da mensagem

. argumento - apontador para a MSG que define o argumento da mensagem

Os campos de cada MSG serão representados nos exemplos deste capítulo, como mostrado na figura 5.6:

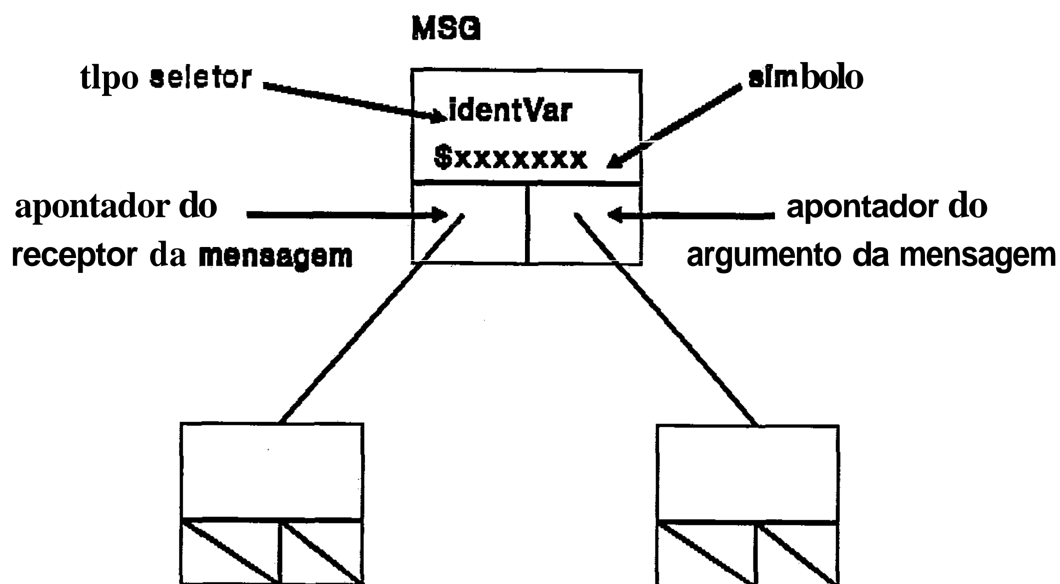
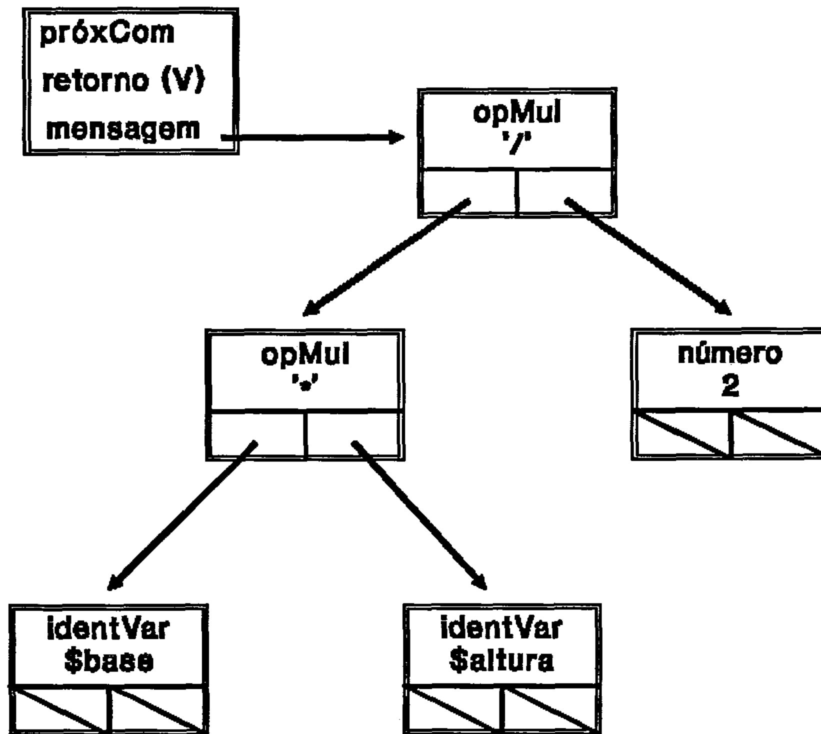


Figura 5.6: Representação do nó da árvore sintática

A figura 5.7 exemplifica um método escrito em MANO (a) e sua respectiva árvore sintática (b).

```
|Janela|
área;
   /* Calcula a área do objeto receptor */
^ ( $base * $altura ) / 2;
```

(a) Método em MANO



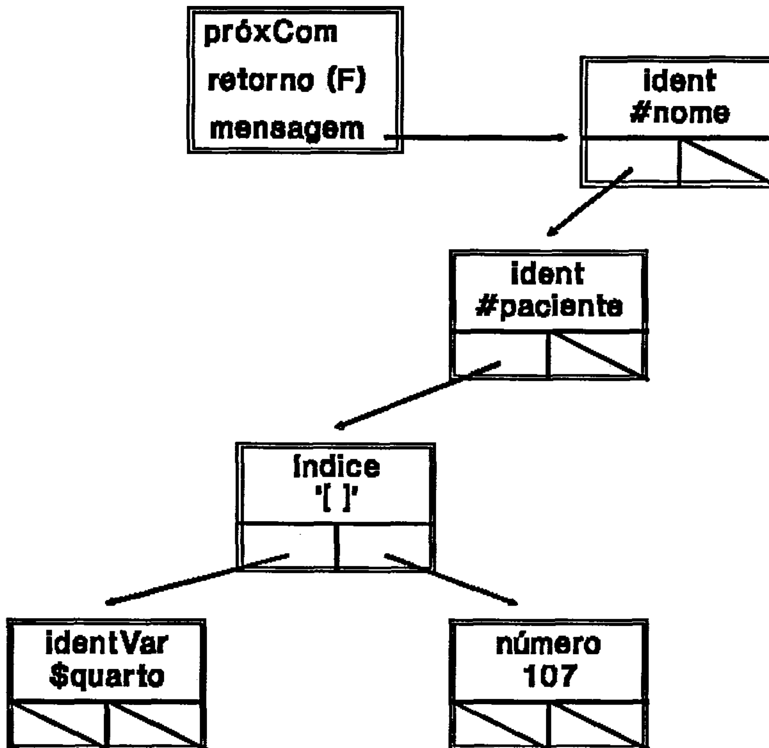
(b) árvore sintática

Figura 5.7: Exemplo de árvore sintática

A linguagem MANO permite ao usuário escrever as mensagens tanto na forma pré-fixa como na forma pós-fixa. A árvore sintática gerada para essas duas formas de mensagens é idêntica. A figura 5.8 mostra uma mesma mensagem escrita nas duas formas (a) *e* a árvore sintática que é gerada pelo Analisador Sintático (b).

```
$nome := nome _do paciente _do $quarto[107];
$nome := $quarto[107].paciente.nome;
```

(a) Mensagens MANO



(b) árvore sintática

Figura 5.8: Mensagem nas formas pré e pós-fixa

Se o ***comando*** analisado possui mais de um argumento, uma instância de MSG–CHAVE é usada. A classe MSG–CHAVE é uma subclasse de MSG e, além das variáveis de instância herdadas da superclasse, define as seguintes variáveis de instância:

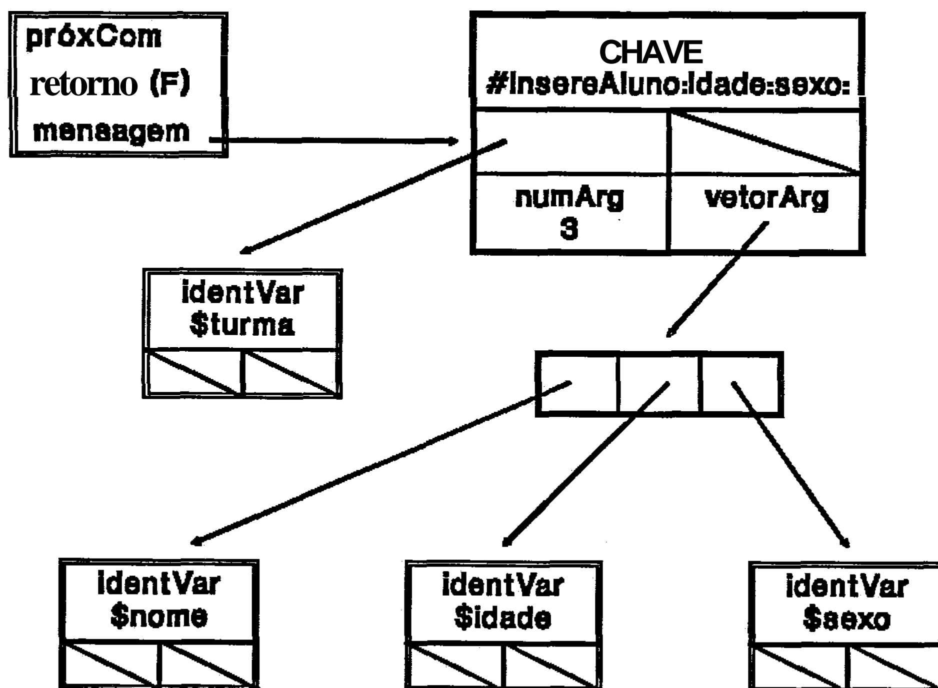
. numArg - número de argumentos da mensagem chave

. <u>vetorArg</u> - apontador para o "array" que contém os apontadores para as MSGs que definem os argumentos da mensagem chave

A figura 5.9 mostra o uso de MSG_CHAVE com a árvore sintática gerada para a mensagem de seletor *#insereAluno:idade:sexo:* e argumentos *$nome, $idade* e *$sexo*.

```
$turma insereAluno: $nome idade: $idade sexo: $sexo;
```

(a) Mensagem MANO com parâmetros



(b) árvore sintática

Figura 5.9: Exemplo de árvore sintática com MSG_CHAVE

O terceiro tipo de **nó** que pode aparecer na árvore sintática é formado pelas instâncias da classe **MSG_BLOCO.** Uma MSG_BLOCO é gerada quando o Analisador Sintático identifica um bloco de mensagens. Como foi visto no capítulo **III,** um bloco de mensagens pode receber parâmetros que são passados no momento de sua ativação. A classe MSG–BLOCO também é uma subclasse da classe MSG e além de possuir as variáveis de instância da classe MSG, acrescenta as seguintes variáveis de instância:

. **listArgBloco** - lista com o nome dos argumentos do bloco

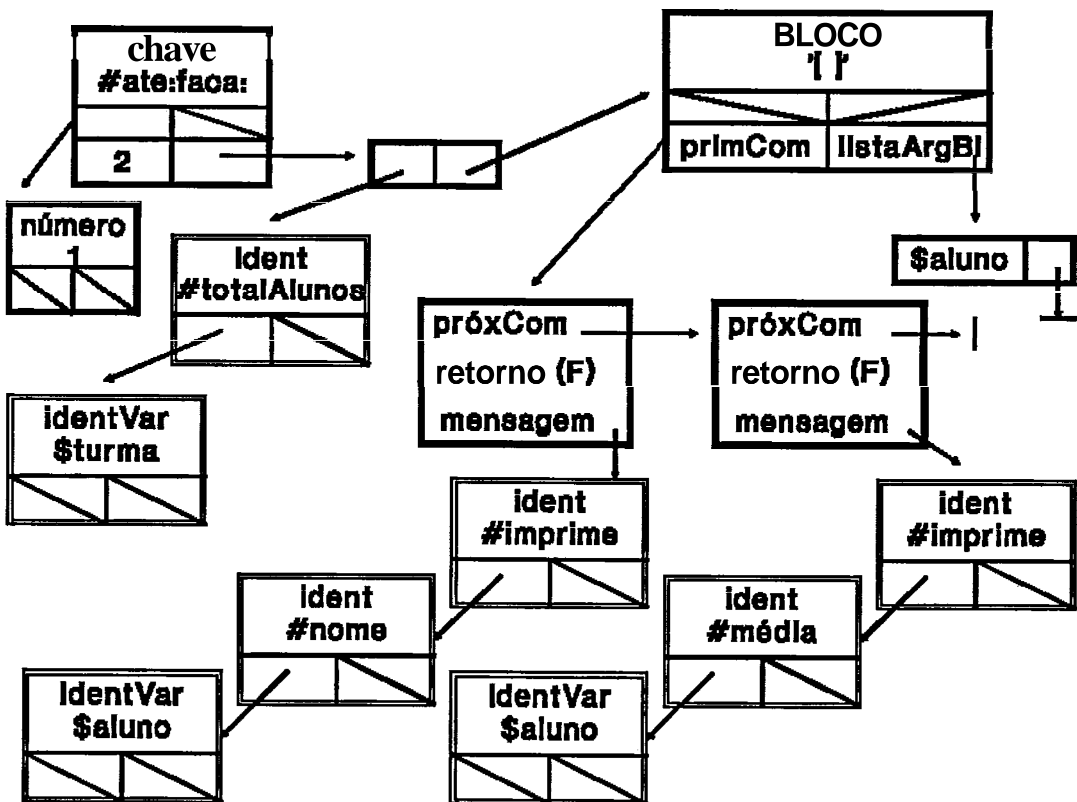
. **listComandos** - lista de comandos do bloco

A lista de comandos usada em uma MSG–BLOCO pertence à mesma classe da lista de comandos usada para armazenar os comandos do método. A figura 5.10 exemplifica o uso de MSG–BLOCO.

```
1 até: $turma.totalAlunos faça:

      % $aluno;
      imprime _o nome  do $aluno;
      imprime _a média-_do $aluno;
};
```

(a) Mensagem MANO



(b) árvore sintática

Figura 5.10: Exemplo de árvore sintática com MSG_BLOCO

O **Analisador** Sintático utiliza estes três tipos de nós **(MSG, MSG_CHAVE** e **MSG_BLOCO)** para construir a árvore sintática de qualquer método escrito em MANO. Após a geração da árvore sintática, um segundo passo (descrito a seguir) é realizado para verificar a semântica dos "comandos" de atribuição.

O Segundo Passo

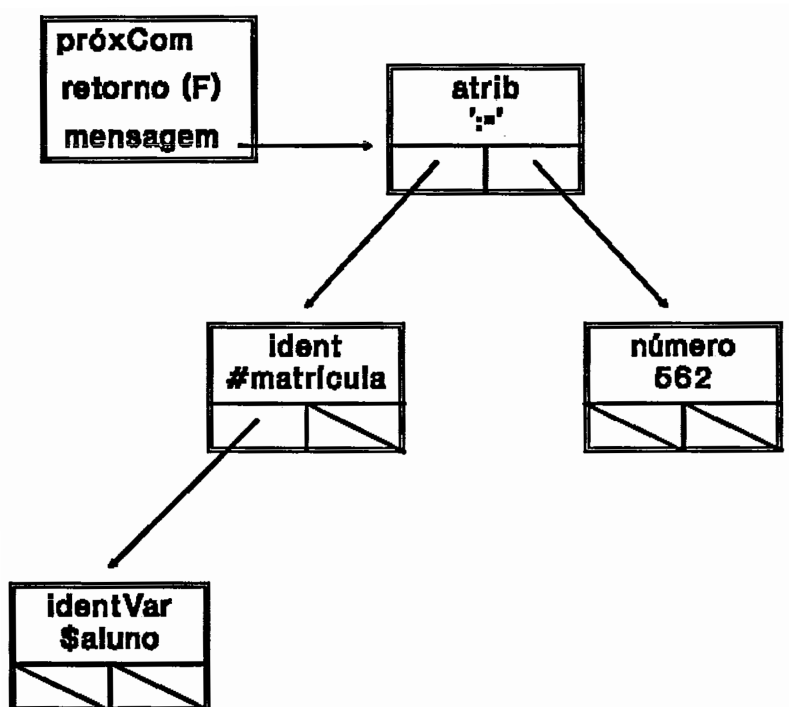
Neste segundo passo da análise sintática, a árvore gerada é percorrida para que possa ser verificado se os símbolos existentes do lado esquerdo dos comandos de atribuição **(:=)** estão semanticamente corretos.

Na maioria das linguagens, o símbolo que aparece do lado esquerdo de um comando de atribuição é um identificador de variável. Porém, em **MANO** o símbolo de atribuição pode ser usado também para formar o seletor de uma mensagem. Existem duas situações em que isto pode ocorrer e nestes casos uma transformação da árvore sintática é realizada.

A primeira situação é quando o símbolo existente do lado esquerdo do comando de atribuição é um identificador de mensagem. Neste caso o símbolo de atribuição **(:=)** é concatenado ao seletor dessa mensagem para formar um novo seletor de mensagem e o nó **(MSG)** de atribuição é eliminado da árvore sintática. A figura 5.11 exemplifica esta situação, mostrando uma mensagem MANO (a), a árvore sintática gerada para esta mensagem (b) e a árvore resultante da transformação do "comando" de envio de mensagem.

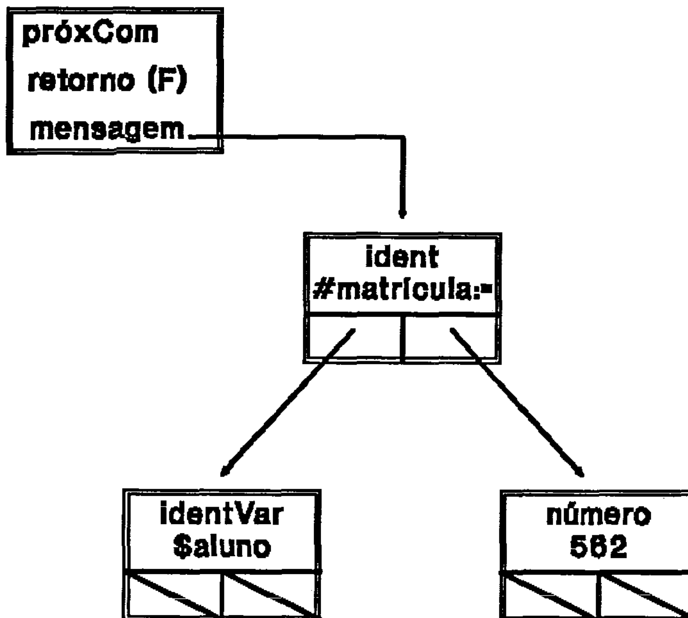
```
$aluno.matrícula := 562;
```

(a) Mensagem MANO



(b) árvore sintática

Figura 5.11: Transformação de atribuição em mensagem



(c) árvore resultante

Figura 5.11: Transformação de atribuição em mensagem

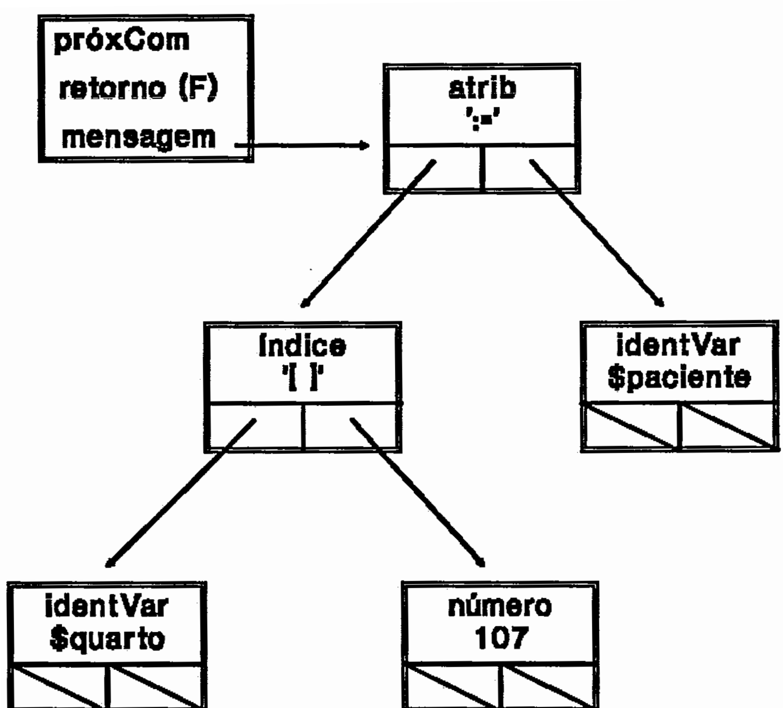
Na figura 5.11, pode-se observar que o nó contendo o símbolo de atribuição é eliminado da árvore sintática após a transformação. Isto significa que o código que será gerado para esta mensagem, será uma instrução de envio de mensagem e não uma instrução de armazenamento, como será visto na próxima seção.

A segunda situação é quando o símbolo existente do lado esquerdo do comando de atribuição é um indice. Neste caso também a árvore sintática é modificada. Uma mensagem cujo seletor é formado por um índice (#[]), tem como função identificar um determinado elemento de um objeto indexado. Quando um identificador de mensagem, do **tipo** índice, é seguido pelo símbolo de atribuição, então este símbolo é concatenado ao de índice para formar um novo identificador de mensagem (#[]:=). Como este novo

identificador de mensagem possui dois argumentos, o novo nó sintático passa a ser do tipo MSG_CHAVE. A figura 5.12 exemplifica esta situação.

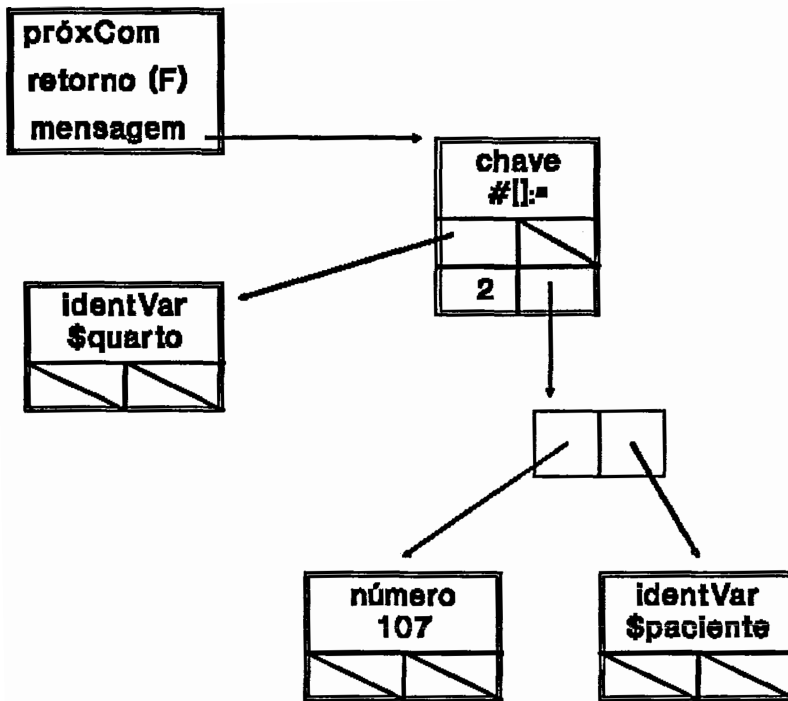
```
$quarto [ 107 ] := $paciente;
```

(a) mensagem MANO



(b) árvore sintática

Figura 5.12: Transformação de MSG em MSG_CHAVE



(c) árvore resultante

Figura 5.12: Transformação de MSG em MSG_CHAVE

V.5 Gerador de Código

O Gerador de Código tem como função gerar as instruções correspondentes ao método que está sendo compiiado. O Gerador de Código gera as instruções a partir da leitura e análise dos nós da árvore sintática e grava essas instruções em uma lista encadeada.

A classe ***Compilador*** possui uma variável de instância que contém um apontador para o primeiro elemento da lista de instruções.

O Gerador de Código foi implementado como uma classe C++, denominada GerCod. Para cada tipo de símbolo identificado em uma MSG da árvore sintática, existe um procedimento específico que é responsável pela geração das instruções. ***O*** método ***gereMsg*** da classe ***GerCod*** identifica o tipo de símbolo armazenado na MSG e ativa o método correspondente.

A seguir estão descritos os procedimentos que são realizados para cada tipo de símbolo identificado.

Identificador de Variável

Sempre que um nó sintático (MSG) possui um identificador de variável, uma instrução de pilha é gerada. O método ***gereIdentVar*** (figura 5.13) verifica a natureza da variável para definir qual a instrução de pilha que será gerada.

Se o identificador de variável corresponde a uma das variáveis especiais ($mim, $vero, $falso e $nulo), então é gerada sua respectiva instrução (112, 113, 114 e 115), conforme especificado na Tabela de Instruções da figura 4.8. Quando o identificador corresponde à variável especial "$super", também é gerada a instrução 112, que empilha o receptor da mensagem.

Se o identificador de variável corresponde a uma variável temporária, a instrução gerada pertence ao intervalo de 16 a 31, dependendo da posição da variável dentro da lista de variáveis temporárias. A lista de variáveis temporárias é gerada pelo Analisador Sintático no inicio da análise do método.

Se o identificador de variável corresponde a uma variável de instância, a instrução gerada pertence ao intervalo de 0 a 15, dependendo da posição da variável dentro da lista de variáveis de instância. A lista de variáveis de instância é gerada pelo Compilador, a partir da verificação da classe sob a qual o método está sendo compilado e

de todas as suas superclasses, para que possam ser identificadas as variáveis de instância herdadas pela classe.

```
gereIdentVar

   se MSG->símbolo = "$mim" ou "$super" ou "$vero" ou
               "$falso" ou "$nulo
   então
      gere 112,112,113,114 ou 115
   senão
      se listaVarTemp contém MSG->símbolo
      então
         gere 15 + posição na listaVarTemp
      senão
         se listaVarInst contém MSG->símbolo
         então
            gere 0 + posição na listaVarInst
         senão
            se listaLiterais contém MSG->símbolo
            então
               gere 64 + posição na listaLiterais
            senão
               envie "Variável não declaradaW
               cancele compilação
```

Figura 5.13: Gere **Identificador** de **Variável**

O próximo passo é verificar se o identificador de variável corresponde a uma variável global. Quando o Analisador Sintático identifica uma variável global, o nome dessa variável é incluído na lista de literais do método. Desta forma, se o identificador corresponde a uma variável global, a instrução gerada pertence ao intervalo de 64 a 95, dependendo da posição do nome da variável dentro da lista de literais do método. A lista de literais é gerada pelo Analisador Sintático e contém cadeias de caracteres que não são referenciadas diretamente pelo conjunto de instruções.

Se o identificador de variável não tiver sido identificado em nenhuma dessas situações anteriores, uma mensagem de variável não declarada é enviada ao usuário e o processo de compilação é interrompido. O sistema **Smalltalk** [DIGI88] permite neste momento, que o usuário escolha por criar uma nova variável global ou cancelar a

compilação. Numa próxima versão da linguagem MANO, este procedimento poderá ser facilmente alterado.

Número

Se um nó sintático (MSG) contém um símbolo do tipo número, é gerada uma instrução que acrescenta este número na pilha de execução. Os objetos cujos valores correspondem aos números -1, 0, 1 e 2 recebem tratamento especial por serem muito frequentes. Um conjunto de instruções é reservado para acrescentar estes objetos na pilha. Os outros números são tratados como literais do método.

O método ***gereNúmero*** (figura 5.14) testa se o símbolo existente na MSG, corresponde a um desses objetos especiais e neste caso gera sua respectiva instrução. Senão, o método ***gereLiteralConstante*** é ativado.

```
gereNúmero

    se MSG->símbolo = -1, 0, 1 ou 2
    então
       gere 116, 117, 118 ou 119
    senão
       gereLiteralConstante
```

Figura 5.14: Gere Número

Literal Constante

Se o tipo do símbolo armazenado no nó sintático é um caractere **(\c),** um nome **(#xxx),** uma cadeia de caracteres ('xxx' ou "xxx"), ou um número diferente de (-1, 0, 1 e 2), então uma instrução pertencente ao intervalo de 32 a 63 é gerada.

O método **gereLiteralConstante** (figura 5.15) apenas verifica a posição do símbolo na lista de literais para definir qual a instrução que será gerada.

```
gereLiteralConstante

    gere 31 + posição na lista de literais
```

Figura 5.15: Gere Literal Constante

Sequência de Identificadores

Quando o **Analisador** Sintático forma uma mensagem unária, isto é, uma mensagem sem argumentos, uma MSG **com** o **tipo** de **símbolo** ***sequência de identificadores*** é incluída na árvore sintática. Como uma mensagem unária não possui argumentos, apenas a sub-árvore correspondente ao receptor da mensagem tem que ser analisada pelo Gerador de Código.

Na primeira etapa, o método ***gereMsgUnária*** (figura 5.16) ativa, de forma recursiva, o método ***gereMsg*** passando como parâmetro o nó MSG->receptor. Após a conclusão da geração de código do receptor, o método gera o código correspondente à mensagem **unária.**

```
gereMsgUnária

    gereMsg( MSG->receptor )

    se listaLiterais contém MSG->símbolo
    então
       gere 207 + posição na lista de literais
    senão
       gereMsgSeletorEspecial
```

Figura 5.16: Gere Mensagem **Unária**

O Compilador utiliza a Lista de Seletores Especiais (figura 5.17), que contém os seletores de mensagens que são usados mais freqüentemente. Os métodos correspondentes a estes seletores serão implementados junto com o projeto do metaesquema do GEOTABA.

```
em:          fim        tam       classe
em:mud:      ==         faça:     copieBl:
prox         avalie     novo      x
proxMud:     avalie:    novo:     y
```

Figura 5.17: Lista de Seletores Especiais

A Tabela de Instruções contém uma instrução para cada seletor especial, os demais seletores de mensagens são tratados como literais do método. O método ***gereMsgUnária*** (figura 5.16), após gerar as instruções correspondentes ao nó do receptor, verifica se o seletor da mensagem é um seletor especial ou um literal. Se for um seletor especial, o método ***gereMsgSeletorEspecial*** é ativado, senão é gerada uma instrução pertencente ao intervalo de 208 a 223, dependendo da posição do seletor na lista de literais.

O método ***gereMsgSeletorEspecial*** (figura 5.18) apenas verifica a posição do seletor na Lista de Seletores Especiais e gera uma instrução pertencente ao intervalo de 192 a 207.

```
gereMsgSeletorEspecial

    gere 191 + posição Lista Seletores Especiais
```

Figura 5.18: Gere Mensagem Seletor Especial

Sempre que uma mensagem de superclasse é enviada, ou seja, o receptor da mensagem é o objeto especial ***$super***, então o Gerador de Código utiliza a instrução

estendida 133, cujo byte extensão contém o número de argumentos da mensagem e a posição do seletor da mensagem na lista de literais. As instruções estendidas, como foi visto no capítulo IV, possuem um ou dois bytes adicionais, possibilitando limites mais flexíveis para a linguagem.

Operadores

Todo **nó** sintático (MSG) cujo tipo de símbolo é um operador (Adicional, Multiplicativo, Relacional ou Lógico) equivale a uma mensagem binária, ou seja, possui uma sub-árvore correspondente ao receptor e uma sub-árvore correspondente ao argumento da mensagem.

O método ***gereOperadores*** (figura 5.19) chama inicialmente, o método ***gereMsg*** passando como parâmetro o nó MSG->receptor e em seguida, chama novamente o mesmo método ***gereMsg*** passando como parâmetro o nó sintático ***MSG->*** argumento. Este procedimento faz com que sejam geradas as instruções que colocam na pilha de execução o objeto receptor e o argumento da mensagem. Em seguida é gerada a instrução de envio de mensagem.

```
gereOperadores

  gereMsg( MSG->receptor )
  gereMsg( MSG->argumento )

  se MSG->símbolo é operador especial
  então
       gere 175 + posição Tabela de Operadores Especiais
  senão
       gereMsgBinária
```

Figura 5.19: Gere Operadores

De forma semelhante aos seletores especiais, um conjunto de operadores especiais é utilizado pelo Compilador (figura 5.20). Para cada operador especial existe uma

instrução especifica. Além desses operadores, outros operadores podem ser criados pelo usuário, mas neste caso os seletores são tratados como literais.

O método ***gereOperadores*** verifica se o seletor da MSG é um operador especial. Se for, é gerada uma instrução pertencente ao intervalo de **176** a **191.** Senão, o método ***gereMsgBinária*** é ativado.

```
+     soma
-     subtração
*     multiplicação
/     divisão
<     menor
>     maior
<=    menor ou igual
>=    maior ou igual
=     igual
~=    diferente
//    divisão inteira
\\    resto da divisão inteira
&     ê lógico
|     ou lógico
<<    desvio de bits positivo ou negativo
@     identificação de ponto
```

Figura 5.20: Lista de Operadores Especiais

O método ***gereMsgBinária*** (figura **5.21)** testa se o seletor ***MSG->símbolo*** é um literal. Se for, uma instrução pertencente ao intervalo de **224** a **239** é gerada, senão o método ***gereMsgSeletorEspecial*** já descrito anteriormente (figura **5.18)** é ativado.

```
gereMsgBinária

   se listaLiterais contém MSG->símbolo
   então
      gere 223 + posição na ListaLiterais
   senão
      gereMsgSeletorEspecial
```

Figura 5.21: Gere Mensagem Binária

Mensagem Com Parâmetros (Chave)

Como foi visto na seção **V.4,** todo **nó** sintático cujo tipo de símbolo é uma chave, é armazenado em um nó da classe ***MSG_CHAVE.*** *A* Tabela de Instruções MANO contém vários grupos de instruções de envio de mensagens. Desta forma, o método ***gereMsgChave*** (figura 5.22) precisa identificar, a partir do número de argumentos da mensagem, qual instrução de envio que será gerada.

Para qualquer quantidade de argumentos que uma mensagem possua, a ordem de geração das instruções é sempre a mesma, ou seja, primeiro é gerada a instrução que **empilha** o objeto receptor da mensagem. Em seguida são geradas instruções que tem como objetivo **empilhar** os argumentos da mensagem, começando pelo primeiro argumento e indo até o último. No final, é gerada uma instrução de envio de mensagem específica para o número de argumentos que a mensagem possui.

```
gereMsgChave

    se MSG_CHAVE->numArg = 1
    então
       gereMsg( MSG_CHAVE->receptor )
       gereMsg( MSG_CHAVE->vetorArg[0] )
       gereMsgBinária
    senão
       se MSG_CHAVE->numArg = 2
       então
          gereMsg( MSG_CHAVE->receptor )
          gereMsg( MSG_CHAVE->vetorArg[0] )
          gereMsg( MSG_CHAVE->vetorArg[1] )
          gereMsgTernária
       senão
          gereMsgComNArg
```

Figura 5.22: Gere Mensagem Chave

O método ***gereMsgChave*** dá um tratamento especial às mensagens contendo um ou dois argumentos, que cobrem a maioria das mensagens. Para essas mensagens são usados os métodos ***gereMsgBinária*** (já descrito anteriormente) e o método

gereMsgTernária (figura **5.23),** que gera instruções pertencentes ao intervalo de 240 a 255 se o seletor da mensagem for um literal, senão gera uma instrução de seletor especial.

```
gereMsgTernária

    se ListaLiterais contém MSG->símbolo
    então
        gere 239 + posição na lista de literais
    senão
        gereMsgSeletorEspecial
```

Figura 5.23: Gere Mensagem Ternária

Se a mensagem possui mais de dois argumentos, então o método ***gereMsgComNArg*** é usado. ***O*** método ***gereMsgComNArg*** (figura 5.24) após gerar as instruções que **empilham** o objeto receptor e os argumentos da mensagem, gera uma das instruções estendidas (131 ou **132),** dependendo do número de argumentos do método.

```
gereMsgComNArg

  gereMsg( MSG_CHAVE->receptor )

  para i variando de 0 a numArg - 1 faça
      gereMsg( MSG_CHAVE->vetorArg[ i ]

  se numArg <= 8
  então
      gere 131
      gere (numArg << 5) + posição na lista de literais
  senão
      gere 132
      gere numArg
      gere posição na lista de literais
```

Figura 5.24: Gere Mensagem com N Argumentos

Se a mensagem possui até oito argumentos, é gerada a instrução 131 e em seguida é gerado o ***byte*** extensão onde estão embutidos o número de argumentos e a posição do seletor na lista de literais. Se a mensagem possui mais de oito argumentos, então é gerada a instrução estendida 132 e em seguida um ***byte*** contendo o número de argumentos e um ***byte*** contendo a posição do seletor na lista de literais.

Atribuição

O segundo passo da fase de análise sintática garante que o tipo de símbolo existente do lado esquerdo de uma atribuição seja um identificador de variável.

Para gerar as instruções a partir de um **nó** sintático do tipo atribuição, é necessário gerar inicialmente as instruções da sub-árvore que corresponde ao argumento da mensagem de atribuição. Em seguida é gerada a instrução que retira da pilha de execução um objeto e o armazena na variável correspondente ao receptor da mensagem de atribuição.

```
gereAtribuição

gereMsg( MSG->argumento )

se listaVarInst contém MSG->receptor->símbolo
então
  gere 95 + posição na listaVarInst
senão
  se listaVarTemp contém MSG->receptor->símbolo
  então
       gere 103 + posição na listaVarTemp
  senão
       se listaLiterais contém MSG->receptor->símbolo
       então
            gere 192 + posição na listaLiterais
       senão
            envie "variável não declarada"
            cancele compilação
```

Figura 5.25: Gere Atribuição

O método ***gereAtribuição*** (figura 5.25) identifica a natureza da variável **receptora** e gera sua instrução correspondente. Se o receptor é uma variável de instância, é gerada uma instrução pertencente ao intervalo de 96 a 103. Se for uma variável temporária, a instrução gerada pertence ao intervalo de 104 a 111. Se for uma variável global, a instrução estendida 130 é gerada e o ***byte*** de extensão é gerado imediatamente após, com seu valor dependendo da posição do nome da variável global na lista de literais. Se o identificador da variável não for encontrado em nenhuma dessas listas, uma mensagem de "variável não declarada" é enviada ao usuário e a compilação do método é interrompida.

Bloco de Mensagens

Conforme descrito no capítulo IV, o Processador utiliza uma estrutura de dados chamada ***contexto de bloco*** para executar as mensagens pertencentes a um **bloco.**

O método ***gereBloco*** (figura 5.26) gera inicialmente, um conjunto de instruções que farão com que o Processador crie o contexto de bloco. A criação do contexto de bloco é feita em tempo de execução pela rotina primitiva associada à mensagem ***#copiebloco. O*** método ***gereBloco*** primeiro gera a instrução 137, que irá **empilhar** o contexto corrente, que é o objeto receptor da mensagem ***#copieBloco.*** Depois é gerada uma instrução que irá **empilhar** o número de argumentos que o bloco possui, esta informação será usada para criar o contexto de bloco. Em seguida é gerada a instrução 200, que corresponde ao envio da mensagem ***#copieBloco.***

No próximo passo é gerada a instrução estendida 164, que é um desvio incondicional que fará com que o Processador "pule" as instruções correspondentes ao **corpo** do bloco de mensagens. Estas instruções só serão executadas quando o contexto de bloco, que também é um objeto, receber a mensagem de seletor ***#avalie*** e tornar-se ***o*** contexto ativo.

Antes de gerar as instruções referentes ao corpo do método, ainda são geradas as instruções que fazem a atribuição dos argumentos do bloco com os valores passados

como **parâmetros** para o bloco. Por último, a lista de comandos do bloco é percorrida e o método ***gereMsg*** é ativado para cada sub-árvore associada aos comandos da lista.

```
gereBloco

    gere 137
    gereNúmero( numArgBloco )
    gere 200

    gere 164,nInstr

    para cada arg em MSG BLOCO->listaArgBl faça
         gere 103 + posição na listaVarTemp

    para cada comando em MSG_BLOCO->listaComBl faça
         gereMsg( comando->MSG )
```

Figura 5.26: Gere Bloco de Mensagens

Mensagens Especiais

A linguagem possui três mensagens para as **quais** o Gerador de Código gera o código de forma otimizada, são elas: ***#seVero:***, ***#seVero:senão:*** e #enquanto:. Para estas três mensagens, o Compilador não gera instruções de envio e sim uma sequência de instruções que envolvem desvios condicionais e incondicionais, dependendo da mensagem. A vantagem desta otimização é que, para a execução dos blocos que são os argumentos dessas mensagens, não são criados contextos de bloco e também não é realizada a busca do seletor da mensagem na hierarquia de classes.

A geração de código para essas mensagens é feita a partir do método ***gereChave*** (descrito anteriormente), uma vez que o tipo de símbolo do nó sintático em que estas mensagens estão armazenadas é o tipo chave.

Na figura 5.22 foram omitidas, por motivo de clareza, as chamadas para os métodos responsáveis pela geração do código dessas mensagens. A alteração desse

método é simples, bastando incluir no seu início, um teste para verificar se o seletor da MSG é um desses três seletores e ativar o respectivo método para fazer a geração especial.

A figura 5.27 mostra o método ***gereMsgSeVero***, que chama inicialmente o método ***gereMsg*** para gerar ***o*** código da sub-árvore correspondente ao receptor da mensagem. Em seguida, é gerada a instrução estendida 172 que faz com que o apontador de instruções seja alterado para saltar as instruções referentes ao bloco de mensagens, caso o valor no topo da pilha seja falso. Por último é ativado o método ***gereComandoComposto***, tendo como parâmetro o argumento da mensagem ***#seVero:***.

```
gereMsgSevero

    gereMsg( MSG_CHAVE->receptor )
    gere 172,nInstr
    gereComandoComposto( MSG_CHAVE->vetorArg[0] )
```

Figura 5.27: Gere Mensagem #severo:

O método ***gereComandoComposto*** (figura 5.28) percorre a lista de comandos de um bloco, chamando para cada comando o método ***gereMsg***.

```
gereComandoComposto

    para cada comando em MSG_BLOCO->listaComBl faça
       gereMsg( comando->MSG )
```

Figura 5.28: Gere Comando Composto

O método ***gereMsgSeVeroSenão*** (figura 5.29) chama inicialmente ***o*** método ***gereMsg*** para gerar os códigos da sub-árvore do receptor da mensagem. Em seguida é

gerada a instrução estendida 172, que faz com que o primeiro bloco não seja executado, se o objeto no topo da pilha for falso.

Em seguida é ativado o método ***gereComandoComposto***, tendo como **parâmetro** o primeiro argumento da mensagem ***#seVero:senão:***. Após as instruções do primeiro bloco de mensagens, é gerada a instrução estendida 164 para que o **processamento** desvie incondicionalmente para a primeira instrução após as **instruções** do segundo bloco. Por último o método ***gereComandoComposto*** é novamente ativado para gerar as instruções referentes ao segundo argumento da mensagem ***#seVero:senão***.

<u>gereMsgSeVeroSenão</u>

```
gereMsg( MSG_CHAVE->receptor )
gere 172,nInstrBloco1
gereComandoComposto( MSG_CHAVE->vetorArg[0] )
gere 164,nInstrBloco2
gereComandoComposto( MSG_CHAVE->vetorArg[1] )
```

Figura 5.29: Gere Mensagem #seVero:Senão:

O método ***gereMsgEnquanto*** (figura 5.30) possui um procedimento semelhante ao método ***gereMsgSe*** (figura **5.27**), apenas acrescentando no final a instrução estendida 164, que é um desvio incondicional que faz com que a execução do método volte para a instrução de desvio condicional 172. Com isto, as instruções referentes ao bloco de mensagens são executadas repetitivamente enquanto o objeto no topo da pilha de execução for o objeto verdadeiro.

<u>gereMsgEnquanto</u>

```
gereMsg( MSG_CHAVE->receptor )
gere 172,nInstr
gereComandoComposto( MSG_CHAVE->vetorArg[0] )
gere 164,nInstr
```

Figura 5.30: Gere Mensagem #enquanto:

Finalização do Método

Todo método é finalizado por uma instrução de retorno. Quando a última mensagem do método está precedida do símbolo de retorno ('^' sinal de circunflexo), normalmente é gerada a instrução 124 que faz com que o objeto localizado no topo da pilha seja devolvido como resultado da mensagem. Porém, se o objeto que estiver no topo da pilha for um objeto especial, isto é, verdadeiro **($vero),** falso ($falso) ou nulo ($nulo), então uma das instruções de retorno especial (121, 122 ou 123) será usada.

Se a última mensagem do método não for uma mensagem de retorno, é gerada automaticamente a instrução 120, que faz com que o próprio objeto receptor da mensagem seja retomado.

Os blocos de mensagens também possuem como última instrução a instrução de retorno 125. Se uma mensagem de retorno for usada dentro de um bloco de mensagens, então, **não será** encerrado apenas a execução **do** bloco ***e*** sim a **execução do** método que contém o bloco de mensagens.

Capítulo VI

Gerente de Armazenamento

VI.l Introdução

O Gerente de Armazenamento (GA) e o Gerente de Execução (GE) formam juntos o núcleo do ProtoGEO, que é o responsável pelo processamento de mensagens MANO. *O* GE tem a função de identificar e executar o método que irá "responder" à mensagem recebida, enquanto que o GA tem a função de deixar disponíveis na memória, os objetos que serão utilizados pelo GE, Além de transferir os objetos que estão armazenados no banco de dados, dos meios secundários para a memória principal, e vice-versa, o GA também é o responsável pela criação dos novos objetos.

Todo objeto quando é criado, além de ter um espaço de memória reservado para o armazenamento de seus campos, recebe um identificador que é chamado de identidade do objeto **(idO).** A identidade do objeto é gerada automaticamente no momento da criação do objeto, ficando associado a ele durante toda a sua existência. **A idO** identifica o objeto **univocamente** em todo o banco de dados.

No sistema ProtoGEO, os objetos não são eliminados explicitamente pelo programador. Em vez disso, um mecanismo de "coleta de lixo" é fornecido pelo sistema. **Todo** objeto inacessível, isto é, que não é referenciado por nenhum outro objeto do banco de dados, tem o seu espaço de memória liberado e sua **idO** fica disponível para ser

usada por futuros objetos que serão criados. Este reaproveitamento de espaço de memória é feito pelo coletor de lixo.

A figura 6.1 mostra a estrutura do Gerente de Armazenamento, já apresentada na seção III.4 (Arquitetura do ProtoGEO). O GA é composto de três módulos: o Gerente do "Buffer" de Objetos, o Gerente do "Buffer" de Páginas e o Gerente de Disco. Neste trabalho, uma primeira versão do Gerente do "Buffer" de Objetos foi implementada para possibilitar o desenvolvimento do Processador e do Compilador da linguagem MANO. Os Gerentes do "Buffer" de Páginas e de Disco serão implementados em uma segunda fase do projeto.

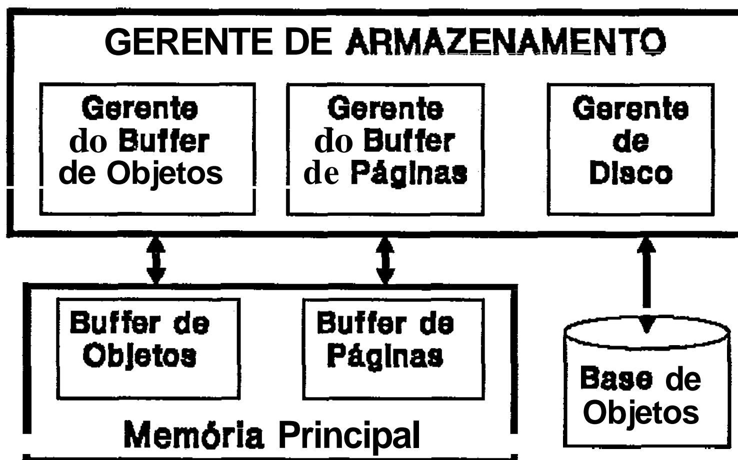


Figura 6.1: Gerente de Armazenamento

Como todos os objetos do banco de dados não podem estar presentes na memória simultaneamente, o sistema ProtoGEO utiliza um mecanismo de memória virtual. Como foi dito anteriormente, o mecanismo de memória virtual do ProtoGEO utiliza o esquema

de "buffer" duplo, que é baseado no sistema LOOM **[KAEL83]** e é utilizado também em diversos sistemas orientados a objetos, como no **ORION [KIM90a], O_2 [VELE89], GemStone [MAIE86a]** e **Vbase [ANDR87].** No esquema de "buffer" duplo, a área de "buffer" da memória principal é dividida para conter o "buffer" de páginas, que armazena as páginas do disco e o "buffer" de objetos, que armazena os objetos residentes na memória.

Neste esquema, quando um objeto é solicitado, primeiro o GA tenta recuperá-lo no "buffer" de objetos. Caso não seja encontrado, ele verifica se a página que contém o objeto está presente no "buffer" de páginas. Em caso afirmativo, o objeto é transferido **para** o "buffer" de objetos, senão, a página deve ser recuperada em disco *e* carregada no "buffer" de páginas. Após a recuperação da página, o objeto é transferido para o "buffer" de objetos.

A vantagem de se utilizar o esquema de "buffer" duplo, é que, segundo Maier **[MAIE89],** as aplicações não convencionais (como é o caso do Projeto TABA), tendem a carregar uma grande quantidade de objetos na memória para então executar as operações computacionais sobre esses objetos. Desta forma, em um esquema que utilize somente o "buffer" de páginas, muitos objetos ficam carregados na memória desnecessariamente, aumentando assim o número de acessos ao disco.

No sistema **ProtoGEO,** os objetos são gerenciados em dois formatos distintos: o formato de disco *e* o de memória. Segundo W. **Kim [KIM90a],** objetos no formato de disco possibilitam maior eficiência de armazenamento e recuperação, enquanto que os objetos no formato de memória permitem que as aplicações manipulem os objetos nos formatos suportados pelas linguagens de programação.

Os objetos armazenados no "buffer" de páginas estão no formato de disco e os objetos armazenados no "buffer" de objetos estão no formato de memória. A conversão de um formato para outro é feita quando o objeto é transferido do "buffer" de páginas para o "buffer" de objetos, quando este está sendo recuperado, ou quando o objeto é transferido do "buffer" de objetos para uma página do "buffer" de páginas, para que este possa ser salvo no banco de dados.

VI.2 Gerente do "Buffer" de Objetos

O Gerente do "Buffer" de Objetos (GBO) tem a função de gerenciar o espaço da memória principal, correspondente ao "buffer" de objetos. No "buffer" de objetos são armazenados os novos objetos criados pelo sistema e também os objetos transferidos do banco de dados.

O GBO implementado nesta primeira fase do projeto é uma versão preliminar, que teve como objetivo principal desenvolver um módulo de gerenciamento de objetos residentes em memória que fornecesse o suporte mínimo necessário para a implementação do Processador e do Compilador da linguagem MANO. Desta forma, diversos aspectos do GBO foram implementados de maneira bastante simplificada como será visto no decorrer deste capítulo.

"Buffer" de Objetos

O "buffer" de objetos corresponde ao pedaço da memória principal que é reservado para o armazenamento dos objetos residentes na memória. Quando um objeto é criado, um pedaço do "buffer" de objetos é reservado para armazenar continuamente os campos deste objeto. Quando o objeto é desalocado, o espaço ocupado por ele fica disponível para o armazenamento de outros objetos. O formato de um objeto **alocado** no "buffer" de objetos é mostrado na figura 6.2.

Os campos do objeto são precedidos por um cabeçalho contendo duas palavras (4 bytes). O campo ***tamanho*** indica o número de palavras do "buffer" que são ocupadas pelo objeto, incluindo o cabeçalho. O campo ***tamanho*** é um inteiro sem sinal e pode variar de 2 até 65536. O segundo campo do cabeçalho do objeto contém a identidade da classe do objeto.

A área de dados contém os campos do objeto. Existem três tipos de objetos, que são diferenciados de acordo com o tipo dos valores armazenados em seus campos, são eles: ***objeto identidade, objeto word*** e ***objeto byte.***

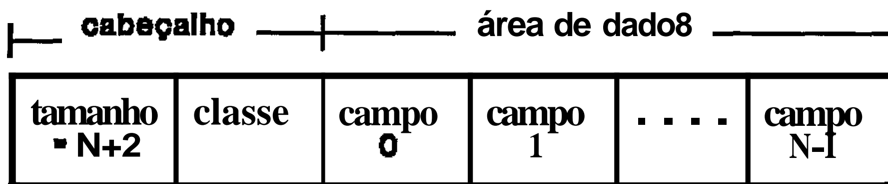


Figura 6.2: Formato de um Objeto

Um obieto identidade é aquele cujos valores armazenados em seus campos correspondem a identidades de objetos (descrito mais a frente). Este tipo de objeto é muito comum, uma vez que no sistema **ProtoGEO** mesmo os valores primitivos (inteiros, caracteres, etc) são modelados como objetos, ou seja, possuem identidade própria.

Um objeto word é aquele cujos campos armazenam valores de 16 ***bits*** sem sinal. ***Objetos word*** são usados, por exemplo, para armazenar instâncias da classe *InteirosLongos*.

Um objeto byte é aquele cujos campos armazenam valores de **8 *bits*.** São usados para armazenar instâncias de classes cujos campos correspondem a ***bytes*.** Por exemplo, instâncias da classe ***Cadeia*** contém em seus campos caracteres, que são armazenados em ***bytes*.**

As instâncias da classe ***Método*** são os únicos objetos do sistema que possuem um formato especial. Um método contém em sua área de dados um conjunto de identidades que armazenam o cabeçalho e os literais do método, seguido de um conjunto de ***bytes*** correspondente às instruções do método.

São funções do GBO, alocar e desalocar espaço no "buffer" de objetos para o armazenamento de objetos. Após algum tempo de uso do sistema, o "buffer" de objetos tende a ficar fragmentado, o que implica na necessidade de um processo de compactação dos objetos residentes no "buffer" para que o espaço disponível fique contínuo.

Nesta primeira fase do projeto, o "buffer" de objetos corresponde à área de "heap" reservada para alocação dinâmica de memória durante a execução dos programas. Os algoritmos de alocação e desalocação de memória, utilizam as funções disponíveis na linguagem C++ para alocar *(new)* e desalocar **(*delete*)** dinamicamente espaços nesta área de "heap".

Tabela de Objetos

Como nos sistemas O_2 [VELE89], ORION [KIM90a] e GemStone [MAIE86a], o sistema ProtoGEO utiliza um esquema de dupla indireção para acessar os objetos na memória. Um objeto pode fazer referência a vários objetos e, ao mesmo tempo, estar sendo referenciado por diversos outros objetos. Se os objetos fizessem referência diretamente ao endereço do objeto na memória, trocar um objeto de posição seria praticamente impossível, uma vez que todas as referências a este objeto teriam que ser localizadas e corrigidas para apontarem para o novo endereço do objeto.

Para solucionar este problema, o sistema utiliza a tabela de objetos e os campos de um objeto, ao invés de referenciar diretamente os objetos na memória, apontam para

a posição da tabela que contém o endereço do objeto na memória. Com isto, se um objeto muda de posição na memória, apenas a entrada na tabela referente ao objeto tem que ser alterada. A figura 6.3 representa este esquema.

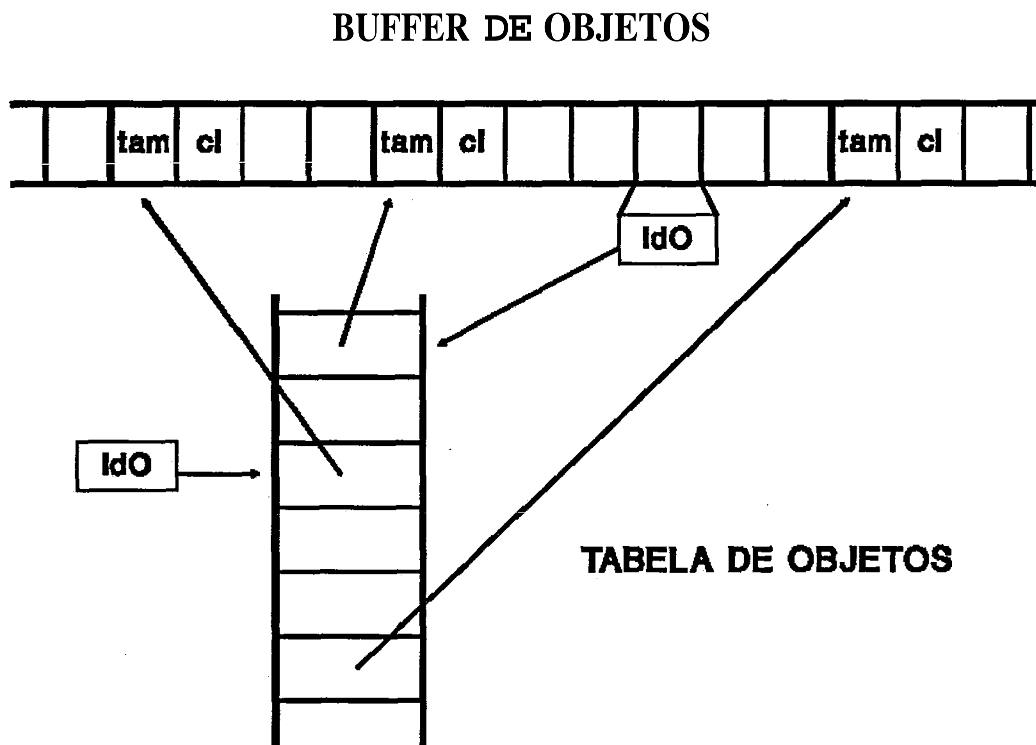


Figura 6.3: Esquema de Dupla Indireção

Identidade de Objetos (idO)

No sistema **ProtoGEO,** a identidade de um objeto residente em memória é composta de um valor de 16 bits (2 bytes). Da mesma forma que na linguagem **Smalltalk [GOLD83]** e no sistema **GemStone [BUTT91],** um tratamento especial é dado aos objetos da classe Inteiros, uma vez que estes objetos são usados com grande **frequência** e não possuem nenhum campo além de seu valor. A identidade de um objeto inteiro

representa totalmente o objeto, ou seja, nenhuma memória adicional ou entrada na tabela de objetos é alocada para armazenar um objeto inteiro.

Figura 6.4: Identidade de Objetos

Para diferenciar um objeto inteiro de uma idO, é utilizado o ***bit*** de menor ordem, como mostra a figura **6.4.** Se o ***bit*** está ligado, significa que os **15** primeiros ***bits*** representam um valor inteiro com sinal. Com isto, os objetos inteiros pertencentes ao intervalo $\pm 2^{14}$ (-16384 até **16383**) são representados diretamente por suas identidades. Valores inteiros fora destes limites são tratados por classes que estão sendo definidas no projeto do metaesquema do GEOTABA.

Se o ***bit*** de menor ordem está desligado, os **15** primeiros ***bits*** formam um índice na tabela de objetos. Com isto, é possível endereçar um total de 2^{15} (32K) objetos na memória. No formato de disco, a identidade do objeto poderá ser formada por exemplo, por valores constituídos de **4** ***bytes*** o que aumentaria o número de objetos endereçáveis

para 2^{31} (64K) objetos. Este tipo de definição será tomada quando da **especificação** do Gerente de Disco, estando fora do escopo deste trabalho.

Entrada de Objetos na Tabela

Cada entrada na tabela de objetos (figura 6.5) possui dois campos: (1) um apontador para a posição da memória no "buffer" de objetos onde estão armazenados os campos do objeto e (2) o contador que registra o número de vezes que o objeto está sendo referenciado por outros objetos.

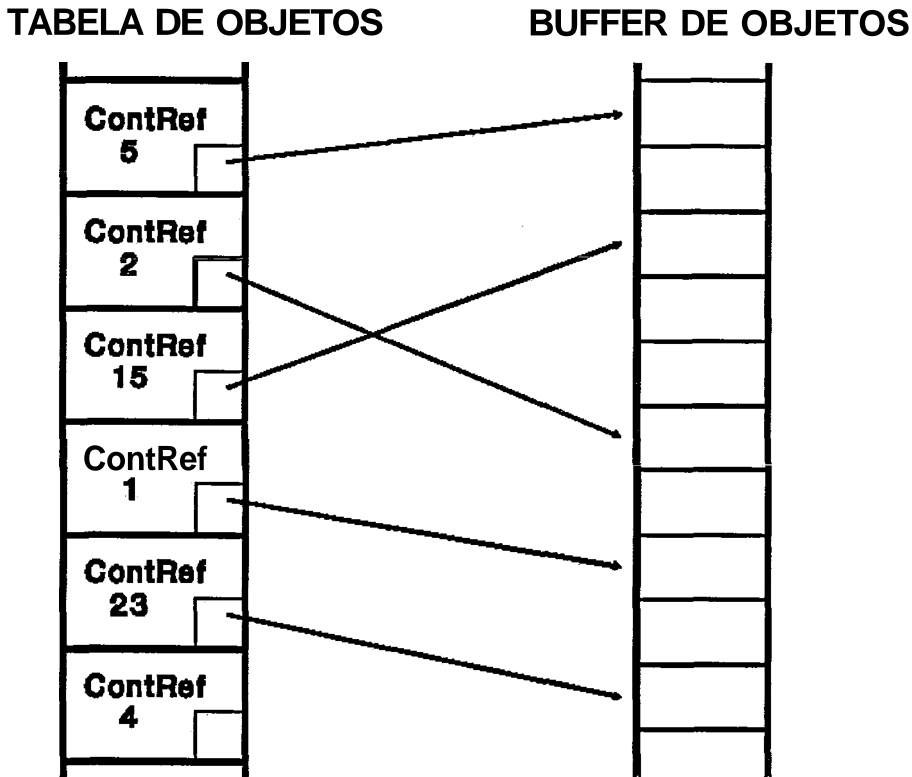


Figura 6.5: Entradas na Tabela de Objetos

O campo contador de referências é usado pelo sistema para fazer a eliminação automática de objetos inacessíveis. Quando este contador chega a zero, significa que

nenhum outro objeto faz referência a ele e o espaço de memória utilizado, bem como sua **idO** podem ser liberados.

Numa segunda etapa do projeto, quando forem implementados o Gerente do "Buffer" de Páginas e o Gerente de Disco, novos campos serão adicionados às entradas da tabela. O sistema **ORION [KIM90a]** por exemplo, mantém para cada entrada na tabela uma estrutura intermediária chamada ROD - Descritor de Objetos Residentes, que possui informações como: identificador da posição lógica do objeto, identificador da posição física do objeto, **idO** da classe do objeto, o "*cache*" de mensagens que é uma estrutura de dados usada para aumentar a velocidade do processo de despacho de mensagens, além de outros campos de controle.

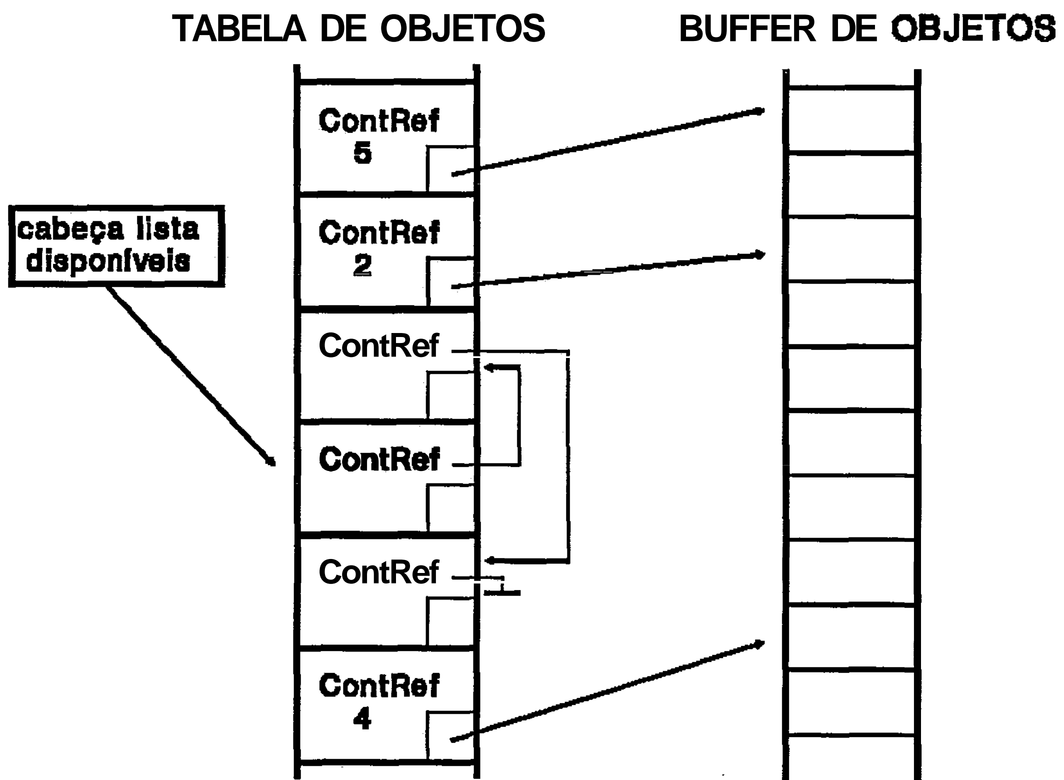


Figura 6.6: Lista de Entradas Disponíveis

Quando o contador de referências de um objeto atinge o valor zero, o sistema automaticamente libera o espaço do "buffer" de objetos que estava sendo ocupado pelo

objeto. A entrada na tabela correspondente à **idO** deste objeto fica disponível para ser **reutilizada.** Uma lista encadeada é usada para armazenar as entradas disponíveis na tabela de objetos. *O* campo ***contRef*** é usado para encadear as entradas disponíveis, como mostra a figura 6.6.

Interface do Gerente do "Buffer" de Objetos

Os objetos manipulados pelo Processador e Compilador MANO estão sempre residentes no "buffer" de objetos. Se um objeto requisitado não estiver presente no "buffer" de objetos, é de responsabilidade do Gerente de Armazenamento localizá-lo no "buffer" de páginas ou em disco e carregá-lo no "buffer" de objetos. Por questões de eficiência, tanto o Processador como o Compilador MANO interagem diretamente com o GBO, acessando e alterando campos de objetos, criando novas instâncias, etc. Para isto, uma interface formada por um conjunto de rotinas (mensagens C+ +) é fornecida pelo GBO.

O GBO foi implementado em C+ +, que é uma linguagem orientada a objetos. Desta forma, uma mesma mensagem pode ser enviada a objetos diferentes, provocando reações comportamentais próprias a cada objeto. Por exemplo, se a mensagem de seletor ***em(3)*** for enviada a um objeto identidade, a **idO** armazenada no quarto campo do objeto será retomada. Se a mesma mensagem for enviada a um objeto byte, o byte armazenado no quarto campo do objeto será retomado. No sistema **ProtoGEO** os campos dos objetos são indicados por índices inteiros relativos a zero, por isso a mensagem ***em(3)*** acessa o quarto campo do objeto.

As funções disponíveis para acessar e alterar os campos dos objetos são as seguintes:

<u>Acesso a Objetos Identidade</u>

receptor.em(i)

Retorna a **idO** armazenada no campo de índice **i** do objeto receptor

receptor.mud(i, obj)

Armazena a **idO** de obj no campo de índice **i** do objeto receptor

<u>Acesso a Objetos Word</u>

receptor.em(**i**)

Retoma o valor de 16 bits armazenado no campo de índice **i** do objeto receptor

receptor.**mud(** i, word)

Armazena o valor de 16 bits (word) no campo de índice i do objeto receptor

<u>Acesso a Objetos Byte</u>

receptor.em(i)

Retoma o valor de 8 bits armazenado no campo de índice **i** do objeto receptor

receptor.**mud(** i, byte)

Armazena o valor de 8 bits (byte) no campo de índice **i** do objeto receptor

Observe que os seletores das mensagens de acesso (em) e alteração (*mud*) são iguais, o que muda é o tipo do argumento e do valor retomado pelo método associado.

O método *mud(* ***i,*** obj **)** da classe ***ObjetoIdentidade*** decrementa o contador de referências do objeto que estava armazenado na posição i e incrementa o contador de referências do objeto que está sendo armazenado. ***O*** Processador MANO utiliza as seguintes rotinas para evitar que um objeto seja eliminado em determinadas situações.

Contador de Referências

receptor.incrRef()

Acrescenta um ao contador de referências do objeto receptor. O receptor tem que ser um objeto identidade

receptor.decrRef()

Decrementa um do contador de referências do objeto receptor. Se o contador de referências ficar com o valor zero o coletor de lixo é ativado.

Os valores armazenados no cabeçalho do objeto são acessados por rotinas específicas, são elas:

Acesso ao Cabeçalho de Objeto

receptor.cl()

Retoma a idO da classe do objeto receptor

receptor.tam()

Retoma o número de campos do objeto receptor

A criação de novas instâncias é solicitada ao GBO enviando-se uma das seguintes mensagens à classe do objeto desejado.

Criação de Instâncias

cl_receptora.obj(t)

Cria uma nova instância de ***objeto identidade*** com t campos que conterão idOs

cl_receptora.objWord(t)

Cria uma nova instância de ***objeto word*** com t campos que conterão valores de 16 ***bits***

cl_receptora.objByte(t)

Cria uma nova instância de ***objeto byte*** com t campos que conterão valores de 8 ***bits***

A interface do GBO fornece algumas funções que permitem ao Processador acessar todas as instâncias de uma determinada classe, em ordem de **idO.**

Enumeração de Instâncias

cl_receptora.primInst()

Retorna a **idO** da primeira instância da classe **receptora** da mensagem, ou seja, a instância de menor **idO**

receptor.prox()

Retorna a **idO** da próxima instância da mesma classe do objeto receptor.

Uma das funções disponíveis na interface GBO **é** a função ***vira()*** que permite fazer a troca de identidades entre dois objetos. Se dois objetos **A** e B trocam de identidades, todas as referências ao objeto **A** passam a referenciar o objeto B e **vice-**versa.

Troca de Identidades

receptor.vira(obj)

Faz com que a entrada na tabela de objetos referente ao objeto receptor aponte para a posição no "buffer" de objetos referenciada pelo objeto obj e vice-versa

Para **acessar** e alterar o valor de um objeto inteiro, pertencente ao intervalo -16384 a 16383, são usadas as seguintes rotinas:

<u>Acesso a Inteiros</u>

receptor.**val()**

Retoma o valor inteiro armazenado na **idO** do objeto receptor

int_**ObjInt(** val)

Retorna a **idO** do objeto inteiro cujo valor é **val**

Além dessas rotinas, a interface GBO fornece as rotinas abaixo que permitem verificar se um valor inteiro pode ser armazenado como objeto inteiro, isto é, se o valor pertence ao intervalo -16384 a 16383 e uma rotina que verifica se uma **idO** corresponde a um **objeto** inteiro **ou** não.

ehObjInt(val)

Retoma verdadeiro se o valor val pode ser representado como uma instância de Inteiros

receptor.**ehInt()**

Retorna verdadeiro se o receptor é uma instância de Inteiros, falso se não

Como foi dito anteriormente, esta primeira versão do GBO é uma versão **simplificada,** desenvolvida para possibilitar a implementação dos módulos do Gerente de Execução do **ProtoGEO.** Quando for desenvolvida a versão definitiva do GBO, algumas rotinas sofrerão alterações na parte de implementação, mas a interface do GBO não será alterada. Com isto, tanto o Processador como o Compilador MANO não sofrerão mudanças decorrentes destas alterações.

Capítulo VII

Conclusões

Este trabalho consistiu da definição e implementação da linguagem de programação orientada a objetos MANO e de seu processador. O objetivo principal foi o de desenvolver uma linguagem para a construção do ProtoGEO, que é o protótipo do SGBDOO GEOTABA.

A linguagem MANO foi definida para que, através dela, pudesse ser validado o modelo de objetos proposto para o GEOTABA. É a ferramenta de programação do nível de implementação do modelo de objetos, estando de acordo com a proposta de uma linguagem de programação de banco de dados (LPBD) como solução para o problema de incompatibilidade entre as linguagens de operação de banco de dados.

É usada na definição de métodos e no envio de mensagens. Além disso, está sendo usada na implementação do próprio ProtoGEO. Seu processador é o responsável pelo processamento de todas as mensagens enviadas aos objetos da base de dados, realizando a busca das mensagens na hierarquia de classes e executando os métodos associados.

A linguagem MANO foi definida para ser usada por um editor que explore os recursos gráficos através do uso de atributos de caracteres. Isto fará com que seus métodos sejam mais facilmente lidos e entendidos. Outro recurso de interface com usuário existente é o fato das mensagens poderem ser escritas utilizando-se preposições, obtendo-se assim uma forma mais próxima da língua portuguesa.

A implementação do Processador MANO baseou-se na especificação do Interpretador Smalltalk, sendo que as instruções MANO equivalem aos "bytecodes"

definidos em [GOLD83]. Devido a esta escolha, a linguagem MANO herdou algumas limitações existentes na linguagem Smalltalk, como por exemplo, o número limitado de variáveis em um método e o tamanho máximo de um objeto, que é de 64 Kbytes. Estas e outras limitações existentes na linguagem Smalltalk é que levaram ao desenvolvimento da linguagem OPAL, no sistema GemStone [COPE84]. Algumas dessas limitações poderão ser resolvidas quando for implementado o Gerente de Armazenamento do ProtoGEO, devido a elas estarem relacionadas com a representação do objeto nos meios de armazenamento. Outras limitações dizem respeito à Tabela de Instruções, cuja alteração é bem mais complexa.

Uma das características previstas no modelo de objetos, mas que ainda não é suportada pela linguagem MANO, é o conceito de múltipla herança. Após um estudo realizado nos protótipos de SGBDOOs, chegou-se a conclusão que este problema está mais ligado ao projeto do metaesquema do GEOTABA, e que as modificações que precisam ser realizadas no Processador para suportar múltipla herança, são de baixa complexidade.

Para que a linguagem pudesse ser usada, foi necessário implementar o Gerente do "Buffer" de Objetos, cuja função principal é permitir a criação e a manipulação de objetos. Estes objetos porém, são mantidos em memória e só serão persistentes quando for implementado o mecanismo de memória virtual do ProtoGEO.

O Compilador, o Processador e o Gerente do "Buffer" de Objetos foram todos implementados em C++ (Turbo C++), rodando em ambiente compatível com o IBM-PC e utilizando o sistema operacional DOS. Para que o projeto fosse facilmente transportável para o ambiente SUN, onde será desenvolvido o GEOTABA, foram usadas apenas funções definidas no C++ padrão [STRO86]. A grande vantagem de usar o ambiente Turbo C++, foi a utilização do seu depurador de programas, que é uma ferramenta de altíssima qualidade.

A próxima etapa para a construção do ProtoGEO deverá ser a implementação do seu metaesquema, ou seja, do conjunto de classes predefinidas que deverão ser fornecidas juntamente com a linguagem. Concluído este projeto, diversos outros poderão ser desenvolvidos, como por exemplo, a implementação da linguagem DEMO, que é a linguagem de programação de banco de dados (LPBD) do GEOTABA e que será

implementada em MANO. *O* editor de métodos dedicado à linguagem MANO e o Gerente de Armazenamento do ProtoGEO, também são exemplos de projetos que poderão brevemente enriquecer o protótipo de SGBDOO ProtoGEO.

Bibliografia

[ANDR87] Andrews, T. & Harris, C. "Combining Language and Database Advances in an Object-Oriented Development Environment", in OOPSLA'87 Conference, (oct. 1987).

[ATKI87] Atkinson, M.P. & Buneman, O.P. "Types and Persistence in Database Programming Languages", ACM Computing Surveys, 19:2, June 1987.

[ATKI90] Atkinsons, M. et al. "The Object-Oriented Database System Manifesto", SIGMOD'90 Proceedings, (Atlantic City, NJ, May 1990).

[BANC88] Bancilhon, F. et al. "The design and implementation of 02, an object-oriented database system", in [DITT88].

[BANE87a] Banerjee, J. et al., "Data Model Issues for Object-Oriented Applications", ACM Trans. on Office Information Systems, 5:1, Jan. 1987.

[BANE87b] Banerjee, J. et al., "Semantics and Implementation of Scheme Evolution in Object-Oriented Databases", SIGMOD'87 Proceedings, (San Francisco, Calif., May 1987).

[BANE88a] Banerjee, J. et al., "Queries in Object-Oriented Databases", Proc. Int. Conf. Data Eng., Fev. 1988.

[BANE88b] Banerjee, J. et al., "Clustering a DAG for CAD Databases", IEEE Trans. on Software Engineering, 14:11, Nov. 1988.

[BLOO87] Bloom, T. & Zdonik, S.B. "Issues in the Design of the Object-Oriented Database Programming Languages", OOPSLA'87 Proceedings, (Orlando, Florida, Oct. 1987).

[BORN82] Borning, A.H. and Ingalls, D.H., "Multiple Inheritance in Smalltalk-80", Proceedings at the National Conference on Artificial Intelligence, Pittsburg, PA, 1982.

[BRET89] Bretl, R. et al., "The GemStone Data Management System", in [KIM89a].

[BUTT91] Butterworth, P. Otis,A. Stein,J. "The GemStone Object Database Management System", Comunications of the ACM, 34:10, Oct. 1991.

[CAMA91] Camargo, Cláudia S. P., Lisboa F., Jugurta e Monte, Luiz C. M. "Uma Implementação de Relações em Smalltalk". Relatório Técnico Es-235191 do Programa de Engenharia de Sistemas - COPPE/UFRJ, março 1991.

[CARE86a] Carey, M.J. et al., "Object and File Management in the EXODUS Extensible Database System", Proceedings of the 12th VLDB, 1986.

[CARE86b] Carey, M.J. et al., "The Architecture of the EXODUS Extensible DBMS", in [DITT86b].

[CARE89] Carey, M.J. et al., "Storage Management for Objects in EXODUS", in [KIM89a].

[CARR90] Carré, B. and Geib, J.M., "The Point of View notion for Multiple Inheritance", ECOOP/OOPSLA'90 Proceedings, Oct. 1990.

[CATT91] Cattell, R.G.G. Object Data Management: Object-oriented *and Extended Relational Database Systems*. Addison-Wesley, Reading, Mass., 1991.

[CHEN91] Cheng, J.R. and A.R. Hurson, "Effective Clustering of Complex Objects in Object-Oriented Databases", SIGMOD'91 Proceedings, 1991.

[CHOU86] Chou, H. T. & Kim, W. "A unifying framework for versions in a CAD environment", Proceedings of the VLDB'86 Conference, (Kyoto, Japan, aug. 1986).

[COPE84] Copeland, G. and Maier, D., "Making Smalltalk a Database System", SIGMOD'84 Proceedings, 1984.

[DATE86] Date, C.J., An Introduction to *Database Systems*, 4th ed., 1986 Addison-Wesley, Reading, MA.

[DEUX90] Deux, O. et al., "The Story of O_2", IEEE Trans. on Knowledge and Data Engineering, 2:1, March 1990.

[DEUX91] Deux, O. et ai., "The O_2 System", Communications of the ACM, 34:10, Oct 1991.

[DIGI88] Digitalk Inc. *Smalltalk/V* 286 Tutorial *and Programming* Handbook, Digitalk Inc., Los Angeles, CA, 1988.

[DITT86a] Dittrich, K.R. "Object-Oriented Database Systems - A Workshop Report", Proc. of the 5th E-R Conference, Dijon, North Holland Publisher Group, 1986.

[DITT86b] Dittrich, K.R. & Dayal, U. Eds, Proceedings of the International *Workshop* on Object-Oriented *Database* Systems, (Asilomar, California, 1986).

[DITT86c] Dittrich, K.R. "Object-oriented Database Systems: the Notion and the Issues", in [DITT86b].

[DITT88] Dittrich, K. R. Ed. Advances *in* Object-Oriented *Database* Systems. 2nd International Workshop on Object-Oriented Database Systems, (Bad Munster am Stein-Ebernburg, FRG, 1988).

[FISH87] Fishman et al., "IRIS: an Object-Oriented DBMS", ACM Trans. on Office Information Systems, Jan. 1987.

[FISH89] Fishman et al., "Overview of the Iris DBMS", in [KIM89a].

[GOLD83] Goldberg, Adele and Robson, David Smalltalk-80: *The Language* and Its Implementation, Addison-Wesley, Reading, Mass., 1983.

[HORN87] Hornick, M.F. and Zdonik, S.B., "A Shared, Segmented Memory System for an Object-Oriented Database", ACM Trans. on Office Information Systems, 5:1, Jan 1987.

[HUDS87] Hudson, S.E. & King, R., "Object-Oriented Database Support for Software Environments", SIGMOD'87 Proceedings, 1987.

[KAEH81] Kaehler, Ted "Virtual Memory for an Object-Oriented Language", Byte, 6:8, Aug 1981.

[KAEH83] Kaehler, Ted and Krasner, Glenn "LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems", in [KRAS83a].

[KAEH86] Kaehler, Ted "Virtual Memory on a Narrow Machine for an Object-Oriented Language", OOPSLA'86 Proceedings, 1986

[KHOS90] Khoshafian, S. & Franklin, M. & Carey, M.J., "Storage Management for Persistent Complex Objects", Information Systems, 15:3, 1990.

[KIM87] Kim, w. et al. "Operations and Implementation of Complex Objects", Proc. 3rd Int'l Conf. on Data Engineering, 1987.

[KIM88a] Kirn, W. and Chou, H. T. "Versions of Schema for Object-Oriented Databases", Proceedings of the VLDB'88 Conference, (Los Angeles, California, 1988).

[KIM88b] Kim, W. et, al., "Integrating an Object-Oriented Programming System with a Database System", OOPSLA '88 Proceedings, (San Diego, CA, Sept. 1988).

[KIM89a] Kim, W. and Lochovsky, F.H. Eds., Object-Oriented Concepts, *Databases, and Applications*. ACM and Addison-Wesley, 1989.

[KIM89b] Kim, W. et al., "Composite objects revisited", SIGMOD'89 Proceedings, (Portland, OR, June 1989).

[KIM89c] Kim, W. et. ai., "Features of the ORION Object-Oriented Database System", in [KIM89a].

[KIM89d] Kim, W. et. al., "Indexing Techniques for Object-Oriented Databases", in [KIM89a].

[KIM90a] Kim, W. et al., "Architecture of the ORION Next-Generation Database System", IEEE Trans. on Knowledge and Data Engeneering, 2:1, March 1990.

[KIM90b] Kim, W. et al., "Object-Oriented Databases Definition and Research Directions", IEEE Trans. on Knowledge and Data Engeneering, 2:1, Sept. 1990.

[KING87] King, R. & Hull, R. "Semantic Database Modelling: Survey, Applications and Research Issues", ACM Computing Surveys, 19:3, sept. 1987.

[KRAS83a] Krasner, Glenn Ed. Smalltalk-80 Bits of History, *Words of Advice,* Addison-Wesley, Reading, Mass., 1983.

[KRAS83b] Krasner, Glenn "The Smalltalk-80 Code File Format", in [KRAS83a].

[LAMB91] Lamb, Charles et al. "The ObjectStore Database System", Communication of the ACM, 34:10, oct. 1991.

[LÉCL88] Lécluse, C., Richard, P. and Vélez, F. "O_2, an object-oriented data model", SIGMOD'88 Proceedings, (Chicago, 1988).

[LÉCL89] Lécluse, C. and Richard, P. "The O_2 Database Programming Languages", Proceedings of the 15th VLDB Conference, (Amsterdam, Aug. 1989).

[LISB90] Lisboa F., Jugurta "Controle de Versões em Banco de Dados Orientado a Objetos". Monografia da cadeira de Banco de Dados Não Convencionais, COPPE/UFRJ, agosto 1990.

[LISB91] Lisboa F., Jugurta e Monte, Luiz C.M. "O Nível Lógico do GEOTABA". Workshop em Banco de Dados Não Convencionais, COPPEIUFRJ, julho 1991.

[MAIE85] Maier, David et al., "Object-Oriented Database Development at Servio Logic", Servio Logic Development Corporation, IEEE Database Engineering, 8:4, Dec. 1985.

[MAIE86a] Maier, D. et al. "Development of an object-oriented DBMS", OOPSLA'86 Proceedings, 1986.

[MAIE86b] Maier, D. and Stein, J. "Indexing in an Object-Oriented DBMS", in [DITT86b].

[MAIE89] Maier, D., "Making Database Systems Fast Enough for CAD Applications", in [KIM89a].

[MATT89] Mattoso, M.L.Q., Souza J. M., Gonçalves L., Degrazia C. e Rossatto M., "GEOTABA: O Sistema de Gerência de Objetos da Estação TABA". Relatório técnico Es 179188 do Programa de Engenharia de Sistemas - COPPE/UFRJ, 1989.

[MELO88] Melo,R.N. *Banco* de *Dados Não Convencionais: A tecnologia do BD e suas áreas de aplicação. VI* Escola de Computação, Campinas, SP, 1988.

[MONT90] Monte, L.C.M. "O Modelo de Objetos do GEOTABA". Exame de Qualificação para doutoramento em Eng. de Sistemas e Computação, COPPEIUFRJ, Rio de Janeiro, 1990.

[MONT92] Monte, L.C.M., Lisboa F., J., Mattoso,M.L.Q. e Souza,J.M. "Contribuição do ProtoGEO ao Projeto do SGBDOO GEOTABA". Anais do 7. Simpósio Brasileiro de Banco de Dados, (Porto Alegre, RS, Maio 1992).

[MOON89] Moon, David A. "The COMMON LISP Object-Oriented Programming Language Standard", in [KIM89a].

[PENN87] Penney,D.J. and Stein,J. "Class Modification in the GemStone Object-Oriented DBMS", OOPSLA'87 Proceedings, (Orlando, FLA, Oct. 1987).

[RIBE88] Ribeiro, C.D. & Mattoso, M.L.Q. "Considerações sobre Controle e Proteção de Objetos no Sistema de Gerência de Objetos do TABA". Relatório Técnico do Programa de Engenharia de Sistemas da COPPE/UFRJ, ES-178188, 1988.

[ROCH90] Rocha, A.R.C. & Souza, J.M., "TABA: A Heuristic Workstation for Software Development", COMPEURO 90; Tel Aviv, Israel, maio 1990.

[SCHW86] Schwarz et al., "Extensibility in the Starburst Database System", in [DITT86b].

[SHYY91] Shyy, Y.M. & Su, S.Y.W. "K: A High-level Knowledge Base Programming Language for Advanced Database Applications", SIGMOD RECORD, 20:2, Jun 1991.

[STEF86] Stefik, M. and Bobrow, D.G., "Object-Oriented Programrning: Themes and Variations", The AI Magazine, Jan. 1986.

[STON87] Stonebraker, M., "The Design of the POSTGRES Storage System", Proceedings of the VLDB'87 Conference, (Brighton, England, Sept. 1987).

[STON90] Stonebraker, M., "Third Generation Database System Manifesto", SIGMOD'90 Proceedings, (Atlantic City, NJ, May 1990).

[STON91] Stonebraker, M. "Managing Persistent Objects in a Multi-Level Store", SIGMOD'91 Proceedings, 1991.

[STRO86] Stroustrup, Bjarne *The* C++ Programming Language, Addison-Wesley, 1986.

[VELE89] Velez, Fernando et al. "The 02 Object Manager: an Overview", Proceedings of the VLDB'89 Conference, (Amsterdam, 1989).