# MARTE : Um Meta-gerador de Aplicações baseado na Reutilização de Templates

*Eduardo Chaves Faria*

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. Ana Regina C. da Rocha, D.Sc.
(Presidente)

Prof. Jano Moreira de Souza, Ph.D.

Prof. Nívio Ziviani, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 1991

FARIA, EDUARDO CHAVES

MARTE: Um Meta-gerador de Aplicações baseado na Reutilização de Templates [Rio de Janeiro] 1991.

XXII, 112 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1991)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Reutilização de Software

I. COPPE/UFRJ II. Título (série).

Às três mulheres da minha vida: Tânia, Débora e Júnia.

# Agradecimentos

À Ana Regina, pelo apoio, incentivo e verdadeira orientação, sem os quais certamente o caminho teria sido bem mais difícil.

Ao Nívio Ziviani, pelo desprendimento e amizade refletidos em todo o seu empenho pessoal, o que tornou possível que eu realizasse este curso.

Ao Jano Moreira, pela participação na banca examinhadora, críticas e idéias acerca da tese e da continuidade do trabalho.

Ao Guilherme, Vera, Luiz Carlos, Trotta e Claudia Werner pela disponibilidade constante em ajudar nas mais diversas situações e necessidades.

A todos da secretaria da Coppe/Sistemas, que muito além de suas obrigações atenderam a todas as minhas solicitações sempre com presteza e eficiência.

Ao amigo Vinicius Maciel, pela sua disponibilidade em discutir idéias e trocar experiências; uma real contribuição para o trabalho.

À minha esposa Tânia que suportou bravamente a minha ausência durante tanto tempo, e soube preencher todos os espaços com o trabalho e a dedicação que me deram a tranquilidade necessária para os estudos em tempo integral.

Às minhas filhas Júnia e Débora pelo amor e carinho autênticos, apesar de não compreenderem o afastamento do pai.

Ao meu Deus, Criador e Mantenedor de todas as coisas, a quem tudo devo, inclusive esta tese.

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

# MARTE : Um Meta-gerador de Aplicações baseado na Reutilização de Templates

Eduardo Chaves Faria

SETEMBRO, 1991

Orientador: Prof. Ana Regina Cavalcanti da Rocha
Programa: Engenharia de Sistemas e Computação

Reutilização é um dos paradigmas fundamentais de desenvolvimento. A sua definitiva concretização na Engenharia de Software depende de uma profunda compreensão dos conceitos envolvidos, e de experimentos que identifiquem a estratégia que represente o melhor caminho para a reusabilidade.

Esta tese apresenta as principais questões relacionadas ao conceito de reutilização de software, e descreve uma abordagem baseada na definição e manipulação de "templates" como solução que busca viabilizar o reuso de código. Trata-se de uma ferramenta de desenvolvimento de software centrada em um gerador de aplicação, cujos padrões de arquitetura são parametrizados na forma de templates externos. Por sua vez, os templates são agrupados e classificados em uma biblioteca segundo a linguagem e domínio de aplicação. Esta abordagem pretende minimizar os problemas técnicos existentes através de uma solução conjugada que visa aproveitar as características relevantes de tecnologias já conhecidas.

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

MARTE : A Meta-Generator of Appications
based on Reuse of Templates

Eduardo Chaves Faria
SEPTEMBER, 1991

Thesis Supervisor: Prof. Ana Regina Cavalcanti da Rocha
Department: System and Computation Engineering

Reuse is one of the fundamental development paradigms. To make reuse a reality in Software Engineering depends on a clear understanding of its concepts and experiments that identify the best strategy to reusability.

This thesis presents the main issues related to software reuse concept and describes an approach based on templates definition and manipulation as a solution that makes the code reuse feasible. It is a question of a software development tool centered on an application generator in which the architecture patterns are external templates. These templates are grouped and classified in a library according to the language and application domain. This approach intends minimize the technical problems through a matched solution that aims to make the most of outstanding features of known technologies.

# SUMÁRIO

# 1 - INTRODUÇÃO

Os primeiros computadores fabricados em escala industrial eram máquinas bastante lentas, com pouca capacidade de memória e de custo extremamente elevado. Estas condições limitavam o porte e volume de problemas que podiam ser processados.

Com o surgimento da microeletrônica os computadores experimentaram uma dramática evolução, crescendo em suas potencialidades e tornando-se mais acessíveis com a diminuição de seus custos. Os problemas levados aos computadores aumentaram em número, tamanho e complexidade. Ao final dos anos 60 deu-se a chamada crise de software. Por esta época tornou-se crítico o desenvolvimento de programas pela absoluta falta de técnicas e metodologias. Foi então que as atenções se voltaram para a questão do software, estabelecendo-se conceitos e disciplinas que impulsionaram esta área.

Ainda que tenham surgido novas linguagens de programação, sido estabelecidos conceitos, técnicas e metodologias, criadas ferramentas ou mesmo ambientes integrados para desenvolvimento de programas, é claramente observado um descompasso entre as evoluções do hardware e do software desde aqueles tempos até os dias de hoje.

A produtividade na criação de software aumentou apenas de 3% a 8% ao ano, nos últimos 20 anos, enquanto a razão custo/desempenho do hardware tem diminuído a 20% ao ano [11], [20].

À medida que o hardware torna-se mais barato, os computadores são usados em novas áreas de aplicação, o que aumenta a demanda por software e exige maior número de profissionais qualificados.

A crise de software atual não é um problema de sistemas pequenos. Os métodos existentes parecem adequados para situações em que apenas um programador, ou uma reduzida equipe de programadores, são envolvidos na construção de um sistema de pequeno ou médio porte. A crise se verifica quando os membros de uma equipe numerosa cooperam no desenvolvimento de um sistema grande e complexo. Em sistemas desta natureza as especificações são obscuras e ambíguas [20] e o desenvolvimento nem sempre é um processo "top-down" estruturado [41].

A idéia de reutilização surge neste contexto como esperança de aumentar a produtividade de software (uma ordem de grandeza ou mais) e diminuir sobremaneira os custos de desenvolvimento.

Esta esperança baseia-se na constatação de que o projetista de software geralmente inicia um projeto como se o produto almejado fosse único e singular. A cada novo sistema começa do mesmo ponto, e quase nada considera dos esforços dispendidos em outros projetos. Esta atitude desconhece o fato de 40 a 60% do código de programa existente ter se repetido em mais de uma aplicação [11].

A reutilização de componentes já conhecidos permite a construção de um sistema com menos produção de software, porém mantendo-se a funcionalidade desejada. Isto não só aumenta a produtividade, como permite amortizar o custo de componentes reusáveis sobre os custos de desenvolvimento de vários sistemas. Além disso, uma economia adicional é obtida com a redução dos custos de manutenção [35], dado que os componentes tendem a ser largamente testados nos sistemas de que fazem parte. Pelo mesmo motivo, o reuso de software conduz a sistemas mais confiáveis.

A reutilização estabelece um mecanismo de transmissão do conhecimento acerca do produto de software (estrutura, componentes e funções). Este conhecimento, muitas vezes obtido durante o processo de construção, é perdido se não existe um meio de ser armazenado e transmitido a outros projetistas [4]. é provável que a perda deste conhecimento ao longo dos anos tenha em muito contribuído para as dificuldades hoje encontradas na Engenharia de Software.

Para que o reuso de software torne-se realidade é preciso definir uma estratégia que represente o melhor caminho para a sua concretização. As diversas tecnologias que viabilizam o reuso de software devem ser devidamente exploradas com experimentos que possam avaliar as suas reais possibilidades.

Esta tese apresenta as principais questões relacionadas ao conceito de reutilização de software, e descreve o sistema MARTE (Meta-gerador de Aplicações baseado na Reutilização de TEmplates) como proposta de mecanismo para viabilizar a sua aplicação. Trata-se de uma ferramenta de desenvolvimento de software centrada em um gerador de aplicação, cujos padrões de arquitetura são parametrizados na forma de templates externos. Por sua vez, os templates são agrupados e classificados em uma biblioteca segundo a linguagem e domínio de aplicação. Este sistema pretende minimizar os problemas técnicos existentes através de uma solução conjugada que visa aproveitar as características relevantes de tecnologias já conhecidas.

Inicialmente, o Capítulo 2 apresenta um estudo detalhado sobre o reuso de software, onde procura-se definir o conceito e suas implicações, relatar experiências do passado, discutir as tecnologias conhecidas para sua aplicação e apontar perspectivas futuras nesta área.

O Capítulo 3 descreve o sistema MARTE através de seus fundamentos e da especificação, projeto e implementação de um protótipo.

Finalmente, no Capítulo 4 são apresentadas algumas conclusões de caráter geral, discutidos os resultados do experimento e destacadas as perspectivas de continuidade deste trabalho.

# 2 - REUTILIZAÇÃO DE SOFTWARE

A Engenharia de Software é a disciplina que reúne conceitos, técnicas e métodos que podem ser utilizados no desenvolvimento de um software confiável e de boa qualidade, a prazos e custos razoáveis. O conceito de reutilização, ainda não totalmente explorado, pode se revelar extremamente útil e poderoso na concretização deste objetivo.

A compreensão do real valor do reuso de software, evitando-se o mito ou entusiasmo exagerado, exige uma avaliação apurada das experiências no passado e um estudo criterioso das possibilidades no presente.

## 2.1 - Conceitos Básicos

A história do progresso das ciências fornece um paradigma para a reusabilidade. De fato, o progresso é realizado pelo desenvolvimento e refinamento de idéias, havendo contribuição entre diferentes áreas. A linguagem é o meio de representação dos princípios teóricos e resultados práticos que viabiliza a transmissão do conhecimento. Na própria ciência da computação alguns dos trabalhos de maior sucesso são a fusão e formalização de técnicas e métodos bem sucedidos [20].

Na área da Engenharia de Software o conceito de reutilização não é novo. Já em 1967 surgiu a proposta de um catálogo de componentes para construção de software, tal qual é feito com componentes mecânicos e eletrônicos [25]. O reuso de trechos de programa através de macros, construções de linguagens de alto nível e bibliotecas de rotinas em código objeto (extensão da linguagem) têm constituído os fundamentos

de engenharia para construção de programas. Outras formas de reuso acontecem através da padronização de software gráfico e software de banco de dados, ou mesmo pelo uso generalizado de pacotes de software.

Na maioria dos exemplos mais simples de reuso de software, programadores reutilizam seus próprios programas ou de programadores da mesma equipe. A nível extremamente informal, pode-se considerar reuso sempre que um programador aproveita experiências vividas por ele mesmo no desenvolvimento de outros programas.

Se o conceito de reutilização de software já é conhecido, e seu potencial reconhecido para aumentar a produtividade e qualidade dos sistemas, quais são os problemas para que isto se concretize?

Segundo Tracz, o reuso de software não é um problema meramente técnico. A maioria dos problemas são de ordem psicológica, sociológica ou econômica [34], [35].

Um bom exemplo disto é o recente sucesso das indústrias de software japonesas. Este sucesso não se baseia em avanços tecnológicos. A reutilização é conseguida pela integração de técnicas conhecidas de diferentes disciplinas, tais como engenharia de produção, gerência de recursos, controle de qualidade, engenharia de software e psicologia industrial.

É crítico estabelecer e forçar padrões sem a motivação de programadores para desenvolver componentes e projetar software com reusabilidade. O programador resiste à idéia do reuso por considerar que inibe a sua criatividade. Um dos segredos do sucesso japonês foi treinar o programador no sentido do

comprometimento com padrões que facilitam o reuso de software. De fato, não há condições de reutilizar software a menos que este seja cuidadosamente projetado para ser reusável [28].

O problema de ordem econômica é o alto investimento inicial exigido para que se possa explorar a reutilização de software. Necessita-se mão de obra, tempo e dinheiro para construir bibliotecas suficientemente volumosas e ricas em componentes de software. A ausência de tais bibliotecas impede que a tecnologia de reutilização possa evoluir espontaneamente [2], [34].

Quanto aos problemas técnicos, estes dependem muito da forma e mecanismo através do qual se dará a reutilização. Em grandes bibliotecas de componentes reusáveis, uma questão importante é determinar métodos de pesquisa para localizar e recuperar o componente certo. Se este componente deve conter informações de projeto, a falta de um modelo adequado para sua representação é um problema fundamental.

### 2.1.1 - O Processo de Reuso

A reutilização de componentes de software envolve três passos essenciais [25]: acesso, entendimento e adaptação.

Inicialmente é preciso encontrar o componente para reuso. Em seguida, há necessidade de compreender a sua função (o que faz) para saber se é preciso modificá-lo. Caso afirmativo, o entendimento da sua estrutura interna (como faz) permite que sejam realizadas alterações visando adaptar o componente aos novos requisitos. Pode-se, ainda, considerar uma quarta etapa

de composição, onde o componente, modificado ou não, é integrado ao software em desenvolvimento.

Boldyreff [4] descreve o processo de reuso mais amplo, apresentado na figura 2.1, incluindo também a identificação de objetos reusáveis e as operações realizadas em cada etapa.

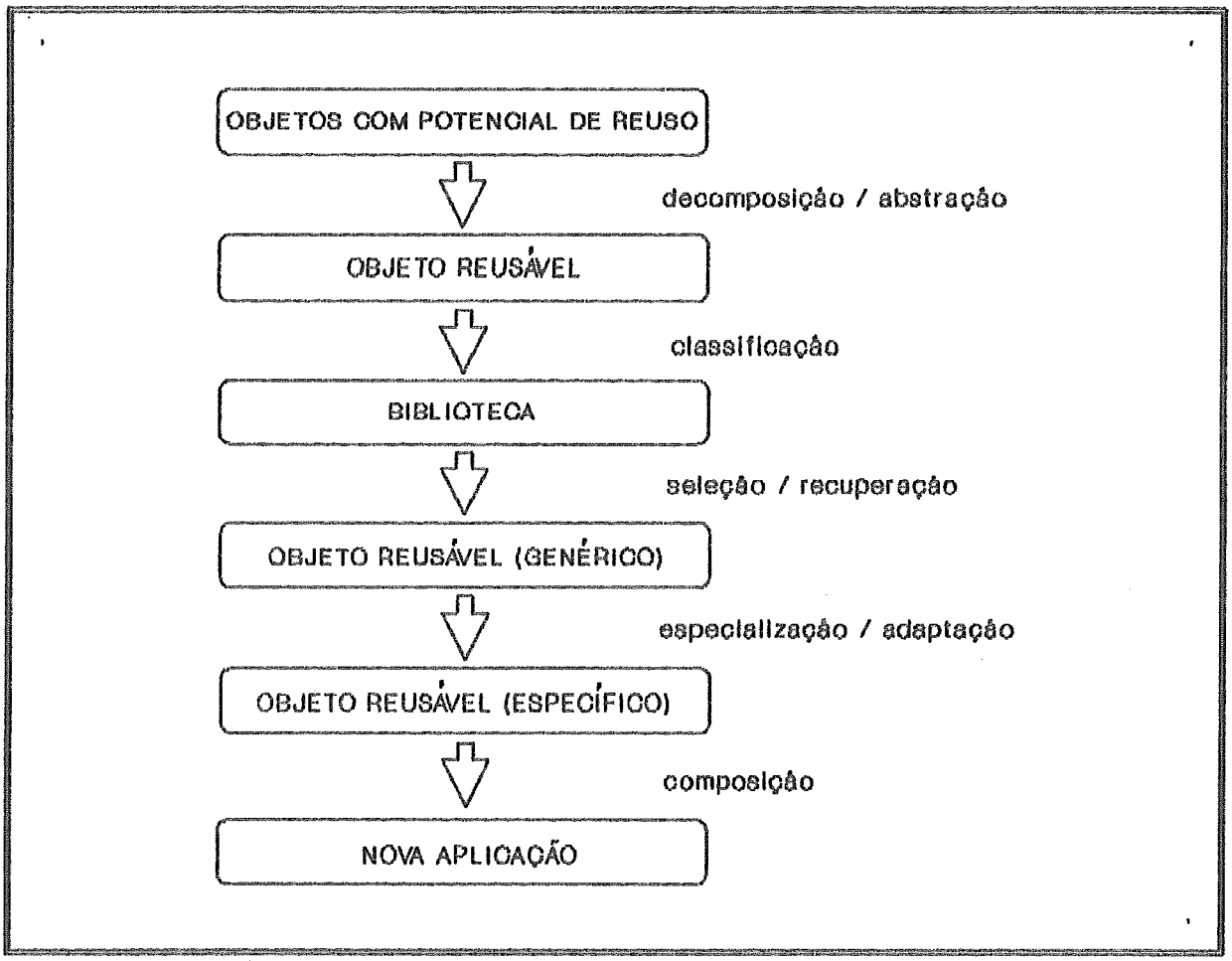



Fig. 2.1 - O processo de reuso [4]

O reconhecimento da oportunidade de reuso envolve a decomposição de grandes sistemas de software em seus conceitos componentes, seguindo-se a extração dos conceitos específicos com potencial de reuso. A partir de um número de conceitos similares específicos é possível abstrair um conceito genérico de software. Uma vez devidamente classificado e catalogado em

uma biblioteca, este conceito pode ser selecionado para reuso. Isto pode requerer a especialização, ou adaptação, deste conceito genérico para atender às especificidades da nova aplicação.

### 2.1.2 - Domínio de Aplicação

O conceito de domínio de aplicação é de fundamental importância no reuso de software. Pode-se definí-lo como sendo uma área de aplicação, ou esfera de atividades, para a qual são desenvolvidos os sistemas de programação [27]. Neighbors enfatiza este conceito ao longo de todo o seu trabalho sobre o sistema DRACO [20].

Quando se pensa inicialmente em reuso é normal pensar em tudo que é comum, geral, largamente aplicável, ou seja, componentes de pequeno a médio tamanho (operações com strings, classificações, pesquisas, etc.). Ocorre que estes componentes respondem apenas por uma pequena porcentagem do tamanho das aplicações típicas. A porção maior incorpora conhecimentos para lidar com o domínio. Por exemplo, um sistema de automação bancária incorpora mais conhecimento sobre processamento de documentos bancários do que sobre pesquisas e operações com strings.

Prieto-Diaz considera que análise de domínio é a chave para o sucesso do reuso de software [26]. Na análise de domínio são identificados objetivos e operações comuns a todos os sistemas dentro do mesmo domínio, e um modelo é definido para descrever seus relacionamentos. Os componentes resultantes desta análise são mais apropriados ao reuso porque eles capturam a funcionalidade essencial requerida no domínio.

Tudo isto não significa que pequenos componentes de aplicação geral não devam ser reutilizados. Apenas o retorno obtido será menor. Neighbors entende que tais componentes pertencem a domínios de modelagem (ou projeto)[20]. Eles escondem o conhecimento de engenharia necessário para construir sistemas. Do ponto de vista hierárquico os domínios de modelagem estão abaixo dos domínios de aplicação. O conhecimento obtido do domínio de aplicação agrega objetos dos domínios de modelagem. Por exemplo, pode-se modelar objetos e operações do domínio de sistemas administrativos em notação do domínio de modelagem banco de dados. Em outra situação, estes objetos poderiam ser modelados de forma diferente para construção de diferentes aplicações.

### 2.1.3 - Reuso e Programação Orientada a Objetos

A reusabilidade (adaptabilidade e potencial de reuso) de um componente de software depende da qualidade da análise de domínio realizada e do nível em que o módulo é parametrizado para refletir os resultados desta análise.

As facilidades de parametrização da linguagem nem sempre suportam o grau e forma de adaptabilidade desejada. Certas características da linguagem facilitam o desenvolvimento de software reusável, mas a linguagem por si mesma não resolve o problema da reutilização [34].

Neste particular, as linguagens de programação orientadas a objeto oferecem maiores vantagens no reuso de módulos de software [32]. Encapsulamento e herança são as principais características destas linguagens que acentuam a reusabilidade.

O encapsulamento significa que um objeto pode ser tratado como uma "caixa preta". Tudo que deve ser conhecido sobre um objeto é seu protocolo ou interface, ou seja, que mensagens ele responde e qual o seu comportamento.

A herança se refere à habilidade para descrever novos objetos, ou classes de objetos, pela especialização de diferenças em relação a outros objetos já conhecidos.

Estas características conduzem a duas formas para reuso de código [17]. Na primeira, utilizam-se diretamente instâncias de classes já existentes. Na segunda forma, o reuso de classes modificadas pode ser feito via sub-classes.

Ainda que possuam as vantagens mencionadas, o reuso de software através de linguagens orientadas a objeto apresenta alguns inconvenientes. A curva de aprendizado destas linguagens é diretamente proporcional ao tamanho e à complexidade da biblioteca de classes, pois o programador deve conhecer as classes a partir da leitura de manuais e códigos de programa. Os sistemas existentes, tais como Smalltalk-80 e Objective-C, não possuem ferramentas suficientemente amigáveis e poderosas para recuperação de classes. O "browser" do sistema Smalltalk-80 é amigável, mas não o bastante para usuários que conhecem pouco sobre as classes definidas.

Além disso, a recuperação de classes torna-se difícil pela diferença de terminologia entre usuários e sistema. Os programadores não podem descrever os requisitos em seus próprios termos.

Estas questões, somadas ao problema clássico de pesquisa e recuperação em grandes bibliotecas, justificam a criação de

ferramentas [10] e ambientes [32] de suporte ao reuso de software orientado a objetos.

Porém, ainda que existam mecanismos para vencer os problemas de ordem técnica acima mencionados, o reuso de software através de linguagens orientadas a objeto esbarra em uma questão de ordem econômica extremamente relevante. A grande maioria do software existente está escrita em linguagens clássicas como Cobol, Fortran, Pascal, etc. A reutilização de todo este software dependerá de sua tradução para as linguagens orientadas a objeto. Porém, os usuários devem mudar a linguagem base para se beneficiarem da reutilização em atividades de manutenção e/ou expansão destes sistemas. Isto implicaria em grandes investimentos no setor de produção, treinamento de todo o pessoal, etc. Esta mesma questão justifica a persistência de certas linguagens de programação.

### 2.1.4 - Reuso e Qualidade de Software

Qualidade é um conceito difícil de ser definido precisamente. Rocha [29] associa qualidade de software ao conjunto de propriedades a serem satisfeitas em determinado grau, de modo que o software corresponda às necessidades do usuário.

Ainda que a qualidade de um software possa ser avaliada, é muito difícil conferir qualidade ao software que não a possui. É neste sentido que deve-se garantir a qualidade ao longo do desenvolvimento de software através da aplicação combinada de controle, revisão, metodologia e teste [24]. Garantir a qualidade do software é reduzir as dúvidas e riscos acerca do desempenho do produto de software.

O principal incentivo para a reutilização é a potencial redução de custos no processo cada vez mais dispendioso de desenvolvimento de software. Contudo, uma vantagem adicional de igual importância é a obtenção de sistemas de melhor qualidade, dado que os componentes reusáveis tendem a ser largamente testados nos sistemas de que fazem parte. Além disto, é possível um esmero maior na produção deste software, dada a redução de seu custo final. A indústria de software japonesa não se interessou apenas no ganho de produtividade, mas perseguiu o reuso também pelo item de qualidade [35].

O reuso associado à qualidade de software é ainda uma questão de investimento. Analogamente ao que acontece no hardware, a economia proporcionada pelo reuso de componentes pode ser utilizada na melhoria dos padrões de controle de qualidade. A distribuição dos custos sobre vários projetos permite, por exemplo, aplicar rigorosa verificação de software nos componentes reusáveis [9]. Normalmente, a verificação e validação rigorosas são consideradas muito dispendiosas e consomem muito tempo para serem utilizadas com frequência. Elas são as primeiras a serem sacrificadas toda vez que é preciso cortar os custos em um projeto. Estas técnicas são aplicadas apenas no desenvolvimento de software com requisitos de muito alta integridade.

Para calcular o benefício proporcionado pelo reuso de software deve-se primeiro estimar o custo das atividades sem reuso, e compará-las ao custo com reuso [2]. É nítido o ganho direto que se obtém na redução de custos de desenvolvimento, porém os ganhos obtidos com a melhoria da qualidade são de uma extensão muito maior. Eles se propagam por todo o ciclo de vida do software, quer seja diminuindo gastos com sua manutenção, ou evitando perdas maiores caso o seu desempenho fosse comprometido.

"A qualidade mais importante de um software reusável é ser um software de qualidade" [35]. Esta afirmativa será mais fortalecida quando se tornar realidade o reuso de software em largo espectro [2], onde será possível reutilizar a especificação de requisitos, projetos, módulos de código, documentação e dados de teste.

## 2.2 - Tecnologias para Reutilização de Software

O conceito de reuso é bastante amplo e pode ser abordado sob diferentes aspectos. Biggerstaff e Richter [3] identificam e avaliam os mecanismos, ou tecnologias, através dos quais pode-se dar a reutilização de software. Conforme a natureza dos componentes envolvidos, as tecnologias de reuso podem ser classificadas em dois grandes grupos: tecnologias de composição e tecnologias de geração.

O primeiro grupo caracteriza-se pelos componentes serem atômicos e idealmente passivos. O reuso consiste na derivação de novos programas pela composição de objetos de software. Estes objetos podem representar o código, projeto ou especificação de programas já desenvolvidos.

Nas tecnologias de geração os componentes reusáveis não são identificados com facilidade. Usualmente, são padrões em um programa gerador. Estes padrões refletem-se de maneira difusa e pouco observável nas estruturas geradas.

Para caracterizar e comparar os diferentes mecanismos em cada grupo, apontando suas vantagens, deficiências e perspectivas, é preciso primeiro estabelecer parâmetros de

avaliação. Biggerstaff e Richter introduzem alguns parâmetros na forma de três dilemas:

GENERALIDADE X BENEFÍCIO

Quanto mais geral a tecnologia, menor o benefício proporcionado. A generalidade é medida pela faixa de domínios de aplicação abrangida pela tecnologia. é neste sentido que um gerador de aplicações é mais vantajoso que uma linguagem de alto nível, porém suas possibilidades de uso são bem menores. A figura 2.2 apresenta uma caracterização de diversas tecnologias segundo este dilema.

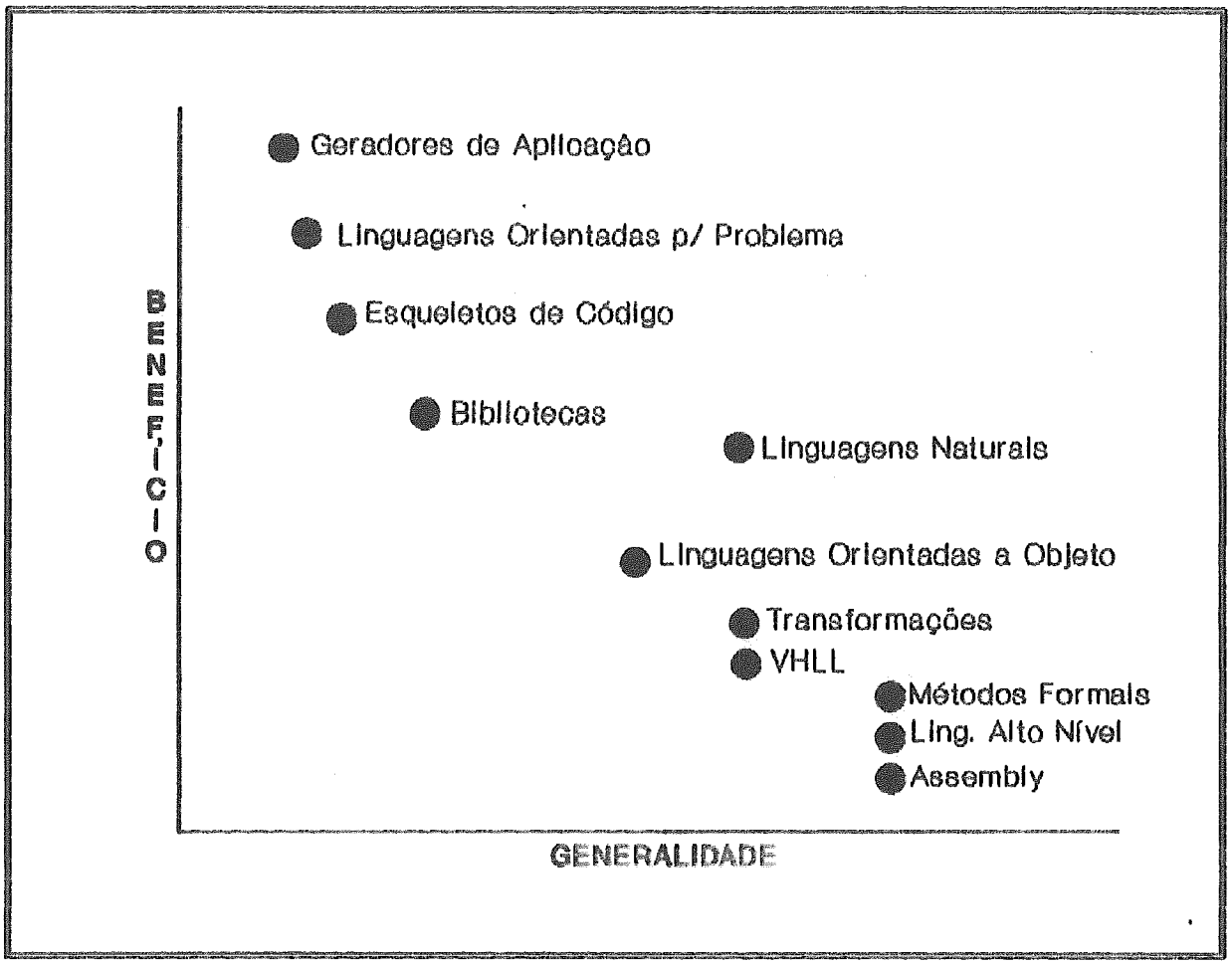



Fig. 2.2 - Generalidade X Benefício [3]

TAMANHO DO COMPONENTE X POTENCIAL DE REUSO

Quanto maior o componente, maior o benefício alcançado com o seu reuso. Contudo, a medida que o componente cresce em tamanho torna-se mais específico, diminuindo as possibilidades de sua aplicação.

CUSTO DE CRIAÇÃO DA BIBLIOTECA

A reutilização de software tem um potencial de retorno mais a longo prazo. Para que haja um benefício significante é necessário uma biblioteca rica e numerosa em componentes reusáveis. Isto por sua vez exige um grande investimento inicial.

## 2.2.1 - Composição

A idéia mais óbvia e natural para a reutilização é criar bibliotecas de módulos de software reusáveis. Um exemplo bem sucedido desta idéia são as clássicas bibliotecas de programas (funções matemáticas e estatísticas, rotinas gráficas, etc.) onde os componentes são aplicados sem a possibilidade de modificação. Nestes casos é preciso saber apenas a função e os parâmetros de cada módulo.

No caso mais geral é necessário adaptar o componente aos novos requisitos. Isto amplia o potencial de reuso na medida em que aumenta a generalidade do mecanismo. Para adaptar o componente, além de conhecer a sua função e seus parâmetros, o programador precisa entender como a função é realizada. Estas informações devem estar armazenadas na biblioteca.

O processo de reuso, conforme descrito na seção 2.1.1 aponta para os seguintes problemas desta tecnologia:

ACESSO AO COMPONENTE

Dado os requisitos do componente desejado, o problema consiste em localizá-lo e recuperá-lo de uma grande biblioteca. Isto significa procurar por todos os componentes similares, porque nem sempre os componentes disponíveis combinam perfeitamente com os requisitos. O programador examina os componentes similares e seleciona aquele que representa menor esforço de adaptação.

A solução proposta por Prieto-Diaz é um esquema de classificação multiface, semelhante aos usados em bibliotecas científicas, com um mecanismo de avaliação de componentes similares [13], [25], [33]. Na classificação multiface cada componente é descrito pela função que realiza, como realiza a função e seus detalhes de implementação. Medidas de distância conceitual entre os termos do vocabulário utilizado permitem identificar as descrições similares.

ENTENDIMENTO DO COMPONENTE

O usuário deve sempre compreender o componente para saber se corresponde exatamente às especificações, ou se é necessária alguma modificação. Neste último caso, ele também deve avaliar os esforços que serão exigidos para adaptação.

O usuário precisa criar um modelo mental do componente para usá-lo de forma apropriada. Esta tarefa pode ser difícil dependendo da experiência do usuário e das características do componente de software (tamanho, complexidade, documentação,

etc). O problema pode ser amenizado pelo uso de sistemas de hipertexto [23] que, integrando texto e diagramas gráficos, representariam as informações pertinentes ao módulo: arquitetura, decisões de projeto, restrições, etc. [3], [4]. Sistemas de hipertexto são instrumentos de representação bastante informais e de grande poder de expressão [41].

ADAPTAÇÃO DO COMPONENTE

A reutilização de software sem a necessidade de modificar componentes é ideal. Na prática, para que haja benefício significativo, a etapa de adaptação ocorre freqüentemente.

A dificuldade aqui encontrada depende do quanto o componente corresponde aos requisitos, do grau de seu entendimento e da habilidade do projetista. Um bom esquema de classificação ajuda minimizando recuperação de componentes irrelevantes. Contudo, a modificação é tarefa essencialmente do domínio humano, havendo poucas ferramentas de auxílio.

### 2.2.1.1 - Reuso de Código Vs. Reuso de Projeto

Conforme a natureza do componente, a reutilização pode ser feita a nível de produtos específicos ou a nível de idéias e conhecimentos utilizados para elaborar tais produtos.

Na construção de software o caminho mais natural é a reutilização de módulos de programa representados em alguma linguagem de programação. Os componentes de maior poder de reuso são aqueles de menor tamanho, que por isto se aplicam a

um amplo domínio de aplicação. Contudo, são de pouco benefício pois implicam em maior esforço para construir grandes estruturas, muitas vezes não compensando a economia de reuso. Aumentando o tamanho do componente diminui a sua reusabilidade, pois torna-se ainda mais específico. Embora o benefício seja maior, são necessários mais esforços para o seu entendimento e sua adaptação.

Independente do tamanho do componente, o fato de estar representado em uma linguagem de programação também diminui o seu potencial de reuso. A linguagem restringe os domínios de aplicação aos quais se aplicam o componente.

Tudo isto impõe um limite de benefício alcançado pelo reuso de código. A experiência mostra que menos da metade de um sistema qualquer pode ser construído a partir de componentes desta natureza [3].

A reutilização de idéias na construção de software corresponde ao reaproveitamento de especificações ou informações de projeto. O reuso a este nível implica em maior benefício, pois amplia o espectro de domínios de aplicação para um mesmo componente.

As informações de projeto não dependem dos aspectos de implementação e consistem no que efetivamente é reutilizado pelo projetista. Em verdade, no processo de reuso de um módulo de programa fonte, entender o componente significa extrair os fatores individuais de projeto escondidos no código. São estes fatores que permitem combinar o componente com os requisitos e, se for o caso, instruir a fase de adaptação. A dificuldade de identificar esta informação advém das inúmeras combinações possíveis de estruturas de dados e algoritmos para implementar a mesma idéia de projeto.

O maior problema do reuso de software neste nível é a falta de um modelo adequado para representar as informações de projeto.

Estas informações incluem não apenas o projeto (restrições, métodos e objetivos) mas também o processo de projeto (decisões e refinamentos) [41]. A informação de projeto não deve manter muitos detalhes a ponto de diminuir a reusabilidade do componente, e nem ser tão pouco detalhada que inviabilize o reuso pelo maior esforço de adaptação requerido.

O modelo de representação deve permitir graus controlados de abstração, ou seja, representar estruturas conceituais precisas mas livres de detalhes. A representação da essência do projeto é que permite aplicar conceitos a partir de um domínio para estruturas em um contexto inteiramente diferente.

Jameson [12] apresenta um modelo para reuso de projeto a partir da representação das definições de um módulo (nome, função e parâmetros) e seus detalhes de concepção (algoritmo). Embora a proposta seja simples e bastante geral, pois independe da linguagem de programação, não existe reuso dos esforços de implementação. O sistema Draco [20] é um exemplo mais sofisticado e com objetivos mais amplos, pois tenciona o reuso em todos os níveis: especificação, projeto e implementação.

### 2.2.1.2 - Extensão de Programas

A técnica de extensão de programas tem como objetivo aumentar a capacidade de um sistema de software dinamicamente, ou seja, sem a necessidade de modificação do código fonte.

Isto envolve um certo reuso, pois um novo sistema pode ser construído a partir da composição de programas existentes.

Este mecanismo encoraja o desenvolvimento de software de forma incremental, diminuindo custos de desenvolvimento de aplicações extensíveis por natureza.

Ainda que o potencial de reuso seja muito pequeno, a extensão de programas é bastante flexível e, quando aplicável, oferece um grande benefício. Notkin descreve o protótipo Interpretador de Extensão (EI) que implementa um mecanismo de extensão dinâmica baseado no sistema Unix [22].

O objetivo mais amplo de integrar sistemas desenvolvidos sob fundamentos diferentes e para ambientes operacionais também diferentes é perseguido por estudos que visam aumentar a inter-operabilidade de programas. A idéia básica é definir um modelo conceitual único capaz de representar objetos que possam ser compartilhados pelos programas componentes [42].

### 2.2.2 - Geração

Sistemas baseados em geração definem outro grupo de tecnologias através das quais torna-se possível a reutilização de software. Rich e Waters chegam mesmo a considerar que "o coração da programação automática é o reuso" [28]. De fato, o reuso não interfere, nem compete, com a automação. Reuso e automação são complementares na medida em que a automação tenta transferir para o computador tanto trabalho quanto possível, enquanto o reuso procura tornar mais eficiente o uso das atividades que não podem ser totalmente automatizadas.

Sistemas desta categoria caracterizam-se por esconderem nas suas estruturas os padrões que estão sendo reutilizados. Enquanto no mecanismo de composição pode-se identificar um componente, antes e após o seu uso, na geração apenas observa-se uma semelhança nas arquiteturas das diferentes instâncias geradas do mesmo programa.

Conforme a propriedade enfatizada, pode-se dividir os sistemas de geração em três categorias distintas, a saber: sistemas baseados em linguagem, sistemas baseados em transformação e geradores de aplicação.

### 2.2.2.1 - Sistemas baseados em Linguagem

Os sistemas geradores baseados em linguagem enfatizam o uso de uma linguagem de largo espectro para descrever o desenvolvimento do software, desde a especificação até a sua implementação final [5]. Os requisitos, representados na linguagem por construções de alto nível, são transformados por refinamentos em primitivas a nível de implementação. Em um dado instante, o sistema que está sendo refinado poderá incluir declarações de diversos níveis de abstração.

A linguagem SETL, baseada em operações sobre conjuntos, é um exemplo significativo desta categoria [7].

Ainda que seja possível o reuso das especificações, as oportunidades são bastante reduzidas dada a generalidade da linguagem. Para que haja maior potencial de reuso esta abordagem deve prover um mecanismo de especialização da linguagem para certos domínios de aplicação [20].

### 2.2.2.2 - Sistemas baseados em Transformação

Os sistemas desta categoria focalizam a função, estrutura e operação de transformações na evolução de especificações de alto nível até programas executáveis. Esta evolução acontece por refinamentos em vários estágios, onde uma transformação substitui alguma construção de alto nível por outra mais concreta que realiza a mesma função. A descrição do sistema é refinada através de uma série discreta de linguagens de espectro estreito.

Nesta categoria de sistemas, os objetos reusáveis são as próprias transformações que podem ser reaplicadas, em combinações diferentes, com ou sem alterações, para geração de sistemas com alguma semelhança [5].

O sistema de desenvolvimento de software Auto Star exemplifica esta tecnologia [43].

### 2.2.2.3 - Geradores de Aplicação

Os geradores de aplicação constituem a categoria de sistemas que oferecem um grande benefício no reuso de padrões de arquitetura reconhecidos em um domínio particular. Estes padrões encontram-se embutidos no projeto do gerador e são reutilizados durante a geração de instâncias específicas dos sistemas desejados.

Os padrões de arquitetura usados na geração são determinados pela identificação de uma família genérica de programas com grandes oportunidades de parametrização. Os

valores dos parâmetros podem ser especificados pelo usuário através de um diálogo mantido com o sistema [1].

A escolha de um domínio de discurso para um gerador de aplicação e o seu projeto correspondente, incluindo a interface para entrada de parâmetros, requer um profundo conhecimento do domínio de aplicação.

Um gerador de aplicação pode ser visto como a saída de uma análise de domínio [11]. Quanto mais fortemente restrito for o domínio de aplicação, maior e mais rápido o benefício, mesmo que isto signifique um poder de reuso limitado. Este benefício pode significar de 60 a 90% de reuso, dependendo da quantidade de lógica que deve ser desenvolvida fora do gerador de aplicação [3].

Luker e Burns definem precisamente um gerador de programas como sendo "um programa que aceita como entrada diálogos estruturados de seres humanos e produz em uma faixa de linguagens de alto nível um programa bem estruturado e comentado, feito sob medida para a especificação e requisitos particulares do usuário". Os principais componentes de um gerador de programas são: módulo de diálogo, módulo intermediário e módulo gerador de código. Em um grande número de casos o módulo intermediário pode ser omitido. Normalmente, entre as suas funções incluiem-se a expansão e otimização de representações intermediárias do programa a ser gerado.

Uma classe particular de geradores de aplicação constitui-se das denominadas Linguagens de Quarta Geração (L4Gs). Tipicamente, o usuário fornece uma especificação de forma interativa e é gerado um programa executável, que tem acesso a uma base de dados e imprime alguns relatórios. Embora estes sistemas proporcionem grandes aumentos de

produtividade de software (quando aplicáveis), a maioria possui a desvantagem de não gerar programas em linguagens tradicionais [31]. Além de questões estratégicas envolvidas (p.ex.: investimento existente, manutenção, etc.), esta característica inviabiliza soluções conjugadas, o que permitiria o reuso de grande quantidade de software já existente.

## 2.3 - Avaliação e Perspectivas

A idéia de reutilização é uma esperança concreta na Engenharia de Software. Pode-se aumentar a produtividade, diminuir os custos e viabilizar a construção de sistemas grandes e complexos.

A efetiva concretização do reuso de software passa pela solução de problemas técnicos e não-técnicos. é preciso criar uma nova consciência nos programadores e projetistas - "software reusado não é o mesmo que software reusável" [35]. Sem que haja planejamento não há reusabilidade suficiente.

é preciso haver vontade política. O reuso de software não acontecerá de forma espontânea. Para sua exploração é imprescindível uma capitalização inicial na formação de grandes repositórios de software.

é preciso definir uma estratégia que representa o melhor caminho para a reusabilidade. As diversas tecnologias que viabilizam o reuso de software devem ser devidamente exploradas com experimentos que possam avaliar as suas reais possibilidades.

"A chave para a reusabilidade de software é a abstração" [4]. é necessário encontrar uma conexão entre o que já é conhecido e o que se requer do conhecimento; entre teorias e conceitos realizados no software existente e aqueles que incluem uma solução para os requisitos. Esta conexão deve estar representada nos componentes.

Há também necessidade de padrões que ultrapassem qualquer componente, ou conjunto de componentes. Estes padrões se relacionam fortemente com a noção de arquiteturas que impõem amplos modelos estruturais em sistemas. Em hardware estes padrões são estabelecidos a nível de chips ("enable", linhas de dados, linhas de endereços, etc.) e de placas (S-100, multibus, etc.), o que permite ligar os componentes.

No caso do software os conceitos reusáveis e padrões de arquiteturas não são generalizados. Eles dependem do domínio de aplicação. Neste sentido, a análise de domínio é o fator chave no sucesso da reusabilidade. Sem estabelecer o domínio de aplicação haverá muito pouco para reuso [20].

Reutilização é de fato um dos paradigmas fundamentais de desenvolvimento [2]. Ainda que os problemas existentes sejam de grande amplitude, sua aplicação na engenharia de software passa pela definição de um mecanismo mais adequado e eficiente. Técnicas e ferramentas já conhecidas, tais como hipertexto e editores orientados por sintaxe, podem ajudar a compor um ambiente que concretize em definitivo o reuso de software.

# 3 - A SOLUÇÃO PROPOSTA

A análise das tecnologias de reuso de software à luz do dilema "generalidade X benefício" (figura 2.2) sugere que, a menos do aparecimento de um novo mecanismo com novas idéias, o caminho para superar o dilema é trabalhar as tecnologias existentes de maneira a vencer as suas limitações. É neste sentido que uma biblioteca de componentes de código pode ser vista como melhoramento do mecanismo representado por linguagens de alto nível. Neste caso, operou-se o conceito de modularização para ganhar em benefício, muito proporcionalmente ao preço de perder em generalidade.

Dentre as tecnologias conhecidas para a reutilização de software, o gerador de aplicação destaca-se pelo grande benefício proporcionado. Contudo, esta característica altamente desejável é mascarada pela baixa generalidade do mecanismo, dado o estreito domínio de aplicação ao qual estão relacionados os seus padrões de arquitetura. Eis aqui a essência do dilema: é exatamente a limitação que proporciona a vantagem. Como aumentar a generalidade e manter o benefício, se não há reusabilidade suficiente sem domínio de aplicação ?

Prieto-Diaz [27] entende que é pela análise de domínio que se derivam as arquiteturas comuns e modelos genéricos que substancialmente influenciam o processo de desenvolvimento de software em uma área específica de problemas. Estes modelos encontram-se embutidos no projeto de um gerador de aplicação, e os programas são produzidos sob medida para atenderem a um conjunto exclusivo de requisitos [16].

A flexibilização dos padrões de arquitetura usados por um gerador é a chave para aumentar a generalidade do mecanismo. Este objetivo pode ser alcançado com a possibilidade do usuário final escolher os padrões dentre um cardápio de alternativas

previamente estabelecidas. O gerador não mais se limitaria a um estreito domínio de aplicação, e seria mantido o grande benefício proporcionado.

Estas considerações são o fundamento da proposta contida no presente trabalho. A idéia básica é conjugar tecnologias dos grupos de geração e composição. Mais especificamente, um gerador de aplicação que utilize padrões de arquitetura como componentes armazenados em uma biblioteca. Os componentes são templates, ou abstrações de código [15], a serem instanciados para cada aplicação em particular. A função do gerador é interpretar os templates especializando-os com os parâmetros fornecidos pelo usuário.

Certamente a solução proposta consegue ampliar o potencial de reuso do gerador de aplicação mas, por outro lado, adiciona alguns dos problemas inerentes às tecnologias de composição. Isto significa que haverá alguma perda no benefício proporcionado, ou seja, será requisitado um esforço maior do usuário para geração do software. Contudo, este esforço limita-se à seleção e recuperação do componente, pois o problema da adaptação praticamente inexiste, dado a especialização automática realizada pelo gerador.

Conclui-se pela necessidade de um ambiente completo, incluindo o gerador, uma biblioteca de templates, mais técnicas e ferramentas adicionais de auxílio ao usuário.

## 3.1 - O Uso de Templates

O significado do termo inglês "template" na Engenharia de Software é difícil de ser estabelecido com precisão, dado o seu uso indiscriminado para representar idéias ou objetos muitas vezes diferentes. De maneira geral, esta palavra encontra tradução no português em sinônimos como gabarito, molde, padrão, esboço de forma, etc. [8]. Na verdade, todas estas palavras aplicam-se para definir a semântica exata do termo template no escopo do presente trabalho, porém nenhuma delas é suficiente.

Na Engenharia de Software, template pode significar uma idéia desde a mais simples até a mais sofisticada, e não necessariamente restrita ao contexto da reutilização.

Um exemplo bastante simples é encontrado no trabalho de Wartik e Penedo [40] que descreve o sistema Fillin, parte integrante do ambiente de desenvolvimento de software SPS (Software Productivity System). Trata-se de uma ferramenta de software que fornece aos usuários, e outras ferramentas do ambiente, uma interface orientada para formas. Uma forma é percebida pelo usuário como sendo uma coleção linear de dados combinados com informações de formatação que determinam o seu "layout". O propósito do Fillin é apresentar esta visão dos dados ao usuário com uma interface contendo comandos para criação, edição e exibição de uma forma.

A ferramenta Fillin baseia-se em um esquema de parametrização que aplica a idéia de templates. Através de uma lingem define-se um template com o "layout" e conteúdo de uma forma. Uma outra linguagem descreve o mapeamento entre a forma e seus dados. Um template compõe-se de um cabeçalho e algumas linhas codificadas. O cabeçalho é uma informação textual que aparece como título no topo da forma. Cada linha codificada

consiste de um ou mais campos que definem os conteúdos da forma. Cada campo pode estar associado a um rótulo ou a um dado. Campos de dados correspondem a locais onde os dados do usuário podem ser armazenados e modificados.

A figura 3.1a apresenta o template que corresponde à forma da figura 3.1b.

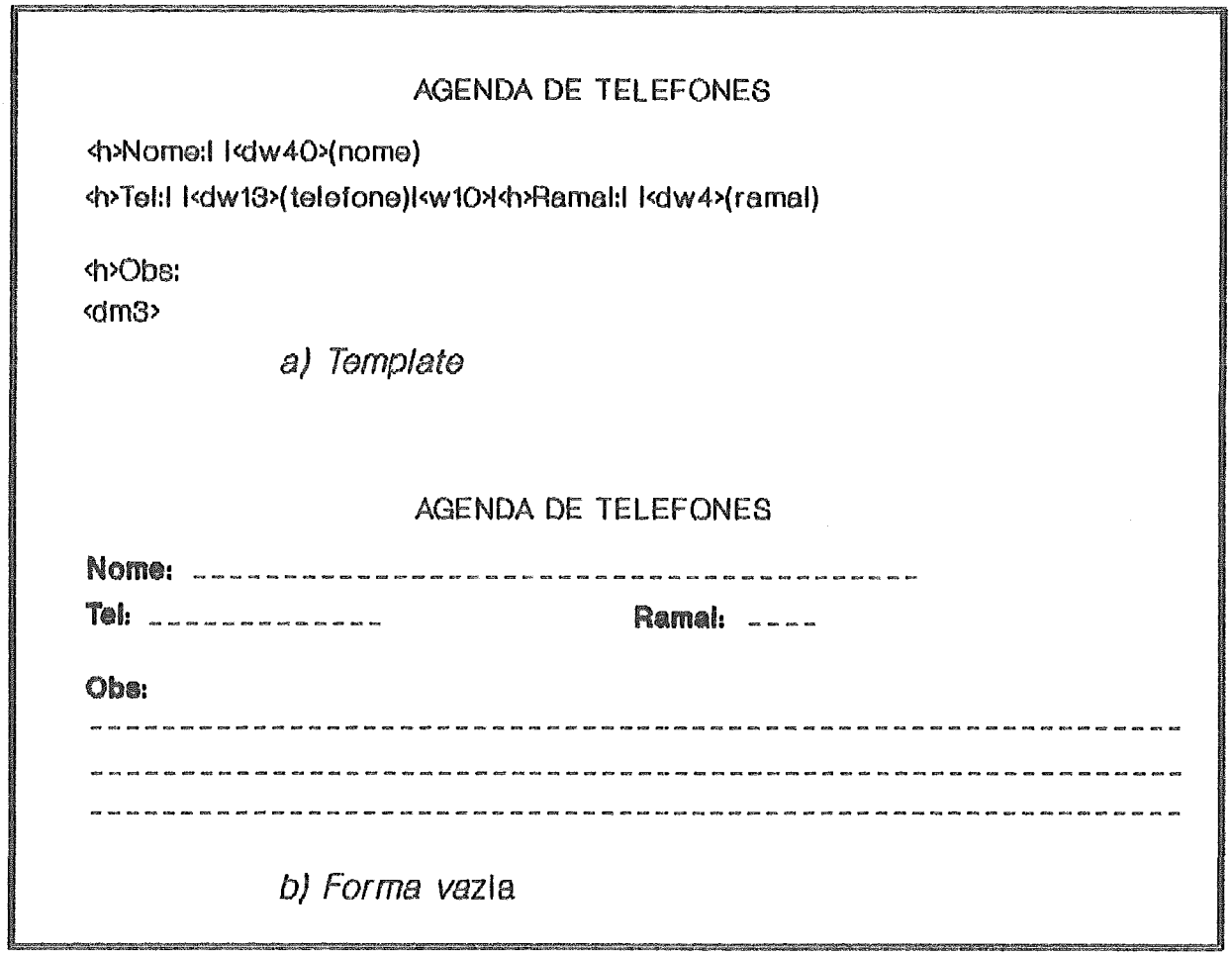



Fig. 3.1 - Uso de templates no sistema Fillin

No template da figura 3.1a, o texto *Agenda de Telefones* corresponde ao cabeçalho que pode ocupar as duas primeiras linhas da forma. O restante do template consiste dos campos, separados por barras verticais (a primeira linha codificada contém 3 campos, a segunda contém 7, ...). Um campo pode ser

formado de três partes, todas elas opcionais: atributos, identificador e dados. O conjunto de atributos de um campo aparece entre os símbolos menor e maior (< e >). No exemplo, o campo *<h>Nome:* tem o atributo *h* que especifica a exibição do rótulo *Nome:* em maior intensidade ("highlight"). O próximo campo é um espaço, usado para se obter um branco antes de um campo de dados. A última área da mesma linha é definida pelo campo *<dw40>(nome)*. O atributo *d* especifica que trata-se de um campo de dados, ou seja, um local onde o usuário entra com um texto. O atributo *w40* significa que o campo é de 40 caracteres, ou seja, ocupa até um máximo de 40 espaços na tela. O identificador *nome* entre parênteses é um nome simbólico para referência ao valor do campo. Finalmente, no exemplo apresentado, a última linha no template contém o atributo *m3* significando tratar-se de um campo multilinha. Este campo permite a entrada de um texto de tamanho arbitrário, porém a área de exbição terá o tamanho de fixo de 3 linhas.

Embora seja bastante simples, o exemplo de template utilizado no sistema Fillin permite parametrizar definições de formatação para entrada de dados. Isto viabiliza o objetivo desta ferramenta em oferecer uma interface única e consistente ao usuário, promovendo a reutilização de software e integração de ferramentas no ambiente SPS.

Outro exemplo de igual simplicidade, porém agora no contexto da reutilização, é encontrado no modelo proposto por Jameson [12] para o reuso de informação de projeto de software. O objetivo é viabilizar a construção da arquitetura compilável de um programa a partir de documentos de projeto independentes de linguagem. Tal arquitetura é composta de fragmentos de módulos documentados que apresentam relacionamentos de hierarquia, parâmetros, descrições funcionais e características algorítmicas da estrutura. Entretanto, nenhum código executável é incluído nos módulos compiláveis. A arquitetura do programa é obtida combinando-se informações de projeto com templates de

módulos padrão dependentes da linguagem alvo. Isto é possível pelo casamento de marcas no arquivo de projetos com marcas idênticas no arquivo de templates, e respectiva transcrição das informações associadas.

A seguir apresenta-se como exemplo do modelo de Jameson um template simplificado para uma "function" compilável em (Turbo) Pascal.

```
- A primeira parte de um arquivo de templates mapeia tipos de
  dados simbólicos em tipos atuais da linguagem.
- As sequências de caracteres %1 são substituídas pelo nome do
  identificador contido no arquivo de projetos.

SINT8       %1 : byte;
SINT16      %1 : integer;
STRING32    %1 : array [32] of char;
...
- A segunda parte do arquivo contém templates específicos da
  linguagem para entidades simbólicas no arquivo de projetos.
-
(* MODULO função <nome_lógico_do_módulo> *)
function <nome_interno>   (* <breve descrição funcional> *)
   (
(* FUNÇÃO
FIM FUNÇÃO *)
...
(* PARÂMETROS *)
   ) : <return type>;          (* <descrição do retorno> *)
(* FIM PARÂMETROS *)
...
begin
(* ALGORITMO *)
(* FIM ALGORITMO *)
end;

(* FIM_MODULO função <nome_lógico_ do_módulo> *)
```

Este template pode ser usado em combinação com as seguintes informações de projeto para geração de um módulo que determina o maior dentre dois valores numéricos.

```
MODULO função  Max

FUNÇÃO - retorna o máximo
A função Max determina o maior valor dentre dois números
inteiros fornecidos como parâmetros.
FIM FUNÇÃO

PARÂMETROS
arg1    SINT16          - u: o primeiro número
arg2    SINT16          - u: o segundo número
return  SINT16          - o: o valor máximo dos dois
FIM PARÂMETROS

ALGORITMO
: se primeiro argumento é )= segundo argumento
:   retorne o primeiro argumento
: senão
:   retorne o segundo argumento
FIM ALGORITMO

FIM_MODULO função Max
```

O processo de obtenção da arquitetura de módulos utiliza ainda informações sobre nomes internos e externos dependentes da linguagem e sistema operacional para todos os módulos definidos no projeto. Tais informações constituem um terceiro arquivo ("group file"), mostrado a seguir para o exemplo dado.

```
MODULO Max Interno_Max          - liga nomes lógico e interno

GRUPO GrupoMax FUNÇÃO ArqMax
   Max                          - nomes de módulos no grupo...
FIM GRUPO
```

Finalmente, combinando as marcas e transcrevendo os textos associados deriva-se o seguinte fragmento de módulo compilável em Turbo Pascal (arquivo ArqMax.pas).

```
(* MODULO função Max *)
function Interno_Max                    (* retorna máximo *)
   (
(* FUNÇÃO
A função Max determina o maior valor dentre dois números
inteiros fornecidos como parâmetros.
FIM FUNÇÃO *)

(* PARÂMETROS *)
    arg1 : integer;         (* u: o primeiro número        *)
    arg2 : integer          (* u: o segundo número         *)
    ) : integer;            (* o: o valor máximo dos dois *)
(* FIM PARÂMETROS *)

begin
(* ALGORITMO *)
(* : se primeiro argumento é >= segundo argumento *)
(* :   retorne o primeiro argumento               *)
(* : senão                                        *)
(* :   retorne o segundo argumento                *)
end;

(* FIM_MODULO função Max *)
```

No modelo de Jameson é a idéia associada à palavra template que conduz à independência de linguagem, na medida em que viabiliza parametrizar a sintaxe da arquitetura gerada.

Em muitos outros casos a palavra template aplica-se a idéias bem mais sofisticadas do que nos exemplos anteriores. Template pode significar um componente de software onde os tipos e representações de dados não são especificados, e o algoritmo pode estar fragmentado em partes relacionadas através de esquemas de código e projeto. Tal componente pode ser instanciado para diferentes aplicações com algoritmo e tipos de dados específicos a partir das informações fornecidas pelo usuário. Com significado semelhante, muitas vezes template é chamado de "schema" na reutilização de software, "frame" em inteligência artificial, ou "cliché" em programação automática. De fato, Tracz [35] relata que "um template captura a estrutura de controle básica, ou arquitetura, de um componente de software e deixa buracos para serem preenchidos antes do seu

uso". Rich e Waters [28] definem um clichê como tendo três partes: um esqueleto que está presente em todas as ocorrências do chichê, partes cujos conteúdos variam de uma ocorrência para outra, e restrições sobre o que pode preencher estas partes.

Volpano e Kieburtz [39] discutem em seu trabalho uma interessante abordagem baseada em templates para o reuso de software. Eles descrevem um método que divide a informação de um componente em duas partes: o algoritmo essencial e a implementação dos dados. Os templates de software são algoritmos abstratos definidos sobre tipos abstratos de dados e sem nenhum compromisso com detalhes da implementação. Uma vez o projetista tendo escolhido um template e implementações específicas para os seus tipos abstratos de dados, o sistema gera uma versão personalizada do algoritmo abstrato, sempre produzindo uma expansão de código significante.

Como ilustração, a seguir é apresentado um template, denominado *sort*, que classifica um número genérico de elementos de uma sequência utilizando os tipos abstratos de dados *Seq* (sequência) e *Bintree* (árvore binária).

```
sort(nil) = niltree
sort(cons(x,y)) = insert(x, sort(y))

insert(n, niltree) = leaf(n)
insert(n, leaf(m)) =
                     if (n<m) then node(leaf(n), m, niltree)
                     else node(niltree, m, leaf(n))
insert(n, node(l, m, r)) =
                     if (n<m) then node(insert(n, l), m, r)
                     else node(l, m, insert(n, r))
```

Para usar o template sort o programador especifica uma diretiva de instanciação em um programa que transfere as implementações relacionadas para os tipos abstratos de dados.

Desde que sort é genérico, a diretiva deve especificar também um tipo para os elementos que serão classificados. Por exemplo, supondo que se deseja instanciar sort para uma função em C que classifica um array de strings, e são disponíveis em uma biblioteca as implementações em C de nomes Carray e Ctree para Seq e Btree, a seguinte diretiva produziria o componente desejado:

```
Assign(y, sort x) Where
     x : Seq(String) / Carray(128)
     y : Bintree(String) / Ctree
EndWhere
```

Os tipos das variáveis *x* e *y* são dados na cláusula Where usando o símbolo '/' ("implementada por"). A variável *x* denota uma sequência de strings implementada por Carray, e *y* denota uma árvore binária implementada por Ctree.

Esta idéia mais sofisticada para templates não existe apenas em modelos e propostas a nível teórico. Sistemas como PARIS [14] e IFS [38] comprovam na prática a importância do uso de templates, especialmente no contexto da reutilização.

O sistema PARIS (PARtially Interpreted Schemas) explora a idéia de que muitos programas, enquanto diferentes em seus detalhes, são fundamentalmente o mesmo programa no abstrato. Ele captura esta semelhança como um "schema" de programa parcialmente interpretado. Sintaticamente, um esquema pode ser um programa ou módulo independente em alguma linguagem de programação, mas contém entidades abstratas tais como predicados, símbolos constantes, domínios não especificados, funções abstratas ou trechos de programa não realizados. O sistema mantém uma biblioteca de esquemas parcialmente interpretados, cada um dos quais armazenado juntamente com

asserções sobre sua aplicabilidade e resultados. Quando um usuário apresenta a descrição de requisitos de um programa, o sistema procura por um esquema que satisfaz a especificação e, se um candidato é encontrado, as entidades abstratas podem ser susbstituídas para obtenção do software desejado.

IFS (Interpretive Frame System) é uma ferramenta de auxílio na construção de sistemas a partir da interconexão de tarefas. Cada tarefa é mapeada no sistema por uma "frame" que define todas as atividades requeridas para a sua realização. A nível de projeto e programação a ferramenta IFS permite o reuso de fragmentos de código que descrevem partes estruturais de um sistema. Estes fragmentos sao definidos em macro templates que podem ser ligeiramente modificados para cada aplicação.

Inúmeros outros exemplos de modelos ou sistemas que utilizam a idéia de templates poderiam ser aqui relacionados. Entre um e outro sempre existirão diferenças de estratégias e técnicas utilizadas na concretização desta idéia, muito embora o objetivo comum seja ganhar em generalidade através da parametrização e/ou abstração de entidades concretas usadas na representação de software.

O uso de templates pode aumentar a reusabilidade de componentes de código de software, mas existe um compromisso entre a complexidade do código gerado e a complexidade do template, em termos de justificar o seu uso. Quanto mais complexo o código gerado, mais difícil pode ser o uso do template. O usuário deve perceber que o esforço para localizar e instanciar o template seja menor do que o esforço em recriar o componente de código. No sistema MARTE este compromisso não é um problema pois, qualquer que seja a complexidade do template, a instanciação é realizada de modo automático pelo mecanismo gerador. O sistema minimiza o esforço dispendido na adaptação de um componente de código para viabilizar o seu reuso.

## 3.2 - Especificação do Sistema MARTE

Os fundamentos lançados para a solução proposta neste trabalho são concretizados na ferramenta de desenvolvimento de software denominada MARTE (Meta-gerador de Aplicação baseado na Reutilização de TEmplates).

O processo completo de reuso de software define dois ambientes no sistema MARTE (figura 3.2):

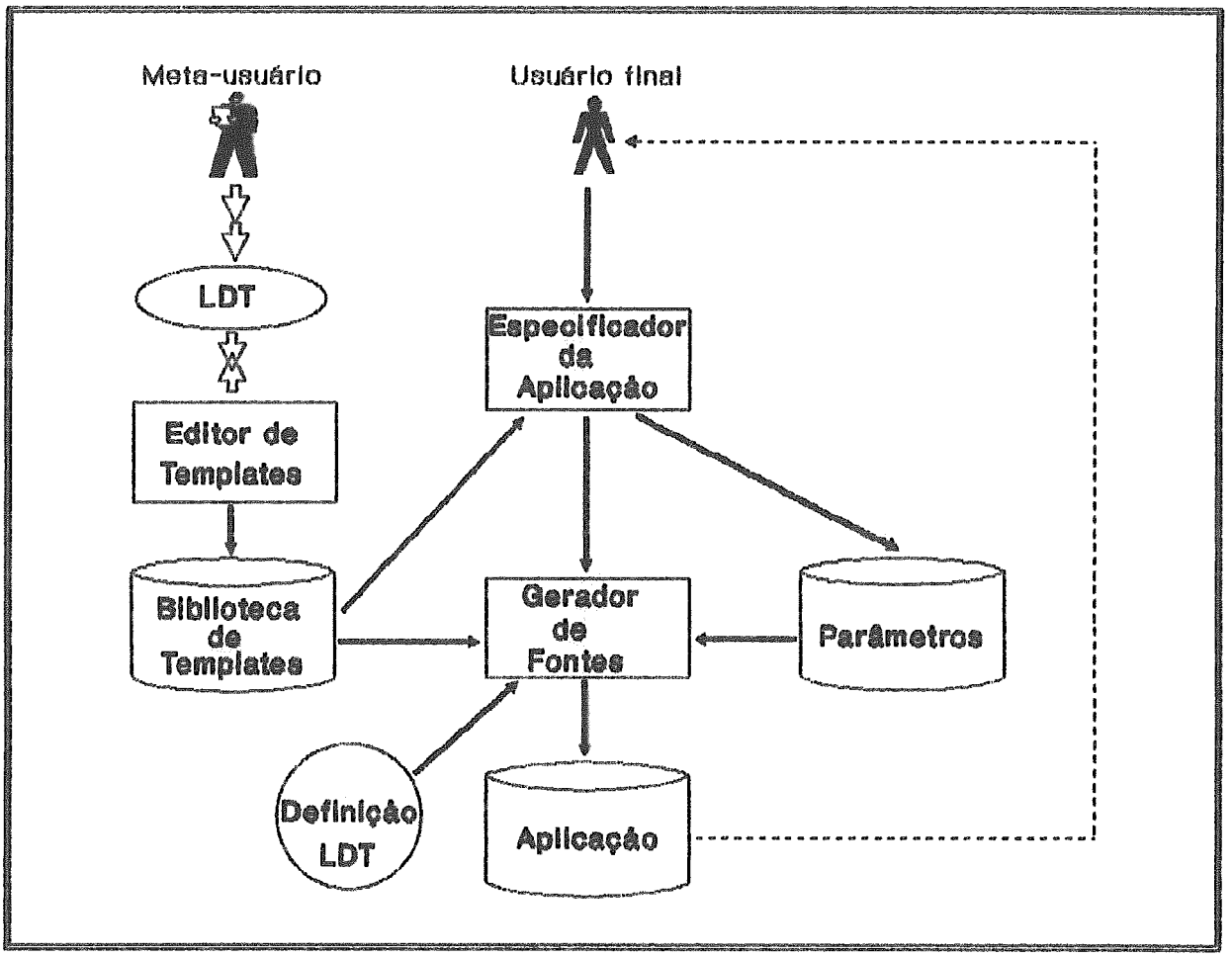



Fig. 3.2 - Visão geral do sistema MARTE

### A) AMBIENTE GERENCIADOR DE TEMPLATES (META-AMBIENTE)

Na tecnologia de composição os objetos de software reusáveis, extraídos de um domínio de aplicação, devem ser

devidamente classificados e catalogados em uma biblioteca. No sistema MARTE estes objetos são os templates, que representam componentes de código abstraídos de seus parâmetros. O meta-usuário descreve o template em um editor especializado através da linguagem LDT (Linguagem de Descrição de Templates).

Um mesmo componente pode ser representado por diversos templates, cada qual associado a uma linguagem de programação diferente. Isto significa que o software gerado é independente não apenas do domínio, mas também da linguagem de programação.

Tipicamente, o meta-usuário é um analista de domínio com especialização nas linguagens utilizadas. Ele deve ser capaz de descrever o template e classificá-lo segundo o domínio, linguagem de programação e demais atributos de seleção.

B) AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE (ADS)

Uma vez classificado e catalogado na biblioteca, o template pode ser reutilizado. O usuário final, de posse dos requisitos de sua aplicação, seleciona o template desejado e introduz os parâmetros necessários. O módulo Especificador de Aplicação ativa o Gerador de Fontes, que produz o código fonte correspondente ao template.

Neste ambiente encontra-se ainda um Editor de Telas de Help que permite a montagem de um hipertexto de auxílio ao usuário da aplicação. O código fonte gerado mais as telas de help constituem o produto de software.

Um diagrama de fluxo de dados completo, construído segundo a notação De Marco [6], apresenta os principais processos em cada ambiente do sistema MARTE (figura 3.3).

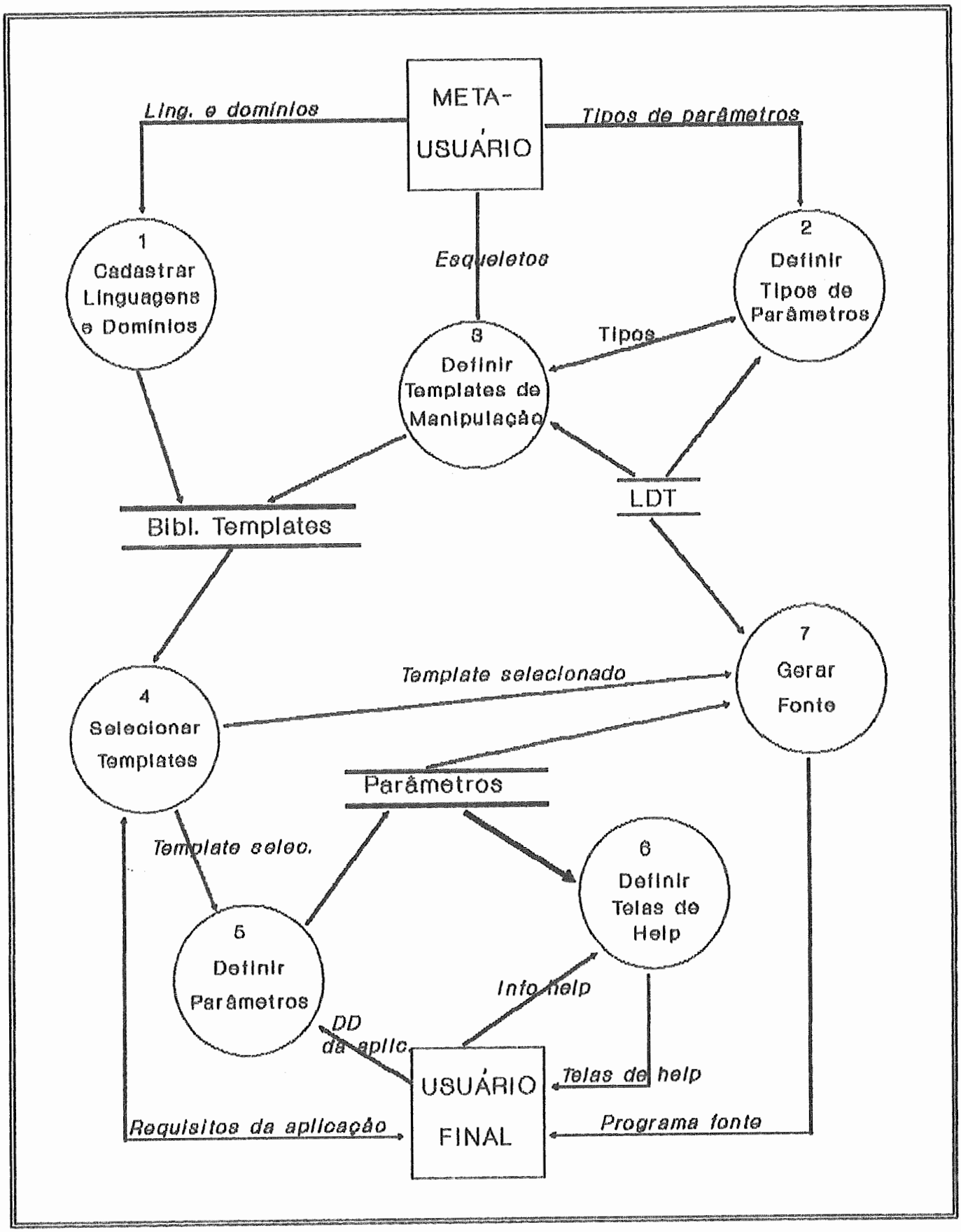



Fig. 3.3 - DFD completo do sistema MARTE

### 3.2.1 - Linguagem de Descrição de Templates (LDT)

O template é o elemento básico para composição de software no sistema MARTE. Ele consiste de uma simples abstração de código onde os parâmetros são representados por esquemas que permitem a instanciação posterior.

A descrição de um template inicia-se com a definição dos tipos de parâmetros que serão manipulados, bem como a estrutura de cada um desses diferentes tipos (nome, atributos, nível de aninhamento dos atributos, etc.). A segunda parte do template é um esqueleto de código que manipula os parâmetros e seus atributos conforme a definição prévia de tipos.

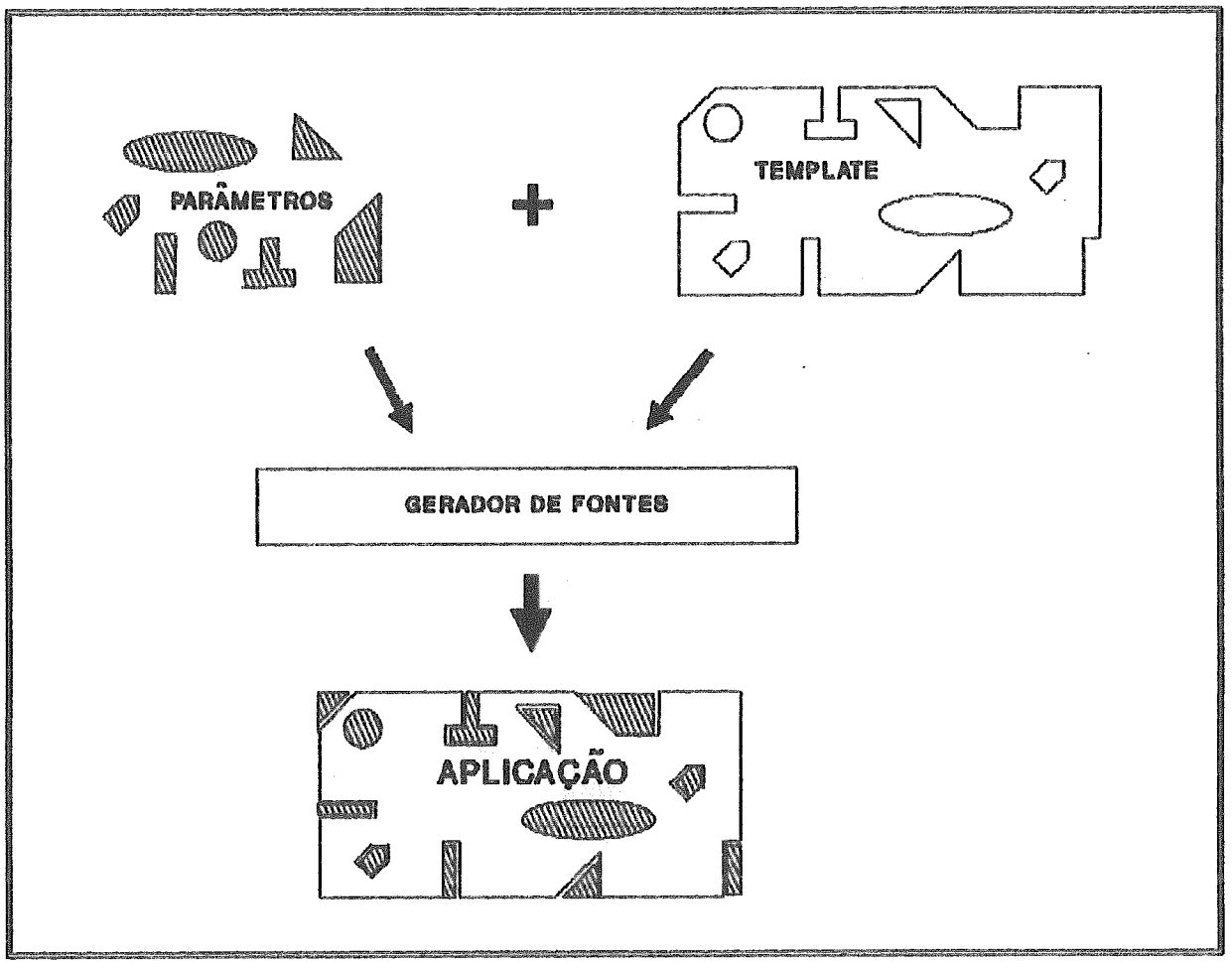



Fig. 3.4 - Esquema de geração do código fonte

No ambiente de desenvolvimento de software, o usuário final deve fornecer os parâmetros conforme a sintaxe e semântica das regras descritas pelo meta-usuário (definição de tipos) quando da criação do template. O gerador de fontes produz o software de aplicação combinando o template com os parâmetros (figura 3.4).

Neste processo, o fonte gerado corresponde efetivamente ao esqueleto de código com os parâmetros e atributos prontamente substituídos.

A linguagem LDT oferece um pequeno, mas suficiente, número de primitivas que podem ser combinadas para a obtenção de construções mais complexas. Ela é formada de três núcleos principais cujas características são descritas a seguir.

### 3.2.1.1 - Definição de Tipos de Parâmetros

Um tipo de parâmetro pode ou não necessitar de atributos que completem sua especificação. Esses atributos, por sua vez, podem ou não necessitar de novos atributos para especificá-los totalmente. Esse aninhamento pode ocorrer em tantos níveis quantos forem necessários.

O exemplo apresentado graficamente na figura 3.5 pode ser explicado como segue:

- a aplicação parametrizada possui 3 tipos ou classes principais de parâmetros: $TP_1$, $TP_2$ e $TP_3$. A classe principal $TP_2$ não necessita de maiores

detalhes para sua completa especificação. Contudo, a classe principal $TP_1$ necessita de dois atributos, um pertencente à classe secundária $TP_{11}$ e outro à classe secundária $TP_{12}$ ou $TP_{13}$, ou seja, os nós irmãos envolvidos por um retângulo pontilhado são considerados nós alternativos. A interpretação do restante da figura é similar aos três casos analisados.

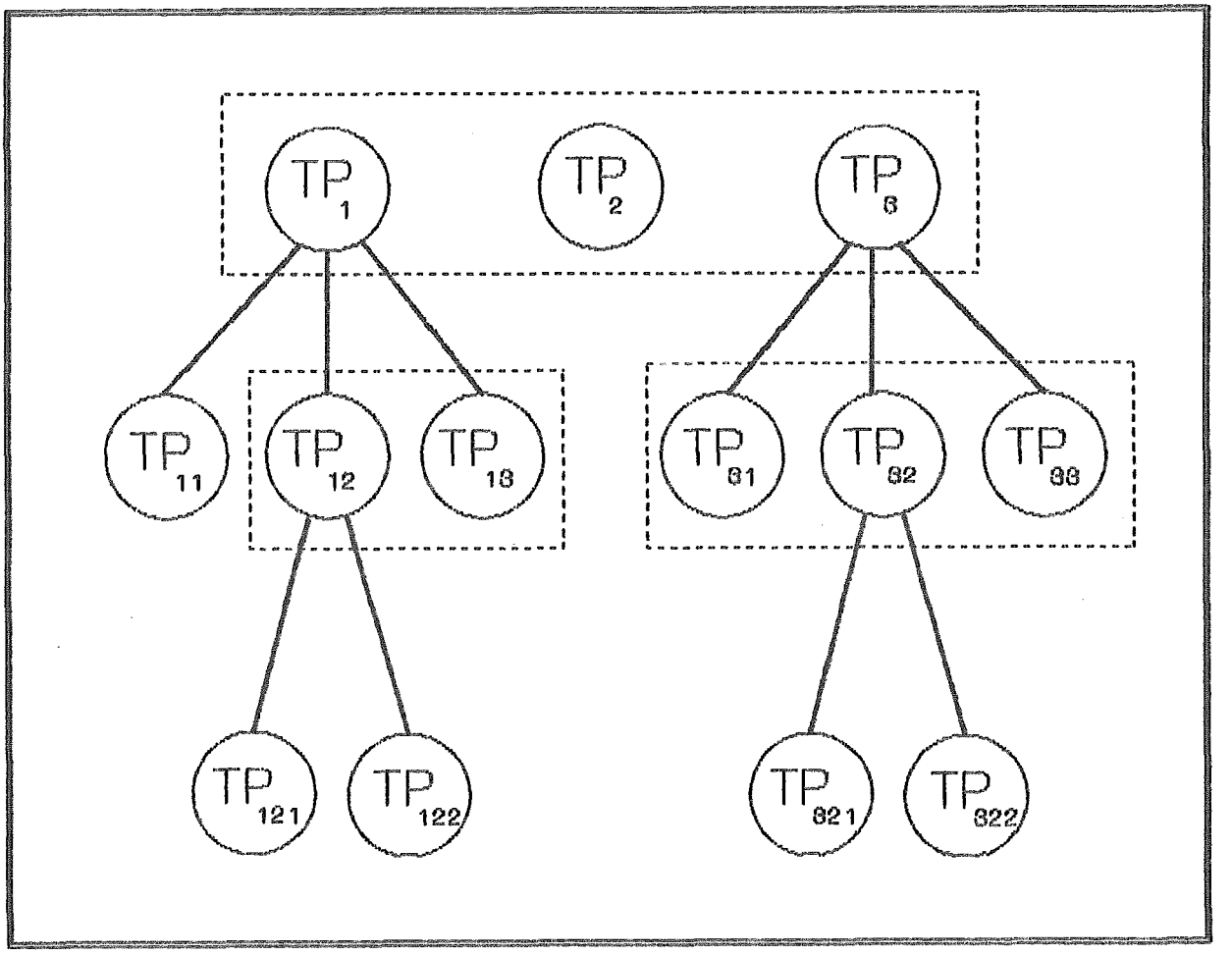



Fig. 3.5 - Exemplo de definição de tipos

Por uma questão de coerência e comodidade de programação, a linguagem exige que na definição de um novo tipo de parâmetro em função de outros tipos, estes já tenham sido declarados previamente (por exemplo, os pares $TP_{12}$-$TP_1$ e $TP_{32}$-$TP_3$ na figura 3.5). Desta forma, a definição de tipos é feita no

sentido "bottom-up", ou seja, a criação de cada árvore de definição é sempre das folhas para a raiz.

A declaração da estrutura de tipos mostrada na figura 3.5 poderia ser a que segue, onde as palavras grifadas são palavras reservadas da linguagem, juntamente com os símbolos especiais.

```
defTipos

  classeAS                    ( classes Alternativas e Secundárias )
    TP11;
    TP12;
    TP1 exige  (classe TP11, TP12);
    TP10;
    classeA1 = (TP1, TP10);                  ( classe alternativa )
    TP21;
    TP221;
    TP222;
    TP22 exige  (classe TP221, TP222);
    TP23;
    TP20;
    classeA2 = (TP21, TP22, TP20);           ( classe alternativa )

  classeP                                    ( classes Principais )
    TP1 exige (classe TP11, classeA1);
    TP2;
    TP3 exige (classe classeA2)
```

Uma outra possibilidade para a declaração dos tipos da figura 3.5 seria:

```
defTipos

  classeAS                    ( classes Alternativas e Secundárias )
    TP12 exige  (classe TP121, TP122);
    classeA1 = (TP12, TP10);                 ( classe alternativa )
    TP22 exige  (classe TP221, TP222);
    classeA2 = (TP21, TP22, TP20);           ( classe alternativa )

  classeP                                    ( classes Principais )
    TP1 exige (classe TP11, classeA1);
    TP2;
    TP3 exige (classe classeA2)
```

Neste caso, as classes terminais (folhas), que não necessitarem de atributos complementares, não precisam ser declaradas explicitamente, exceto no caso de ser uma classe principal.

Até este ponto, os únicos atributos que podem ser associados com os parâmetros são atributos que determinam hierarquia (pertencer às classes x, y, ...). Isto é útil mas não satisfaz a todas as necessidades. é preciso haver uma forma de associar valores aos parâmetros.

Seja novamente o exemplo da figura 3.5, e o fato de um parâmetro pertencer à classe $TP_{1.1}$ não ser totalmente significativo, a menos que sejam fornecidos dois valores adicionais. Esta nova situação é apresentada graficamente na figura 3.6, onde um valor associado a parâmetro (VAP) é representado por um nó terminal (folha) na forma de um quadrado.

Neste caso, não existe a possibilidade de valores alternativos.

A nova declaração para a árvore de raiz $TP_1$ seria:

```
defTipos

  classeAS
    TP1.1 exige ( vap V1, V2 ):
    TP1.2 exige ( classe TP1.2.1, TP1.2.2 ):
    classeA1 = ( TP1.2, TP1.3 );

  classeP
    TP1 exige  ( classe TP1.1, classeA1 );
```

O valor associado a um parâmetro poderia ser de qualquer tipo pré-definido: real, inteiro, caractere, cadeia de caracteres, lógico, etc. Porém, conforme definição da linguagem LDT só existe a possibilidade do tipo cadeia de caracteres, dada sua amplitude de uso e facilidade de implementação.

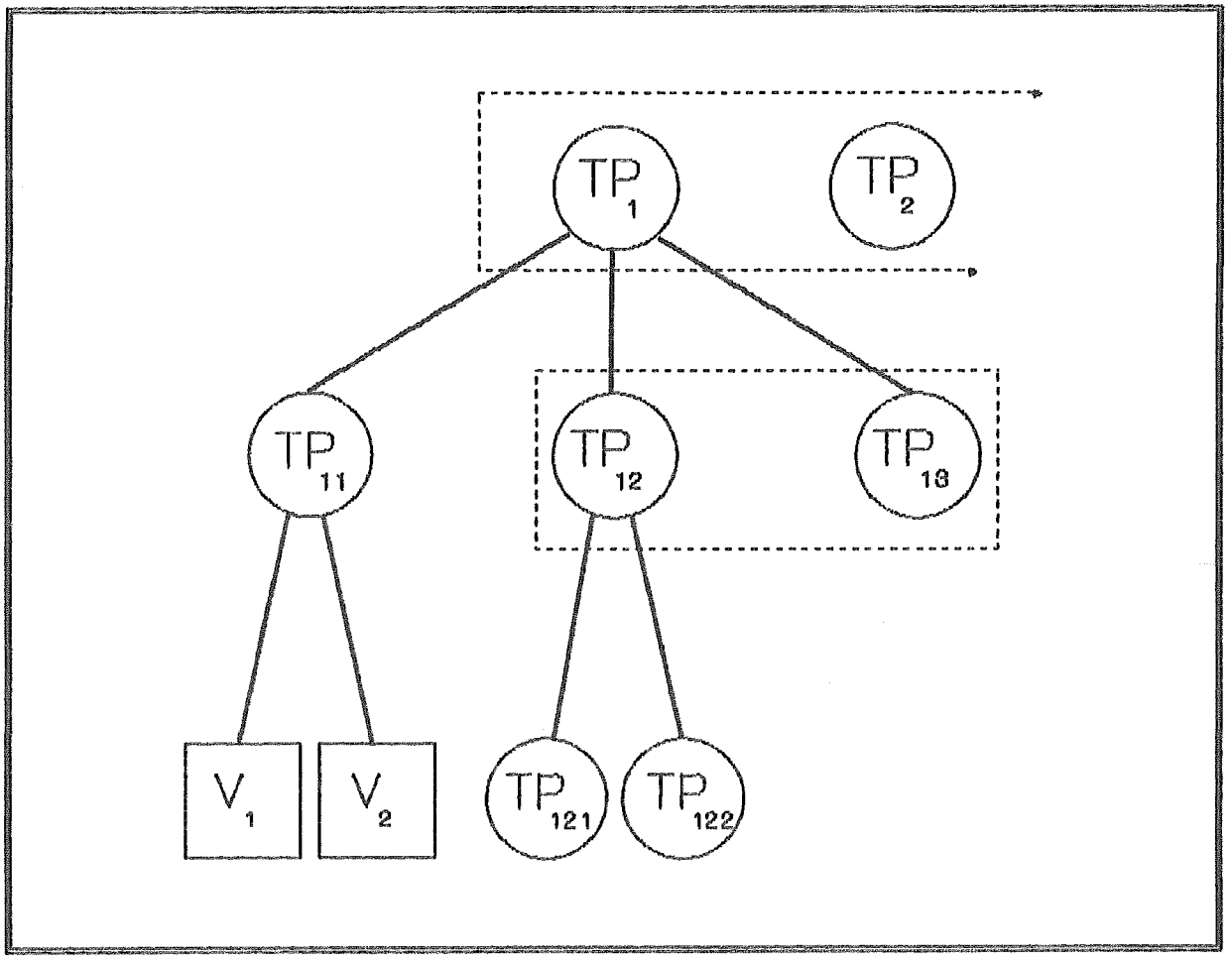



Fig. 3.6 - Valores Associados a Parâmetros

Completando este núcleo da linguagem, ainda existe a possibilidade do tipo de parâmetro constante, ao qual está associado um valor. Por exemplo:

```
constante
  NomeDoSistema = 'MARTE';
  PrefixoArqDados = 'Dat';
  PrefixoArqIndice = 'Idx';
```

### 3.2.1.2 - Definição de Parâmetros

O usuário final fornece os parâmetros que serão combinados com o template por ele selecionado. Estes parâmetros deverão compor estruturas de dados que estarão amarradas ao projeto e arquitetura do gerador. Se durante a geração do código estas estruturas fossem utilizadas, o gerador não seria independente da aplicação. Para solucionar este problema adotou-se uma estrutura de dados do tipo texto como única interface usuário-gerador e template-gerador, sendo os parâmetros fornecidos juntamente com a definição de tipos e esqueletos de código.

Obviamente, o usuário final fornece os parâmetros através de um módulo de diálogo extremamente amigável e flexível. O sistema se encarrega de codificar as informações do usuário na forma de um texto segundo as regras da LDT.

Este núcleo da linguagem permite a representação dos parâmetros específicos a serem utilizados na manipulação do template. Todo parâmetro tem um nome próprio e pertence, necessariamente, a alguma das classes principais. O fato de pertencer a alguma classe implica na obrigatoriedade de fornecimento dos atributos exigidos pela mesma, quando for o caso.

Seja o exemplo de definição de tipos da figura 3.6 seguida de uma definição de parâmetros:

---

```
defTipos
  classeA5
    TP11 exige ( var V1, V2 );
    TP12 exige ( classe TP121, TP122 );
    classeA1 = ( TP12, TP10 );
    TP02 exige ( classe TP021, TP022 );
    classeA2 = ( TP01, TP02, TP00 );
```

```
classeP
   TP1 exige classe TP11, classeA1 );
   TP2;
   TP3 exige ( classe classeA2 )

parâmetros
   'NomeP1' : TP1 ( TP11 ('abc', 'xyz'), TP13 );
   'NomeP2' : TP2;
   'NomeP3' : TP1 ( TP11 ('aaa', 'bbb'), TP13 (TP131, TP132) );
   'NomeP4' : TP3 ( TP31 );
   'NomeP5' : TP3 ( TP32 (TP321, TP322) );
   'NomeP6' : TP3 ( TP33 )
```

---

O trecho que representa a definição de parâmetros pode ser interpretado da seguinte forma:

- Foram declarados seis parâmetros: dois na classe principal $TP_1$, um na $TP_2$ e três na $TP_3$. Os identificadores NomeP1, NomeP2,..., NomeP6 são os nomes próprios dos parâmetros.

- O parâmetro NomeP1 possui os seguintes atributos: pertence às classes $TP_1$, $TP_{11}$ e $TP_{13}$, possui o valor associado V1 igual a 'abc', e o valor associado V2 igual a 'xyz'.

- O parâmetro NomeP2 possui apenas o seguinte atributo: pertence à classe $TP_2$ (supõe-se que apenas este atributo seja suficiente para caracterizá-lo).

- O parâmetro NomeP3 possui os seguintes atributos: pertence às classes $TP_1$, $TP_{11}$, $TP_{13}$, $TP_{131}$ e $TP_{132}$, possui o valor associado V1 igual a 'aaa' e o valor associado V2 igual a 'bbb'.

Observa-se que $TP_{4.1}$, embora seja uma classe válida, é desnecessária, pois pertencer à classe $TP_4$ implica em pertencer à classe $TP_{4.1}$, e vice-versa. Este não seria o caso se existisse, por exemplo, uma classe principal assim definida:

$TP_4$ exige ( classe $TP_{4.1}$ )

Isto significa que, no contexto da utilização, o relacionamento entre os tipos de parâmetros não é hierárquico como transparece na figura 3.5. Este relacionamento seria melhor representado na forma de uma rede. A limitação existente é que nenhuma das classes principais seja utilizada por outra.

### 3.2.1.3 - Manipulação de Parâmetros e Tipos de Parâmetros

Este núcleo fornece suporte para que, de dentro do template, o usuário saiba quais tipos de parâmetros foram fornecidos, bem como meios para recuperar seus atributos. Na definição da linguagem LDT este núcleo é bastante primitivo, apresentando o mínimo de comandos necessários.

A instrução mais simples é o comando de saída *escreva*, que permite escrever uma cadeia de caracteres no arquivo de saída. Segue um exemplo.

```
defTipos
  constante
    NomeDoSistema = 'MARTE';
esqueleto
  escreva ('O nome do sistema é ', NomeDoSistema, '.')
fimEsqueleto
```

A saída seria:

O nome do sistema é MARTE.

Uma estrutura de controle condicional simples é implementada pelo comando *se-então-fimSe*. As instruções contidas na estrutura serão executadas caso o resultado da avaliação da expressão lógica seja verdadeiro; caso contrário não serão consideradas.

Uma estrutura de controle repetitiva é implementada pelo comando *para-faça-fimPara*, que permite a navegação por todos os parâmetros que pertençam a uma determinada classe, principal ou secundária. A navegação é feita na mesma ordem em que foram declarados os parâmetros. Segue um exemplo.

```
defTipos
  classeAS
    Sexo = (Masc, Fem);
  classeP
    Pessoa exige  ( classe Sexo; vap Idade );

parâmetros
  'Junia': Pessoa ( Fem,  '4'  );
  'Tiago': Pessoa ( Masc, '25' );
  'Pedro': Pessoa ( Masc, '9'  );
  'Maria': Pessoa ( Fem,  '44' );
  'Jose' : Pessoa ( Masc, '17' );

esqueleto
  escreva ( 'Mulheres : ');
  para  Fem  faça
    escreva ( parâmetroAtual (Fem) )
  fimPara;
  escreva ( 'Pessoas com 25 anos : ');
  para Pessoa faça
    se  vap (Idade, Pessoa) = '25' então
      escreva ( parâmetroAtual(Pessoa) )
    fimSe
  fimPara
fimEsqueleto
```

A saída seria:

```
Mulheres :
 Junia
 Maria

Pessoas com 25 anos :
 Tiago
```

A função parâmetroAtual (classe) retorna o identificador do parâmetro ativo. Assim, a semântica do primeiro comando para-faça-fimPara do exemplo seria a seguinte:

(1) Selecione o primeiro parâmetro associado com a classe Fem;

(2) Se foi possível executar o passo (1), faça o parâmetro selecionado ser o parâmetro ativo da classe Fem;

(2a) Execute os comandos dentro da estrutura de repetição;

(2b) Selecione o próximo parâmetro que tem associação com a classe Fem;

(2c) Se foi possível executar o passo (2b), faça o parâmetro selecionado ser o parâmetro ativo da classe Fem;

(2c1) retornar ao passo (2a).

Finalmente, a função vap (NomeVap, NomeClasse) retorna o valor indicado pelo primeiro argumento (NomeVap) e associado ao parâmetro ativo da classe indicada pelo segundo argumento (NomeClasse). O parâmetro ativo é controlado pelo comando de repetição.

### 3.2.1.4 - Exemplo Completo

Apresenta-se um exemplo simples da declaração de um registro na linguagem (Turbo) Pascal. Os parâmetros a serem fornecidos são o nome e o tipo de cada um de seus campos. O nome do registro será considerado uma constante na definição.

a) Definição dos tipos de parâmetros:

```
defTipos
  constante
    NomeReg = 'Exemplo1';
  classeP
    Campo exige ( var TipoDoCampo )
```

b) Definição dos parâmetros:

```
parâmetros
   'Matricula' : Campo ('integer');
   'NotaFinal' : Campo ('real');
   'Conceito'  : Campo ('char');
```

c) Manipulação:

```
esqueleto
  escreva (NomeReg, ' = record');
  para Campo faca
    escreva ('    ', parâmetroAtual (Campo), ' : ',
                     var (TipoDoCampo, Campo), ';' )
  fimPara;
  escreva ('end;');
fimEsqueleto
```

d) Fonte gerado:

```
Exemplo1 = record
  Matricula : integer;
  NotaFinal : real;
  Conceito  : char;
end;
```

É importante ressaltar que a instanciação de um template não consiste na substituição estática de seus parâmetros. Pode haver uma extensiva geração de código e consequente benefício pela economia de esforços proporcionada. Isto pode ser verificado no exemplo anterior, ainda que existam situações onde o volume de código gerado é bem mais significativo.

Outro aspecto de grande relevância é a possibilidade de monitorar a entrada de parâmetros a partir da definição de tipos. Isto é fundamental para viabilizar a geração do software independente da linguagem de programação, visto que não deve ser exigido do usuário final conhecer a sintaxe da linguagem alvo. Além disto, o usuário não tem de saber quais os parâmetros são necessários para instanciar o template.

O exemplo completo apresentado anteriormente solicita do usuário o identificador de cada campo do registro e seu respectivo tipo na linguagem de programação. Contudo, se existir a declaração de uma "classe alternativa" com a relação de tipos dispossíveis na linguagem alvo, o processo de entrada poderia ser completamente dirigido através das definições de tipos. Assim, as seguintes alterações viabilizariam o esquema:

```
defTipos
  constante
    NomeReg = 'Exemplo1'
  classeAS
    TipoDoCampo = (Integer, Real, Char, String)
  classeP
    Campo exige ( classe TipoDoCampo )
```

Embora o exemplo seja simples, e tenha considerado apenas os tipos básicos da linguagem Pascal, a representação é possível para todos os casos em que esteja envolvido um conjunto escalar de valores.

O grafo sintático completo da Linguagem de Descrição de Templates (LDT) é apresentado no Apêndice A.

### 3.2.2 - A Biblioteca de Templates

A parametrização dos padrões do gerador conduz a uma independência do código gerado com relação ao domínio e linguagem de programação. Sendo assim, um template é classificado quanto a estes ítens, além de outros critérios que possam caracterizá-lo em maiores detalhes.

A característica fundamental da biblioteca decorre da possibilidade da descrição de um template a partir de outros que já estejam definidos e catalogados. Isto permite aplicar o conceito de reutilização dentro da própria ferramenta, viabilizando o desenvolvimento progressivo e incremental de programas. Outra consequência relevante é a representação natural do histórico de refinamento do template, ou seja, o sistema "guarda o raciocínio" utilizado na elaboração do software representado pelo template. Dentre inúmeras vantagens, isto facilita o entendimento do componente.

O processo construtuivo da biblioteca ("bottom-up") permite ainda que subsistemas por inteiro sejam representados na forma de templates, sendo assim candidatos à reutilização. Isto viabiliza a construção de grandes sistemas pelo uso de grandes subsistemas como parte do processo de raciocínio [20]. Por outro lado, o dilema "tamanho do componente X potencial de reuso" é minimizado pela possibilidade da configuração de componentes de maior tamanho atendendo às especificações do usuário.

O Apêndice D apresenta um pequeno exemplo da definição de um template segundo este processo.

Para viabilizar o esquema, os templates constituem uma estrutura de dados na forma de hipertexto, onde cada nodo

corresponde a um template, que pode estar sendo referenciado por qualquer outro nodo. As ligações são representadas pelos identificadores dos templates, que são inseridos na descrição por meio de uma sinalização apropriada.

De fato, um sistema de hipertexto é ideal para modelar um o editor de templates pela sua capacidade bidimensional de navegação [21], ou seja, uma dimensão **linear** usada no movimento para frente e para trás ao longo de páginas dentro de um mesmo documento, e uma dimensão **não-linear** que através de ligações de hipertexto permite um movimento bidirecional sobre diferentes documentos. Neste esquema, o documento é um template, ou seja, um texto escrito em LDT que corresponde a um componente de software. Conforme o tamanho do template é necessário visualizá-lo por meio de paginação. As ligações correspondem aos templates de nível mais baixo usados no processo de composição e podem ser visualizados com facilidade.

O mesmo hipertexto é utilizado para representar informações que ajudam no entendimento do componente. Uma descrição funcional detalhada do componente, informações de projeto, etc., podem ser visualizadas pelo usuário através da seleção de ligações disponíveis.

O editor de templates compõe o sistema de hipertexto e tem a capacidade de verificar a sintaxe das descrições segundo as regras da linguagem LDT. Além disto, a edição do esqueleto de código sofre uma consistência segundo os tipos de parâmetros a ele associados. Finalmente, o editor ainda oferece ao usuário um cardápio de com as estruturas da LDT (templates?).

A estratégia para composição dos templates é obtida da própria linguagem de programação utilizada. As regras e construções existentes na linguagem naturalmente são utilizadas

quando ocorre a expansão dos templates embutidos em um esqueleto de código.

A figura 3.7 introduz um modelo com as principais entidades e relacionamentos identificados no sistema.

### 3.2.2.1 - Modelo de Classificação

Um sistema de classificação tem como objetivo básico ser eficiente para permitir ao usuário encontrar com facilidade o template que melhor se ajusta aos seus requisitos. Além disso, deve ser extensível de maneira a suportar o crescimento da biblioteca em número de componentes, e adaptável o suficiente para atender à diferentes domínios de aplicação e satisfazer às necessidades de usuários com diferentes características. Tal sistema deve incluir um esquema de classificação, um instrumento para recuperação e um mecanismo para avaliação de componentes similares.

O modelo de classificação adotado é uma adaptação da abordagem proposta no sistema ESF-ROSE, ambiente para reuso de software do projeto ESF (Eureka Software Factory) [18]. Trata-se de um modelo derivado da classificação multiface proposta por Prieto-Diaz [25], com a diferença de que uma faceta é estruturada em vários níveis de abstração (o que conduz à eficiência). Isto permite superar a principal desvantagem do esquema multiface, ou seja, uma faceta ser constituída por uma lista plana de termos em linguagem natural, o que aumenta o tempo de pesquisa.

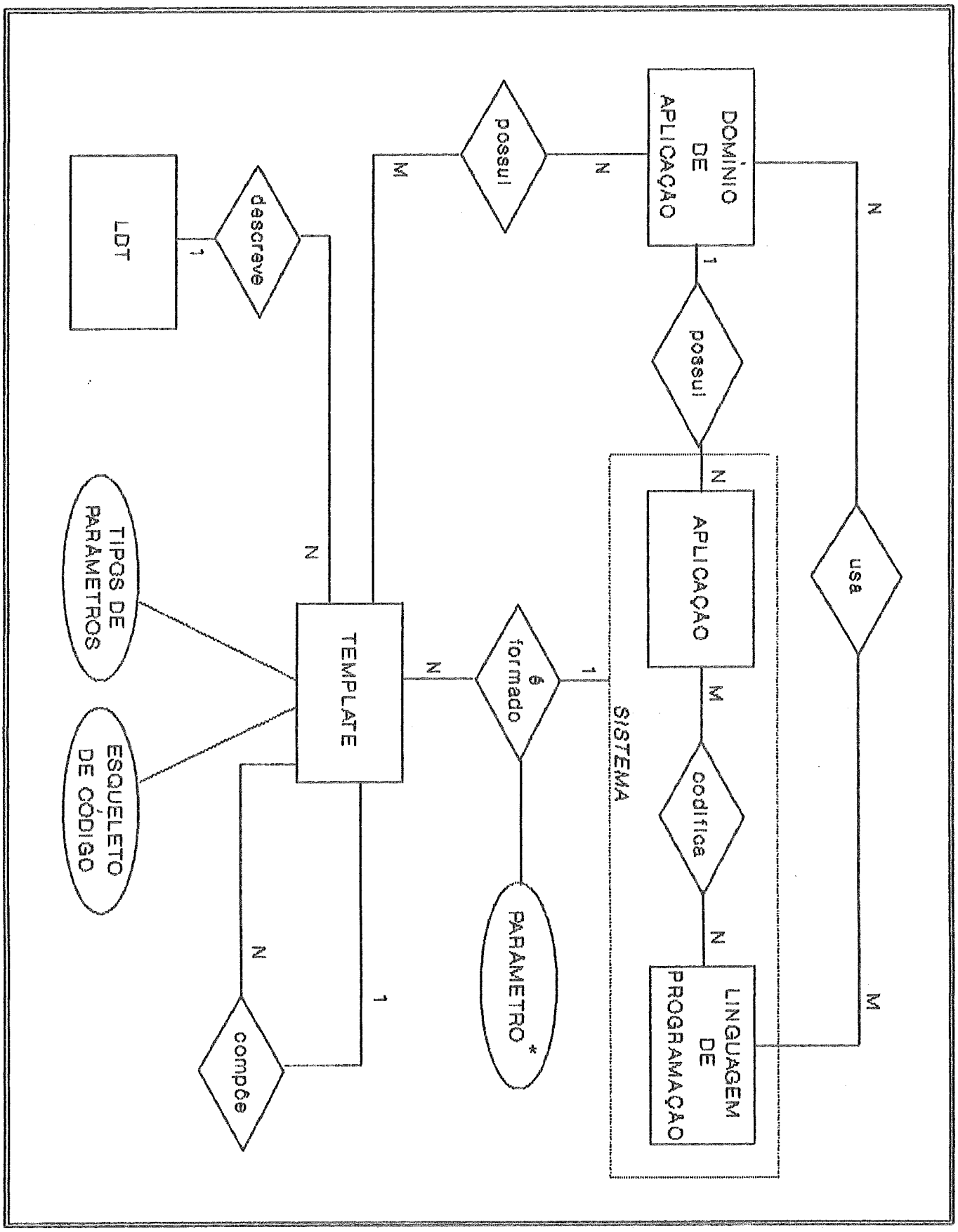



Fig. 3.7 - Entidades e relacionamentos no sistema MARTE

A classificação multiface propõe a reconstrução do universo de conhecimento a partir dos tópicos de interêsse, ou do vocabulário de objetos particulares a serem classificados. Neste processo, os tópicos são analisados em suas classes componentes elementares, e estas classes são agrupadas na forma de listas. Uma classe descreve um conceito do universo de discurso. Os grupos de classes elementares que constituem o esquema são denominados facetas. Estas são consideradas algumas vezes como perspectivas, pontos de vista ou dimensões de um domínio particular.

No sistema MARTE podem ser definidos diversos critérios, ou facetas, dentro do esquema de classificação (o que conduz à adaptabilidade). Esta definição pode se dar ao longo da evolução da biblioteca de templates (o que conduz à extensibilidade). Contudo, dada a generalidade da ferramenta, os critérios **Domínio de Aplicação** e **Linguagem de Programação** devem estar presentes obrigatoriamente.

Um critério (ou faceta) de classificação contém um conjunto de classes. Uma classe é um grupo de elementos que compartilham pelo menos uma característica não compartilhada por membros de outras classes [25]. Uma **ligação de referência** é criada entre o template e a classe que melhor o descreve para um dado critério. Desta forma, um template normalmente pertencerá a diversas classes dentro de diferentes critérios, o que refletirá os seus vários aspectos (facetas). A cada ligação de referência pode ser associado um peso, determinado conforme o grau de semelhança entre as características do template e as propriedades da classe. Este peso é um número utilizado na avaliação de templates equivalentes, e como auxílio ao usuário quando este navega pelo esquema de classificação.

O conjunto de classes dentro de um critério é construído no sentido "bottom-up". Neste processo, duas ou mais

classes de um mesmo nível são sintetizadas em uma única classe de nível mais alto através dos mecanismos de agregação ou generalização.

Além das ligações hierárquicas, classes com algumas propriedades semelhantes podem ser associadas através de ligações semânticas. Isto permite ao sistema realizar pesquisas alternativas para encontrar algum template mais adequado ao usuário, solucionando de forma simples o problema da avaliação de componenentes similares.

Uma classe é caracterizada pelo seguinte conjunto de atributos:

- o nome da classe (identificador, palavra-chave);

- uma descrição informal de suas propriedades;

- uma lista de sinônimos, isto é, termos que se aplicam ao mesmo conceito definido pela classe. O uso de sinônimos evitam descritores duplicados ou ambíguos durante o processo de recuperação;

- uma lista de propriedades, ou seja, termos que descrevem características invariantes de uma classe. Por exemplo, a classe de nome "comunicar", cujos componentes de alguma forma transferem informações entre dois pontos, pode ser caracterizada pelas propriedades: origem, destino e suporte de comunicação. Uma propriedade é um par (atributo, valor), onde o valor associado ao atributo é usado na avaliação de componentes similares.

A figura 3.8 mostra a representação esquemática do modelo de classificação no sistema MARTE.

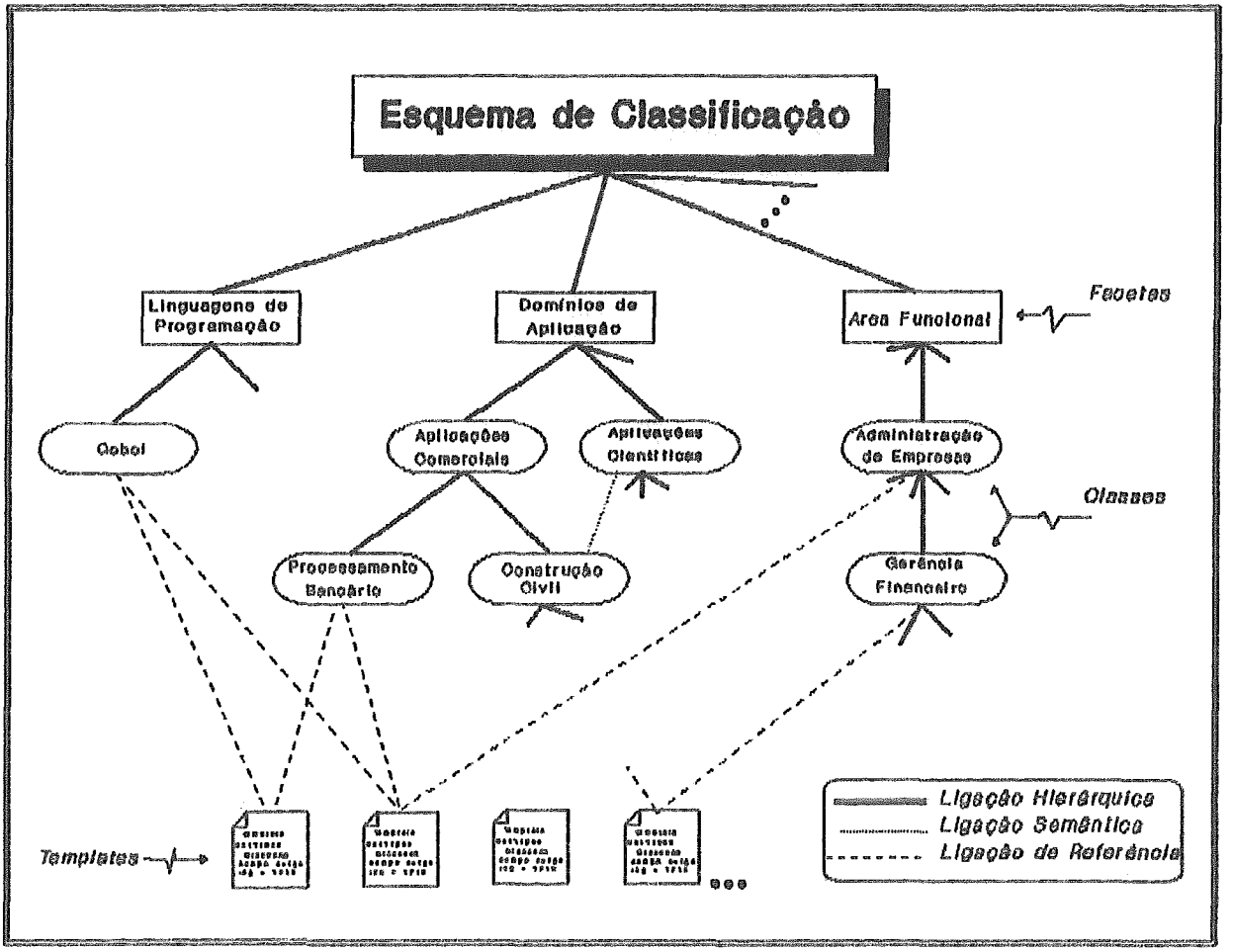



Fig. 3.8 - Modelo de Classificação

O esquema de classificação é a base para recuperação de um template pelo usuário. A pesquisa pode ser realizada de duas formas distintas:

1) O usuário não é capaz de especificar exatamente os requisitos por uma consulta baseada em palavras-chaves. Neste caso, ele navega pelo esquema de classificação e marca as classes de interêsse dentro de vários critérios. O sistema pesquisa pelos templates que pertencem à interseção das classes selecionadas e constitui uma lista para processamento posterior;

2) O usuário tem uma boa idéia do conteúdo da biblioteca e conhece as características exatas do template desejado. Ele formula uma consulta fornecendo os critérios para a pesquisa, e um descritor com as palavras-chaves (prorpriedades) dentro de cada critério. Uma outra alternativa mais simples é fornecer diretamente a classe que melhor caracteriza o template em cada criério.

### 3.2.3 - Interface com Usuário

O sistema MARTE é totalmente interativo e dirigido por "menus". O usuário pode navegar com facilidade pela estrutura hierárquica de funções disponíveis, selecionando cada opção pelo simples toque da tecla associada. Um conjunto básico de padrões de interação (teclas especiais, "layout" de tela, sonorização, etc.) é válido em todo o sistema, o que confere um alto grau de coerência à interface com o usuário. Aliado a esta coerência, a possibilidade da configuração de alguns destes padrões torna possível estabelecer uma interface adequada a cada tipo de usuário. Por exemplo, um usuário mais experiente pode suprimir as mensagens do sistema (avisos, confirmações, etc.) ou desabilitar a sonorização.

De uma forma geral, o sistema MARTE obedece às seguintes especificações concernentes à interface com usuário:

1) O "layout" da tela define áreas específicas para identificação do contexto, mensagens de êrro, informes do sistema, linha de "status" e área de trabalho;

2) A identificação do contexto permite ao usuário se localizar no sistema através do caminho percorrido na estrutura hierárquica de "menus";

3) Uso de cores e sonorização para caracterizar, e padronizar, ações comuns a todos os contextos (por exemplo, mensagens de êrro);

4) Definição pelo usuário de teclas especiais ("hotkeys") para acesso direto às funções de maior interêsse;

5) Recursos configuráveis pelo próprio usuário (vídeo, impressora, sonorização, mensagens, etc.);

6) Recursos para edição de dados, incluindo os seguintes comandos: movimentação de cursor (caractere à direita, caractere à esquerda, palavra seguinte, palavra anterior, início e fim de campo), eliminação de texto (caractere, palavra, linha e campo), modos de operação (inserção e sobrescrita), confirmação e cancelamento.

O sistema MARTE utiliza extensivamente a técnica de hipertexto como instrumento básico de interação com o usuário. Embora existam características específicas em cada caso, um padrão de interface único é válido para todo hipertexto. Os seguintes recursos utilizam-se desta técnica para representação e manipulação dos dados:

- sub-sistema de auxílio ao usuário ("help on-line");
- editor de templates (e sua documentação);
- editor de telas de help da aplicação.

Em qualquer dos módulos acima, o sistema de hipertexto possui os seguintes comandos básicos padronizados:

- rolamento e paginação de telas (por linha e página);

- troca e seleção de ligações (navegação sobre as palavras-chaves e ativação da ligação associada);

- retorno ao último nodo consultado, no caso de ter havido navegação sobre ligações do hipertexto;

- salvamento do hipertexto com a gravação física dos arquivos em disco (apenas na edição);

Conforme o recurso em uso, o sistema de hipertexto deve ser capaz de manipular telas de auxílio do sistema MARTE, templates, documentação dos templates ou telas de help da aplicação. Isto determina um escopo de visualização e um conjunto de funções disponíveis para cada tipo de nodo. Além dos comandos básicos padronizados, são válidas as seguintes especificações:

Telas de Auxílio do sistema MARTE

- de uma tela de auxílio só é possível editar e visualizar outras telas de auxílio;

- tecla especial de acesso direto à tela de auxílio ("help on help") para o contexto de edição de telas de auxílio;

- tecla especial que ativa o Indice Geral de ligações que permite o acesso direto às telas de auxílio existentes (apenas na visualização).

Templates

- de nodo template só é possível editar e visualizar outros nodos templates;

- tecla especial de acesso direto ao nodo de documentação do template (o usuário passa ao modo de documentação e pode assim editar e visualizar toda a documentação );

- tecla especial de acesso direto à tela de auxílio para o contexto de edição de templates (passa ao modo de tela de auxílio).

Documentação de Templates

- de um nodo de documentação de template só é possível editar e visualizar outros nodos de documentação de templates;

- tecla especial de acesso direto à tela de auxílio para o contexto de edição de documentacão (passa ao modo de tela de auxílio).

Telas de Help da Aplicação

- de uma tela de help da aplicação só é possível editar e visualizar outras telas de help;

- tecla especial de acesso direto à tela de auxílio para o contexto de edição de telas de help (passa ao modo de tela de auxílio).

## 3.3 - Desenvolvimento do Protótipo

A construção de um protótipo para o sistema MARTE conforme as suas especificações é que torna possível avaliar a solução proposta neste trabalho. O objetivo inicial foi abranger ao máximo os recursos especificados para o sistema, de forma que o usuário possa experimentar as reais dificuldades para reutilização de um template. Não obstante, algumas facilidades foram implementadas de forma um pouco diferente, ou

simplesmente não existem no protótipo, dada a sua relativa importância frente ao esforço de desenvolvimento requerido.

### 3.3.1 - O Projeto

O desenvolvimento do projeto para o protótipo do sistema MARTE foi conduzido segundo a técnica de projeto orientado pelo fluxo de dados apresentada por Myers [19]. Neste sentido, a arquitetura do sistema em seu nível mais alto mapeia os processos identificados no diagrama de fluxo de dados da especificação funcional (figura 3.9).

O sistema é composto de três grandes módulos, onde os dois primeiros correspondem ao meta-ambiente (Gerenciador de Templates) e ambiente de desenvolvimento de software (ADS), e o terceiro módulo agrega funções de suporte ao sistema como um todo (Manutenção do Sistema), incluindo o gerenciamento de usuários e a configuração dos recursos de hardware.

Para maior flexibilidade e manutenibilidade do sistema MARTE, as regras da Linguagem de Descrição de Templates são parametrizadas na forma de uma Tabela Sintática que fica armazenada em arquivo texto de nome MT_LDT.DAT (Apêndice B). Isto permite alterações livres e imediatas na sintaxe da linguagem, não obstante para acrescentar, ou modificar, alguma rotina semântica seja necessário alterar o código do sistema.

O projeto da base de dados levou em consideração a disponibilidade de um gerenciador de arquivos sequenciais indexados, representado na figura 3.9 pelo módulo pré-definido de nome Gerenciador de Arquivos.

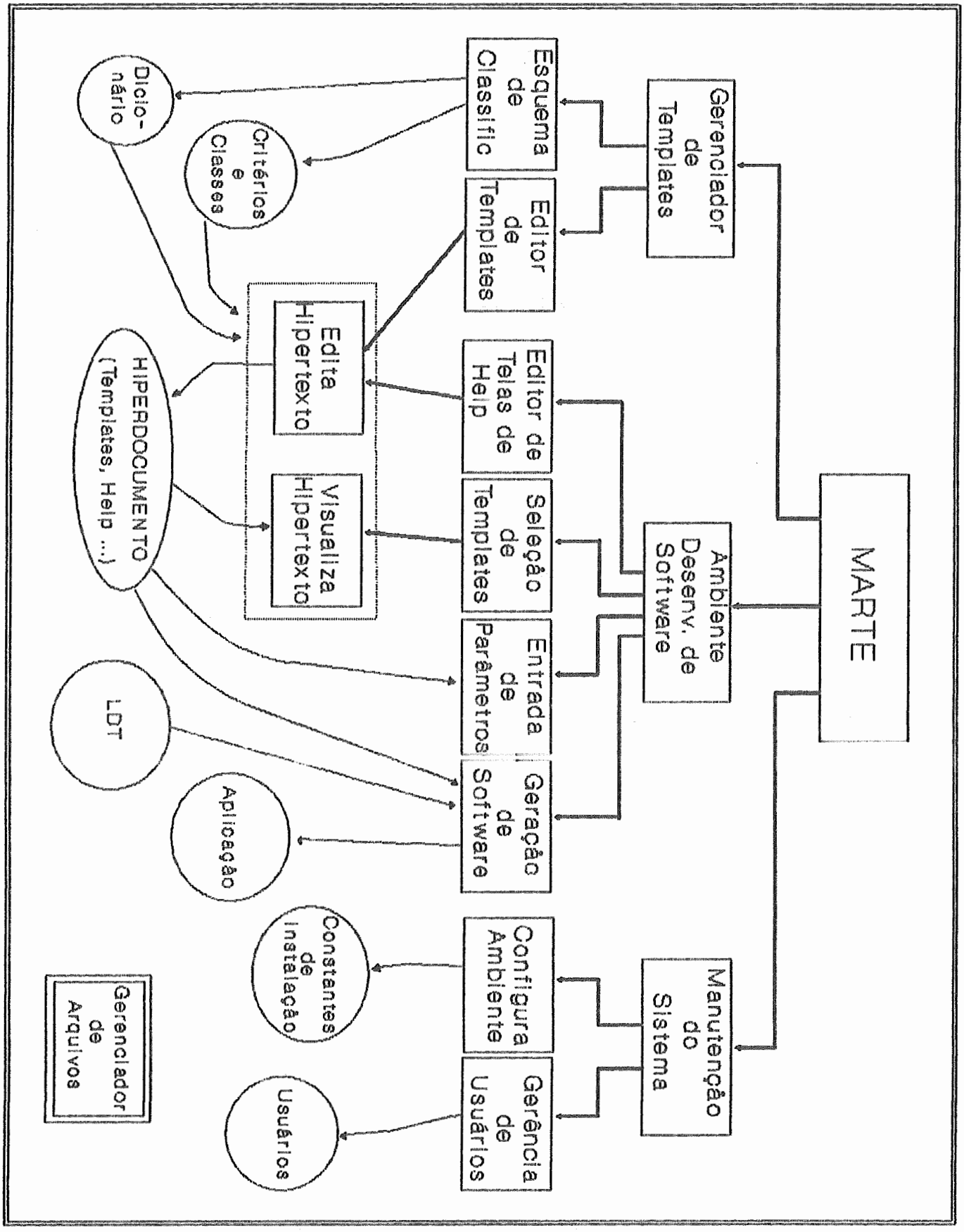



Fig. 3.9 - Visão geral da arquitetura do sistema MARTE

### 3.3.1.1 - Sistema de Hipertexto

O fator determinante ao longo de todo o projeto foi a definição do **Sistema de Hipertexto**, especialmente da estrutura de arquivos que armazena o hiperdocumento. Esta tarefa foi norteada pela necessidade de manipular nodos de natureza diversa, o que exige contextos diferenciados para edição e visualização do hipertexto. Além disto, o sistema de hipertexto foi simplificado permitindo apenas ligações **nodo a nodo** e **ponto a nodo**, visto não haver razão de particionar um documento (template, tela de help, etc.) e fazer ligações ponto a ponto.

A estrutura de dados externa que representa o hipertexto é constituída dos seguintes arquivos (figura 3.10):

Conjunto de nodos (MT_HPTXN.DAT)

- tipo do nodo (Auxílio, Template, Documentacao, Help)
- identificador do nodo
- descrição textual
- tamanho em nº de linhas
- número de pontos que referenciam o nodo
- apontador para a 1ª linha do nodo

Pontos de referência (ligações) (MT_HPTXP.DAT)

- identificador do ponto (palavra-chave)
- apontador para o nodo referenciado

Linhas de texto nos nodos (MT_HPTXL.DAT)

- apontador para a linha anterior
- apontador para a linha seguinte
- apontador para o nodo a que pertence a linha
- conteúdo da linha

Indices do arquivo de nodos (MT_HPTXN.IDX)

. referência ao nodo através do seu código

Indices do arquivo de pontos (MT_HPTXP.IDX)

. referência ao ponto através do seu identificador (permite referência indireta ao nodo associado)

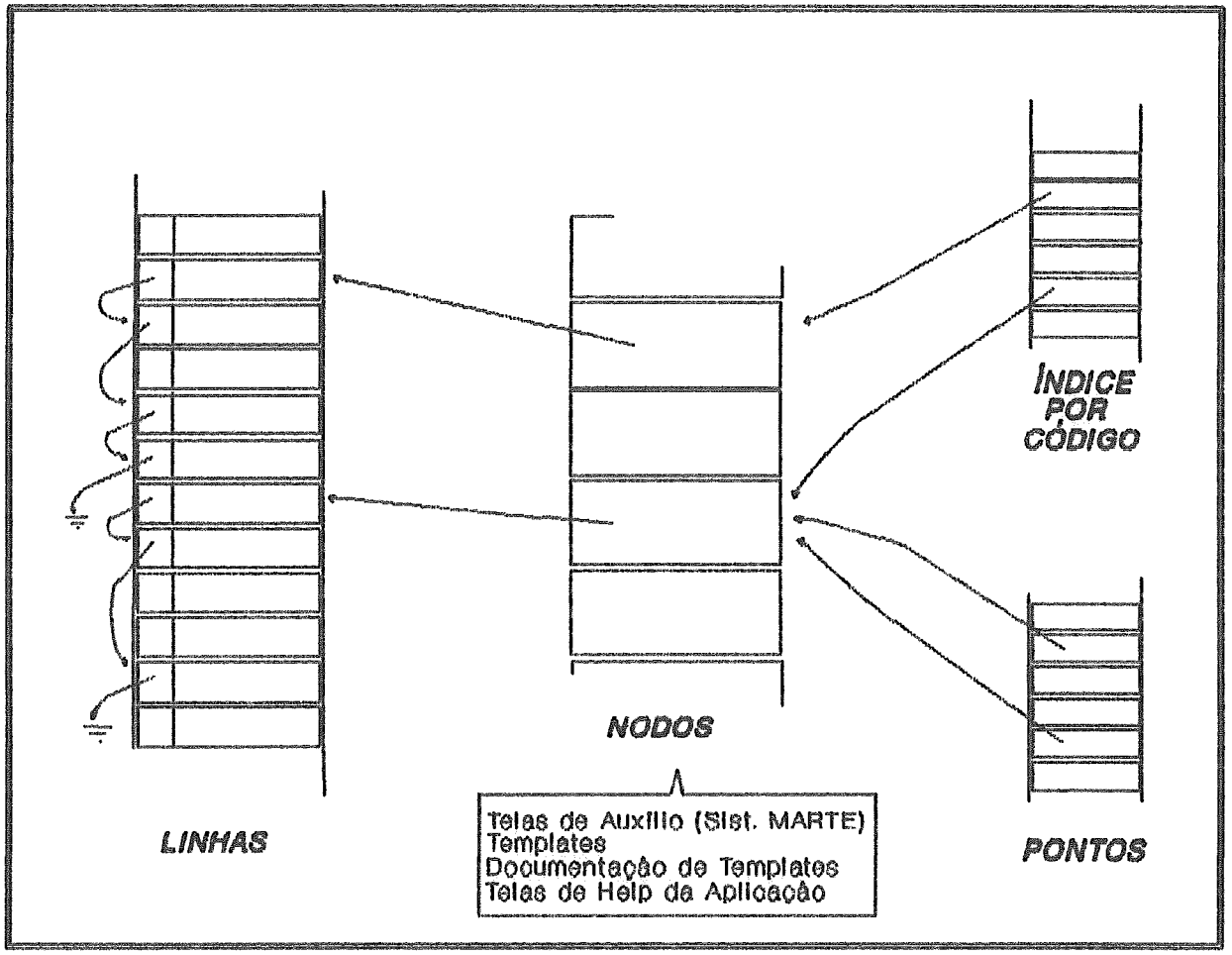



Fig. 3.10 - Estrutura de arquivos do hipertexto

Para simplificar a interface dos módulos é declarada uma estrutura de hipertexto padrão do tipo denominado HptxFile, que engloba todos estes arquivos e mais as constantes parametrizadas MaxHptxKeyLen (tamanho da maior palavra-chave possível) e MaxHptxLineLen (tamanho da maior linha de texto permitida). Esta é a única estrutura de dados necessária para se criar e/ou manipular um hipertexto.

Os seguintes módulos constituem o sistema de hipertexto e são disponíveis externamente:

CriaHptx

.*função:* cria os arquivos de um hipertexto.

.*parâmetros:* estrutura do tipo HptxFile (E/S), nomes externos dos arquivos e constantes parametrizadas.

AbreHptx

.*função:* abre os arquivos de um hipertexto.

.*parâmetros:* estrutura do tipo HptxFile (E/S), nomes externos dos arquivos e constantes parametrizadas.

FechaHptx

.*função:* fecha os arquivos de um hipertexto.

.*parâmetros:* estrutura do tipo HptxFile (E/S).

EditHptx

.*função:* permite a edição dos nodos de um hipertexto.

.*parâmetros:* estrutura do tipo HptxFile (E/S), palavra-chave de referência ao nodo, coordenadas da janela de edição.

MostraHptx

.*função:* permite a visualização dos nodos de um hipertexto.

.*parâmetros:* estrutura do tipo HptxFile (E/S), palavra-chave de referência ao nodo, palavra-chave para o Indice Geral, coordenadas da janela de visualização.

Finalmente, a estrutura de dados adotada permite a edição a nodos com tamanho variável (em nº de linhas), e otimiza o uso de espaço em disco armazenando apenas os caracteres editados em cada linha.

### 3.3.1.2 - Gerenciador de Templates

O ambiente Gerenciador de Templates possui os recursos que permitem ao meta-usuário definir e manipular os templates a partir de dois grandes módulos: o Editor de Templates e o Esquema de Classificação (figura 3.9).

O *Editor de Templates* corrresponde ao Sistema de Hipertexto operando no modo de edição de templates. Simplesmente são inicializadas algumas constantes e feita uma chamada ao módulo EditHptx, passando como parâmetro a palavra-chave do nodo a ser editado igual a nulo. O editor do hipertexto passa direto ao menu inserção/pesquisa, que permite ao usuário catalogar um novo template ou pesquisar por algum já existente para alteração. A partir deste ponto, o sistema de hipertexto oferece todos os recursos necessários para atualização da biblioteca de templates e sua documentação.

A criação de um novo template exige a sua classificação com vistas ao acesso futuro para geração do software. O meta-usuário deve, primeiramente, definir os critérios, classes e respectivas propriedades através do módulo *Esquema de Classificação*. A figura 3.11 mostra as principais estruturas de dados externas utilizadas para representar o modelo de classificação adotado no sistema MARTE (figura 3.8). Os seguintes atributos e relacionamentos podem ser observados:

Classes

.*conteúdo:* arquivo de dados contendo os critérios e classes.

.*atributos:* código hierárquico interno, identificador da classe e descrição textual detalhada.

.obs: Ao definir uma classe ou critério o usuário fornece ainda os sinônimos, propriedades e ligações semânticas que serão armazenados em arquivos auxiliares. O código interno

permite diferenciar entre classe e critério, além de organizar as classes de forma hierárquica. Trata-se de um conjunto de até 10 dígitos separados aos pares por meio de pontos (99.99.99.99.99). Este código é transparente ao usuário e permite até 5 níveis hierárquicos, incluindo o primeiro nível reservado para os critérios.

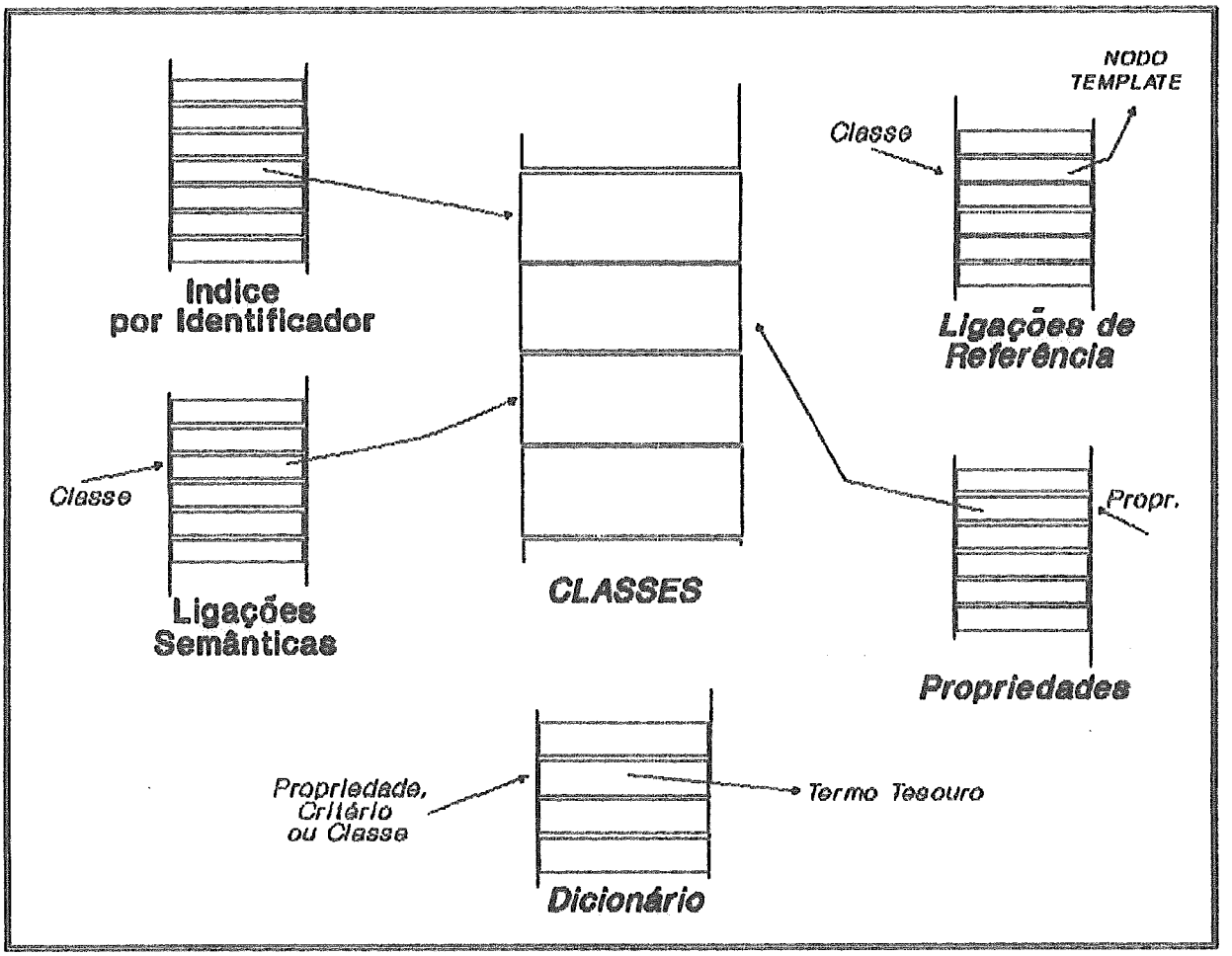



Fig. 3.11 - Estrutura de arquivos do Esquema de Classificação

Indices por Identificador

.*conteúdo:* arquivo de índices por identificador para as classes e critérios.

.*obs:* O usuário utiliza o identificador para referência direta a uma classe ou critério (chave primária).

Dicionário

.*conteúdo:* arquivo de dados contendo os sinônimos para classes, critérios e propriedades associados a seus termos principais (termo tesouro).

.*obs:* A pesquisa é feita através de um arquivo de índices por sinônimo. Apenas os termos tesouro são utilizados na classificação de templates. Um módulo específico permite a manutenção do dicionário diretamente pelo usuário ("off-line").

Propriedades

.*conteúdo:* arquivo de índices para referência das classes associadas a uma dada propriedade.

.*obs:* Um arquivo de dados contendo as propriedades e respectivos identificadores de classes assegura a integridade da base de dados.

Ligações Semânticas

.*conteúdo:* arquivo de índices para referência das classes relacionadas semanticamente.

.*obs:* Um arquivo de dados contendo os identificadores de ambas as classes que definem as ligações semânticas assegura a integridade da base de dados.

Ligações de Referência

.*conteúdo:* arquivo de índices para referência aos templates (nodos do hipertexto) caracterizados por uma dada classe.

.*obs:* A referência ao nodo é criada no momento da definição de um novo template, onde o meta-usuário fornece uma classe dentro de cada critério relevante para sua classificação. Um arquivo de dados contendo os identificadores de classes e respectivos códigos de templates assegura a integridade da base de dados.

### 3.3.1.3 - Ambiente de Desenvolvimento de Software

Este módulo possui os recursos que permitem ao usuário final configurar a sua aplicação e gerar o código fonte correspondente (figura 3.9).

Inicialmente o usuário localiza o template mais adequado aos requisitos de sua aplicação através do módulo **Seleção de Templates**. Para tanto, são disponíveis as seguintes opções de consulta à biblioteca:

Seleção Direta: O usuário conhece exatamente qual o template de interêsse e fornece o seu código. Caso este código seja válido, o sistema pede confirmação ao usuário, permitindo a visualização do template e de sua documentação. Isto é conseguido ativando o Sistema de Hipertexto no modo de visualização de templates. Algumas constantes são inicializadas antes da chamada ao módulo MostraHptx, passando como parâmetro a palavra-chave do nodo a ser visualizado, ou seja, o código do template fornecido pelo usuário. Uma vez no contexto de visualização do template, o usuário tem acesso direto à documentação correspondente através de uma tecla especial.

Busca Exploratória por Templates: O usuário não conhece o template e tampouco é capaz de especificar exatamente os requisitos por uma consulta baseada em palavras-chaves. Este módulo permite que ele selecione um template inspecionando diretamente a biblioteca através de uma lista com os códigos e as descrições dos templates existentes. A lista é classificada em ordem alfabética de código, e o usuário pode comandar a visualização do template desejado.

Busca Exploratória através das Classes: Como no caso anterior, o usuário também não consegue especificar os requisitos por uma consulta baseada em palavras-chaves. Este módulo permite que ele navegue pelo esquema de classificação e marque uma classe dentro de cada critério de interêsse. O sistema pesquisa pelos templates que pertencem à interseção das classes selecionadas e constitui uma lista ordenada para processamento posterior. O usuário pode então visualizar cada template da lista na ordem computada pelo sistema a partir dos pesos assinalados às ligações de referência.

Recuperação Sistemática por Identificadores de Classes: O usuário tem uma boa idéia do conteúdo da biblioteca e conhece as características exatas do template desejado. Ele formula uma consulta fornecendo os critérios para a pesquisa, e o identificador da classe que melhor caracteriza o template em cada criério. O sistema encontra os templates referenciados por cada classe e constitui uma lista ordenada com aqueles que fazem parte do conjunto interseção, ou seja, são referenciados por todas as classes fornecidas na consulta. O usuário pode então visualizar cada template da lista na ordem sugerida pelo sistema a partir dos pesos assinalados às ligações de referência.

Recuperação Sistemática por Propriedades: O usuário conhece exatamente os requisitos de sua aplicação. Ele formula uma consulta fornecendo um descritor de palavras-chaves (propriedades) que melhor caracterizam o template em cada criério de interêsse. O sistema encontra as classes cujas propriedades mais se ajustam aos descritores da consulta, utilizando para isto os pesos associados às propriedades. A partir deste ponto a pesquisa é processada como no caso anterior.

Nas duas últimas opções de consulta o sistema pode, sob comando do usuário, reprocessar a pesquisa utilizando as ligações semânticas para recuperar os componentes similares.

Uma vez selecionado o template, o usuário fornece os parâmetros necessários através do módulo *Entrada de Parâmetros*. A interação com o sistema é bastante amigável, sendo monitorada tanto quanto possível pela definição de tipos contida no template. Por exemplo, quando for o caso dos valores possíveis estarem presentes na definição de tipos (classes alternativas), o sistema orienta a entrada o parâmetro através de menus (exemplo na seção 3.2.1). Quando houver a necessidade de entrar com coordenadas de vídeo, o usuário pode desenhar "layouts" diretamente na tela por meio de teclas e funções especiais.

Finalmente, através do módulo *Geração de Software* pode-se obter o código fonte da aplicação combinando o template com os parâmetros fornecidos. O usuário unicamente deve comandar a geração e observar os avisos e mensagens de erro relacionados pelo sistema. Em uma situação normal são criados três arquivos de saída com as extensões TPL (template completo), PAR (parâmetros) e MRT (programa fonte). O nome dos arquivos é o mesmo da aplicação, atribuído internamente pelo usuário.

O projeto completo do gerador foi baseado na estrutura funcional para um compilador apresentada no trabalho de Setzer e Melo [30]. Sua arquitetura básica inclui um analisador léxico e um analisador sintático controlado pela tabela sintática da LDT. O automato finito do analisador léxico é apresentado no Apêndice C.

O desenvolvimento de uma nova aplicação presupõe a sequência das ações conforme foram apresentadas, ou seja, seleção do template, entrada de parâmetros e geração do

software. Não obstante, o arquivo do template completo já pode existir e ser usado para gerar um novo fonte com novos parâmetros. Neste caso a etapa de seleção do template é dispensável. O sistema é capaz de gerenciar casos desta natureza e manter a integridade do processo.

O usuário pode ainda definir um hipertexto de auxílio para a sua aplicação através do módulo *Editor de Telas de Help*. Este consiste basicamente de uma chamada ao Sistema de Hipertexto no modo de edição de Telas de Help, ativando-se a rotina EditHptx após as devidas inicializações. O módulo é completado por uma opção que permite extrair as telas de help do hipertexto e gerar um arquivo texto padrão.

### 3.3.1.4 - Emissão de Relatórios

Visto que o principal produto do sistema MARTE são os arquivos da aplicação do usuário, este módulo contém apenas relatórios a serem utilizados na conferência de dados ou como documentação permanente. As seguintes opções são disponíveis:

Esquema de Classificação: São listados os critérios e classes, incluindo todos os seus atributos (identificador, descrição, sinônimos, etc.). Critérios de seleção: apenas uma classe, apenas as classes dentro de um critério ou todas as classes existentes.

Hierarquia de Classes: Constitui uma lista hierarquizada das classes, incluindo apenas o identificador. Critérios de seleção: apenas as classes dentro de um critério ou todas as classes existentes.

Biblioteca de Templates: Imprime uma lista de templates com todos os atributos (código, classificação, etc.), incluindo a sua definição. Critérios de seleção: um único template, apenas os templates referenciados por uma dada classe ou todos os templates existentes.

### 3.3.1.5 - Manutenção do Sistema

Todo sistema de programação deve possuir um conjunto de recursos que determinam as suas condições operacionais. No sistema MARTE tais recursos constituem o módulo Manutenção do Sistema (figura 3.9), a saber:

- Informe dos Arquivos
- Consistência da Base de Dados
- Configuração do Ambiente
- Gerência de Usuários

O *Informe dos Arquivos* é uma rotina que exibe na tela do vídeo estatísticas sobre a base de dados. Para cada arquivo é fornecido o número de registros em uso (ativos) e o total de registros ocupados, incluindo os registros apagados pelo usuário. Embora simples, esta função permite um controle sobre o crescimento das estruturas e sobre a área de ocupação no dispositivo de armazenamento.

A *Consistência da Base de Dados* é um relatório que determina as possíveis irregularidade existentes nos arquivos do sistema. Em particular, verificam-se os dados considerando relacionamentos cuja consistência não é possível em tempo de edição. Este procedimento visa manter a integridade da base de dados sem perder na eficiência do sistema.

O módulo *Configuração do Ambiente* reúne diversos recursos que determinam as condições de interação com o sistema. Os seguintes grupos de parâmetros podem ser configurados pelo usuário:

Msgs, Sons, Relógio e E/S do Vídeo: Emissão de mensagens (com exceção de mensagens de êrro), sonorização, exibição do relógio do sistema e como acessar o vídeo (via memória ou via sistema operacional).

Constantes da Impressora: Códigos especiais constituídos de 1, 2 ou 3 constantes para configurar modos de impressão, comandos, conjunto de caracteres e demais características da impressora em uso.

Teclas Quentes ("HotKeys"): Teclas especiais que permitem o acesso direto a determinados pontos de interação do sistema. O usuário pode definir teclas de funções (F2, F3, ...) diretamente como "hotkeys", ou suas combinações com as teclas Shift, Ctrl e Alt. Para isto, ele associa à tecla escolhida o caminho que vai do menu principal até o ponto de interação (contexto) desejado. Existem duas teclas quentes que não podem ser alteradas: F1 que ativa o sistema de auxílio, e F10 que conduz diretamente ao menu principal. A ativação de uma "hotkey" implica no abandono da operação corrente.

Cores do Sistema: tabela de cores utilizadas pelo sistema para destacar molduras, opções de menu, mensagens de erro, ligações de hipertexto, etc.

Todas as constantes de configuração são armazenadas em um único arquivo de dados (INSTALL.DAT), e a referência é conseguida por identificadores internos agrupados em um arquivo de índices (INSTALL.IDX).

Finalmente, o módulo *Gerência de Usuários* contém as rotinas necessárias para manutenção de um cadastro dos usuários autorizados a operar o sistema (figura 3.9). Cada usuário possui os seguintes atributos:

. nome completo do usuário.

. código de acesso (chave primária).

. senha individual.

. prioridade (dígito de 0 a 9) que define quais as funções o usuário tem autorização para executar. Este atributo permite diferenciar entre meta-usuário e usuário final.

. grau de supervisão (dígito de 0 a 9) que define o acompanhamento do sistema durante a sessão de trabalho do usuário. Pode ser criado um histórico com a data, hora e tipo transação executada pelo usuário, tanto mais detalhado quanto for o seu grau de supervisão. Este mecanismo permite caracterizar falhas de operação do sistema, acompanhar o treinamento de novos usuários, ajudar na avaliação dos operadores, etc.

O acesso aos dados do cadastro de usuários é controlado segundo a hierarquia estabelecida pelas prioridades. Isto significa que um dado usuário só pode manipular as informações dos usuários de menor prioridade. A senha individual é exibida apenas para o próprio usuário.

As estruturas de dados externas são representadas por um arquivo de dados (USUÁRIOS.DAT) e um arquivo de índices por código (USUÁRIOS.IDX).

### 3.3.2 - Aspectos da Implementação

O protótipo do sistema MARTE foi implementado para computadores da linha IBM-PC, operando sob o sistema operacional DOS. A linguagem de programação foi o Pascal, na sua implementação TurboPascal, versão 5.0 [37]. Para o gerenciamento da base de dados foi utilizado o pacote Database Toolbox, versão 4.0 [36]. Este módulo contém declarações e rotinas para o gerenciamento de arquivos sequenciais indexados através de estruturas em árvore-B.

O código fonte do sistema distribui-se em 30 arquivos (extensão .PAS) que refletem a estrutura do projeto, a saber:

| | | |
|---|---|---|
| MT | : programa principal | ("program") |
| MT_VARS | : Declarações do Sistema | ("unit") |
| MT_GLOB | : Rotinas Globais ao sistema | ("unit") |
| MT_GT | : Gerenciador de Templates | ("unit") |
| MT_GT_EC | : Esquema de Classificação | ("unit") |
| MT_GT_ET | : Editor de Templates | ("unit") |
| MT_ADS | : Ambiente de Desenvolvimento de Software | ("unit") |
| MT_ADSST | : Seleção de Templates | ("unit") |
| MT_ADSEP | : Entrada de Parâmetros | ("unit") |
| MT_ADSGS | : Geração de Software | ("unit") |
| MT_ADSTH | : Telas de Help da Aplicação | ("unit") |
| MT_GF | : Gerador de Fontes | ("unit") |
| MT_GF_VA | : Declarações do Gerador de Fontes | ("include") |
| MT_GF_AL | : Analisador Léxico | ("include") |
| MT_GF_AS | : Analisador Sintático | ("include") |
| MT_GF_UT | : Utilitários do Gerador de Fontes | ("include") |
| MT_HPTX | : Sistema de Hipertexto | ("unit") |
| MT_HPTXE | : Edição do Hipertexto | ("include") |
| MT_HPTXS | : Visualização do Hipertexto | ("include") |
| MT_HPTXF | : Manipulação dos Arquivos do Hipertexto | ("include") |
| MT_HPTXU | : Utilitários do Sistema de Hipertexto | ("include") |
| MT_RE | : Relatórios | ("unit") |
| MT_MS | : Manutenção do Sistema | ("unit") |

| | | |
|---|---|---|
| UTIL_DCL | : Declarações Gerais | ("unit") |
| UTIL_GR1 | : Utilitários de Uso Geral | ("unit") |
| UTIL_GR2 | : Utilitários de Uso Geral | ("unit") |
| UTIL_INS | : Configuração do Ambiente | ("unit") |
| UTIL_MAN | : Utilitários para Manutenção | ("unit") |
| UTIL_REL | : Utilitários para Emissao de Relatórios | ("unit") |
| UTIL_USR | : Gerência de Usuários | ("unit") |

Na última compilação geral foram contabilizadas cerca de 15.000 linhas de código, excluindo os módulos Esquema de Classificação e Entrada de Parâmetros ainda não integrados ao sistema. Também não foi considerado o módulo pré-definido Database Toolbox. Nesta versão o programa objeto (MT.EXE) totalizou 162 Kbytes em um único módulo de execução.

# 4 - CONCLUSÕES

Reutilização é de fato um dos paradigmas fundamentais de desenvolvimento [1]. Ainda que os problemas existentes sejam de grande amplitude, sua aplicação na Engenharia de Software passa pela definição de um mecanismo mais adequado e eficiente. Técnicas e ferramentas já conhecidas, tais como hipertexto e editores orientados por sintaxe, podem ajudar a compor um ambiente que concretize em definitivo a reutilização de software.

Embora não apresente nenhuma novidade do ponto de vista técnológico, o sistema MARTE é um exemplo simples de como combinar idéias e mecanismos já existentes para compor uma ferramenta que busca viabilizar o reuso de software. Em particular, a solução proposta neste trabalho apresenta algumas vantagens significativas:

- a flexibilidade do gerador, pelo uso de padrões externos (independência da linguagem e domínio de aplicação), aumenta de forma significativa a generalidade do mecanismo e, conseqüentemente, o benefício alcançado será maior;

- uma biblioteca de padrões fragmentados e relacionados de forma a viabilizar o desenvolvimento incremental de software, aumenta o potencial de reuso na medida em que permite a geração de pequenas rotinas, trechos de programas ou sub-programas, ao invés do uso restrito de um padrão monolítico que represente todo um sistema;

- os problemas apresentados pelo uso de uma biblioteca de componentes (acesso ao componente,

potencial de reuso, etc.) são compensados pelo grande benefício proporcionado pelo mecanismo de geração. Diminui a necessidade de adaptações dos componentes devido à parametrização dos padrões.

- o desenvolvimento incremental a partir da composição de padrões permite a reutilização de grandes componentes que podem ser incorporados na biblioteca. Isto também minimiza o investimento inicial exigido pelo mecanismo da composição.

Independente do reuso de software, o sistema MARTE pode ser visto como um gerador de programas que consegue ser genérico e flexível o suficiente para acomodar qualquer classe de aplicações pertencentes a qualquer domínio. Afinal, é muito mais fácil definir um novo template no sistema MARTE do que desenvolver um novo programa gerador a cada padrão de arquitetura diferente.

Considerando o reuso de software, a dificuldade existente na definição de um template, em grande parte devido ao uso de uma linguagem específica (LDT), é compensado pela sua reutilização no próprio sistema. Por outro lado, o maior esforço exigido para a compreensão do template é minimizado pelo mecanismo de geração, uma vez que é automatizada toda e qualquer alteração com a finalidade de instanciar o template para um conjunto específico de parâmetros.

O sistema MARTE pode ser melhorado em diversos aspectos, todos eles visando facilitar o trabalho do usuário para maior benefício no reuso de um componente. Por exemplo, o editor de templates pode ser orientado pela sintaxe da LDT, o que livraria o usuário do conhecimento de mais esta linguagem. A seleção exploratória através das classes, ou diretamente sobre

os templates, pode ser mais amigável pelo uso de navegadores ("browser") sofisticados, que tirem maior proveito da estrutura do hipertexto e do esquema de classificação. O hipertexto poderia incluir nodos com informações gráficas para documentação de um template, tais como diagramas de projeto, esquemas, fluxogramas, etc.

A pesquisa da abordagem proposta neste trabalho pode ser continuada no futuro por meio de duas alternativas. A primeira delas é buscar maior generalidade ou poder de abstração para o template, de maneira que seja possível parametrizar também as estruturas de dados e parte dos algoritmos representados. Neste sentido, alguns trabalhos sugerem o uso de tipos abstratos de dados [39] e de técnicas da Inteligência Artificial [14].

A segunda alternativa consiste em determinar regras de composição para a definição de tipos de parâmetros, de maneira que o desenvolvimento de templates possa ser incremental também a nível destas definições. Isto permitirá evitar a duplicação de definições em templates aninhados, e livrará o meta-usuário de gerenciar este problema.

Finalmente, vale a pena pesquisar a possibilidade da geração (semi-) automática de um template a partir do programa fonte. Dentre vários benefícios, isto permitiria o reuso de todo software já existente através da construção rápida e barata de ricas e volumosas bibliotecas.

O sistema MARTE aponta para o caminho da combinação de diferentes tecnologias de forma a minimizar as suas limitações e somar as suas vantagens. É preciso encontrar um ponto de equilíbrio entre os mecanismos já conhecidos de forma a otimizar as características desejáveis para maior reusabilidade.

# Apêndice A
# Diagrama Sintático da LDT

Apresenta-se a Linguagem de Descrição de Templates (LDT) representada gramaticalmente por meio de grafos sintáticos, o que permite uma melhor visualização e facilita a compreensão de suas estruturas. Os nós que correspondem a caixas arredondadas são símbolos terminais, e os nós representados por caixas retangulares são símbolos não-terminais (figuras A.1 a A.4). Maiores detalhes sobre as regras de construção desses grafos podem ser encontradas no trabalho de Setzer e Melo [30].

O significado de alguns dos símbolos terminais utilizados nos grafos sintáticos são apresentados abaixo:

*idNovo* : identificador que aparece, obrigatoriamente, pela primeira vez;

*idClasseS* : identificador de classe secundária;

*idClasseA* : identificador de classe alternativa;

*idClasseP* : identificador de class principal;

*cadeia* : cadeia de caracteres, tal como definido pela maioria das linguagens de programação;

*vap* : valor associado a parâmetro;

*idVap* : identificador de valor associado a parâmetro.

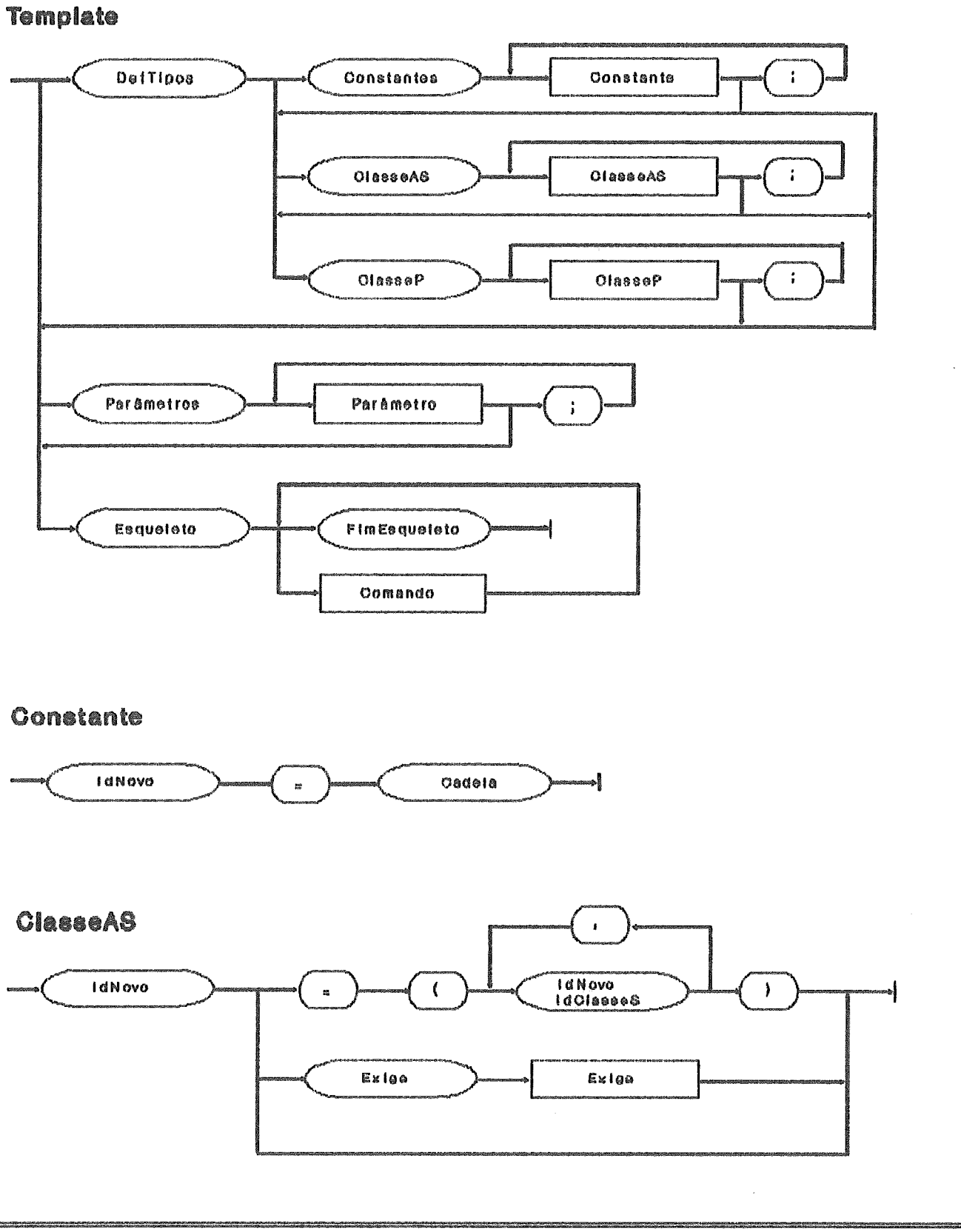
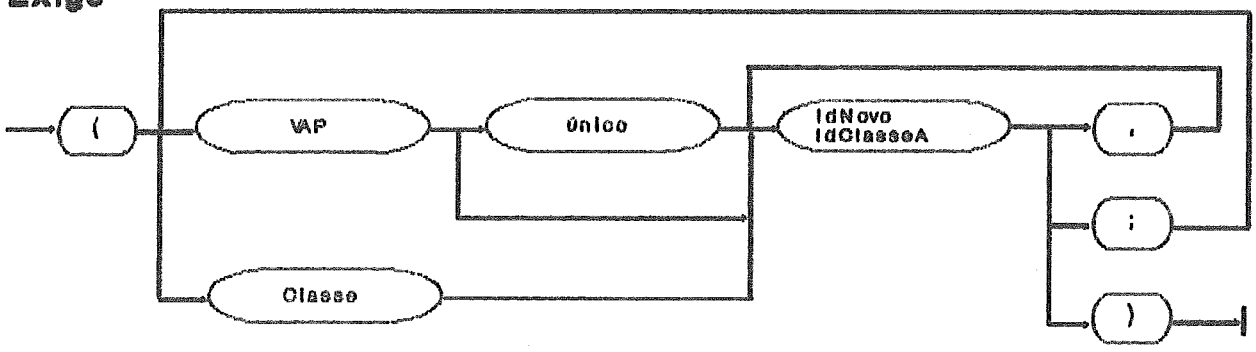


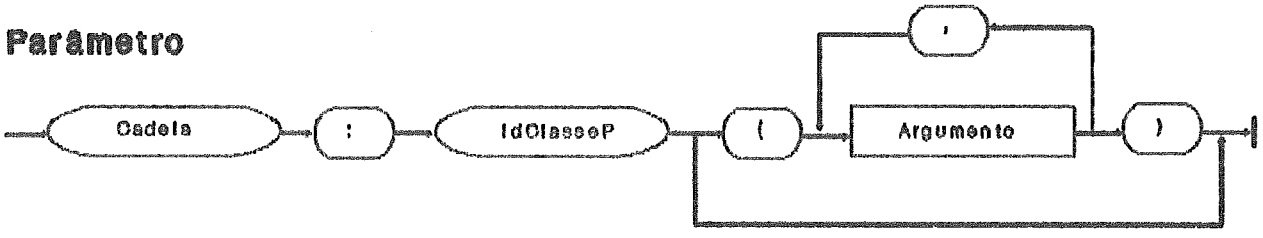

Fig. A.1 - Diagrama Sintático da LDT

**Exige**




**ClasseP**


**Parâmetro**




**Argumento**

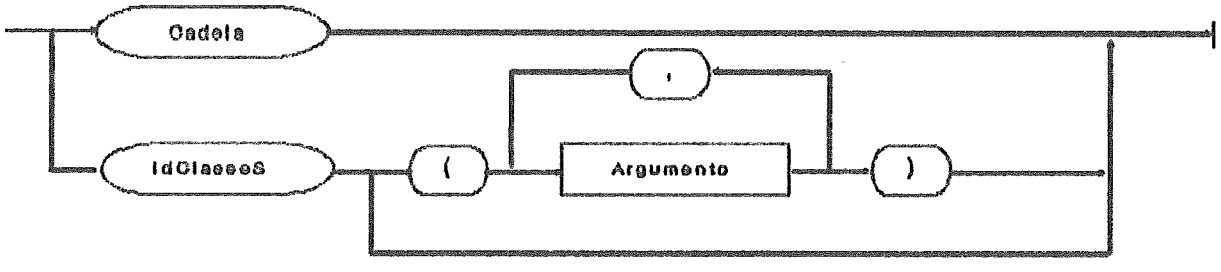



Fig. A.2 - Diagrama Sintático da LDT (cont.)

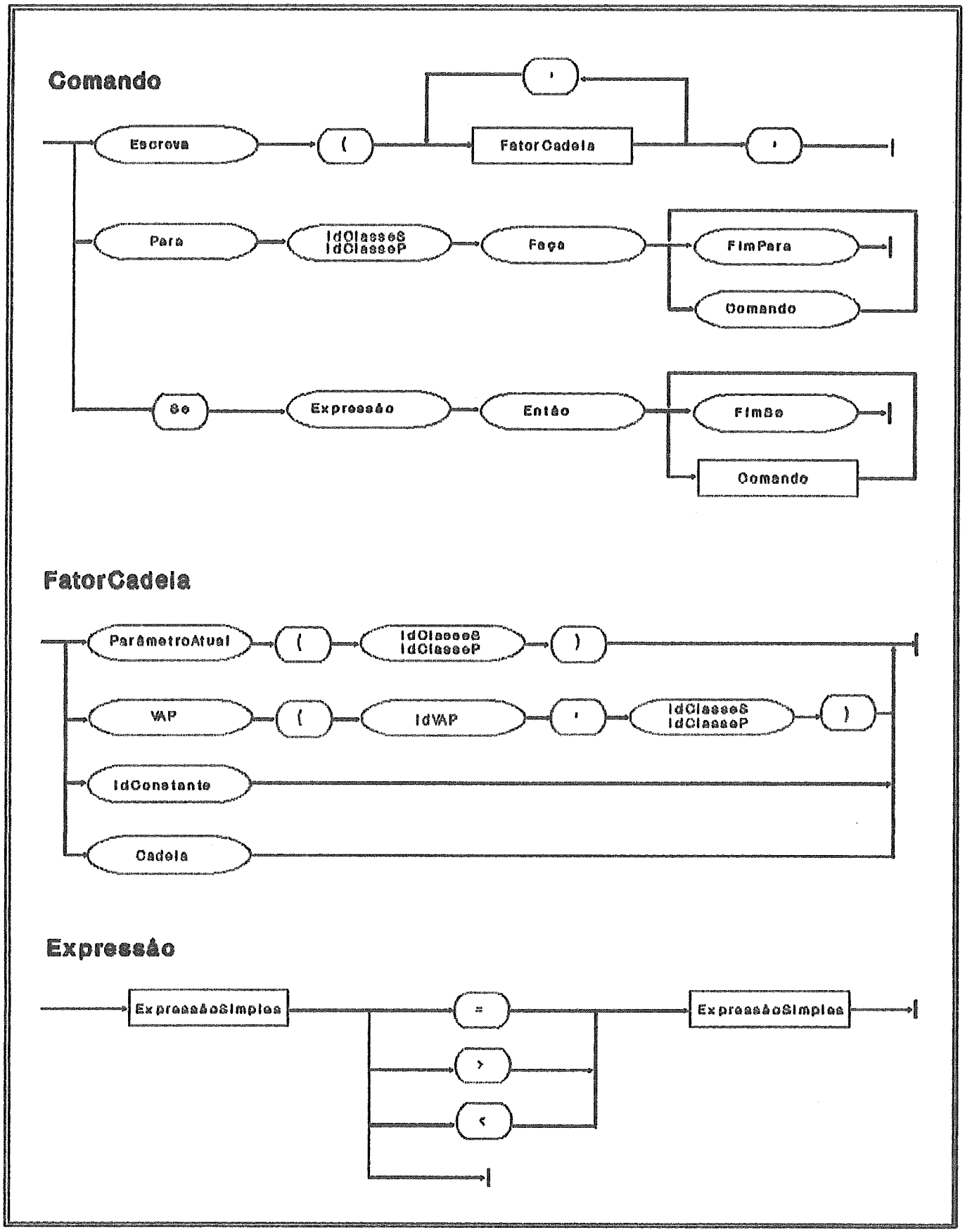
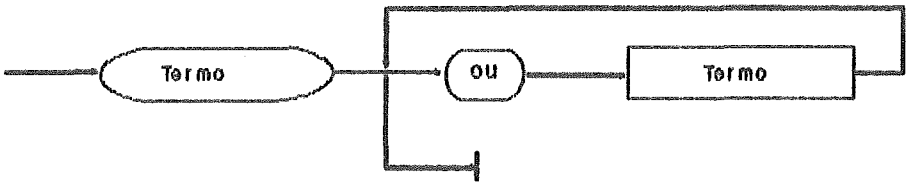
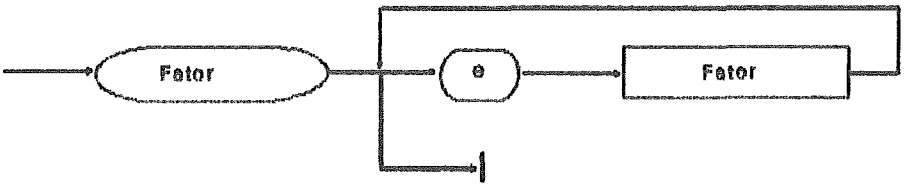



Fig. A.3 - Diagrama Sintático da LDT (cont.)

**ExpressãoSimples**




**Termo**




**Fator**

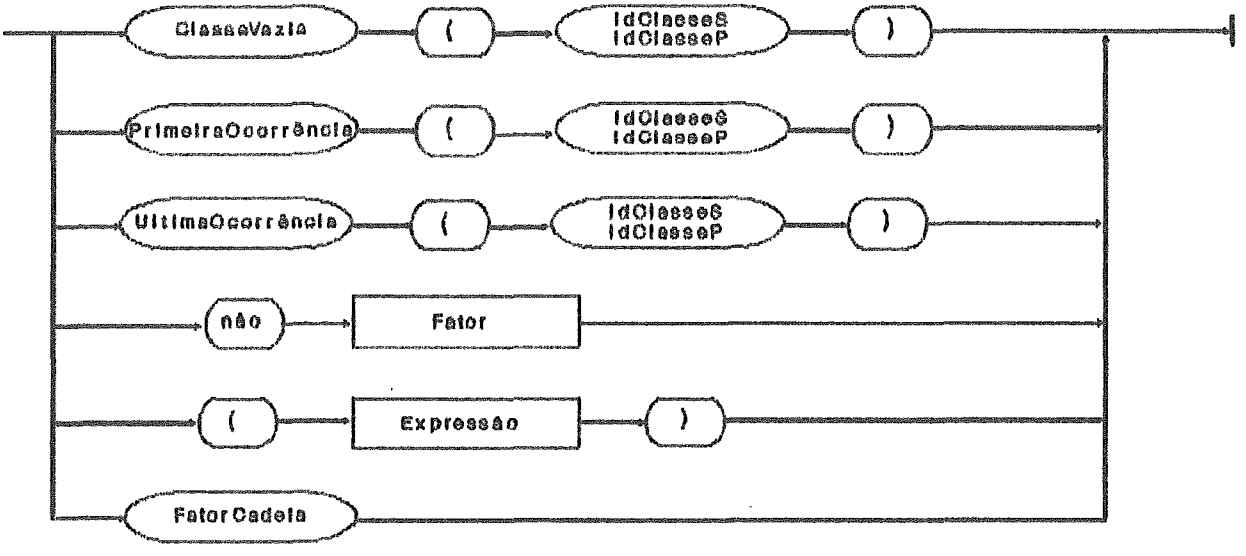



Fig. A.4 - Diagrama Sintático da LDT (cont.)

# Apêndice B
# Tabela Sintática da LDT

Do diagrama da LDT (Apêndice A) é derivada uma Tabela Sintática de formato mais adequado para utilização no sistema, em especial pelo módulo Gerador de Fontes. Além das regras da linguagem esta tabela inclui para alguns dos nós um número da rotina semântica que deve ser chamada pelo Analisador sintático logo após o reconhecimento do nó.

A Tabela Sintática é armazenada em um arquivo texto onde são válidas as seguintes convenções:

- Cada subgrafo de um não-terminal inicia com um registro especial contendo o seu nome precedido do código 'C' (de *cabeça*) na primeira coluna. Por exemplo, o primeiro registro do subgrafo do não-terminal Parâmetros teria o conteúdo CPARAMETROS.

- Os nós dos grafos são introduzidos por registros contendo o identificador do nó precedido na primeira coluna pelo código 'T' para terminais e 'N' para não-terminais. Exemplos: NPARAMETROS e TCONSTANTE.

- Todo registro identificador de nó terminal ou não-terminal é seguido de um registro contendo os atributos número do nó, número do nó alternativo, número do nó sucessor e número da rotina semântica, nesta ordem. Estes registros não contêm a numeração absoluta dos nós, mas sim um número relativo ao primeiro nó de cada subgrafo de um não-terminal. Este esquema de numeração proporciona grande flexibilidade na codificação do grafo. O módulo de carregamento deve transformar estes números relativos em absolutos, quando da inicialização das estruturas internas. O número 0 (zero) é usado quando não há rotina semântica, nó alternativo ou nó sucessor.

```
CTEMPLATE
TDEFTIPOS
1    11   2    0
TCONSTANTE
2    8    9    0
NCLASSEAS
3    0    4    0
T;
4    5    3    0
TCLASSEPRINCIPAL
5    11   6    0
NCLASSEPRINCIPAL
6    0    7    0
T;
7    8    6    0
TCLASSEAS
8    5    3    0
NCONSTANTE
9    0    10   0
T;
10   8    9    0
TPARAMETROS
11   14   12   0
NPARAMETROS
12   0    13   0
T;
13   14   12   0
TESQUELETO
14   0    15   11
TFIMESQUELETO
15   16   0    0
NCOMANDO
16   0    15   0
CCOMANDO
TESCREVA
1    6    2    0
T(
2    0    3    0
NFATORCADEIA
3    0    4    57
T,
4    5    3    0
T)
5    0    0    0
TPARA
6    12   7    23
T
7    0    8    25
T_ID
8    10   17   24
TFIMPARA
9    11   7    26
T_VAZIO
10   0    0    0
NCOMANDO
11   0    9    0
TSE
12   0    13   58
NEXPRESSAO
13   0    16   45
TFIMSE
14   15   0    28
NCOMANDO
15   0    14   0
TENTAO
16   0    14   27
TFACA
17   0    9    0
CFATOR
TCLASSEVAZIA
1    5    2    0
T(
2    0    3    0
T_ID
3    0    4    40
T)
4    0    0    0
TPRIMEIRAOCORRENCIA
5    9    6    0
T(
6    0    7    0
T_ID
```

```
7    0    8    41
T)
8    0    0    0
TULTIMAOCORRENCIA
9    13   10   0
T(
10   0    11   0
T_ID
11   0    12   42
T)
12   0    0    0
TNAO
13   15   14   0
NFATOR
14   0    0    44
T(
15   18   16   0
NEXPRESSAO
16   0    17   0
T)
17   0    0    0
NFATORCADEIA
18   0    0    43
CFATORCADEIA
TPARAMETROATUAL
1    5    2    0
T(
2    0    3    0
T_ID
3    0    4    52
T)
4    0    0    0
TVALOR
5    11   6    0
T(
6    0    7    0
T_ID
7    0    8    53
T,
8    0    9    0
T_ID
9    0    10   54
T)
10   0    0    0
T_ID
11   12   0    55
T_CADEIA
12   0    0    56
CCONSTANTE
T_ID
1    0    2    16
T=
2    0    3    0
T_CADEIA
3    0    0    17
CCLASSEAS
T_ID
1    0    2    1
T=
2    7    3    2
T(
3    0    4    0
T_ID
4    0    5    3
T,
5    6    4    0
T)
6    0    0    0
TEXIGE
7    9    8    0
NEXIGE
8    0    0    0
T
9    0    0    0
CEXIGE
T(
1    0    2    0
TVALOR
2    3    8    4
TCLASSE
3    0    4    5
T_ID
```

```
4    0    5    6
T,
5    6    4    0
T;
6    7    2    0
T)
7    0    0    0
TUNICO
8    4    4    12
CCLASSEPRINCIPAL
T_ID
1    0    2    7
TEXIGE
2    4    3    0
NEXIGE
3    0    0    0
T
4    0    0    0
CPARAMETROS
T_CADEIA
1    0    2    8
T:
2    0    3    0
T_ID
3    0    4    9
T(
4    8    5    10
NARGUMENTO
5    0    6    0
T,
6    7    5    10
T)
7    0    8    0
T
8    0    0    15
CARGUMENTO
T_CADEIA
1    2    0    13
T_ID
2    0    3    14
T(
3    7    4    10
NARGUMENTO
4    0    5    0
T,
5    6    4    10
T)
6    0    7    0
T
7    0    0    15
CEXPRESSAO
NEXPRESSAOSIMPLES
1    0    2    0
T=
2    3    6    48
T>
3    4    6    49
T<
4    5    6    50
T
5    0    0    0
NEXPRESSAOSIMPLES
6    0    0    51
CEXPRESSAOSIMPLES
NTERMO
1    0    2    0
TOU
2    4    3    45
NTERMO
3    0    2    47
T
4    0    0    0
CTERMO
NFATOR
1    0    2    0
TE
2    4    3    45
NFATOR
3    0    2    46
T
4    0    0    0
F
```

# Apêndice C
## Autômato Finito do Analisador Léxico

A figura C.1 apresenta o Automato Finito usado no projeto do Analisador Léxico, que é módulo integrante do Gerador de Fontes. As seguintes convenções são válidas para a figura:

*L* : letras ['a'..'z', 'A'..'Z'];

*D* : dígitos ['0'..'9'];

*C* : todo o conjunto de caracteres;

*C - {x}* : todos os caracteres exceto os de *x*;

_ : sublinha ("underscore");

' : apóstrofe;

< : abre parêntese angular;

> : fecha parêntese angular;

! : exlamação;

*b* : espaço em branco;

*FDL* : fim de linha (CR ou CR LF);

*FDA* : fim de arquivo;

*ASn* : ação semântica de número *n*.

Ações Semânticas:

*AS1* : inicializa Átomo e IndÁtomo;

*AS2* : concatena o caractere lido no final de Átomo;

*AS3* : distingue identificador de palavra reservada e, no segundo caso, atualiza IndÁtomo;

*AS4* : concatena o caractere lido no final de Átomo;

*AS5* : determina a classe do átomo como Cadeia;

*AS6* : determina classe do átomo como FDA;

*AS7* : distingue símbolo especial de símbolo reservado e, no segundo caso, atualiza IndÁtomo.

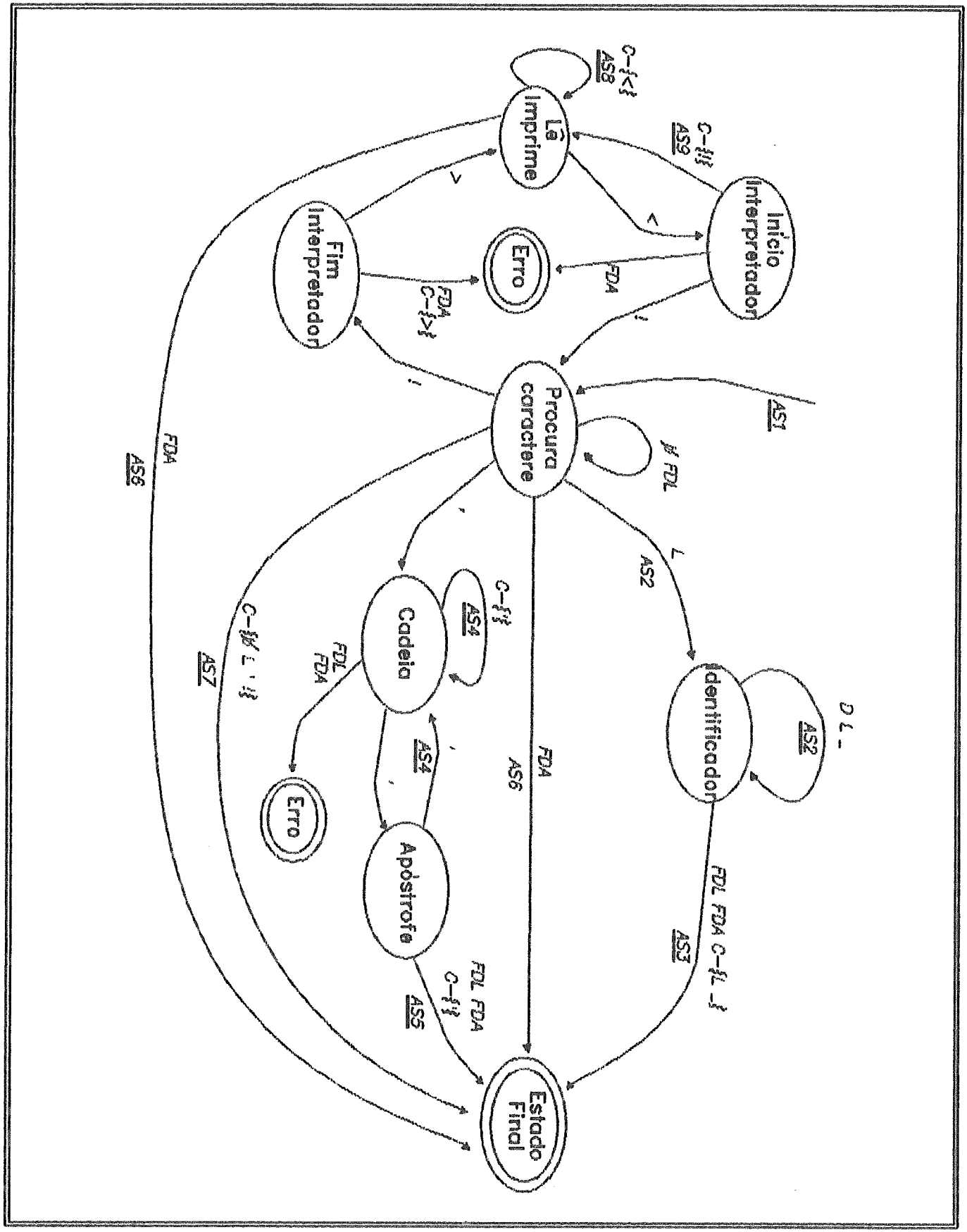



Fig. C.1 - Autômato Finito do Analisador Léxico

# Apêndice D
# Exemplo Completo

O objetivo aqui é mostrar como desenvolver um template de de forma incremental fazendo uso extensivo da linguagem LDT. Apresenta-se como exemplo a idéia de um template para um software de entrada de dados. Não houve preocupação de definir todas as partes do template, apenas o suficiente para que o exemplo fôsse completo no que diz respeito ao processo de definição e ao uso dos recursos existentes.

O template de nome EntradaDeDados (programa) é construído a partir dos templates DeclaraRegistro (trecho de programa) e FormataTela (procedimento). Por sua vez, o template FormataTela referencia o template ExibeStr (procedimento), sendo este último um exemplo de template sem parâmetros. Note-se que cada template possui apenas as definições dos tipos por ele manipulados, o que determina quais serão os parâmetros exigidos do usuário. A linguagem de programaãão é o (Turbo) Pascal.

As seguintes convenções são válidas na implementação do protõtipo do sistema:

- As sequências de caracteres <! e !> ativa e desativa o modo de interpretação, respectivamente. Isto permite escrever trechos mais longos do template (p.ex.: comentários) que serão copiados diretamente para o fonte gerado. A manipulação correta desta sinalização permite saltar linhas e colunas para o alinhamento do texto na saída.

- Toda referência a um template é feita através de seu código inserindo-o entre dois símbolos '@' (*arroba*) .

Template: ExibeStr
Descrição: Procedimento que exibe um texto em uma dada posição da tela do vídeo

```
(!
ESQUELETO !)
{ exibe  Texto na posicao X, Y da tela }
procedure ExibeStr (X, Y : byte;  Texto : string);
begin
gotoXY (X, Y);
write (Texto)
end;
(!
FIMESQUELETO !)
```

Template: FormataTela
Descrição: Procedimento que formata a tela do vídeo para edição de um registro.

```
(!
DEFTIPOS
  CONSTANTE
    NomeDoRegistro = 'DadosPessoais'
  CLASSEPRINCIPAL
    Campo EXIGE (VALOR X, Y, Titulo)

ESQUELETO !)
{ Procedure que formata tela para entrada dos dados do registro }
procedure  (! ESCREVA ('Tela',  NomeDoRegistro, ';')  !)

@ExibeStr@

begin      (! ESCREVA ('{ ', NomeDoRegistro, ' }')  !)
(!
PARA Campo FACA
   ESCREVA ('ExibeStr (', VALOR (X, Campo), ', ', VALOR (Y, Campo),
                                  ', ''', VALOR (Titulo, Campo), ''');' )  !)
(!
FIMPARA  !)
end;     (!
FIMESQUELETO  !)
```

Template: DeclaraRegistro
Descrição: Trecho de programa que declara o registro "DadosPessoais".

```
(!
DEFTIPOS
  CONSTANTE
    NomeDoRegistro = 'DadosPessoais'
  CLASSEPRINCIPAL
    Campo EXIGE ( VALOR Tipo )

ESQUELETO  !)
{ Declaracao do Registro }
Type
  (! ESCREVA ( 'Reg', NomeDoRegistro ) !) = Record
       (!
     PARA Campo FACA
       ESCREVA ( PARAMETROATUAL (Campo), ' : ', VALOR (Tipo, Campo), ';' ) !)
       (!
     FIMPARA
     ESCREVA ( 'end;' )  !)

Var
  (! ESCREVA (NomeDoRegistro, ' : Reg', NomeDoRegistro, ';')
FIMESQUELETO !)
```

Template: EntradaDeDados
Descrição: Programa (incompleto) para edição de dados de um registro

```
(!
DEFTIPOS
  CONSTANTE
    NomeDoPrograma = 'EntradaDeDados' ;
    NomeDoRegistro = 'DadosPessoais'
  CLASSEPRINCIPAL
    Campo  EXIGE ( VALOR Tipo, X, Y, Titulo )

ESQUELETO !)
{ programa exemplo para formatar uma tela de entrada de dados }
program  (! ESCREVA ( NomeDoPrograma, ';' )  !)
Uses
  CRT;

@DeclaraRegistro@
@FormataTela@

begin      { EntradaDeDados }
ClrScr;
(!
ESCREVA ( 'Tela', NomeDoRegistro, ';') !)
end. (!
FIMESQUELETO   !)
```

Parâmetros para o template "EntradaDeDados" (exemplo).

```
'Nome'    : Campo ( 'String[40]', '01', '10', 'Nome completo :' );
'Idade'   : Campo ( 'Integer',    '01', '12', 'Idade :'         );
'Salario' : Campo ( 'Real',       '01', '14', 'Salario bruto :' )
```

Programa fonte gerado para o template "EntradaDeDados" com os parâmetros acima.

```
{ programa exemplo para formatar uma tela de entrada de dados }

program  EntradaDeDados;
Uses
  CRT;

{ Declaracao do Registro }
Type
  RegDadosPessoais = Record
       Nome : String[40];
       Idade : Integer;
       Salario : Real;
       end;

Var
  DadosPessoais : RegDadosPessoais;



{ Procedure que formata tela para entrada dos dados do registro }
procedure  TelaDadosPessoais;

{ exibe  um Texto na posicao X, Y da tela do video }
procedure ExibeStr (X, Y : byte;  Texto : string);
begin
gotoXY (X, Y);
write (Texto)
end;

begin  { TelaDadosPessoais }
ExibeStr (01, 10, 'Nome completo :');
ExibeStr (01, 12, 'Idade :');
ExibeStr (01, 14, 'Salario bruto :');
end;



begin      { EntradaDeDados }
ClrScr;
TelaDadosPessoais;
end.
```

# REFERÊNCIAS BIBLIOGRÁFICAS

[1] ALLEN, Pat; BURNS, Alan. "Program Generation for Ada - A Case Study". Software - Pratice and Experience, 18(12), pp. 1125-1138, Dezembro 1988.

[2] BARNES, B. H.; BOLLINGER, T. B. "Making Reuse Cost-Effective". IEEE Software, pp. 13-24, Janeiro 1991.

[3] BIGGERSTAFF, Ted; RICHTER, Charles. "Reusability Framework Assesment, and Directions". IEEE Software, 4(2), pp. 41-48, Março 1987.

[4] BOLDYREFF, Cornélia. "Reuse, Software Concepts, Descriptive Methods and the Practitioner Project". Software Engineering Notes, 14(2), pp. 25-31, Abril 1989.

[5] CHEATHAM, T. E. "Reusability Through Program Transformation". IEEE Transactions on Software Engineering, SE-10(5), pp. 589-594, Setembro 1984.

[6] DE MARCO, Tom. "Análise Estruturada e Especificação de Sistema". Editora Campus, Rio de Janeiro, 1989.

[7] DUBINSKY, Ed et Al. "Reusability of Design for Large Software Systems: An Experment with the SETL Optimizer". In: Software Reusability - Concepts and Models, Addison-Wesley, New York, 1989.

[8] FÜRSTENAU, Eugênio. "Dicionário de Termos Técnicos Inglês-Português". Editora Globo, Vols. I e II, Porto Alegre, 1976.

[9] GEARY, K. "The practicalities of introducing large-scale softwa re re-use". Software Engineering Journal, pp. 172-176, Setembro 1988.

[10] GIRARDI, M. R.; PRICE, R. T. "Especificação de uma Ferramenta de Apoio à Reutilização de Software no Desenvolvimento Orientado a Objetos". IV SBES, pp. 231-244, Águas de São Pedro, 1990.

[11] HOROWITZ, Ellis; MUNSON, John B. "An Expansive View of Reusable Software". IEEE Transactions on Software Engineering, SE-10(5), Setembro 1984.

[12] JAMESON, Kevin W. "Model for the Reuse of Software Design Information". 11th International Conference of Software Engineering, pp. 205-216, Janeiro 1989.

[13] JONES, Gerald; PRIETO-DIAZ, Ruben. "Building and Managing Software Libraries". IEEE 12th Compsac, pp. 228-236, Outubro 1988.

[14] KATZ, Shmuel et Al. "PARIS: A System for Reusing Partially Interpreted Schemas". In: Software Reusability - Concepts and Models, Addison-Wesley, New York, 1989.

[15] LUBARS, Mitchell D. "Code reusability in the large versus code reusability in the small". Software Engeneering Notes, 11(1), pp. 21-28, Janeiro 1986.

[16] LUKER, P. A.; BURNS, A. "Program Generators and Generation Software". The Computer Journal, 29(4), pp. 315-321, 1986.

[17] MEYER, Bertrand. "Reusability: The Case for Object-Oriented Design". IEEE Software, 4(2), pp. 50-64, Março 1987.

[18] MOINEAU, Thierry et Al. "Towards a Generic and Extensible Reuse Environment". Proceedings of 2nd. International Workshop on Software Engineering & its Aplications, Toulouse, France, Dezembro 1989.

[19] MYERS, Glenford J. "Composite Structured Design". Van Nostrand, 1978.

[20] NEIGHBORS, James M. "DRACO: A Method for Engineering Reusable Software Systems". In: Software Reusability - Concepts and Models, Addison-Wesley, New York, 1989.

[21] NIELSEN, Jakob. "The Art of Navigating through Hypertext". Communications of the ACM, 33(3), pp. 297-310, Março 1990.

[22] NOTKIN, David; GRISWOLD, Willian G. "Extension and Software Development". 10th International Conference on Software Engineering, pp. 274-283, Abril 1988.

[23] PIMENTEL, Maria da Graça C. "Sistemas Hipertexto: Discussões e uma Proposta". XXII Congresso Nacional de Informática (Sucesu), pp. 128-134, São Paulo, 1989.

[24] PRESSMAN, R. S. "Software Engineering". McGraw-Hill, 1982.

[25] PRIETO-DIAZ, Ruben; FREEMAN, Peter. "Classifying Software for Reusability". IEEE Software, 4(1), pp. 6-16, Janeiro 1987.

[26] PRIETO-DIAZ, Ruben. "Domain Analysis for Reusability". IEEE 11th Compsac, pp. 23-29, Outubro 1987.

[27] PRIETO-DIAZ, Ruben. "Domain Analysis: An Introduction". Software Engineering Notes, 15(2), pp. 47-54, Abril 1990.

[28] RICH, Charles; WATERS, Richard C. "Automatic Programming: Myths and Prospects. IEEE Computer, pp. 40-51, Agosto 1988.

[29] ROCHA, Ana R. C. "Análise e Projeto Estruturado de Sistemas". Editora Campus. Rio de Janeiro, 1987.

[30] SETZER, Valdemar W.; MELO, Inês S. H. "A Construção de um Compilador". Editora Campus, Rio de Janeiro, 1983.

[31] STAMPS, David. "CASE Vs. 4GLs", Datamation, pp. 29-32, Agosto 1989

[32] TARUMI, Hiroyuki; AGUSA, Kiyoshi; OHNO, Yutaka. "A Programming Environment Supporing Reuse of Object-Oriented Software". 10th International Conference on Software Engineering, pp. 265-273, Abril 1988.

[33] TEIXEIRA, Mário M. R.; VELASCO, Flavio R. D. "Uma Ferramenta para Auxílio na Reutilização de Software". III SBES, pp. 253-268, Recife, 1989.

[34] TRACZ, Will. "Software Reuse Myths". ACM Sigsoft - Software Engineering Notes, 13(1), pp. 17-21, Janeiro 1988.

[35] TRACZ, Will. "Software Reuse Maxims". ACM Sigsoft - Software Engineering Notes, 13(4), pp. 28-32, Outubro 1988.

[36] "Turbo Pascal Database Toolbox". Borland International Inc., Scotts Valley, Version 4.0, 1987.

[37] "Turbo Pascal Reference Guide". Borland International Inc, Scotts Valley, Version 5.0, 1988.

[38] VO, Kiem-Phong. "IFS: A Tool to Build Application Systems". IEEE Software, pp. 29-36, Julho 1990.

[39] VOLPANO, D. M.; KIEBURTZ, R. B. "The Templates Aproach to Software Reuse". In: Software Reusability - Concepts and Models, Addison-Wesley, New York, 1989.

[40] WARTIK, S. P.; PENEDO, M. H. "Fillin: A Reusable Tool for Form-Oriented Software". IEEE Software, pp. 61-69, Março 1986.

[41] WEBSTER, Dallas E. "Mapping the Design Information Representation Terrain". IEEE Computer, pp 8-23, Dezembro 1988.

[42] WILEDEN, J. C. et Al. "Specification Level Interoperability". Proceedings of 12th International Conference Software Engineering, pp. 74-85, 1990.

[43] ZHU, Ming-Yuan. "Auto Star - A Software Development System". ACM Sigplan Notices, 24(3), pp. 31-45, Março 1989.