UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

SOFTWARE PARA GERAÇÃO DE CÓDIGO FONTE A PARTIR DO REPOSITÓRIO DA FERRAMENTA CASE SYSTEM ARCHITECT

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

MARCO AURÉLIO WEHRMEISTER

BLUMENAU, JUNHO/2001

2001/1-52

SOFTWARE PARA GERAÇÃO DE CÓDIGO FONTE A PARTIR DO REPOSITÓRIO DA FERRAMENTA CASE SYSTEM ARCHITECT

MARCO AURÉLIO WEHRMEISTER

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. José Roque Voltolini da Silva — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. José Roque Voltolini da Silva

_____

Prof. Marcel Hugo

_____

Prof. Everaldo Artur Grahl

AGRADECIMENTO

A meus pais Nelson Wehrmeister e Berta Gnewuch Wehrmeister pelo apoio, incentivo ao estudo, carinho e dedicação.

Ao professor José Roque Voltolini da Silva que me orientou neste trabalho com paciência, dedicação e sabedoria.

Aos meus irmãos e a minha namorada, pela compreensão, carinho e amizade.

Aos Amigos do Barney, que sempre me acompanharam desde o segundo grau até os dias de hoje, nos momentos sérios, nos estudos e principalmente nos momentos festivos.

Aos demais amigos e todas as pessoas que me apoiaram de uma forma direta ou indireta para a realização deste trabalho.

A todos muito obrigado.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho descreve o desenvolvimento e a construção de um protótipo de software capaz de gerar código fonte, a partir de um diagrama de classes armazenado no repositório da ferramenta CASE *System Architect*, para diversas linguagens, desde que exista uma especificação para as classes da linguagem. A especificação das classes de uma linguagem é escrita na metalinguagem LEL (Linguagem de Especificação de Linguagens), criada e descrita neste trabalho. No final deste trabalho é apresentado um estudo de caso contendo um diagrama de classes para o problema apresentado, bem como a especificação das linguagens CDL, Java e *Object Pascal*, escritas na metalinguagem LEL, utilizadas para a geração de código fonte.

ABSTRACT

This work describes the developing and the construction of a software prototype that is capable to produce source code, from a class diagram stored in the repository of CASE tool System Architect, for several languages, since exist a specification to the language classes. The language classes specification is write in the metalanguage LLS (Language for Language Specification), created and described in this work. A case study is presented, in the end of this work, containing a class diagram for the presented problem, as well as the specifications for CDL, Java and Object Pascal languages, written in the metalanguage LLS, used for source code generation.

1 INTRODUÇÃO

A capacidade de criar software não está acompanhando a evolução do hardware. À medida que o tempo passa, os softwares tornam-se cada vez mais complexos, possuindo grande quantidade de funcionalidades. Para Martin (1995), é necessário uma revolução industrial do software.

Segundo Martin (1995) a revolução a qual se refere, provavelmente virá das técnicas orientadas a objeto combinadas com ferramentas *Computer Aided Software Engineering* (CASE), geradores de código, programação visual e desenvolvimento baseado em repositórios. O objetivo deste conjunto de ferramentas é maximizar a reusabilidade de código, construindo e armazenando objetos complexos para posterior utilização, tornando o desenvolvimento de software mais rápido.

Os geradores de código são ferramentas que produzem código sem nenhum erro de sintaxe a partir de projetos, gráficos e especificações de alto nível (Martin, 1995). O código deve ser gerado a partir de tabelas de decisão, regras, diagramas de ação, diagramas de eventos, diagramas de transição de estado, representação de objetos e suas propriedades e relacionamentos, e assim por diante.

Para a construção de geradores de código, é comum utilizar os princípios e técnicas da construção de compiladores. Segundo Aho (1995), “...umas poucas técnicas básicas da construção de compiladores podem ser usadas para a construção de tradutores para uma ampla variedade de linguagens e máquinas...”.

Segundo Setzer (1986), um compilador é um programa de computador que tem a finalidade de traduzir ou converter um programa escrito em uma linguagem fonte para um programa escrito em uma linguagem destino. Existem duas partes na compilação, a parte da análise que divide o programa fonte em partes constituintes e cria uma representação intermediária, e a parte da síntese que constrói o programa alvo desejado a partir da representação intermediária (Aho, 1995).

Tendo em vista a necessidade dos desenvolvedores de software de possuir ferramentas para a geração de código fonte, a partir de especificações modeladas em ferramentas CASE, o presente trabalho apresenta um protótipo de software que permite a geração de código fonte a partir do repositório da ferramenta CASE System Architect (Popkin, 2001). O software

permite gerar código fonte para diversas linguagens de programação, desde que exista um arquivo externo no formato texto utilizando uma notação *Backus Naur Form* (BNF) estendida para especificar a linguagem.

1.1 OBJETIVOS

O objetivo principal deste trabalho foi desenvolver um protótipo de software para a geração de código fonte em diversas linguagens programação, desde que sua especificação esteja em um arquivo texto utilizando a notação BNF estendida, a partir do repositório da ferramenta CASE *System Architect*.

1.2 ORGANIZAÇÃO DO TEXTO

No capitulo 1 são apresentados a introdução, bem como os objetivos deste trabalho.

No capitulo 2 é apresentada a fundamentação teórica do trabalho. São apresentados ambientes de descrição e transformação de dados, considerações sobre linguagens de programação, compiladores e orientação a objetos. São apresentadas ainda informações sobre o repositório da ferramenta CASE *System Architect* e ainda uma breve descrição das linguagens CDL, Java e *Object Pascal*.

No capítulo 3 é apresentado o desenvolvimento do trabalho, incluindo a descrição da especificação e da implementação do protótipo.

Por fim no capítulo 4 é apresentada a conclusão do trabalho e também sugestões para a extensões deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados ambientes de descrição e transformação de dados, considerações sobre linguagens de programação, compiladores, orientação a objetos. Serão apresentadas ainda informações sobre o repositório da ferramenta CASE *System Architect* e ainda uma breve descrição das linguagens CDL, Java e *Object Pascal*.

2.1 AMBIENTES DE DESCRIÇÃO E TRANSFORMAÇÃO DE DADOS

Consideram-se ambientes de descrição e transformação de dados, todos aqueles softwares que possuem a capacidade de ler dados de algum local específico, para que possam ser convertidos ou transformados em outro formato, que não o seu formato de origem.

Pode-se citar ambientes de descrição e transformação de dados a *Extensible Stylesheet Language* (XSL), que utiliza a *Extensible Marckup Language* (XML). Como outro ambiente de transformação de dados pode-se citar o de Kramel (2000). A seguir serão dadas maiores informações sobre os referidos exemplos, com a finalidade de conhecer o seu funcionamento.

2.1.1 EXTENSIBLE STYLESHEET LANGUAGE

A *Extensible Stylesheet Language* (XSL), segundo Day (2000), é uma linguagem de especificação de estilos para documentos marcados utilizando a *Extensible Marckup Language* (XML).

A XML é um subconjunto da *Standard Generalized Markup Language* (SGML), que é um padrão internacional para definir descrições de estruturas e conteúdo de diferentes tipos de documentos eletrônicos (W3C, 2000a). Sua principal vantagem é permitir que SGML genérico seja disponibilizado, recebido e processado através da internet.

Segundo Flynn (2000), XML não é uma simples linguagem de marcação predefinida, mas sim uma metalinguagem – uma linguagem que descreve outras linguagens – que permite definir sua própria marcação.

Documentos XML são constituídos de unidades de armazenamento chamadas entidades, que geralmente são compostos de marcações de início e fim de *tag* e o dado propriamente dito entre as referidas marcações. Um elemento pode opcionalmente possuir um ou mais atributos, o qual é o par nome-valor separado pelo sinal de igual. No quadro 1 pode-se ver um exemplo de um documento XML contendo elementos e atributos de elementos.

Quadro 1 – DOCUMENTO XML

```
1   <?xml version="1.0"?>
2   <CATALOG>
3     <CD Price=“10.90”>
4       <TITLE>Empire Burlesque</TITLE>
5       <ARTIST>Bob Dylan</ARTIST>
6       <COUNTRY>USA</COUNTRY>
7       <COMPANY>Columbia</COMPANY>
8       <YEAR>1985</YEAR>
9     </CD>
10    <CD Price=“10.90”>
11      <TITLE>Greatest Hits</TITLE>
12      <ARTIST>Dolly Parton</ARTIST>
13      <COUNTRY>USA</COUNTRY>
14      <COMPANY>RCA</COMPANY>
15      <PRICE>9.90</PRICE>
16      <YEAR>1982</YEAR>
17    </CD>
18  </CATALOG>
```

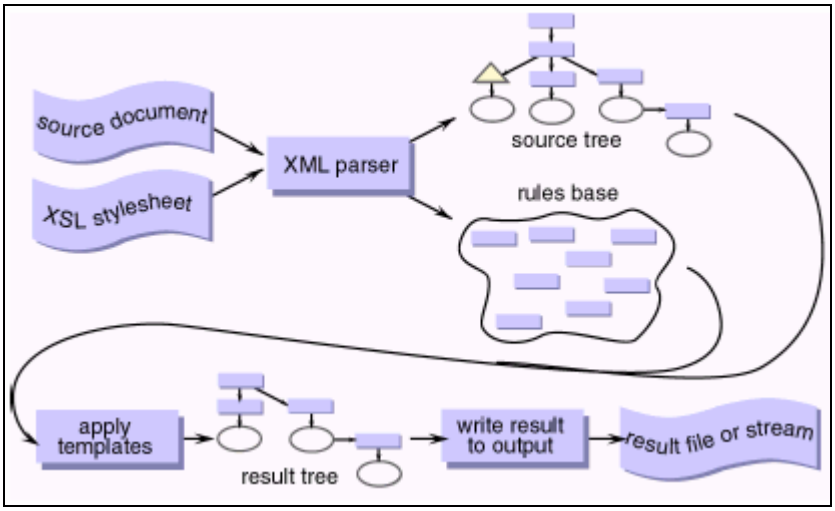
Um documento XML nada mais é que um arquivo texto contendo uma série de elementos, sendo que cada qual pode conter um ou mais sub-elementos e assim sucessivamente (Flynn, 2000). No quadro 1 pode-se identificar os marcadores de início (tudo o que está entre o sinal de menor e o sinal de maior) e fim de *tag* (análogo ao marcador de inicio de *tag*, diferindo apenas pelo fato de haver a barra após o sinal de menor), bem como os dados (que estão entre os *tags* de inicio e fim de elemento). Vale ressaltar também que nas linhas 3 e 10 existe o atributo “Price” com o valor de 10,90 para o elemento “CD”.

Segundo W3C (2000b) a XSL consiste de duas partes, a primeira é uma linguagem para transformar documentos XML e a segunda um vocabulário XML para especificar semânticas de formatação. Um estilo de formatação XSL especifica a apresentação de uma classe de documentos XML descrevendo como uma instância dessa classe é transformada em um outro documento XML que utiliza o vocabulário de formatação.

Segundo Day (2000) o tradutor XML converte um documento fonte em uma árvore fonte, então é feita a leitura dos estilos de formatação XSL e organizadas as regras para

pesquisá-las de forma eficiente. Em seguida o processador XSL percorre a árvore fonte partindo da raiz, tentando combinar cada nodo com sua regra de formatação. Se esta combinação é obtida, a regra é combinada na árvore de resultado e o processo continua até que a árvore fonte seja visitada completamente. No final, o processador XSL percorre a árvore resultado e copia o que ele encontra para um arquivo de saída. A fig. 1 ilustra este processo.

Figura 1 - PROCESSO DE TRANSFORMAÇÃO DE UM DOCUMENTO XML



Fonte: Day (2000)

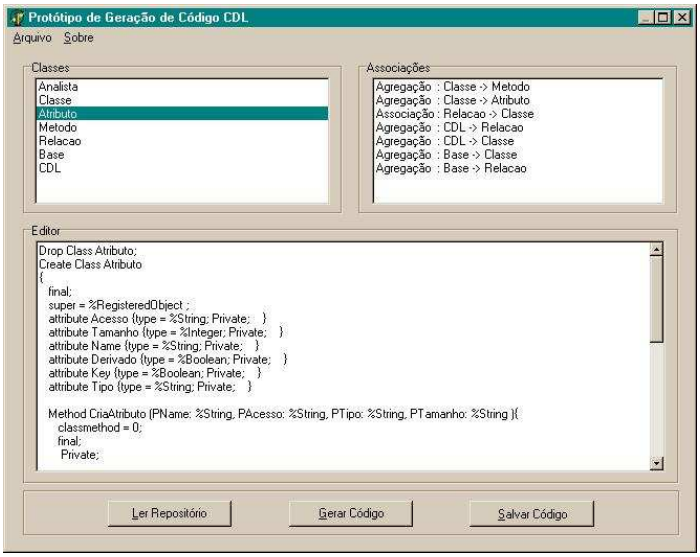
2.1.2 AMBIENTE DE TRANSFORMAÇÃO DE KRAMEL

O ambiente de transformação de dados resultante do trabalho de conclusão de curso apresentado em Kramel (2000), consiste num protótipo que permite a geração de código *Class Definition Language* (CDL) para o banco de dados Caché (Intersystems, 2000) a partir de um diagrama de classes armazenado no repositório da ferramenta CASE System Architect.

Segundo Kramel (2000) o analista de sistemas é responsável por configurar o diretório onde se encontra a base de dados do repositório. Após esta configuração ele pode iniciar o processo de geração de código, que está dividida em três tarefas: leitura da base de dados, geração de código fonte e por último salvar o código que foi gerado. A fig. 2 mostra a janela do protótipo apresentado por Kramel (2000).

Um das limitações do protótipo segundo Kramel (2000) é a geração de código fonte apenas para CDL, uma vez que a geração de código fonte foi implementada de forma “fixa” dentro do protótipo. Outra limitação do protótipo é que durante a execução do software só é possível ler o repositório da ferramenta CASE System Architect de apenas um lugar. Quando se deseja ler outro repositório (localizado em um local diferente do inicial), deve-se sair do protótipo e configurar o caminho para o referido repositório e tornar a executá-lo.

Figura 2 – GERAÇÃO E CÓDIGO FONTE CDL PELO PROTÓTIPO DE KRAMEL



Fonte: Kramel (2000)

2.2 LINGUAGENS DE PROGRAMAÇÃO

Ferreira (1985) define linguagem como “o uso da palavra articulada ou escrita como meio de expressão e comunicação entre pessoas”. Segundo Menezes (1998), esta definição não é suficientemente precisa para permitir o desenvolvimento matemático de uma teoria sobre linguagens.

Segundo Aho (1995) uma linguagem de programação pode ser definida pela descrição da aparência de seus programas (a sintaxe da linguagem) e do que os mesmos significam (a semântica da linguagem). Uma linguagem de programação segundo Menezes (1998) pode ser vista de duas formas: como uma entidade livre de contexto, ou seja, sem qualquer significado associado; ou como uma entidade juntamente com uma interpretação de seu significado.

A seguir serão apresentados conceitos básicos referentes às linguagens de programação de um modo geral.

2.2.1 CONCEITOS BÁSICOS

Um alfabeto é um conjunto finito de símbolos, sendo que um símbolo é uma entidade abstrata básica a qual não é definida formalmente (Menezes, 1998). Como exemplo pode-se citar o alfabeto romano {a, b, c, ..., z}. Para Lewis (2000) qualquer objeto pode estar em um alfabeto. Do ponto de vista formal, um alfabeto é simplesmente um conjunto finito de qualquer tipo. Cabe ainda ressaltar que um conjunto vazio também é considerado um alfabeto.

Segundo Lewis (2000) uma palavra em um alfabeto é uma seqüência finita de símbolos. Em vez de escrever palavras com parênteses e virgulas, com escreve-se outras seqüências, simplesmente justapõe-se os símbolos. Exemplificado, em vez de escrever a palavra “melancia” na forma (m, e, l, a, n, c, i, a), escrevem-se os símbolos justapostos na forma melancia. A palavra de um só símbolo é o próprio símbolo. Segundo Menezes (2000) a palavra vazia, representada pelo símbolo ε, é uma palavra sem símbolo. Seja um alfabeto representado por $\sum$, então $\sum^*$ denota o conjunto de todas as palavras possíveis sobre $\sum$. Analogamente, $\sum^+$ representa o conjunto de todas as palavras possíveis sobre $\sum$ excetuando-se a palavra vazia, ou seja, $\sum^+ = \sum^* - \{\varepsilon\}$.

Segundo Aho (1995) o comprimento de uma palavra w, usualmente escrita |w|, é o número de ocorrências de símbolos que compõe a palavra w. Por exemplo, banana é uma palavra de comprimento seis. A cadeia vazia (ε) tem o comprimento zero.

O prefixo de uma palavra é qualquer seqüência de símbolos inicial da palavra. Analogamente o sufixo é qualquer seqüência de símbolos final da palavra. Uma *subpalavra* de uma palavra é qualquer seqüência de símbolos contígua da palavra, sendo assim, qualquer prefixo e sufixo de uma palavra é uma *subpalavra* (Menezes, 1998).

Segundo Menezes (1998) uma linguagem formal é um conjunto de palavras sobre um alfabeto. Seja o alfabeto $\sum = \{a, b\}$:

a) o conjunto vazio e o conjunto formado pela palavra vazia são linguagens sobre $\sum$ (Obviamente $\{\} \neq \{\varepsilon\}$);
b) o conjunto de palíndromos (palavras que tem a mesma leitura da esquerda para a direita e vice-versa) sobre $\sum$ é um exemplo de linguagem infinita. Assim, ε, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, ... são palavras desta linguagem.

A concatenação é uma operação binária, definida sobre uma linguagem, a qual associa a cada par de palavras uma palavra formada pela justaposição da primeira com a segunda. Uma concatenação é denotada pela justaposição dos símbolos que representam as palavras componentes (Menezes, 1998). Supondo as palavras v, w, t a operação de concatenação satisfaz às seguintes propriedades: associatividade, v(wt) = (vw)t e elemento neutro à esquerda e à direita, εw = w = wε. A concatenação sucessiva de uma palavra (com ela mesma), representada na forma de um expoente w^n onde w é uma palavra e n indica o número de concatenações sucessivas, é definida indutivamente a partir da concatenação binária como segue: quando $w \neq \varepsilon$, $w^0 = \varepsilon$ e $w^n = w^{n-1}w$, para $n > 0$; quando $w = \varepsilon$, $w^n = \varepsilon$, para $n > 0$; w^n é indefinida para $n = 0$.

Segundo Neto (1987) para que seja possível a correta compreensão das linguagens de programação, é essencial que estas sejam decritas de maneira completa e isenta de ambigüidades. Uma descrição desta natureza só é viável na prática através do uso de notações formais. Para a descrição formal de linguagens de programação, são utilizadas notações matemáticas formais. Uma maneira de formalizar uma linguagem de programação é especificar leis de formação que definem de maneira rigorosa o modo de formar textos corretos. Esta formalização recebe o nome de gramática.

Segundo Menezes (1998) uma gramática é uma quádrupla ordenada G = (V, T, P, S) onde:

a) V é o conjunto finito de símbolos variáveis ou não-terminais;
b) T é o conjunto finito de símbolos terminais, disjunto de V;
c) P é o conjunto finito de pares denominado regras de produção, tal que a primeira componente é a palavra de $(V \cup T)^+$ e a segunda componente é a palavra de $(V \cup T)^*$;

d) S é o elemento de V denominado símbolo de partida.

Segundo Aho (1995) os terminais são os símbolos básicos e a partir dos quais as cadeias são formadas. A palavra *token* é considerada um sinônimo de terminal quando se referem a gramáticas para linguagens de programação.

Os não-terminais são variáveis sintáticas que denotam cadeias de caracteres. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática. Também impõem uma estrutura hierárquica na linguagem que é útil tanto para a análise sintática quanto para a tradução (Aho, 1995). O símbolo de partida de uma gramática é um não-terminal, cujo conjunto de cadeias que o mesmo denota é a linguagem definida pela gramática.

Segundo Aho (1995) as produções de uma gramática especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Uma regra de produção (α, β) segundo Menezes (2000) é representada por $\alpha \rightarrow \beta$. As regras de produção definem as condições de geração das palavras de uma linguagem. Uma seqüência de regras de produção da forma $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, ..., $\alpha \rightarrow \beta_n$, (mesma componente do lado esquerdo) pode ser abreviada como uma única produção na forma $\alpha \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$. A aplicação de uma regra de produção é denominada derivação de uma palavra. A aplicação sucessiva de regras de produção permite derivar as palavras da linguagem representadas pela gramática.

Seja G = (V, T, P, S) uma gramática, uma derivação é um par da relação denotada por $\Rightarrow$ como domínio em $(V \cup T)^+$ e contra-domínio em $(V \cup T)^*$ (Menezes, 1998). Um par (α, β) é representado $\alpha \Rightarrow \beta$. A relação $\Rightarrow$ é indutivamente definida pois:

a) para toda a produção da forma $S \rightarrow \beta$ (a primeira componente é o símbolo inicial de G) tem-se $S \Rightarrow \beta$;
b) para todo o par $\alpha \Rightarrow \beta$, onde $\beta = \beta_u\beta_v\beta_w$, se $\beta_v \rightarrow \beta_t$ é regra de P então $\beta \Rightarrow \beta_u\beta_t\beta_w$.

Portanto, uma derivação é a substituição de uma *subpalavra* de acordo com uma regra de produção. Quando for desejado explicitar a regra de produção $p \in P$ que define a derivação $\alpha \Rightarrow \beta$, a notação $\alpha \Rightarrow^p \beta$ é usada. Sucessivos passos de derivação são definidos como segue:

a) $\Rightarrow^*$ representa zero ou mais passos de derivações sucessivas;

b) $\Rightarrow^+$ representa um ou mais passos $\Rightarrow^i$ representa exatos passos de derivações sucessivas, onde i é um número natural.

Segundo Menezes (1998) uma gramática é considerada um formalismo de geração, pois permite derivar (“gerar”) todas as palavras da linguagem que representa. Seja G = (V, T, P, S) uma gramática, a linguagem gerada por G, denotada por L(G), é composta por todas as palavras de símbolos terminais a partir do símbolo inicial S, ou seja, $L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$.

2.2.2 TÉCNICAS DE ESPECIFICAÇÃO DE LINGUAGENS

Segundo Kowaltowski (1983) uma linguagem de programação deve ter uma definição precisa antes de ser implementada. Esta definição consiste em duas partes: a definição da sintaxe e a definição da semântica da linguagem.

Existem métodos formais para especificação completa tanto da sintaxe como da semântica de linguagens de programação. Entretanto, as descrições que usam os métodos atualmente existentes tendem, em geral, ser muito complexas (Kowaltowski, 1983). Na prática, usam-se métodos formais para especificar a maior parte dos aspectos sintáticos da linguagem, sendo que algumas restrições sintáticas adicionais, bem como a semântica, são especificadas por meios informais.

Segundo Menezes (1998) os formalismos usados para o tratamento sintático de linguagens de programação podem ser classificados nos seguintes tipos:

a) operacional: define-se um autômato ou uma máquina abstrata, baseada em estados, em instruções primitivas e na especificação de como cada instrução modifica cada estado. Uma máquina abstrata deve ser suficientemente simples para não permitir dúvida sobre a execução de seu código. Também é dito um formalismo reconhecedor, no sentido que permite uma análise de uma dada entrada para verificar se é “reconhecida” pela máquina. As principais máquinas são autômato finito, autômato com pilha e máquina de Turing;
b) axiomático: associam-se regras às componentes da linguagem. As regras permitem afirmar o que será verdadeiro após a ocorrência de cada cláusula considerando o

que era verdadeiro antes da ocorrência. A abordagem axiomática é sobre gramáticas (regulares, livres de contexto, sensíveis ao contexto e irrestritas). Uma gramática também é dita um formalismo *gerador* no sentido em que permite verificar se um determinado elemento da linguagem é "gerado";

c) denotacional: também é denominado formalismo funcional. Define-se uma função que caracteriza o conjunto de palavras admissíveis na linguagem. Em geral, trata-se de uma função construída a partir de funções elementares de forma composicional (horizontalmente) no sentido em que a linguagem denotada pela função pode ser determinada em termos de suas funções componentes.

Segundo Lewis (2000) um método alternativo de especificar uma linguagem é descrever como um espécime genérico na linguagem é produzido. Esses geradores de linguagem não são algoritmos, uma vez que não são projetados para responder a perguntas e não são completamente explícitos sobre o que fazer (por exemplo, como escolher qual palavra a, ab ou abb deva ser escrita), mas são meios importantes e úteis de representar linguagens.

A seguir serão dadas informações sobre linguagens livres de contexto que são linguagens geradas a partir de gramáticas livres de contexto.

2.2.3 LINGUAGENS LIVRES DE CONTEXTO

Uma linguagem livre de contexto é definida a partir de uma gramática livre de contexto (Menezes, 1998). Uma gramática livre de contexto G é uma gramática do tipo G = (V, T, P, S) com a restrição de que qualquer regra de produção de P é da forma $A \rightarrow \alpha$, onde A é uma variável de V e α uma palavra de $(V \cup T)^*$, isto é, uma gramática livre de contexto é uma gramática onde o lado esquerdo das produções contém exatamente uma variável (não-terminal).

Segundo Aho (1995), uma gramática livre de contexto é uma notação amplamente aceita para especificar a sintaxe de uma linguagem, além do que pode ser usada como auxílio para guiar a tradução de programas.

Segundo Menezes (1998) o nome “livre de contexto” se deve ao fato de um não-terminal A derivar α sem depender (“livre”) de uma análise dos símbolos que antecedem ou sucedem A (“contexto”) na palavra que esteja sendo derivada.

Segundo Kowaltowski (1983) “...a notação *Bakus Naur Form* (BNF) é usada para representar uma classe de definições que em teoria de linguagens formais são chamadas de gramáticas livres de contexto”.

2.2.3.1 BAKUS NAUR FORM (BNF)

Segundo Howe (1997) a *Bakus Naur Form* originalmente chamada de *Bakus Normal Form*, é uma meta-sintaxe formal usada para expressar gramáticas livres de contexto. A BNF é uma das notações meta-sintáticas mais comumente utilizada para especificação da sintaxe de linguagens de programação, conjuntos de comandos, entre outros. A BNF é largamente utilizada para descrição de linguagens.

Os meta-símbolos utilizados na BNF são:

a) “::=” que indica é definido por;
b) “|” que indica “ou”;
c) “<” e “>” que indica não-terminais.

Existe uma grande variedade de extensões da BNF. A *Extended Bakus Naur Form* é uma delas, a qual introduziu os colchetes “[...]” para cercar os itens opcionais, adicionou também os sufixos “*” para o fechamento de Kleene e “+” para indicar um ou mais itens, as chaves “{...}” para confinar uma lista de alternativas (Howe, 1997).

O quadro 2 apresenta um trecho de uma definição em BNF.

Quadro 2 – EXEMPLOS DE DEFINIÇÃO UTILIZANDO A BNF

```
<identifier> ::= <letter> { <letter> | <digit> }*
<if_statement> ::= if <boolean_expression> then <statement_sequence> [ else
                    <statement_sequence> ] end if ;
```

2.2.3.2 ÁRVORE DE DERIVAÇÃO

Uma árvore de derivação ou árvore gramatical mostra, graficamente, como o símbolo de partida de uma gramática deriva uma cadeia da linguagem. Se um não-terminal A possui

uma produção A $\rightarrow$ XYZ, então uma árvore de derivação pode ter um nó rotulado A, com três filhos rotulados X, Y, Z da esquerda para a direita (Aho, 1995) .

Segundo Menezes (1998) em algumas aplicações como compiladores e processadores de texto, freqüentemente é conveniente representar a derivação de palavras na forma de árvore, partindo do símbolo inicial como a raíz, e terminando em folhas de terminais.

Dada uma gramática livre de contexto, uma árvore gramatical possui as seguintes propriedades:

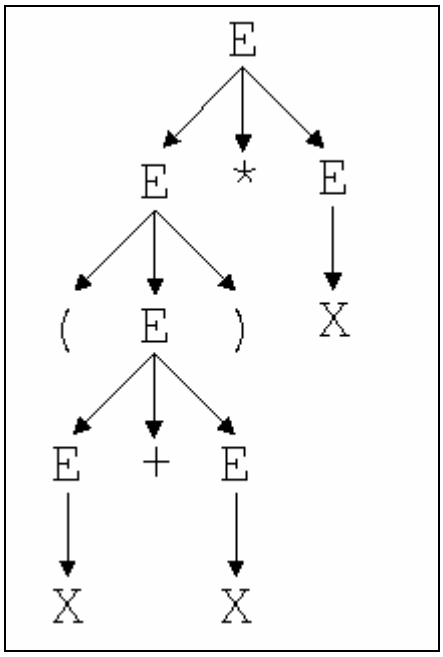
a) a raiz é rotulada pelo símbolo de partida;
b) cada folha é rotulada por um terminal ou por ε;
c) cada nó interior é rotulado por um não-terminal;
d) se a é um não-terminal rotulando algum nó interno e X_1, X_2, ...X_n são os rótulos dos filhos deste nó, da esquerda para a direita, então A $\rightarrow$ X_1 X_2 X_n é uma produção. Com X_1, X_2, ...X_n figurando no lugar de símbolos que sejam terminais e não-terminais. Como um caso especial, se A $\rightarrow$ ε, então um nó rotulado A deve possuir um único filho rotulado ε.

A fig. 3 representa a árvore de derivação para a cadeia (x+x)*x da gramática G = ({E}, {+,*,(,),x}, P, E), onde P = {E $\rightarrow$ E+E | E*E | (E) | x}.

Segundo Lewis (2000) uma derivação mais a esquerda existe em cada árvore de derivação e pode ser obtida começando do rótulo raiz A, substituindo-se repetidamente o não-terminal mais à esquerda na cadeia atual, de acordo com a regra sugerida pela árvore de derivação.

Ainda segundo Lewis (2000) uma derivação mais a direita é aquela que não precede qualquer derivação. Ela é obtida da árvore de derivação expandindo sempre o não-terminal mais à direita na cadeia atual.

Figura 3 – EXEMPLO DE ÁRVORE DE DERIVAÇÃO



Fonte: Menezes (1998)

2.2.3.3 AMBIGUIDADE

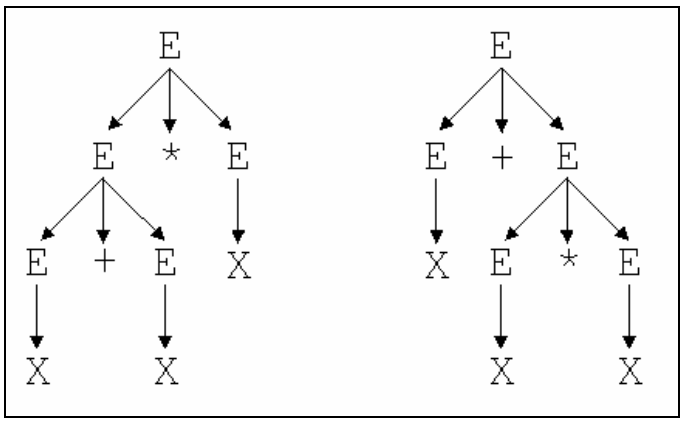
Eventualmente uma mesma palavra pode ser associada a duas ou mais árvores de derivação, determinando uma ambigüidade (Menezes, 1998). Em muitas aplicações como, por exemplo, no desenvolvimento e otimização de alguns tipos de algoritmos de reconhecimento, é conveniente que a gramática usada não seja ambígua. Entretanto nem sempre é possível eliminar ambigüidades. Na realidade é fácil definir linguagens para as quais qualquer gramática livre de contexto é ambígua (Menezes, 1998).

Segundo Aho (1995) uma gramática que produza mais de uma árvore de derivação para alguma sentença é dita ambígua. Uma gramática ambígua é aquela que produz mais de uma derivação à esquerda, ou à direita, para a mesma sentença.

A fig. 4 apresenta um exemplo de árvore de derivação ambígua para a cadeia x+x*x da gramática G = ({E}, {+,*,(,),x}, P, E), onde P = {E $\rightarrow$ E+E | E*E | (E) | x}.

Para algumas aplicações, consideram-se métodos através dos quais pode-se usar gramáticas ambíguas, juntamente com regras de inambiguidade que “descartam” árvores de derivação indesejáveis, deixando somente uma árvore para cada sentença (Aho, 1995).

Figura 4 – EXEMPLO DE ÁRVORE DE DERIVAÇÃO AMBÍGUA



Fonte: Menezes (1998)

2.2.3.4 RECURSÃO À ESQUERDA

Segundo Aho (1995) uma gramática é recursiva à esquerda se possui um não-terminal A tal que exista uma derivação $A \Rightarrow^{+} A\alpha$ para alguma cadeia α.

Os métodos de análise sintática *top-down* não podem processar gramáticas recursivas à esquerda pelo fato do algoritmo entrar em *loop* infinito. Conseqüentemente uma transformação que elimine a recursão à esquerda é necessária (Aho, 1995).

Ainda segundo Aho (1995) não importa quantas produções-A existam, pode-se eliminar a recursão imediata das mesmas pela seguinte técnica:

a) primeiro agrupa-se as produções-A como $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$ onde nenhum β_i começa por um A;
b) após, substitui-se as produções-A por $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$ e $A' \rightarrow \alpha_1 A' \mid A\alpha_2 A' \mid ... \mid A\alpha_m A' \mid \varepsilon$.

Como exemplo tem-se a gramática do quadro 3, representada na notação BNF. No quadro 4 é apresentada a gramática após ser eliminada a recursão à esquerda.

Quadro 3 – EXEMPLO DE GRAMÁTICA COM RECURSÃO À ESQUERDA

```
<E> ::= <E> + <T> | <T>
<T> ::= <T> * <F> | <F>
<F> ::= (<E>) | id
```

Fonte: Aho (1995)

Quadro 4 – EXEMPLO DE GRAMÁTICA LIVRE DA RECURSÃO À ESQUERDA

```
<E> ::= <T><E'>
<E'> ::= + <T><E'> | ε
<T> ::= <F><T'>
<T'> ::= *<F><T'> | ε
<F> ::= (<E>) | id
```

Fonte: Aho (1995)

2.2.3.5 FATORAÇÃO À ESQUERDA

A fatoração à esquerda é uma transformação útil para a criação de uma gramática adequada à análise sintática preditiva. A idéia básica está em, quando não estiver claro qual das duas produções alternativas usar para expandir um não-terminal A, deve-se reescrever as produções-A e postergar a decisão até que se tenha visto o suficiente da entrada para realizar a escolha certa (Aho, 1995).

Segundo Aho (1995) se $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ são duas produções-A, e a entrada começa por uma cadeia não vazia derivada a partir de α, então não se sabe se vai expandir A em $\alpha\beta_1$ ou $\alpha\beta_2$. Entretanto pode-se postergar a decisão expandindo A para αA'. Então após enxergar a entrada derivada a partir de α expande-se A' em β_1 ou em β_2. O quadro 5 apresenta uma gramática com o problema do "else-vazio" e o quadro 6 apresenta a solução utilizando a fatoração à esquerda.

Quadro 5 – EXEMPLO DE GRAMÁTICA COM PROBLEMA DO "ELSE-VAZIO"

```
<cmd> ::= if <expr> then <cmd> | if <expr> then <cmd> else <cmd> | a
<expr> ::= b
```

Fonte: Aho (1995)

Quadro 6 – EXEMPLO DE GRAMÁTICA LIVRE DO PROBLEMA DO "ELSE-VAZIO"

```
<cmd> ::= if <expr> then <cmd><cmd2> | a
<cmd2> ::= else <cmd> | ε
<expr> ::= b
```

Fonte: Aho (1995)

2.3 COMPILADORES

Segundo Aho (1995) um compilador é um programa que lê um programa escrito em uma linguagem (a linguagem fonte) e o traduz num programa equivalente numa outra linguagem (a linguagem alvo). Como importante parte deste processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte.

Segundo Neto (1987) pode-se dizer que um tradutor, como pode ser classificado um programa desta natureza, efetua uma conversão entre duas linguagens: dada uma linguagem de entrada, denominada linguagem fonte, o compilador tem a função de converter automaticamente textos escritos nesta linguagem (textos-fonte) em textos correspondentes equivalentes (textos-objeto), apresentados em uma linguagem de saída denominada, linguagem objeto. Esta conversão deve preservar, no texto-objeto, todas as informações, contidas no texto-fonte, que contribuam para a execução correta do programa.

Existem duas partes na compilação: a análise e a síntese. A parte de análise divide o programa fonte nas partes constituintes e cria uma representação intermediária do mesmo. A de síntese constrói o programa alvo desejado, a partir da representação intermediária. Das duas, a síntese requer as técnicas mais especializadas (Aho, 1995).

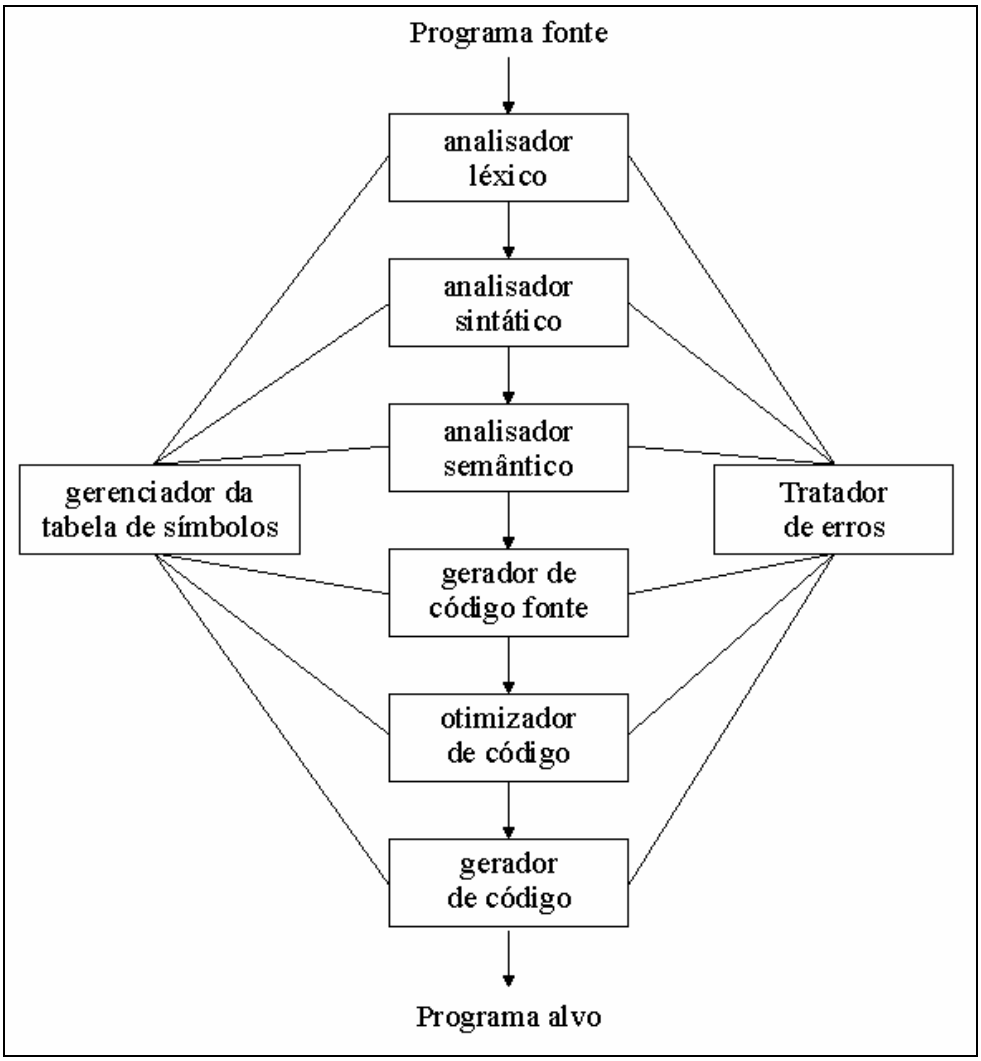
Segundo Watt (1993) dentro de um compilador, o programa fonte é submetido a uma série de transformações antes de um programa objeto ser gerado. Estas transformações são chamadas de fases. Na fig. 5 pode ser vista uma composição típica de um compilador.

As fases de análise léxica, sintática e semântica formam o núcleo da parte de análise do compilador. Analogamente, as fases do gerador de código intermediário, otimizador de código e gerador de código fazem parte do núcleo da parte de síntese do compilador. As atividades de gerenciamento da tabela de símbolos e a de manipulação de erros que são mostradas interagindo com as seis fases, serão informalmente chamadas de “fases” (Aho, 1995).

No desenvolvimento de um compilador, normalmente deseja-se decompor o compilador em módulos, sendo que cada módulo é responsável por uma fase em particular. A forma de como um compilador é desenvolvido afeta sua modularidade, seu requerimento de

tempo e espaço, e o número de passos sobre o programa que está sendo compilado (Watt, 1993).

Figura 5 – COMPOSIÇÃO TÍPICA DE UM COMPILADOR



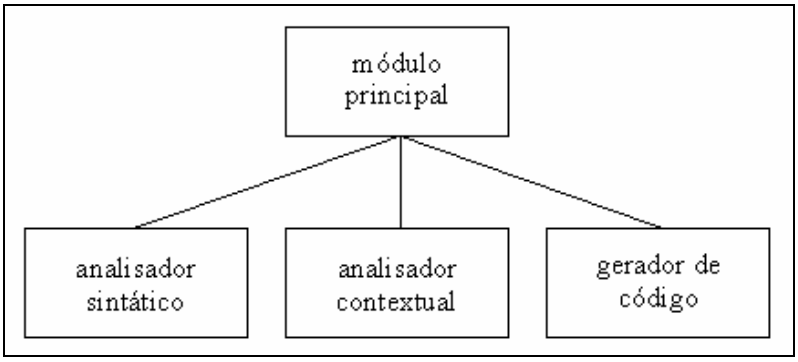
Fonte: Aho (1995)

Segundo Watt (1993) um passo é uma passagem completa de um programa fonte, ou de uma representação intermediária do programa fonte. Um compilador de uma passagem faz uma única passagem do programa fonte, já o compilador de múltiplas passagens, como o próprio nome sugere, faz várias passagens no programa fonte.

Um compilador de múltiplas passagens consiste num módulo principal que dirige outros três módulos inferiores, o analisador sintático, o analisador de contexto e o gerador de código. Primeiro o módulo principal chama o analisador sintático que lê o programa fonte

analisando-o e construindo uma árvore sintática. Em seguida o compilador chama o analisador de contexto que percorre a árvore sintática, checando-a. Finalmente o módulo principal chama o gerador de código que percorre a árvore e gera o programa objeto. A fig. 6 mostra a estrutura de um compilador de múltiplas passagens (Watt, 1993).

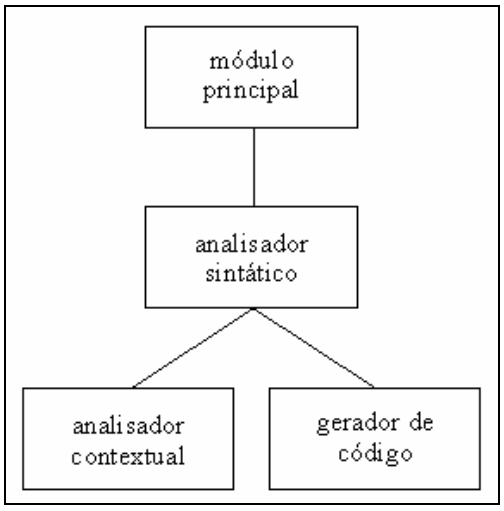
Figura 6 – ESTRUTURA DE UM COMPILADOR DE MÚLTIPLAS PASSAGENS



Fonte: Watt (1993)

Segundo Watt (1993) no compilador de uma passagem o analisador sintático é quem comanda as outras fases da compilação. Um compilador com esta estrutura faz um único passo sobre o programa fonte. A análise contextual e a geração de código são executadas durante a análise sintática. Assim que uma sentença é analisada, o analisador sintático chama o analisador de contexto para executar a checagem, e logo em seguida chama o gerador de código para gerar algum código objeto. Em seguida o analisador sintático continua a análise do programa fonte. A fig. 7 apresenta a estrutura de um compilador de uma passagem.

Figura 7 – ESTRUTURA DE UM COMPILADOR DE UMA PASSAGEM



Fonte: Watt (1993)

2.3.1 ANALISADOR LÉXICO

Segundo Aho (1995) o analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma seqüência de *tokens* que o *parser* (programa responsável pela análise e tradução) utiliza para a análise sintática. Essa interação é normalmente implementada fazendo-se com que o analisador léxico seja uma sub-rotina ou uma co-rotina do *parser*. Ao receber do *parser* o comando “obtenha o próximo *token*”, o analisador léxico lê os caracteres de entrada até que possam identificar o próximo *token*.

Segundo Neto (1987) um *token* ou átomo são trechos elementares completos do programa-fonte, com identidade própria, e representam os elementos terminais da gramática. Estes trechos são individualizados para efeito de análise por parte dos demais programas do compilador. A informação mais importante para análise sintática é a classe a qual pertence o *token*, já para o módulo de geração de código, o mais importante é o valor do *token*.

As vantagens da separação da parte de análise do compilador em análise léxica e análise sintática, segundo Kowaltowski (1983) são:

a) toda parte referente à representação dos terminais está concentrada numa única rotina do compilador, tornando mais simples as modificações da representação;
b) as regras de formação dos átomos são mais simples do que o resto da sintaxe, permitindo o uso de técnicas de análise mais simples e mais eficientes;
c) certos problemas criados pelas linguagens de programação, como o uso de palavras-chaves como identificadores, podem ser resolvidos pelo analisador léxico, mantendo a clareza conceitual do analisador sintático;
d) uma parte apreciável do tempo de compilação corresponde à análise léxica. A sua implementação como uma rotina separada do compilador facilita a introdução de certas otimizações, ou o uso de linguagens de montagem, enquanto o resto do compilador está escrito numa linguagem de alto nível. Muitos sistemas possuem instruções de máquina especialmente apropriadas para a análise léxica. Neste caso, parte do compilador que depende do sistema utilizado poderá estar concentrada numa única rotina.

Segundo Neto (1987) o analisador léxico executa usualmente uma série de funções, não obrigatoriamente ligadas à análise léxica propriamente dita, porém todas de grande importância como infraestrutura para operações das partes do compilador mais ligadas à tradução propriamente dita do texto-fonte. Como exemplos de funções desempenhadas pelo analisador léxico pode-se citar:

a) extração e classificação dos átomos;
b) eliminação de delimitadores e comentários;
c) conversão numérica;
d) tratamento de identificadores;
e) identificação de palavras reservadas;
f) recuperação de erros;
g) listagens;
h) geração de tabelas de referências cruzadas;
i) definição e expansão de macros;
j) interação com o sistema de arquivos;
k) compilação condicional;
l) controles de listagens.

2.3.2 ANALISADOR SINTÁTICO

Segundo Kowaltowski (1983) a análise sintática tem sido objeto de pesquisa muito intensa e existem muitos métodos que podem ser usados. Alguns são aplicáveis a qualquer gramática livre de contexto, outros só se aplicam a certas classes mais restritas de gramáticas. Os métodos variam bastante quanto a sua eficiência, simplicidade de implementação, aplicabilidade a linguagens de programação específicas.

O analisador sintático obtém uma cadeia de *tokens* proveniente do analisador léxico e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte. Espera-se também que o analisador sintático relate quaisquer erros de sintaxe de uma forma inteligível (Aho, 1995).

O analisador sintático, para Neto (1987), é o segundo grande bloco componente dos compiladores, e que se pode caracterizar como o mais importante. A função principal deste

módulo é de promover a análise da seqüência de como os átomos componentes do texto-fonte se apresentam, a partir da qual efetua a síntese da árvore de sintaxe do mesmo, com base na gramática da linguagem-fonte. Como centralizador das atividades da compilação, o analisador sintático opera, em compiladores dirigidos pela sintaxe, como elemento de comando da ativação dos demais módulos do compilador, efetuando decisões acerca de qual módulo deve ser ativado em cada situação da análise do texto-fonte.

Supõe-se que o objetivo da análise sintática é a construção da árvore de derivação ou, então, apenas a decisão se a cadeia dada é ou não uma sentença. O analisador sintático não precisa construir explicitamente uma árvore, mas as suas ações equivalem conceitualmente a realizar esta tarefa (Kowaltowski, 1983).

Em compiladores de múltiplas passagens, segundo Watt (1993), a estrutura das sentenças do programa fonte deve ser representada explicitamente de alguma forma. A escolha desta representação é a principal decisão do desenvolvimento. Uma representação conveniente e largamente utilizada é a árvore de sintaxe.

Segundo Neto (1987) a análise sintática engloba, em geral, diversas funções de grande importância como:

a) identificação de sentenças;
b) detecção de erros de sintaxe;
c) recuperação de erros;
d) correção de erros;
e) montagem da árvore abstrata da sentença;
f) comando da ativação do analisador léxico;
g) comando do modo de operação do analisador léxico;
h) ativação de rotinas da análise referente às dependências de contexto da linguagem;
i) ativação de rotinas de análise semântica;
j) ativação de síntese de código objeto.

Segundo Aho (1995) existem três tipos gerais de analisadores sintáticos. Os métodos universais de análise sintática, tais como o algoritmo de Cocke-Younger-Kasami e o de Early, podem tratar qualquer gramática livre de contexto. Esses métodos porém são muito

ineficientes para se usar em um compilador de produção. Os métodos comumente utilizados são classificados como ascendentes (*bottom-up*) e descendentes (*top-down*). Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem árvores do topo (raiz) para o fundo (folhas) enquanto os *bottom-up* começam pelas folhas e trabalham árvore acima até a raiz. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

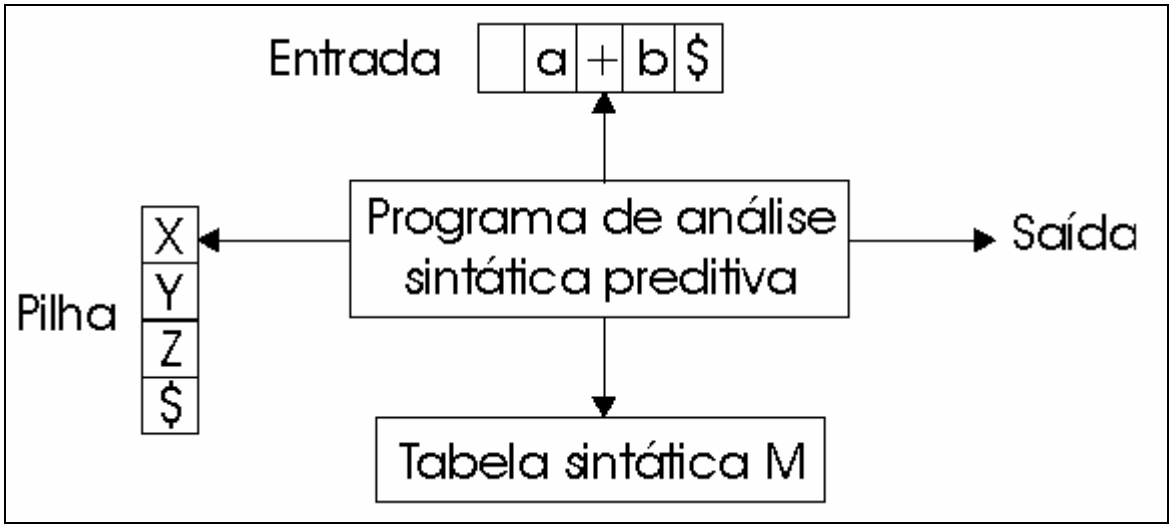
2.3.2.1 ANALISADOR SINTÁTICO TOP-DOWN

A análise sintática *top-down* pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada. Equivalentemente, pode ser vista como uma tentativa de se construir uma árvore gramatical, para a cadeia de entrada, a partir da raiz, criando os nós da árvore gramatical em pré-ordem. O analisador sintático *top-down* sem retrocesso é chamado de analisador sintático preditivo. Analisadores sintáticos com retrocesso não são vistos com freqüência, pois o retrocesso é raramente necessitado para analisar sintaticamente construções de linguagens de programação e na análise de linguagens naturais o retrocesso ainda é ineficiente (Aho, 1995).

Segundo Aho (1995) em muitos casos, escrevendo-se cuidadosamente uma gramática, eliminando-se a recursão à esquerda e fatorando-se à esquerda a gramática resultante, pode-se obter uma nova gramática processável por um analisador sintático preditivo. Para construir um analisador sintático preditivo, precisa-se conhecer, dado o símbolo corrente de entrada *a* e o não-terminal *A* a ser expandido, qual das alternativas da produção $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$ é a única que deriva uma cadeia começando por a. Ou seja, a alternativa adequada precisa ser detectável examinando-se apenas o primeiro símbolo da cadeia que a mesma deriva.

É possível construir um analisador preditivo não-recursivo mantendo explicitamente uma pilha, ao invés de implicitamente através de chamadas recursivas. O problema-chave durante a análise preditiva é determinar qual produção deve ser aplicada a um dado não-terminal. O analisador não-recursivo da fig. 8 procura pela produção a ser aplicada numa tabela sintática, que é uma matriz bidimensional M[*A*, *a*] onde *A* é um não-terminal e *a* é um terminal ou o símbolo $ (que define vazio ou fim de cadeia).

Figura 8 – MODELO DE ANALISADOR SINTÁTICO PREDITIVO NÃO-RECURSIVO



Fonte: Aho (1995)

Um exemplo de tabela sintática para a gramática G = ({E, E', T, T', F}, {+,*,(,),id}, P, E), onde P = {E → TE', E' → +TE' | ε, T → FT', T' → *FT' | ε, F → (E) | id} pode ser visto no quadro 7.

Quadro 7 – EXEMPLO DE TABELA SINTÁTICA

Não-Terminal	Símbolos de entrada					
	Id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Fonte: Aho (1995)

Os algoritmos de análise sintática preditiva não recursiva e o de construção de uma tabela sintática preditiva são apresentados respectivamente nos quadros 8 e 9.

O algoritmo 2 do quadro 9 pode ser aplicado a qualquer gramática G para produzir uma tabela sintática M. Para algumas gramáticas G, por exemplo, se forem recursivas à esquerda ou ambíguas, algumas entradas serão multiplamente definidas.

Quadro 8 – ALGORITMO DE ANÁLISE SINTÁTICA PREDITIVA NÃO RECURSIVA

Algoritmo 1 – Análise sintática preditiva não recursiva

Entrada. Uma cadeia w e uma tabela sintática M para a gramática G.

Saída. Se w estiver em L(G), uma derivação mais à esquerda de w; o contrário, uma indicação de erro.

Método. Inicialmente, o analisador sintático está numa configuração na qual possui somente $S na pilha, com S, o símbolo de partida de G ao topo e w$ no buffer de entrada.

Faça *ip* apontar para o primeiro símbolo de w$;
repetir
    seja X o símbolo ao topo da pilha e *a* o símbolo apontado por *ip*;
    se X for um terminal ou $ **então**
        se X = *a* **então** remover X da pilha e avançar *ip* **senão** *erro()*
    senão /* X é um não-terminal */
        se M[X, a] = $X \rightarrow Y_1Y_2 ...Y_k$ **então**
        inicio
            remover X da pilha;
            empilhar Y_k, Y_{k-1}, ..., Y_1, com Y_1 ao topo da pilha;
            escrever a produção $X \rightarrow Y_1Y_2 ...Y_k$
        fim
        senão *erro()*
até que X = $ /* a pilha está vazia */

Fonte: Aho (1995)

Quadro 9 – ALGORITMO DA CONSTRUÇÃO DE UMA TABELA SINTÁTICA PREDITIVA

Algoritmo 2 – Construção de uma tabela sintática preditiva

Entrada. Uma gramática G.

Saída. Tabela sintática M.

Método.

1. Para cada produção $A \rightarrow \alpha$ da gramática, execute os passos 2 e 3.
2. Para cada terminal *a* em PRIMEIRO(α), adicione $A \rightarrow \alpha$ a M[A, *a*].
3. Se ε estiver em PRIMEIRO(α), adicione $A \rightarrow \alpha$ a M[A, *b*], para cada terminal *b* em SEGUINTE(A). Se ε estiver em PRIMEIRO(α) e $ em SEGUINTE(A), adicione $A \rightarrow \alpha$ a M[A, $].

Faça cada entrada indefinida de M ser erro.

Fonte: Aho (1995)

Segundo Aho (1995) uma gramática que não possui entradas multiplamente definidas é dita LL(1). O primeiro “L” significa varredura da entrada da esquerda para a direita (*left to*

right); o segundo, a produção de uma derivação mais a esquerda (*left linear*); e o “1” o uso de um único símbolo de entrada como *lookahead* a cada passo para tomar as decisões sintáticas.

As gramáticas LL(1) possuem várias propriedades distintas. Pode-se mostrar que um gramática G é LL(1) se, e somente se, sempre que $A \rightarrow \alpha \mid \beta$ forem duas produções distintas de G, rigorosamente as seguintes condições (Aho, 1995):

a) α e β não derivem, ao mesmo tempo, cadeias começando pelo mesmo terminal *a*, qualquer que seja *a;*
b) no máximo um dos dois, α ou β, derive a cadeia vazia;
c) se $\beta \Rightarrow^* \varepsilon$, então α não deriva qualquer cadeia começando por um terminal no conjunto de terminais que pode figurar imediatamente à direita de A em alguma forma sentencial.

2.3.2.2 ANALISADOR SINTÁTICO BOTTOM-UP

Segundo Aho (1995) a análise sintática *bottom-up* também conhecida por análise gramatical de empilhar e reduzir, tenta construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas (o fundo) e trabalhando árvore acima em direção a raiz (o topo). Pode-se pensar neste processo como o de “reduzir” uma cadeia w ao símbolo de partida de uma gramática. A cada passo de redução, uma subcadeia particular, que reconheça o lado direito de uma produção, é substituída pelo símbolo à esquerda daquela produção e, se a subcadeia tiver sido escolhida corretamente a cada passo, uma derivação mais à direita terá sido rastreada na ordem inversa.

Uma forma fácil de implementar a análise gramatical de empilhar e reduzir é chamada análise sintática de precedência de operadores, mas um método muito mais geral de análise *bottom-up* é a análise LR, que é utilizada por vários geradores automáticos de analisadores sintáticos (Aho, 1995).

Segundo Aho (1995) um *handle* é uma subcadeia que reconhece o lado direito de uma produção e cuja redução ao não-terminal do lado esquerdo da produção representa um passo ao longo do percurso de uma derivação mais à direita. Em muitos casos, a subcadeia β mais à esquerda que reconhece o lado direito de uma produção $A \rightarrow \beta$ não é um *handle*, porque uma

redução pela produção $A \rightarrow \beta$ produz uma cadeia que não pode ser reduzida ao símbolo inicial da gramática.

Uma derivação mais a direita na ordem inversa pode ser obtida “podando-se os *handles*”. Ou seja, começa-se pela primeira cadeia de terminais w que se deseja decompor. Se w for uma sentença da gramática em questão, então $w = \gamma_n$, onde é γ_n a enésima forma sentencial à direita de alguma derivação mais à direita ainda desconhecida $S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow ... \Rightarrow \gamma_n = w$. Para reconstruir esta derivação na ordem inversa, localiza-se o *handle* β_n em γ_n e substitui-se β_n pelo lado direito de alguma produção $A_n \rightarrow \beta_n$, de modo a se obter a enésima menos uma forma sentencial à direita γ_{n-1} (Aho, 1995).

Repete-se, em seguida, esse processo. Isto é, localiza-se o *handle* β_{n-1} em γ_{n-1} e o reduz de forma a obter a forma sentencial à direita γ_{n-2}. Continuando esse processo, produz-se uma forma sentencial à direita consistindo somente no símbolo de partida S e então pára-se e anuncia-se o término com sucesso da análise sintática. O reverso da seqüência de produções usadas nas reduções é a derivação mais à direita para a cadeia de entrada (Aho, 1995).

2.3.3 ANALISADOR SEMÂNTICO

Segundo Aho (1995) a fase de análise semântica verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subseqüente de geração de código. Utiliza a estrutura hierárquica determinada pela fase análise sintática, a fim de identificar os operadores e operandos das expressões e enunciados.

Segundo Neto (1987) denomina-se semântica de uma sentença o exato significado por ela assumido dentro do texto em que tal sentença se encontra, no programa-fonte. Semântica de uma linguagem é a interpretação que se pode atribuir ao conjunto de todas as suas sentenças.

Ao contrário da sintaxe, que é facilmente formalizável, com a ajuda de metalinguagens de baixa complexidade, a semântica, apesar de também poder ser expressa formalmente, exige para isto notações substancialmente mais complexas, de aprendizagem mais difícil e em geral apresentando simbologias bastante carregadas e pouco legíveis. Assim sendo, na grande

maioria das linguagens de programação, a semântica tem sido especificada informalmente, via de regra através de textos em linguagem natural (Neto, 1987).

Segundo Neto (1987) as ações semânticas encarregam-se, em geral, de todas as tarefas do compilador que não sejam as análises léxica e sintática, sendo sua execução promovida por iniciativa do analisador sintático, em compiladores dirigidos pela sintaxe. Tipicamente as ações semânticas englobam funções tais como:

a) criação e manutenção de tabelas de símbolos;
b) associar aos símbolos os correspondentes atributos;
c) manter informações sobre o escopo dos identificadores;
d) representar tipos de dados;
e) analisar restrições quanto à utilização dos identificadores;
f) verificar o escopo dos identificadores;
g) identificar as declarações contextuais;
h) verificar a compatibilidade dos tipos;
i) efetuar o gerenciamento de memória;
j) representar o ambiente de execução dos procedimentos;
k) efetuar a tradução do programa;
l) implementar um ambiente de execução;
m) realizar a comunicação entre dois ambientes de execução;
n) fazer a geração de código;
o) realizar a otimização.

Segundo Aho (1995) a fim de traduzir uma construção de linguagem de programação, um compilador pode precisar manter o valor de muitas quantidades além do código gerado pela construção. Pode-se falar abstratamente de atributos associados às construções. Um atributo pode representar qualquer quantidade, por exemplo, um tipo, uma cadeia de caracteres, uma localização de memória ou o que for. Os valores para os atributos são computados através de “regras semânticas” associadas às produções da gramática.

Existem duas notações para associar regras semânticas às produções, definições dirigidas pela sintaxe e esquemas de tradução. As definições dirigidas pela sintaxe são especificações de alto nível para as traduções, que escondem detalhes de implementação e

liberam o usuário de especificar exatamente a ordem na qual as regras semânticas são avaliadas. Os esquemas de tradução indicam a ordem na qual as regras semânticas são avaliadas e assim permitem que alguns detalhes de implementação sejam evidenciados. Conceitualmente, com os dois esquemas, analisa-se sintaticamente o fluxo de *tokens* de entrada, constrói-se a árvore sintática, e em seguida, percorre-se a árvore de forma necessária, avaliando as regras semânticas a cada nó (Aho, 1995).

2.3.3.1 DEFINIÇÕES DIRIGIDAS PELA SINTAXE

Uma definição dirigida pela sintaxe, segundo Aho (1995), é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto associado de atributos, dividido em dois subconjuntos, chamados de atributos sintetizados e atributos herdados.

O valor de um atributo sintetizado em um nó é computado a partir dos valores dos atributos dos filhos daquele nó na árvore gramatical. O valor de um atributo herdado é computado a partir dos valores dos atributos dos irmãos e pai daquele nó (Aho, 1995). O quadro 10 apresenta um exemplo de regras semânticas associadas às produções de uma gramática.

Quadro 10 – EXEMPLO DE REGRAS SEMÂNTICAS

PRODUÇÕES	**REGRAS SEMÂNTICAS**
L $\rightarrow$ E **n**	*Imprimir(E.val)*
E $\rightarrow$ E_1 + T	*E.val* := E_1*.val* + *T.val*
E $\rightarrow$ T	*E.val* := *T.val*
T $\rightarrow$ T_1 * F	*T.val* := T_1*.val* x *F.val*
T $\rightarrow$ F	*T.val* := *F.val*
F $\rightarrow$ (E)	*F.val* := *E.val*
F $\rightarrow$ **digito**	*F.val* := **digito**.*lexval*

Fonte: Aho (1995)

Segundo Aho (1995) as regras semânticas estabelecem dependências entre os atributos, as quais serão representados por um grafo. A partir do grafo de dependências, deriva-se uma ordem de avaliação para as regras semânticas. A avaliação das regras semânticas define os

valores dos atributos nos nós da árvore gramatical para uma dada cadeia de entrada. Uma árvore gramatical mostrando os valores dos atributos a cada nó é denominada de uma árvore gramatical anotada. O processo de computar o valor dos atributos a cada nó é chamado de anotação ou decoração da árvore.

Numa definição dirigida pela sintaxe, assume-se que os terminais tenham somente atributos sintetizados, na medida em que a definição não providencie quaisquer regras semânticas para os mesmos. Os valores dos atributos dos terminais são usualmente fornecidos pelo analisador léxico. Sobretudo, é assumido para o símbolo de partida que os mesmo não tenha quaisquer atributos herdados, a menos que seja estabelecido o contrário (Aho, 1995).

2.3.3.1.1 ATRIBUTOS SINTETIZADOS

Segundo Aho (1995) os atributos sintetizados são usados extensivamente na prática. Uma definição dirigida pela sintaxe que usa exclusivamente atributos sintetizados é dita uma definição S-atribuída. Uma árvore gramatical para uma definição S-atribuída pode ser sempre anotada através da avaliação das regras semânticas para os atributos de cada nó, de baixo para cima, das folhas para a raiz.

2.3.3.1.2 ATRIBUTOS HERDADOS

Um atributo herdado é aquele cujo valor a um nó de uma árvore gramatical é definido em termos de pai e/ou irmãos daquele nó. Os atributos herdados são convenientes para expressar a dependência de uma construção de linguagem de programação no contexto em que a mesma figurar. Por exemplo, pode-se usar um atributo herdado para controlar se um identificador aparece ao lado esquerdo ou direito de um comando de atribuição a fim de decidir se é necessário o endereço ou o valor de um identificador. Apesar de ser sempre possível reescrever uma definição dirigida pela sintaxe de forma a se usar somente atributos sintetizados, estas definições com atributos herdados são freqüentemente mais naturais (Aho, 1995).

2.4 ORIENTAÇÃO A OBJETOS

Segundo Rumbaugh (1994) modelagem e projetos baseados em objetos é um novo modo de estudar problemas com utilização de modelos fundamentados em conceitos do

mundo real. A estrutura básica é o objeto, que combina a estrutura e o comportamento dos dados em uma única entidade.

A análise de sistemas no mundo orientado a objetos é feita analisando-se os objetos e os eventos que interagem com esses objetos. O projeto de software é feito reusando-se classes de objetos existentes e, quando necessário, construindo-se novas classes. Ao modelar uma empresa, os analistas devem identificar seus tipos de objetos e as operações que fazem com que esses objetos se comportem de certas maneiras (Martin, 1995).

2.4.1 OBJETOS E CLASSES

Segundo Rumbaugh (1994) um objeto é simplesmente alguma coisa que faz sentido no contexto de uma aplicação. Todos os objetos possuem identidade e são distinguíveis, por exemplo, duas maçãs da mesma cor, formato e textura continuam sendo maçãs distintas.

Na análise orientada a objetos, segundo Martin (1995), tipo de objeto é uma noção conceitual, que especifica uma família de objetos sem estipular como o tipo e o objeto são implementados. O termo classe refere-se à implementação de software de um tipo de objeto.

2.4.2 ATRIBUTOS E MÉTODOS

Segundo Tonsing (2000) os atributos representam as características do objeto, por exemplo o objeto caneta, possui como atributos: tamanho, cor, fabricante e modelo. Segundo Rumbaugh (1994), diferentes instâncias podem ter valores iguais ou diferentes para um dado atributo. Um atributo deve ser um puro valor de dado, e não um objeto. Um objeto não deve ser modelado com atributo e sim com uma associação entre objetos.

Segundo Rumbaugh (1994), uma operação é uma função ou transformação que pode ser aplicada a objetos ou por estes a uma classe. Contratar e Despedir são exemplos de operações da classe Empresa. A mesma operação pode ser aplicada a classes diferentes. Uma operação assim é dita polimórfica, isto é, a mesma operação pode tomar formas diferentes em classes diferentes. Um método é a implementação de uma operação em uma classe.

O ato de empacotar ao mesmo tempo dados e métodos é denominado encapsulamento. O objeto esconde seus dados de outros objetos e permite que os dados sejam acessados por

intermédio de seus próprios métodos. O encapsulamento oculta os detalhes de sua implementação interna aos usuários do objeto. Os usuários entendem quais operações do objeto podem ser solicitadas, mas não conhecem os detalhes de como a operação é executada. O encapsulamento permite que as implementações do objeto sejam modificadas sem exigir que os aplicativos que as usam sejam também modificadas (Matrin, 1995).

2.4.3 LIGAÇÃO E ASSOCIAÇÃO

Ligações e associações são os meios para estabelecer relacionamentos entre objetos e classes. Segundo Rumbaugh (1994) uma ligação é uma conexão física ou conceitual entre instancias de objetos. Por exemplo, Joe Smith trabalha para a empresa Simples. Matematicamente, uma ligação é definida como uma tupla, isto é, uma lista ordenada de instâncias de objetos. Uma ligação é uma instância de uma associação.

2.4.4 AGREGAÇÃO

Segundo Rumbaugh (1994) agregação é o relacionamento “parte-todo” ou “uma-parte-de” no qual os objetos que representam os componentes de alguma coisa são associados a um objeto que representa a estrutura inteira.

A agregação é uma forma estreitamente acoplada de associação com algumas questões semânticas a mais. A propriedade mais significativa da agregação é a transitividade, isto é, se A faz parte de B e B faz parte de C, então A faz parte de C. A agregação é também, anti-simétrica, isto é, se A faz parte de B então B não faz parte de A (Rumbaugh, 1994).

2.4.5 GENERALIZAÇÃO E HERANÇA

Segundo Rumbaugh (1994) generalização e herança são abstrações poderosas para o compartilhamento de semelhanças entre classes, ao mesmo tempo em que suas diferenças são preservadas.

Generalização é o relacionamento de uma classe e uma ou mais versões refinadas dela. A classe que estiver em processo de refinamento é chamada de superclasse e cada versão refinada é denominada subclasse. Por exemplo Equipamento é uma superclasse de Bomba e

Tanque. Os atributos e operações comuns a um grupo de subclasses são incluídos na superclasse e compartilhados por todas as subclasses. Diz-se que a subclasse herda as características da superclasse (Rumbaugh, 1994).

2.4.6 UNIFIED MODELLING LANGUAGE

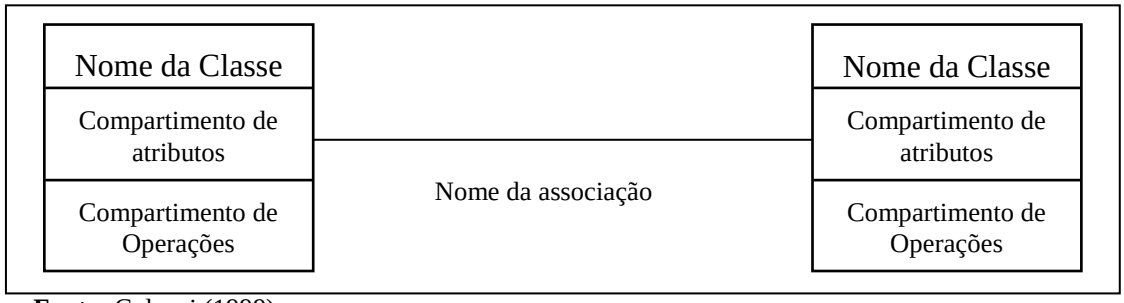
Segundo Colanzi (1999) a *Unified Modelling Language* (UML), consiste de um metamodelo e uma notação, inteiramente descrita em uma série de documentos disponíveis pela *Rational Software Corporation*. É uma linguagem para especificação, construção, visualização e documentação dos componentes de um sistema, especialmente aqueles fortemente ligados a software. Seu foco principal está em criar uma linguagem de modelagem padronizada, não em criar um processo padronizado.

2.4.6.1 DIAGRAMA DE CLASSES

Segundo Colanzi (1999) um diagrama de classe é uma coleção de elementos estáticos do modelo declarativo, tais como classes, tipos, e seus relacionamentos, conectados uns aos outros e aos seus conteúdos como um grafo.

Uma classe é um descritor para um conjunto de objetos com estrutura, comportamento e relacionamentos similares. Nos diagramas de classes da UML uma classe é representada por um retângulo com três compartimentos, conforme indicado na fig. 9. O primeiro compartimento é para o nome da classe, o segundo para os seus atributos e o último para as suas operações. As associações são mostradas como linhas que conectam classes. Existe uma variedade de adornos para indicar as propriedades das associações, como nome da associação, cardinalidade, papéis, direcionamento, etc. Nomes de papéis e direcionamento são úteis para deixar claro a ordem de leitura dos relacionamentos, principalmente em auto-relacionamentos (Rational, 1997).

Figura 9 – NOTAÇÃO DE CLASSES E ASSOCIAÇÕES



Fonte: Colanzi (1999)

Segundo Rational (1997) os detalhes de uma expressão do tipo do atributo não são especificados pela UML, pois dependem da sintaxe de expressão suportada por uma particular especificação ou da linguagem de programação que está sendo usada. Um atributo é mostrado como uma *string* textual. A sintaxe padrão é *visibilidade nome* : *tipo = valor_inicial* { *property string* }, onde:

a) *visibilidade* é um dos seguintes símbolos: "+" visibilidade pública, "#" visibilidade protegida, "-" visibilidade privada. O símbolo de visibilidade pode ser suprimido. Neste caso, a ausência do símbolo deve ser entendida como "não mostrada" e não como "não definida";
b) *nome* é uma *string* identificadora;
c) *tipo* é uma especificação dependente do tipo da implementação de um atributo;
d) *valor_inicial* é uma expressão dependente de linguagem para o valor inicial de um objeto criado recentemente. É opcional e neste caso o sinal "=" também é omitido. Um construtor explícito para um novo objeto pode modificar o valor inicial *default*;
e) *property-string* indica valores de propriedades que são aplicadas ao elemento. É opcional e neste caso as chaves "{}" também são omitidas.

Assim como acontece com os atributos, os detalhes da expressão de tipo dos métodos também não são especificados pela UML. A sintaxe padrão é *visibilidade nome (lista_parâmetros) : tipo_retorno { property string },* onde:

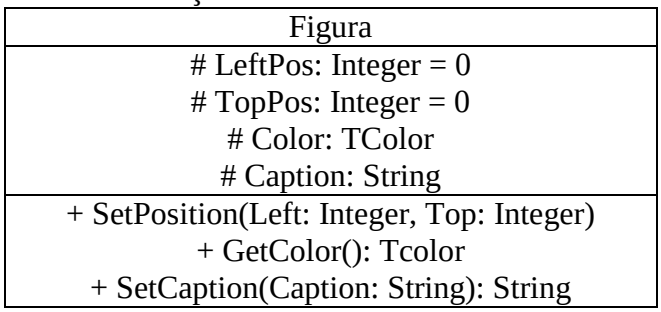
a) *visibilidade nome* é semelhante ao apresentado para atributos;
b) *tipo_retorno* é uma especificação dependente do tipo da implementação do valor retornado pela operação. Se omitido, a operação não retorna valor;

c) *lista_parâmetros* é uma lista de parâmetros formais separados por vírgula, cada um especificado segundo a seguinte sintaxe: *nome* : *tipo* = *valor-padrão*, onde
   - *nome* é o nome do parâmetro formal;
   - *tipo* é uma especificação dependente da implementação do tipo;
   - *valor_padrão* é um valor opcional para o parâmetro;

d) *property-string* é semelhante ao apresentado para atributo.

A fig.10 mostra um exemplo de notação da classe Figura, contendo atributos e métodos.

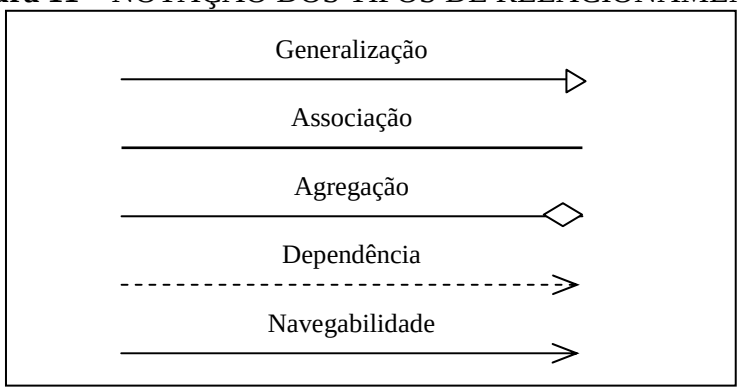
Figura 10 – NOTAÇÃO DE ATRIBUTOS E MÉTODOS



Fonte: Colanzi (1999)

Uma associação é mostrada como linhas conectando símbolos de classes. As linhas podem ter adornos que indicam suas propriedades. Associações ternárias ou de ordens superiores são representadas como um diamante conectado às classes por linhas. Uma seqüência de linhas conectadas é chamado de um "caminho". Na fig 11 é apresentada a notação dos tipos de relacionamentos existentes nos diagramas de classes da UML. Maiores detalhes a respeito da notação dos diagramas de classes são encontrados em Rational (1997).

Figura 11 – NOTAÇÃO DOS TIPOS DE RELACIONAMENTO

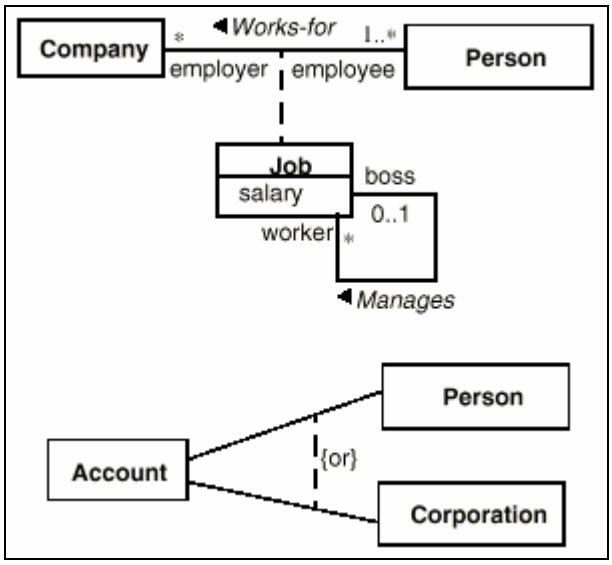


Fonte: Rational (1997)

Os caminhos podem conter os seguintes tipos de adornos:

a) nome da associação: designa um nome para a associação. É opcional. É mostrado próximo ao caminho. A *string* pode conter um pequeno triângulo sólido indicando a direção de leitura da associação. Este triângulo não tem nenhum significado semântico; é puramente descritivo;
b) classe de associação: designa uma associação que tem propriedades como uma classe, tais como atributos, operações e outras associações. Isto está presente se, e somente se, a associação é uma classe de associação. É representado por um símbolo de classe conectado ao caminho da associação por uma linha pontilhada. É geralmente utilizada para armazenar atributos de relacionamento;
c) associação-*or*: uma associação *or* indica uma situação na qual apenas uma das associações potenciais podem ser instanciadas em um certo tempo por um único objeto. Isto é representado como uma linha pontilhada ligando as duas ou mais associações, as quais todas devem ter uma classe em comum, com uma *string* de restrição "{or}" rotulando a linha pontilhada. A fig 12 apresenta um exemplo de associação-*or*;
d) indicador de agregação: um pequeno símbolo em forma de diamante é colocado ao final do caminho para designar agregação. O diamante é conectado a ponta do caminho que conecta a classe que é agregada. A agregação é opcional. Se o diamante estiver preenchido, então está sendo representado uma forma de agregação forte chamada de "composição";
e) multiplicidade: a *string* de multiplicidade especifica o limite de cardinalidade que é permitido a um conjunto assumir. A multiplicidade pode ser especificada para associações, composições, repetições, e outros propósitos. Essencialmente uma multiplicidade é um subconjunto aberto de inteiros não-negativos.

Figura 12 – NOTAÇÃO PARA ASSOCIAÇÃO, CLASSE DE ASSOCIAÇÃO, ASSOCIAÇÃO-OR E MULTIPLICIDADE



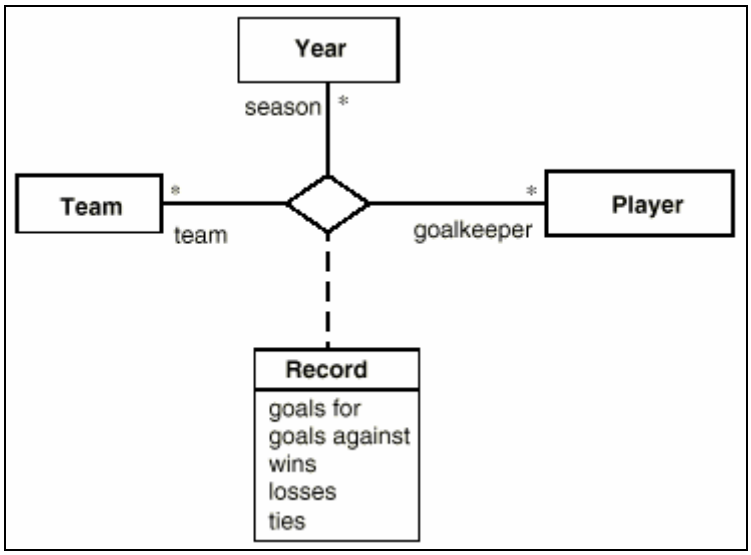
Fonte: Rational (1997)

Segundo Rational (1997) uma associação n-ária é uma associação entre três ou mais classes. Cada instância da associação é uma n-tupla de valores das respectivas classes. Uma associação binária é um caso especial que contém notação própria. Uma associação n-ária é representada por um diamante grande (grande se comparado com o terminador de caminho) com um caminho do diamante para cada uma das classes participantes. O nome da associação, se houver, é mostrado próximo ao diamante. A multiplicidade pode ser indicada, contudo, agregações não são permitidas. A fig. 13 apresenta um exemplo de associação n-ária, um símbolo de classe de associação pode ser conectado ao diamante por uma linha pontilhada. Isto indica que a associação n-ária tem atributos, operações e/ou associações.

Segundo Colanzi (1999) uma composição é uma forma de agregação com forte propriedade e tempo de vida da parte coincidente com o todo. A agregação é um tipo de associação que representa um relacionamento que indica que uma classe é parte-de uma outra classe. Um relacionamento de composição indica que uma classe é composta por outras, ou seja, indica que uma classe agrega outras. Composição pode ser representada por um diamante sólido (preenchido) como um terminador de um caminho de uma associação. Alternativamente, a UML provê uma forma graficamente aninhada, que em muitos casos,

pode ser mais conveniente para demonstrar composição. A fig. 14 apresenta a notação para composição.

Figura 13 - ASSOCIAÇÃO TERNÁRIA QUE TAMBÉM É UMA CLASSE DE ASSOCIAÇÃO

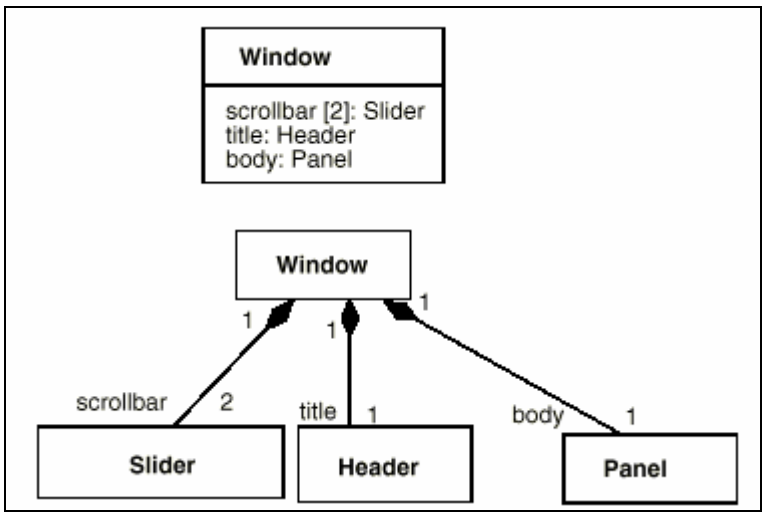


Fonte: Rational (1997)

Segundo Colanzi (1999) generalização é um relacionamento entre um elemento mais geral e um elemento mais especifico que é totalmente consistente com o primeiro elemento e contém informações adicionais. A generalização é utilizada para representar relacionamentos de herança. É usado para classes, pacotes, casos de uso e outros elementos.

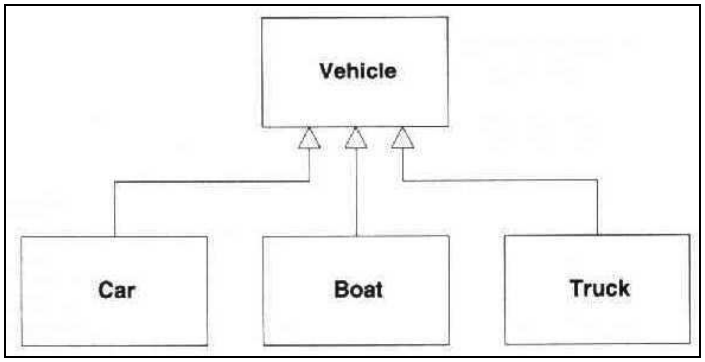
Segundo Rational (1997) a generalização é representada como um caminho (uma linha sólida) do elemento mais específico (tal como uma subclasse) para o elemento mais geral (como uma superclasse), com um triângulo no final do caminho, onde este encontra o elemento mais geral. Um grupo de caminhos de generalização para uma dada superclasse pode ser representado como uma árvore com segmentos compartilhados (inclusive o triângulo) para a superclasse, ramificando-se em múltiplos caminhos para cada subclasse. Se um rótulo é colocado sobre o triângulo de generalização compartilhado por vários caminhos de generalização para subclasses, o rótulo aplica-se a todos os caminhos. Um exemplo de generalização é apresentado na fig 15.

Figura 14 – EXEMPLO DE COMPOSIÇÃO



Fonte: Rational (1997)

Figura 15 – EXEMPLO DE GENERALIZAÇÃO



Fonte: Rational (1997)

A existência de subclasses adicionais em um modelo e que não estão sendo mostradas em um diagrama particular pode ser representada pelo sinal de reticências (...) no lugar dessas subclasses. É importante ressaltar que este sinal não indica que classes adicionais possam ser conectadas futuramente. Indica que classes adicionais existem mas não estão sendo mostradas no momento (Rational, 1997).

2.5 SYSTEM ARCHITECT

Segundo Popkin (2001) a ferramenta CASE *System Architect* é composta por um rico conjunto de componentes que permitem a captura, desenho, modelagem e criação de sistemas

corporativos. Todas as informações são armazenadas em um repositório multi-usuário chamado enciclopédia.

O *System Architect* proporciona suporte integrado para todas as quatro áreas de análise e projeto de sistemas, modelagem de negócio, modelagem de objetos, modelagem de dados relacional e análise estruturada (Popkin, 2001).

A modelagem de objetos é proporcionada através do suporte global à notação UML, com suporte a engenharia reversa de várias linguagens diferentes (Popkin, 2001).

2.5.1 ENCICLOPÉDIA

Segundo Popkin (2001) a enciclopédia do *System Architect* é uma base de dados relacional localizada em um único subdiretório em um computador. A relação entre a enciclopédia e o subdiretório é um para um, isto é, uma enciclopédia está em um subdiretório e um subdiretório contém apenas uma enciclopédia.

Todos os diagramas e todas as definições, aquelas associadas a diagramas e as que não estão associadas a diagramas, estão na enciclopédia. Adicionalmente, outros objetos também estão na enciclopédia, tais como, estilos, *metafiles*, entre outros (Popkin, 2001).

Segundo Popkin (2001) o repositório guarda as definições dos componentes que fazem parte de um projeto. Pode-se atribuir algum tipo de informação em cada diagrama. Pode-se, também, incluir informações sobre componentes não-gráficos, tais como, elementos de dados, estruturas de dados, atributos, requisitos, planos de teste, objetos de negócio, entre outros. Cada elemento gráfico e não-gráfico no *System Architect* possui, pelos menos, a propriedade “Description”, na qual pode-se colocar textos sobre a razão deste objeto no projeto.

Segundo Popkin (2001) a enciclopédia contém:

a) uma base de dados relacional, consistindo em duas tabelas e alguns índices;
b) um arquivo para cada diagrama gráfico;
c) um metafile (WMF) para cada diagrama gráfico;
d) quatro arquivos que determinam a configuração da enciclopédia;
e) um arquivo de “lock” se o repositório estiver rodando em um ambiente de rede;

f) um ou mais estilos.

A base de dados relacional é constituída de duas tabelas principais e alguns índices para navegação. Estas tabelas são a ENTITY.DBF e a RELATN.DBF, que estão no formato DBase III Plus que facilita o acesso às informações por ser um padrão bem difundido (Popkin, 2001).

A seguir serão descritos os arquivos ENTITY.DBF e RELATN.DBF.

2.5.1.1 ENTITY.DBF

Segundo Popkin (2001) cada entrada na tabela de entidades é unicamente definida por sua classe, tipo e nome. Quando a classe for igual a "*Diagram*" ou igual a "*Definition*", cada combinação classe/tipo/nome deve ser única. De outra forma, quando a classe for "Symbol", as entradas podem ou não ser únicas. Símbolos duplicados em um diagrama são apropriados para alguns outros diagramas, como os Diagramas de Fluxo de Dados (DFD), e não para outros como o Modelo de Entidades e Relacionamentos (MER).

Cada entrada na tabela de entidades pertence a uma das três classes: *Diagram* (código = 1), *Symbol* (código = 2) e *Definition* (código = 3). Cada uma das classes possui seu próprio conjunto de valores de tipo permitidos, por exemplo, uma entrada na tabela de entidades onde a classe for 1 e o tipo também for 1 representa um diagrama para fluxo de dados de Gane & Sarson (Popkin, 2001).

A tabela de entidades possui os seguintes treze campos:

a) NAME que é um campo que armazena 31 caracteres, que representa o nome do usuário;
b) ID que é um número inteiro de 32 bits, atribuído pelo sistema, que é um número identificador;
c) CLASS que é a código da classe que pode representar um diagrama (1), símbolo (2) ou definição (3);
d) TYPE que representa o código do tipo, que junto com o código da classe representa um tipo de informação;
e) NUMBER é o número do símbolo;

f) TOARROW é um valor booleano, que indica a presença da cabeça de uma seta no final de um símbolo de linha;
g) FROMARROW é um valor booleano, que indica a presença da cabeça de uma seta no inicio de um símbolo de linha;
h) TOASSOC é o código indicando a que tipo de cardinalidade será desenhada no final de um símbolo linha. Os códigos possíveis são:
   - não definido (código = 0);
   - um e apenas um (código = 1);
   - zero ou um (código = 2);
   - um ou mais (código = 3);
   - zero, um ou mais (código = 4);
   - muitos (código = 5);
   - desconhecido (código = 6);
   - não definido (código = 7);
   - superclasse (código = 8);
   - subclasse (código = 9);
i) FROMASSOC é o código indicando a que tipo de cardinalidade será desenhada no início de um símbolo linha. Os códigos são os mesmos utilizados em TOASSC;
j) UPDATDATE é a última data onde houve atualização. Em um ambiente multiusuário, pode também representar a data quando o item foi travado;
k) UPDATTIME é a hora da última atualização, representada no formato de 24 horas (HHMMSS). Em um ambiente multiusuário, também representa a hora quando o item foi travado;
l) AUDIT é uma entrada inserida pelo usuário ou o último identificador de auditoria. Em um ambiente multiusuário, também representa o ID do usuário que travou o item;
m) DESCRIPTN é um campo texto onde grande quantidade de texto, contendo propriedade do item, é armazenada;

2.5.1.2 RELATN.DBF

Segundo Popkin (2001) o arquivo de relacionamentos descreve todas as relações que existem entre as entidades contidas dentro do arquivo de entidades. Para aumentar a velocidade do processamento, cada relacionamento é inserido na tabela duas vezes, uma indo e outra voltando. Assim, dado o diagrama A e o símbolo B, a tabela de relacionamentos contem o par “A contém B” e “B está contido em A”.

A tabela de relacionamentos possui os três seguintes campos:

a) ID que é um número inteiro de 32 bits, indicando o ID da primeira entidade;
b) ID2 que é um número inteiro de 32 bits, indicando o ID da segunda entidade;
c) RELATION que representa o relacionamento entre as duas entidades. Os relacionamentos são sempre agrupados usando valores pares e ímpares, tendo os significados apresentados no quadro 11.

Quadro 11 – TABELA DE CÓDIGOS DA TABELA RELATN.DBF

Código	**Relacionamento**	**Relacionamento entre as entidades**
2 3	Contém Está contido em	Um diagrama contém um símbolo. Um símbolo está contido num diagrama.
4 5	Expande para Expande de	Um símbolo expande para um diagrama. Um diagrama expande de um símbolo.
6 7	Conecta Conecta	Um símbolo conecta a um início de linha. Um início de linha conecta um símbolo.
8 9	Conecta Conecta	Um símbolo conecta a um fim de linha. Um fim de linha conecta um símbolo.
10 11	Conecta Conecta	Um módulo conecta e envia dados por um *flag symbol*. Um *flag symbol* conecta e recebe dados de um módulo.
12 13	Conecta Conecta	Um módulo conecta e envia dados de um *flag symbol*. Um *flag symbol* conecta e provém dados a um módulo.
14 15	Usa É usado por	Uma expressão usa dados (elementos ou estruturas). Dados (elementos ou estruturas) são usados por uma expressão.
16 17	Explicado por Explica	Um símbolo é explicado por um comentário. Um comentário explica um símbolo.
18 19	Endereça Endereçado por	Um símbolo endereça um requerimento, plano de teste. Um requerimento, plano de teste é endereçado por um símbolo.
20 21	Definido por Define	Um símbolo é definido por uma definição. Uma definição define um símbolo.
22 23	Qualificado por Qualifica	Um símbolo linha é “qualificado por” um *flag*. Um *flag* “qualifica” um símbolo linha.

24 25	É uma instância Instanciado por	Uma definição “é uma instância” de uma definição. Uma definição é “instanciada por” uma definição.
26 27	Identifica É identificado por	Um objeto “identifica” outro objeto. Um objeto tem um outro objeto como parte de sua chave.
28 29	Embute É embutido por	Um símbolo embute completamente outro símbolo. Um símbolo é completamente embutido por outro símbolo.
40 41	É filho de É pai de	Um símbolo é filho de outro símbolo. Um símbolo é pai de outro símbolo.
42 43	É o primeiro filho de Tem o primeiro filho	Um símbolo é o primeiro filho de outro símbolo. Um símbolo tem outro símbolo como primeiro filho.
44 45	É o próximo irmão de É o irmão anterior de	Um símbolo é o próximo irmão de outro símbolo. Um símbolo é o irmão anterior de outro símbolo.
48 49	Originado de É origem de	Um objeto é originado de uma definição. Uma definição é a origem de um objeto derivado.
50 51	É baseado em É a base para	Um objeto é baseado em uma definição. Uma definição é a base para um objeto derivado.

Fonte: Popkin (2001)

2.6 BANCO DE DADOS CACHÉ E A LINGUAGEM CDL

Segundo Intersystems (2000) o Caché é um banco de dados de alto desempenho com capacidade de modelar o mundo real e com integração com o SQL. O Caché combina o poder da tecnologia de orientação a objetos com o desempenho de uma estrutura de dados multidimensional, permitindo criar aplicações de banco de dados com todas as vantagens do modelo orientado a objetos e o modelo relacional.

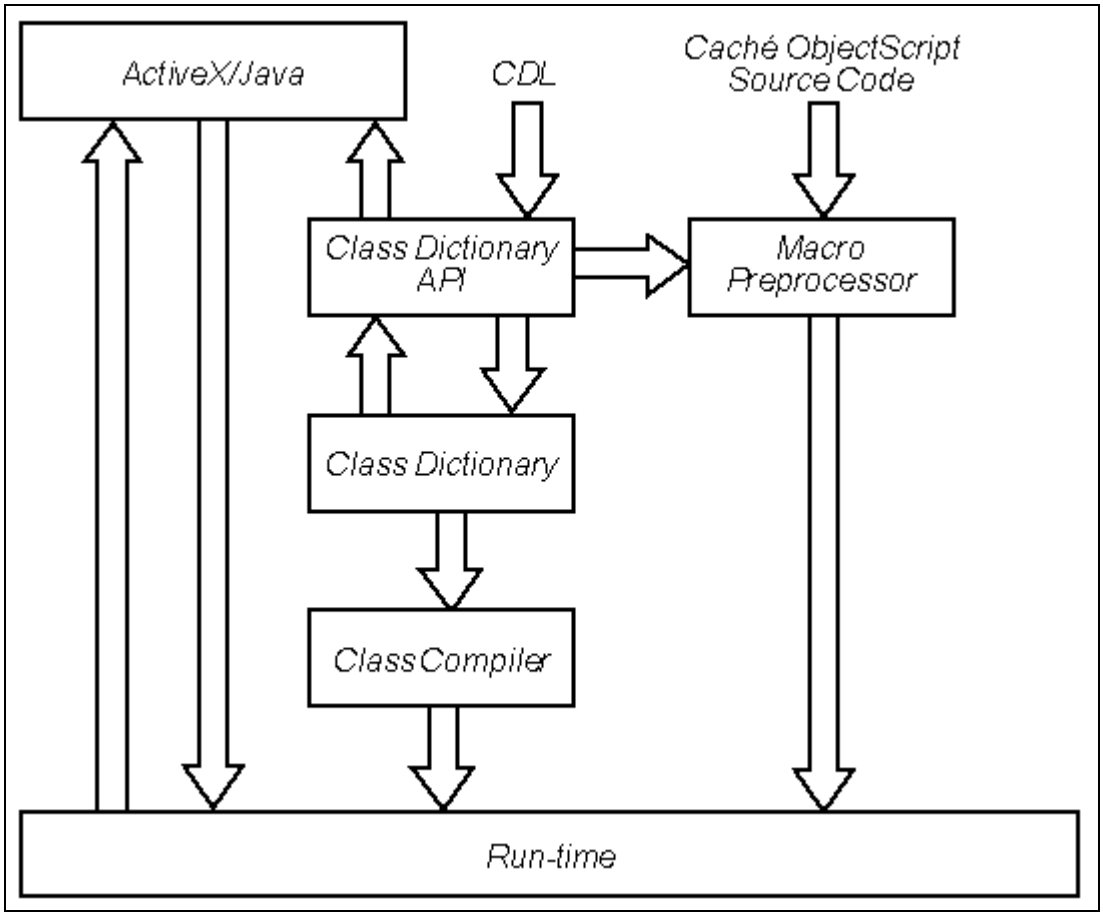
A tecnologia de objetos do Caché (*Caché Objects*) (fig. 16) consiste nos seguintes subsistemas (Intersystems, 2000):

a) *Caché Object Architect*: é um ambiente com interface *Graphic User Interface* (GUI) que permite a manipulação completa das classes de *Caché Objects*;
b) *Caché Object Server for ActiveX*: é um componente ActiveX que permite a exposição de objetos Caché como objetos nativos ActiveX para utilização através de ferramentas de desenvolvimento como o Delphi e Visual Basic;
c) *Caché Object Server for Java*: permite a exposição de objetos Caché como objetos nativos Java;
d) *Class Dictionary*: é responsável por armazenar as definições de classes do usuário e do sistema Caché. Cada *namespace* do Caché possui uma *Class Dictionary*;

e) *Class Dictionary Application Progam Interface* (*Class Dictionary API*): consiste em um conjunto de programas responsáveis pela comunicação entre o *Class Dictionary* e o restante dos componentes do *Caché Objects*;

f) *Class Compiler*: compila as definições de classe armazenadas no *Class Dictionary*.

Figura 16 – ESTRUTURA DO CACHÉ OBJECTS



Fonte: Intersystems (2000)

Segundo Intersystems (2000) a *Class Definition Language* (CDL) é uma linguagem do Caché para definição de classes. Uma definição de classes em CDL pode ser definida em um arquivo texto (usando um editor de texto ou talvez gerado a partir de uma ferramenta) e usado com o Caché ou exportado para o uso em qualquer um dos ambientes de desenvolvimento suportados.

A sintaxe para especificação de uma classe em CDL é a apresentada no quadro 12. Em NOME_CLASSE coloca-se o nome da classe que se quer definir. Em CLASS_DEFINITION,

pode-se modificar a definição da classe através do uso de uma ou mais palavras reservadas, algumas delas são mostradas no quadro 13. Cada palavra reservada é opcional e possui um valor padrão, se nenhum valor for explicitamente especificado.

Quadro 12 – SINTAXE DA DEFINIÇÃO DE UMA CLASSE EM CDL

```
Create Class NOME_CLASSE
{
   CLASS_DEFINITION
}
```

Fonte: Intersystems (2000)

Quadro 13 – PALAVRAS RESERVADAS PARA DEFINIÇÃO DE CLASSE EM CDL

ABSTRACT	Especifica que uma classe não pode ter instâncias; para tipos de dados especifica que a classe não pode ser usada como tipo de atributo. Como padrão, classes não são abstratas. As subclasses não herdam o valor da palavra reservada ABSTRACT.
ARRAY	Especifica a definição de uma atributo do tipo array pertencente a classe. Como padrão, classes não contêm arrays.
ATTRIBUTE	Especifica a definição de um atributo de valor simples pertencente a classe. Como padrão, classes não contêm atributos.
BINARYSTREAM	Especifica a definição de atributos de *streams* binários (BLOB) pertencente a classe. Como padrão, classes não contêm atributos de *streams* binários.
CHARACTERSTREAM	Especifica a definição de atributos de *streams* de caracteres (CLOB) pertencente a classe. Como padrão, classes não contêm atributos de *streams* de caracteres.
DATATYPE	Especifica que a classe é um tipo de dado. Como padrão, classes não são tipos de dados.
DESCRIPTION	Disponibiliza uma descrição opcional para a classe. Como padrão as classes não possuem descrição.
FINAL	Especifica que a classe não pode ter subclasses. Como padrão, classes não são finais.
LIST	Especifica a definição de uma lista de atributos. Como padrão, classes não contêm listas de atributos.
METHOD	Especifica a definição de um método. Como padrão, classes não possuem métodos.
PERSISTENT	Especifica que o objeto é persistente e especifica que definição de armazenamento usar. Como padrão classes não são persistentes.
SUPER	Especifica que a classe herda as características de uma ou mais classes. Como padrão, classes não têm superclasse.

Fonte: Intersystems (2000)

Em CDL, a sintaxe para a definição dos atributos em uma classe é "*attribute NAME {ATTRIBUTE_DEFINITIONS}*" onde NAME é o nome do atributo e

ATTRIBUTE_DEFINITIONS é uma ou mais palavras reservadas que modificam a definição do atributo, apresentadas no quadro 14.

Quadro 14 – PALAVRAS RESERVADAS PARA A DEFINIÇÃO DE UM ATRIBUTO EM CDL

DESCRIPTION	Disponibiliza uma descrição opcional para o atributo. Como padrão os atributos não possuem descrição.
FINAL	Especifica que uma subclasse não pode sobreescrever o atributo. Como padrão um atributo não é final.
INITIAL	Especifica o valor inicial para o atributo. Como padrão, atributos não tem valor inicial.
MULTIDIMENSIONAL	Especifica que um atributo tem as caracteristicas de uma *array* multidimensional. Como padrão, atributos não são *arrays* multidimensionais.
PRIVATE	Especifica que um atributo é privado. Como padrão atributos não são privados.
REQUIRED	Especifica que o valor do atributo precisa ser atribuido antes de que a classe possa ser armazenada no disco. Como padrão, valores de atributos não são obrigatórios.
TRANSIENT	Especifica que um atributo não é armazenada no banco de dados.
TYPE	Especifica o nome de uma classe associada a um atributo, que pode ser uma classe de tipo de dado, uma classe persistente. Como padrão, o tipo de um atributo é %String. Os seguintes tipos são disponibilizados pelo Caché: ▪ %Binary: representa valores binários; ▪ %Boolean: representa um valor booleano; ▪ %Currency: representa um valor com precisão de 8 casas decimais; ▪ %Date: representa uma data; ▪ %Float: representa um número de ponto flutuante; ▪ %Integer: representa um número inteiro; ▪ %List: representa dados em uma lista; ▪ %Name: representa um nome na forma “Sobrenome, Nome”; ▪ %Numeric: representa valores numéricos de precisão variada; ▪ %Status: representa um código de erro; ▪ %String: representa uma cadeia de caracteres; ▪ %Time: representa um valor de hora; ▪ %TimeStamp: representa um valor de data e hora.

Fonte: Intersystems (2000)

Os métodos são definidos, em CDL, da forma “*method NAME_METHOD(PARAM_LIST) {METHOD_DEFINITIONS}*”, onde NAME_METHOD é o nome do método, PARAM_LIST é a lista de parâmetros no formato “PARAM_NAME : PARAM_TYPE”, onde PARAM_NAME é o nome do parâmetro e PARAM_TYPE é o tipo

de dado do parâmetro igual ao tipo de dado para atributos (quadro 14). As definições de mais de um parâmetro são separados por vírgula. METHOD_DEFINITIONS são palavras reservadas que modificam a definição dos métodos (quadro 15).

Quadro 15 – PALAVRAS RESERVADAS PARA A DEFINIÇÃO DE UM MÉTODO EM CDL

CALL	Especifica uma rotina que executa quando o método é chamado. Como padrão um método não chama uma rotina.
CLASSMETHOD	Especifica que o método é um método de classe. Como padrão, um método é um método de instância.
CODE	Especifica código *ObjectScript* para executar quando o método é chamado. Como padrão, um método não tem código a executar.
DESCRIPTION	Disponibiliza uma descrição opcional para o método. Como padrão os métodos não possuem descrição.
EXPRESSION	Especifica uma expressão simples para substituir a chamada do método. Como padrão, métodos não são expressões.
FINAL	Especifica que uma subclasse não pode sobreescrever o método. Como padrão um método é virtual.

Fonte: Intersystems (2000)

Um exemplo de arquivo contendo código fonte CDL é apresentado no quadro 31 do anexo 2.

2.7 LINGUAGEM JAVA

Segundo Niemeyer (1997) Java é uma linguagem de programação para programação utilizando redes de computadores que foi desenvolvida pela Sun Microsystems. O Java já é muito utilizada para fazer animações em páginas *Web*. A linguagem e o ambiente Java é robusta suficiente para suportar novos tipos de aplicações, como *browsers* dinamicamente extensíveis.

Segundo Flanagan (1997) as principais características da linguagem Java são:

a) a linguagem Java dá suporte para a programação orientada a objetos;
b) o Java é uma linguagem interpretada. O compilador Java gera código para a *Java Virtual Machine* (JVM), ao invés de gerar código nativo de máquina. Para executar um programa escrito em Java, usa-se o interpretador Java que executa o código compilado. Como o código Java é independente de plataforma, os programas Java

podem ser executados em qualquer plataforma, desde que exista uma JVM (interpretador e sistema de *run-time*) portada para este ambiente;

c) arquitetura neutra e portável: como os programas Java são compilados em uma arquitetura neutra, aplicações Java podem executar em qualquer sistema, desde que o sistema tenha uma JVM. Isto é particularmente importante para aplicações que executem sobre a internet ou outros ambientes com redes heterogêneas. Mas a arquitetura neutra não é útil apenas no escopo de aplicações em rede. Pode-se desenvolver uma aplicação que com o mesmo código compilado rodem em um PC, Unix ou em um Machintosh;
d) dinâmico e distribuído: o Java é uma linguagem dinâmica, sendo assim qualquer classe pode ser carregada em um interpretador Java, que já esta rodando, em qualquer momento. Estas classes carregadas dinamicamente podem ser instanciadas dinamicamente. O Java é também chamada de linguagem distribuída, isto é, pois disponibiliza suporte a rede em alto nível. Estas características permitem que um código Java seja “baixado” e executado através da internet;
e) Java é uma linguagem simples. Os desenvolvedores do Java tentaram criar uma linguagem que um programador aprendesse rapidamente, assim o número de construções da linguagem foi mantido relativamente baixo;
f) Java foi desenvolvido para que possibilitasse uma escrita extremamente confiável, ou seja, permite desenvolver softwares robustos. A falta de ponteiros é um exemplo de aumento de robustez da linguagem. A manipulação de exceções é outra característica que torna a linguagem Java robusta;
g) segurança: uma das características mais perseguidas no desenvolvimento do Java. Isto é especialmente importante pela natureza distribuída do Java. Num nível mais baixo, segurança anda de mão dada com a robustez, isto é, programas Java não podem esquecer ponteiros de memória, nem ler memória fora dos limites de uma *string*;
h) alta-performance: Java é uma linguagem interpretada, assim sendo, nunca terá a mesma performance que um programa compilado escrito, por exemplo, em C. Mas a velocidade de execução está aumentando, pois um programa compilado na versão 1.0 do Java era cerca de vinte vezes mais lento que um programa compilado em C,

em ambiente Unix, o mesmo programa compilado na versão 1.1 do Java é duas vezes mais rápido que se fosse compilado na versão 1.0;

i) Java é uma linguagem *multithread*, pois suporta várias *threads* em execução ao mesmo tempo que podem cuidar de tarefas diferentes. Um benefício importante da *multithread* é que melhora a performance da interatividade com o usuário em aplicações gráficas.

Segundo Flanagan (1997) todo programa Java consiste em uma definição de classe pública que contém um método chamado *main()*, que é o ponto de entrada de todas as aplicações Java. O método *main()* é o ponto onde o interpretador Java começa a execução o programa.

Segundo Niemeyer (1997) as classes são os blocos para a construção das aplicações Java. Uma classe pode conter métodos, variáveis, códigos de instanciação e outras classes. O quadro 16 mostra a sintaxe para a definição de uma classe em Java.

Quadro 16 – SINTAXE PARA A DEFINIÇÃO DE UM CLASSE EM JAVA

```
class CLASS_NAME extends SUPERCLASS
{
   FIELD_DECLARATION
   METHOD_DECLARATION
}
```

Fonte: Niemyer (1997)

Como pode-se notar no quadro 16, CLASS_NAME representa o nome da classe que está sendo definida. SUPERCLASSE é o nome da superclasse para a classe que está sendo definida, se a classe não possui uma superclasse, o trecho “extends SUPERCLASS” é omitido.

Seguindo a sintaxe da definição da classe tem-se, dentro das chaves FIELD_DECLARATION e METHOD_DECLARATION que representam respectivamente a declaração de atributo e a declaração de método.

A sintaxe para a declaração de um atributo é “VISIBILITY TYPE ATTRIBUTE_NAME;” onde VISIBILITY é a visibilidade do atributo que pode ser *private*, *protected*, *public*, representando respectivamente visibilidade privada, protegida e pública;

TYPE é o tipo de dados do atributo (maiores informações sobre os tipos válidos podem ser encontrados em Niemeyer (1997) e Flanagan (1997)); ATTRIBUTE_NAME é o nome do atributo.

A sintaxe para a declaração de um método é “VISIBILITY TYPE METHOD_NAME(PARAM_LIST){};” onde VISIBILITY e TYPE são idênticos à visibilidade e tipos de dados para atributos; METHOD_NAME é o nome do método; PARAM_LIST é a lista de parâmetros do método no formato “TYPE PARAM_NAME” onde TYPE é idêntico ao tipo de atributo e PARAM_NAME é o nome do parâmetro.

Um exemplo de arquivo contendo código fonte Java é apresentado no quadro 32 do anexo 3.

2.8 BORLAND DELPHI E A LINGUAGEM OBJECT PASCAL

Segundo Pacheco (2000) o Borland Delphi é uma ferramenta *Rapid Application Develpment* (RAD), e também, uma ferramenta para desenvolvimento de aplicações que utilizam banco de dados. O Delphi combina a facilidade de utilização de um ambiente visual de desenvolvimento, a velocidade e o poder de um compilador de 32 bits otimizado, e a capacidade de manipulação de dados através de um mecanismo escalável e robusto.

A base da linguagem do Delphi é uma versão orientada a objetos da linguagem Pascal que a Borland chama de *Object Pascal*. *Object Pascal* proporcionou o aumento de produtividade da linguagem Pascal com a adição de (Pacheco, 2000):

a) manipulação de exceções, que permite detectar e recuperar graciosamente os erros de *runtime*;
b) *Runtime Type Information* (RTTI) que permite determinar o tipo de um objeto em tempo de execução;
c) suporte a interfaces, que faz com que o desenvolvimento no *Component Object Model* (COM) seja fácil e ainda permita construir aplicações com fundamentos robustos;
d) *strings* sem limite de comprimento, que permite escrever códigos manipuladores de *string* sem precisar se preocupar com limite de tamanho;

e) tipo de dados *Currency* para cálculos monetários mais exatos;
f) *assertions* estilo C++ que ajuda a escrever código mais robusto;
g) tipo de dados *Variant* que permite alcançar tecnologias como *Object Linking and Embedding* envolvendo dados “sem-tipo”.

Segundo Pacheco (2000) o Delphi possui um *framework* de classes chamado *Visual Component Library* (VCL). Similarmente a maioria dos *frameworks* de classes para o Microsoft *Windows*, como a *Microsoft Foundation Classes* (MFC), a VCL disponibiliza uma facilidade para a complicada *Application Programing Interface* (API) do *Windows*. A maior vantagem da VCL sobre os outros *frameworks* é a perfeita integração com o ambiente visual de desenvolvimento. Cada componente que se “arrasta” da paleta de componentes do Delphi para um formulário, é um elemento ou controle da VCL.

Segundo Lischner (2000) o *Object Pascal* é uma linguagem modular e o módulo básico é denominado *unit*. Para compilar e *linkar* um programa em Delphi, precisa-se de um arquivo fonte de programa e quaisquer unidades adicionais na forma de fonte ou objeto. O arquivo de programa normalmente é chamado de projeto, pois o projeto pode ser um programa ou uma biblioteca. Cada *unit* está dividida em seções que podem ser: *interface*, *implementation*, *initialization* e *finalization*. Dentro de cada seção pode haver diversas declarações como declaração de tipos, classes, variáveis, procedimento, funções, entre outros.

Segundo Borland (1995) uma classe é uma estrutura que consiste num número fixo de componentes. Os componentes possíveis de uma classe são atributos, métodos e propriedades. Diferentemente de outros tipos de dados, uma classe pode ser declarada apenas da parte da declaração de tipos no extremo de um programa ou unidade. Sendo assim uma classe não pode ser declarada em uma declaração de variáveis, nem dentro de um procedimento, função ou método. O quadro 17 mostra a sintaxe para a declaração de uma classe em *Object Pascal*.

Segundo o quadro 17, CLASS_NAME representa o nome da classe. A palavra reservada “*class*” é obrigatória, o valor entre parênteses indica a superclasse da classe que está sendo declarada. Se os parênteses forem omitidos, a superclasse será a classe TObject da VCL. Uma classe possui três níveis de visibilidade: o privado (private), o protegido

(protected) e o público (public e published). Detalhes sobre a visibilidade podem ser encontrados em Borland (1995).

Quadro 17 – SINTAXE PARA DECLARAÇÃO DE UMA CLASSE EM OBJECT PASCAL

```
CLASS_NAME = class(SUPERCLASS)
Private
     FIELD_DECLARATION
     METHOD_DECLARATION
     PROPERTY_DECLARATION
Protected
     FIELD_DECLARATION
     METHOD_DECLARATION
     PROPERTY_DECLARATION
Public
     FIELD_DECLARATION
     METHOD_DECLARATION
     PROPERTY_DECLARATION
Published
     PROPERTY_DECLARATION
End;
```

Fonte: Borland (1995)

Continuando a definição da classe, FIELD_DECLARATION representa a declaração de atributo. Esta declaração é no formato “ATTRIBUTE_NAME : ATTRIBUTE_TYPE;” onde ATTRIBUTE_NAME é o nome do atributo e ATTRIBUTE_TYPE é o tipo de dados que o atributo representará. Os tipos de dados permitidos podem ser encontrados em Borland (1995).

A declaração METHOD_DECLARATION, define método. O formato é “procedure METHOD_NAME(PARAM_LIST);” ou “function METHOD_NAME(PARAM_LIST) : RETURN_TYPE;”, que representa respectivamente um procedimento e um função, onde METHOD_NAME é o nome do método; PARAM_LIST é a lista de parâmetros do método no mesmo formato que a declaração de atributos. O separador utilizado entre dois parâmetros é a vírgula. No caso do método ser um função RETURN_TYPE representa o tipo de dado de retorno.

A declaração PROPERTY_DECLARATION representa uma propriedade. A sintaxe é “property PROP_NAME: PROP_TYPE read PROP_READFUNC write PROP_WRITEFUNC”, onde PROP_NAME é o nome da propriedade; PROP_TYPE é o tipo

de dado da propriedade; PROP_READFUNC e PROP_WRITEPROC são, respectivamente, os métodos que lêem e escrevem o valor da propriedade.

Um exemplo de arquivo contendo código fonte *Object Pascal* é apresentado no quadro 33 do anexo 4.

3 DESENVOLVIMENTO DO TRABALHO

O presente trabalho resultou na criação de uma metalinguagem, a *Linguagem de Especificação de Linguagens* (LEL) para especificação de classes em linguagens de programação e, também, em um protótipo de software que possibilita gerar código fonte para diversas linguagens a partir do repositório da ferramenta CASE *System Architect*, desde que haja uma especificação (escrita em LEL em um arquivo texto) para a linguagem a ser gerada.

A seguir serão informados detalhes sobre a especificação e implementação da metalinguagem LEL e do protótipo de software.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O objetivo do desenvolvimento deste trabalho foi criar uma ferramenta capaz de gerar código fonte para qualquer linguagem de programação a partir de um diagrama de classes, modelado a partir da ferramenta CASE *System Architect*.

As informações para geração de código fonte serão obtidas de um repositório criado através da ferramenta CASE *System Architect*, para um diagrama de classes.

Para tanto, é necessário que haja um formalismo para especificar o formato do código fonte que se deseja gerar. A partir do formalismo (metalinguagem) um interpretador irá dirigir-se para a criação de um arquivo texto, o qual conterá o código fonte para uma determinada linguagem. Este código fonte é apenas um esqueleto das classes, cabendo ao usuário (programador) do software gerador escrever o código fonte para os métodos, de forma a prover funcionalidade para as classes.

A seguir, será apresentada a especificação do protótipo para o programa descrito.

3.2 ESPECIFICAÇÃO

Nesta seção serão apresentadas as especificações da metalinguagem LEL e do protótipo de software para a geração de código fonte.

3.2.1 METALINGUAGEM LEL

A meta linguagem LEL é uma linguagem livre de contexto, isto é, a gramática de geração desta linguagem é uma gramática livre de contexto. A gramática da metalinguagem LEL está livre de ambigüidade, fatorada à esquerda e sem recursão à esquerda. No quadro 18 é apresentada a especificação através da gramática livre de contexto da metalinguagem LEL em notação BNF.

Quadro 18 – ESPECIFICAÇÃO DA METALINGUAGEM LEL EM BNF

```
<ESPECIFICACAO> ::= <PRODUCAO> <LISTA_PRODUCOES>

<LISTA_PRODUCOES> ::= (<PRODUCAO> <LISTA_PRODUCOES>) | ∈

<PRODUCAO> ::= id '->' <ITEM> <LISTA_ITENS> ';'

<ITEM> ::= <COMANDO> | id | cadeia

<LISTA_ITENS> ::= (<ITEM> <LISTA_ITENS>) | ∈

<COMANDO> ::= '<' <COMANDOS> '>'

<COMANDOS> ::= 'nl' | 'bks' | ('reset' <ELEMENTO>) | ('readnext'
        <ELEMENTO>) | ('geto' <ATRIBUTO>) | <COMANDO_IF> | ('var' id
        <VALOR>)

<COMANDO_IF> ::= 'if' <EXPRESSAO_BOOLEANA> 'then' <LISTA_ITENS>
        <PARTE_ELSE>

<PARTE_ELSE> ::= ('else' <LISTA_ITENS>) | ∈

<ELEMENTO> ::= 'class' | 'attribute' | 'method' | 'parameter'

<ATRIBUTO> ::= <ELEMENTO> '.' id

<EXPRESSAO_BOOLEANA> ::= <EXPRESSAO_SIMPLES> <LISTA_EXPRESSAO_BOOLEANA>

<LISTA_EXPRESSAO_BOOLEANA> ::= (<OPERACAO_RELACIONAL> <EXPRESSAO_SIMPLES>)
        | ∈

<OPERACAO_RELACIONAL> ::= '=' | '<' | '<=' | '>' | '>='

<EXPRESSAO_SIMPLES> ::= <TERMO> <LISTA_EXPRESSAO_SIMPLES>

<LISTA_EXPRESSAO_SIMPLES> ::= ('or' <TERMO>) | ∈

<TERMO> ::= <FATOR> <LISTA_TERMO>

<LISTA_TERMO> ::= ('and' <FATOR>) | ∈

<FATOR> ::= ('(' <EXPRESSAO_BOOLEANA> ')') | ('not' <FATOR>) | <VALOR> |
        <CMD_VALUE>

<VALOR> ::= 'true' | 'false' | numero_int | cadeia

<CMD_VALUE> ::= '<' <CMD_VALUES> '>'

<CMD_VALUES> ::= ('empty' <ELEMENTO>) | <ATRIBUTO> | id
```

No quadro 19 são apresentadas as definições dirigidas pela sintaxe para a metalinguagem LEL, a qual é uma extensão da BNF, sendo que os atributos associados a cada elemento da gramática aparecem entre chaves imediatamente em seguida ao elemento. Na definição dirigida pela sintaxe apresentada no quadro 19, cada produção que tem mais de uma alternativa é escrita repetida, tantas vezes quantas forem as opções para a produção (isto torna a visualização dos atributos mais legível). Por exemplo, a produção "<ITEM> ::= <COMANDO> | *id* | *cadeia*" é escrita três vezes, cada qual com uma opção, ficando representada como "<ITEM> ::= <COMANDO>", "<ITEM> ::= *id*" e "<ITEM> ::= *cadeia*".

Quadro 19 – DEFINIÇÕES DIRIGIDAS PELA SINTAXE DA METALINGUAGEM LEL

```
<ESPECIFICACAO> ::= <PRODUCAO> {ESPECIFICACAO.ptr=PRODUCAO.ptr}
        <LISTA_PRODUCOES> {PRODUCAO.prox=LISTA_PRODUCOES.ptr}

<LISTA_PRODUCOES> ::= <PRODUCAO> {LISTA_PRODUCOES.ptr=PRODUCAO.ptr}
        <LISTA_PRODUCOES1> {PRODUCAO.prox=LISTA_PRODUCOES1.ptr}

<LISTA_PRODUCOES> ::= ∈ {LISTA_PRODUCOES.ptr=nil}

<PRODUCAO> ::= id {se não ExisteProducao(id) então PRODUCAO.ptr=CriaProducao(id) senão ERRO}
        '->' <ITEM> { PRODUCAO.item=ITEM.ptr} <LISTA_ITENS>
        {ITEM.prox=LISTA_ITENS.ptr} ';'

<ITEM> ::= <COMANDO> {ITEM.ptr=COMANDO.ptr}

<ITEM> ::= id {ITEM.ptr=CriaElemento(NaoTerminal, id)}

<ITEM> ::= cadeia {ITEM.ptr=CriaElemento(Literal,cadeia)}

<LISTA_ITENS> ::= <ITEM> {LISTA_ITENS.ptr=ITEM.ptr} <LISTA_ITENS1>
        {ITEM.prox=LISTA_ITENS1.ptr}

<LISTA_ITENS> ::= ∈ {LISTA_ITEM.ptr=nil}

<COMANDO> ::= '<' <COMANDOS> '>' {COMANDO.ptr=COMANDOS.ptr}

<COMANDOS> ::= 'nl' {COMANDOS.ptr=CriaComandoNL}

<COMANDOS> ::= 'bks' {COMANDOS.ptr=CriaComandoBKS}

<COMANDOS> ::= 'reset' <ELEMENTO> {COMANDOS.ptr=CriaComandoReset(ELEMENTO.valor)}

<COMANDOS> ::= 'readnext' <ELEMENTO>
        {COMANDOS.ptr=CriaComandoReadnext(ELEMENTO.valor)}

<COMANDOS> ::= 'geto' <ATRIBUTO> {COMANDOS.ptr=CriaComandoGeto(ATRIBUTO.elemento,
        ATRIBUTO.nome)}
```

```
<COMANDOS> ::= <COMANDO_IF> {COMANDOS.ptr=COMANDO_IF.ptr}

<COMANDOS> ::= 'var' id <VALOR> {CriaVar(id, VALOR.tipo);
        COMANDOS.ptr=CriaComandoVar(id,VALOR.valor)}

<COMANDO_IF> ::= 'if' <EXPRESSAO_BOOLEANA> 'then' <LISTA_ITENS>
        <PARTE_ELSE> {COMANDO_IF.ptr=CriaComandoIF(EXPRESSAO_BOOLEANA.ptr,
        LISTA_ITENS.ptr, PARTE_ELSE.ptr)}

<PARTE_ELSE> ::= 'else' <LISTA_ITENS> {PARTE_ELSE.ptr=LISTA_ITENS.ptr}

<PARTE_ELSE> ::= ∈ {PARTE_ELSE.ptr=nil}

<ELEMENTO> ::= 'class' {ELEMENTO.valor=class}

<ELEMENTO> ::= 'attribute' {ELEMENTO.valor=attribute}

<ELEMENTO> ::= 'method' {ELEMENTO.valor=method}

<ELEMENTO> ::= 'parameter' {ELEMENTO.valor=parameter}

<ATRIBUTO> ::= <ELEMENTO> {ATRIBUTO.elemento=ELEMENTO.valor} '.' Id
        {ATRIBUTO.nome=id; ATRIBUTO.tipo=PegaTipoAtributo(ATRIBUTO.elemento,
        ATRIBUTO.nome)}

<EXPRESSAO_BOOLEANA> ::= <EXPRESSAO_SIMPLES>
        {LISTA_EXPRESSAO_BOOLEANA.i=EXPRESSAO_SIMPLES.ptr;
        LISTA_EXPRESSAO_BOOLEANA.tipo= EXPRESSAO_SIMPLES.tipo}
        <LISTA_EXPRESSAO_BOOLEANA>
        {EXPRESSAO_BOOLEANA.ptr=LISTA_EXPRESSAO_BOOLEANA.s;
        EXPRESSAO_BOOLEANA.tipo= LISTA_EXPRESSAO_BOOLEANA.tipo}

<LISTA_EXPRESSAO_BOOLEANA> ::= <OPERACAO_RELACIONAL> <EXPRESSAO_SIMPLES> { se
        LISTA_EXPRESSAO_BOOLEANA.tipo=EXPRESSAO_SIMPLES.tipo então inicio
        LISTA_EXPRESSAO_BOOLEANA.s=CriaNo(LISTA_EXPRESSAO_BOOLEANA.i,
        OPERACAO_RELACIONAL.valor, EXPRESSAO_SIMPLES.ptr);
        LISTA_EXPRESSAO_BOOLEANA.tipo=Booleano fim senão ERRO}

<LISTA_EXPRESSAO_BOOLEANA> ::= ∈
        {LISTA_EXPRESSAO_BOOLEANA.s=LISTA_EXPRESSAO_BOOLEANA.i}

<OPERACAO_RELACIONAL> ::= '=' {OPERACAO_RELACIONAL.valor='='}

<OPERACAO_RELACIONAL> ::= '<' {OPERACAO_RELACIONAL.valor='<'}

<OPERACAO_RELACIONAL> ::= '<=' {OPERACAO_RELACIONAL.valor='<='}

<OPERACAO_RELACIONAL> ::= '>' {OPERACAO_RELACIONAL.valor='>'}

<OPERACAO_RELACIONAL> ::= '>=' {OPERACAO_RELACIONAL.valor='>='}

<EXPRESSAO_SIMPLES> ::= <TERMO> {LISTA_EXPRESSAO_SIMPLES.i=TERMO.ptr;
        LISTA_EXPRESSAO_SIMPLES.tipo=TERMO.tipo} <LISTA_EXPRESSAO_SIMPLES>
```

{EXPRESSAO_SIMPLES.ptr= LISTA_EXPRESSAO_SIMPLES.s;
EXPRESSAO_SIMPLES.tipo=LISTA_EXPRESSAO_SIMPLES.tipo}

<LISTA_EXPRESSAO_SIMPLES> ::= 'or' <TERMO> *{**se** (LISTA_EXPRESSAO_SIMPLES.tipo=booleano) **e** (TERMO.tipo=booleano) **então inicio** LISTA_EXPRESSAO_SIMPLES.s=CriaNo(LISTA_EXPRESSAO_SIMPLES.i,'or', TERMO.ptr); LISTA_EXPRESSAO_SIMPLES.tipo=Booleano **fim senão** ERRO}*

<LISTA_EXPRESSAO_SIMPLES> ::= ∈
{LISTA_EXPRESSAO_SIMPLES.s=LISTA_EXPRESSAO_SIMPLES.i}

<TERMO> ::= <FATOR> *{LISTA_TERMO.i=FATOR.ptr; LISTA_TERMO.tipo=FATOR.tipo}*
LISTA_TERMO *{TERMO.ptr=LISTA_TERMO.s; TERMO.tipo=LISTA_TERMO.tipo}*

<LISTA_TERMO> ::= 'and' <FATOR> *{ **se** (LISTA_TERMO.tipo=booleano) **e** (FATOR.tipo=booleano) **então inicio** LISTA_TERMO.s=CriaNo(LISTA_TERMO.i,'and',FATOR.ptr); LISTA_TERMO.tipo=Booleano **fim senão** ERRO}*

<LISTA_TERMO> ::= ∈ *{LISTA_TERMO.s= LISTA_TERMO.i}*

<FATOR> ::= '(' <EXPRESSAO_BOOLEANA> ')' *{FATOR.ptr=EXPRESSAO_BOOLEANA.ptr; FATOR.tipo= EXPRESSAO_BOOLEANA.tipo}*

<FATOR> ::= 'not' <FATOR> *{FATOR.ptr=CriaNo(FATOR.ptr,'not',nil); FATOR.tipo=Booleano}*

<FATOR> ::= <VALOR> *{FATOR.ptr=Valor.ptr; FATOR.tipo=VALOR.tipo}*

<FATOR> ::= <CMD_VALUE> *{FATOR.ptr=CMD_VALUE.ptr; FATOR.tipo=CMD_VALUE.tipo}*

<VALOR> ::= 'true' *{Valor.ptr=CriaFolha(Booleano, True); VALOR.tipo=Booleano}*

<VALOR> ::= 'false' *{Valor.ptr=CriaFolha(Booleano, False); VALOR.tipo=Booleano }*

<VALOR> ::= *numero_int* *{Valor.ptr=CriaFolha(Numero, numero_int); VALOR.tipo=Numerico}*

<VALOR> ::= *cadeia* *{Valor.ptr=CriaFolha(Caracter, cadeia); VALOR.tipo=Cadeia}*

<CMD_VALUE> ::= '<' <CMD_VALUES> '>' *{CMD_VALUE.ptr=CMD_VALUES.ptr; CMD_VALUE.tipo=CMD_VALUES.tipo}*

<CMD_VALUES> ::= 'empty' <ELEMENTO> *{CMD_Values.ptr=CriaFolhaEmpty(ELEMENTO.valor); CMD_VALUES.tipo=Booleano}*

<CMD_VALUES> ::= <ATRIBUTO> *{CMD_Values.ptr=CriaFolhaATRIBUTO(ATRIBUTO.elemento, ATRIBUTO.nome); CMD_VALUES.tipo=ATRIBUTO.tipo}*

<CMD_VALUES> ::= *id* *{CMD_Values.ptr=CriaFolhaVar(id); CMD_VALUES.tipo=PegaTipoVar(id)}*

3.2.2 PROTÓTIPO DE GERAÇÃO DE CÓDIGO FONTE

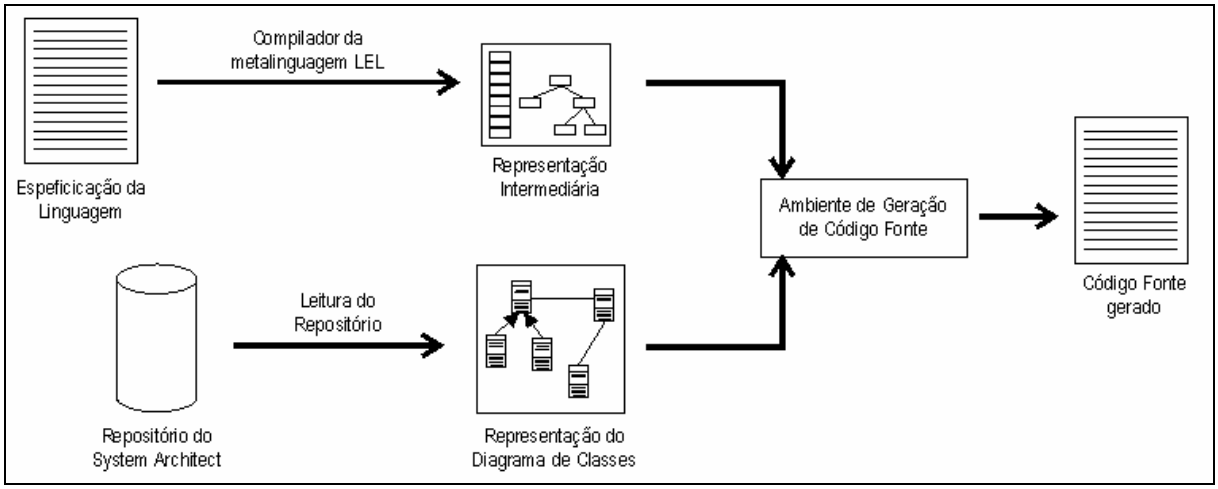
O protótipo de software permite a geração de código fonte para as classes, a partir da especificação de uma linguagem escrita na metalinguagem LEL, obtendo informações do repositório da ferramenta CASE *System Architect*.

O protótipo foi dividido em cinco partes distintas, as quais são:

a) a representação, em memória, de um diagrama de classes;
b) a leitura do repositório da ferramenta CASE *System Architect*;
c) o compilador da metalinguagem LEL;
d) a representação intermediária da especificação da linguagem;
e) o ambiente de geração de código fonte.

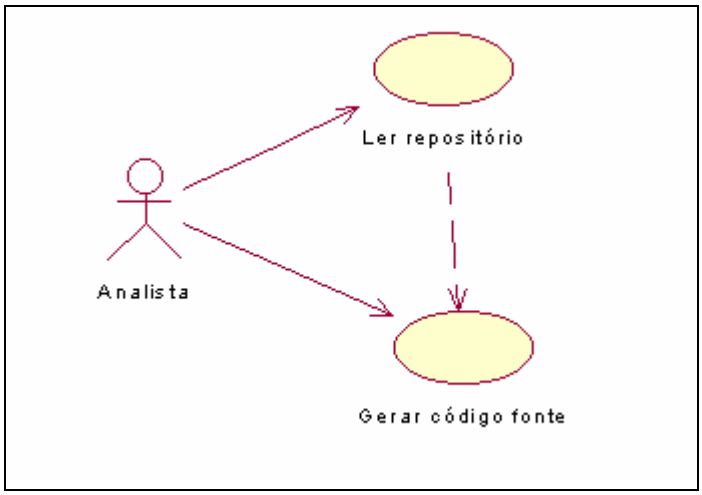
A interação das partes componente do protótipo está representada na fig. 17. Como pode-se observar, primeiramente é feita a leitura do diagrama de classe que se encontra no repositório da ferramenta CASE *System Architect*, gerando a representação em memória do diagrama de classes. Em seguida a especificação que se encontra em um arquivo texto é compilada pelo compilador da metalinguagem LEL, gerando a representação intermediária da especificação da linguagem. Por fim, o ambiente de geração de código fonte utiliza as representações do diagrama de classes e da especificação da linguagem para gerar o código fonte.

Figura 17 – INTERAÇÃO DAS PARTES COMPONENTES DO PROTÓTIPO



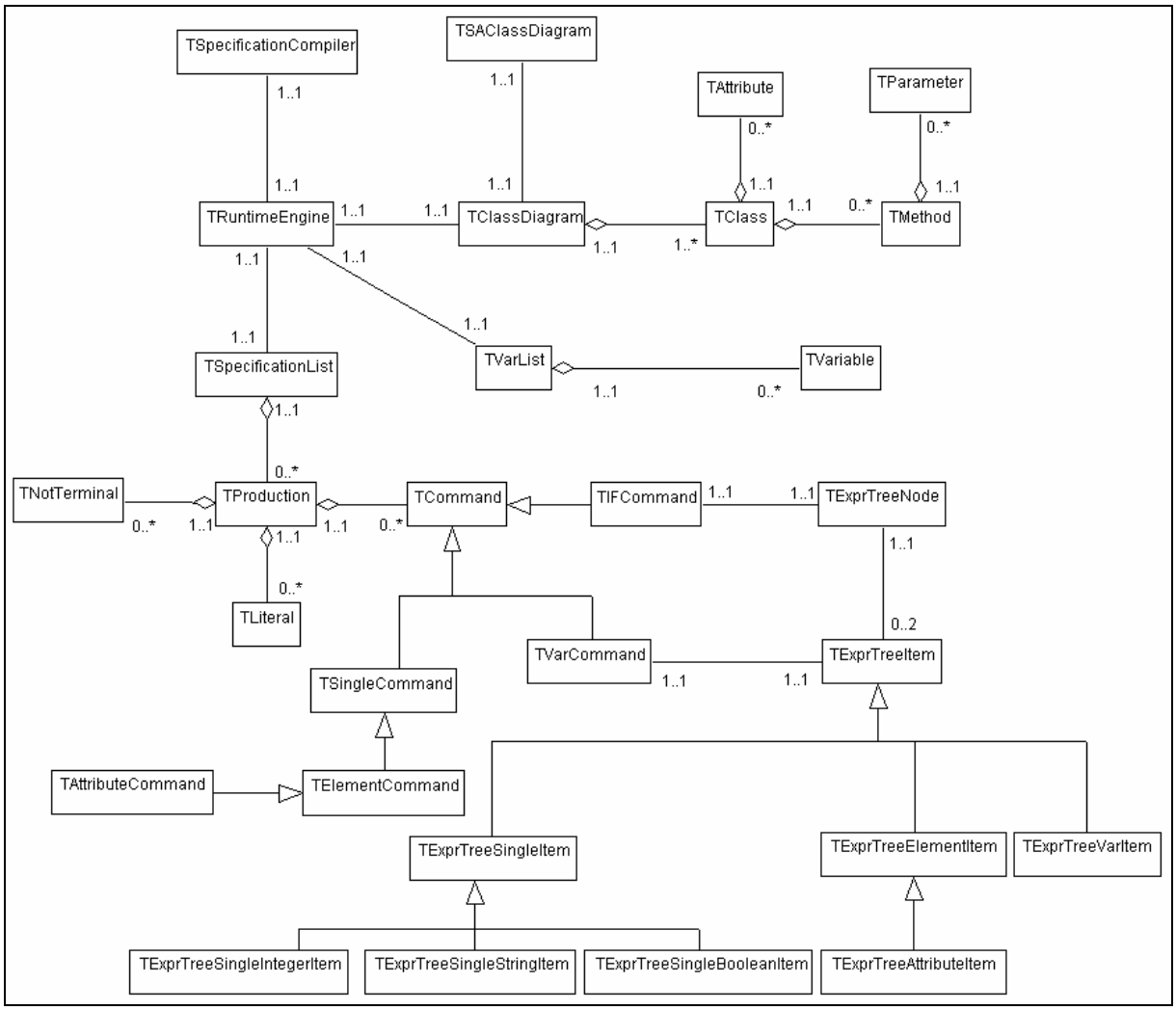
O protótipo foi desenvolvido utilizando a orientação a objetos, sendo assim, cada parte componente do protótipo é representada através de um diagrama de classes utilizando a notação da UML. A modelagem em UML do protótipo feita utilizando a ferramenta CASE *Rational Rose* ao invés do *System Architect*, devido a sua facilidade de utilização em relação ao *System Architect*. A fig. 18 apresenta o diagrama de caso de uso para o protótipo. Existem dois casos de uso sendo que no primeiro o analista faz a leitura do repositório e no segundo o analista gera o código fonte das classes que foram lidas do repositório, gerando uma dependência entre os dois casos de uso.

Figura 18 – DIAGRAMA DE CASO DE USO



A fig. 19 apresenta o diagrama de classes completo do protótipo. Para efeito de simplificação do diagrama de classes, estão sendo apresentadas apenas as classes e seus respectivos relacionamentos. Mais adiante todas as classes serão apresentadas exibindo seus atributos e métodos, agrupadas conforme a sua funcionalidade, isto é, conforme a divisão apresentada na fig. 17.

Figura 19 – DIAGRAMA DE CLASSES DO PROTÓTIPO



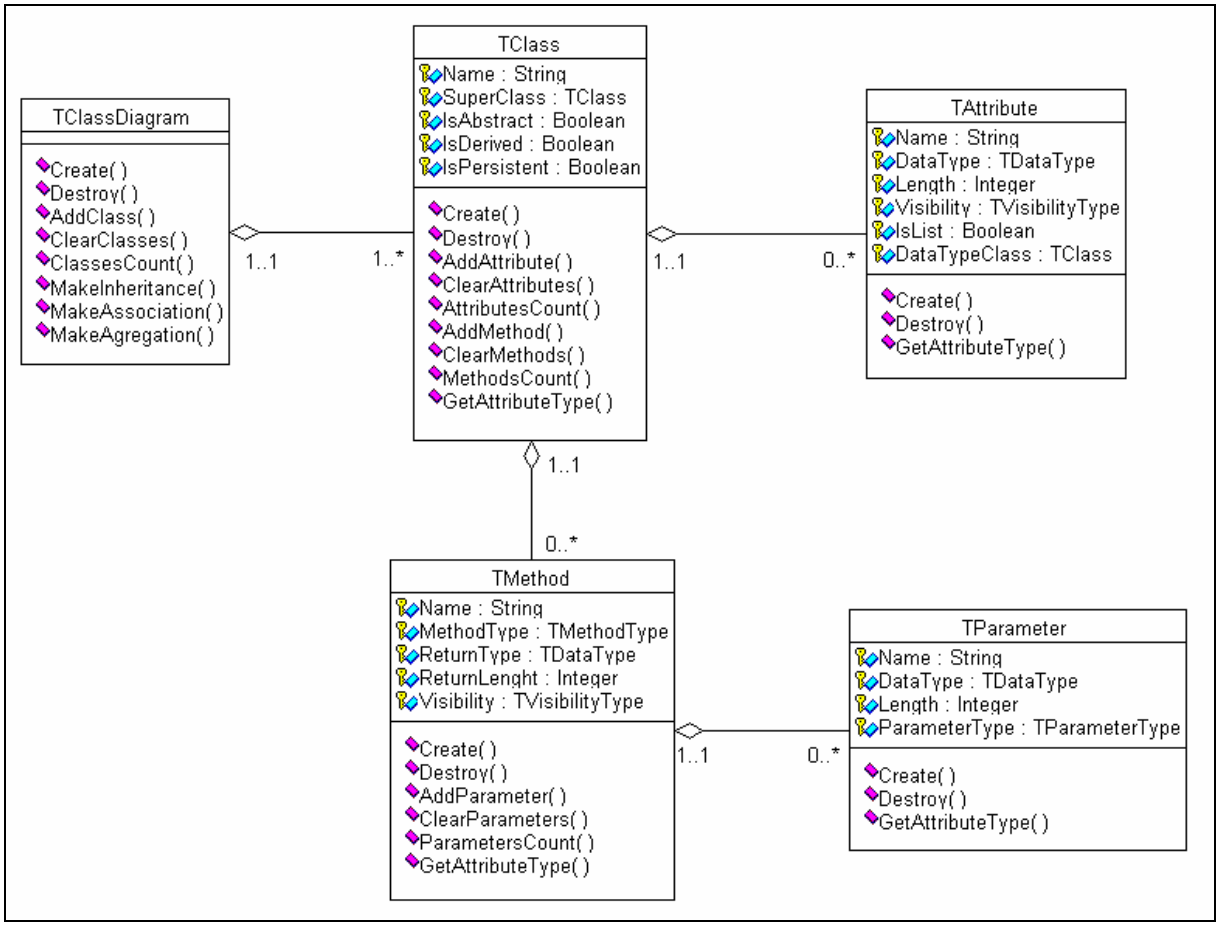
3.2.2.1 REPRESENTAÇÃO DO DIAGRAMA DE CLASSES

A fig. 20 apresenta o diagrama de classes para a representação do diagrama de classes em memória, gerado a partir da leitura do repositório da ferramenta CASE *System Architect*.

A classe central é a TClassDiagram, onde são adicionadas as classes do diagrama, bem como, onde são feitas as relações de herança, associação e agregação entre as classes do diagrama. A classe TClassDiagram não possui atributos, apenas métodos que permitem, como já foi mencionado anteriormente, adicionar classes e fazer relações entre as classes.

A classe TClass representa uma classe que contém atributos, representados pela classe TAttribute, e os métodos, representados pela classe TMethod. Cada método, ainda pode ou não possuir parâmetros.

Figura 20 – DIAGRAMA DE CLASSES DA REPRESENTAÇÃO EM MEMÓRIA DE UM DIAGRAMA DE CLASSES



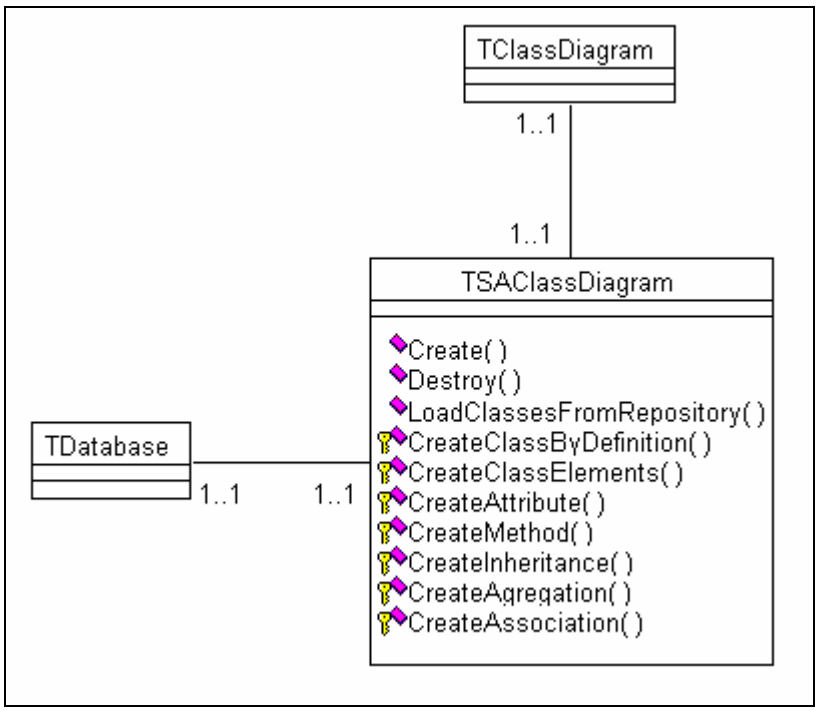
3.2.2.2 LEITURA DO REPOSITÓRIO DO SYSTEM ARCHITECT

A fig. 21 apresenta o diagrama de classes para leitura do repositório da ferramenta CASE *System Architect*.

A classe principal desta parte do protótipo é a classe TSAClassDiagram que é responsável pela leitura do repositório do *System Architect*. A classe associada TClassDiagram não é detalhada no diagrama de classes da fig. 21, pois está especificada na fig. 20. A classe TDatabase é uma classe componente da VCL do ambiente de programação Delphi que disponibiliza um conjunto de funções para a manipulação de bases de dados, que

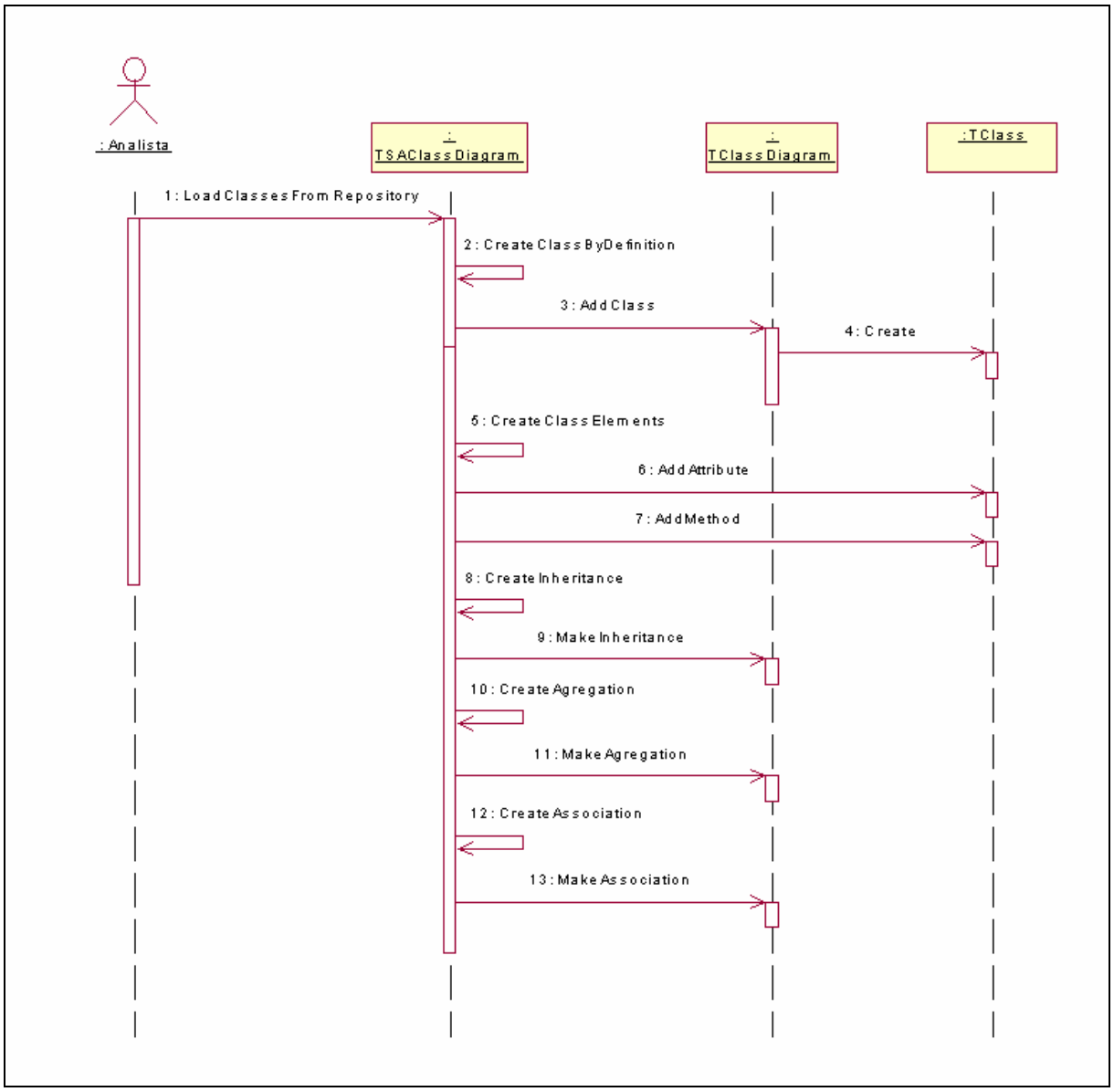
se fazem necessárias para a leitura dos arquivos ENTITY.DBF e RELATN.DBF que compõe o repositório do *System Architect*. Maiores informações sobre as funcionalidades da classe TDatabase podem ser encontradas na ajuda *on-line* do ambiente Delphi, bem como em Pacheco (2000) e em Lichner (2000).

Figura 21 - DIAGRAMA DE CLASSES PARA LEITURA DO REPOSITÓRIO DO SYSTEM ARCHITECT



Na fig. 22 é apresentado o diagrama de seqüência para a leitura do repositório, onde é apresentada a seqüência das chamadas de métodos e a interação entre as classes.

Figura 22 – DIAGRAMA DE SEQUÊNCIA PARA A LEITURA DO REPOSITÓRIO DO SYSTEM ARCHITECT



3.2.2.3 COMPILADOR DA LINGUAGEM LEL

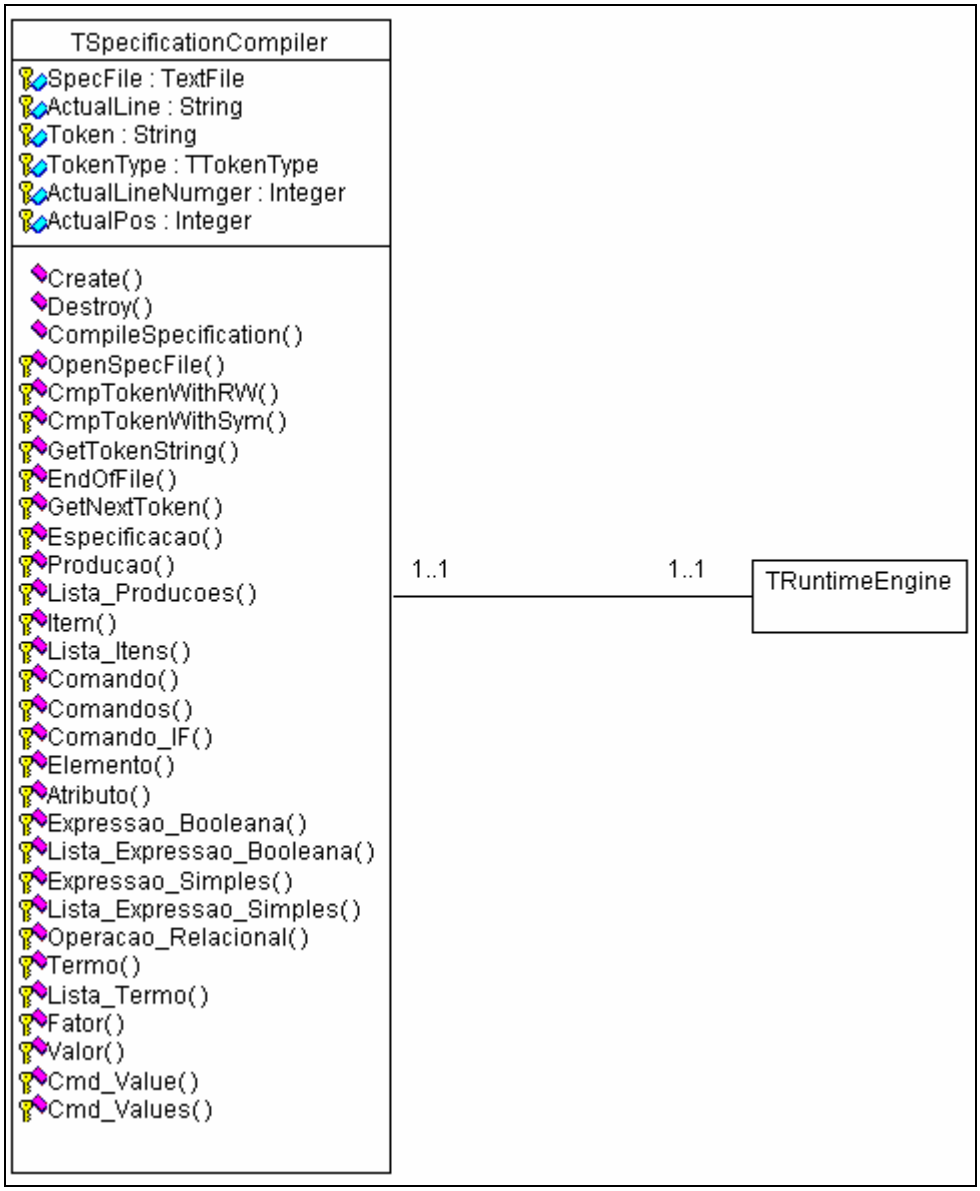
A fig. 23 apresenta o digrama de classes do compilador da linguagem LEL.

O compilador da linguagem LEL é composto de apenas uma classe, a TSpecificationCompiler. Nesta classe estão contidos o analisador léxico, o analisador sintático e o analisador semântico da linguagem LEL. Isso se deve pelo fato do analisador sintático dirigir o processo de compilação, pois é ele quem faz chamadas ao analisador léxico,

assim como é dentro da implementação dos métodos do analisador sintático que se encontram muitas das ações semânticas. Ainda quando ocorre algum erro sintático, o compilador exibe a linha e coluna do arquivo texto da especificação (que são informações pertencentes ao analisador léxico) onde se detectou o erro sintático.

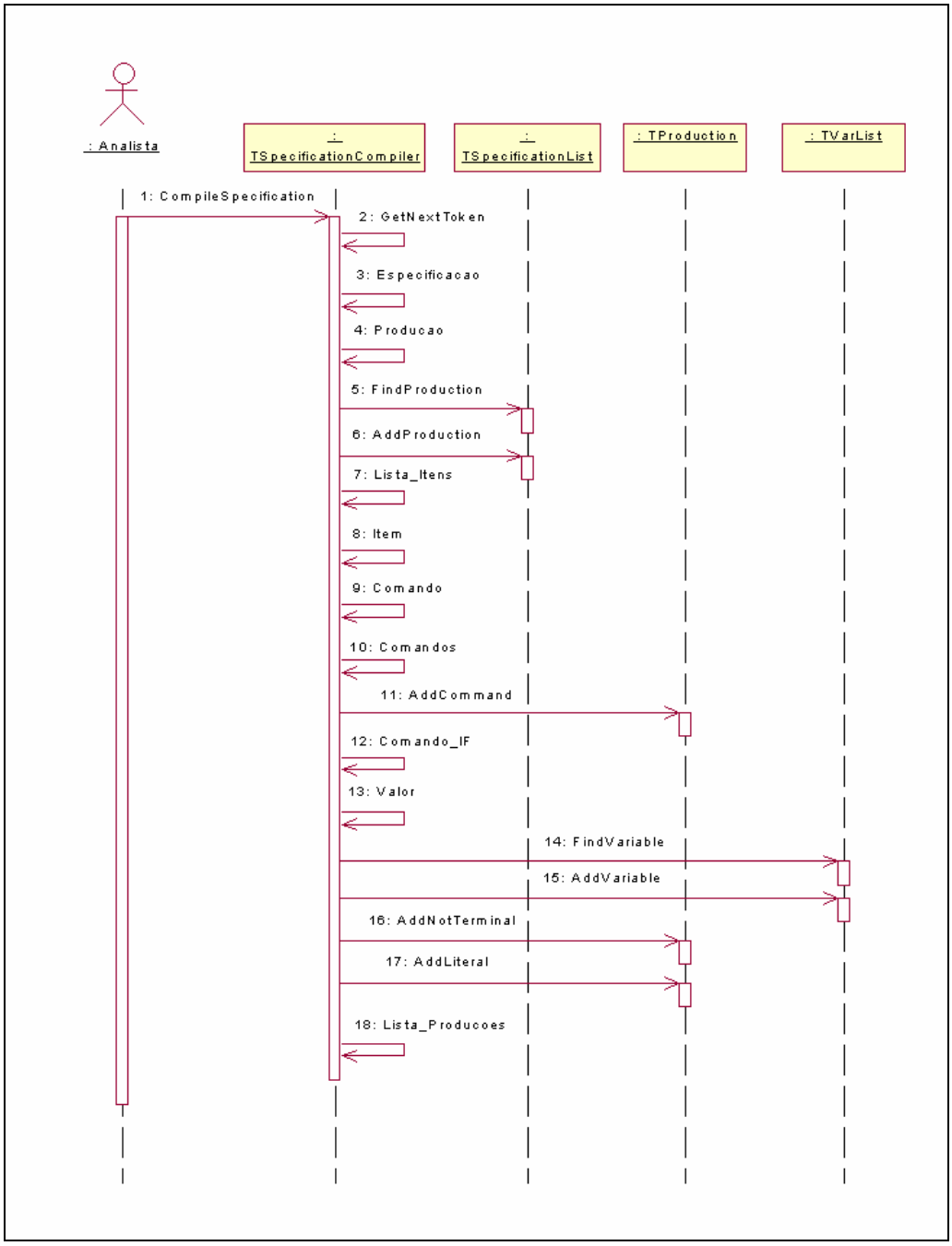
A classe TRuntimeEngine está associada à classe TSpecificationCompiler. A classe TRuntimeEngine é responsável pela geração do código fonte, a partir da representação intermediária que o compilador da linguagem LEL gera. Detalhes sobre a classe TRuntimeEngine serão descritos na sessão sobre a geração de código fonte.

Figura 23 – DIAGRAMA DE CLASSES DO COMPILADOR DA LINGUAGEM LEL



A fig. 24 apresenta o diagrama de seqüência para o compilador da linguagem LEL.

Figura 24 – DIAGRAMA DE CLASSES DO COMPILADOR DA LINGUAGEM LEL



O analisador léxico foi implementado usando os atributos SpecFile, ActualLine, Token, TokenType, ActualLineNumber, ActualPos e os métodos GetNextToken e EndOfFile.

O analisador sintático e o semântico foram implementados através dos métodos Especificação, Producao, Lista_Producoes, Item, Lista_Itens, Comando, Comandos, Comando_IF, Elemento, Atributo, Expressao_Booleana, Lista_Expressao_Booleana, Expressao_Simples, Lista_Expressao_Simples, Operacao_Relacional, Termo, Lista_Termo, Fator, Valor, Cmd_Value e Cmd_Values. As ações semânticas, na sua grande maioria, estão representadas como parâmetros dos métodos.

Durante a fase de compilação a representação intermediária da especificação é criada, ou seja, conforme o compilador vai percorrendo o texto fonte da especificação ele vai criando a representação intermediária.

3.2.2.4 REPRESENTAÇÃO INTERMEDIÁRIA DA ESPECIFICAÇÃO DA LINGUAGEM

Para uma melhor visualização dos métodos e dos atributos das classes o diagrama de classes para a representação intermediária da especificação da linguagem está apresentado em duas figuras (fig.25 e fig.26). Na fig. 25 são apresentadas as classes que representam os elementos da linguagem. As classes na fig. 26 são responsáveis pela avaliação de expressões em comandos de condição, como o comando “<if ... then ... else ...>”.

Na fig. 25 a classe TRuntimeEngine possui uma lista de produções (representadas pela classe TSpecificationList) e uma lista de variáveis (representadas pela classe TVarList). Cada produção (classe TProduction) pode conter, em seu lado direito, vários comandos (TCommand), vários literais (TLiteral) e não-terminais (TNotTerminal). Como a classe TCommand é uma classe abstrata (não pode ser instanciada), os comandos são representados por suas subclasses:

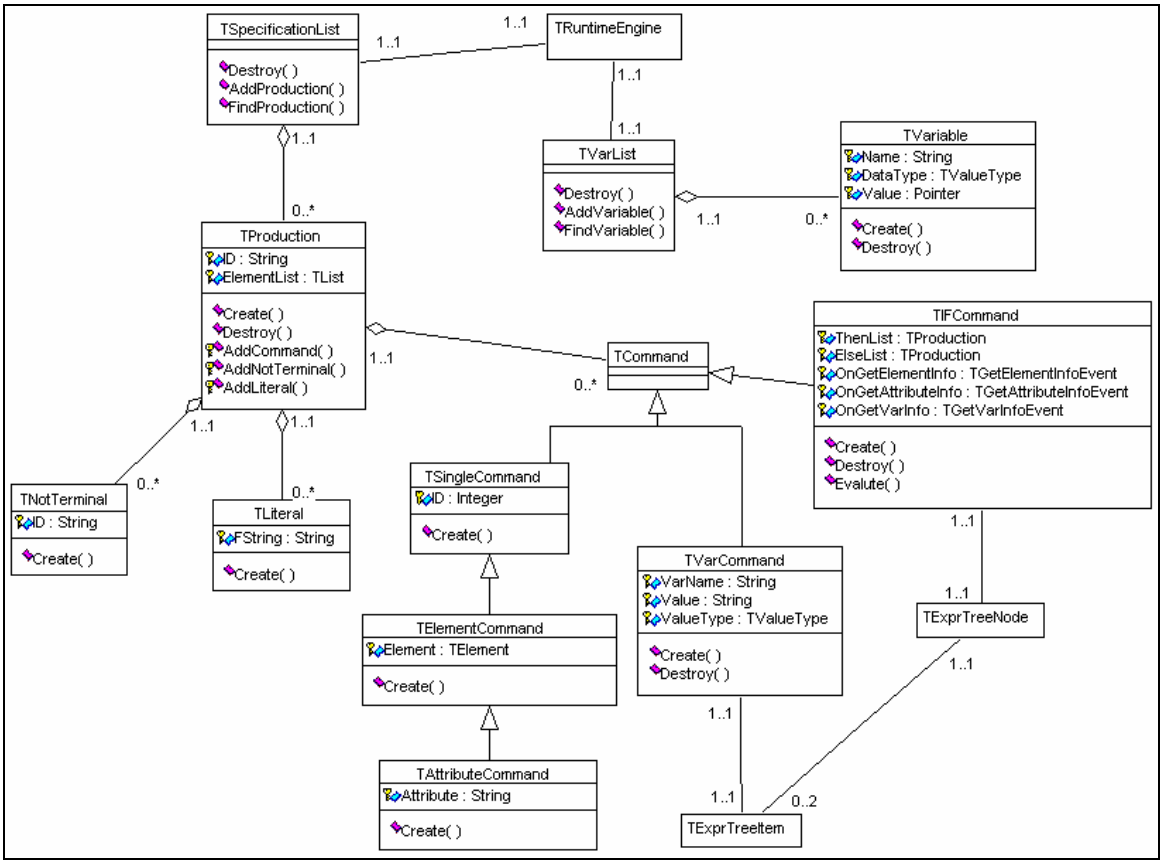
a) TSingleCommand, que representa um comando sem argumentos, como o comando “<bks>”;
b) TElementCommand, que representa comandos que possuem como argumento algum elemento da linguagem. Como exemplo pode-se citar o comando “<readnext Class>”;

c) TAttributeCommand, que representa comandos que possuem como argumentos um elemento e um atributo deste elemento. O comando "<geto class.name>" é um exemplo;

d) TVarCommand, que representa um comando que armazena um valor em uma variável temporária no ambiente de geração de código;

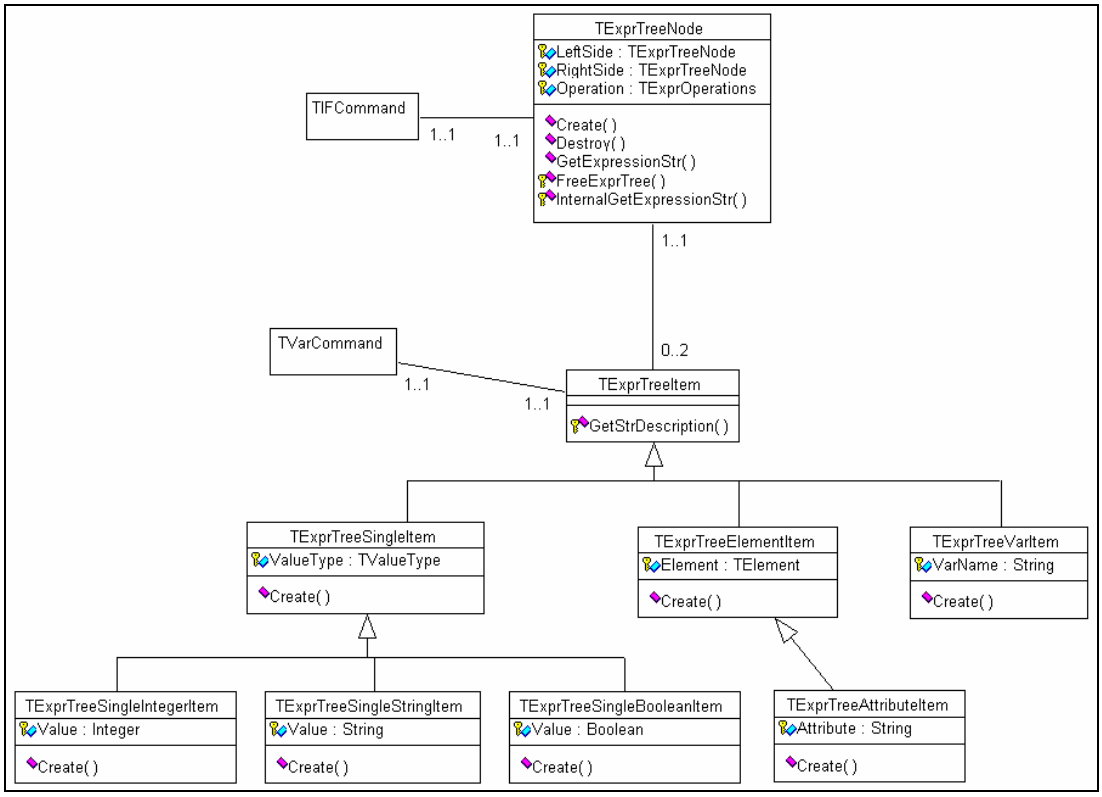
e) TIFCommand, que representa um comando condicional.

Figura 25 – DIAGRAMA DE CLASSES PARA A REPRESENTAÇÃO INTERMEDIÁRIA DA ESPECIFICAÇÃO DA LINGUAGEM



A classe principal da avaliação de expressões em um comando condicional (fig. 26) é a classe TExprTreeNode que representa um nó da árvore de avaliação de expressões. A raiz desta árvore esta associada à classe TIFCommand. A classe TExprTreeItem representa uma folha da árvore de expressões. Uma folha pode conter um valor inteiro (TExprTreeSingleIntegerItem), uma cadeia (TExprTreeSingleStringItem), um valor booleano (TExprTreeSingleBooleanItem), um elemento (TExprTreeElementItem) ou uma variável (TExprTreeVarItem).

Figura 26 - DIAGRAMA DE CLASSES PARA A REPRESENTAÇÃO INTERMEDIÁRIA DA ESPECIFICAÇÃO DA LINGUAGEM



3.2.2.5 AMBIENTE DE GERAÇÃO DE CÓDIGO FONTE

A fig. 27 apresenta o diagrama de classes para o ambiente de geração de código fonte.

A classe TRuntimeEngine é a principal responsável pela geração de código fonte, a qual está associada a:

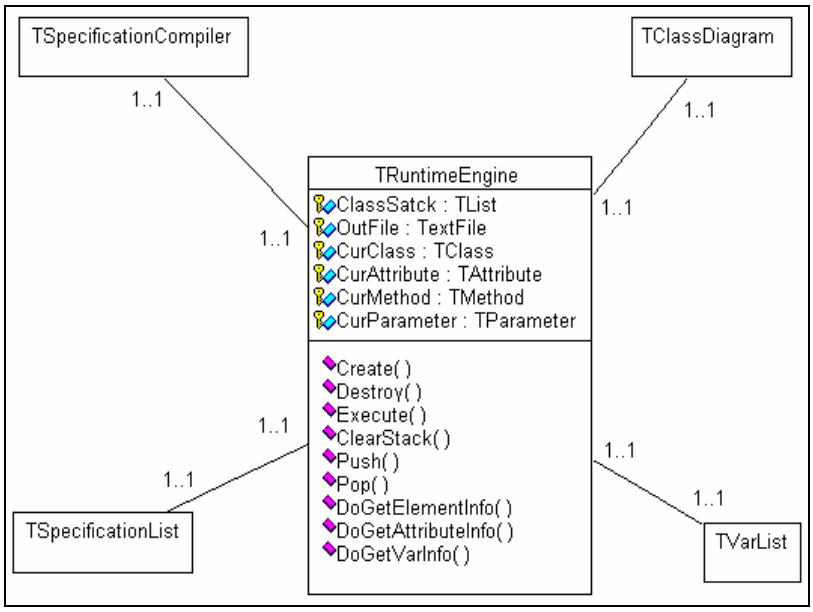
a) um diagrama de classes, que será usado como fonte de dados referentes ao diagrama de classes, ou seja, fonte de informações sobre as classes que estão no repositório;

b) um compilador da linguagem LEL, que é encarregado de preencher o TRuntimeEngine com a representação intermediária da linguagem especificada;

c) uma lista de produções (TSpecificationList) que é criada pelo compilador LEL;

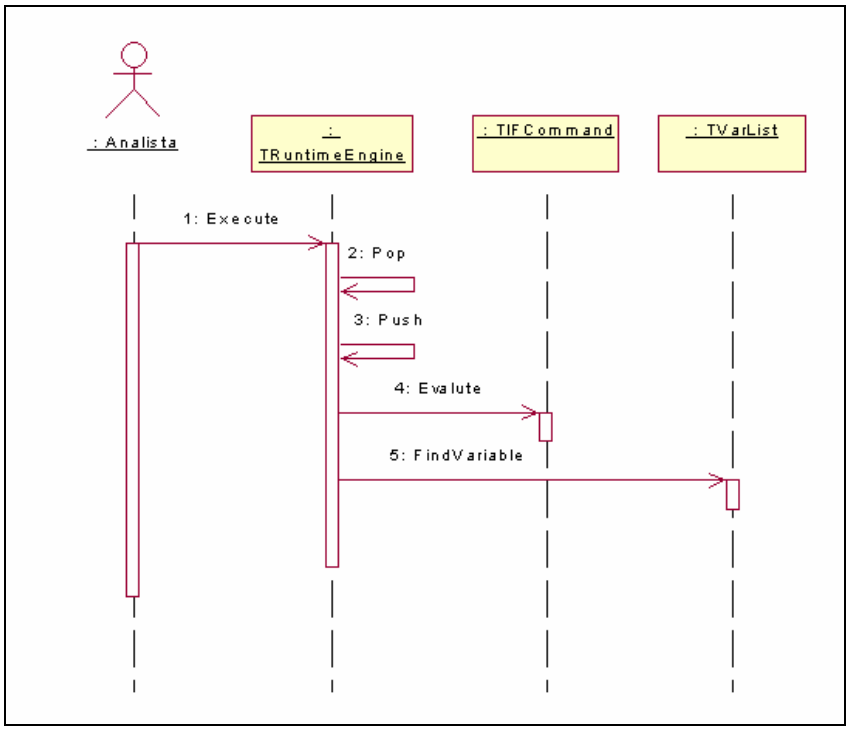
d) uma lista de variáveis, também preenchida pelo compilador LEL, com base nas variáveis encontradas no programa fonte escrito em linguagem LEL.

Figura 27 – DIAGRAMA DE CLASSE PARA O AMBIENTE DE GERAÇÃO DE CÓDIGO FONTE



A fig. 28 apresenta o diagrama de seqüência para a geração de código fonte. O ambiente de geração de código percorre a representação intermediária da linguagem, escrevendo um arquivo com o texto resultante da linguagem especificada no texto fonte.

Figura 28 – DIAGRAMA DE SEQUÊNCIA PARA A GERAÇÃO DE CÓDIGO FONTE



3.3 IMPLEMENTAÇÃO

Considerações sobre as técnicas utilizadas para implementação do protótipo, bem como a forma de operação do mesmo, serão apresentadas nesta seção.

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O protótipo de software que permite a geração de código fonte através do repositório da ferramenta CASE *System Architect* foi desenvolvido utilizando o ambiente de desenvolvimento Borland Delphi 5.0.

No desenvolvimento do protótipo foi utilizada a orientação a objetos, permitindo uma separação bem clara das partes componentes do protótipo, além de facilitar a manutenção do código, pelo fato das funcionalidades estarem sempre agrupadas nas classes que compõe o protótipo.

A seguir serão apresentados detalhes sobre a leitura do repositório, sobre o compilador da linguagem LEL e sobre o ambiente de geração de código fonte.

3.3.1.1 LEITURA DO REPOSITÓRIO DO SYSTEM ARCHITECT

A leitura do repositório do *System Architect* baseia-se no diagrama de seqüência da fig. 22. No quadro 20 é apresentado o trecho de código do método LoadClassesFromRepository da classe TSAClassDiagram, responsável pela leitura do repositório.

No quadro 20 todas as informações referentes a um elemento que está armazenado no repositório encontram-se armazenadas no campo DESCRIPTN da tabela ENTITY.DBF. O campo DESCRIPTN é um campo *memo*, ou seja, pode armazenar várias linhas de texto, sendo que as informações estão armazenadas em duas linhas, uma contendo o identificador entre três pares de colchetes e outra contendo o valor. Por exemplo, para saber se uma classe é abstrata, verifica-se a linha contendo o identificador “[[[Abstract]]]” e na linha seguinte o valor “T” para verdadeiro e “F” para falso.

Quadro 20 – MÉTODO LOADCLASSESFROMREPOSITORY

```
procedure TSAClassDiagram.LoadClassesFromRepository(const Path: String);
var SQLClasses: TQuery;
    xClass: TClass;
begin
```

```
  if   FDatabase <> Nil then
       FDatabase.Free;
  FDatabase:=TDatabase.Create(Nil);
  try
    FDatabase.DriverName:=STR_DRIVERNAME;
    FDatabase.DatabaseName:=STR_DATABASENAME;
    FDatabase.Params.Add(STR_PATH + Path);
    //FDatabase.Directory:=Path;
    FDatabase.Open;

    SQLClasses:=TQuery.Create(Nil);
    try
      SQLClasses.DatabaseName:=STR_DATABASENAME;
      SQLClasses.SQL.Clear;
      SQLClasses.SQL.Add(STR_SELECT_ENTITY);
      SQLClasses.Prepare;
      // Procura as classes
      SQLClasses.Params[0].AsString:=STR_DEFINITION;
      SQLClasses.Params[1].AsString:=STR_CLASS;
      SQLClasses.Open;
      SQLClasses.First;
      FClassDiagram.ClearClasses;
      while not SQLClasses.Eof do
            begin
              xClass:=CreateClassByDefinition(SQLClasses.FieldByName(STR_FLD_NAME).AsString,
SQLClasses.FieldByName(STR_FLD_DESCRIPTN).AsString);
              CreateClassElements(xClass, SQLClasses.FieldByName(STR_FLD_ID).AsInteger);
              SQLClasses.Next;
            end;
      // Procura as heranças
      SQLClasses.Close;
      SQLClasses.Params[0].AsString:=STR_SYMBOL;
      SQLClasses.Params[1].AsString:=STR_INHERITED;
      SQLClasses.Open;
      SQLClasses.First;
      while not SQLClasses.Eof do
            begin
              CreateInheritance(SQLClasses.FieldByName(STR_FLD_DESCRIPTN).AsString);
              SQLClasses.Next;
            end;
      // Procura as associção
      SQLClasses.Close;
      SQLClasses.Params[0].AsString:=STR_SYMBOL;
      SQLClasses.Params[1].AsString:=STR_ASSOCIATION;
      SQLClasses.Open;
      SQLClasses.First;
      while not SQLClasses.Eof do
            begin
              if ((SQLClasses.FieldByName(STR_FLD_FROMASSC).AsString = STR_AGREGATION_ALL)
                   or(SQLClasses.FieldByName(STR_FLD_FROMASSC).AsString= STR_AGREGATION_PART))
              and ((SQLClasses.FieldByName(STR_FLD_TOASSC).AsString = STR_AGREGATION_ALL)
                    or (SQLClasses.FieldByName(STR_FLD_TOASSC).AsString= STR_AGREGATION_PART))
then
                 begin
                   CreateAgregation(SQLClasses.FieldByName(STR_FLD_FROMASSC).AsString,
                                    SQLClasses.FieldByName(STR_FLD_DESCRIPTN).AsString);
                 end
              else
                 begin
                   CreateAssociation(SQLClasses.FieldByName(STR_FLD_FROMASSC).AsString,
                                     SQLClasses.FieldByName(STR_FLD_TOASSC).AsString,
                                     SQLClasses.FieldByName(STR_FLD_DESCRIPTN).AsString);
                 end;
              SQLClasses.Next;
            end;
    finally
      SQLClasses.Free;
    end;

  finally
    FDatabase.Free;
    FDatabase:=Nil;
```

```
  end;
end;
```

3.3.1.2 COMPILADOR DA LINGUAGEM LEL

O compilador da linguagem LEL foi escrito conforme o diagrama de seqüência da fig. 24. No quadro 21 é apresentado o método Valor da classe TSpecificationCompiler, que reconhece o não terminal VALOR, da gramática especificada no quadro 18. O método Valor representa a essência da implementação do compilador LEL, pois além da análise sintática, também faz a análise semântica, através da especificação de atributos herdados e sintetizados.

Quadro 21 – MÉTODO VALOR

```
function TSpecificationCompiler.Valor(var ptr: Pointer;
  var ValueType: TValueType): Boolean;
begin
  if   CmpTokenWithRW(RW_TRUE) then
       begin
         Result:=True;
         ptr:=TExprTreeSingleBooleanItem.Create(True);
         ValueType:=vtBoolean;
       end
  else
  if   CmpTokenWithRW(RW_FALSE) then
       begin
         Result:=True;
         ptr:=TExprTreeSingleBooleanItem.Create(False);
         ValueType:=vtBoolean;
       end
  else
  if   FTokenType = ttNumber then
       begin
         Result:=True;
         ptr:=TExprTreeSingleIntegerItem.Create(StrToInt(FToken));
         ValueType:=vtNumber;
       end
  else
  if   FTokenType = ttString then
       begin
         Result:=True;
         ptr:=TExprTreeSingleStringItem.Create(GetTokenString);
         ValueType:=vtString;
       end
  else
       raise Exception.CreateFmt(STR_TOKEN_NOT_EXPECTED, [FToken,
                                                          FActualLineNumber,
                                                          FActualPos]);
  GetNextToken;
end;
```

O analisador sintático foi implementado criando-se um método para cada não terminal da linguagem LEL (quadro 18). Sendo assim, cada vez que há necessidade de reconhecer um não terminal, faz-se a chamada do seu método correspondente. Por exemplo, para reconhecer o não terminal Valor chama-se o método Valor.

As ações semânticas são feitas dentro da implementação do método do não terminal, conforme as ações semânticas especificadas no quadro 19. Por exemplo, os atributos herdados e sintetizados são implementados como parâmetros dos métodos.

3.3.1.3 AMBIENTE DE GERAÇÃO DE CÓDIGO FONTE

O ambiente de geração de código fonte foi implementado de acordo com o diagrama de seqüências da fig. 28. No quadro 22 é apresentado o código fonte para o método Execute da classe TRuntimeEngine, responsável pela geração de código fonte conforme a representação intermediária gerada pelo compilador LEL.

Quadro 22 – MÉTODO EXECUTE

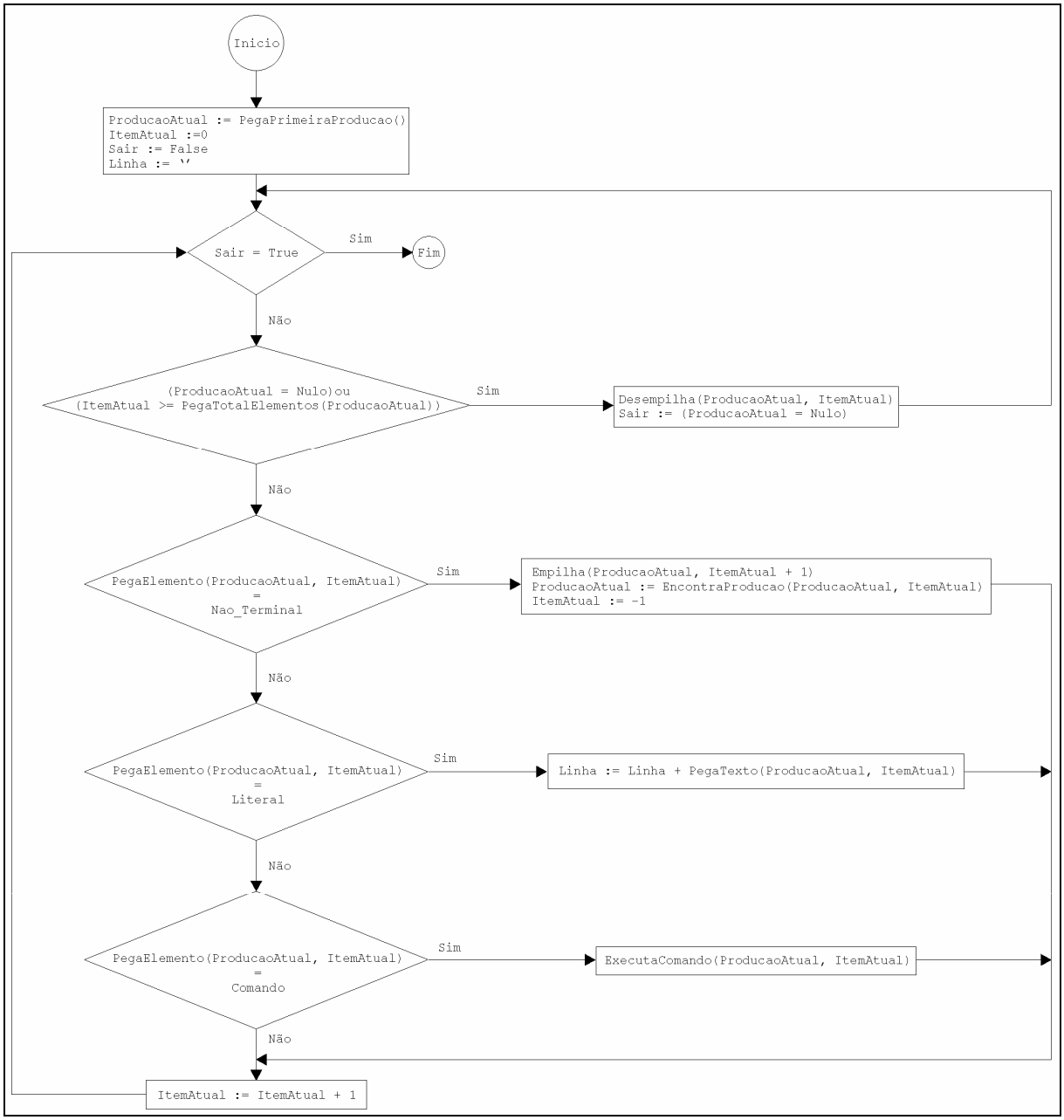
```
procedure TRuntimeEngine.Execute(const OutFileName: String);
var ActualProduction: TProduction;
    ActualItem: Integer;
    NotTerminal: TNotTerminal;
    Literal: TLiteral;
    Command: TCommand;
    Sair: Boolean;
    Line: String;
    actClass, actAttribute, actMethod, actParameter: Integer;
    Variable: TVariable;

begin
  FCurClass:=Nil;
  FCurAttribute:=Nil;
  FCurMethod:=Nil;
  FCurParameter:=Nil;

  actClass:=0;
  actAttribute:=0;
  actMethod:=0;
  actParameter:=0;

  AssignFile(FOutFile, OutFileName);
  Rewrite(FOutFile);
  try
    FCallStack.Clear;
    if   FSpecificationList.Count > 0 then
         begin
           ActualProduction:=FSpecificationList.Items[0];
           ActualItem:=0;
           Sair:=False;
         end
    else
         Sair:=True;
    Line:='';
    while not Sair do
          begin
            if (ActualProduction=nil)or(ActualItem>=ActualProduction.FElementList.Count) then
                 begin
                   Pop(ActualProduction, ActualItem);
                   Sair:=ActualProduction = Nil;
                 end
            else
                 begin
                   if   ActualProduction.Elements[ActualItem] is TNotTerminal then
                        begin
                          NotTerminal:=TNotTerminal(ActualProduction.Elements[ActualItem]);
                          Push(ActualProduction, ActualItem + 1);
```

```
                        ActualProduction:=FSpecificationList.FindProduction(NotTerminal.FID);
                        if   ActualProduction = nil then
                             raise Exception.CreateFmt(STR_PRODUCTIONNOTEXIST,
                                        [NotTerminal.FID]);
                         ActualItem:=-1;
                       end
                  else
                  if   ActualProduction.Elements[ActualItem] is TLiteral then
                       begin
                         Literal:=TLiteral(ActualProduction.Elements[ActualItem]);
                         Line:=Line + Literal.FString + ' ';
                       end
                  else
                  if   ActualProduction.Elements[ActualItem] is TCommand then
                       begin
                         Command:=TCommand(ActualProduction.Elements[ActualItem]);
                         if   Command is TAttributeCommand then
                              ProcessAttributeCommand(TAttributeCommand(Command))
                         else
                         if   Command is TElementCommand then
                              ProcessElementCommand(TElementCommand(Command))
                         else
                         if   Command is TSingleCommand then
                              ProcessSingleCommand
                         else
                         if   Command is TIfCommand then
                              ProcessIfCommand(TIfCommand(Command))
                         else
                         if   Command is TVarCommand then
                              begin
                              Variable:=FVarList.FindVariable(TVarCommand(Command).FVarName);
                                if   Variable <> nil then
                                     begin
                                       if   TVarCommand(Command).FValueType = vtNumber then
                                            Variable.AsInteger:=TexprTreeSingleIntegerItem(
                                                    TVarCommand(Command).FValueItem).Value
                                       else
                                       if   TVarCommand(Command).FValueType = vtBoolean then
                                            Variable.AsBoolean:=TExprTreeSingleBooleanItem(
                                                       TVarCommand(Command).FValueItem).Value
                                       else
                                       if   TVarCommand(Command).FValueType = vtString then
                                            Variable.AsString:=TExprTreeSingleStringItem(
                                                      TVarCommand(Command).FValueItem).Value
                                     end;
                              end;
                       end;
                  inc(ActualItem);
                end;
          end;
    if   Line <> '' then
         writeln(FOutFile, Line);
  finally
    CloseFile(FOutFile);
  end;
end;
```

O principio da implementação da geração do código fonte é relativamente simples. A fig. 29 apresenta o fluxograma da implementação. Primeiramente pega-se a primeira produção que está na lista de produções. Percorre-se os elementos do lado direito da produção, se o elemento for uma literal, o seu texto será escrito no arquivo, se for um comando, o comando é executado, ou se for um não terminal, a produção corrente é empilhada juntamente com o índice do próximo elemento. Em seguida procura-se, na lista de produções, a produção do não

terminal encontrado, e começa-se a percorrer o lado direito desta produção. Após o último elemento do lado direito da produção ser avaliado, desempilha-se a produção e índice do próximo elemento. Repete-se este processo até que a pilha estiveja vazia e o último elemento da produção corrente seja avaliado.

Figura 29 – FLUXOGRAMA DA GERAÇÃO DE CÓDIGO FONTE



Na seção seguinte são apresentados os passos de configuração e operação do protótipo e da ferramenta CASE *System Architect* para que seja feita a geração de código fonte através do protótipo.

3.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

O processo para geração de código fonte utilizando o protótipo está dividido em três partes:

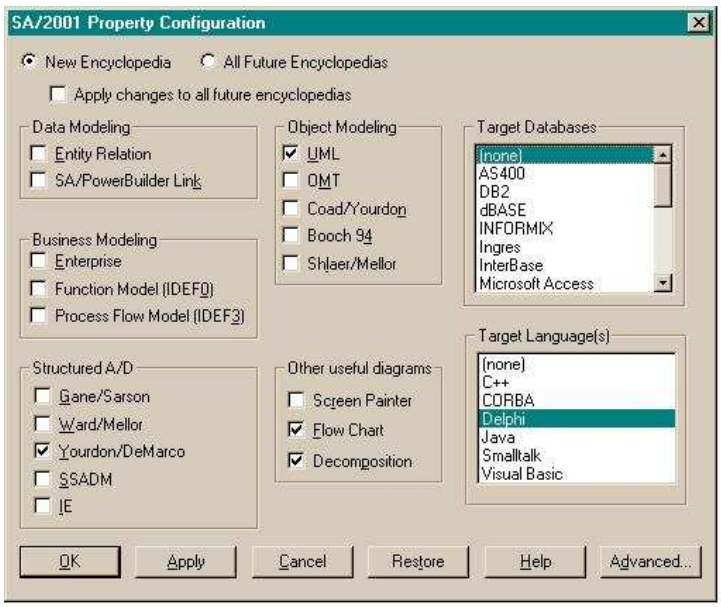
a) criação do arquivo texto contendo a especificação, escrita na metalinguagem LEL, de classes em uma linguagem;
b) criação do diagrama de classes na ferramenta CASE *System Architect*;
c) geração do código fonte através do protótipo.

Para a criação do arquivo texto contendo a especificação de classe em uma linguagem, pode-se utilizar qualquer editor de texto que permita salvar o arquivo como simples texto, sem qualquer caractere de marcação ou formatação.

Antes de iniciar a criação do diagrama de classes, no *System Architect*, deve-se configurar a ferramenta conforme o descrito na tela da fig. 30, que pode ser acessada através do menu *Tools\Customize Method Support\Encyclopedia Configuration*. Esta configuração se faz necessária para que todas as informações necessárias para a leitura do repositório sejam gravadas no campo DESCRIPTN da tabela ENTITY.DBF.

Durante o desenho do diagrama de classes no *System Architect*, pode-se notar que não existe uma padronização de tipos de dados para os atributos, nem para a definição dos parâmetros e do tipo de retorno de um método. Sendo assim, foi necessário criar uma padronização para estas situações. O quadro 23 apresenta a padronização dos tipos de dados reconhecidos. A padronização para a descrição dos parâmetros de um método é “TipoParametro NomeParametro: TipoDado”, onde TipoParametro pode assumir os valores “in” ou “out”, que identificam, respectivamente, se o parâmetro é de entrada ou saída. NomeParametro é o nome do parâmetro. TipoDado identifica o tipo de dado do parâmetro, conforme descrito no quadro 23. Quando existe mais de um parâmetro, usa-se o ponto e virgula (“;”) como separador de parâmetros.

Figura 30 – CONFIGURAÇÃO DA FERRAMENTA CASE SYSTEM ARCHITECT



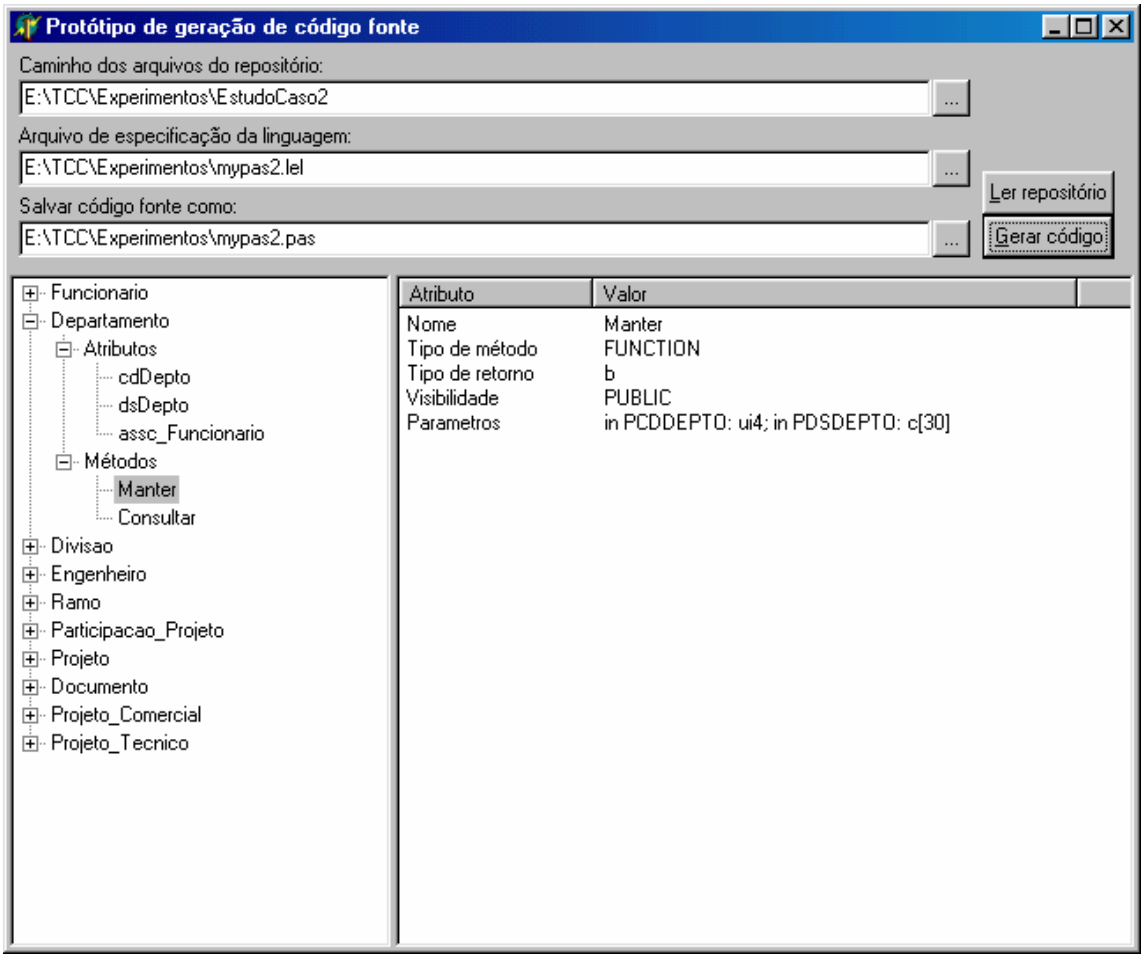
Tendo definida a especificação da representação das classes para uma linguagem específica, pode-se gerar o código fonte utilizando o protótipo desenvolvido. Maiores informações sobre a linguagem LEL são apresentadas no anexo 1.

A fig. 31 apresenta a janela principal do protótipo. Para gerar o código fonte, primeiramente deve-se especificar o caminho para o diretório onde se encontra o repositório da ferramenta CASE *System Architect*. Em seguida clicar no botão “Ler repositório” para que as classes sejam lidas do repositório e carregadas em memória.

Quadro 23 – TIPOS DE DADOS

Tipo	Significado
O	Qualquer tipo
I1	Inteiro com sinal de 1 byte
I2	Inteiro com sinal de 2 bytes
I4	Inteiro com sinal de 4 byte
UI1	Inteiro sem sinal de 1 byte
UI2	Inteiro sem sinal de 2 bytes
UI4	Inteiro sem sinal de 4 bytes
F4	Número de ponto flutuante de 4 bytes
F8	Número de ponto flutuante de 8 bytes
F10	Número de ponto flutuante de 10 bytes
C	Um único caractere
C[Tamanho]	Cadeia de caracteres
B	Booleano
P	Ponteiro
CLASS	Classe

Figura 31 – JANELA PRINCIPAL DO PROTÓTIPO



As informações lidas do repositório podem ser vistas em uma árvore na parte inferior esquerda da janela. Nesta árvore pode-se ver as classes com seus respectivos atributos e métodos. Ao selecionar algum item desta árvore suas informações são detalhadas em uma lista à direita da árvore.

Após a leitura do repositório, deve-se especificar o arquivo texto contendo a especificação das classes para uma linguagem e, também, o nome e onde se deseja salvar o código fonte gerado. Em seguida clica-se no botão “Gerar código” e se não houver nenhum erro no arquivo com a especificação, aparecerá uma janela de diálogo informando que a geração de código foi concluída com sucesso. Caso contrário, a janela de diálogo apresentará a linha e a coluna, dentro do arquivo texto da especificação, onde ocorreu o erro de compilação.

A seguir é mostrado um estudo de caso para exemplificar a operação do protótipo.

3.3.3 ESTUDO DE CASO

Para exemplificar a utilização do protótipo foi elaborado um estudo de caso de um sistema fictício para uma empresa de engenharia. O enunciado do estudo de caso é mostrado no quadro 24.

Quadro 24 – ESTUDO DE CASO

A empresa de engenharia EngCorp controla os dados pessoais de seus funcionários que podem ser administradores, engenheiros, secretária, etc. Para fins de acompanhamento de gerência são emitidos regularmente relatórios gerenciais.

Os funcionários quando são cadastrados registram um código, nome, estado civil, CIC. Cada funcionário é lotado sempre em algum departamento, que por sua vez pertence sempre a uma divisão da empresa (Finanças, Produção, etc). Quando os funcionários são engenheiros, são registrados o número do CREA e o ramo de atuação na engenharia.

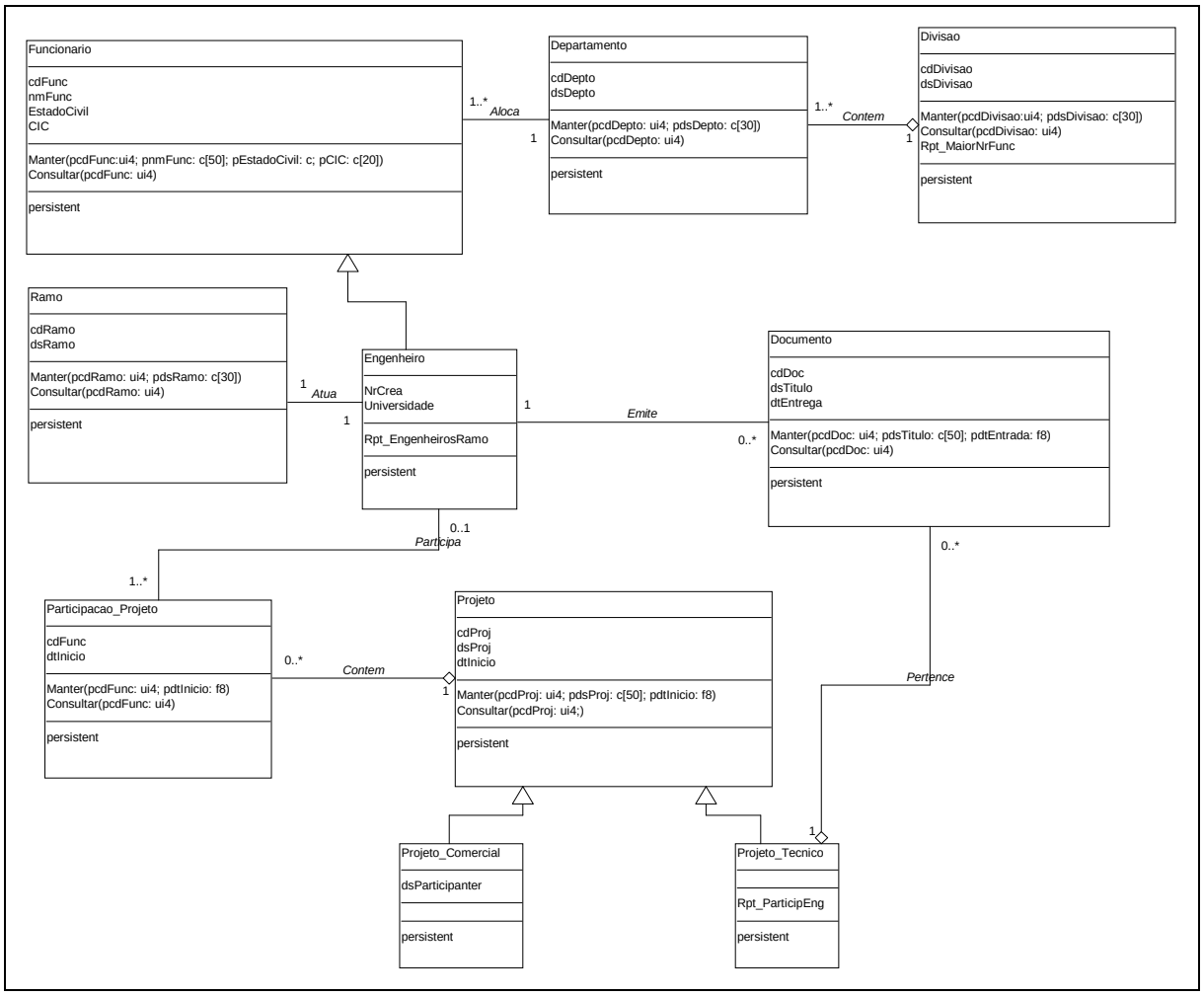
Os engenheiros podem ou não participar de vários projetos que por sua vez devem ter um ou mais engenheiros participantes. Estes projetos quando são de caráter técnico, envolvem o controle dos dados sobre todos os documentos emitidos durante o projeto como título do documento entregue, data de entrega e seu responsável. Quando trata-se de projetos comerciais há apenas a citação dos parceiros externos que irão colaborar no projeto.

Sobre a participação dos engenheiros nos projetos sempre é registrada a data de início e de sua atuação no projeto. Os engenheiros podem ser formados em universidades públicas ou privadas.

O gerente recebe trimestralmente o relatório de engenheiros classificados por ramo da engenharia, de engenheiros com maior participação em projetos técnicos, de divisão com maior número de funcionários.

Primeiramente foi criado o diagrama de classes no *System Architect*, salvando-o no repositório. A fig. 32 apresenta o diagrama de classes para este estudo de caso. A seguir são criados três arquivos de especificação. Um contendo a especificação das classes para o CDL; outro para Java e um para o *Object Pascal*, apresentados respectivamente nos quadros 25, 26 e 27.

Figura 32 – DIAGRAMA DE CLASSES



Quadro 25 – ARQUIVO DE ESPECIFICAÇÃO PARA O CDL

```
CACHE -> <readnext class> <if not <empty class> then CLASSDEF CACHE>;
CLASSDEF -> DROP_CLASS <nl> CREATE_CLASS <nl>;
DROP_CLASS -> 'Drop Class' <geto class.name> ';' <nl>;
CREATE_CLASS -> 'Create Class' <geto class.name> <nl> CLASS_DESCRIPTION;
CLASS_DESCRIPTION -> '{' <nl>
                     <if <class.isAbstract> then 'Abstract;' <nl>>
                     'Super =' <if not (<class.superclass> = '') then <geto class.superclass>
                                 else '%RegisterObject'> <bks>';' <nl>
                     <if <class.ispersistent> then 'Persistent;' <nl>>
                     ATRIBUTES_DESC <nl>
                     METHODS_DESC;
ATRIBUTES_DESC -> <readnext attribute> <if not <empty attribute> then ATRIBUTEDEF
ATRIBUTES_DESC>;
ATRIBUTEDEF -> <if <attribute.islist> then 'list'> 'attribute' <geto attribute.name>
            '{ type ='
            <if <attribute.DataType> = 1 then '%Integer' else
            <if <attribute.DataType> = 2 then '%Integer' else
            <if <attribute.DataType> = 3 then '%Integer' else
            <if <attribute.DataType> = 4 then '%Integer' else
            <if <attribute.DataType> = 5 then '%Integer' else
            <if <attribute.DataType> = 6 then '%Integer' else
            <if <attribute.DataType> = 7 then '%Float' else
            <if <attribute.DataType> = 8 then '%Float' else
            <if <attribute.DataType> = 9 then '%Float' else
            <if <attribute.DataType> = 10 then '%String' else
```

```
            <if <attribute.DataType> = 11 then '%String' else
            <if <attribute.DataType> = 12 then '%Boolean' else
            <if <attribute.DataType> = 14 then <geto Attribute.TypeClassName>
            else 'ERRO'>>>>>>>>>>>>> <bks> ';'
            <if <attribute.Visibility> = 0 then 'Private' else
            <if <attribute.Visibility> = 2 then 'Public' else 'ERRO'>> <bks> '; }' <nl>;
METHODS_DESC -> <readnext method> <if not <empty method> then METHODDEF METHODS_DESC>;
METHODDEF -> 'method' <geto method.name> PARAMETER_DESC '{' <nl>
           <if <method.MethodType> = 1 then 'ReturnType =' RETURN_TYPE <nl>>
           <if <method.Visibility> = 0 then 'Private' else
           <if <method.Visibility> = 2 then 'Public' else 'ERRO'>> <bks> ';' <nl>
           'code = {}' <nl> '}' <nl>;
PARAMETER_DESC -> <readnext parameter>
                  <if not <empty parameter> then <bks> '(' <bks> _PARAMETER
                                                 <readnext parameter> PARAMETER_LIST
                                                 <bks> ')'>;
PARAMETER_LIST -> <if not <empty parameter> then <bks> ';' _PARAMETER <readnext parameter>
PARAMETER_LIST>;
_PARAMETER ->  <geto parameter.name> <bks> ':'
               <if <parameter.DataType> = 1 then '%Integer' else
               <if <parameter.DataType> = 2 then '%Integer' else
               <if <parameter.DataType> = 3 then '%Integer' else
               <if <parameter.DataType> = 4 then '%Integer' else
               <if <parameter.DataType> = 5 then '%Integer' else
               <if <parameter.DataType> = 6 then '%Integer' else
               <if <parameter.DataType> = 7 then '%Float' else
               <if <parameter.DataType> = 8 then '%Float' else
               <if <parameter.DataType> = 9 then '%Float' else
               <if <parameter.DataType> = 10 then '%String' else
               <if <parameter.DataType> = 11 then '%String' else
               <if <parameter.DataType> = 12 then '%Boolean'
               else 'ERRO'>>>>>>>>>>>>>;
RETURN_TYPE -> <if <method.returntype> = 1 then '%Integer' else
               <if <method.returntype> = 2 then '%Integer' else
               <if <method.returntype> = 3 then '%Integer' else
               <if <method.returntype> = 4 then '%Integer' else
               <if <method.returntype> = 5 then '%Integer' else
               <if <method.returntype> = 6 then '%Integer' else
               <if <method.returntype> = 7 then '%Float' else
               <if <method.returntype> = 8 then '%Float' else
               <if <method.returntype> = 9 then '%Float' else
               <if <method.returntype> = 10 then '%String' else
               <if <method.returntype> = 11 then '%String' else
               <if <method.returntype> = 12 then '%Boolean'
               else 'ERRO'>>>>>>>>>>>>>;
```

Quadro 26 – ARQUIVO DE ESPECIFICAÇÃO PARA JAVA

```
JAVA -> 'import java.util.*;' <nl><nl> JAVA_CLASSES;
JAVA_CLASSES -> <readnext class> <if not <empty class> then CLASS_DEF JAVA_CLASSES>;
CLASS_DEF -> 'class' <geto class.name>
              <if not (<class.superclass> = '') then 'extends' <geto class.superclass>> <nl>
              '{' <nl> ATRIBUTES_DESC <nl> METHODS_DESC '}' <nl><nl>;
ATRIBUTES_DESC -> <readnext attribute> <if not <empty attribute> then ATRIBUTE_DEF
ATRIBUTES_DESC>;
ATRIBUTE_DEF -> <if <attribute.Visibility> = 0 then 'private' else
            <if <attribute.Visibility> = 1 then 'protected' else
            <if <attribute.Visibility> = 2 then 'public' else 'ERRO'>>>
            <if <attribute.IsList> then 'ArrayList' else
            <if <attribute.DataType> = 1 then 'byte' else
            <if <attribute.DataType> = 2 then 'short' else
            <if <attribute.DataType> = 3 then 'int' else
            <if <attribute.DataType> = 4 then 'byte' else
            <if <attribute.DataType> = 5 then 'short' else
            <if <attribute.DataType> = 6 then 'int' else
            <if <attribute.DataType> = 7 then 'float' else
            <if <attribute.DataType> = 8 then 'double' else
            <if <attribute.DataType> = 9 then 'double' else
            <if <attribute.DataType> = 10 then 'char' else
            <if <attribute.DataType> = 11 then 'String' else
            <if <attribute.DataType> = 12 then 'boolean' else
            <if <attribute.DataType> = 14 then <geto Attribute.TypeClassName>
```

```
            else 'ERRO'>>>>>>>>>>>>>>
            'f' <bks> <geto attribute.name> <bks> ';' <nl>;
METHODS_DESC -> <readnext method> <if not <empty method> then METHOD_DEF METHODS_DESC>;
METHOD_DEF -> <if <method.Visibility> = 0 then 'private' else
              <if <method.Visibility> = 1 then 'protected' else
              <if <method.Visibility> = 2 then 'public' else 'ERRO'>>>

              <if <method.MethodType> = 0 then 'void' else RETURN_TYPE>
               <geto method.name>
               <bks> '(' PARAMETER_DESC ')'
               <nl> '{' <nl> <if <method.MethodType> = 1 then RETURN_DECLARATION> <nl> '}'
<bks> ';' <nl>;
PARAMETER_DESC -> <readnext parameter> <if not <empty parameter> then _PARAMETER <readnext
parameter> PARAMETER_LIST>;
PARAMETER_LIST -> <if not <empty parameter> then <bks> ',' _PARAMETER <readnext parameter>
PARAMETER_LIST>;
_PARAMETER ->  <if <parameter.DataType> = 1 then 'byte' else
               <if <parameter.DataType> = 2 then 'short' else
               <if <parameter.DataType> = 3 then 'int' else
               <if <parameter.DataType> = 4 then 'byte' else
               <if <parameter.DataType> = 5 then 'short' else
               <if <parameter.DataType> = 6 then 'int' else
               <if <parameter.DataType> = 7 then 'float' else
               <if <parameter.DataType> = 8 then 'double' else
               <if <parameter.DataType> = 9 then 'double' else
               <if <parameter.DataType> = 10 then 'char' else
               <if <parameter.DataType> = 11 then 'String' else
               <if <parameter.DataType> = 12 then 'boolean'
               else 'ERRO'>>>>>>>>>>>>>
               <geto parameter.name>;
RETURN_TYPE -> <if <method.returntype> = 1 then 'byte' else
               <if <method.returntype> = 2 then 'short' else
               <if <method.returntype> = 3 then 'int' else
               <if <method.returntype> = 4 then 'byte' else
               <if <method.returntype> = 5 then 'short' else
               <if <method.returntype> = 6 then 'int' else
               <if <method.returntype> = 7 then 'float' else
               <if <method.returntype> = 8 then 'double' else
               <if <method.returntype> = 9 then 'double' else
               <if <method.returntype> = 10 then 'char' else
               <if <method.returntype> = 11 then 'String' else
               <if <method.returntype> = 12 then 'boolean'
               else 'ERRO'>>>>>>>>>>>>>;
RETURN_DECLARATION -> <if <method.returntype> = 1 then 'return 0;' else
               <if <method.returntype> = 2 then 'return 0;' else
               <if <method.returntype> = 3 then 'return 0;' else
               <if <method.returntype> = 4 then 'return 0;' else
               <if <method.returntype> = 5 then 'return 0;' else
               <if <method.returntype> = 6 then 'return 0;' else
               <if <method.returntype> = 7 then 'return 0;' else
               <if <method.returntype> = 8 then 'return 0;' else
               <if <method.returntype> = 9 then 'return 0;' else
               <if <method.returntype> = 10 then 'return '''';' else
               <if <method.returntype> = 11 then 'return null' else
               <if <method.returntype> = 12 then 'return false;'
               else 'ERRO'>>>>>>>>>>>>>;
```

Quadro 27 – ARQUIVO DE ESPECIFICAÇÃO PARA OBJECT PASCAL

```
DELPHI -> 'Unit Unit1;' <nl><nl>
          'interface' <nl><nl>
          'uses classes;' <nl><nl>
          'type' <nl>
          CLASS_FORWARD_DECLARATIONS <nl>
          CLASS_INTERFACE <nl>
          'implementation' <nl><nl>
          <reset class>
          CLASS_IMPLEMENTATION <nl>
          'end.';
CLASS_FORWARD_DECLARATIONS -> <readnext class> <if not <empty class> then
CLASS_FORWARD_DECLARATION CLASS_FORWARD_DECLARATIONS>;
CLASS_FORWARD_DECLARATION -> ' ' <geto class.name> '= class' <bks> ';' <nl>;
CLASS_INTERFACE -> <readnext class> <if not <empty class> then CLASS_DEF CLASS_INTERFACE>;
CLASS_DEF -> ' ' <geto class.name> '= class(' <bks>
             <if not (<class.superclass> = '') then <geto class.superclass> else 'TObject'>
<bks> ')' <nl>
             PRIVATE_SESSION
             PROTECTED_SESSION
             PUBLIC_SESSION
             '  end;' <nl><nl>;
PRIVATE_SESSION -> <var R1 0> '  private' <nl>
                   ATRIBUTES_DESC
                   METHODS_DESC;

PROTECTED_SESSION -> <var R1 1> '  protected' <nl>
                     ATRIBUTES_DESC
                     METHODS_DESC;
PUBLIC_SESSION -> <var R1 2> '  public' <nl>
                  ATRIBUTES_DESC
                  METHODS_DESC;
ATRIBUTES_DESC -> <readnext attribute> <if not <empty attribute> then ATRIBUTE_COND
ATRIBUTES_DESC>;
ATRIBUTE_COND -> <if <attribute.Visibility> = <R1> then ATRIBUTE>;
ATRIBUTE -> '    F' <bks> <geto attribute.name> ':'
            <if <attribute.IsList> then 'TList' else
            <if <attribute.DataType> = 1 then 'ShortInt' else
            <if <attribute.DataType> = 2 then 'Smallint' else
            <if <attribute.DataType> = 3 then 'Longint' else
            <if <attribute.DataType> = 4 then 'Byte' else
            <if <attribute.DataType> = 5 then 'Word' else
            <if <attribute.DataType> = 6 then 'Cardinal' else
            <if <attribute.DataType> = 7 then 'Real' else
            <if <attribute.DataType> = 8 then 'Double' else
            <if <attribute.DataType> = 9 then 'Extended' else
            <if <attribute.DataType> = 10 then 'Char' else
            <if <attribute.DataType> = 11 then 'String' else
            <if <attribute.DataType> = 12 then 'Boolean' else
            <if <attribute.DataType> = 13 then 'Pointer' else
            <if <attribute.DataType> = 14 then <geto Attribute.TypeClassName>
            else 'ERRO'>>>>>>>>>>>>>>>> <bks> ';' <nl>;
METHODS_DESC -> <readnext method> <if not <empty method> then METHOD_COND METHODS_DESC>;
METHOD_COND -> <if <method.Visibility> = <R1> then METHOD_HEAD>;
METHOD_HEAD -> <if <method.MethodType> = 0 then '    procedure' else '    function'>
               <geto method.name> PARAMETER_DESC
               <if <method.MethodType> = 1 then RETURN_TYPE> <bks> ';' <nl>;
PARAMETER_DESC -> <readnext parameter>
                  <if not <empty parameter> then <bks> '(' <bks> _PARAMETER
                                                 <readnext parameter> PARAMETER_LIST
                                                 <bks> ')'>;
PARAMETER_LIST -> <if not <empty parameter> then <bks> ';' _PARAMETER <readnext parameter>
PARAMETER_LIST>;
_PARAMETER ->  <geto parameter.name> <bks> ':'
               <if <parameter.DataType> = 1 then 'ShortInt' else
               <if <parameter.DataType> = 2 then 'Smallint' else
               <if <parameter.DataType> = 3 then 'Longint' else
               <if <parameter.DataType> = 4 then 'Byte' else
               <if <parameter.DataType> = 5 then 'Word' else
               <if <parameter.DataType> = 6 then 'Cardinal' else
```

```
               <if <parameter.DataType> = 7 then 'Real' else
               <if <parameter.DataType> = 8 then 'Double' else
               <if <parameter.DataType> = 9 then 'Extended' else
               <if <parameter.DataType> = 10 then 'Char' else
               <if <parameter.DataType> = 11 then 'String' else
               <if <parameter.DataType> = 12 then 'Boolean' else
               <if <parameter.DataType> = 13 then 'Pointer'
               else 'ERRO'>>>>>>>>>>>>>;
RETURN_TYPE -> ':'
               <if <method.returntype> = 1 then 'ShortInt' else
               <if <method.returntype> = 2 then 'Smallint' else
               <if <method.returntype> = 3 then 'Longint' else
               <if <method.returntype> = 4 then 'Byte' else
               <if <method.returntype> = 5 then 'Word' else
               <if <method.returntype> = 6 then 'Cardinal' else
               <if <method.returntype> = 7 then 'Real' else
               <if <method.returntype> = 8 then 'Double' else
               <if <method.returntype> = 9 then 'Extended' else
               <if <method.returntype> = 10 then 'Char' else
               <if <method.returntype> = 11 then 'String' else
               <if <method.returntype> = 12 then 'Boolean' else
               <if <method.returntype> = 13 then 'Pointer'
               else 'ERRO'>>>>>>>>>>>>>;
CLASS_IMPLEMENTATION -> <readnext class> <if not <empty class> then CLASS_IMPL
CLASS_IMPLEMENTATION>;
CLASS_IMPL -> <nl> '{' <geto class.name> '}' <nl>
              METHOD_IMPL;
METHOD_IMPL -> <readnext method> <if not <empty method> then METHOD_BODY METHOD_IMPL>;
METHOD_BODY -> <nl><if <method.MethodType> = 0 then 'procedure' else 'function'>
               <geto class.name> <bks>'.'<bks> <geto method.name> PARAMETER_DESC
               <if <method.MethodType> = 1 then RETURN_TYPE> <bks> ';' <nl>

               'begin' <nl><nl> 'end;' <nl>;
```

Após ter concluido a criação dos arquivos com as especificações das classes nas linguagens CDL, Java e *Object Pascal*, e também ter o diagrama de classes salvo no repositório do *System Architect*, executa-se o protótipo.

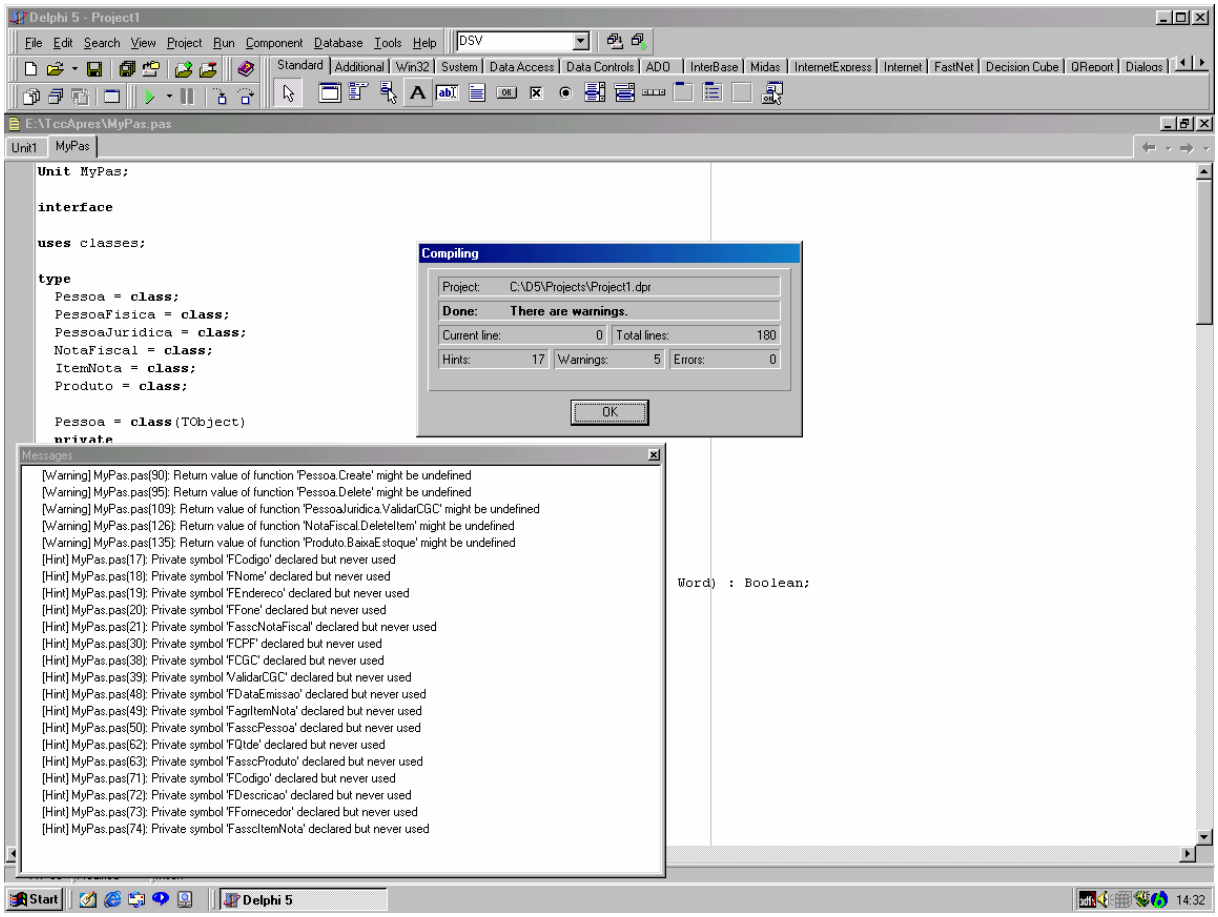
Uma vez dentro do protótipo, indica-se o caminho onde o diagrama de classes foi salvo e clica-se no botão “Ler Repositório” para carregar o diagrama de classes na memória. Em seguida indica-se o arquivo de especificação do CDL e, também, o nome do arquivo que se deseja salvar. Em seguida clica-se no botão “Gerar Código” e o arquivo contendo o “esqueleto” das classes em CDL é gerado. Repete-se este processo para os arquivos contendo a especificação em Java e *Object Pascal*.

Nos anexos 2, 3 e 4 pode-se ver respectivmente o código fonte em CDL, Java e *Object Pascal* gerado a partir do repositório salvo para este estudo de caso e dos arquivos de especificação apresentados no quadros 25, 26 e 27.

O sucesso da compilação do código gerado depende exclusivamente de como a linguagem foi especificada. Se a especificação escrita na metalinguagem LEL não estiver correta, o código fonte gerado também não estará. Caso exista erros na especificação em LEL,

estes são apresentados durante a compilação deste código fonte. A fig. 33 apresenta o código *Object Pascal* gerado pelo protótipo, compilado com sucesso no ambiente Borland Delphi 5. Como o código fonte gerado é apenas um esqueleto das classes, alguns avisos foram gerados indicando, por exemplo, que nenhum valor de retorno de uma função foi especificado, cabendo ao programador completar a implementação dos métodos das classes.

Figura 33 – CÓDIGO FONTE GERADO COMPILADO COM SUCESSO



3.4 RESULTADOS E DISCUSSÃO

Nesta seção são apresentadas algumas características entre a metalinguagem LEL e a XSL, e ainda o protótipo é confrontado com o protótipo descrito em Kramel (2000).

No quadro 28 são apresentadas características entre a meta linguagem LEL e o XSL.

Quadro 28 – CARACTERÍSTICAS DA METALIGUAGEM LEL E DO XSL

	LEL	**XSL**
Objetivo	Especificação das classes em uma linguagem de programação	Formatar a apresentação dos dados de arquivo XML
Fonte de dados	Representação, em memória, de um diagrama de classes lido a partir do repositório do *System Architect*	Arquivo XML
Ambiente de transformação	Protótipo apresentado neste trabalho	*Parser* XML
Saída dos dados	Arquivo texto contendo o código fonte para as classes na linguagem especificada	Dependendo de onde se encontra o *parser* XML, pode apresentar a saída na tela ou em um arquivo externo
Estilo da linguagem	Especificam-se as classes de uma linguagem, a BNF	Especifica-se o estilo de formatação para cada elemento do arquivo XML

As informações expostas no quadro 28, sobre a metalinguagem LEL e o XSL são muito parecidas. Ambas permitem especificar como deve ser a saída para o dados que estão em uma fonte de dados, permitindo uma grande flexibilidade para a transformação de formato destes dados.

Comparando o protótipo apresentado em Kramel (2000) e o protótipo apresentado neste trabalho, verifica-se que o presente protótipo permite uma maior flexibilidade para a geração de código fonte. O protótipo apresentado em Kramel (2000) permite apenas a geração de código fonte CDL, que foi implementado de forma “fixa” dentro do protótipo, enquanto o presente protótipo permite gerar código fonte para várias linguagens. Para tanto desenvolveu-se uma forma de especificação para representação das classes em uma linguagem (metalinguagem LEL), a qual se deseja gerar o código fonte.

O protótipo apresentado neste deste trabalho elimina algumas limitações impostas no protótipo de Kramel (2000), como:

a) geração de código fonte para várias linguagens, através da especificação das classes de uma linguagem, utilizando a metalinguagem LEL;
b) permitir a leitura de vários repositórios, armazenados em locais diferentes, sem ter a necessidade de fechar o protótipo, configurar o caminho do novo repositório que se deseja ler, e entrar no protótipo novamente.

4 CONCLUSÕES

O objetivo principal deste trabalho que foi desenvolver um protótipo de software para a geração de código fonte de diversas linguagens, desde que sua especificação esteja em um arquivo texto utilizando a notação BNF estendida, a partir do repositório da ferramenta CASE *System Architect*, foi atendido.

Para possibilitar a geração de código fonte, foi desenvolvida a metalinguagem LEL, que é utilizada para a especificação das classes em uma linguagem de programação. A metalinguagem LEL foi desenvolvida com base em linguagens livres de contexto, tendo a BNF como um método de especificação de sua sintaxe. A linguagem livre de contexto se mostrou adequada para o desenvolvimento da metalinguagem LEL.

Para o desenvolvimento do compilador da metalinguagem LEL, foi utilizada a análise sintática *top-down* como técnica de implementação do analisador sintático. A análise *top-down* se mostrou eficiente, porém se a gramática a ser implementada for muito extensa, ou seja, possua um número muito grande de produções, o analisador sintático necessitará de muita memória para a execução. Isto pelo fato de cada produção ser um procedimento do analisador sintático, sendo que a pilha de chamadas de procedimentos poderá exceder sua capacidade gerando o erro de *stack-overflow*.

Quanto a ferramenta CASE *System Architect*, vale ressaltar que é uma ferramenta flexível, pois permite realizar tanto a modelagem estrutural como a modelagem orientada a objetos. Porém esta flexibilidade trouxe dificuldades para a leitura do repositório, como a falta de padronização para o detalhamento das entidades nos diagramas modelados. Sendo assim houve a necessidade da criação de uma padronização para os tipos de dados na definição de alguns elementos no diagrama de classes, para que a leitura do repositório pudesse ser feita sem maiores problemas.

Sobre as limitações impostas ao protótipo, pode-se destacar:

a) a leitura do repositório ficou limitada a ler apenas três recursos primordiais da orientação a objetos que são a herança, a associação e a agregação;
b) a padronização dos tipos de dados ficou limitada a quinze tipos de dados preestabelecidos.

Finalizando, cabe ressaltar que a principal vantagem que o protótipo traz, graças a metalinguagem LEL, é a flexibilidade para geração do código fonte. Com a metalinguagem LEL é possível especificar uma grande quantidade de linguagens de programação disponíveis no mercado, como Java, *Object Pascal*, C++, entre outras. Juntos, a ferramenta CASE *System Architect* e o protótipo desenvolvido neste trabalho, formam um conjunto de ferramentas que permitem a modelagem das classes através do diagrama de classes da UML e a geração do "esqueleto" destas classes, na forma de código fonte, para a linguagem que o usuário necessitar, bastando apenas ele escrever a especificação desta linguagem.

4.1 EXTENSÕES

Para extensões deste trabalho sugere-se:

a) fazer com que a leitura do repostório da ferramenta CASE *System Architect* inclua outros elementos do diagrama de classes como as associações n-árias (três ou mais classes), dependência;
b) incluir uma forma flexível de especificar a padronização de tipos de dados, possibilitando a inclusão de novos tipos de dados;
c) implementar o analisador sintático com outra técnica como a análise *bottom-up*, análise LR;
d) estender a metalinguagem LEL, incluindo por exemplo, comandos de repetição e a permissão que variáveis possuam escopo.

ANEXO 1 – COMANDOS DA METALINGUAGEM LEL

Neste anexo são apresentadas as características da metalinguagem LEL.

Uma especificação para classes de uma linguagem, escrita na metalinguagem LEL, é composta por um conjunto de produções. Cada produção é inicia com um nome identificador, seguido dos caracteres "->" e de uma lista de itens que pode ser:

a) comandos, que estão entre os sinais "<" e ">". No quadro 29 são apresentados os comando da linguagem LEL;
b) não-terminais, que são os nomes identificadores de outras produções. Quando o ambiente de geração de código fonte encontrar um não-terminal (do lado direito da produção) ele é imediatamente avaliado;
c) literais, que são cadeias de caracteres entre aspas simples, que são escritas no arquivo de saída.

Quadro 29 – COMANDOS DA METALINGUAGEM LEL

Comando	**Descrição**
NL	Pula para a próxima linha no arquivo de saída.
BKS	Remove um caractere do final da linha no arquivo de saída. Por *default*, após escrever um literal, é deixado um espaço em branco.
RESET <Elemento>	Reinicia um elemento, que pode ser: ▪ *class*, que representa uma classe; ▪ *attribute*, que representa um atributo; ▪ *method*, que representa um método; ▪ *parameter*, que representa uma parâmetro.
READNEXT <Elemento>	Lê o elemento imediatamente seguinte ao elemento atual. O elemento é o mesmo que no comando RESET.
GETO <Atributo>	Escreve o valor de um atributo de um elemento, no formato: ELEMENTO.ATRIBUTO. Os elemento são os mesmo que os apresentado no comando RESET. A lista de atributos para os respectivos elemento é apresentada no quadro 30.
IF <Expressão> THEN Itens [ELSE Itens]	Comando de condição. Caso a expressão seja verdadeida, executa os itens após a palavra "THEN", caso contrário, executa os itens após a palavra "ELSE".
VAR Nome_Var <Valor>	Comando que cria e/ou atribui um valor para uma variável. O valor pode ser uma *string*, um inteiro ou

	um booleano.

Quadro 30 – ATRIBUTOS PARA OS ELEMENTOS

Elemento	**Atributo**	**Descrição**	**Tipo**
Class	*Name*	Indica o nome da classe.	*String*
	SuperClass	Indica o nome da super classe.	*String*
	IsAbstract	Indica se a classe é abstrata ou não.	Booleano
	IsDerived	Indica se a classe é derivada ou não.	Booleano
	IsPersistent	Indica se a classe é persistente ou não.	Booleano
Attribute	*Name*	Indica o nome do atributo.	*String*
	DataType	Indica o tipo de dado do atributo. Os tipos de dados válidos são apresentados no quadro 23.	Inteiro, onde: ▪ I1 = 1; ▪ I2 = 2; ▪ I4 = 3; ▪ UI1 = 4; ▪ UI2 = 5; ▪ UI4 = 6; ▪ F4 = 7; ▪ F8 = 8; ▪ F10 = 9; ▪ C = 10; ▪ C[Tamanho] = 11; ▪ B = 12; ▪ P = 13; ▪ CLASS = 14.
	Length	Indica o tamanho do atributo.	Inteiro
	Visibility	Indica a visibilidade do atributo. Os valores representam a visibilidade privada, protegida, publica.	Inteiro, onde: ▪ Privada = 0; ▪ Protegida = 1; ▪ Publica = 2.
	IsList	Indica se o atributo é ou não uma lista.	Booleano
	TypeClassName	Se o tipo for uma outra classe, indica seu nome.	*String*
Method	*Name*	Indica o nome do método.	*String*
	MethodType	Indica se o método é um procedimento ou uma função.	Inteiro, onde: ▪ Procedimento = 0; ▪ Função = 1.
	ReturnType	Se o método é uma função, indica o tipo de dado do retorno. O tipo de dado é o mesmo que o tipo de dado do atributo.	Inteiro
	ReturnLength	Se o método é uma função, indica o tamanho do tipo de retorno.	Inteiro
	Visibility	Indica a visibilidade do método. A visibilidade é a mesma que a visibilidade do atributo.	Inteiro
Parameter	*Name*	Indica o nome do parâmetro.	*String*
	DataType	Indica o tipo de dados do parâmetro. O tipo de dado é o mesmo que o tipo de dado do atributo.	Inteiro
	Length	Indica o tamanho do parâmetro.	Inteiro
	ParameterType	Indica o tipo do parâmetro, se o parâmetro é de entrada ou de saída.	Inteiro, onde: ▪ Entrada = 0; ▪ Saída = 1.

ANEXO 2 – CÓDIGO FONTE CDL

No quadro 31 é apresentado o código fonte CDL (para o estudo de caso apresentado no quadro 24) gerado pelo protótipo.

Quadro 31 – CÓDIGO FONTE CDL

```
Drop Class Funcionario ;

Create Class Funcionario
{
Super = %RegisterObject;
Persistent;
attribute cdFunc { type = %Integer; Private; }
attribute nmFunc { type = %String; Private; }
attribute EstadoCivil { type = %String; Private; }
attribute CIC { type = %String; Private; }
attribute assc_Departamento { type = Departamento; Private; }

method Manter(PCDFUNC: %Integer; PNMFUNC: %String; PESTADOCIVIL: %String; PCIC: %String) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDFUNC: %Integer) {
ReturnType = %Boolean
Public;
code = {}
}

Drop Class Departamento ;

Create Class Departamento
{
Super = %RegisterObject;
Persistent;
attribute cdDepto { type = %Integer; Private; }
attribute dsDepto { type = %String; Private; }
list attribute assc_Funcionario { type = Funcionario; Private; }

method Manter(PCDDEPTO: %Integer; PDSDEPTO: %String) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDDEPTO: %Integer) {
ReturnType = %Boolean
Public;
code = {}
}

Drop Class Divisao ;

Create Class Divisao
{
Super = %RegisterObject;
Persistent;
attribute cdDivisao { type = %Integer; Private; }
attribute dsDivisao { type = %String; Private; }
list attribute agr_Departamento { type = Departamento; Private; }

method Manter(PCDDIVISAO: %Integer; PDSDIVISAO: %String) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDDIVISAO: %Integer) {
ReturnType = %Boolean
```

```
Public;
code = {}
}
method Rpt_MaiorNrFunc {
Public;
code = {}
}

Drop Class Engenheiro ;

Create Class Engenheiro
{
Super = Funcionario;
Persistent;
attribute NrCrea { type = %Integer; Private; }
attribute Universidade { type = %String; Private; }
list attribute assc_Documento { type = Documento; Private; }
attribute assc_Ramo { type = Ramo; Private; }
list attribute assc_Participacao_Projeto { type = Participacao_Projeto; Private; }

method Rpt_EngenheirosRamo {
Public;
code = {}
}

Drop Class Ramo ;

Create Class Ramo
{
Super = %RegisterObject;
Persistent;
attribute cdRamo { type = %Integer; Private; }
attribute dsRamo { type = %String; Private; }
attribute assc_Engenheiro { type = Engenheiro; Private; }

method Manter(PCDRAMO: %Integer; PDSRAMO: %String) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDRAMO: %Integer) {
ReturnType = %Boolean
Public;
code = {}
}

Drop Class Participacao_Projeto ;

Create Class Participacao_Projeto
{
Super = %RegisterObject;
Persistent;
attribute cdFunc { type = %Integer; Private; }
attribute dtInicio { type = %Float; Private; }
attribute assc_Engenheiro { type = Engenheiro; Private; }

method Manter(PCDFUNC: %Integer; PDTINICIO: %Float) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDFUNC: %Integer) {
ReturnType = %Boolean
Public;
code = {}
}

Drop Class Projeto ;

Create Class Projeto
{
Super = %RegisterObject;
```

```
Persistent;
attribute cdProj { type = %Integer; Private; }
attribute dsProj { type = %String; Private; }
attribute dtInicio { type = %Float; Private; }
list attribute agr_Participacao_Projeto { type = Participacao_Projeto; Private; }

method Manter(PCDPROJ: %Integer; PDSPROJ: %String; PDTINICIO: %Float) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDPROJ: %Integer) {
ReturnType = %Boolean
Public;
code = {}
}

Drop Class Documento ;

Create Class Documento
{
Super = %RegisterObject;
Persistent;
attribute cdDoc { type = %Integer; Private; }
attribute dsTitulo { type = %String; Private; }
attribute dtEntrega { type = %Float; Private; }
attribute assc_Engenheiro { type = Engenheiro; Private; }

method Manter(PCDDOC: %Integer; PDSTITULO: %String; PDTENTRADA: %Float) {
ReturnType = %Boolean
Public;
code = {}
}
method Consultar(PCDDOC: %Integer) {
ReturnType = %Boolean
Public;
code = {}
}

Drop Class Projeto_Comercial ;

Create Class Projeto_Comercial
{
Super = Projeto;
Persistent;
attribute dsParticipanter { type = %String; Private; }


Drop Class Projeto_Tecnico ;

Create Class Projeto_Tecnico
{
Super = Projeto;
Persistent;
list attribute agr_Documento { type = Documento; Private; }

method Rpt_ParticipEng {
Public;
code = {}
}
```

ANEXO 3 - CÓDIGO FONTE JAVA

No quadro 32 é apresentado o código fonte Java (para o estudo de caso apresentado no quadro 24) gerado pelo protótipo.

Quadro 32 – CÓDIGO FONTE JAVA

```
import java.util.*;

class Funcionario
{
private int fcdFunc;
private String fnmFunc;
private char fEstadoCivil;
private String fCIC;
private Departamento fassc_Departamento;
public boolean Manter( int PCDFUNC, String PNMFUNC, char PESTADOCIVIL, String PCIC )
{
return false;
};
public boolean Consultar( int PCDFUNC )
{
return false;
};
}
class Departamento
{
private int fcdDepto;
private String fdsDepto;
private ArrayList fassc_Funcionario;
public boolean Manter( int PCDDEPTO, String PDSDEPTO )
{
return false;
};
public boolean Consultar( int PCDDEPTO )
{
return false;
};
}
class Divisao
{
private int fcdDivisao;
private String fdsDivisao;
private ArrayList fagr_Departamento;
public boolean Manter( int PCDDIVISAO, String PDSDIVISAO )
{
return false;
};
public boolean Consultar( int PCDDIVISAO )
{
return false;
};
public void Rpt_MaiorNrFunc( )
{
};
}
class Engenheiro extends Funcionario
{
private int fNrCrea;
private String fUniversidade;
private ArrayList fassc_Documento;
private Ramo fassc_Ramo;
private ArrayList fassc_Participacao_Projeto;
public void Rpt_EngenheirosRamo( )
{
};
}
class Ramo
```

```
{
private int fcdRamo;
private String fdsRamo;
private Engenheiro fassc_Engenheiro;
public boolean Manter( int PCDRAMO, String PDSRAMO )
{
return false;
};
public boolean Consultar( int PCDRAMO )
{
return false;
};
}
class Participacao_Projeto
{
private int fcdFunc;
private double fdtInicio;
private Engenheiro fassc_Engenheiro;
public boolean Manter( int PCDFUNC, double PDTINICIO )
{
return false;
};
public boolean Consultar( int PCDFUNC )
{
return false;
};
}
class Projeto
{
private int fcdProj;
private String fdsProj;
private double fdtInicio;
private ArrayList fagr_Participacao_Projeto;
public boolean Manter( int PCDPROJ, String PDSPROJ, double PDTINICIO )
{
return false;
};
public boolean Consultar( int PCDPROJ )
{
return false;
};
}
class Documento
{
private int fcdDoc;
private String fdsTitulo;
private double fdtEntrega;
private Engenheiro fassc_Engenheiro;
public boolean Manter( int PCDDOC, String PDSTITULO, double PDTENTRADA )
{
return false;
};
public boolean Consultar( int PCDDOC )
{
return false;
};
}
class Projeto_Comercial extends Projeto
{
private String fdsParticipanter;
}
class Projeto_Tecnico extends Projeto
{
private ArrayList fagr_Documento;
public void Rpt_ParticipEng( )
{
};
}
```

ANEXO 4 - CÓDIGO FONTE DELPHI

No quadro 33 é apresentado o código fonte Java (para o estudo de caso apresentado no quadro 24) gerado pelo protótipo.

Quadro 33 – CÓDIGO FONTE DELPHI

```
Unit Unit1;

interface

uses classes;

type
  Funcionario = class;
  Departamento = class;
  Divisao = class;
  Engenheiro = class;
  Ramo = class;
  Participacao_Projeto = class;
  Projeto = class;
  Documento = class;
  Projeto_Comercial = class;
  Projeto_Tecnico = class;

  Funcionario = class(TObject)
  private
    FcdFunc : Cardinal;
    FnmFunc : String;
    FEstadoCivil : Char;
    FCIC : String;
    Fassc_Departamento : Departamento;
  protected
  public
    function Manter(PCDFUNC: Cardinal; PNMFUNC: String; PESTADOCIVIL: Char; PCIC: String) :
Boolean;
    function Consultar(PCDFUNC: Cardinal) : Boolean;
  end;

  Departamento = class(TObject)
  private
    FcdDepto : Cardinal;
    FdsDepto : String;
    Fassc_Funcionario : TList;
  protected
  public
    function Manter(PCDDEPTO: Cardinal; PDSDEPTO: String) : Boolean;
    function Consultar(PCDDEPTO: Cardinal) : Boolean;
  end;

  Divisao = class(TObject)
  private
    FcdDivisao : Cardinal;
    FdsDivisao : String;
    Fagr_Departamento : TList;
  protected
  public
    function Manter(PCDDIVISAO: Cardinal; PDSDIVISAO: String) : Boolean;
    function Consultar(PCDDIVISAO: Cardinal) : Boolean;
    procedure Rpt_MaiorNrFunc;
  end;

  Engenheiro = class(Funcionario)
  private
    FNrCrea : Cardinal;
    FUniversidade : String;
    Fassc_Documento : TList;
    Fassc_Ramo : Ramo;
```

```
    Fassc_Participacao_Projeto : TList;
  protected
  public
    procedure Rpt_EngenheirosRamo;
  end;

  Ramo = class(TObject)
  private
    FcdRamo : Cardinal;
    FdsRamo : String;
    Fassc_Engenheiro : Engenheiro;
  protected
  public
    function Manter(PCDRAMO: Cardinal; PDSRAMO: String) : Boolean;
    function Consultar(PCDRAMO: Cardinal) : Boolean;
  end;

  Participacao_Projeto = class(TObject)
  private
    FcdFunc : Cardinal;
    FdtInicio : Double;
    Fassc_Engenheiro : Engenheiro;
  protected
  public
    function Manter(PCDFUNC: Cardinal; PDTINICIO: Double) : Boolean;
    function Consultar(PCDFUNC: Cardinal) : Boolean;
  end;

  Projeto = class(TObject)
  private
    FcdProj : Cardinal;
    FdsProj : String;
    FdtInicio : Double;
    Fagr_Participacao_Projeto : TList;
  protected
  public
    function Manter(PCDPROJ: Cardinal; PDSPROJ: String; PDTINICIO: Double) : Boolean;
    function Consultar(PCDPROJ: Cardinal) : Boolean;
  end;

  Documento = class(TObject)
  private
    FcdDoc : Cardinal;
    FdsTitulo : String;
    FdtEntrega : Double;
    Fassc_Engenheiro : Engenheiro;
  protected
  public
    function Manter(PCDDOC: Cardinal; PDSTITULO: String; PDTENTRADA: Double) : Boolean;
    function Consultar(PCDDOC: Cardinal) : Boolean;
  end;

  Projeto_Comercial = class(Projeto)
  private
    FdsParticipanter : String;
  protected
  public
  end;

  Projeto_Tecnico = class(Projeto)
  private
    Fagr_Documento : TList;
  protected
  public
    procedure Rpt_ParticipEng;
  end;


implementation


{ Funcionario }
```

```
function Funcionario.Manter(PCDFUNC: Cardinal; PNMFUNC: String; PESTADOCIVIL: Char; PCIC:
String) : Boolean;
begin

end;

function Funcionario.Consultar(PCDFUNC: Cardinal) : Boolean;
begin

end;

{ Departamento }

function Departamento.Manter(PCDDEPTO: Cardinal; PDSDEPTO: String) : Boolean;
begin

end;

function Departamento.Consultar(PCDDEPTO: Cardinal) : Boolean;
begin

end;

{ Divisao }

function Divisao.Manter(PCDDIVISAO: Cardinal; PDSDIVISAO: String) : Boolean;
begin

end;

function Divisao.Consultar(PCDDIVISAO: Cardinal) : Boolean;
begin

end;

procedure Divisao.Rpt_MaiorNrFunc;
begin

end;

{ Engenheiro }

procedure Engenheiro.Rpt_EngenheirosRamo;
begin

end;

{ Ramo }

function Ramo.Manter(PCDRAMO: Cardinal; PDSRAMO: String) : Boolean;
begin

end;

function Ramo.Consultar(PCDRAMO: Cardinal) : Boolean;
begin

end;

{ Participacao_Projeto }

function Participacao_Projeto.Manter(PCDFUNC: Cardinal; PDTINICIO: Double) : Boolean;
begin

end;

function Participacao_Projeto.Consultar(PCDFUNC: Cardinal) : Boolean;
begin

end;
```

```
{ Projeto }

function Projeto.Manter(PCDPROJ: Cardinal; PDSPROJ: String; PDTINICIO: Double) : Boolean;
begin

end;

function Projeto.Consultar(PCDPROJ: Cardinal) : Boolean;
begin

end;

{ Documento }

function Documento.Manter(PCDDOC: Cardinal; PDSTITULO: String; PDTENTRADA: Double) : Boolean;
begin

end;

function Documento.Consultar(PCDDOC: Cardinal) : Boolean;
begin

end;

{ Projeto_Comercial }

{ Projeto_Tecnico }

procedure Projeto_Tecnico.Rpt_ParticipEng;
begin

end;

end.
```

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfredo V.; SETHI, Ravi; ULMAN, Jeffrey. **Compiladores:** princípios, técnicas e ferramentas**.** Massachusetts: Addison Wesley Publisching Co., 1995.

BORLAND INTERNATIONAL INC. **Object pascal language guide**. Scotts Valley: Borland, 1995.

COLANZI, Thelma Elita. **Análise e projeto orientados a objeto**. Paraná: UNIPAR, 1999.

DAY, Don R. **Hands-on XSL,** XSL for fun and diversion. Estados Unidos, mar. 2000. Disponível em <http://www-106.ibm.com/developerworks/xml>. Acesso em: 23 abr. 2000.

FERREIRA, Aurélio Buarque de H. **Minidicionário da língua portuguesa.** Rio de Janeiro: Nova Fronteira, 1985.

FLANAGAN, David. **Java in a nutshell.** Estados Unidos: O´Reilly, 1997.

FLYNN, Peter. **Frequently asked questions about the extensible markup language.** Estados Unidos, jul. 2000. Disponível em <http://www.ucc.ie/xml>. Acesso em: 22 abr. 2001.

HOWE, Denis. **The free online dictionary of computing.** Inglaterra, nov. 1997. Disponível em <http://foldoc.doc.ic.ac.uk/>. Acesso em: 26 abr. 2001.

INTERSYSTEMS CORPORATION. **Caché development guide.** Estados Unidos: Intersystems Corporation, 2000.

KOWALTOWSKI, Tomasz. **Implementação de linguagens de programação.** Rio de Janeiro: Guanabara Dois, 1983.

KRAMEL, Danilo. **Protótipo de software para a geração de código CDL através do repositório da ferramenta CASE system architect.** 2000. 92 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LEWIS, Harry R.; PAPADIMITRIOU, Christos H. **Elementos de teoria da computação.** Tradução Edson Furmankiewicz. Porto Alegre: Bookman, 2000.

LISCHNER, Ray. **Delphi:** o guia essencial. Tradução Daniel Vieira. Rio de Janeiro: Campus, 2000.

MARTIN, James; ODELL, James. **Análise e projetos orientados a objetos.** Tradução José Carlos Barbosa dos Santos. São Paulo: Makron Books, 1995.

MENEZES, Paulo Blauth. **Linguagens formais e autômatos.** 2ª Ed. Porto Alegre: Sagra Luzzatto, 1998.

NETO, João José. **Introdução à compilação.** Rio de Janeiro: Livros Técnicos e Científicos Editora, 1987.

NIEMEYER, Patrick; PECK, Joshua. **Exploring java.** Estados Unidos: O´Reilly, 1997.

PACHECO, Xavier; TEIXEIRA, Steve. **Delphi 5 developers guide.** Indianápolis: Sams Publishing, 2000.

POPKIN SOFTWARE SYSTEMS INC. **System architect 2001 user guide**. New York: 2001.

RATIONAL SOFTWARE. **UML notation guide.** Estados Unidos, set. 1997. Disponível em <http://www.rational.com/uml>. Acesso em: 13 mai. 2001.

RUMBAUGH, James et al. **Modelagem e projetos baseados em objetos**. Rio de Janeiro: Campus, 1994.

SETZER, Valdemar W.; MELO, Inês S. Homem de. **A construção de um compilador.** Rio de Janeiro: Campus, 1986.

TONSING, Sérgio Luiz. **Projeto orientado a objetos e U.M.L.** Araçatuba, dez. [2000?]. Disponível em < http://www.geocities.com/Athens/Olympus/1307/>. Acesso em: 15/05/2001.

WATT, David A. **Programming language processors:** compilers and interpreters. Inglaterra: Pretince Hall International, 1993.

W3C WORLD WIDE WEB CONSORTIUM. **Extensible markup language (XML) 1.0.** Estados Unidos, out. 2000a. Disponível em <http://www.w3.org/XML>. Acesso em: 20 fev. 2001.

W3C WORLD WIDE WEB CONSORTIUM. **Extensible stylesheet language (XSL) 1.0.** Estados Unidos, nov. 2000b. Disponível em <http://www.w3.org/TR/xsl>. Acesso em: 20 fev. 2001.